Universidad Carlos III de Madrid

TESIS DOCTORAL

Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning

Autor

Álvaro Torralba Arias de Reyna

Directores

Dr. D. Daniel Borrajo Millán y
Dr. D. Carlos Linares López

Departamento de Informática. Escuela Politécnica Superior

Leganés, 2 de Junio de 2015

To Rosa and my parents

Acknowledgements

Thanks to everyone that supported me during all these years. Doing this thesis has been a great experience thanks to their support and all the good moments we shared.

In the first place I want to thank Rosa because her support and help were essential to finish the thesis. It's just impossible to express here the motivation she gave me or how much she strived to help me to succeed. Her friendly and not so friendly (but always good) advice were always of utmost importance to me, I probably should have follow them more often!

Of course, I want to thank my advisors, Carlos Linares López and Daniel Borrajo, because they gave me the opportunity of doing this thesis in the first place. They introduced me into the planning world and guided me with their personal and academic advice, which I have always sincerely appreciated. They also helped me without hesitation whenever I needed. Sharing the office with Carlos for more than two years was a whole experience, learning a lot and bothering him with my questions. I just hope it was as great for him as it was to me.

Also, to the other people that I have collaborated with during this time. My visit to Bremen, where I met Stefan Edelkamp and Peter Kissmann, was a key point for the development of my research. They did not only introduce me to one the main topic of my thesis, symbolic search, but also helped me to start researching in these topic. Also, Vidal Alcázar because it was really fun to work with him and always talk about planning. I really wish to have the opportunity to work with them again in the future.

I also want to thank Peter Kissmann, Anders Johnnson and Malte Helmert for their valuable comments that helped me out to improve my work. I'm truly grateful of all the effort they did into their reviews.

Thanks to the people that worked with me in Universidad Carlos III de Madrid. Especially with those in the "labo": Isa, Jesús, Sergio, Vidal, Moisés, Nerea, JC, Pulido, Eze, Ruben, Emilio, and our invited guest, Nacho. Because I had a great time with them and everything is easier when you work in a friendly environment. The disinterested help of Isa made me a lot easier to finish the thesis from the distance. Thanks also to Javier and Eloy who worked with me in the "TIMI" project. Also, to everyone else in the Planning and Learning Group, with a special remark for Nely who always helped me with all the administrative stuff. Even if our paths temporarilly split, I'll always consider myself part of the PLG group. I'll miss those seminars about planning in which we always end up discussing about silly details.

Por supuesto, quiero agradecer el apoyo de mi familia. A mi madre que siempre se desvive por ayudarme en todo lo que puede. Desde luego, sin su apoyo incondicional que ya llevo recibiendo desde hace casi 30 años, hoy no podría estar escribiendo estos agradecimientos. A mi padre, al que le agradezco todos esos discursos y consejos que me dió durante estos años. A mi abuela, Esperanza, que siempre me da todo su cariño. A Elian, porque me apoyó en el momento que decidí lanzarme a esta aventura. A mis tios, Encarna y Pedro, y mi primo Miguel. A Lucia por ser parte de mi familia.

To my lifelong friends: David, Sonia, Borja, Samer, Altieri, Acebron, Luis, Irene y Natalia because, although everything that they know about my research is that “a planner cannot even cut a pizza”, we have had great times together, always laughing with funny anecdotes no matter how far away we are. To Tomás, for all those long conversations about important and trivial stuff.

To Jörg Hoffmann for giving me the chance to continue in academia even before I finished my PhD and giving me a warm welcome to Saarbrücken. Also to all my ping-pong friends

Finally, I’d like to thank all the people that made me possible to get here, my teachers from Palomeras Bajas, Paco, Lourdes, etc. My teachers and mates in the university, Pablo, Rober, Lidia, etc. I’m sorry I cannot mention you all.

Thank you all so much!

Resumen

La Planificación Automática puede ser definida como el problema de encontrar una secuencia de acciones (un plan) para conseguir una meta, desde un punto inicial, asumiendo que las acciones tienen efectos deterministas. La Planificación Automática es independiente de dominio porque los planificadores toman como información inicial una descripción del problema y deben resolverlo sin ninguna información adicional. Esta tesis trata en particular de planificación automática óptima, en la cual las acciones tienen un coste asociado. Los planificadores óptimos deben encontrar un plan y probar que no existe ningún otro plan de menor coste.

La mayoría de los planificadores óptimos están basados en la búsqueda de estados explícita. Sin lugar a dudas, esta aproximación ha sido la dominante en planificación automática óptima durante los últimos años. No obstante, la búsqueda simbólica se presenta como una aternativa interesante. En la búsqueda simbólica, se representan conjuntos de estados como diagramas de decisión binarios (binary decision diagrams, BDDs). La representación en forma de BDDs, además de reducir la memoria necesaria para almacenar los estados de la búsqueda, también permite al planificador realizar operaciones sobre conjuntos de estados y reducir así el tiempo de búsqueda.

En esta tesis, proponemos dos mejoras ortogonales para la planificación basada en búsqueda simbólica. En primer lugar, estudiamos diferentes métodos para mejorar la computación de la "imagen", operación que calcula el conjunto de estados sucesores a partir de un conjunto de estados. Esta operacion es normalmente el cuello de botella de los planificadores simbólicos. Posteriormente, analizamos cómo explotar las invariantes de estado para mejorar el rendimiento de la búsqueda simbólica. Estas propuestas suponen una mejora significativa en el desempeño de los algoritmos simbólicos en la mayoría de los dominios analizados. Además, la versión mejorada del algoritmo simbólico de búsqueda bidireccional ciega es una de las aproximaciones más exitosas para la planificación óptima, a pesar de no estar guiada por ninguna heurística.

Los planificadores basados en búsqueda de estados explícita utilizan normalmente una heurística admisible. Estas estiman el coste desde un estado a la meta de forma optimista. Las heurísticas deben derivarse automáticamente de la descripción del problema y pueden ser clasificadas en diferentes familias de acuerdo con las ideas fundamentales en las que se basan. Hemos analizado dos tipos de heurísticas de abstracción con el objetivo de extrapolar las mejoras que se han realizado en la búsqueda explícita durante los últimos años a la búsqueda simbólica. Las heurísticas analizadas son: las bases de datos de patrones (pattern databases, PDBs) y una generalización de estas, merge-and-shrink (M&S). Mientras que las PDBs se han utilizado con anterioridad en búsqueda simbólica, hemos estudiado el uso de M&S, que es más general. En esta tesis se muestra que determinados tipos de heurísticas de M&S (aquellas que son generadas mediante una estrategia de "merge" lineal) pueden ser representadas como BDDs, con un coste computacional polinomial en el tamaño de la abstracción y la descripción del problema; y por lo tanto, pueden ser utilizadas de forma eficiente en la búsqueda simbólica. También proponemos una nueva heurística, "symbolic perimeter merge-and-shrink" (SPM&S) que combina la fuerza de la búsqueda hacia atrás simbólica con la flexibilidad de M&S. Los resultados experimentales muestran que SPM&S es capaz de superar, no solo las dos técnicas que combina, sino también otras heurísticas del estado del arte.

Finalmente, hemos integrado las abstracciones simbólicas de perímetro, SPM&S, en la búsqueda simbólica bidireccional. La heurística utilizada en la búsqueda bidireccional es calculada mediante otra búsqueda simbólica bidireccional en el espacio de estados abstracto. Se muestra cómo, aunque en general la combinación de la búsqueda bidirecional simbólica y las abstracciones tienen un comportamiento similar al que muestra la búsqueda simbólica ciega, este puede resolver más problemas en dominios particulares.

En resumen, esta tesis estudia diferentes propuestas para planificación óptima basada en búsqueda simbólica. Hemos implementado diferentes planificadores simbólicos basados en la búsqueda bidireccional y las abstracciones de perímetro. Los resultados experimentales muestran cómo los planificadores presentados como resultado de este trabajo son altamente competitivos y frecuentemente superan al resto de planificadores del estado del arte.

Abstract

Domain-independent planning is the problem of finding a sequence of actions for achieving a goal from an initial state assuming that actions have deterministic effects. It is domain-independent because planners take as input the description of a problem and must solve it without any additional information. In this thesis, we deal with cost-optimal planning problems, in which actions have an associated cost and the planner must find a plan and prove that no other plan of lower cost exists.

Most cost-optimal planners are based on explicit-state search. While this has undoubtedly been the dominant approach to cost-optimal planning in the last years, symbolic search is an interesting alternative. In symbolic search, sets of states are succinctly represented as binary decision diagrams, BDDs. The BDD representation does not only reduce the memory needed to store sets of states, but also allows the planner to efficiently manipulate sets of states reducing the search time.

We propose two orthogonal enhancements for symbolic search planning. On the one hand, we study different methods for image computation, which usually is the bottleneck of symbolic search planners. On the other hand, we analyze how to exploit state invariants to prune symbolic search. Our techniques significantly improve the performance of symbolic search algorithms in most benchmark domains. Moreover, the enhanced version of symbolic bidirectional search is one of the strongest approaches to domain-independent planning even though it does not use any heuristic.

Explicit-state search planners are commonly guided with admissible heuristics, which optimistically estimate the cost from any state to the goal. Heuristics are automatically derived from the problem description and can be classified into different families according to their underlying ideas. In order to bring the improvements on heuristics that have been made in explicit-state search to symbolic search, we analyze two types of abstraction heuristics: pattern databases (PDBs) and a generalization of them, merge-and-shrink (M&S). While PDBs had already been used in symbolic search, we analyze the use of the more general M&S heuristics. We show that certain types of M&S heuristics (those generated with a linear merging strategy) can be represented as BDDs with at most a polynomial overhead and, thus, efficiently used in symbolic search. We also propose a new heuristic, symbolic perimeter merge-and-shrink (SPM&S) that combines the strength of symbolic regression search with the flexibility of M&S heuristics. Our experiments show that SPM&S is able to beat, not only the two techniques it combines, but also other state-of-the-art heuristics.

Finally, we integrate our symbolic perimeter abstraction heuristics in symbolic bidirectional search. The heuristic used by the bidirectional search is computed by means of another symbolic bidirectional search in an abstract state space. We show how, even though the combination of symbolic bidirectional search and abstraction heuristics has an overall performance similar to the simpler symbolic bidirectional blind search, it can sometimes solve more problems in particular domains.

In summary, this thesis studies different enhancements on symbolic search. We implement different symbolic search planners based on bidirectional search and perimeter abstraction heuristics. Experimental results show that the resulting planners are highly competitive and often outperform other state-of-the-art planners.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this chapter we set the objectives of the thesis. First, we formally describe the classical planning problem we deal with and the SAS$^+$ representation we will use. We then briefly review two state-of-the-art techniques for optimal planning: heuristic search and symbolic search.

Then, we define the concrete goals of this thesis. We also describe the settings of all the experiments that we will conduct throughout this research. Finally, we present a brief outline of the rest of the document.

1.1 Classical Planning

The planning discipline follows the current trend of model-based Artificial Intelligence (AI) approaches that develop solvers for well-defined mathematical models (Geffner, 2014). Solvers are programs that take a description of a particular model instance and automatically compute its solution. In contrast to other old approaches to AI, which addressed ill-defined problems, solvers are general because they must deal with any problem that fits the model. Even if the models are not tractable in the general case, the solvers must exploit the structure of the problems in order to solve them as efficiently as possible.

Automated planning deals with problems in which an agent must achieve a set of goals by executing a sequence of actions (Ghallab et al., 2004; Russell and Norvig, 2010). Domain-independent planning is not only a theoretical exercise, but also has a good amount of practical applications in which it was used to solve diverse problems such as web-service composition (Hoffmann et al., 2009; Ordóñez et al., 2014), natural-language generation (Koller and Hoffmann, 2010), greenhouse logistics (Helmert and Lasinger, 2010), controlling modular printers (Ruml et al., 2011), intermodal transportation problems (García et al., 2013), among others.

There is a variety of planning models, depending on their assumptions about the agent and the environment. The most basic model is classical planning, characterized by having a complete control and knowledge of the environment:

- Deterministic actions: the effect of actions taken by the agent is always the same and it is known in advance.

- Static: the environment only changes when the agent executes an action. No feedback from the environment is needed.

- Fully-observable: fully-known initial state. As the result of an action is deterministic, the agent always knows the state.

Classical planning problems correspond to a path-finding search in a directed labeled graph whose nodes are the states and whose edges represent the transitions made possible by each action. In classical planning the solution is a plan, i. e., a sequence of actions that transform the initial state into a state satisfying the goals.

To specify the problems to the planners it is necessary to have a problem specification language in which all the characteristics of the problem are described. Some examples are STRIPS (Stanford Research Institute Problem Solver) (Fikes and Nilsson, 1971) or ADL (Action Description Language) (Pednault, 1994). The Planning Domain Description Language (PDDL) is a standard language to define planning problems in a domain-independent way. PDDL was originally defined in 1998 (Ghallab et al., 1998), based on STRIPS and ADL to carry out the International Planning Competition (IPC) and has evolved along different editions of the competition. PDDL1.2, the official language in the first edition of the IPC, separates the definition of a problem into two parts: domain and problem instance. The *domain* expresses the object types and available actions and the *problem instance* indicates the particular initial state and goals of the task. The distinction between domain and problem is important because problems are commonly classified by their domain. The reason is that problems of the same domain usually share a common structure. The benchmark domains that we will use in the empirical evaluation are analyzed in Section 1.5.1.

All problems considered in this thesis are expressed in PDDL. PDDL describes problems in predicate logic. However, planners are not required to reason in predicate logic. Most planners transform the PDDL problem definitions in predicate logic to other representations more suitable for the techniques they implement. A typical transformation is to *ground* the planning task, i. e., transform it to an equivalent propositional definition, so that states are defined by a set of Boolean propositions. In the context of this thesis we will assume a representation based on the SAS$^+$ representation (Bäckström and Nebel, 1995) in which states are defined by means of finite-domain variables. The translation PDDL to SAS$^+$ is performed automatically (Helmert, 2009). The finite-domain representation is presented in detail in Section 1.1.2.

1.1.1 Cost-Optimal Classical Planning

Several questions may be asked about a classical planning instance, defining different types of problems: plan existence, satisficing planning and optimal planning. Plan existence is the decision problem of given a planning instance determine whether there exists a plan or not. Satisficing planning consists of finding a plan as good as possible, i. e., those with lower cost are preferred. Finally, optimal planning consists of finding a plan and proving that it is optimal, i. e., that no plan with lower cost exists.

Even though all these questions can be shown to be PSPACE-COMPLETE (Bylander, 1994), in practice, optimal planning is much harder than satisficing planning. In most cases, proving that a solution is optimal takes more computational resources than finding any solution.

This thesis is about computing optimal solutions to classical planning problems. In the next sections, we give the formal definition of classical planning problems and succinctly review the state-of-the-art techniques to solve them optimally.

1.1.2 Formalization of Classical Planning: Finite-Domain Representation

A finite-domain variable planning task is defined as a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$. $\mathcal{V}$ is a set of *state variables*, and every variable $v \in \mathcal{V}$ has an associated *finite domain* D_v. A *partial state* p is a function on a subset of variables $\mathcal{V}_p \subseteq \mathcal{V}$ that assigns each variable $v \in \mathcal{V}_p$ a value in its domain, $p[v]$. A *state* s is a complete assignment to all the variables. A *fluent* is an assignment to a single variable and is usually identified as a pair variable-value, $\langle v, val \in D_v \rangle$. Thus, a partial state p can be defined as a set of fluents and is associated with the set of states that satisfy the partial assignment, $\{s \mid p[v] = s[v] \forall v \in \mathcal{V}_p\}$. We denote as $p|'_{\mathcal{V}}$ the projection of p over the set of variables $\mathcal{V}' \subseteq \mathcal{V}_p$, i. e., as the partial assignment over $\mathcal{V}'$ having the same value as p for variables it is defined for.

s_0 is the *initial state* and $s_\star$ is the partial state that defines the goals. $\mathcal{O}$ is a set of operators (also called actions), where each operator is a tuple $o = (pre(o), \mathit{eff}(o), c(o))$, where $pre(o)$ and $\mathit{eff}(o)$ are partial assignments over $\mathcal{V}_{pre(o)}$ and $\mathcal{V}_{\mathit{eff}(o)}$ that represent the *preconditions* and *effects* of the operator, respectively, and $c(o) \mapsto \mathbb{R}_0^+$ is the non-negative cost of o. The set of preconditions, $pre(o)$ can be split depending on whether they are affected by the operator effects or not. The *prevail* conditions of the operators are the subset of preconditions that are not affected by the effects of the operator, $prev(o) = \{f_i \mid f_i \in pre(o) \text{ and } v \notin \mathcal{V}_{\mathit{eff}(o)}\}$.

An operator $o \in \mathcal{O}$ is applicable (in progression) in a state s if $pre(o) \subseteq s$. The state $o(s)$ resulting from the application of o in s is defined as $o(s) = s|_{\mathcal{V} \setminus \mathcal{V}_{\mathit{eff}(o)}} \cup \mathit{eff}(o)$. While applicability of operators in progression is defined over complete states, in order to perform regression from the goals of the problem we define the reversed applicability over partial states. An operator $o \in \mathcal{O}$ is applicable in a partial state s over $\mathcal{V}_s$ in regression if s is consistent with the operator effects and prevails, $\forall v \in \mathcal{V}_s : (v \notin \mathcal{V}_{\mathit{eff}(o)} \text{ or } s[v] = \mathit{eff}(o)[v])$ and $(v \notin \mathcal{V}_{prev(o)} \text{ or } s[v] = prev(o)[v])$ and is *relevant* to s, $V_s \cap \mathcal{V}_{\mathit{eff}(o)} \neq \emptyset$. The resulting partial state s' obtained from the application in regression of o in s is defined as $s' = (s \cup \mathit{eff}(o))|_{\mathcal{V} \setminus \mathcal{V}_{pre(o)}} \cup pre(o)$.

A planning task defines a *state space* as a labeled transition system $\Theta(\Pi) = (\mathcal{S}, L, T, s_0, S_\star)$, where $\mathcal{S}$ is the set of all states. L is a set of transition labels corresponding to the operators of the planning task. T is the set of transitions, where each transition is a tuple $\langle s, l_o, s' \rangle$ with $s, s' \in \mathcal{S}$ and $l_o \in L$ and $\langle s, l_o, s' \rangle \in T$ if and only if o is applicable in s, giving s' as result. In that case, we also write $s \xrightarrow{l_o} s'$ to denote that there is a transition from s to s' labeled with l_o. Finally, s_0 is the initial state and $S_\star \subset \mathcal{S}$ is the set of states satisfying $s_\star$.

A solution *plan* is a sequence of operators, $\pi = (o_1, \ldots, o_n)$ related to a sequence of states $(s_0, s_1, \ldots, s_n)$ such that s_0 is the initial state and $s_\star \subseteq s_n$ and s_i results from executing the operator o_i in the state s_{i-1}, $\forall i = 1..n$ in progression. The cost of a plan is the sum of the cost of its operators, $c(\pi) = \sum_{o_i \in \pi} c(o_i)$. A plan is *optimal* if no other plan of lower cost exists.

STRIPS and SAS$^+$ have been shown to have equivalent expressive capabilities (Bäckström and Nebel, 1995). Still, we chose the finite-domain representation definition because of its practical advantages. For example, finite-domain variables encoding takes implicitly into account invariants of the problem (see Section 1.1.4). Second, it has associated the causal graph and domain transition graphs described in the following section to capture the structure of the planning task. For a detailed description of how the finite-domain representation can be automatically obtained from the classical STRIPS representation, we refer the reader to the paper by Helmert (2009).

1.1.3 Causal Graph and Domain Transition Graphs

The causal graph is a directed graph that represents the interaction between variables in the planning task. Each node is associated with a finite-domain variable and there is an edge from p to q if the value of variable q *depends* on the value of p. It was first used in the context of generation of hierarchical abstractions (Knoblock, 1994; Bacchus and Yang, 1994). Later, it was defined for unary operator tasks (Brafman and Domshlak, 2003). We use the definition given by (Helmert, 2004) in the context of heuristic search planning:

Definition 1.1 (Causal Graph)**.** *Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a finite-domain variable planning task. Its causal graph is a directed graph $(\mathcal{V}, A)$ containing an arc (u, v) if and only if $u \neq v$ and there exists an operator $(pre(o), \mathit{eff}(o), c(o)) \in \mathcal{O}$ such that $v \in \mathcal{V}_{\mathit{eff}(o)}$ and $u \in \mathcal{V}_{\mathit{eff}(o)} \cup \mathcal{V}_{pre(o)}$.*

The causal graph of a planning task offers a global view of the interaction between variables. On the other hand, domain transition graphs describe the transitions between values of a single variable of the problem, and their relation with other variables (Jonsson and Bäckström, 1998).

Definition 1.2 (Domain Transition Graph)**.** *Consider a finite-domain variable planning task with variable set $\mathcal{V}$, and let $v \in \mathcal{V}$. The domain transition graph G_v is the labeled directed graph with vertex set $\mathcal{D}_v$ and which contains an arc (d, o, d') if and only if $d \neq d'$ and there is an operator $(pre(o), \mathit{eff}(o), c(o))$ where $(pre(v) = d$ or $v \notin \mathcal{V}_{pre(o)})$ and $\mathit{eff}(v) = d'$. For each arc (d, o, d') we say that there is a transition of v from d to d' enabled by o.*

1.1.4 Invariants and Mutexes

Invariants are properties that are satisfied by all states reachable from the initial state. We are interested in two types of invariants: mutex invariants and monotonicity invariant groups.

Definition 1.3 (Mutually exclusive fluents)**.** *A set of fluents $M = \{p_1, \ldots, p_m\}$ is a set of mutually exclusive fluents of size m (mutex of size m) if and only if any state s such that all elements of M are true in s can be proven to be unreachable from the initial state.*

Since there are exponentially many sets of fluents, usually only *mutex pairs* are considered ($m = 2$). A *mutex group* is a set of fluents so that every pair of the set is a mutex pair. Even restricting ourselves to mutex pairs, the problem of finding all mutex pairs is still intractable. Hence, we rely on sound but incomplete algorithms (all the pairs they find are guaranteed to be mutex but they do not necessarily find all of them):

- *Invariant monotonicity* (Helmert, 2009): It proves the invariant by induction: checking the initial state and proving that if the invariant holds in a state, it also holds in its successors. Thus, mutex pairs are inferred from the operator description if every operator that adds a fluent in the group also deletes another.

- h^m (Haslum and Geffner, 2000): h^m heuristics are an admissible estimation of the cost of reaching each m-tuple of fluents. When computing the heuristic on the initial state, $h^m(s_0)$, it can be proven that some pairs cannot be reached and, consequently, they are mutex pairs. In practice, mutex pairs are computed using $h^2(s_0)$, since the cost of computing h^m grows exponentially with m.

Invariant monotonicity groups extend the mutex group invariants by requiring one element in the mutex group to be true in every reachable state. While mutexes represent groups of fluents such that *at-most-1* can be true at the same time, invariant groups require *exactly-1* fluent to be true in any state.

Definition 1.4 (Invariant monotonicity groups)**.** *Monotonicity invariant groups are sets of fluents such that exactly one fluent must be true in every reachable state.*

Monotonicity invariant groups can be automatically inferred from the planning task description by the same invariant monotonicity used to infer mutexes. For the *exactly-1* invariant to hold, it is necessary that any operator that deletes a fluent in the set, also adds another. The automatic PDDL to SAS$^+$ translation method mentioned above uses monotonicity invariant groups to determine the SAS$^+$ encoding of the task. *exactly-1* invariants are variables in the SAS$^+$ encoding and *at-most-1* invariants can be easily transformed to *exactly-1* invariants by adding an auxiliary *none-of-those* fluent to the SAS$^+$ task.

Invariants are useful to prune states in the search since any state that violates an invariant can be safely pruned. By definition, a forward search from the initial state will never find a state violating an invariant, but this is not the case in other types of searches that will be considered in this thesis, like regression or searches in abstract state spaces.

1.2 Planning as Heuristic Search

Most state-of-the-art cost-optimal planners employ heuristic search (Bonet and Geffner, 2001). Search algorithms traverse the state space graph, $\Theta = (\mathcal{S}, L, T, s_0, S_\star)$ in order to find a path between the initial state and a state satisfying the goals. In contrast to satisficing planning, in order to prove optimality search algorithms must exhaustively explore all the states for which it cannot be proved that they do not belong to an optimal solution. This can be done with best-first search algorithms among others. Best-first search algorithms maintain two lists of search nodes: the *open* and the *closed* lists. The *open* list contains all the *generated states* that have not yet been expanded. The *closed* list contains all the expanded states, whose successors have already been generated.

Uniform-cost search is a best-first search[1] that expands the nodes in ascending order of g-value (Dijkstra, 1959). The g-value of a node n, $g(n)$, is the cost of reaching n from the initial state, i. e., the sum of the action costs in the best path found from s_0 to n. As action costs cannot be negative, the g-values of expanded states never decrease. Thus, whenever a goal state is selected for expansion, an optimal plan has been found. However, the algorithm may expand an exponential number of nodes in the size of the planning task.

In order to relieve the exponential behavior of the algorithm, we introduce a heuristic that *informs* the algorithm. The *remaining cost* of a state s, $h^*(s)$ is defined as the cost of a cheapest path from s to a state in $S_\star$, or ∞ if there is no such path. A *heuristic* is a function $h : \mathcal{S} \to \mathbb{R}_0^+ \cup \{\infty\}$ which estimates the remaining cost of a state. A heuristic is *perfect* if it coincides with h^*. A heuristic is *admissible* if it never overestimates the remaining cost, that is, $\forall s : h(s) \leq h^*(s)$. A heuristic is *consistent* if for every transition $(s, l_o, s') \in T$, $h(s) \leq h(s') + c(o)$. Note that every consistent heuristic is also admissible if and only if it correctly estimates a cost of 0 for goal states (as it happens for all the heuristics we will consider). In this thesis we are interested in admissible heuristics that prove lower bounds for the cost of solving the problem from an arbitrary state s. Once a (possibly suboptimal) plan is known with cost C, we can prune any node n with $f(n) = g(n) + h(n) \geq C$.

Algorithms that do not make use of a heuristic, such as uniform-cost search are often called blind or uninformed. We are interested in admissible algorithms that are guaranteed to return optimal solutions when guided by an admissible heuristic. Some examples are A* (Hart et al., 1968) and iterative-deepening A* (IDA*) (Korf, 1985) that uses memory linear in the solution length. In planning, the former is more common due to the presence of a huge number of transpositions (different action sequences that lead to the same state) and the fact that memory is not always the scarcest resource because the heuristics are computationally demanding.

[1]"Uniform-cost Search" is also known as Dijkstra's algorithm or "Weighted Breadth-First Search". "Uniform-cost search" is arguably a more precise name (Felner, 2011), so we will adopt it in this thesis.

A* expands the nodes with lowest $f(n) = g(n) + h(n)$ first so that no node with $f(n) > h^*(s_0)$ is ever expanded. If the heuristic is consistent, A* will never re-expand a node (because when it expands a node, it has optimal g-value). This guarantees optimality on the number of node expansions up to tie-breaking, so that no other algorithm with the same heuristic and no additional information can find the optimal solution with less expansions (Hart et al., 1968; Dechter and Pearl, 1985). Besides, storing all expanded states in a closed list allows A* to prune duplicate states, which is mandatory given the high number of permutations in most domains. This has made A* the predominant algorithm for cost-optimal planning. Note that uniform-cost search is equivalent to an A* search without a heuristic ($h(s) = 0$ for all $s \in \mathcal{S}$).

There is an important exchange of ideas between the planning and the heuristic search communities. The heuristic search community focuses on research on heuristics and search algorithms. Thus, some works relevant for this thesis come from the heuristic search area. The main difference between the two fields is that, in heuristic search, a solver is made for problems of a single domain, with an ad-hoc heuristic and many optimizations in the implementation of the algorithm. In planning a unique solver, the planner, must be able to solve problems of different domains. Another difference has been the focus on different types of domains. In heuristic search, the focus is made on more combinatorial domains, i. e., permutation puzzles like the sliding-tile puzzle or the Rubik's Cube (Culberson and Schaeffer, 1998; Korf, 1997). In planning the benchmark domains are more diverse and some of them are even polynomially solvable (Helmert, 2003; Helmert, 2006a). Even those cases are interesting because it is necessary to automatically find and exploit the characteristics that make them easy.

1.2.1 Domain-Independent Planning Heuristics

In planning, heuristics are automatically derived from the problem description. Current state-of-the-art heuristics for planning can be classified in five different families (Helmert and Domshlak, 2009): delete relaxation, critical paths, landmarks, abstractions and the recent flow-based heuristics (Bonet, 2013; Bonet and van den Briel, 2014a).

Delete Relaxation Heuristics The delete relaxation heuristic h^+ (Hoffmann and Nebel, 2001) estimates the cost of solving a task Π as the optimal cost of the relaxed planning task without deletes (Π^+). Π^+ is derived from Π by ignoring the negative effects of all operators, i. e., considering that they add more facts to be true but do not delete any fact that has already been achieved. Theoretical and experimental analysis have proven that h^+ is a very informative heuristic in a good number of planning benchmarks (Hoffmann, 2005; Helmert and Mattmüller, 2008; Betz and Helmert, 2009). Unfortunately, computing h^+ in the general case is NP-HARD, so it is not feasible to use it to guide the search. Instead, other relaxations are applied to obtain polynomial approximations of h^+. In most cases, admissibility is dropped as it happens in the additive heuristic h^{add} (Bonet and Geffner, 2001), the FF-heuristic h^{FF} (Hoffmann and Nebel, 2001) and others. However, in optimal planning we are interested in admissible estimations of h^+ such as h^{max} (Bonet and Geffner, 2001) or the LM-CUT heuristic, h^{LMcut} (Helmert and Domshlak, 2009).

Critical paths The h^m heuristics (Haslum and Geffner, 2000) estimate the cost of reaching a set of atoms as the cost of the most costly subset of size m. They can be computed by a reachability analysis considering tuples of facts of cardinality up to m. It is also possible to compute h^m as the h^{max} heuristic on a modified planning task Π^m, whose atoms represent tuples of atoms of the original task (Haslum, 2009). The complexity of computing h^m is exponential in m but polynomial of grade m in the size of the planning task for a fixed m. It can be shown that $h^1 \equiv h^{max}$ and $h^n \equiv h^*$, where n is the number of finite-domain variables. Moreover, $h^m \leq h^{m+1}$ for any $1 \leq m \leq n-1$. Thus, the parameter m establishes a trade-off between informativeness and efficiency to compute. In practice, h^2 has been successfully used, since the heuristic is often computationally too expensive for greater values of m.

Landmarks Landmarks are facts that must be true at some point in every plan. In satisficing planning they have been used as sub-goals in the search (Porteous et al., 2001) or as inadmissible heuristics (Richter et al., 2008). Some examples are h_{LA}, a heuristic based on cost-partitioning over landmarks (Karpas and Domshlak, 2009), and LM-CUT (Helmert and Domshlak, 2009), h^{LMcut}. LM-CUT is based on *disjunctive action landmarks*, i. e., a set of planning actions so that in every optimal plan at least one action in the set must be executed. Even though they are inspired by different ideas, most landmark-based heuristics are estimations of the delete-relaxation heuristic h^+. Indeed, redefining the LM-CUT heuristic as a hitting set problem over all landmarks of the relaxed task is equivalent to h^+ (Bonet and Helmert, 2010).

Abstraction Heuristics Abstraction heuristics transform the problem to a simpler problem that is tractable and use the solution cost of the simpler problem as an estimation in the original problem. Usually, these transformations are homomorphisms that map states in the original state space to a smaller set of abstract states. Transitions are preserved in the abstract state space, so that admissibility of the resulting heuristic is always guaranteed. Different types of abstraction functions can be defined, such as Pattern Databases (PDBs) (Culberson and Schaeffer, 1998; Edelkamp, 2001; Haslum et al., 2007), Merge-and-Shrink (M&S) (Helmert et al., 2014) or CounterExample-Guided Cartesian Abstractions (Seipp and Helmert, 2013). In this thesis we will study in more detail PDBs and M&S abstractions. A detailed review of abstractions is presented in Chapter 6.

Flow-based heuristics Flow-based heuristics are a recent new family of heuristics that map the planning problem into an integer programming task, in which variables are associated with propositions and actions of the problem and represent the number of times that the proposition should be achieved, or the actions are applied. Constraints forbid the application of actions if their preconditions had not been achieved enough times (considering those propositions that are deleted by the actions). In order to solve the problem, it is once again relaxed into a linear programming task which is easily solved (van den Briel et al., 2007; Bonet, 2013). Moreover, additional constraints may be incorporated in this linear programming encoding, such as disjunctive action landmarks, in order to get more informed estimations (Bonet and van den Briel, 2014a).

In summary, in the last years, there has been a lot of effort in developing well-informed heuristics that pay off the extra effort to compute. Indeed, while blind search was a state-of-the-art approach in cost-optimal planning in 2008, in IPC-2011, several heuristics like merge-and-shrink or LM-CUT exhibited stronger performance.

1.2.2 Combining Heuristic Estimates

Several estimates $h_1(s), \ldots, h_k(s)$ may be computed for a state s using different heuristics or even different configurations of the same heuristic. When aggregating those heuristic estimations, one must be careful to preserve admissibility. The simplest way to do so is to take the *maximum* of those values $h(s) = \max\{h_i(s)\}$. Maximizing over multiple heuristics can be very beneficial (Holte et al., 2006).

Even more informed estimations can be made by an *additive* schema (Korf and Felner, 2002; Felner et al., 2004), i. e., a controlled sum of the individual heuristic estimations. If the cost of each operator is not taken into account twice by different heuristics then admissibility is preserved. Thus, if each heuristic h_i only takes into account a subset of operators $\mathcal{O}_i$ (all the other operators are considered to have a cost of 0) and those subsets are disjoint ($\mathcal{O}_i \cap \mathcal{O}_j = \emptyset, \forall i \neq j$) the resulting estimate is admissible. A more general way to guarantee admissibility in an additive setting is a cost-partitioning schema (Katz and Domshlak, 2010b). A cost-partitioning is a function that divides the cost of the operators between the heuristics.

In the case of abstraction heuristics, the optimal cost-partitioning can be determined in polynomial time with respect to the size of the abstract state spaces through a Linear Programming (LP) encoding (Katz and Domshlak, 2010b). However, the encoding is usually too large and expensive to solve, taking up to hours to evaluate a single state, which makes unfeasible using it as a heuristic. A way to make it practical is to determine the optimal cost-partitioning of very simple abstractions for some states and use the maximum of those cost-partitionings for every state in the search (Karpas et al., 2011).

Finally, recent work has also proposed alternative linear programming encodings to aggregate the heuristic estimations (Pommerening et al., 2013), based on which operators contribute to each heuristic. This is less informed than the optimal cost-partitioning but the size of the LP encoding is much more reasonable and can be solved on a per-state basis. Moreover, multiple constraints inferred from different heuristics may be combined in a single linear programming encoding, leading to more informed heuristics. An LP-framework can also be useful as a basis to compare different heuristics (Pommerening et al., 2014).

1.2.3 Search Directionality: Regression and Bidirectional Search

Heuristic search can be performed in the forward or backward direction. Forward search starts from the initial state and searches towards the goal states. Thus, forward search performs *progression*, i. e., whenever a state s is expanded their successors are generated by applying the operators on s. Backward search, on the other hand, performs *regression*, starting from the goal states of the problem and advancing towards the initial state. Whenever a state is expanded the algorithm generates their predecessors, i. e., all the states s' such that $s' = o(s)$ for an operator o.

Since the first heuristic search approaches to planning, the search has been performed in both directions, e. g., the HSP and HSPr planners (Bonet and Geffner, 2001). However, most of the research has focused on forward search because it provided better empirical results and there are intrinsic difficulties for the backward search in planning. In planning, there is usually an exponential number of goal states in the size of the planning task, so they cannot be explicitly represented. Instead, partial states are used. This is appropriate since the goal of the planning task is defined as a partial state even when there are exponentially many goal states. However, the partial-state representation has problems, too. For example, duplicate detection becomes harder due to the *subsumption of partial states*, a phenomenon that occurs when a partial state is a subset of another. In that case, both partial states are not detected as duplicates by typical hash-table techniques, even though one is contained in the other.

A second problem of backward search is that, in most domains, there is a huge number of *spurious states*, i. e., states that are not reachable from the initial state. In those cases, most of the search might be performed over irrelevant parts of the state space. While in forward search there are also *dead-end* states from which the goals are not reachable, they are not so common as *spurious states* because by definition the problem has a solution if and only if the initial state is not a dead end, while it may have exponentially many spurious goal states. An empirical analysis in planning domains showed that the negative impact of spurious states is larger than that of subsumption in regression (Alcázar et al., 2014).

In the rest of the thesis, when we use the term *explicit-state* backward search, we mean partial-state backward search. We use this term in contrast with symbolic search that will be presented next.

Despite the problems of regression in planning, there are good reasons to consider both search directions. Planning state spaces are usually asymmetric, so that some problems are much easier when searching in backward direction. This is called *directionality* of the domains (Massey, 1999). Recent works have also considered to revisit the use of regression in planning, extrapolating advances in forward search of the last decade to the case of regression (Alcázar et al., 2013).

Moreover, there is no need of restricting the search to one single direction. Bidirectional search interleaves two searches: a forward search from the initial state to the goal and a regression search from the goal states to the initial state. When both frontiers meet, a solution for the problem has been found, though proving optimality requires continuing the search until stronger conditions hold. The promise of bidirectional search is to reduce the search depth by half (performing two searches of depth $\frac{d}{2}$ instead of one of depth d). As the search complexity is exponential in the search depth, there is potentially an exponential gain.

However, the use of heuristics is not simple in bidirectional search. Bidirectional heuristic search has been thoroughly studied in the heuristic search community (Kwa, 1989; Kaindl and Kainz, 1997; Felner et al., 2010), concluding that in many cases it is not better than a simple A* search. Even though finding a solution may be easier, the bidirectional search is not better to prove optimality. This, together with the drawbacks of regression in planning, has been an impediment for applications of bidirectional heuristic search for planning.

1.3 Symbolic Search Planning

Contrary to explicit search, which is the most common way of searching in planning, symbolic search takes advantage of succinct data structures, such as Binary Decision Diagrams (BDDs) (Bryant, 1986), to represent sets of states. Furthermore, the symbolic representation also allows us to manipulate sets of states through efficient operations. Therefore, the symbolic version of search algorithms are similar to standard explicit-state search algorithms, but expand entire sets of states at once.

Symbolic reachability with BDDs was originally introduced in Model Checking (McMillan, 1993) and it was brought later to heuristic search with algorithms like symbolic breadth-first search and a symbolic version of A*, BDDA* (Edelkamp and Reffel, 1998). The model-checking integrated planning system (MIPS) (Edelkamp and Helmert, 2001) was the first planner using symbolic BDDA* for domain-independent classical planning. Some alternative implementations of BDDA* have been proposed. ADDA* (Hansen et al., 2002) uses ADDs to represent the open and closed lists of the search. Set A* (Jensen et al., 2002) refined the partitioning in a matrix representation of g- and h-values. Additionally, Symbolic Pattern Database Heuristics were proposed for domain-independent planning (Edelkamp, 2001; Edelkamp, 2002; Edelkamp, 2005) and later extended to Symbolic Partial PDBs (Edelkamp and Kissmann, 2008b). Finally, *state-set branching* (Jensen et al., 2008) considered the partitioning of the transition relation according to the heuristic value.

Most recent advances in symbolic search planning have been promoted by the planner GAMER (Edelkamp and Kissmann, 2009), who won the optimal track of IPC-2008. An improved version of GAMER with a better variable ordering and PDB selection strategy (Kissmann and Edelkamp, 2011) also participated in the IPC-2011, though with more modest results. The advances in heuristics for explicit-state search were quite effective in a number of domains, suggesting that using good heuristics may be more important than the efficient symbolic exploration of the state spaces. However, GAMER still beats explicit-state planners in several domains (Kissmann, 2012), so that symbolic search is still an important technique to take into account. Chapter 2 fully describes the symbolic search approaches of symbolic search planning that we will use in the rest of the thesis.

1.4 Objectives of the Thesis

The ultimate goal of this thesis is to improve the current state-of-the-art in cost-optimal planning. To do so, we focus on two particular techniques: symbolic search and abstraction heuristics. In spite of all the previous efforts that successfully showed the benefits of these techniques, there are still some opportunities to improve their performance. The particular goals that we pursue in this thesis are:

- Identify current bottlenecks of symbolic planning and improve the performance of symbolic planners. In particular, we identified two points of improvement for symbolic planners:

    1. Image computation: An analysis of the performance of symbolic search planners shows that the main bottleneck is the successor generation performed with the image operation. We propose and analyze different methods to optimize image computation.
    2. While state invariants have been used to prune states in explicit-state regression search, no analysis has been made about using them on symbolic search. We consider the use of state invariants in symbolic search.

- On the other hand, recent advances on heuristics for optimal planning have only been applied to the case of explicit-state search, including M&S abstraction heuristics or LM-CUT. Symbolic planners use symbolic PDB abstractions. As PDB abstractions were shown to be a special case of M&S abstractions, there is an opportunity to use better heuristics in symbolic search.

    We show how M&S heuristics can be used in symbolic A* search, and perform an empirical comparison with symbolic PDBs. We also study how symbolic search can be used to generate more powerful M&S heuristics.

- Finally, the good results of bidirectional symbolic search motivate us to use heuristics in that setting. We study how the well-known difficulties of bidirectional heuristic search are less problematic in the case of symbolic search with perimeter abstraction heuristics.

1.5 Methods of Empirical Evaluation

The implicit goal of this thesis is to increase our understanding of planning problems and techniques to develop more efficient domain-independent planners. In this section we describe the methods that we will follow to perform the empirical evaluation of the planning techniques. Fortunately, there have been many efforts in the automated planning community to develop methods and tools to determine how good a planner is and compare the performance of different planners. The International Planning Competition (IPC) is an event organized since 1998 periodically,[2] with the objective of comparing the performance of current state-of-the-art planners as fairly as possible.

[2]The IPC was typically organized every two years until 2008. Since then, it has been arranged every three years.

IPC rules have become a *de facto* standard to compare planner performance and evaluate research in planning. All the benchmarks are described in the PDDL language, which is used to provide domain and problem descriptions to planners. As the product of the 2008 and 2011 IPC editions, a software that automates the experimentation was released, with the goal of improving the experimentation (Linares López et al., 2013). For the thesis experiments we use a new version of the IPC evaluation software, made by Carlos Linares López. The software automatically tracks the execution of the planners in the desired benchmarks, measuring the time and memory usage. Moreover, it validates all the solution plans using VAL, the automatic validation tool (Howey et al., 2004).

The settings are taken from IPC-2014. All planners are executed with a time bound of 30 minutes and a limit of 4 GB of memory usage. We used two different hardware settings for the experiments. In Part I, the experiments are run on a single core of an Intel Xeon X3470 processor at 2.93 GHz. In Parts II and III, all experiments were conducted on a cluster of Intel E5-2660 machines with 64 GB of main memory and 16 cores running at 2.20 GHz.

1.5.1 Benchmark Suite

A set of diverse benchmark problems is defined in each competition to test the planners' capabilities of solving problems of different kinds. Planning problems are classified in domains. Problems of the same domain are of the same type, having a common structure but varying in difficulty. As new domains are introduced in each competition to guarantee that solvers are domain-independent, there is a good set of domains and problems to test planner capabilities.

In this thesis, we consider a benchmark set of 44 domains from all competitions since IPC-1998 until IPC-2011, with 1396 problems in total. The selected domains only use a subset of PDDL that uses a STRIPS description, without numerical fluents, conditional effects, derived predicates or other ADL features that are commonly translated into them. The techniques and algorithms used in this thesis can easily be extended to deal with more PDDL features. Indeed, support of conditional effects in symbolic search algorithms was explicitly required to participate in the IPC-2014. However, domains with more expressive PDDL features are not good for comparison with other state-of-the-art optimal planners. Therefore, we stick to the standard benchmark for optimal planning that has been used in many previous works in the literature, such as the LM-CUT (Helmert and Domshlak, 2009) or merge-and-shrink (Helmert et al., 2014) heuristics. Since IPC-2008, there is a specific track for optimal planners with different problem instances. For the domains of the 2008 and 2011 editions, our benchmark set contains all the problems used in the optimal track.

Table 1.1 summarizes the characteristics of the IPC domains we consider. The *Source* column shows the edition of the competition in which the domain was first used. Next, the # column is the number of problems of each domain. Action costs are a relevant characteristic important to explain the results obtained by different techniques. The table depicts the maximum number of action costs in a problem and the set of costs used. Domains can be classified in different categories according to the distribution of action costs. *Unit-cost* domains have only actions of cost 1 and can be interpreted as plan length minimization domains. Also, domains having *zero-cost* actions require a special treatment, since the applicability of those actions does not count at all for the plan cost. Finally, we have to distinguish domains with a large number of different costs and/or a large distance between the minimum and maximum cost, because heuristic search approaches can be very sensitive to the variability or ratio of action costs (Cushing et al., 2011; Wilt and Ruml, 2011).

Domain	Source	# Instances	Action Costs \#	Action Costs Costs
GRID	IPC-98	5	1	1
GRIPPER	IPC-98	20	1	1
LOGISTICS98	IPC-98	35	1	1
MPRIME	IPC-98	35	1	1
MYSTERY	IPC-98	30	1	1
BLOCKSWORLD	IPC-2000	35	1	1
FREECELL	IPC-2000	80	1	1
LOGISTICS00	IPC-2000	28	1	1
MICONIC	IPC-2000	150	1	1
DEPOT	IPC-2002	22	1	1
DRIVERLOG	IPC-2002	20	1	1
ROVERS	IPC-2002	40	1	1
SATELLITE	IPC-2002	36	1	1
ZENOTRAVEL	IPC-2002	20	1	1
AIRPORT	IPC-2004	50	1	1
PIPESWORLD-NT	IPC-2004	50	1	1
PIPESWORLD-T	IPC-2004	50	1	1
PSR-SMALL	IPC-2004	50	1	1
OPENSTACKS06	IPC-2006	30	1	1
PATHWAYS-NONEG	IPC-2006	30	1	1
TPP	IPC-2006	30	1	1
TRUCKS	IPC-2006	30	1	1
ELEVATORS08	IPC-2008	30	13	$\{0, 6, \ldots, 37\}$
OPENSTACKS08	IPC-2008	30	2	$\{0, 1\}$
PARCPRINTER08	IPC-2008	30	22	$\{0, 125, \ldots, 224040\}$
PEG-SOLITAIRE08	IPC-2008	30	2	$\{0, 1\}$
SCANALYZER08	IPC-2008	30	2	$\{1, 3\}$
SOKOBAN08	IPC-2008	30	2	$\{0, 1\}$
TRANSPORT08	IPC-2008	30	37	$\{1, 10, \ldots, 190\}$
WOODWORKING08	IPC-2008	30	6	$\{5, 10, 15, 20, 30, 45\}$
BARMAN11	IPC-2011	20	2	$\{1, 10\}$
ELEVATORS11	IPC-2011	20	13	$\{0, 6, \ldots, 37\}$
FLOORTILE11	IPC-2011	20	4	$\{1, 2, 3, 5\}$
NOMYSTERY11	IPC-2011	20	1	1
OPENSTACKS11	IPC-2011	20	2	$\{0, 1\}$
PARCPRINTER11	IPC-2011	20	22	$\{0, 125, \ldots, 224040\}$
PARKING11	IPC-2011	20	1	1
PEG-SOLITAIRE11	IPC-2011	20	2	$\{0, 1\}$
SCANALYZER11	IPC-2011	20	2	$\{1, 3\}$
SOKOBAN11	IPC-2011	20	2	$\{0, 1\}$
TIDYBOT11	IPC-2011	20	1	1
TRANSPORT11	IPC-2011	20	23	$\{1, 10, \ldots, 190\}$
VISITALL11	IPC-2011	20	1	1
WOODWORKING11	IPC-2011	20	6	$\{5, 10, 15, 20, 30, 45\}$

Table 1.1: Table of domains in our benchmark suite, classified by the IPC edition in which they were proposed.

1.5.2 Evaluation metrics

We measure the performance of planners with two types of metrics: coverage and time score. Note that metrics that take into account the quality of solution plans found do not apply in our case since we are only interested in optimal plans. Thus, quality simplifies to coverage in this case.

Coverage The coverage is the total number of problems solved from the benchmark within the time and memory bounds. It is the official metric for the optimal track of the IPC since 2008.

Time score While coverage only takes into account whether a planner solves a problem or not, time score metrics reward planners for solving the problems as early as possible. Planners get a score in the interval $(0, 1]$ for each problem solved. The fastest planner is awarded one whole point, while other planners solving the same problem in more time receive a fraction of a point. If a planner does not solve the problem before the time limit it gets 0 points for that problem. The total score of a planner in a domain is the sum of its score in all the problems of that domain.

Several equations may be defined to determine the score of the planners. In the context of this thesis, we will use the time score metric used in the learning track of IPC-2011. Let t^* be the minimum time in seconds required by the fastest planner to solve the problem, rounding all the times to the upper second. Then, the time score of a planner that solves the problem in t, is computed following Equation 1.1. Any planner solving the problem in less than one second receives the maximum score. A $\log_{10}$ is applied to the ratio between the time of the planner and the best time, to diminish the impact of small differences in time, due to the exponential nature of planning tasks.

$$\text{time\_score}(t, t^*) = \begin{cases} 1, & \text{if } t \leq 1s \\ \frac{1}{1+\log_{10}(t/t^*)}, & \text{if } t > 1s \end{cases} \tag{1.1}$$

However, the time metric has two important drawbacks. On the one hand, it is not independent of irrelevant alternatives. Since the scores depend on the planners under consideration (t^* changes depending on the planners included in the comparison), adding a new planner may change the relative score of two planners. That means that the winner between two planners may depend on whether a third planner is included, biasing the results. The interpretation of the results should take this into account, using the coverage to determine which planner is stronger in general. Time score is useful to complement the coverage results because it reveals differences between planners in domains where coverage is the same.

On the other hand, the time score results on one table are incomparable to others because a different t^* was used to compute the score of each planner in a problem. This makes impossible a direct comparison of results in different chapters. To address this, total coverage is reported in all cases so that it is possible to compare approaches from different chapters. Moreover, in Chapters 5 and 10 we directly compare the most important configurations of each chapter.

Total score As we perform experiments in domains from different IPCs, the benchmarks are not completely uniform. On the one hand, some domains have more problems than others. The extreme cases are the domains GRID with only 5 problems and MICONIC with up to 150 problems. On the other hand, IPC-2011 reused all IPC-2008 domains and some of the problems of those domains. In order to report results as exact as possible, and comparable with other reportings in the literature, we use all the available problems for all the domains and report results of benchmark sets separately for the domain version of IPC-2008 and IPC-2011.

However, in order to perform a comparison as fair as possible, we do not use the sum of the results in each domain as the "total" score. In the spirit of giving the same weight to every domain, we will normalize total scores according to the number of problems of each domain. Moreover, IPC-2008 and IPC-2011 versions are accumulated, removing all duplicate problems.[3]

[3]This is not applied for the IPC-2006 version of OPENSTACKS because there is a substantial difference with the IPC-2008 and IPC-2011 versions in that it does not use action costs.

1.5.3 Planners

In order to evaluate the results of our research, in this thesis we have worked with two different planning systems that participated in the last editions of the IPC:

- **GAMER:** developed by Peter Kissmann and Stefan Edelkamp (2008); (2009); (2011); (2012). GAMER is a symbolic search planner that uses both symbolic bidirectional uniform-cost search and symbolic A* with symbolic PDBs.

- **FAST DOWNWARD PLANNING SYSTEM:** initially developed by Malte Helmert and Silvia Richter (2004); (2006) and then improved and greatly extended by numerous people including (in alphabetical order) Erez Karpas, Michael Katz, Gabi Röger, Jendrik Seipp and Matthias Westphal. Other substantial contributions have been made by Moritz Gronbach, Silvan Sievers, Raz Nissim and Manuela Ortlieb. FAST DOWNWARD is a heuristic search planner, highly configurable to run different search algorithms and heuristics. Configurations for optimal planning are based on A* search with several heuristics including PDBs (Haslum et al., 2007), M&S (Helmert et al., 2014) and LM-CUT (Helmert and Domshlak, 2009). Also, it is remarkable that a good amount (ten out of twelve) of the planners that were presented to the optimal track of IPC-2011 were based on FAST DOWNWARD.

For BDD manipulation, we use the version 2.5.0 of the CUDD library, developed by Somenzi (2012).

1.6 Thesis Outline

The main body of the thesis is divided into two parts. Part I comprises Chapters 2 - 5 and studies and develops symbolic search planning approaches. It starts by describing symbolic search planning basics and related work in Chapter 2. Chapters 3 and 4 analyze how to improve the performance of symbolic blind and symbolic heuristic search planning by implementing efficient image computation and using constraints derived from the problem, respectively. Part I finishes with an empirical analysis of symbolic blind search methods, showing that the previously mentioned improvements turn symbolic planning in a state-of-the-art optimal planning technique.

In Part II, from Chapters 6 to 9, with the goal of pushing the performance of symbolic optimal planning even further, we consider using abstractions in symbolic search planning. Chapter 6 is a survey of abstraction heuristic methods for planning, considering works in explicit and symbolic search. Chapter 7 presents how to carry one of the best abstraction techniques for domain-independent planning, the merge-and-shrink framework, to symbolic heuristic planning. We further develop the integration of merge-and-shrink abstractions and symbolic search in Chapter 8, where merge-and-shrink is used in combination with perimeter abstraction heuristics to develop a new heuristic for explicit-state planning. Finally, Chapter 9 puts all together, considering the use of symbolic bidirectional heuristic search informed by a diverse kind of abstraction heuristics.

The thesis concludes in Part III. Chapter 10 summarizes the empirical results of the thesis, comparing all the techniques proposed against other state-of-the-art planners. In Chapter 11, we gather the main results presented in this thesis and propose several research lines that could continue our work. Hope you enjoy ☺.

Part I

Symbolic Search Planning

Chapter 2

Symbolic Search Planning

Symbolic search performs operations over sets of states instead of working with individual states. These sets of states are succinctly represented as logical functions, often getting memory and time advantages over explicit-state search. In this chapter, we explain the basics of symbolic search in planning, as well as an in-depth explanation of symbolic search algorithms that will be important to understand the contributions of this thesis. Moreover, we present some implementation details that are a minor contribution of this thesis.

The chapter is organized as follows. First, we describe how to represent planning tasks as logical formulæ. Then, Binary Decision Diagrams (BDDs) are proposed as the data-structure used to represent those formulæ. Afterwards, we present the symbolic versions of the algorithms we will work with: uniform-cost search, bidirectional uniform-cost search and A*. Finally, we discuss some alternative representations to BDDs and justify the selection of BDDs to perform symbolic search planning.

2.1 Symbolic Representation of Planning Tasks

Symbolic search was originally proposed in the area of model checking (McMillan, 1993). It takes advantage of succinct data structures to represent sets of states through their *characteristic functions*. Given a set of states S, its *characteristic function* f_S is a Boolean function $f_S(x_1 \dots x_n) : \mathcal{S} \rightarrow \{\top, \bot\}$ that represents whether a given state belongs to S, where $\top$ stands for *true* and $\bot$ for *false*. The input of the function is the bit-vector description of a state represented by binary variables x_i, so that the function signature may be written as $f_S : \{0, 1\}^n \rightarrow \{\top, \bot\}$. Each finite-domain variable $v \in \mathcal{V}$ with domain D_v is represented with $\lceil \log_2 |D_v| \rceil$ binary variables.[1] The function returns $true$ ($\top$) if and only if the state belongs to the set of states S. To simplify the notation, we use the same symbol, S, to denote a set of states and its characteristic function. It will be inferred from the context whether we are referring to a set of states or its characteristic function. In terms of search, states are usually grouped by their g- or h-value. Whenever all the states in a set have the same g- or h-value, we will refer to the set as S_g and S_h, respectively.

The use of a characteristic function allows us not only to succinctly represent sets of states, but also to operate with them through function transformations. For example, the union ($\cup$) and intersection ($\cap$) of sets of states are derived from the disjunction ($\vee$) and conjunction ($\wedge$) of their characteristic functions, respectively. Also, the complement set (S^c) corresponds with the negation ($\neg$)

[1] Based on this transformation, in the rest of the thesis we assume SAS$^+$ variables to be binary without loss of generality.

of the characteristic function. Quantification of variables is another type of function transformation commonly encountered in symbolic search. The existential quantification of a variable $\upsilon \in \mathcal{V}$ with respect to a function f removes the dependency of f on variable υ such that $\exists \upsilon : f = f|_{\upsilon=\top} \vee f|_{\upsilon=\perp}$, where $f|_{\upsilon=\top}$ and $f|_{\upsilon=\perp}$ are the sets of assignments to variables $\mathcal{V} \setminus \{\upsilon\}$ so that they make f true whenever $\upsilon = \top$ and $\upsilon = \perp$, respectively. Thus, the result of the existential quantification corresponds to the projection of the set of states over the set of variables $\mathcal{V} \setminus \{\upsilon\}$. Similarly, the universal quantification of a variable $\upsilon \in \mathcal{V}$ with respect to a function f removes the dependency of f on variable υ such that $\forall \upsilon : f = f|_{\upsilon=\top} \wedge f|_{\upsilon=\perp}$. This means that when a variable $\upsilon \in \mathcal{V}$ is universally quantified away the resulting function f' is satisfied only for those cases in which f is true for every possible assignment of υ ($\upsilon = \top$ and $\upsilon = \perp$).

The set of operators $\mathcal{O}$ is represented with one or more *Transition Relations* (TRs). A TR is a function $T_c(x, x')$ that represents one or more operators with the same cost, c. TRs are defined using two sets of variables, the *source*-set and the *target*-set of variables, here represented as x and x', respectively. Both sets of variables have the same cardinality as the set of variables of the characteristic function, so that each variable in the *source*- and *target*-sets corresponds to a variable in the characteristic function: the variables of the *source*-set correspond to the preconditions of the operators of the TR and the variables of the *target*-set correspond to the effects. Given a set of states S_g and a TR T_i, the *image* operation is used to compute the set of successor states that can be reached from any state in S_g by applying any operator represented by the TR. The image corresponds to the operation $image(S_g, T_i) := (\exists x\, (S_g \wedge T_i))\, [x' \leftrightarrow x]$. The conjunction $S_g \wedge T_i$ corresponds to all pairs of states $\langle s, s' \rangle$ such that $s \in S_g$ and $s' = o(s)$ for an operator o represented by TR_i. Then, the existential quantification ignores predecessor states and the variable swapping, $[x' \leftrightarrow x]$, represents the resulting states s' with the standard set of variables x.

Similarly, the *pre-image* operation computes the set of predecessor states, i. e., states that can reach some state in S_g through some operator in the TR, $pre\text{-}image(S_g, T_i) = \exists x' \left((S_g(x)[x \leftrightarrow x']) \wedge T_i(x, x')\right)$. A detailed description of image computation and the representation of transition relations is given in Chapter 3.

2.2 Binary Decision Diagrams

Decision Diagrams are data structures inspired by the graphical representation of a logical function. Reduced Ordered Binary Decision Diagrams, ROBDDs for short (Bryant, 1986), are the most popular data structure to represent the logical formulæ employed in symbolic search. They consist of (a) Binary Decision Diagrams (i. e., each internal node has only two successors), (b) a fixed variable ordering on every path from the root to a sink, and (c) application of two reduction rules.

Definition 2.1 (Binary Decision Diagram)**.** *A BDD is a rooted labeled directed acyclic graph with two types of* terminal nodes *or* sinks*: $\top$ and $\perp$.* Non-terminal *or* inner *nodes are defined by a 3-tuple $p = \langle x_i, p_0, p_1 \rangle$, where x_i is a variable and p_0 and p_1 are either sink or inner nodes representing functions that do not depend on x_i. For a node $\langle x_i, p_0, p_1 \rangle$, p_0 corresponds to the case of assigning variable x_i the value false, thus giving the so-called* low *(or* 0*) successor; p_1 corresponds to the case of assigning variable x_i the value true, giving the* high *(or* 1*) successor. For any assignment of the variables on a path from the root to a sink, the represented function will be evaluated to the value labeling the sink.*

The tuple p that defines an inner node is in fact a function obtained from doing a Shannon expansion (Shannon, 1938) with respect to x_i. In particular, the function represented by p is $(x_i \wedge p_1) \vee (\overline{x}_i \wedge p_0)$.

Definition 2.2 (Ordered Binary Decision Diagram)**.** *An ordered binary decision diagram (OBDD) is a BDD with the additional requirement that variables on any path from the root to the sinks always appear in the same order.*

If variables are named according to the position in the variable ordering, for any node $p = \langle x_i, p_0, p_1 \rangle$, where $p_0 = \langle x_j, p'_0, p'_1 \rangle$ and $p_1 = \langle x_k, p''_0, p''_1 \rangle$ then $j > i$ and $k > i$.

Definition 2.3 (Reduced Ordered Binary Decision Diagram)**.** *A reduced ordered binary decision diagram (ROBDD) is an OBDD where the following two reduction rules are applied:*

1. *If both edges of a node p point to the same node, i. e., $p_0 = p_1$, remove the node p and redirect all the references to p to its child p_0.*
2. *Whenever two nodes p, p' are duplicate, that is, $p = \langle x_i, p_0, p_1 \rangle$ and $p' = \langle x_i, p_0, p_1 \rangle$, p and p' are merged into a single node.*

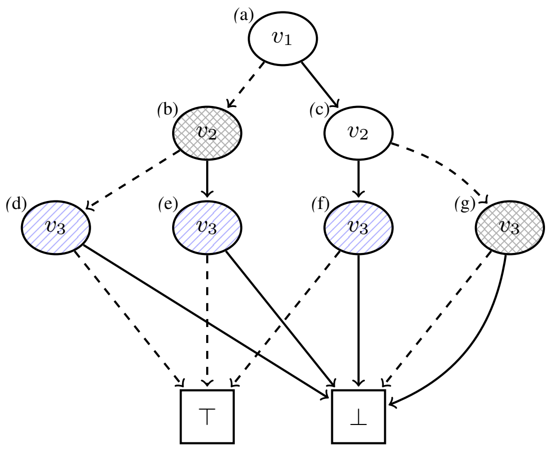
We say that an OBDD is partially reduced *if it satisfies the second reduction rule but not the first one, i. e., it contains nodes whose edges both point to the same node.*

Figure 2.1 shows an example of applying the reduction rules to an OBDD to obtain its reduced version. This is the typical graphical representation of BDDs that we will use in the examples throughout the thesis. The graphic representation uses squares and ellipses to depict terminal and inner nodes, respectively. Inner nodes in the same row are labeled with the same variable v_i, according to the variable ordering $v_1, \ldots, v_n$. Each inner node has two outgoing edges: a dashed edge to the low successor ($v_i = 0$) and a solid edge to the high successor ($v_i = 1$). In this case, we have named nodes with letters to refer to them unequivocally.

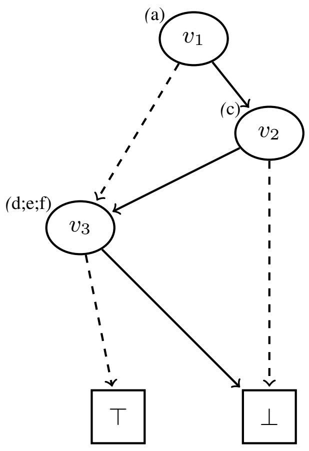
The OBDD in Figure 2.1a violates both reduction rules and gets reduced to the ROBDD depicted in Figure 2.1b after applying the reduction rules. On the one hand, rule (ii) is violated by nodes *(d)*, *(e)*, and *(f)*, since they are equivalent, i. e., their edges point to the same nodes. Therefore, they are substituted by a single node in the reduced version, *(d;e;f)*. On the other hand, rule (i) is violated by nodes *(g)* and the replacement of *(b)* after applying rule (ii), since they are redundant, i. e., their two edges point to the same node. This is obvious in the case of *(g)*, which points to the node 0. In the case of *(b)*, while it initially points to two different nodes, *(d)* and *(e)*, they get reduced to *(d;e;f)*. Therefore, *(b)* and *(g)* are eliminated, redirecting all the arcs that point them to their only child. This illustrates that the BDD reduction rules must be applied iteratively until a fixpoint is reached. This can be done in time linear in the number of nodes in the BDD with a two-phase bucket sort algorithm (Sieling and Wegener, 1993).

ROBDDs are often also called BDDs for short. Given that we only consider ROBDDs in the remainder of this thesis we use BDD as a synonym of ROBDD. The definition of BDDs ensures their canonicity (Bryant, 1986): for any Boolean function f and variable ordering, there exists a unique BDD representing f. The application of reduction rules provides up to exponential memory gains in the number of variables when representing some functions with respect to the nodes needed by a non-reduced BDD. This exponential gap between a reduced and a non-reduced BDD is due to the elimination of duplicate nodes, since a BDD may have an exponential number of different paths to the same node — so that the non-reduced BDD has an exponential number of duplicate nodes in the number of variables. The first reduction rule, on the other hand, can only provide up to linear memory gain. Therefore, partially reduced BDDs have an exponential gain in the same cases as BDDs.

Moreover, tight bounds can be proven for the complexity of BDD operations as long as all the involved BDDs share the same variable ordering (Bryant, 1986). Equivalence comparison and negation are performed in constant time. The *apply* operation of two BDDs $f \circ g$, used to compute



(a) Non-reduced OBDD



(b) ROBDD

Figure 2.1: Example of BDD reduction rules for the function $(\neg v_1 \vee (v_1 \wedge v_2)) \wedge \neg v_3$. The non-reduced BDD shown in the left part has three equivalent nodes and two redundant nodes. The right part shows the ROBDD representation of the same function.

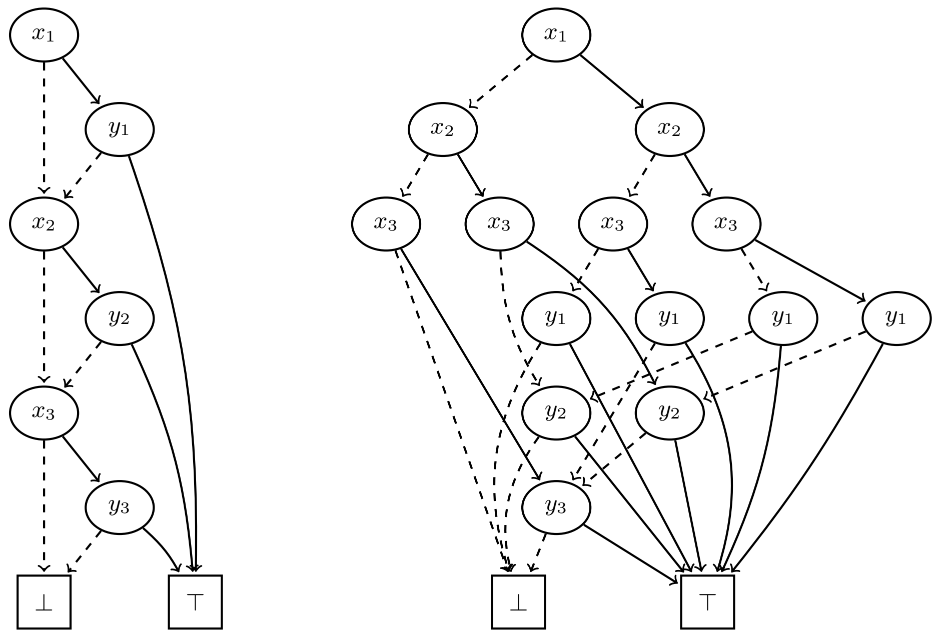
the disjunction, conjunction and other binary operations over two Boolean functions, has quadratic space and time complexity on the size of the BDDs, $|f| \times |g|$ — the conjecture that the time complexity is linear on $|f| + |g| + |f \circ g|$ was recently disproved (Yoshinaka et al., 2012; Bollig, 2014). Unfortunately, not all operations involved in symbolic search are polynomial. Disjunction or conjunction of several BDDs has exponential complexity on the number of BDDs. Finally, existential and universal quantification have exponential complexity on the number of quantified variables.

2.2.1 BDD Variable Ordering

The size of a BDD that describes a given function is fully determined by the variable ordering. For certain kinds of functions there is an exponential gap between the complexity of describing them with one variable ordering or another (Bryant, 1986; Bryant, 1992). Figure 2.2 shows the BDD representation of a function over $2n$ variables that has polynomial size for the $x_1, y_1, \dots, x_n, y_n$ ordering and exponential size for the $x_1, \dots, x_n, y_1, \dots, y_n$ ordering. Therefore, choosing a suitable variable ordering is fundamental for efficient symbolic solvers. However, finding an optimal variable ordering that minimizes the size of a BDD has been proven to be CO-NP-COMPLETE (Bryant, 1986). Good approximations of the optimal variable ordering are not easy to obtain, either. Bollig and Wegener proved that finding an ordering so that the resulting BDD contains at most s nodes (with s being specified beforehand) is NP-COMPLETE (Bollig and Wegener, 1996). Sieling has proved that, unless P $=$ NP, there is no polynomial time algorithm that can calculate a variable ordering so that the resulting BDD contains at most c times as many nodes as the one with optimal variable ordering for any constant $c > 1$ (Sieling, 2002).

In the literature, two approaches have been considered to select variable orderings that make BDDs as succinct as possible within the available time and memory constraints:

- **Variable ordering optimization:** Variable ordering optimization algorithms take as input a function (usually represented as a BDD with an arbitrary variable ordering) and determine a



(a) Polynomial size: $x_1, y_1, x_2, y_2, x_3, y_3$ (b) Exponential size: $x_1, x_2, x_3, y_1, y_2, y_3$

Figure 2.2: Example of BDD variable orderings that cause the representation of a function $f(x_1, \ldots, x_n, y_1, \ldots, y_n) = (x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n)$ ($n = 3$ in our example) to be exponential or polynomial in n (Kissmann, 2012).

suitable variable ordering that reduces the BDD size. They are usually based on a traversal of the space of possible variable orderings.

- **Domain analysis for variable ordering:** Planning problems have an inherent structure that can be exploited to determine a good variable ordering. Based on the observation that related variables should usually be close in the variable ordering (as $\langle x_k, y_k \rangle$ in the example of Figure 2.2), identifying relationships between the variables may define a good criterion to optimize the variable ordering.

Also, one might consider a *static* variable ordering that is fixed throughout the whole search process, or a *dynamic* variable ordering that may change at some point. Since BDDs describing different layers of the search may benefit from different variable orderings, dynamic variable ordering approaches have the potential to outperform static variable ordering strategies. Indeed, it has been empirically demonstrated that state-of-the-art strategies for deciding static orderings are better than random orderings but they are not competitive with optimized variable orderings for each layer (Kissmann and Hoffmann, 2013; Kissmann and Hoffmann, 2014).

Nevertheless, we follow GAMER's strategy to select a static variable ordering through domain analysis (Kissmann and Edelkamp, 2011). GAMER takes into account the variable dependencies captured by the causal graph as a criterion to determine the variable ordering (Kissmann and Edelkamp, 2011). The variable ordering optimization is seen as a minimization of the quadratic distance of the variables with dependencies, which is optimized through a randomized algorithm. GAMER uses a local search optimization procedure that tries to place related variables as close as possible. Variables

v_i and v_j are related if there exists an operator $o \in \mathcal{O}$ so that $v_i \in \mathcal{V}_{pre(o)} \cup \mathcal{V}_{eff(o)}$ and $v_j \in \mathcal{V}_{eff(o)}$ or *vice versa*. Recent research has shown empirically that GAMER's ordering is among the best static orderings and is significantly better than a random ordering though still arbitrarily far from optimal (Kissmann and Hoffmann, 2013).

An alternative to GAMER's static variable ordering could be the usage of dynamic variable reordering. This approach was recently explored in symbolic search planning by DYNAMIC-GAMER (Kissmann et al., 2014). Empirical results show that DYNAMIC-GAMER outperforms the basic GAMER, showing the benefits of dynamic variable reordering.

2.2.2 Limits and Possibilities of the BDD Representation

Large benefits can be obtained when sets with exponentially many states are represented with polynomially sized BDDs. Thus, efficiency of BDD-based planning greatly depends on the size of BDDs describing "interesting" sets of states. In the case of symbolic search planning some sets of interest are the set of goal states, the set of all reachable states (from the initial state or the goal states) and subsets of reachable states satisfying additional properties, such as the set of all states reachable with a minimum cost exactly equal to g.

Theoretical analyses on the complexity of representing those states in some planning domains and games have proved them to have polynomial upper bounds and exponential lower bounds for the number of nodes needed to represent different relevant sets of states (Hung, 1997; Edelkamp and Kissmann, 2008a; Edelkamp and Kissmann, 2011). Examples of exponential lower bounds are the set of all reachable states in BLOCKSWORLD or in sliding tile puzzles like N-PUZZLE, or the set of goal states in CONNECT-4 or TIC-TAC-TOE. Examples of polynomial upper bounds are the set of all reachable states in CONNECT-4 ignoring terminal states, TIC-TAC-TOE and the goal states in domains that have a compact SAS$^+$ representation such as BLOCKSWORLD or sliding tile puzzles. Remarkably, a polynomial lower bound has been proven for all the sets of states reachable with arbitrary g in the GRIPPER domain, allowing symbolic planners to solve instances of GRIPPER in polynomial time and memory.

2.2.3 Algebraic Decision Diagrams

Algebraic Decision Diagrams (ADDs) (Bahar et al., 1997) are an extension of BDDs to represent functions with a different domain than $\{\top, \bot\}$. They are typically used to represent numerical functions $f : S \rightarrow \mathbb{N}$. The definition of an ADD resembles the definition of ROBDDs, but using an arbitrary number of terminal nodes with different labels.

As discussed in Section 2.1, heuristic functions may be represented directly as numerical functions by using ADDs or as a vector of BDDs, one per heuristic value. The former is best for explicit-state search, since one single look-up into the ADD suffices to retrieve the heuristic value of a state. The latter, however, is considered more appropriate for symbolic search because it eases the retrieval of the subset of states having a particular heuristic value. Both representations are closely related, which is not very surprising given the resemblance of the definition of BDDs and ADDs. Proposition 2.1 ensures that we can always transform an ADD representation to a sequence of BDDs in polynomial time and with at most a polynomial overhead in memory with respect to the ADD size.

Proposition 2.1. *Given an ADD A_f representing a function $f : \mathcal{S} \rightarrow \{1, \ldots, K\}$ it can be transformed to a sequence of BDDs, $B_1, \ldots, B_K$, each of size $|B_i| \leq |A_f|$ in time $\mathrm{O}(K|A_f|)$.*

Proof. Each BDD B_i has at most $|A_f|$ nodes. Consider the OBDD B_i' that results from substituting terminal nodes of A_f: i by $\top$ and the rest by $\bot$. Then, B_i' is a non-reduced BDD with the same number of nodes as A_f. Applying reduction rules can only decrease nodes so that the reduced BDD has at most as many nodes as A_f, $|B_i| \leq |B'_i| = |A_f|$.

Regarding the time complexity, it is obtained from the complexity of BDD reduction, which can be done in time $O(|B'_i|)$ (Sieling and Wegener, 1993). □

2.2.4 Alternative Representation Schemes

Though in the context of this thesis we consider the symbolic representation in the form of BDDs and ADDs, there are several alternatives in the literature to represent Boolean or integer functions. Here we analyze those alternatives, discussing their advantages and disadvantages to justify the selection of BDDs and ADDs.

Boolean-function representation is a well-studied topic in the field of Knowledge Representation. There are a good number of alternative compilation languages and their relation has been thoroughly studied (Darwiche and Marquis, 2002). Relevant aspects to compare different compilation languages are succinctness, polytime queries and polytime transformations. If a language is more succinct, it will represent the same function in less memory. Queries obtain information without changing the represented function and transformations include operations like conjunction, disjunction, etc. In the context of symbolic search planning, our algorithms require a representation of the sets of states as succinct as possible, as well as efficient apply/quantification transformations.

- **Zero-suppressed Decision Diagram (ZDD)** (Minato, 1993) is a variant of BDDs designed to represent subsets, so that the default value should be false. ZDDs follow the same rules as BDDs except for the reduction rules. ZDDs do not eliminate nodes with the same left and right children, but instead they eliminate all the nodes whose 1-edge points to $\perp$. They have similar properties as BDDs and, though most literature has focused on developing algorithms for BDDs, most of them are also applicable to ZDDs. The main difference is the meaning of default variables, i. e., those that do not appear in the decision diagram. While in BDDs missing variables are irrelevant, i. e., the function result does not depend on the variable, in ZDDs they are assumed to be false (the function is automatically false if the variable is true). Thus, the ZDD reduction rules resemble the closed-world assumption made in STRIPS propositional planning, where it is assumed that any fact that has not been mentioned does not hold in the state. However, since variables missing from partial states (used to define the operator effects or the goals of the problem) are assumed to take any possible value, the BDDs reduction rule seem to be more appropriate in this case.

- **Deterministic Decomposable Negation Normal Form (d-DNNF)** (Darwiche, 2001) is a generalization of BDDs, characterized by requiring determinism and decomposability in a Negation Normal Form (NNF). An NNF is a directed acyclic and/or graph in which the leaves are associated to a literal or its negation. It is deterministic if the siblings of all *or* nodes represent logically contradictory functions. It is decomposable if the siblings of *and* nodes do not share variables. Being more general than BDDs, d-DNNF can represent functions in less space. However, it is not known if they have a polytime equivalence checking and sentential entailment and operations such as disjunction or conjunction cannot be performed in polynomial time unless $P = NP$ (Darwiche and Marquis, 2002). Though this has not been an impediment to use d-DNNF in conformant planning (Palacios Verdes, 2009) or to represent h^+ in planning with penalties and rewards (Bonet and Geffner, 2008), symbolic search algorithms have traditionally preferred the use of BDDs because they are considered suitable for the efficient manipulation of sets of states involved in the search.

- **Sentential Decision Diagrams (SDDs)** (Darwiche, 2011) are another generalization of BDDs which have been recently proposed. They branch on arbitrary sentences instead of variables. Thus, they can be characterized by variable trees instead of variable orderings. Tighter upper bounds are known for the size of SDDs than for BDDs and they retain important properties such as canonicity and polytime operations. Moreover, it has been shown that for certain kinds of functions and variable orderings in which the BDD representation has exponential size, there exists a dissection of that variable ordering so that the resulting SDD has polynomial size (Xue et al., 2012). It remains unproven if there are functions polynomially representable with an SDD for which the size of any BDD (with every possible variable ordering) is exponential. Thus, SDDs are a very promising option to substitute BDDs, though they are not as mature and developed as BDDs.

For the representation of numerical functions there are also some alternatives to ADDs, like Affine Algebraic Decision Diagrams (AADDs) (Sanner and McAllester, 2005) or Normalized Algebraic Decision Diagrams (NADDs) (Ossowski and Baier, 2006). However, ADDs are the alternative closest to BDDs and search is usually performed through sets of states.

In summary, there are solid alternatives to BDDs in the literature that are able to represent the same functions using less memory. However, BDDs have certain properties (uniqueness, efficient apply operation) that make them more suitable to symbolic search. Therefore, in the rest of the thesis, BDDs and ADDs will be assumed whenever we refer to a symbolic representation.

2.3 Basics of Symbolic Search

Symbolic search planning performs search in state spaces by representing sets of states as BDDs and operating with them. In this thesis, we consider the symbolic version of different heuristic search algorithms, such as unidirectional or bidirectional uniform-cost search and A* search. Before describing in detail the symbolic version of these algorithms, we set some basics that are common to all of them.

In symbolic search, as in heuristic search (see Section 1.2), algorithms use two lists of states: the *open* and the *closed* list. The open list, $open$, stores the states reached from the initial state that have not been expanded yet. The algorithms generate states and insert them into the open list. Expanded states are removed from $open$ and inserted into $closed$.

In symbolic search, both lists are represented as BDDs. Optimal algorithms need to consider the cost in which the states were reached or expanded. Therefore, $open$ is a list of BDDs where $open_i$ represents the sets of states reached with $g = i$ that have not been expanded yet. Similarly, $closed$ is a list of BDDs where $closed_i$ represents the set of states that were expanded with $g = i$. Additionally, the closed BDD, $closed_*$, represents all states that have already been expanded, i. e., $closed_* = \bigvee_i closed_i$.

Next, two auxiliary procedures commonly needed for symbolic search algorithms are presented: breadth-first search and the solution reconstruction procedure.

2.3.1 Reachability Analysis with Breadth-First Search

Symbolic search benefits of expanding sets of states with the same g-value at once. In domains with 0-cost actions, in order to expand all states with that g-value, all the states reachable with 0-cost actions must be collected before expanding the set of states. This reachability analysis can be performed with breadth-first search, which computes all the reachable states from a given set of

Algorithm 2.1: Symbolic BFS

```
   Input: Initial set of states: $S$
   Input: Transition relations: $\mathcal{T}$
   Input: States to prune: $prune$
   Input: Goal states: $S_\star$
   Output: Set of all states reachable from $S$ with $\mathcal{T}$ pruning $prune$
1  $S \leftarrow S \wedge \neg prune$
2  $closed \leftarrow S$
3  while $S \neq \bot$ and $S \wedge S_\star = \bot$ do
4      $S \leftarrow \left(\bigvee_{T_i \in \mathcal{T}} image(S, T_i)\right) \wedge \neg prune \wedge \neg closed$
5      $closed \leftarrow closed \vee S$
6  return $closed$
```

states, S. Algorithm 2.1 performs a Symbolic Breadth First Search (SBFS) from S using a set of transition relations $\mathcal{T}$. Additionally, the algorithm takes as input a set of states, $prune$, to prune the search and a set of goal states, $S_\star$, to stop the search as soon as a goal state is found. Applied with the whole set of transition relations $\mathcal{T}$, and with empty $closed$ and $S_\star$ sets, SBFS will return the set of all reachable states with smaller or equal distance from the initial state than the nearest goal state. However, breadth-first search does not guarantee to obtain optimal-cost solutions if costs are not uniform, so we only use it as an auxiliary procedure. This algorithm is used with the subset of 0-cost transitions, $\mathcal{T}_0$ by the other search algorithms presented below, pruning states that have been already closed in other layers of the search.

The algorithm starts by removing states in $prune$ from the initial states and inserting them into closed (lines 1 and 2). Then, at each iteration, the current set of states is expanded removing duplicates and states in $prune$ and inserting the result in the closed set (line 4). Finally, when the current set of states, S, is empty or contains a goal state (line 3), the algorithm returns all the set of states that have been reached (line 6).

2.3.2 Solution Reconstruction

Solution reconstruction works differently than in explicit-state search, because symbolic search algorithms do not keep track of the parents of generated states. The path from the initial state to a given state is reconstructed using the closed list that contains all the expanded states classified by their g-value. Algorithm 2.2 computes an optimal plan from the initial state, s_0, to a target state arbitrarily selected from the set of target states, S_{target}, whose g^*-value is known. The algorithm is a simplified implementation of a backward A* search algorithm with the perfect heuristic g^*, which is stored in the closed list $closed$. In the absence of 0-cost operators, the algorithm is guaranteed to run in time linear in the plan length, the number of operators, and the number of variables, $\mathrm{O}\left(|\pi||\mathcal{O}||\mathcal{V}|\right)$.

To simplify the exposition, Algorithm 2.2 omits the details needed to handle 0-cost actions. In such domains, all the algorithms perform a BFS with 0-cost actions as described by Algorithm 2.1. In order to reconstruct the solution, the layers in the BFS are stored separately for their use in the solution reconstruction. Then, the solution reconstruction algorithm uses a separated counter to track in which 0-cost layer a state appears the first time. The loop over all operators in line 4 of Algorithm 2.2 does not take into account 0-cost actions and is only performed for states in the first 0-cost layer. For other states, a similar loop is performed only with 0-cost actions, looking for a predecessor state in the previous 0-cost layer.

Algorithm 2.2: ConstructSolution.

```
Input: Source and target states: $s_0$
Input: Set of target states: $S_{target}$
Input: $g^*$-value of target states: $g_{target}$
Input: Operators: $\mathcal{O}$
Input: Closed list: $closed$
Output: Cost-optimal plan: $\pi = [a_1, \ldots, a_n]$
1  $s_{target} \leftarrow$ SelectArbitraryStateFrom($S_{target}$)
2  $\pi \leftarrow [\,]$
3  while $s_{target} \neq s_0$ do
4      foreach $o \in \mathcal{O}$ do
5          $s \leftarrow s$ s.t. $o(s) = s_{target}$
6          $g \leftarrow g_{target} - c(o)$
7          if $s \in closed_g$ then
8              $s_{target} \leftarrow s$
9              $g_{target} \leftarrow g$
10             $\pi \leftarrow o \,\|\, \pi$
11             break
12 return $\pi$
```

2.4 Symbolic Uniform-Cost Search

Uniform-cost search, also known as Dijkstra's algorithm, is a search algorithm to find shortest paths in graphs (Dijkstra, 1959). Uniform-cost search always expands the states with lowest g-value, so that the best path found to any expanded state is guaranteed to be optimal. Thus, when the algorithm selects a goal state for expansion, an optimal solution plan has been found. Uniform-cost search may be used both in progression or in regression. The main difference is that forward search (performing progression) starts at the initial state, s_0, and advances towards the goal states, $S_\star$, while backward search (performing regression) starts at $S_\star$ and advances towards s_0 by means of the inverted operators.

Algorithm 2.3 details a symbolic version of uniform-cost search for the case of forward search. In addition to the planning task, the algorithm takes as input the set of transition relations, $\mathcal{T}$, which is a set of BDDs T_c for each action cost c.

Starting at the initial state, the algorithm iterates over all values that g may take during the search. If at some point the open list is empty (i. e., it does not contain any state), it means that the entire state space has been expanded without ever reaching a goal state, so that the algorithm has proven that the problem is unsolvable (line 12).

At each step, symbolic uniform-cost search expands the set of states with minimum g-value, g_{min} (line 4). Before fully expanding the current set of states, $open_{g_{min}}$, first of all a breadth-first search is performed using only the zero-cost operators, in T_0. The BFS algorithm removes duplicates (pruning all states that were already expanded in previous iterations, $closed_*$) and returns all states that have the same distance, g_{min}, from the initial state (line 5). Then, newly generated states are inserted into $closed_{g_{min}}$ (line 6). If at this point a goal state is contained in the set $open_{g_{min}}$, the algorithm reconstructs a solution plan and returns it (line 7). Otherwise, $open_{g_{min}}$ is expanded. For every action cost, c, it applies the image with T_c, the TR representing the operators with cost c, and

Algorithm 2.3: Symbolic Uniform-Cost Search.

```
    Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
    Input: Transition relations: $\mathcal{T}$
    Output: Cost-optimal plan or "no plan"
 1  $open_0 \leftarrow \{s_0\}$
 2  $closed \leftarrow \emptyset$
 3  while $open$ is not empty do
 4      $g_{min} \leftarrow \min\{g : open_g \neq \bot\}$
 5      $open_{g_{min}} \leftarrow$ BFS($open_{g_{min}}, T_0, closed_*, S_\star$)
 6      $closed_{g_{min}} \leftarrow open_{g_{min}}$
 7      if $open_{g_{min}} \wedge S_\star \neq \bot$ then
 8          return ConstructSolution($s_0, open_{g_{min}} \wedge S_\star, g_{min}, \mathcal{O}, closed$)
 9      for all $T_c \in \mathcal{T}, c > 0$ do
10          $open_{g_{min}+c} \leftarrow open_{g_{min}+c} \vee image\left(open_{g_{min}}, T_c\right)$
11      $open_{g_{min}} \leftarrow \bot$
12  return "no plan"
```

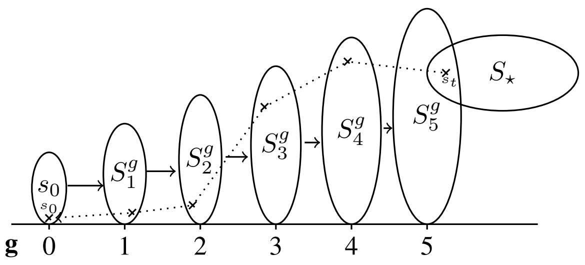


Figure 2.3: Uniform-cost search and solution reconstruction example. Sets of states involved in the search are drawn as ellipses. The crosses represent the solution reconstruction path from the target goal state s_t to the initial state s_0.

adding the result to the corresponding successor bucket $open_{g_{min}+c}$ (line 10). Finally, the expanded states are removed from $open_{g_{min}}$ (line 11).

Figure 2.3 depicts a diagram representing uniform-cost search. Throughout the thesis, we will use similar diagrams to illustrate examples of how different algorithms work. These examples provide a schematic view of the sets of states involved in the search, both in the open and closed lists. BDDs that represent sets of states are drawn as ellipses over an axis that represents the g-values of the search. The leftmost BDD corresponds to s_0. The successor generation is represented through arrows from the predecessor state sets to the state sets where successors are inserted. To keep the examples as simple as possible, we will usually assume unit-cost domains, so that each set of states only inserts states on the next g-bucket. In this example, we scale the ellipses to indicate that the set of states and the BDDs representing them are usually larger when g increases. As soon as the intersection between a set of states and the set of goal states is not empty, the solution reconstruction procedure may be initiated with any of the goal states found.

2.5 Symbolic Bidirectional Uniform-Cost Search

Bidirectional uniform-cost search performs two searches in the two possible directions in an interleaved manner, i. e., at each step, it automatically decides whether to continue the backward or forward search, taking advantage of the direction in which it performs best in each domain. Newly generated states are compared with the set of expanded states from the other search direction. In case of a match, a solution plan has been found, though it is not necessarily optimal. Rather, it is necessary to continue until the last plan is proven to be optimal (Nicholson, 1966; Torralba et al., 2013a).

The symbolic version of bidirectional uniform-cost search is detailed in Algorithm 2.4. In large parts its working is the same as that of standard uniform-cost search, since at each iteration the *Step* procedure expands a set of states in a similar way in either the forward or backward direction. The *Step* procedure, as any step in uniform-cost search, takes the set of states with minimum g and calls the BFS procedure to remove duplicate states in $closed_*$ and get all states reachable with 0-cost actions (line 14). Then, all the states with a g-value of g_{min} are inserted into the closed list (line 15) and expanded, generating the successor states with the image operation (line 19) and inserting them into $open$ (line 21).

The *UpdatePlan* procedure checks whether a new better plan has been found over an expanded state (line 16) or a newly generated state (line 20). The check for generated states is not strictly needed, but it is an optimization that allows the algorithm to decrease the cost of the best plan found so far, w_{total}, as soon as possible. This reduces the number of image computations by skipping those that cannot possibly lead to a better plan (line 18). Also, after checking whether a plan exists, states in $closed_*$' can be removed from $Succ$ since they have already been expanded in the opposite direction so that their optimal distance to the goal is known. Since the check ignores states in the $open$ list of the opposite frontier, *UpdatePlan* must be called again when states are expanded in order to ensure that no plan is missed.

To check if a new plan has been found, *UpdatePlan* compares the newly expanded/generated states with the closed list of the opposite search. In case of a match, a new plan has been found. The cost of the new plan is the sum of the g-values of the intersected states in both searches (line 31). If this cost is smaller than the smallest cost found so far (w_{total}), then w_{total} is updated and the corresponding solution path π can be created. The plan can be retrieved with two calls to the *ConstructSolution* algorithm that finds the forward path from s_0 to one state in the intersection of both frontiers (line 32) and the backward path, using the reversed operators, from $s_\star$ to the state resulting of applying π_f to the initial state, $\pi_f(s_0)$ (line 33). The plan is the concatenation of the forward path and the reversed backward path (line 34).

The algorithm may stop when the sum of the minimum g-values of generated states for the forward and backward searches is at least w_{total}, the total of the cheapest solution path found so far. In previous work, it was shown that this stopping condition guarantees that an optimal solution has been found (Nicholson, 1966; Goldberg and Werneck, 2005). Since the g-value for each search is monotonically increasing in time, so is their sum. After the condition is met, every state s removed from a priority queue will be such that the costs of the solution paths from s_0 to s and from s to $s_\star$ will be at least w_{total}, which implies that no solution path of cost less than w_{total} exists.

At each step, the algorithm may decide whether to perform a forward or backward step (line 8). A typical criterion to decide the direction of the search is Ira Pohl's cardinality principle that picks at each step the direction with fewer frontier states (Pohl, 1969). In symbolic search, however, expanding fewer states does not necessarily imply that the search is easier. Therefore, in the same spirit as the cardinality principle, GAMER bases the decision on which direction is more promising, i. e., easier to expand. The algorithm estimates the time needed for the following step in each direction

Algorithm 2.4: Symbolic Bidirectional Uniform-Cost Search

```
Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
Input: Transition relations: $\mathcal{T}$
Output: Cost-optimal plan or "no plan"
1  $fOpen_0 \leftarrow \{s_0\}$
2  $bOpen_0 \leftarrow S_\star$
3  $fClosed \leftarrow bClosed \leftarrow \bot$
4  $g_f \leftarrow g_b \leftarrow 0$
5  $w_{total} \leftarrow \infty$
6  $\pi \leftarrow$ "no plan"
7  while $g_f + g_b < w_{total}$ and not $fOpen$ is empty and not $bOpen$ is empty do
8      if NextStepDirection($fOpen, bOpen$) = $Forward$ then
9          $fOpen, fClosed, g_f, \pi, w_{total} \leftarrow$ Step ($fOpen, fClosed, g_f, bClosed, \mathcal{T}, \pi, w_{total}$)
10     else
11         $bOpen, bClosed, g_b, \pi, w_{total} \leftarrow$ Step ($bOpen, bClosed, g_b, fClosed, \mathcal{T}^{-1}, \pi, w_{total}$)
12 return $\pi$

13 Procedure Step ($open, closed, g_{min}, closed', \mathcal{T}, \pi, w_{total}$)
14     $open_{g_{min}} \leftarrow$ BFS ($open_{g_{min}}, T_0, closed_*, \emptyset$)
15     $closed_{g_{min}} \leftarrow open_{g_{min}}$
16     $\pi, w_{total} \leftarrow$ UpdatePlan ($\pi, w_{total}, open_{g_{min}}, g_{min}, closed'$)
17     for all $T_c \in \mathcal{T}, c > 0$ do
18         if $g_{min} + c < w_{total}$ then
19             $Succ \leftarrow image(open_{g_{min}}, T_c) \wedge \neg closed_*$
20             $\pi, w_{total} \leftarrow$ UpdatePlan ($\pi, w_{total}, Succ, g_{min} + c, closed'$)
21             $open_{g_{min}+c} \leftarrow open_{g_{min}+c} \vee (Succ \wedge \neg closed'_*)$
22     $open_{g_{min}} \leftarrow \bot$
23     $g_{min} \leftarrow \min\{g \mid open_g \neq \bot\}$
24     return $open, closed, g_{min}, \pi, w_{total}$

25 Procedure UpdatePlan ($\pi, w_{total}, S_g, g, closed'$)
26     if $S_g \wedge closed' \neq \bot$ then
27         for all $i \in \{0 \cdots \mid closed'_i \neq \bot\}$ do
28             if $g + i \geq w_{total}$ then
29                 break
30             if $S_g \wedge closed'_i \neq \bot$ then
31                 $w_{total} \leftarrow g + i$
32                 $\pi_f \leftarrow$ConstructSolution ($s_0, S_g \wedge closed'_i, \mathcal{O}, fClosed$)
33                 $\pi_b \leftarrow$ConstructSolution ($s_\star, \pi_f(s_0), \mathcal{O}^{-1}, bClosed$)
34                 $\pi \leftarrow \pi_f \parallel \pi_b^{-1}$
35     return $\pi, w_{total}$
```

as explained below and the direction with lower estimated time is the one selected for expansion. This greedy policy works well under the assumption that, as the search progresses, sets of states are usually harder to represent.

In this thesis, we slightly changed the way in which GAMER estimates the time needed to perform the next step. GAMER only considers the time taken by the previous step, choosing the direction that was faster in its last step. Also, in order to avoid an immediate failure in domains where backward search is unfeasible, GAMER does not allow the first backward step to take more than 30 seconds and later steps to take more than 2.5 times the slowest forward step. If any of these limits are exceeded, the backward search is disabled and the planner reverts to simple forward uniform-cost search (while still retaining the already generated backward layers). We slightly change the strategy of GAMER in two ways:

1. We consider interrupting any step in case it takes too much time. This makes the planner slightly more robust, because a bad decision, such as taking a step that takes too much time, does not necessarily halt the planner. The time needed to perform the interrupted step is re-estimated and it will be attempted again if the opposite frontier is harder.

2. We take into account the size of the current frontier in order to get more accurate estimations.

To compute the estimated time for step k, t_k, we take into account the time spent on the previous step, t_{k-1}, and the BDD sizes for the state set to be expanded and the state set expanded in the previous step, s_k and s_{k-1}, respectively. In general, we assume a linear relation between BDD size and image computation time. However, this might not work well at the beginning of the search, where t_k is too low and the BDD size ratio is too large. Thus, we estimate t_k as shown in Equation 2.1, using the time of the previous layer whenever it was below one second and the linear estimation for all the other cases.

$$t_k = \begin{cases} 0, & \text{if } k = 0 \\ t_{k-1}, & \text{if } k > 0 \wedge t_{k-1} \leq 1s \\ t_{k-1}\frac{s_k}{s_{k-1}}, & \text{if } k > 0 \wedge t_{k-1} > 1s \end{cases} \tag{2.1}$$

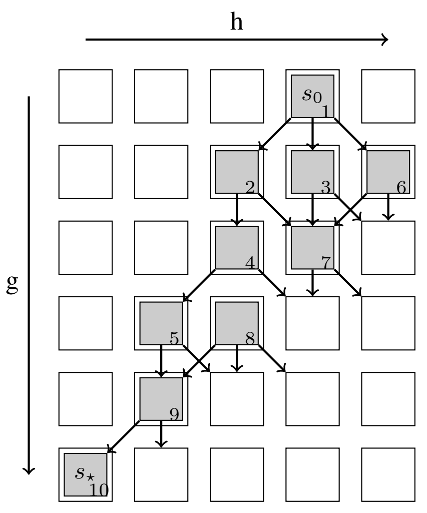
This estimation, though not perfect, is often accurate enough to determine the best direction. However, the time needed to complete a single step may grow exponentially in the frontier size, invalidating our assumption of a linear ratio. In most cases this is not a problem because the estimations are updated after every step, but sometimes it may lead to exhausting the available memory or time. For example, in some domains, the first backward step takes much longer than a forward step. Therefore, each step is given a maximum allotted time and is truncated when exceeding it. The allotted time to perform a step is the double of the time estimated for the opposite direction. When a step is truncated, its estimation must be updated to a higher value, to avoid selecting the same step indefinitely. Therefore, we assign as estimation the double of the time spent until failure.

2.6 Symbolic A* Search

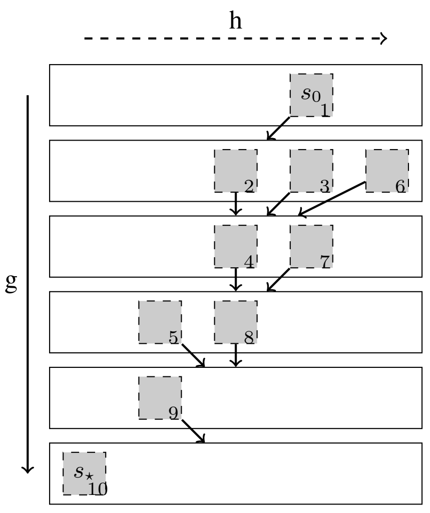
A* (Hart et al., 1968) is a best-first search algorithm. As such, it makes use of a priority queue and expands the node with the minimal value according to some criterion. While in uniform-cost search this criterion is only the g-value, A* makes use of a heuristic function h that estimates the distance to the goal. A* expands the node n with smallest $f(n) = g(n) + h(n)$ value. Thus, if the heuristic is admissible, when a goal state is expanded, all the states with $f < f^*$ (the cost of an optimal solution) have been expanded and the plan is guaranteed to be optimal. In this thesis, we focus on abstraction heuristics that are not only admissible but also consistent. Consistency is an important property because A* with a consistent heuristic does not re-expand any state and the g-value of any expanded state is equal to the real cost, g^*. As no state is re-expanded, the number of state expansions in A* is optimal (up to tie-breaking criteria as discussed below) among algorithms using the same heuristic without additional information (Hart et al., 1968; Dechter and Pearl, 1985).

A* is not a unique algorithm, but a family of algorithms because its criterion is not fully specified. Since there may be many nodes with minimum f-value, different versions of A* may be defined depending on the tie-breaking criterion used to select which node will be expanded next among those with minimum f-value. In explicit-state search, the tie-breaking criterion is to expand nodes with larger g-value (and therefore smaller h-value) first because they are estimated to be closer to the goal and this attempts to expand goal states as soon as possible, terminating the algorithm. In case that more than one state is best according to this criterion, tie-breaking is usually based on FIFO order.

Symbolic A* (alias BDDA*) was first introduced by Edelkamp and Reffel (1998) and integrated in the planning context by Edelkamp and Helmert (2001) in the model checking integrated planning system MIPS. As usual with A*, BDDA* expands states in ascending order of $f = g + h$. The difference is that BDDA* groups sets of states with the same g and h-value, called g, h-buckets, being able to expand all of them at once. The heuristic function in symbolic search is precomputed prior to the search and represented as a list of BDDs, $heur$, one per possible heuristic value. The heuristic evaluation is done with a conjunction: given a set of states S and the $heur_i$ BDD, $S \wedge heur_i$ corresponds to the subset of states that have a heuristic value equal to i. As the heuristic is assumed to be consistent, the f-value monotonically increases, since states with a given f-value are expanded before states with larger f and successor states will never have smaller f-values than their parents. We say that A* explores f-diagonals, where the term f-diagonal refers to all states with a given f-value, making reference to the matrix representation used by the algorithm (see Figure 2.4 as example).



(a) Matrix BDDA*



(b) Lazy BDDA*

Figure 2.4: Example of BDDA* implementations. The greyed cells are expanded in the order specified by the numbers. Arrows denote successor buckets.

As in explicit-state search, multiple tie-breaking criteria may be used to decide which g, h-bucket to expand next. However, in practice, buckets with minimum g-value are preferred, i. e., the opposite criterion to that used in explicit-state search. This expansion order may be detrimental on the last f-diagonal because all the states with $f = f^*$ are expanded. However, in symbolic search the drawback is compensated because it avoids the re-expansion of buckets. When buckets with lower g-value are expanded, new states may be inserted in buckets with larger values. Thus, when expanding the buckets in ascending order of g, once a g, h-bucket has been expanded, no new states will be inserted into it afterwards, i. e., the algorithm does not need to re-expand any bucket. Preferring to expand first buckets with maximum g-value, as in explicit-search, may regenerate an already expanded bucket up to an exponential number of times in the number of different g-values on the same f-diagonal, losing the advantages of symbolic search.

Multiple variants of BDDA* exist across the literature, varying the representation of state sets involved in the search. ADDA* (Hansen et al., 2002) is an alternative implementation with ADDs, while Set A* (Jensen et al., 2002) refines the partitioning in a matrix representation of g- and h-buckets in the open list. Next, we will describe the two implementations we consider in this thesis, which we call Matrix BDDA* and Lazy BDDA*. GAMER uses the Matrix BDDA* implementation. We propose a variant, Lazy BDDA*, that slightly defers the heuristic evaluation. Our new variant is relevant not only for efficiency reasons, but also because it will be helpful for techniques developed later in the thesis.

2.6.1 Symbolic A* Using a Matrix Representation

Algorithm 2.5 details the workings of BDDA*, using a matrix representation for the open list. $open$ is a (two-dimensional) matrix of BDDs, where the first dimension corresponds to the g-value and the second one to the h-value. The closed list, $closed$, stores the states that are already expanded as in the previous algorithms. Each element of this list is a BDD and corresponds to states sharing the same h-value.

The algorithm initializes $open$ with the initial state, s_0, inserted into the $0, h(s_0)$-bucket (line 1). Since the heuristic is represented as a list of BDDs, $heur$, $h(s_0)$ is computed by performing a conjunction of s_0 with each BDD $heur_i$ and returning the value for which the conjunction is not empty, $\{s_0\} \wedge heur_i \neq \perp$. The search iterates over f-diagonals, and along each diagonal in the order of increasing g-values (as illustrated in Figure 2.4). Thus, at each step we select the minimum f-value and then the minimum g-value possible for the selected f (lines 4 and 5). Of course the h-value is set to $f - g$ so that the equation $f = g + h$ holds.

In order to expand the current g, h-bucket, the algorithm performs a breadth-first search with the 0-cost actions to get all reachable states with the same g-value, $Succ$ (line 7). The BFS procedure uses $closed_*$ to remove all duplicate states that have already been expanded. This is admissible since the heuristic is consistent, so states are always closed with their optimal g-value and they do not need to be re-expanded. Then, the heuristic is applied to the successor states, $Succ$, and they are inserted in the open list (line 9). Again, we assume consistency of the heuristic so there is no need to check any h' smaller than h. The heuristic evaluation of the successor states can be interleaved with the BFS algorithm in order to avoid expanding states with h' greater than h but we omit the details in the pseudocode for the sake of clarity.

Next, the algorithm proceeds to expand the bucket $open_{g,h}$. When a bucket with h-value 0 is selected for expansion the algorithm checks whether it contains a goal state (the heuristic is admissible and, therefore, it is goal-aware). If it does, the algorithm has successfully found an optimal solution plan (line 12). Other than that, the actual expansion is similar to that used in Algorithm 2.3, with the exception that we have to determine the h-values of all the successor states. For this we

Algorithm 2.5: Symbolic A* with Matrix Representation.

```
Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
Input: Consistent heuristic function $h(s)$ represented as BDDs: $heur$
Output: Cost-optimal plan or “no plan’
1  $open_{0,h(s_0)} \leftarrow \{s_0\}$
2  $closed \leftarrow \bot$
3  while $open$ is not empty do
4      $f \leftarrow \min\{f' \mid \exists g', h' : f' = g' + h', open_{g',h'} \neq \bot\}$
5      $g \leftarrow \min\{g' \mid \exists h' : g' = f - h', open_{g',h'} \neq \bot\}$
6      $h \leftarrow f - g$
7      $Succ \leftarrow$ BFS($open_{g,h}, T_0, closed_*, S_\star$)
8      for all $heur_{h'} \in heur,\ h \leq h' < \infty$ do
9          $open_{g,h'} \leftarrow open_{g,h'} \vee (Succ \wedge heur_{h'})$
10     $closed_g \leftarrow closed_g \vee open_{g,h}$
11     if $h = 0$ and $open_{g,h} \wedge S_\star \neq \bot$ then
12         return ConstructSolution($s_0, open_{g,h} \wedge S_\star, \mathcal{O}, closed$)
13     for all $T_c \in \mathcal{T}, c > 0$ do
14         $Succ \leftarrow image(open_{g,h}, T_c)$
15         for all $heur_{h'} \in heur,\ h - c \leq h' < \infty$ do
16             $open_{g+c,h'} \leftarrow open_{g+c,h'} \vee (Succ \wedge heur_{h'})$
17 return “no plan”
```

use again a conjunction with the BDDs of $heur$ (line 16). All states returned by the conjunction have the corresponding h'-value and are thus inserted into the specified bucket in the matrix. Again, consistency of the heuristic is assumed so that heuristic values h' smaller than $h + c$ can be skipped.

In case the action costs are very diverse the matrix can become rather sparse. For such cases, Kissmann and Edelkamp proposed to use a sparse matrix representation that does not store the entire matrix but only the buckets that actually contain some states (Kissmann and Edelkamp, 2011).

2.6.2 Lazy BDDA*

The heuristic computation is in many cases one important bottleneck for the A* algorithm, especially for some domain-independent heuristics. One possible way to deal with this problem is to avoid evaluating all the states immediately upon generation. In explicit-state search, deferred evaluation (Richter and Helmert, 2009) is a known technique to optimize the number of heuristic evaluations by delaying the heuristic evaluation of nodes until they are expanded. Nodes in the open list are assigned the heuristic value of their parent. These are inadmissible estimates so they have been used in non-optimal planning, but they could easily be adapted for the optimal case by reducing each estimation by the cost of the transition from the parent to that node, like in pathmax propagation (Méro, 1984). Another approach that avoids evaluating all heuristic estimates is Lazy A* (Tolpin et al., 2013). Lazy A* combines different heuristics, some of which are more computationally expensive than others, in a lazy way. The expensive heuristic estimates are only computed whenever the computationally “cheap” estimates have not already pruned the states.

In order to alleviate the effort made in heuristic computation, we propose a variation of the Matrix

Algorithm 2.6: Lazy BDDA*.

```
Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
Input: Consistent heuristic function $h(s)$ represented as BDDs: $heur$
Output: Cost-optimal plan or "no plan"
1  $open_0 \leftarrow \{s_0\}$
2  $closed \leftarrow \bot$
3  $f \leftarrow h(s_0)$
4  $g \leftarrow -1$
5  while $open$ is not empty do
6      $curr = \bot$
7      while $curr = \bot$ do
8          if $\exists g' > g \mid open_{g'} \neq \bot, heur_{f-g'} \neq \bot$ then
9              $g \leftarrow \min\{g' > g \mid open_{g'} \neq \bot, heur_{f-g'} \neq \bot\}$
10         else
11             $f \leftarrow \min\{f' > f \mid \exists g' open_{g'} \neq \bot, \exists h' heur_{h'} \neq \bot, f' = g' + h'\}$
12             $g \leftarrow \min\{g' \mid open_{g'} \neq \bot, heur_{f-g'} \neq \bot\}$
13         $h \leftarrow f - g$
14         $curr \leftarrow open_g \wedge heur_h$
15     $curr \leftarrow$ BFS($curr, T_0, closed_*, S_\star$)
16     $open_g \leftarrow curr \wedge \neg heur_h$
17     $curr \leftarrow curr \wedge heur_h$
18     $closed_g \leftarrow closed_g \vee curr$
19     if $h = 0$ and $curr \wedge S_\star \neq \bot$ then
20         return ConstructSolution($s_0, curr \wedge S_\star, \mathcal{O}, closed$)
21     for all $T_c \in \mathcal{T}, c > 0$ do
22         $Succ \leftarrow image(curr, T_c)$
23         $open_{g+c} \leftarrow open_{g+c} \vee Succ$
24 return "no plan"
```

implementation of BDDA*. Lazy BDDA*, or List BDDA*, as it was originally called (Edelkamp et al., 2012), is a variation of BDDA* in which the open list buckets are only distinguished by their g-value. We say that it is lazy because instead of computing the heuristic value when states are generated before their insertion into the open list, it delays the heuristic computation until the states are going to be expanded, reminding of Lazy A* (Tolpin et al., 2013).

Algorithm 2.6 shows the pseudo-code of Lazy BDDA*. The open list, $open$, organizes the generated states in a list according to their g-value and computes the conjunction with the heuristic only on demand. Instead of classifying all the successors upon generation according to their h-value, Lazy BDDA* inserts all into the same bucket identified only by their g-value. At each iteration, the next bucket to be expanded, $curr$, is extracted from $open$. First, the algorithm seeks the next valid pair of f and g values, i. e., the bucket with minimum f and minimum g so that there is a bucket $open_g$ and a h-value satisfying $f = g + h$.

As in other algorithms, the BFS procedure removes duplicates in $closed_*$ from $curr$ and gets all states reachable from $curr$ by 0-cost actions. As explained for the matrix version of BDDA*, duplicate pruning with respect to all closed states, $closed_*$, is admissible when the heuristics are consistent. States without the current h-value should not be expanded yet so they are removed from $curr$ and inserted into $open_g$. The rest of the algorithm is similar to Matrix BDDA*, though states are not evaluated upon generation but just inserted into $open_{g+c}$ (line 23).

Lazy BDDA* defers heuristic evaluation until a set of states is going to be expanded (line 14). The reasoning behind this strategy is to defer the heuristic calculation by computing the conjunction of the sets of states with the heuristic only when it is needed for expansion in the currently traversed f diagonal. This has the potential to save time in the final f diagonals, as the h-values of the states that will never be expanded are irrelevant and thus will not be calculated in Lazy BDDA* (as opposed to the matrix-based implementation). Moreover, just like in the explicit-state Lazy A*, when computing the maximum of multiple heuristics, if a heuristic proves some states to have an f-value of f', larger than the current f-value, we may skip the computation of other heuristics over those states until exploring the f'-diagonal.

Finally, the representation of the $open$ list as g-buckets instead of the g, h-buckets used by BDDA* requires additional disjunctions. There are no theoretical guarantees about the size of BDDs representing those buckets, since the additional disjunctions could cause an exponential blow-up or an exponential gain with respect to the BDD size, depending on the structure of the state space being traversed.

2.7 Symbolic Pattern Databases

Abstraction heuristics use a mapping from the original state space to a smaller abstract state space and use the optimal solution cost in the abstract state space as an estimation for the original problem. Pattern databases are a type of abstraction heuristics that, in the planning literature, is usually associated with the projection of the planning task over a subset of variables. A PDB is characterized by memorizing an abstract state space, storing the shortest path distance from each abstract state to the set of abstract goal states (Culberson and Schaeffer, 1998; Edelkamp and Schrödl, 2012). A formal description of abstraction heuristics and pattern databases in particular can be found in Chapter 6. Here, we present the method used by GAMER to generate heuristics that we will use to evaluate our techniques with Symbolic A* search.

GAMER uses symbolic partial pattern databases. A partial pattern database (Anderson et al., 2007) is a PDB that is not fully computed, but rather its calculation is stopped after either a pre-defined distance to the nearest goal or a pre-defined time-out has been reached. If all abstract states with a distance of d have been generated, then all the other abstract states can safely be assigned a value equal to $d + 1$. *Symbolic PDBs* (Edelkamp, 2002) are PDBs that have been constructed symbolically, using symbolic backward uniform-cost search. In contrast to the posterior compression of the state set (Ball and Holte, 2008), the construction by Edelkamp (2005) works on the compressed representation, allowing larger databases to be constructed.

GAMER uses an automatic pattern selection procedure (Kissmann and Edelkamp, 2011) to generate a well-informed PDB, similar to the one proposed for explicit-search planners (Haslum et al., 2007). Though GAMER automatically decides whether to use abstraction or to perform regression on the original problem, we only consider the automatic pattern selection procedure, since partial PDBs on the original state space are similar to bidirectional uniform-cost search, with the disadvantage that the better search direction is not automatically chosen.

Chapter 3

Image Computation

As introduced in Section 2.1 (see page 17), the image operation is used to compute the set of reachable states in a single step from a given set of states with the same g-value, S_g, and a transition relation T_i. The image operation is often the most time-consuming process in symbolic search, so it is important to perform it as efficiently as possible.

In this chapter we study different approaches to perform the image computation. Image computation is closely related to the representation of transition relations, i. e., the BDD representation of planning operators. As baseline we take the state-of-the-art planner GAMER that uses a separate TR to represent each operator of the planning task.

We propose three different improvements over the previous image computation methods: splitting TRs of single operators into two different BDDs to avoid the use of auxiliary variables; using a decision tree to filter away TRs whose preconditions are not satisfied by at least one state in the expanded set; and aggregating several TRs minimizing the size of the intermediate BDDs generated when computing the union of successors.

The chapter is organized as follows. First, in Section 3.1 we explain how the image is computed in GAMER. Sections 3.2, 3.3, and 3.4 present three methods to compute the image that are the main contribution of this chapter. We empirically evaluate the performance of these methods in Section 3.5 and conclude with a summary of the chapter in Section 3.6.

3.1 Basics of Image Computation

In symbolic search planning, planning operators are described in the *transition relations* (TRs). As seen in Section 2.1, a TR is a relation between predecessor and successor states, i. e., it represents all the pairs of states $\langle s, s' \rangle$ such that $s' = o(s)$ for an operator o represented by the TR. Thus, if sets of states are described as logical functions over some set of variables x, TRs also use a set of auxiliary variables, x', to represent successor states.

Also, in planning both x and x' have a direct correspondence with the set of variables $\mathcal{V}$: $|x| = |x'| = |\mathcal{V}|$ and each v_i has an associated x_i and x'_i. The variable ordering is usually optimized to efficiently represent sets of states, i. e., it orders variables x that represent sets of states as explained in Section 2.2.1 on page 20. To represent the TRs, we also must take into account the auxiliary variables from x'. Arguably, x_i and x'_i are closely related, since for any operator o with $v_i \notin \mathcal{V}_{\mathit{eff}(o)}$, x'_i is assigned the value of x_i. As the auxiliary variables only appear in the TRs, the most logical order is alternating the variables from x and x' $(x_1, x'_1, x_2, x'_2, \dots x_n, x'_n)$ as proposed, e. g., by

Burch et al. (1994).

The *image* operation of a set of states with respect to a given transition relation can be decomposed into basic logical operations over BDDs as shown by Equation 3.1.

$$image(S_g, T_i) := \exists x\, ((S_g \wedge T_i))\, [x' \leftrightarrow x] \tag{3.1}$$

Thus, the operation is carried out in three steps:

1. $I_1 := S_g \wedge T_i$: The conjunction filters preconditions on x, removing all states in which the precondition does not hold, and applies effects on x'. The result is a BDD storing all the pairs of predecessor and successor states.

2. $I_2 := \exists x\ I_1$: The existential quantification of the predecessor variables x removes the variables that belong to the predecessor states. The result is the set of successor states.

3. $image(S_g, T_i) := I_2[x' \leftrightarrow x]$: denotes the swap of the two sets of variables, setting the value of the successor states in terms of the variables in x.

In practice, steps 1 and 2 (conjunction and existential quantification) are performed in a single BDD operation, the so-called relational product, which is often implemented in a more efficient way than the sequential application of conjunction and quantification (Burch et al., 1994).

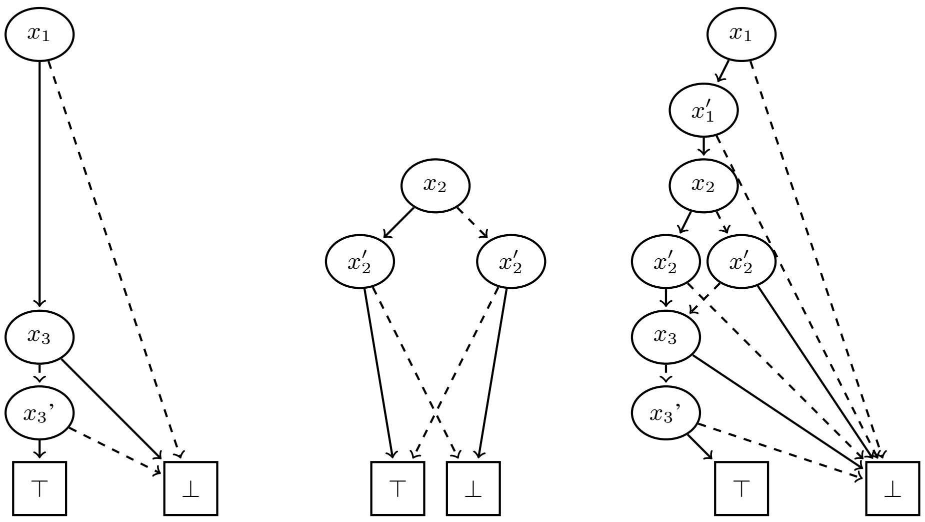
The related *pre-image* operation, used to perform regression, is decomposed into similar operations. In this case, variables are swapped first and the existential quantification is performed over x' instead of x. The formal definition of the pre-image operator is $pre\text{-}image(S_g, T_i) = \exists x'\, ((S_g[x \leftrightarrow x']) \wedge T_i)$. Even though we are focusing our discussion in this chapter on image computation, the same conclusions can be obtained about pre-image computation for all purposes.

Efficient image computation is key to perform a symbolic reachability analysis, useful for both symbolic search planning and symbolic model checking. A major challenge is the representation of the TRs, as having a single TR per group of operators that have the same cost c (a so-called monolithic TR) is often unfeasible because, in the worst case, a TR uses exponential memory in the number of operators that it represents. A traditional solution is to use a conjunctive or disjunctive partitioning of the transition relations (Burch et al., 1991b). This way, a list of TRs $\{T_{c,0}, \ldots, T_{c,n}\}$ whose images (or pre-images in regression) are combined by conjunction ($S_{g+c} = \bigwedge_{i=0}^{n} image(S_g, T_{c,i})$) or disjunction ($S_{g+c} = \bigvee_{i=0}^{n} image(S_g, T_{c,i})$) are used instead of the theoretical $S_{g+c} = image(S_g, T_c)$.

Whether it is better to use a conjunctive or a disjunctive partitioning depends on the characteristics of the problem being modeled. In symbolic model checking, a conjunctive partitioning is often better suited because most studied systems are synchronous, in which the transitions from predecessor to successor states are expressed as a set of rules that are all applied in parallel. For this reason, research has generally been focused on the problem of scheduling the conjunction and quantification operations during image computation (Burch et al., 1991a; Chauhan et al., 2001), with a few works combining both types of partitioning (Moon et al., 2000). In planning, though, disjunctive partitioning is more natural, since planning operators are applied sequentially. Jensen et al. (2008) first proposed the partitioning of TRs so that each partition contains operators that change the f-value of the states by the same amount. The precomputation of Δf was obtained from the cost of the operators of the problem and the increase or decrease of the heuristic value. This limits their method to problems and heuristics where Δh can be automatically derived from the problem description.

Due to the inadequateness of using a single T_c in some domains, GAMER uses a TR per operator. We now describe how TRs that represent a single operator $o \in \mathcal{O}$ are computed. Given an operator $o = (pre(o), eff(o), c(o))$, its associated transition relation T_o is obtained as the conjunction of its literals, describing the preconditions with variables from x and the effects with variables from x'. Any variable not modified by o must also be explicitly encoded so that it keeps its value after the application of o. This is encoded in the TR as a bi-implication of the form $biimp\,(x_i, x'_i) = (x_i \wedge x'_i) \vee (\overline{x}_i \wedge \overline{x}'_i)$. Thus, if we assume that the function $fBDD(v, val, x)$ returns the BDD that corresponds to the fluent $\langle v, val\rangle$ represented with the set of variables x, the TR of a single operator is computed as follows:

$$T_o = \bigwedge_{\langle v,val\rangle \in pre(o)} fBDD(v, val, x) \wedge \bigwedge_{\langle v,val\rangle \in eff(o)} fBDD(v, val, x') \wedge \bigwedge_{k \in \mathcal{V} \setminus \mathcal{V}_{eff(o)}} biimp(x_k, x'_k)$$



(a) $v_1 \wedge \neg v_3 \wedge v'_3$ (b) $biimp(v_2, v'_2)$ (c) TR(o)

Figure 3.1: Transition Relation that represents a single operator o, $pre(o) = \{x_1, \neg x_3\}$, $eff(o) = x_3$.

The TR of a single operator is efficiently representable, as exemplified in Figure 3.1. First, the preconditions and effects are simple conjunctions of facts, which requires a BDD whose size is linear in the number of relevant variables (see Figure 3.1a). Second, the bi-implications are trivially representable as long as the variables from x and x' that correspond to the same variable $v \in \mathcal{V}$ are adjacent in the variable ordering, as they are in our ordering schema. If this is the case, every pair of related variables will be represented by a BDD with three nodes which lead to the same node when the bi-implication function evaluates to true, as in Figure 3.1b. Finally, as all the bi-implication, precondition and effect BDDs are independent of each other, the TR relative to the operator (see Figure 3.1c) is just a concatenation of the BDDs.

When multiple TRs are used, the successors are sorted by their g value and aggregated in the same S_g. Even if there are several BDDs that represent sets of states with the same g, BDDs are always merged two at a time. Since the computational cost of most BDD operations depends on their size, it is important to keep the intermediate BDDs as small as possible. Therefore, even though the result will always be the same, the order in which disjunctions are applied has an impact on the performance of image computation. To model the order in which disjunctions are applied, GAMER represents them in a binary tree, called *disjunction tree* (see Figure 3.2).[1] Each internal node applies the disjunction of the result of its left and right branches. Each leaf node is associated with an operator, so that it represents the image result with respect to that operator's TR. GAMER constructs a balanced disjunctive tree by arbitrarily splitting the operators in each internal node into two almost equally sized partitions.

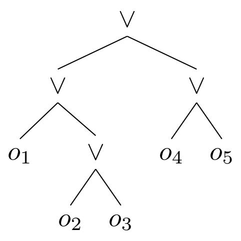


Figure 3.2: Example of disjunction tree. Leaf nodes are associated with an operator. Internal nodes correspond to the disjunction of the result of its left and right branches.

Next, we propose different ways to compute the image for symbolic search, taking advantage of the characteristics of planning benchmarks. First, we consider how the set of auxiliary variables x' may be avoided when using a TR per operator. Then, we present a conjunction tree whose goal is to improve how the operator preconditions are matched. Finally, we discuss other disjunctive partitioning criteria to represent the TRs.

3.2 Image without Auxiliary Variables

As previously mentioned, TRs are usually represented as functions over two sets of variables: the predecessor set x and the successor set x'. However, the semantics of a planning operator establishes that the effect does not depend on the preconditions, i. e., preconditions and effects can be described independently. It is possible to take advantage of this fact in two different ways. First, all the variables not appearing in the effects must retain their values. When computing the image using T_o, the existential quantification and variable swapping only need to be applied over variables modified by $\mathit{eff}(o)$. Therefore, the bi-implication term may be omitted from T_o, using instead $\widetilde{T_o}$, which is defined as: $\widetilde{T_o} = \bigwedge_{\langle v,val\rangle \in pre(o)} \mathit{fBDD}(v, val, x) \wedge \bigwedge_{\langle v,val\rangle \in \mathit{eff}(o)} \mathit{fBDD}(v, val, x')$. Then, if we assume that $x_o \subseteq x$ is the set of variables modified by o and $x'_o \subseteq x'$ is the corresponding set of variables from x', the image is computed as follows:

$$\mathit{image}(S_g, T_o) = (\exists x_o \ (S_g \wedge T_o))\,[x'_o \leftrightarrow x_o]$$

Avoiding the explicit representation of the bi-implication term allows to divide T_o into a precondition T_o^{pre} over x and an effect T_o^{eff} over x' such that $T_o = T_o^{pre} \wedge T_o^{\mathit{eff}}$. In fact, once preconditions and effects are encoded separately there is no need to relate the sets x and x' anymore, which allows us to represent the TRs using only the set of predecessor variables x, which is equivalent to using $\mathcal{V}$ with no further modification. This way the variables representing the successor states x' as well as the swap operation are no longer needed, which means that the intermediate BDDs during image

[1]The original version of GAMER did not explicitly model the disjunction tree, though used an equivalent iterative algorithm to apply the disjunctions in a balanced way.

computation will be more succinct and hopefully the image operation will be slightly faster. The new image operation is defined as follows:

$$image(S, T_o) = (\exists x_o\ (S_g \wedge T_o^{pre})) \wedge T_o^{eff}$$

3.3 Conjunction Trees

When considering multiple TRs it is advisable to avoid checking their applicability individually. This is especially important when a TR per operator is used instead of a partitioning schema, as the number of grounded operators can become very large in some planning problems. Some explicit-state planners employ speed-up techniques to filter non-applicable operators efficiently, such as the successor generator used by Fast Downward (Helmert, 2006b). In Fast Downward the operators are organized in a decision tree (see Figure 3.3) similar to the structures used by RETE networks to detect the triggering of a rule (Forgy, 1982). In these decision trees each leaf node contains a set of operators that have the same preconditions. Every internal node is associated with a variable $v \in \mathcal{V}$ and has an edge for every value i of v and an additional "don't care" edge. An operator $o \in \mathcal{O}$ is propagated down the i edge if and only if $\langle v = i \rangle \in pre(o)$ and down the "don't care" edge if $v \notin \mathcal{V}_{pre(o)}$. To compute the successors of a state s, the tree is traversed, omitting branches labeled with unsatisfied preconditions and always following "don't care" edges. An operator is applicable in s if and only if it is in a leaf reached by that traversal.

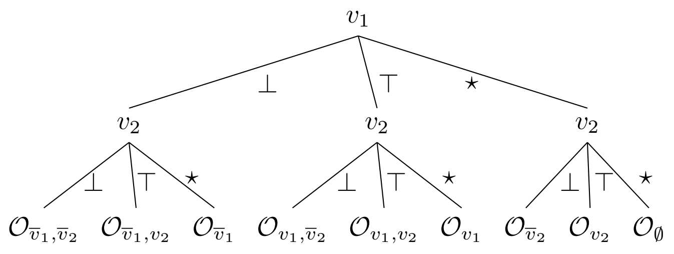

v_1
$\perp$
$\top$
$\star$
v_2
v_2
v_2
$\perp$
$\top$
$\star$
$\mathcal{O}_{\overline{v}_1,\overline{v}_2}$
$\mathcal{O}_{\overline{v}_1,v_2}$
$\mathcal{O}_{\overline{v}_1}$
$\mathcal{O}_{v_1,\overline{v}_2}$
$\mathcal{O}_{v_1,v_2}$
$\mathcal{O}_{v_1}$
$\mathcal{O}_{\overline{v}_2}$
$\mathcal{O}_{v_2}$
$\mathcal{O}_{\emptyset}$

Figure 3.3: Conjunction tree for applying an operator's precondition on a set of states. Each internal node corresponds to one binary variable, v_i, classifying the operators depending on whether they have precondition $v_i = \top$, $v_i = \perp$, or no precondition on v_i ($v_i = \star$).

This approach carries over to BDDs as follows. As all the variables are binary, each internal node only has three children c_0, c_1 and $c_\star$, dividing the operators into three sets: those that require $\overline{v}$ as a precondition, those that require v as a precondition and those whose applicability does not depend on v at all. Applicability of operators is different for progression and regression search, so different conjunction trees are needed. Conjunction trees for forward search take into account the preconditions of operators. Conjunction trees for backward search take into account the preconditions of the inverted actions, i. e., their effects and prevail conditions. In the presence of zero-cost operators, all algorithms studied in Chapter 2 apply breadth-first search using only the subset of zero-cost operators until a fix-point is reached. Since regular and zero-cost operators are applied over different sets of states, two different conjunction trees are needed: one with the zero-cost operators and another one with the rest. Therefore, in order to apply bidirectional search in domains with zero-cost operators up to four different conjunction trees are needed: fw-zero, fw-cost, bw-zero, and bw-cost.

In symbolic search the operators are not applied over a single state but rather over sets of states. Nevertheless, operators only need to be applied over states that satisfy the corresponding conditions.

Algorithm 3.1: CT-image: Image using the conjunction tree

```
Input: node: Root of the $CT$ subtree.
Input: $S_g$: BDD of states for which the applicable actions are calculated.
Output: Set of pairs $\langle c, S_c \rangle$ where $S_c$ is a set of successor states generated with an operator
        of cost $c$.
1 if $S_g = \bot$ then return $\emptyset$
2 if node is leaf then
3     return $\bigcup_{o \in node.\mathcal{O}} \{\langle c(o), image(S_g, T_o) \rangle\}$
4 $v \leftarrow node.variable$
5 $r_0 \leftarrow CT\text{-}image(node.c_0, S_g \wedge \neg v)$
6 $r_1 \leftarrow CT\text{-}image(node.c_1, S_g \wedge v)$
7 $r_\star \leftarrow CT\text{-}image(node.c_\star, S_g)$
8 return $r_0 \cup r_1 \cup r_\star$
```

We take advantage of this by precomputing the subset of states relevant to a given TR prior to the image operation. This is done by splitting the original set of states into subsets as the successor tree is traversed. This way, the subset on which a TR is applied once a leaf node is reached is the subset of states that satisfy the preconditions of the TR. We call this method *conjunction tree* (CT).

Algorithm 3.1 shows how to compute the image of a BDD using the CT. It takes as input a set of states S_g and the root node of the CT and returns the sets of successor states, associated with the cost of the operators that generated them. At every inner node one recursive call is made for each child node, applying the corresponding conjunction between the set of states S_g and the precondition $v \in \mathcal{V}$ associated with the node. Moreover, if there are no states satisfying the preconditions of a branch, it is not traversed. When a leaf node is reached, $image(S'_g, T_o)$ is computed. Since S'_g at the leaves is already the result of computing the conjunction of the original S_g and the operator preconditions, here we should only account for the effects. Thus, in the leaf nodes the image is computed as $image(S'_g, T_o) = (\exists x_o S'_g) \wedge T_o^{eff}$.

The CT has several advantages over the regular approach: first, if S_g does not contain any state matching the conditions of a branch of the tree, all the operators regarding that branch are ignored, which reduces the number of individual image operations that are needed. Second, if several operators share the same preconditions, the conjunction of S_g with that set of preconditions is computed only once. Finally, the conjunction with the preconditions is done with BDDs whose size usually decreases as we go deeper in the CT, so the time required to do the individual conjunctions will be shorter.

An overhead may occur due to the computation and storage of intermediate BDDs in memory, though. The conjunction with a precondition should only be used when the benefits are estimated to compensate the overhead, as when a precondition is shared between many operators. Thus, the CT can be parametrized with a parameter *min operators conjunction*, so that the intermediate conjunction with a partial precondition is only computed when needed for at least that number of operators. If the number of operators that should go down some edge is less than *min operators conjunction*, they are propagated down the *don't care* branch $c_\star$ instead, and marked so that we know that not all their preconditions have been checked. When computing the image of marked operators, the conjunction with their preconditions is needed. When *min operators conjunction* is set to 1 we have the full tree strategy and when it is set to ∞ the CT consists of only one leaf node containing all the operators, which is equivalent to not having a tree at all. Setting the parameter to intermediate values produces intermediate strategies.

The performance of the CT depends on the order in which it checks the conditions. We tried two different heuristic criteria to generate the CT. The first one, the *level* criterion, is to use the same variable ordering as in the BDD representation. This aims at making the conjunctions as simple as possible, since a conjunction with the first variable of a BDD is trivial. Our second criterion, *dynamic*, is to use the variable that appears most often in the preconditions of the operators down that branch. This is a greedy strategy, aiming to reuse the computation of the conjunction for as many operators as possible.

3.4 Unions of Transition Relations

Even though having a monolithic TR per action cost is often unfeasible due to its size, computing the union of TRs of only a subset of operators may be beneficial. The union of a set of TRs $\mathcal{T}$ is a new TR, $Union(\mathcal{T})$, such that the image with respect to $Union(\mathcal{T})$ is equivalent to the disjunction of the images of the individual TRs in $\mathcal{T}$, $image(S_g, Union(\mathcal{T})) = \bigvee_{T \in \mathcal{T}} image(S_g, T)$.

$Union(\mathcal{T})$ cannot be represented with separated preconditions and effects as described in Section 3.2 though, as the bi-implications are needed to ensure that the generated successors are correct. However, not all the bi-implications need to be explicitly included: a $T_o \in \mathcal{T}$ must include a bi-implication related to some variables x_i only if x_i is modified by some other $T'_o \in \mathcal{T}$ (and not by T_o). Thus, $Union(\mathcal{T})$ may be computed as shown in Equation 3.2, where X_T and $X_{\mathcal{T}}$ are the sets of variables in the effects of operators represented by the transition relation T and any transition relation in the set $\mathcal{T}$, respectively.

$$Union(\mathcal{T}) = \bigvee_{T \in \mathcal{T}} \Big(T \wedge biimp_{x_i \in X_{\mathcal{T}} \setminus X_T}(x_i, x'_i) \Big) \tag{3.2}$$

A critical decision is which operators should be joined together in a single TR while ensuring that the resulting $Union(\mathcal{T})$ is tractable to compute. Algorithm 3.2 is a simple algorithm for creating a disjunctive partitioning starting from the set of one TR per each operator. Recall that, as the cost of reaching a state must be preserved, only operators with the same cost c may be aggregated. Therefore, we call the algorithm once per different cost, i, with $\mathcal{T}_i = \{T_o \mid o \in \mathcal{O}, c(o) = i\}$.

Given a set of TRs, the algorithm iteratively aggregates them until only one element in the set of candidates $\mathcal{T}$ is left or the maximum allotted time is exceeded, represented by the parameter *maxTime*. The parameter *maxNodes* is a limit on the number of nodes that the transition relation BDDs may have. $\mathcal{T}_{res}$ is the set of TRs that cannot be aggregated anymore. If *maxNodes* is exceeded during the computation of the union, the TRs whose union was attempted are added to $\mathcal{T}_{res}$ instead to the candidate set $\mathcal{T}$. This means that, even if the disjunction of a TR with some other T' could be possible, once they are in $\mathcal{T}_{res}$ they are not considered for aggregation anymore and are returned at the end in the resulting set of TRs.

The most important aspect of the algorithm is the selection of the TRs to be aggregated in line 3. This is important, not only because it may impact the algorithm performance, but also because, if a single $Union(\mathcal{T}_c)$ cannot be computed for some cost c, it is important to have a set of TRs as balanced in size as possible. As we impose a limit on the maximum size of a TR, balancing the size of the TRs will often lead to a better partitioning with a lower number of TRs. We define three strategies, based on different criteria, to select which TRs should be unified:

Disjunction Tree (T^{DT}) This criterion employs the balanced disjunction tree shown in Figure 3.2. Instead of computing the union of the successors generated with the TRs that appear at the leaf nodes,

Algorithm 3.2: Aggregate

```
Input: $\mathcal{T}$: Set of elements to aggregate.
Input: maxTime, maxNodes: Time and node bounds.
Output: Set of elements aggregated.
1  $\mathcal{T}_{res} = \emptyset$
2  while $|\mathcal{T}| > 1$ and currentTime $<$ maxTime do
3      Select $T_i, T_j \in \mathcal{T}$
4      $\mathcal{T} \leftarrow \mathcal{T} \setminus \{T_i, T_j\}$
5      $T' \leftarrow Union(\{T_i, T_j\}, maxTime - currentTime, maxNodes)$
6      if Union succeeded then
7          $\mathcal{T} \leftarrow \mathcal{T} \cup \{T'\}$
8      else
9          $\mathcal{T}_{res} \leftarrow \mathcal{T}_{res} \cup \{T_i, T_j\}$
10 return $\mathcal{T}_{res} \cup \mathcal{T}$
```

the actual TRs are merged. TRs that share the same parent node are merged in a bottom-up fashion until *maxNodes* is exceeded in that branch or the algorithm runs out of time.

The strategy T^{DT} uses an arbitrary order of operators, which means that the resulting set of TRs may end up being imbalanced. For example, if a small TR is merged with another TR whose size is close to *maxNodes* and the union of both TRs is bigger than *maxNodes*, then the first TR will not be considered for merging anymore. If this occurs, it may happen that the set of final TRs contains individual TRs that could have been merged without exceeding *maxNodes*.

Smallest TRs (T^{SM}) is a simple criterion to obtain TRs as balanced as possible in size. T^{SM} aggregates the two smallest TRs at every step.

Conjunction Tree (T^{CT}) The previous strategies have the disadvantage of not exploiting the synergy between operators — the union of two TRs whose operators have similar preconditions and effects tends to be more succinct —, which may result in bigger intermediate BDDs. To exploit the similarity between operators, the CT described in Section 3.3 can also be used as the policy employed to aggregate TRs. The new policy T^{CT} aggregates TRs of operators present in the same branch of the CT just like T^{DT} does. This ensures that there will be some degree of similarity between the TRs because at least one subset of the preconditions will be shared between all the operators. Also, this allows us to use the CT in combination with a disjunctive partitioning in a seamless way: TRs are merged up the tree from the leaves until Algorithm 3.2 finishes, in which case the merged TRs will be the new leaf nodes of the CT. This keeps the property of the CT that only states that can be expanded by some operator are propagated down the tree even if the TRs of the individual operators are merged.

3.5 Empirical Evaluation

In this section, we empirically evaluate the different techniques proposed in this chapter and test the impact of the different parameters defined. We use these image computation methods within the symbolic search algorithms implemented in GAMER: forward/backward uniform-cost (see Section 2.4 on page 26), bidirectional uniform-cost (see Section 2.5 on page 28) and A* search (see Section 2.6 on page 30). All the settings of our experiments, including the benchmarks and metrics used, were described in Section 1.5 on page 10. The main goal of our evaluation is to measure the impact that the different methods of image computation have on the performance of the algorithms. We consider the following configurations:

- TR^1: GAMER's original image computation. TR^1 uses a TR for each single operator. This is the baseline against which we compare the rest of the image computation techniques proposed in this chapter.

- TR^{1+}: Uses a TR for each single operator, but it separately represents the preconditions and effects of the operator avoiding the use of auxiliary variables. See Section 3.2 on page 40.

- CT: uses the conjunction tree to match the action preconditions. As explained in Section 3.3 on page 41, it has two different parameters that configure how the conjunction tree is constructed:

    1. Variable ordering in the CT, i. e., the order in which the variables are checked for the operator preconditions. We contemplated two criteria: level (CT^{L}) and dynamic (CT^{D}) and compared them against the reversed level (CT^{RL}) and random selection (CT^{R}).
    2. *min operators conjunction*: the minimum number of operators in each leaf of the CT. Apart from the "standard" version with value 1, we perform experiments with 5, 10 and 20: CT_5, CT_{10} and CT_{20}.

- Union of TRs (T^{DT}, T^{SM} and T^{CT}): Represents multiple operators with a single TR, instead of using a TR for each operator. The aggregation algorithm proposed in Section 3.4 on page 43 combines as many operators as possible, while guaranteeing that the final TRs fit in the available memory. It uses three parameters:

    1. Aggregation strategy: the criteria used to select which TRs are aggregated at each step in the algorithm: disjunction tree (T^{DT}), smallest (T^{SM}), and conjunction tree (T^{CT}).
    2. Maximum TR size: Limit on the number of nodes that the BDDs describing TRs may have. In our experiments, we used values ranging from 1000 (T_{1k}) to one million (T_{1M}).
    3. Maximum time: Fixed to 60 seconds. We did not perform a deep analysis of this parameter since the maximum BDD size is more relevant to control the resources invested to aggregate the TRs (memory is the most scarce resource in this case).

Given the large amount of configurations and algorithms, reporting all the combinations is not feasible. Instead, we divide our evaluation into several steps:

1. First, in Section 3.5.1, we evaluate the performance of bidirectional uniform-cost search with the different image computation methods. We perform a detailed analysis of different parameter configurations of the methods we have proposed in this chapter and compare them to the previous approach in GAMER. We chose bidirectional uniform-cost search for our detailed analysis because it already contemplates the case of forward and backward search. Moreover, it is less biased than Symbolic A*, because it does not rely on any abstraction selection mechanism that may alter our conclusions.

2. Once all the image computation methods have been evaluated, we check whether the main conclusions obtained from the bidirectional uniform-cost case can be extrapolated to unidirectional uniform-cost and Symbolic A*, in Sections 3.5.2 and 3.5.3, respectively. Additionally, we perform a comparison of forward and backward search and see whether there are differences in our conclusions between image and pre-image computation.

3.5.1 Image Computation on Bidirectional Uniform-Cost Search

In this section, we analyze the performance of bidirectional uniform-cost search when using different image computation methods, comparing the original image computation of GAMER, TR^1, against the new approaches proposed in this chapter. We first present an overview of the different methods and, afterwards, a detailed analysis of the configuration parameters.

Table 3.1 shows the time score results of our different approaches to image computation. As the results highlight, the new image computation methods outperform the previous image approach of GAMER, not only in terms of total coverage and score, but also on a domain per domain basis. The time score results show that the new image computation methods provide an improvement in almost all domains (except in PSR-SMALL, where all the problems are solved, and PARKING, where no configuration solves any problem). This means that the new image computation methods make symbolic search more efficient, being able to solve problems in less time. Moreover, this increase in efficiency allows the planner to solve more problems in 21 out of 44 domains. Given the exponential growth in problem complexity in most domains this reveals significant performance gains.

However, not all the new approaches have the same results. The first new approach, TR^{1+}, which divides each TR into a precondition and an effect BDD, already dominates the results of the old GAMER approach, having equal or better performance in all domains except TIDYBOT and the two versions of PIPESWORLD. In total, it solves 36 more problems than TR^1 across 17 domains so the improvement can be relevant in practice.

Another interesting comparison is that of TR^{1+}, CT^{L} and CT^{L}_{20}. CT^{L} is an extension of TR^{1+} that uses a complete conjunction tree. CT^{L}_{20} is a mix of both that aims to reduce the overhead of the conjunction tree by disabling it whenever a precondition is shared among less than 20 operators. At first glance, we observe that the results of CT^{L} seem to be a bit worse than those of TR^{1+}, though it proves to be quite useful in a few domains like FREECELL and, especially, TIDYBOT where it is the best method. Unsurprisingly, in these domains the problem instances have lots of operators with many preconditions. However, in several domains such as BLOCKSWORLD, SOKOBAN, or TRANSPORT the coverage of the planner decreases when using CT^{L} instead of TR^{1+}. This result confirms our intuition that, while the conjunction tree successfully takes advantage of checking the same precondition for several operators, there is an overhead that can be harmful in some domains. Fortunately, the overhead can be limited by setting the *min operators conjunction* parameter to 20 operators, CT^{L}_{20}. In most domains, the time score of CT^{L}_{20} is just the maximum of TR^{1+} and CT^{L}, so the parameter is successfully able to control when it is useful to use the conjunction tree approach. Moreover, there are some cases in which CT^{L}_{20} is better than both TR^{1+} and CT^{L}. For example, in FREECELL and GRIPPER the use of the complete conjunction tree is already beneficial, but considering a reduced conjunction tree is even better.

Even though all the new image approaches improve the results of our base planner, GAMER, the clear winners among our image computation variants are the approaches that aggregate TRs. The total time score of T^{DT}_{100k} is 806.67, very close to the total coverage of 814. This means that T^{DT}_{100k} is reliably the fastest image computation method in almost every domain. In the few domains where it is not the fastest one, its performance is close to the fastest. Moreover, T^{DT}_{100k} obtains the

	TR^1	TR^{1+}	CT^{L}	CT^{L}_{20}	T^{DT}_{100k}
Airport (50)	20.82	20.96	18.80	21.24	***23.97**
Barman (20)	5.80	5.92	5.65	5.98	**8.00**
Blocksworld (35)	19.83	***22.60**	19.01	***22.77**	20.10
Depot (22)	4.08	***5.71**	4.13	***5.68**	4.70
Driverlog (20)	9.42	11.11	9.98	11.22	***14.00**
Elevators08 (30)	16.85	*20.39	*20.02	*20.84	***25.00**
Elevators11 (20)	12.79	15.34	15.04	15.52	**19.00**
Floortile11 (20)	7.39	9.39	8.13	9.37	***12.00**
Freecell (80)	9.52	9.70	10.78	12.93	***20.00**
Grid (5)	1.57	1.57	1.63	1.67	**2.00**
Gripper (20)	15.91	17.38	17.55	**19.54**	18.21
Logistics 00 (28)	16.44	18.37	17.34	18.42	**19.96**
Logistics 98 (35)	3.23	4.00	3.44	3.76	**5.00**
Miconic (150)	61.79	82.40	81.71	82.51	***113.00**
MPrime (35)	16.25	17.61	18.43	***20.27**	**20.36**
Mystery (30)	*12.56	*12.65	12.08	12.56	***13.33**
NoMystery11 (20)	10.67	13.63	13.25	13.89	***15.68**
Openstacks08 (30)	21.95	23.70	23.74	24.49	**30.00**
Openstacks11 (20)	13.69	14.88	14.47	15.01	**20.00**
Openstacks06 (30)	9.40	9.81	9.19	10.05	***19.98**
ParcPrinter08 (30)	17.91	19.69	17.67	18.85	***23.62**
ParcPrinter11 (20)	12.73	14.54	12.67	14.40	***17.32**
Parking11 (20)	0.00	0.00	0.00	0.00	0.00
Pathways-noneg (30)	4.77	4.82	4.50	4.88	**5.00**
Peg-Solitaire08 (30)	25.09	*29.43	*28.01	***29.88**	***29.70**
Peg-Solitaire11 (20)	15.52	*19.33	*17.68	***19.89**	***19.70**
Pipesworld-nt (50)	*12.90	*12.36	*13.69	*12.86	**13.85**
Pipesworld-t (50)	11.64	10.69	10.82	11.97	***17.00**
PSR-small (50)	**49.56**	**49.45**	**49.45**	**49.51**	**49.74**
Rovers (40)	10.84	11.87	11.68	11.91	**14.00**
Satellite (36)	5.99	*7.66	*7.62	*7.65	***9.00**
Scanalyzer08 (30)	8.87	10.02	10.50	10.54	**11.68**
Scanalyzer11 (20)	5.96	7.05	7.67	7.71	**8.94**
Sokoban08 (30)	*21.40	*21.70	19.19	*21.71	***27.97**
Sokoban11 (20)	15.01	15.38	14.07	15.38	**19.97**
Tidybot11 (20)	*10.76	7.83	***11.41**	***11.31**	10.95
TPP (30)	7.18	7.62	7.28	7.62	**8.00**
Transport08 (30)	10.10	*12.19	11.60	*12.72	***14.00**
Transport11 (20)	5.74	*7.96	5.84	*8.54	***10.00**
Trucks (30)	8.32	9.18	8.79	9.16	***11.00**
VisitAll (20)	10.42	11.65	11.34	11.66	**11.94**
Woodworking08 (30)	17.15	18.90	17.83	18.91	**22.00**
Woodworking11 (20)	11.45	13.10	12.04	13.01	**16.00**
Zenotravel (20)	7.31	*9.81	9.04	*9.58	***11.00**
Total (1396)	596.58	669.35	644.76	687.37	**806.67**
Score (36)	14.95	16.47	15.92	17.00	**19.36**
Total Cov (1396)	748	784	770	792	**814**
Score Cov (36)	18.38	18.92	18.63	19.14	**19.61**

Table 3.1: Time score of bidirectional uniform-cost search with different configurations of image computation: the original image of Gamer, TR^1, against TR^{1+}, conjunction tree, CT^{L} and CT^{L}_{20}, and TR aggregation, T^{DT}_{100k}. The best configurations per domain and those deviating in only 1% are highlighted in bold and, in domains where there are differences in coverage, the configurations with best coverage are marked with *.

best coverage in many domains, solving 66 more problems than the original image computation of GAMER (27 of those are in MICONIC). Other image computation methods outperform T^{DT}_{100k} only in a few domains and in those cases by a small margin.

The performance gain of the best new method for image computation, T^{DT}_{100k}, with respect the old GAMER approach is depicted in the plot of Figure 3.4. The plot reflects the overwhelming superiority of the approach that aggregates multiple TRs. T^{DT}_{100k} is better in the vast majority of problems, solving problems up to two orders of magnitude faster than the baseline and being only clearly worse on four problems of the whole benchmark set.

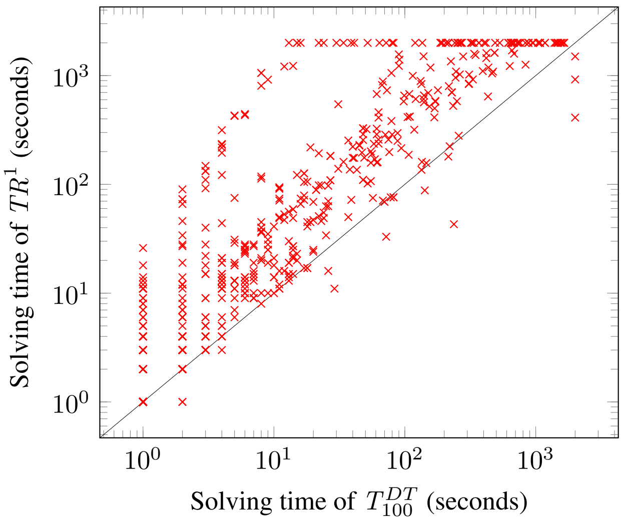


Figure 3.4: Comparison of solving time of the new T^{DT}_{100k} image computation versus the old GAMER image computation. Unsolved problems are assigned a time of 2000 seconds.

Parameter Configuration of the Conjunction Tree

We take a closer look at the impact that the different parameters have on the performance of the conjunction tree, defined in Section 3.3. We compare the criteria to select the variables that split the operators in each internal node of the conjunction tree, as well as the *min operators conjunction* parameter.

We consider four different heuristic criteria to select the variables that split the operators in each node of the conjunction tree. We compare the two criteria introduced in Section 3.3 on page 41, level and dynamic, against two baseline approaches, reversed level and random. The level criterion selects the variables in the same ordering used to represent the BDDs. The dynamic criterion selects the variable that appears in more operator preconditions. Reversed level selects the variables in the reverse order than the level criterion. Finally, the random criterion just selects variables at random.

Table 3.2 shows the total time score of each configuration, normalized according to the number of problems in each domain. The first trend observed is that configurations using the conjunction tree with a *min operators conjunction* value greater than one (the full conjunction tree) but distinct from ∞ (no conjunction tree at all) are better than both extremes. Using the conjunction tree is a good idea, but only when the same precondition is shared between several operators. However, the exact value of *min operators conjunction* does not seem to matter that much. Any value around 10 or 20 is a reasonable choice and the results are quite similar even with a value of 5. This means that, as soon as there are 5 or more operators with the same precondition, applying the conjunction tree will speed up the image computation.

On the other hand, comparing the variable selection strategies, for low values of *min operators conjunction*, the level and dynamic criteria are better than reversed level or random. Increasing the *min operators conjunction* parameter reduces the number of variables that can be selected, so that all tie-breaking strategies converge to similar performance with *min operators conjunction* equal to 20. While none of the strategies is worse than not using the conjunction tree (∞) when using an adequate value of the *min operators conjunction* parameter, the dynamic and level strategies are more stable than reversed level or random, obtaining good results with any values of *min operators conjunction* greater or equal to 5. The configuration with best coverage is reversed level (CT^{RL}), which makes the most of the conjunction tree being able to solve 11 more problems than TR^{1+} (∞). The dynamic and level (CT^{L}) strategies get slightly worse results, though the difference is not very significant.

	1	5	10	20	∞
CT^{L}	17.31 (770)	18.61 (792)	**18.74** (790)	**18.73** (792)	18.13 (784)
CT^{D}	17.67 (778)	**18.72** (791)	**18.84** (793)	**18.79** (793)	18.13 (784)
CT^{RL}	16.61 (761)	18.27 (784)	18.51 (787)	**18.80** (**795**)	18.13 (784)
CT^{R}	16.48 (759)	18.17 (782)	18.53 (787)	**18.75** (792)	18.13 (784)

Table 3.2: Total time score and coverage of bidirectional uniform-cost search with different configurations of the conjunction tree, CT. Each row represents a selection criterion: level (CT^{L}), dynamic (CT^{D}), reversed level (CT^{RL}) and random (CT^{R}). Each column represents a value of the *min operators conjunction* parameter, where 1 corresponds to using the complete conjunction tree and ∞ stands for the configuration without conjunction tree (TR^{1+}) for all the selection criteria. The best configurations and time scores deviating in only 1% from the best are highlighted in bold.

As a conclusion, using a conjunction tree may speed up the image computation whenever the planning operators are represented in isolation and enough operators share some preconditions. In order to avoid the overhead of checking each precondition separately, it is necessary to introduce a parameter *min operators conjunction*. Moreover, the results are not highly sensitive to the exact value of the parameter if it is set to a reasonable value.

Parameter Configuration of TR Aggregation

As we have seen in the results of Table 3.1 on page 47, representing several operators with a single TR produces a significant increase in the performance of symbolic search. However, since representing all the operators with a single TR may be unfeasible, one must decide which operators should be represented together. Our aggregation algorithm (see Algorithm 3.2), which automatically derives a disjunctive partitioning of the TR, takes two different parameters to control the resulting partitioning: the aggregation strategy and the maximum TR size. In Section 3.4 on page 43 we defined three criteria to select which TRs to merge in each algorithm iteration. The disjunction tree (T^{DT}) aims to obtain TRs balanced in the number of operators. The smallest TRs strategy (T^{SM}) always selects the two smallest TRs. On the other hand, the maximum TR size controls the memory used to represent the TRs. The conjunction tree strategy (T^{CT}) aggregates TRs with the same preconditions so that the conjunction tree can be used whenever more than one TR is needed to describe all the operators. If the maximum TR size is set to 1, no TRs are aggregated and the algorithm behaves as the original TR^1. If the maximum TR size is set to ∞, the planner will use a monolithic TR for each action cost, but may exceed the available memory in the initialization. Given the memory limit of 4GB, we set the parameter to different values ranging from 1000 nodes (1k) to one million (1M).

	1	1k	10k	100k	1M	∞
T^{DT}	14.73 (748)	18.16 (807)	18.62 (809)	**18.90** (814)	18.70 (814)	18.19 (804)
T^{SM}		16.17 (778)	17.26 (780)	18.02 (798)	18.04 (798)	17.77 (790)
T^{CTl}		18.28 (817)	18.54 (814)	18.53 (812)	18.29 (809)	17.73 (796)
T^{CTl}_{20}		18.25 (817)	18.67 (**821**)	18.53 (813)	18.35 (808)	17.83 (792)
T^{CTd}		18.23 (818)	18.46 (815)	18.23 (811)	18.13 (809)	17.61 (791)
T^{CTd}_{20}		18.14 (815)	18.49 (813)	18.34 (813)	18.09 (805)	17.80 (793)

Table 3.3: Total time score and coverage of bidirectional uniform-cost search with different configurations of TR aggregation. Each row represents a selection criterion: disjunction tree (T^{DT}), smallest TRs (T^{SM}) and conjunction tree (T^{CT}). Each column represents a value of the maximum TR size parameter that controls the memory used to represent the TR. A value of 1 corresponds to the original GAMER image computation (TR^1) and a value of ∞ represents the monolithic TR approach.

Table 3.3 shows the time score and total coverage of each parameter configuration. The results show that, while aggregating TRs has an important impact on the planners' performance, the parameters that control how to aggregate have limited relevance.

We have tried different values for the maximum size of the TRs. The configuration without any TR aggregation only solves 748 instances, 73 less than the best configuration in terms of coverage. However, when aggregating a few TRs, up to a maximum of 1000 nodes is enough to get a coverage of 818, only three problems behind the maximum coverage achieved. The fastest configurations are reliably those setting the maximum size of TRs in 10000 or 100000. For larger TRs, the coverage slightly decreases, mainly because the TRs use too much memory or the process cannot even terminate.

Regarding the criteria to select the order in which the TRs are aggregated, T^{SM} performs clearly worse than the other criteria, both in coverage and time score, but the differences between the rest of the versions are not large enough to consider them significant. Thus, we conclude that any of these strategies which attempt to aggregate "similar" TRs with respect to different definitions are all valid strategies to decide which TRs should be aggregated. In the rest of the thesis we will use the T^{DT}_{100k} criterion. Even though there are other configurations with slightly better coverage, it is the fastest approach according to our time score measure. Also, given that all the methods are close to each other, we selected the simpler approach that does not use any conjunction tree.

3.5.2 Image Computation in Unidirectional Uniform-Cost Search

In the previous subsection, we analyzed the impact that the new image computation methods have on the performance of bidirectional symbolic uniform-cost search. In this section, we compare the results in unidirectional search and examine whether the new image techniques have more impact in forward or backward search. Table 3.4 shows the summary scores of the image computation methods on unidirectional uniform-cost search, both in forward and backward directions. For both the disjunction-based image and the conjunction tree approaches, we selected the same configurations as in the bidirectional search table.

The main conclusions are similar to those in the bidirectional search case, with T^{DT}_{100k} being the best configuration for image computation in almost every domain and the new image computation approaches outperforming the base symbolic planner, TR^1.

	fw				*bw*			
	TR^1	TR^{1+}	CT^{L}_{20}	T^{DT}_{100k}	TR^1	TR^{1+}	CT^{L}_{20}	T^{DT}_{100k}
TOTAL (1396)	595.05	618.50	645.25	**740.36**	306.73	390.25	396.19	**467.42**
SCORE (36)	14.79	15.22	15.98	**18.09**	7.39	9.12	9.22	**10.43**
TOTAL COV (1396)	722	734	749	**768**	453	534	538	**541**
SCORE COV (36)	17.63	17.73	18.13	**18.58**	10.71	12.22	12.29	**12.34**

Table 3.4: Summary time and coverage scores of symbolic forward and backward uniform-cost search using different image computation techniques. Best results in each direction are highlighted in bold.

Finally, the comparison between forward and backward search depicts a clear advantage for the forward case. Regression search outperforms forward search only in five domains — two if we take coverage into account. The results of Table 3.4 are not the definitive comparison of forward and backward blind search in this thesis. Instead, they are a motivation for Chapter 4, where we will analyze the usage of state invariants in symbolic search and how they affect the search performance. For a detailed comparison of blind search methods we refer the reader to Section 4.7.4 on page 76.

3.5.3 Image Computation in Symbolic A* Search

In this section we evaluate the image computation techniques in symbolic A* search with symbolic PDBs. We use the Lazy BDDA* implementation described in Section 2.6.2 on page 33. The symbolic PDBs are generated with GAMER's hill climbing method (Kissmann and Edelkamp, 2011) in a precomputation phase that is terminated after 15 minutes. The image computation methods are used both in the symbolic PDB generation and in the symbolic A* algorithm. Table 3.5 shows the summarized coverage results of the main image approaches proposed in this chapter. Time score metrics are omitted because they are not representative of A* performance, given that all the algorithms spend a maximum of 15 minutes in the PDB generation before starting the search.

	TR^1	TR^{1+}	CT^{L}	CT^{L}_{20}	T^{DT}_{100k}
TOTAL COV (1396)	730	744	711	753	**792**
SCORE COV (36)	18.38	18.43	17.89	18.72	**19.64**

Table 3.5: Summary total coverage and score of symbolic A* search using different image computation techniques.

Once again, the conclusions regarding the comparison of image computation techniques are similar to those obtained with bidirectional symbolic uniform-cost search. This reveals that the advantage of the new image computation methods is not limited to a particular algorithm, but they improve the performance of symbolic search planning algorithms in general. Comparing the results in Table 3.5 with those of symbolic bidirectional uniform-cost search (see Table 3.1) we can observe that the performance of A* is similar to bidirectional uniform-cost search. They have almost the same score, although the latter solves 22 problems more in total. A more detailed comparison of symbolic A* and symbolic bidirectional uniform-cost search is delayed until Section 5.2 on page 84, where all our improvements to symbolic search have been presented.

3.6 Summary

In this chapter we have presented and analyzed image computation methods for symbolic search planning. Our starting point was the state-of-the-art symbolic search planner GAMER that computes the image using a transition relation for each planning operator. We proposed three different alternatives for image computation:

1. Instead of representing each planning operator by means of a single BDD, represent the preconditions and effects independently. That way, no auxiliary set of variables is needed in order to represent the successor states.

2. Apply the preconditions of operators at the same time using the conjunction tree, an approach similar to the one used in explicit-state search planning.

3. Aggregate several operators in a single transition relation, controlling the memory used and avoiding to exceed a given memory limit.

Our analysis has shown that the three approaches improve the previous image computation of GAMER in terms of coverage and time score. Moreover, the experimental results show a dominance across most domains, so that the new image computation methods can replace the previous image computation of GAMER.

The best technique to compute the image, in terms of time-efficiency, is using a single monolithic transition relation to describe all the operators, whenever it can be represented with a reasonable amount of memory. However, as already noticed by previous works, the TR that describes all the planning operators can easily exceed the available memory (that was the main reason to split it into a different TR per planning operator). Thus, one of the main conclusions of our analysis is that for efficient image computation, it is best to represent the planning operators with as few TRs as possible. Whenever possible, all the operators of the task with a given cost should be represented by a single TR.

When constructing a monolithic TR within a reasonable memory limit is impossible, a disjunctive partitioning of the TR is necessary. Previous work in model checking had already suggested similar partitionings. In this chapter, we have proposed an algorithm to derive the disjunctive partitioning automatically, according to several heuristic criteria. Our analysis shows the importance of keeping the size of the TRs as balanced as possible. Of the different parameter configurations that we proposed, T^{DT}_{100k} was among the ones with best performance. Therefore, all the experiments in the rest of the thesis will use T^{DT}_{100k} to perform image computation.

On the other hand, we observed that it is better to avoid the usage of an auxiliary set of variables, representing conditions and effects separately. Unfortunately, this cannot be done when representing more than one operator with the same transition relation since, in general, the operators have different preconditions. Therefore, the best image computation techniques still need the auxiliary set of variables to represent the relation between the precondition and the effect of the operators.

This chapter is an extended version of a previously published work (Torralba et al., 2013a), a collaboration with Stefan Edelkamp and Peter Kissmann.

Chapter 4

State Invariants in Symbolic Search

The most successful uses of symbolic search in planning so far have been regular bidirectional search and the generation of abstraction heuristics (Edelkamp and Reffel, 1998; Torralba et al., 2013b) for use in both symbolic and explicit search algorithms. A common point of these methods is that they require performing regression on the goals of the problem.

Regression in planning is considered to be less robust than progression, mainly due to the generation of numerous spurious states in regression. Spurious states are defined as states that are not reachable from s_0 (Bonet and Geffner, 2001), although other definitions exist (Zilles and Holte, 2010). The other main drawback of regression is subsumption of states, i. e., when a partial state s' is a superset of another already generated state s'' and $g(s') \geq g(s'')$ (in which case we say that s' is subsumed by s''). If this occurs, s' can be safely pruned, as a solution path that goes through s' must be at least as costly as a solution path that goes through s''. Subsumption of states is an important problem in explicit-state search because a hash table, the usual method of duplicate detection, cannot recognize it (Eyerich and Helmert, 2013; Alcázar et al., 2013); however BDDs automatically detect subsumption addressing this problem in symbolic search.

One common way to decrease the negative impact of spurious states is the use of *state invariants*, i. e., constraints that must hold in every non-spurious state. State invariants are precomputed before starting the search and, during the search, every state that violates a constraint is pruned. In this chapter we describe the most relevant state invariants in planning, mutexes and invariant groups that have been typically used to improve regression search (Bonet and Geffner, 2001) and abstraction heuristics (Haslum et al., 2005). Moreover, state invariants are not necessarily limited to pruning spurious states in regression. Considering state-invariant constraints that must hold in any *valid* state that can be part of a plan, constraints can also be used to prune irrelevant states that cannot possibly reach the goal, commonly known as *dead ends*.

In our case, we use state invariants in a symbolic setting, for symbolic forward and backward search and symbolic PDBs. Our main contribution in this chapter is to analyze how to encode these constraints symbolically and use them to prune states in symbolic search.

The chapter is organized as follows. The two types of state invariants that we consider, mutexes and invariant groups, are presented in Sections 4.1.1 and 4.1.2, respectively. In Section 4.1.3 we explain related work that uses state invariants to prune operators and detail the algorithm that we use to automatically derive state invariants. Related work that uses state invariants in partial-state regression search is reviewed in Section 4.2. Our main contribution is to study three methods to encode and use invariants in symbolic search. First, we present the symbolic encoding of state-invariant constraints as BDDs in Section 4.3. Section 4.4 proposes to encode the constraints in the transition relations, i. e., the symbolic representation of planning operators. Section 4.5 considers other uses of state invariants that do not require to prune all invalid states. Finally, we discuss the distinctive properties of abstractions with respect to state invariants in Section 4.6. The chapter concludes with the empirical evaluation (Section 4.7) and a summary of conclusions (Section 4.8).

4.1 State-Invariant Constraints

To alleviate the impact of spurious states, constraints obtained from state invariants of the problem have been employed in explicit-state planners (Bonet and Geffner, 2001; Haslum et al., 2005; Alcázar et al., 2013). A *state-invariant constraint* is a logical formula that must hold in every state that can be part of a plan. We say that a state is *valid* if and only if satisfies all the constraints. Constraints are useful because *invalid* states that violate a state-invariant constraint can be pruned during the search since they cannot be part of the solution, i. e., they are either unreachable or dead ends.

We may divide state-invariant constraints depending on whether they detect unreachable or irrelevant states. A *forward constraint* must hold in every state that is reachable from the initial state and a *backward constraint* must hold in every state from which the goal can be reached. Forward constraints cannot possibly prune any state in forward search, since no unreachable state can be generated by forward search in the first place. Backward constraints are not useful in regression search by similar reasons. Thus, forward and backward constraints are used to prune backward and forward search, respectively.

We obtain a set of constraints in a preprocessing step and then use them to prune the search. In this section, we introduce the two types of state invariants that we consider and then present the algorithm that we use to discover them.

4.1.1 Mutexes

The most common state invariants used to prune spurious states are pairs of mutually exclusive fluents, more generally called *mutexes*.[1]

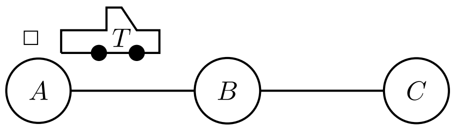
Definition 4.1. *(Mutex pair) A pair of fluents $M = \langle f_1, f_2 \rangle$ are mutually exclusive fluents if there is no state s that may belong to a solution path such that $M \subseteq s$.*

The standard definition of mutex refers to tuples of facts that cannot be reached from the initial state. Definition 4.1 is more general since it also considers states that cannot reach the goal. Moreover, our general definition may include pairs of facts that can be proven not to be part of any plan, even if we do not know for sure whether states are unreachable or dead-ends.

Thus, we distinguish two different types of mutexes, depending whether they were discovered in *forward* or *backward* direction. Intuitively, a *forward* mutex is a pair of fluents that cannot simultaneously be true in any reachable state. A *backward* mutex is a pair of fluents that cannot simultaneously be true in any state that can reach the goal, i. e., any state which is not a dead end. Therefore, while forward mutexes are useful to prune backward search (because they cannot be reached from the initial state), backward mutexes detect dead ends in forward search.

For example, consider a task of a simplified variant of the *Trucks* domain, a typical logistics domain with time constraints in which packages must be delivered with trucks before a given time.

[1]Previous works have also considered not only pairs but sets of $m \geq 2$ fluents that cannot simultaneously appear in the same state (Alcázar, 2014), even though experimental results are usually restricted to $m = 2$. In this thesis, for simplicity, we consider mutex pairs even though our results could be extended to the general case.



Package:	$v_p = \langle A, B, C, T \rangle$
Truck:	$v_T = \langle A, B, C \rangle$
Time:	$v_t = \langle 0, 1, 2, 3, 4 \rangle$

Figure 4.1: Mutex examples in the *Trucks* domain. Initially, the truck and the package are at location A and the goal is to deliver the package at C before time expires.

Our task, shown in Figure 4.1, has a package, a truck and three locations. The variables are the position of the package v_p, the position of the truck, v_T, and the current time v_t. The goal is to deliver the package ($v_p = C$) before a given time $t \leq 4$. The operators are load or unload a package and move the truck. All actions take a time step.

In this domain, we can find forward and backward mutexes. An example of forward mutex is that at time 1 the truck cannot be at location C. It is impossible since the truck is at location A at time 0 so at location 1 it can only be at A or B. Similarly, an example of backward mutex is that, at time 3, the package cannot be at B, since there would be no remaining time to deliver the package.

The most common method to find mutexes is the h^2 heuristic (Haslum and Geffner, 2000). h^2 performs a reachability analysis in P^2 (Haslum, 2009), a semi-relaxed version of the original problem in which the atoms are actually pairs of fluents. For each pair of fluents, a lower bound for the cost of reaching that tuple is computed. Pairs with an infinite h^2-value are unreachable in P^2 and, therefore, are mutexes.

Another method for finding mutexes is the monotonicity analysis generally employed to generate a finite-domain representation of the problem (Helmert, 2009). This monotonicity analysis ensures that the number of fluents true at the same time that belong to a set $M_a = \{f_0, f_1, \dots, f_n\}$ can never increase. Hence, if the number of fluents of M_a true in the initial state is 1 (formally, $|M_a \cap s_0| = 1$), then all pairs of fluents of M_a are mutexes ($\forall p_m = \{f_i, f_j \mid f_i, f_j \in M_a \wedge f_i \neq f_j\} : p_m$ is a mutex).

In terms of efficiency, computing the monotonicity analysis is in most cases much more efficient that computing h^2, as it works exclusively with the domain definition and the initial state. However, the set of mutexes found by the monotonicity analysis is a strict subset of the set of mutexes found by h^2 (Alcázar, 2014). For this reason, in this thesis we will use h^2 to compute mutexes.

h^2 can be computed backwards too in order to find backward mutexes (Haslum, 2008). This is done by reversing the domain as proposed by (Massey, 1999) and performing a backwards reachability analysis in P^2 after disambiguating the goal to deduce which fluents are false in $s_\star$ with mutexes computed forward.

4.1.2 Invariant Groups

Mutexes are not the only state invariants that have been used to prune spurious states (Alcázar et al., 2013). Invariant groups are sets of fluents related to some invariant of the problem. The aforementioned monotonicity analysis was devised to find *"at-most-1"* invariant groups to use them as variables of a finite-domain representation (Helmert, 2009). These *"at-most-1"* invariant groups can be derived from any binary mutex computation method: any set $M_a = \{f_0, f_1, \dots, f_n\}$ such that $\forall p_m \in \{f_i, f_j \mid f_i, f_j \in M_a \wedge f_i \neq f_j\}\ \ p_m$ is mutex is an *"at-most-1"* invariant group. Thus, *a priori* these invariant groups do not offer more information than taking into account individual binary mutexes.

Nevertheless, *"at-most-1"* invariant groups can also be used to derive stronger constraints. In particular, by performing a reachability analysis considering the negation of all facts in the group, one can automatically infer that an *"at-most-1"* invariant group is in fact an *"exactly-1"* invariant group. *"exactly-1"* invariant groups are strictly more constrained than *"at-most-1"* invariant groups as one can infer an additional logical constraint from them: if $M_a = \{f_0, f_1, \ldots, f_n\}$ is an *"exactly-1"* invariant group then $(f_0 \vee f_1 \vee \ldots \vee f_n)$ is a state invariant of the problem. This constraint represents the lower bound of 1 that defines the *"at-least-1"* property of the invariant group.

Note that all the variables $v \in \mathcal{V}$ of finite-domain representations of problems are *"exactly-1"* invariant groups, but not all the *"exactly-1"* invariant groups of the problem need to be variables. As an example consider *Blocksworld*, where there are invariant groups stating that each block can only rest on one other block or the table at any given time, and others stating that only one other block can rest on a given block (or it is empty). However, usually only one of these sets is represented as a variable, as otherwise two variables would have a value corresponding to the same fluent. Also note that in some problems there may be no *"exactly-1"* invariant groups or they may not be useful for their usage as variables; if this is the case regular *"at-most-1"* invariant groups can be completed with an additional fluent representing that none of the facts in the group is true to turn them into *"exactly-1"* invariant groups (Helmert, 2006b). From this point on whenever the term "invariant group" is used, we will mean *"exactly-1"* invariant group unless otherwise specified.

4.1.3 Pruning Invalid Operators

The use of state invariants is not limited to prune states in the search. Any partial state (e. g., preconditions of operators) can be detected as *invalid*, i. e., contradictory with the state invariant constraints. The disambiguation of a partial state consists of determining whether there is a valid assignment for variables with undefined value that is compatible with all the state invariants (Alcázar et al., 2013). Disambiguation is performed by solving a CSP using the facts that have undefined value as variables and the state invariants as constraints. If there is no valid assignment then the partial state is not valid. When only one possible assignment is valid for some variable, then the partial state may be "completed" adding an additional fluent (Alcázar, 2014).

We say that an operator is *invalid* if it contradicts the state-invariants constraints. Invalid operators are never applicable in reachable states or the result is a dead-end state. When grounding a planning instance, *invalid* operators may be instantiated if done naïvely. Most planners prune instantiated operators with unreachable fluents in their preconditions, but no additional method is used to detect them. Even if these operators do not generate spurious states in progression because they are never applicable, they may have a negative impact when combined with other techniques, like regression, abstractions and delete-relaxation heuristics.

The forward computation of h^2 used to discover mutexes can also find that an operator is invalid if its preconditions are unreachable by the heuristic. Similarly, the backward computation of h^2 can discover other invalid operators that necessarily lead to dead-end states. Thus, as a side effect of the computation of h^2, we remove invalid operators from the planning task. Moreover, we disambiguate the preconditions of all the instantiated operators and remove those that are invalid.

4.1.4 Preprocessing Algorithm to Discover State Invariants

Before starting the search, we use a preprocessing algorithm to compute state invariants and simplify the task (Alcázar and Torralba, 2015). The problem is simplified by removing all operators detected as invalid as well as any invalid fluent that is unreachable or irrelevant. Moreover, it performs a fixpoint computation, where state invariants discovered in one iteration are used to strength the reasoning of the algorithm in subsequent iterations. Our preprocessing phase is described in Algorithm 4.1.

Algorithm 4.1: Fixpoint computation of invariants.

```
Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
Output: Updated planning problem: $\Pi'$
Output: Sets of state invariants: $M_{fw}$, $M_{bw}$, $Ex_1$
1  $Ex_1, M_{fw} \leftarrow$ MonotonicityAnalysis($\Pi$)
2  $M_{bw} \leftarrow \emptyset$
3  $fw \leftarrow \top$
4  $update_{fw} \leftarrow \top$
5  $update_{bw} \leftarrow \top$
6  $\mathcal{O}' \leftarrow \mathcal{O}$
7  while $update_{fw} \vee update_{bw}$ do
8      if $fw \wedge update_{fw}$ then
9          $M'_{fw}, \mathcal{O} \leftarrow$ ComputeForwardH2($\Pi, M_{bw}, \mathcal{O}'$)
10         if $M'_{fw} \neq M_{fw}$ then
11             $M_{fw} \leftarrow M'_{fw}$
12             $update_{bw} \leftarrow \top$
13     else if $\neg fw \wedge update_{bw}$ then
14         $M'_{bw}, \mathcal{O} \leftarrow$ ComputeBackwardH2($\Pi, M_{fw}, \mathcal{O}'$)
15         if $M'_{bw} \neq M_{bw}$ then
16             $M_{bw} \leftarrow M'_{bw}$
17             $update_{fw} \leftarrow \top$
18     $update_{fw}, update_{bw}, \mathcal{O}' \leftarrow$ DisambiguateActions($\mathcal{O}, M_{fw}, M_{bw}$)
19     $fw \leftarrow \neg fw$
20 $\mathcal{V} \leftarrow$ SimplifyVariables($\mathcal{V}, M_{fw}, M_{bw}, Ex_1$)
21 return $\langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle, M_{fw}, M_{bw}, Ex_1$
```

First the algorithm computes the monotonicity analysis commonly used to define the finite-domain representation of the planning task (Helmert, 2009). This analysis infers an initial set of forward mutex and *"exactly-1"* invariant groups (line 1). Then, the algorithm performs several iterations alternating the forward and backward computation of h^2 (lines 9 and 14). `ComputeForwardH2` and `ComputeBackwardH2` compute the set of mutexes according to the h^2 heuristic in progression and regression. This computation is strengthened with the invariants computed by previous iterations in two different ways:

1. Mutexes from the opposite direction are pruned in the reachability analysis performed by h^2.

2. Instead of considering the original set of operators, h^2 uses the set of disambiguated operators, $\mathcal{O}$', that include additional preconditions inferred from the state invariants.

As mentioned in Section 4.1.3 on page 56, as a side effect h^2 removes invalid operators, returning the subset of operators that remain valid. The `DisambiguateActions` function disambiguates the preconditions and effects of the operators, deducing additional preconditions. The operators with additional preconditions ($\mathcal{O}'$) are used in the following computations of h^2 to infer additional constraints. However, the additional preconditions are not introduced in the operators of the planning task, $\mathcal{O}$.

Whenever new mutexes are found in the opposite direction or the operator disambiguation modifies the operators, h^2 is recomputed in order to find new mutexes with the additional constraints. Thus, the main loop is repeated until a fixpoint is reached. At the end, `SimplifyVariables` removes the variable values that cannot occur in any valid state and removes any variable having a single value afterwards.

4.2 Use of State Invariants in Partial-State Regression Search

State invariants have been used in partial-state heuristic regression search since the very beginnings (Bonet and Geffner, 2001). Every generated state that violates a constraint is pruned from the search. State invariants allow pruning spurious states after they are generated. However, more efficient alternatives that avoid the generation of spurious states exist. For instance, the use of *e-deletion* (Vidal and Geffner, 2005), another invariant of the problem, avoids the generation of spurious states in explicit-state search by modifying the definition of applicability in regression (Alcázar et al., 2013).

An operator o *e-deletes* a fluent f if f must be false in every state resulting from the execution of a sequence of operators whose last operator is o. Hence, an operator o *e-deletes* a fluent f if f is mutex with $\mathit{eff}(o)$ or $\mathit{prev}(o)$.

For example, in the *Blocksworld* domain the action *(stack b c) e-deletes (on a b)* because it adds *(clear b)*, which is mutex with *(on a b)*. This can be inferred even if *(on a b)* is not mentioned by *(stack b c)*. In multi-valued representations, deleting a fluent f means changing the value of the variable $v \in \mathcal{V}$ it corresponds to, which is equivalent to adding a fluent mutex with f. Hence, the first case is a particular instance of the third case in multi-valued representations.

To avoid the generation of spurious states from a given state s in regression, one must make sure not to use an operator that *e-deletes* some fluent $f \in s$ to generate a successor state. Formally, if $e\text{-}del(o)$ is the set of fluents *e-deleted* by an operator $o \in \mathcal{O}$, o is not applicable in regression in a partial state s if $e\text{-}del(o) \cap s \neq \emptyset$. An intuitive way of understanding the concept of *e-deletion* in explicit-state regression is to consider the set of fluents $e\text{-}del(o)$ as negative preconditions of o in regression. *e-deletion* can also be used in abstract state spaces, as will be commented later in Section 4.6, after presenting how the concept of *e-deletion* can be adapted for symbolic search in Section 4.4.

4.3 Encoding State Invariants as BDDs

Pruning spurious states has been considered essential in heuristic backward search since long ago (Bonet and Geffner, 2001). Binary mutexes allow pruning invalid states that otherwise would be considered for expansion during search. Expanding such states may lead to an exponential decrease in performance, as none of the successors of an invalid state may lead to a plan. The use of mutexes in explicit-state search is straightforward: simply prune every state s such that fluents $f_i, f_j \in s$ are mutex.

Despite the impact that the use of state invariants has in explicit-state regression, this technique has not been employed in symbolic search. Although it is obvious that a *per state* application of mutexes in symbolic search is not practical, there are alternatives. In particular, we propose *creating a BDD that represents in a succinct way all the states that would be pruned if state invariants were used*. This BDD, that we call the *constraint BDD* ($cBDD$), can be used to discard all the invalid states that have been generated during the search in a similar way as it is done with the closed list for duplicate detection purposes.

The $cBDD$ is created in the following way. Every binary mutex is a conjunction of fluents f_i, f_j ($f_i \neq f_j$) such that, if $f_i, f_j \in s$, then state s is invalid. Hence, the set of states that can be pruned using mutexes consists of those in which at least one such conjunction of fluents is true. This way, the logical expression represented by $cBDD$ is the disjunction of all the mutexes found with h^2 (represented by the conjunction of both fluents).

Constraints derived from *"exactly-1"* invariant groups are encoded in a similar way. Given an *"exactly-1"* invariant group $\theta = f_1, \dots, f_k$, two types of constraints may be deduced: first, the set of all the mutexes of the form $f_i \wedge f_j$ if $f_i \neq f_j$; second, the fact that at least one fluent $f_i \in \theta$ must be true in every valid state. The first constraint overlaps with the mutexes computed with h^2, so it is not necessary to consider it again. The latter however can be encoded as an additional *"at-least-1"* constraint of the form $\neg(f_1 \vee f_2 \vee \dots \vee f_k)$ for every *"exactly-1"* invariant group.

Formally, if M is the set of binary mutexes found by h^2 and I_g the set *"exactly-1"* invariant groups:

$$cBDD = \left(\bigvee_{\langle f_i, f_j \rangle \in M} f_i \wedge f_j \right) \vee \left(\bigvee_{\langle f_1, \dots, f_k \rangle \in I_g} \neg f_1 \wedge \dots \wedge \neg f_k \right)$$

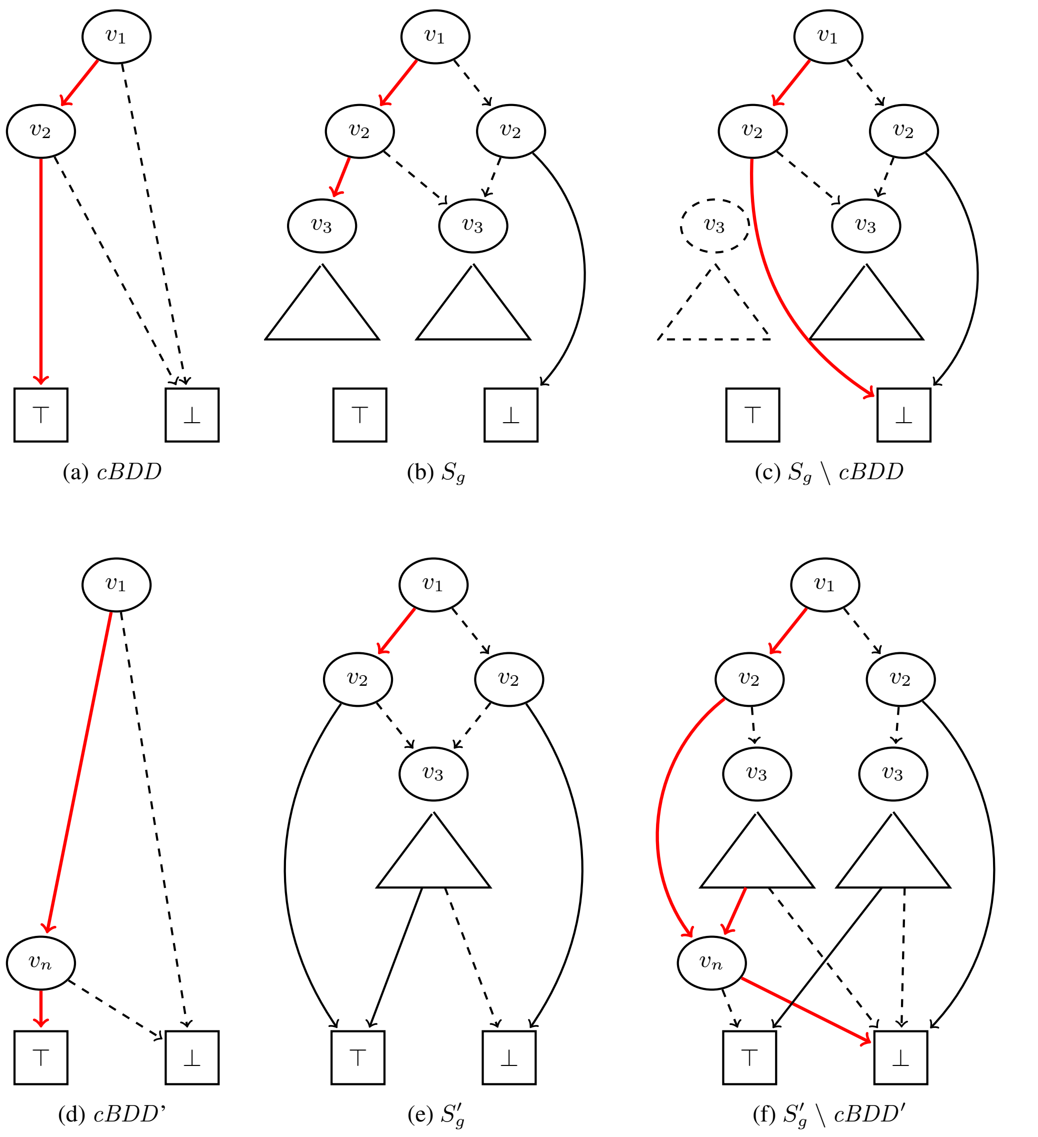
Even though each mutex and *"at-least-1"* constraint is efficiently representable with only one node per fluent, the size of $cBDD$ is exponential in the number of encoded constraints in the worst case. This follows from the results provided by Edelkamp and Kissmann (2011) on the complexity of the representation of partial states in BDDs. To ensure that we can represent $cBDD$, we use a disjunctive partitioning, dividing $cBDD$ into k BDDs: $cBDD = cBDD_1 \vee cBDD_2 \vee \dots \vee cBDD_k$.

To obtain an efficient partitioning, we follow Algorithm 3.2, *Aggregate*, described on page 44 that computes a disjunctive partitioning of a set of BDDs. The algorithm is initialized with the BDDs of individual constraints, and they are aggregated until they are larger than a given threshold. The order in which these disjunctions are applied affects the efficiency of the procedure and the number of BDDs used to represent $cBDD$. First, for each variable $v_i \in \mathcal{V}$ we compute a BDD describing all binary mutexes of fluents relative to both v_i and v_j with $j > i$. Then, we iteratively merge the BDDs of mutex constraints related to each variable, plus the *"at-least-1"* constraints.

Invalid states determined by state invariants are pruned by computing the difference $S_g \setminus cBDD$ of a newly generated set of states S_g with the constraint BDD $cBDD$. In terms of BDD manipulation this is done by computing the logical conjunction of S_g with the negation of $cBDD$: $S_g \wedge \neg cBDD$. This operation is the same as the one done in symbolic search with the BDD that represents the set of closed states, used to prune duplicates. Extending the operation to the case where we have more than one $cBDD$ is straightforward: $S_g \wedge \neg cBDD_1 \wedge \dots \wedge \neg cBDD_k$. In the rest of the section we assume that $cBDD$ is represented in a single BDD, without loss of generality.

An important remark about the usefulness of pruning states with $cBDD$ is necessary, though. In general, the size of a BDD is not proportional to the number of states it represents. This means that there is no guarantee that pruning invalid states will help in symbolic search, as opposed to the explicit-state case. Figure 4.2 exemplifies this with two cases: one positive in which an arbitrarily large part of the BDD is pruned, and one negative in which most of the BDD gets duplicated.

Figure 4.2b shows a BDD S_g that contains invalid states that violate the mutex $(v_1 \wedge v_2)$, represented by the BDD in Figure 4.2a. In this case, every state represented by any path that satisfies $v_1 \wedge v_2$ could be safely pruned. In practice, this means that the subgraph whose root is the node that corresponds to v_3 down the path $v_1 \wedge v_2$ can be removed, with a significant potential to reduce the



(a) *cBDD* (b) S_g (c) $S_g \setminus cBDD$

(d) *cBDD*' (e) S'_g (f) $S'_g \setminus cBDD'$

Figure 4.2: Example of applying a mutex constraint to a BDD that represents a set of states. The upper row corresponds to a positive example, in which $S_g \setminus cBDD$ is smaller than S_g. The lower row corresponds to a negative example, in which $S'_g \setminus cBDD'$ is bigger than S'_g. Following the solid edge corresponds to reaching the high successor; following the dashed one corresponds to reaching the low successor.

size of S_g. If we look at Figure 4.2c, we can see that computing the difference $S_g \setminus cBDD$ has this precise effect, as evaluating v_2 after following the path that satisfies $v_1 \wedge v_2$ leads directly to the *false* sink node.

Again, there is no guarantee that the resulting $S_g \setminus cBDD$ will be smaller: first, computing $S_g \setminus cBDD$ may increase the size of the BDD if the added constraints make reference to variables in $\mathcal{V}$ that did not appear initially in S_g; second, as the "removed" subgraph is not necessarily isolated —in the sense that it may overlap with other subgraphs of S_g— the gain in compactness obtained from merging nodes of the "removed" subgraph with other nodes of S_g may be lost.

Another factor that might have negative impact is the variable ordering, when the variables of $cBDD$ are not close together, like in Figure 4.2d. In this case the subgraph in S'_g whose root corresponds to v_3 must be split in two to represent that it actually matters whether it is reached by a path containing v_1 or $\neg v_1$, as there is a constraint relevant to v_1.

Of course, when multiple constraints are involved, estimating the increase in compactness obtained by using $cBDD$ is more complex. For example, it may happen that $cBDD$ is not representable within the available memory. In that case, if all the constraints are relative to non-goal variables, it is not feasible to remove all invalid states from the goal description, because the result would be at least as large as $cBDD$. The negative impact of computing the conjunction of $S_\star$ with $cBDD$ would be such that the backward search would not even start. Alternatives to the computation of the difference $S_g \setminus cBDD$ will be analyzed in Section 4.5.

Finally, a clarification about the source of the binary mutexes is needed. As described in Section 4.1.1, mutexes can be computed in forward and backward direction. Forward mutexes may be violated only by backward search, and backward mutexes may be violated only by forward search. Consequently, it is not useful to encode both types of mutexes in the same BDD, as they should be exploited only in the appropriate direction. For this reason we will use two constraint BDDs, one for forward search and another one for backward search. Analogously, the constraints derived from the *"exactly-1"* invariant groups are obtained from a forward reachability method, the monotonicity analysis. Hence, such constraints should be encoded only in the $cBDD$ used by the backward search. For simplicity, throughout this work we will use $cBDD$ to refer to either constraint BDD and we will assume that the corresponding BDD is used in the appropriate cases.

4.4 Encoding Constraints in the TRs

Encoding the state invariants in the $cBDD$ allows the search to prune invalid states after they are generated. However, as mentioned in Section 4.2, in explicit-state regression *e-deletion* can be used to avoid the generation of spurious states (rather than pruning them after their generation). Inspired by *e-deletion*, in this section we consider whether constraints can be directly encoded in the TRs to avoid the generation of invalid states.

Note however, that the conditions of *e-deletion* are not directly applicable in our case due to differences in the representation. *e-deletion* constraints are applied as negative preconditions of the operators in regression because partial states in explicit-state regression search leave the value of some variables undefined. Therefore a partial state implicitly considers states that violate invariants related to undefined variables and the additional preconditions avoid applying operators to those states. In symbolic search, however, we want to prune as many invalid states as possible, so that BDDs involved in the search do not contain any state detected as invalid. Given that the set of states being expanded does not contain invalid states, encoding the additional *e-deletion* constraints as preconditions in the regression search is redundant. Furthermore, *e-deletion* does not guarantee that the resulting set of states does not contain invalid states.

Our encoding takes as input an operator and a set of forward and backward constraints (i. e., logical formulas such as mutexes or *"at-least-1"* constraints that hold in every valid state) and define a modified operator so that it does not generate any invalid state. However, not all constraints are encoded in every operator. We first identify which constraints must be encoded in each operator $o \in \mathcal{O}$. This comes from the fact that, although replicating all the constraints in the TRs is possible, this may lead to a great degree of redundancy. We show that, by assuming that the set of states to be expanded does not contain invalid states, we can safely consider only a subset of constraints in each TR while avoiding the generation of invalid states. Therefore, we denote a constraint as relevant for an operator if it may become violated after the application of the operator.

Definition 4.2. *(Relevant Constraint) A backward constraint c is* relevant in progression *for an operator o if and only if a state s exists such that $s \models c$ and $o(s) \not\models c$.*

A forward constraint c is relevant in regression *for an operator o if and only if a state s exists such that $o(s) \models c$ and $s \not\models c$.*

This definition of relevant constraints allows us to define the *constrained* version of an operator. To avoid the generation of invalid states, the relevant constraints for progression must be encoded as negative effects and the relevant constraints for regression must be encoded as negative preconditions. This ensures that the encoded constraints are not violated after the *image* and *pre-image* computation respectively, so that no state that violates the constraints is generated during the symbolic exploration.

As an example, consider Figure 4.1 on page 55. Consider the operator that moves the truck from A to B at time step 0, *move*$(A, B, 0)$. A relevant *"at-most-1"* constraint in progression for this operator is $\neg(v_t = 1 \wedge v_p = A)$. This mutex constraint holds because there would be no time to deliver the package before time 4. It is relevant because there exists at least one state such that we make the constraint false by applying the operator. For example, if we apply the operator in the initial state, with both the package and the truck in location A, we reach a state in which the package is at A and $v_t = 1$. A constraint that is irrelevant for the operator is $\neg(v_t = 2 \wedge v_p = A)$ which is also a constraint of the problem by the same argument. However, our move operator is not directly related either to $v_t = 2$ or $v_p = A$ so it cannot produce a state that violates that mutex constraint unless the constraint already holds in the predecessor state.

We identify the conditions for a constraint to be relevant for an operator below in Propositions 4.2, 4.3, and 4.5, but first, we define how relevant constraints are used to avoid the generation of invalid states during the search. Forward constraints are encoded in the preconditions of the operator to avoid the generation of invalid states when the operator is applied in regression. This slightly abuses the notation because forward constraints are arbitrary logical formulas and we defined $pre(o)$ as a set of fluents. However, the meaning in both cases is that they must hold in any state in which the operator is applicable, and hence, any state generated by the operator in regression. Adding the forward constraints as preconditions does not affect the semantics of the operator in forward search, since any state reachable from the initial state necessarily satisfies the constraints. Moreover, since the the operator is represented as a logical formula (the TRs, see page 39) in symbolic search, encoding additional formulas in the preconditions is straightforward.

Backward constraints are encoded in a similar way. To do so, we define the notion of postconditions of an operator. The *postconditions* of an operator o, $post(o)$, are conditions that must hold in any generated state after the execution of the operator. By definition, all the prevails and effects of the operator are postconditions, $prev(o) \cup \mathit{eff}(o) \subseteq post(o)$. This conditions can indeed be transformed into prevail conditions by subtracting the effects. For example, if $\neg(p \wedge q)$ is a postcondition of o, $p \in \mathit{eff}(o)$, and $q \notin delo$ then we may deduce $\neg q \in prev(o)$. Expressing the constraints over the operators in terms of postconditions simplifies the exposition. Definition 4.3 states how to encode forward and backward constraints in the operators.

Definition 4.3. *(Constrained Operator) Let $o \in \mathcal{O}$ be an operator and let C_o^{fw} and C_o^{bw} be the sets of constraints relevant with respect to o in progression and regression respectively.*

Then the constrained operator o^c derived from o becomes o with extra preconditions $pre(o^c) = pre(o) \wedge \bigwedge_{c_i \in C_o^{bw}} c_i$ and postconditions $post(o^c) = post(o) \wedge \bigwedge_{c_i \in C_o^{fw}} c_i$.

Using o^c for all $o \in \mathcal{O}$ suffices to guarantee that the successor set does not contain invalid states as long as the source set of states does not contain invalid states. We divide our statement in Theorems 4.4 and 4.1, that prove it for the case of progression and regression, respectively.

4.4.1 Relevant Constraints in Regression

Theorem 4.1. *Let $S \subseteq \neg cBDD$ be a state set that does not contain states detected as invalid and o^c the constrained version of an operator $o \in \mathcal{O}$. Let S' be the resulting state set from applying o^c in regression to S. Then $S' \subseteq \neg cBDD$, i. e., it does not contain states that can be detected as invalid.*

Proof. Proof by contradiction. Suppose there is a state $s' \in S'$ that violates some constraint c. As c was satisfied by every state in S, by definition c is a relevant constraint for o in regression. Then $c \in pre(o^c)$, so o is not applicable on s'. □

Next, Propositions 4.2 and 4.3 show the sufficient and necessary conditions for mutexes and *"at-least-1"* groups to be relevant in regression. Essentially the relevant constraints are those that contain fluents that may be *added* or *removed* when o is applied in regression.

Proposition 4.2. *(Relevant forward mutex in regression) Let M be a mutex of size m, $M = \neg(f_1 \wedge f_2 \wedge \dots \wedge f_m)$. Let $o \in \mathcal{O}$ be a valid operator and its* set of undefined preconditions $\mathcal{V}_u(o) = \mathcal{V}_{\mathit{eff}(o)} \setminus \mathcal{V}_{pre(o)}$.

Then M is relevant in regression for operator o if and only if:

1. *M is not implied by the preconditions of the operator $pre(o)$.*
2. *For some fluent $f_i = \langle v_i, x\rangle \in M$, either (2a) $f_i \in pre(o) \setminus prev(o)$ or (2b) $v_i \in \mathcal{V}_u(o) \wedge f_i \notin post(o)$.*

Proof. To prove the *if* part of the statement, we show that if the conditions (1) and (2) hold then, there exists a state s such that (i) $s \not\models M$ and (ii) o is applicable in s and (iii) $o(s) \models M$. We define s as any state such that $f_k \in s$ for all $f_k \in M$ and the rest of fluents compatible with $pre(o)$ (such an assignment exists when the operator is valid and condition (1) holds). Then s satisfies the three statements:

i $s \not\models M$: By definition $s \not\models M$, since all $f_k \in M$ are true in s.

ii o is applicable in s: By the construction of s, the operator preconditions are not contradictory with the value of variables other than v_k for $f_k \in M$. If any fluent $f_j \in M$ contradicts the preconditions of o, then condition (1) leads to contradiction.

iii $o(s) \models M$: Necessarily, due to rule (2), $f_i' = \langle v_i, y\rangle$ is true in $o(s)$, where $f_i = \langle v_i, x\rangle$ and $x \neq y$. We have two cases (2a) or (2b). If (2b) holds, this is automatic, since $f_i \notin post(o)$ and $v_i \in \mathcal{V}_{\mathit{eff}(o)}$. If (2a) holds, $f_i \in pre(o)$ implies that o deletes f_i (because f_i would be in $prev(o)$ instead). Therefore, f_i is false in $o(s)$ and $o(s) \models M$.

To prove the *only if* case, note that if (1) does not hold, then there does not exist s such that $s \not\models M$ and o is applicable in s, so M is not relevant for o. We may deduce condition (2) assuming that M is relevant for o in regression, i. e., there exists a state s such that $M \models o(s)$ and $M \not\models s$. Then, for some fluent $f_i = \langle v_i, x\rangle$, $f_i \notin o(s)$, $f_i \in s$. Therefore, $s[v_i] \neq o(s)[v_i]$, which implies $v_i \in \mathcal{V}_{\mathit{eff}(o)} \land f_i \notin \mathit{eff}(o)[v_i]$. Two cases are possible with respect to the preconditions of o, that correspond to conditions (2a) and (2b):

(a) $v_i \in \mathcal{V}_{pre(o)}$. As o must be applicable in s, $f_i \in pre(o)$.

(b) $v_i \notin \mathcal{V}_{pre(o)}$ and $v_i \in \mathcal{V}_u(o)$ immediately follows.

□

Note that condition (1) of proposition 4.2 only means that the constraint is redundant because it is already required by the pre- (in regression), post- (in progression) or prevail-conditions (in both) of the operator. If a mutex constraint that violates condition (1) is encoded in the operator, it does not modify the operator since it is a redundant constraint. This is not the case for condition (2) that identifies constraints that are not needed when assuming that the predecessor set of states does not contain invalid states but they are not redundant and could change the semantics of the operator if they were encoded in it.

Proposition 4.3. *(Relevant* "at-least-1" *invariant in regression) Let M_{inv} be an* "at-least-1" *invariant, $M_{inv} = f_1 \lor f_2 \lor \cdots \lor f_k$, and let $o \in \mathcal{O}$ be a valid operator.*
Then M_{inv} is relevant in regression for o if and only if:

1. *M_{inv} is not implied by the preconditions of the operator $pre(o)$.*

2. *For some fluent $f_i = \langle v_i, x\rangle \in M_{inv}$, $f_i \in \mathit{eff}(o)$.*

Proof. To prove the *if* part of the statement, we show that if the conditions (1) and (2) hold then, there exists a state s such that (i) $s \not\models M_{inv}$ and (ii) o is applicable in s and (iii) $o(s) \models M_{inv}$. We define s as any state such that $f_k \notin s$ for all $f_k \in M_{inv}$ and the rest of fluents compatible with $pre(o)$ (such an assignment exists when the operator is valid). Then s satisfies the three statements:

i $s \not\models M_{inv}$: By definition $s \not\models M_{inv}$, since all $f_k \in M_{inv}$ are false in s.

ii o is applicable in s: By the construction of s, the operator preconditions are not contradictory with the value of variables other than v_k for $f_k \in M_{inv}$. Finally if any fluent $f_j \in M_{inv}$ contradicts the preconditions of o, then condition (1) leads to contradiction.

iii $o(s) \models M_{inv}$: Necessarily $f_i \in o(s)$ since $f_i \in \mathit{eff}(o)$ due to rule (2).

To prove the *only if* case, note that if (1) does not hold, then there does not exist s such that $s \not\models M_{inv}$ and o is applicable in s, so M is not relevant for o. We may deduce condition (2) from M_{inv} being relevant in regression for o, i. e., is satisfied in $o(s)$ but not in s. Then, for some fluent $f_i = \langle v_i, x\rangle \in M_{inv}$, $f_i \notin s$, $f_i \in o(s)$. Therefore, $s[v_i] \neq s'[v_i] = x$, which implies $\mathit{eff}(o)[v_i] = x$ and, therefore, $f_i \in \mathit{eff}(o)$. □

4.4.2 Relevant Constraints in Progression

Theorem 4.4. *Let $S \subseteq \neg cBDD$ be a state set that does not contain invalid states and o^c the constrained version of an operator o. Let S' be the resulting state set from applying o^c in progression to S. Then $S' \subseteq \neg cBDD$, i. e., it does not contain invalid states.*

Proof. Proof by contradiction. Suppose there is a state $s' \in S'$ that violates some constraint c. As c was satisfied by every state in S, by definition c is a relevant constraint for o in progression. Then $c \in post(o^c)$, so s' cannot be the result of applying operator o to any state. □

Next, Proposition 4.5 shows the sufficient and necessary conditions for mutexes to be relevant in progression. *"at-least-1"* groups cannot possibly be relevant in progression, since we only infer them with a forward reachability analysis.

Proposition 4.5. *(Relevant backward mutex in progression) Let M be a backward mutex of size m, $M = \neg(f_1 \wedge f_2 \wedge \cdots \wedge f_m)$. Let $o \in \mathcal{O}$ be a valid operator.*
Then M is relevant for operator o in progression if and only if:

1. *M is not implied by the postconditions of the operator, $post(o)$.*
2. *For some fluent $f_i = \langle v_i, x\rangle \in M$, $f_i \in eff(o)$.*

Proof. To prove the *if* part of the statement, we show that if the conditions (1) and (2) hold then, there exists a state s such that (i) $s \models M$ and (ii) o is applicable in s and (iii) $o(s) \not\models M$.

We define s as any state such that $f_i \notin s$ and $f_k \in s$ for all $k \neq i$ s.t. $f_k \in M$ and $f_k \notin post(o)$ and the rest of fluents are compatible with $pre(o)$ (such an assignment exists when the operator is valid and condition (1) holds). Then s satisfies the three statements:

i $s \models M$: By definition $s \models M$, since the fluent $f_i \in M$ is false in s.

ii o is applicable in s: By the construction of s, the operator preconditions are not contradictory with the value of variables other than v_k for $f_k \in M$. As $f_i \in eff(o)$, the precondition of o accepts a value for v_i not equal to f_i. The same holds for any f_j such that $f_j \in eff(o)$. Finally, for $f_k \in M, f_k \notin eff(o)$, f_k is compatible with $prev(o)$ by condition (1) since $prev(o) \subseteq post(o)$.

iii $o(s) \not\models M$: Necessarily, due to rule (2), f_i is true in $o(s)$. Every other $f_k \in M$ is true in $o(s)$ because either it belongs to $eff(o)$ or it was already in s. Condition (1) ensures that no fluent f_k already in s is deleted by o.

To prove the *only if* case, note that if (1) does not hold, then there does not exist s such that $o(s) \not\models M$, so M is not relevant for o in progression. We may deduce condition (2) assuming that M is relevant for o in progression, that is, M is satisfied in s but not in $o(s)$. Then, for some fluent $f_i = \langle v_i, x\rangle$, $f_i \notin s$, $f_i \in o(s)$. Therefore, $s[v_i] \neq o(s)[v_i]$, which implies $v_i \in \mathcal{V}_{eff(o)}$. As $o(s)$ is the result of applying o in s, $eff(o)[v_i] = x$ and $f_i \in eff(o)$. □

Propositions 4.2, 4.3, and 4.5 identify under which conditions the constraints are relevant for an operator in progression and regression. They complete Definition 4.3 of a constrained operator. Recall that according to Theorems 4.1 and 4.4 constrained operators ensure that no invalid states are generated in the forward or backward search. As mentioned before, for this property to hold we must ensure that the parent BDD does not contain invalid states. In progression this is always the case, as in any solvable instance s_0 represents a single valid state. On the other hand, $s_\star$ may contain spurious states, as $s_\star$ is in most cases partially defined. In this case, to guarantee that $s_\star$ does not contain invalid states, we compute the difference $s_\star \setminus cBDD$ to remove them from the goal description. When constraints are encoded in the TRs, this difference must be computed only once, before starting the search. This means that there will be no further overhead in using $cBDD$, with the additional advantage that $cBDD$ can be discarded afterwards to free memory.

4.5 BDD Minimization

In symbolic search, the performance of the logical operations is often heavily linked to the size of the BDDs they operate with. Both memory and time (in terms of BDD manipulation) benefit from working with BDDs that succinctly represent a given Boolean function. The main motivation for using a *constraint BDD* is to remove invalid states so the BDDs that represent sets of states are smaller. However, computing the difference with the $cBDD$ does not guarantee that the resulting BDD will be smaller, as analyzed in Section 4.3.

In the literature, mainly in works published by the Model Checking community, several *"don't care" minimization* algorithms have been proposed (Coudert and Madre, 1990; McMillan, 1996; Hong et al., 1997). *"Don't care" minimization* aims to succinctly represent a given function when only part of it is relevant. They receive a *function BDD f* and an additional *constraint BDD c* (also called *restrict* or *care BDD*), and aim to find a BDD g of minimum size that represents f's function in an incompletely specified way, such that $g(s) = f(s)$ for all $s \in c$.

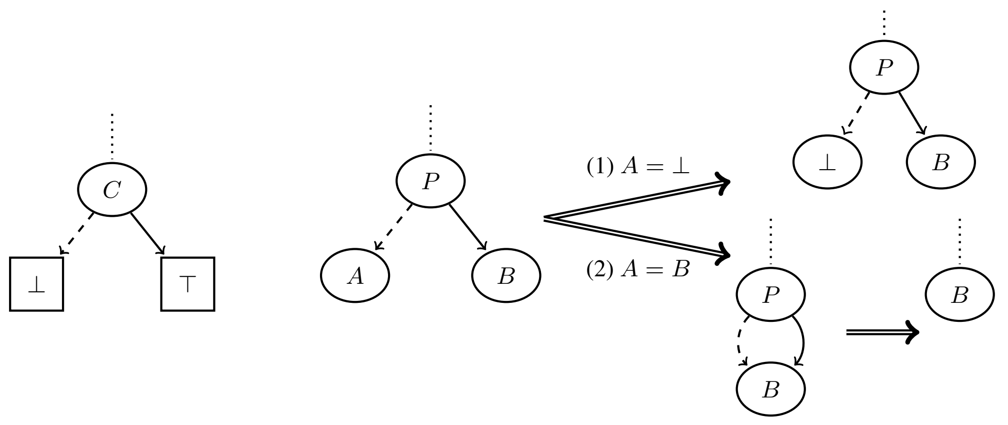
A plausible way of exploiting *"don't care" minimization* is to assume that f corresponds to BDDs that represent sets of states and that c may be any BDD that imposes some kind of restriction over f (Edelkamp and Helmert, 2001). In our case, both the complement (that is, the negation) of the *closed list BDD* and $cBDD$ can correspond to the definition of c. Hence, using minimization algorithms instead of the conjunction may prove useful, as it may be possible to obtain smaller BDDs. The intuition behind the use of minimization algorithms is to allow representing invalid states if this means that the BDDs will be smaller. These operations are more expensive to compute than the conjunction but, if g is smaller, the computation of a subsequent *image* and *pre-image* operation (which are the most expensive symbolic operations in most planning instances) may require less time.

Most BDD minimization algorithms are based on the concept of *sibling substitution*. Sibling substitution consist of replacing a node by its sibling, so that both become identical and their parent may be removed from the BDD, as illustrated in Figure 4.3. Sibling substitution requires identifying which nodes in f are *"don't care"* nodes, i. e., nodes in f such that the path that leads to them evaluates to *false* in c. In the example, the node A is a *"don't care"* node because the path from the BDD root to A evaluates to $\perp$ in $cBDD$. As we "don't care" about those values, node A can be substituted by anything we want. In the case of a typical conjunction, A is replaced by $\perp$. Sibling substitution replaces A by B instead, so that node P is not necessary anymore and the total number of nodes in the BDD decreases.

The following are the minimization algorithms considered in this work:

- *constrain* (Coudert and Madre, 1990): it performs sibling-substitution recursively, replacing nodes that correspond to *don't care* values. If g is larger than f, f is returned instead. A generalization of *constrain* was also proposed under the name of *generalized-cofactor* (McMillan, 1996).

- *restrict* (Coudert and Madre, 1990): refinement of *constrain* that ensures that, whenever a



(a) Care BDD (b) Conjunction $P \wedge \neg C$ (1) versus sibling substitution (2)

Figure 4.3: Example of sibling substitution. Consider an arbitrary path from the root node that leads to node C of the care BDD and node P of the other BDD. Then, node A belongs to the *don't care* set, so A can be substituted by B and their parent P is removed by BDD reduction rules.

node did not originally appear in f, both branches of c that stem from that node are merged. This guarantees that nodes that did not originally appear in f will not appear in g, although g may still be bigger.

- *leaf-identifying compaction* (Hong et al., 2000): it has two phases. First, it marks edges that can be redirected to a leaf node and edges whose pointed nodes are to be preserved from sibling substitution. Then, the result is obtained by redirecting edges to leaf nodes whenever possible and applying sibling substitution on non-marked edges. Thanks to the edge-marking phase, it ensures that g is smaller than f, as opposed to *restrict* and *constrain*.

- *non-polluting-and*: hybrid between regular *conjunction* and *restrict*. It performs a conjunction but, like in the *restrict* operation, whenever a node did not originally appear in f, both branches of c that stem from that node are merged.

In most algorithms, using *"don't care" minimization* instead of conjunction is straightforward. However, one must be careful to avoid introducing spurious paths in the state space, since some states are being expanded even if they have not really been reached. Introducing states from the closed list never supposes a problem, because these states were already expanded with lower cost so any new spurious path is not optimal. States in *cBDD*, however, are introduced in the search space even if they cannot be reached with that cost. In order to guarantee that these invalid states do not form part of any plan, we rely on the definition of forward and backward mutexes, which guarantees that any state violating them will not have a path to any goal and initial state, respectively. Thus, using *"don't care" minimization* in unidirectional search algorithms instead of conjunction cannot introduce any spurious path. However, in bidirectional search spurious paths may be introduced when *"don't care" minimization* is used in both search directions. This can happen when there exists a path between two invalid states each added in a different search direction. To avoid this scenario, we only use *"don't care" minimization* in the backward direction, which is the one with more mutexes available to prune the search. In the forward direction, we use the standard conjunction to remove all invalid states, avoiding the generation of spurious paths.

4.6 Constrained Symbolic Abstraction Heuristics

The use of regression is not limited to backward search in the original state space. For instance, PDBs (Culberson and Schaeffer, 1998) perform regression over the goals in an abstraction, α, of the original problem to create a lookup table that is used as an estimation of the optimal distance in the original problem. A detailed description of abstraction heuristics is provided in Chapter 6. PDBs in explicit-search that make use of mutexes are known as *constrained PDBs* (Haslum et al., 2005), cPDBs for short. cPDBs perform two types of pruning in the regression search over the abstract state space:

1. Prune all the abstract states that violate a state invariant of the problem.
2. Prune transitions $s^\alpha \xrightarrow{o} t^\alpha$ when the abstract states, s^α and t^α, are incompatible with the semantics of the operator,[2] i. e., when the operator can only be applied over invalid states in s^α or can only generate invalid states in t^α.

The use of pruning in cPDBs may simplify the regression search by reducing the size of the abstract state space. More importantly, cPDBs may potentially prune spurious paths, i. e., solution plans in the abstract state space that do not have a corresponding plan in the original problem. Thus, cPDBs are able to derive heuristics more informed than standard PDBs and are always as good as them.

As mentioned in Section 2.7, GAMER uses a symbolic version of PDBs for their use in symbolic search (Edelkamp, 2002; Kissmann and Edelkamp, 2011). In this work, we propose *symbolic constrained PDBs*, a constrained version of symbolic PDBs. In symbolic cPDBs, we exploit constraints in regression as studied in previous sections, either by encoding them in a $cBDD$ or by using constrained TRs. However, there are some subtleties related to the usage of constraints in an abstract search space that make it different from backward search in the original problem. Our contribution here is to analyze how the considerations that we made in the previous sections about the encoding of constraints in symbolic search apply in the case of PDBs.

The difference is that some variables of the problem are ignored in the abstract state space. Hence, we classify constraints into three classes depending if their variables are abstracted or not. *Full constraints* are entirely in the pattern, *partial constraints* have some fluents in the pattern and some abstracted away and *null constraints* are those without fluents in the pattern. These constraint types are related with the pruning methods mentioned before:

1. Full constraints can be used to prune invalid abstract states. Partial or null constraints cannot be used for these purposes because some of their fluents are abstracted so that they are never violated. As an example, take a mutex constraint $m = \neg(f_1 \wedge f_2)$. If we ignore the variable related to f_2, then we do not know whether it is true so that no abstract state will violate the constraint m.
2. Partial constraints can be used to prune transitions in which the semantics of the operator are incompatible with the abstract states. This corresponds to the second pruning method mentioned before.
3. Finally, null constraints are never useful for pruning the search. They could help to infer other constraints, but here we are assuming that the set of all constraints inferred is provided as input. Thus, in the following, we ignore this type of constraints and assume that all the constraints are either full or partial.

[2]The original work of (Haslum et al., 2005) considered only the case where o cannot be applied in regression on t^α. We extend here the definition to consider incompatibilities with s^α too.

Another remarkable difference (applicable to explicit-search too) is that, while mutexes discovered by backward h^2 are redundant in regression and do not provide additional pruning during backward search, in the abstract space those mutexes may be violated. Therefore, when working with abstractions all constraints must be used independently of the method used to infer them.

Encoding constraints as BDDs As in the case of symbolic backward search, the simplest method to use the constraints in symbolic search is to encode them in a BDD, $cBDD$. As explained in Section 4.3, $cBDD$ corresponds to the disjunction of all the constraints and represents the set of states that violate any of them. A similar $cBDD$ can be used in an abstract state space to prune invalid abstract states during the search. The only difference is that, as explained before, constraints that refer to a fluent outside the pattern should be ignored, because they are never violated by abstract states.

Using a $cBDD$, we accomplish the first pruning method proposed by Haslum et al. in explicit cPDBs. The second one, however, depends on the operator that is being applied, so that one has to individually check the operators that only support that transition for invalid states. However, as we explained in Section 3.4, operators in symbolic search may be merged, which makes it unfeasible to check them individually. Therefore, unlike the case of search in the original state space, the $cBDD$ method does not perform all the pruning possible.

Encoding constraints in the TRs Our second pruning method consists of encoding the constraints in the TRs to avoid the generation of invalid states (see Section 4.4). We identified which subset of constraints is necessary to encode in the TR of each operator to guarantee that no invalid state is generated. For this, we assumed that the predecessor set of states does not contain invalid states. Full constraints are treated as in the case of the original search, getting the same guarantees of not generating any invalid abstract states. This suffices to perform the first type of pruning, as with the $cBDD$ approach.

However, our assumption does not hold anymore for partial constraints in the abstract search. In PDB abstractions the value of abstracted variables is undefined, so abstract states correspond to partial states. Consider the set of states associated to an abstract state, S_i^α, that corresponds to all the possible assignments that the abstracted variables may take. Partial constraints may be violated for some states in S_i^α but not for others. S_i^α cannot be pruned without loss of admissibility because there exist valid states associated to it. As an example, take the constraint of our trucks example, $\neg(v_t = 1 \wedge v_p = A)$. If we ignore the time, an abstract state is composed of the position of the package and the position of the truck. Consider the abstract state s_0^α $\langle v_p = A, v_T = A\rangle$. Obviously we cannot prune s_0^α since it is the abstracted initial state. And still, for some values of the time variable v_t, the state would be invalid. This is what enables the second type of pruning: identifying in which cases an operator can only be applied over or produce invalid states inside an abstract state.

This is related to the aforementioned use of *e-deletion* to modify the applicability of operators in partial-state regression (Alcázar et al., 2013). *e-deletion* avoids the generation of some invalid states even when a subset of the variables has been abstracted away. Unfortunately, once again *e-deletion* does not perform all the pruning that is possible with exactly-one constraints. A counter-example is easily constructed with four fluents a, b, c and d subject to two constraints: $\neg(a \wedge b)$ and $b \vee c \vee d$. Consider an operator o such that $post(o) = a$. Then, a is a precondition for the operator in regression, o^r. The constraints allow us to immediately infer that $a \in pre(o^r) \implies \neg b \in pre(o^r) \implies (c \vee d) \in pre(o^r)$. If we abstract away variables a and b, o^r is not applicable in any state having $\neg c \wedge \neg d$, even though it does not e-delete any of them.

A simple way to consider all the possible inferences is to encode all the constraints in the TR representing the original operator and then abstract away the variables not in the pattern. Variables are abstracted away with existential quantification: there is a transition between two abstract states s^α, t^α if and only if a transition $s \xrightarrow{o} t$ exists in the original state space such that $\alpha(s) = s^\alpha$ and $\alpha(t) = t^\alpha$.

This encoding ensures that all the possible pruning is performed. The original TR represents all the pairs of states s, t such that there exists a transition $s \xrightarrow{o} t$ in the original state space. Including all the constraints, we remove all the pairs related to invalid states. The result of the existential quantification is just the transitions between abstract states s^α, t^α such that a valid pair of states s, t supports the transition.

4.7 Empirical Evaluation

In this section, we empirically evaluate the different techniques proposed in this chapter and test the impact of the different parameters we defined. The main goal of our evaluation is to measure the impact that the state invariants have in symbolic search algorithms and find the best way to encode and use them.

We implement our new invariant methods on top of our version of GAMER, with the improved image computation of Chapter 3. To differentiate it from GAMER, we call our planner Constrained GAMER or CGAMER for short. The h^2 preprocessor that removes invalid operators as described in Section 4.1.3 on page 56 is implemented on top of the FAST DOWNWARD preprocessor. All the symbolic search algorithms use the T^{DT}_{100} image computation method, that aggregates TRs based on the disjunction tree criterion, as described in Section 3.4 on page 43. All the settings of our experiments, including the benchmarks and metrics used, were described in Section 1.5.

Our evaluation proceeds in several steps:

1. First, in Section 4.7.1, we study the state invariants found by the preprocessor in each domain. Clearly, the relevance of state invariant pruning will depend on how many invariants are found, so reporting the preprocessor data is important to understand the results presented next.

2. Our analysis of state invariant pruning on symbolic bidirectional uniform-cost search is reported in Section 4.7.2. Again, bidirectional uniform-cost search is suitable for our analysis because it does not depend on any heuristic and the time score is representative of the planner performance.

3. In Section 4.7.3, we consider the BDD minimization methods as an alternative to state invariant pruning. Again, our evaluation is performed under symbolic bidirectional search because of the same reasons.

4. After evaluating the methods proposed in this chapter under bidirectional search, we consider unidirectional uniform-cost and Symbolic A*, in Sections 4.7.4 and 4.7.5, respectively.

4.7.1 State Invariants in Benchmark Domains

Before analyzing the impact of using state-invariant constraints to prune search algorithms, we report exactly which constraints are found by our preprocessor. Table 4.1 show a summary of the constraints found in each domain by FAST DOWNWARD preprocessor with and without the h^2 inference and operator disambiguation described in Section 4.1.3 on page 56.

The first columns show data relative to the largest problem in the domain, $\mathcal{P}$, the one with most operators among those with most facts. *Facts* and $\mathcal{O}$ show the number of facts and grounded

	$\mathcal{P}$							$\mathcal{D}$		
	$\mathcal{P}$	*Facts*	$\mathcal{O}$	$\mathcal{O}^{h2}$	$\mathcal{M}^{fw}$	$\mathcal{M}^{bw}$	*ex1*	$\%_{\mathcal{O}-}$	$\%_{h2}^{fw}$	$\%_{h2}^{bw}$
AIRPORT (50)	049	22338	13100	13100	967703	0	11	74	67	100
BARMAN (20)	019	343	1016	908	995	0	10	13	95	–
BLOCKSWORLD (35)	016	342	578	578	2890	0	18	0	7	–
DEPOT (22)	014	1602	22252	16712	54880	0	55	17	13	–
DRIVERLOG (20)	012	1302	15456	15456	216	0	6	0	0	–
ELEVATORS08 (30)	029	172	1152	1152	0	0	0	0	–	–
ELEVATORS11 (20)	010	154	1008	1008	0	0	0	0	–	–
FLOORTILE11 (20)	017	288	1104	684	3120	525	4	37	0	100
FREECELL (80)	039	484	25380	25362	8900	0	4	0	73	100
GRID (5)	004	1189	15186	15182	611	0	1	0	79	–
GRIPPER (20)	019	214	338	338	252	0	42	0	0	–
LOGISTICS 00 (28)	011	275	650	650	0	0	0	0	–	–
LOGISTICS 98 (35)	027	19487	115136	115136	0	0	0	0	–	–
MICONIC (150)	119	180	3600	3600	0	0	0	0	–	–
MPRIME (35)	013	1671	60906	57210	990	0	0	9	100	–
MYSTERY (30)	013	1601	45872	41196	1837	30	0	46	100	100
NOMYSTERY11 (20)	008	407	8890	7954	2493	0	0	34	100	100
OPENSTACKS08 (30)	029	205	2380	2380	184	0	0	0	100	–
OPENSTACKS11 (20)	019	175	1740	1740	176	0	0	0	100	–
OPENSTACKS06 (30)	029	1104	40300	40300	15025	0	100	3	100	–
PARCPRINTER08 (30)	019	820	1051	275	5875	661	141	65	100	100
PARCPRINTER11 (20)	015	768	461	89	2868	586	134	67	100	100
PARKING11 (20)	019	826	23958	23232	22275	0	22	4	48	–
PATHWAYS-NONEG (30)	029	1014	2232	2232	1465	9	0	0	100	100
PEG-SOLITAIRE08 (30)	029	100	185	177	49	0	0	18	100	100
PEG-SOLITAIRE11 (20)	019	100	185	177	49	0	0	5	100	–
PIPESWORLD-NT (50)	049	2436	13696	13153	42426	0	0	7	100	–
PIPESWORLD-T (50)	049	1534	93316	92510	21705	0	5	2	77	100
PSR-SMALL (50)	049	121	49	45	102	0	22	2	100	–
ROVERS (40)	039	3411	26371	26371	48	0	0	0	100	–
SATELLITE (36)	031	5275	629383	629383	53	0	4	0	100	–
SCANALYZER08 (30)	020	360	25092	25092	2754	0	18	21	27	–
SCANALYZER11 (20)	018	360	27540	27540	2754	0	18	18	21	–
SOKOBAN08 (30)	028	696	1496	1256	9120	0	13	22	8	100
SOKOBAN11 (20)	017	490	1176	648	2126	0	12	22	7	100
TIDYBOT11 (20)	019	765	30488	2758	9347	0	67	74	99	–
TPP (30)	029	2032	43440	14928	144	0	0	41	100	–
TRANSPORT08 (30)	019	608	10980	10980	0	0	0	0	–	–
TRANSPORT11 (20)	015	222	3408	3408	0	0	0	0	–	–
TRUCKS (30)	029	8709	59696	44002	3765	3514	0	38	29	100
VISITALL (20)	019	361	440	440	57	0	0	0	100	–
WOODWORKING08 (30)	019	377	1962	949	878	0	51	50	89	100
WOODWORKING11 (20)	019	343	1823	897	853	0	36	50	89	100
ZENOTRAVEL (20)	012	820	32780	32780	0	0	0	0	–	–

Table 4.1: Preprocessor results. $\mathcal{P}$ selects a representative problem of the domain: number of facts ($Facts$), operators before and after h^2 disambiguation ($\mathcal{O}$ and $\mathcal{O}^{h2}$), forward and backward mutexes ($\mathcal{M}^{fw}$ and $\mathcal{M}^{bw}$) and exactly one invariants ($ex1$). $\mathcal{D}$ shows the average for all the instances of the percentage of removed invalid operators ($\%_{\mathcal{O}-}$) and percentage of mutexes found by h^2 ($\%_{h2}^{fw}$ and $\%_{h2}^{bw}$).

operators of $\mathcal{P}$. $\mathcal{O}^{h2}$ is the number of operators after applying disambiguation with respect to the h^2 constraints. This gives an idea of the maximum magnitude of the problems in each domain, before and after applying the h^2 preprocessor. The next columns show the number of state invariants found in the problem, including forward and backward mutexes, $\mathcal{M}^{fw}$ and $\mathcal{M}^{bw}$, and *exactly-1* invariant groups, $ex1$. We do not count the state invariants that are part of the finite-domain encoding of the task, i. e., the mutexes or *exactly-1* invariants groups that affect to values of the same variable. The last three columns show statistics of the constraints across all the problems in the domain. $\%_{\mathcal{O}-}$ is the average percentage of operators removed from the problem thanks to operator disambiguation using all the constraints. $\%_{h2}^{fw}$ and $\%_{h2}^{bw}$ are the average of the percentage of mutexes that were found by the h^2 inference in the preprocessor in contrast to the monotonicity invariant analysis already present in FAST DOWNWARD preprocessor. Note that the percentages are not null in some domains where no constraints were found in the largest problem, indicating that some invariants were found for some smaller problems in those domains.

Noticeably, there are only a few domains where no constraints can be found: ELEVATORS, LOGISTICS, MICONIC, TRANSPORT and ZENOTRAVEL, all of them being transportation domains. In all other domains there are forward mutexes that can be used to prune a backward search, for example. Backward mutexes, which are useful to prune forward search, are much less abundant, being present in just 15 domains and not in all problems in many cases.

The use of disambiguation removes operators that cannot be part of an optimal path. Whenever there are only forward mutexes, the removed operators are not applicable for any state in the forward search anyway. Nevertheless, removing the operators may benefit the performance since invalid operators may decrease heuristic performance or, in the case of symbolic search, complicate the representation of the TRs. As shown by the column $\%_{\mathcal{O}^-}$, the percentage of invalid operators is not negligible, representing more than half of the operators in some domains like TIDYBOT, AIRPORT, PARCPRINTER or WOODWORKING.

In summary, our preprocessor infers a good number of h^2 constraints in most domains and they help to significantly simplify the problem by removing invalid operators in many cases. In the next sections we evaluate the impact that using these constraints for pruning has in the symbolic search performance.

4.7.2 State Invariants in Bidirectional Uniform-Cost Search

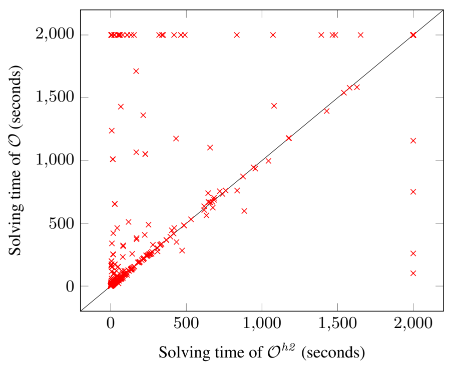
In this section we analyze the impact of using state-invariant constraints to prune symbolic bidirectional uniform-cost search. Our base planner is the original GAMER planner, that only uses state invariants to derive the SAS$^+$ encoding. We call this configuration $\mathcal{O}$, to denote that it uses the original sets of operators. Our first configuration, $\mathcal{O}^{h2}$, does not use state invariants to prune the search, but removes the invalid operators from the task before starting the search. The rest of the configurations prune all the states that violate any invariant found by the preprocessor and they only differ on how the constraints are encoded. $\mathcal{M}^{1k}$ and $\mathcal{M}^{100k}$ use the BDD encoding proposed in Section 4.3 with a maximum number of nodes of 1,000 and 100,000, respectively. In $e\text{-}del$ and $e\text{-}del^+$ constraints are encoded directly in the TRs with the method explained in Section 4.4. $e\text{-}del^+$ include all discovered constraints in the TR of each operator and $e\text{-}del$ encodes only the necessary constraints for each operator.

Table 4.2 shows the results of the state invariant configurations. We also show a direct comparison of the runtimes of these configurations for each problem instance in the four plots of Figure 4.4. The main conclusion that can be drawn from these results is that the usage of state invariants increases performance dramatically in all the domains where some invariants are actually found. In total, the best configuration solves up to 862 problems, 75 more than the version that ignores state invariants. Given that the difficulty of the problem instances increases exponentially in many domains, such increase in performance is very significant.

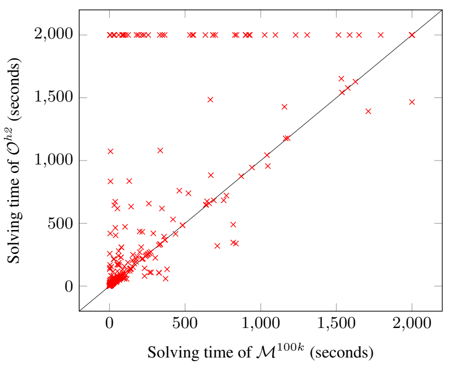
Just using the state invariants to remove invalid operators from the problem already produces a great speed up, as well as an increase in coverage of 27 problems. The domains where a good number of invalid operators were detected, such as PARCPRINTER, TIDYBOT, TPP or WOODWORKING, are greatly simplified. Figure 4.4a compares the runtimes in all solved problems with and without h^2 operator pruning, highlighting the great advantage of pruning operators. It can be concluded that the time spent by the preprocessor analysis is usually negligible with respect to the total time invested by the planner, even in those cases where no invariants are found.

	$\mathcal{O}$	$\mathcal{O}^{h2}$	$\mathcal{M}^{1k}$	$\mathcal{M}^{100k}$	$e\text{-}del$	$e\text{-}del^{+}$
Airport (50)	15.26	23.56	24.00	24.51	***25.67**	19.75
Barman (20)	5.37	5.26	8.31	9.75	***12.00**	7.06
Blocksworld (35)	16.64	16.87	30.34	29.99	***31.83**	15.13
Depot (22)	4.24	3.60	*7.36	*7.49	***7.84**	3.47
Driverlog (20)	13.30	13.38	13.76	**14.00**	13.77	13.75
Elevators08 (30)	**25.00**	**25.00**	**24.88**	**24.91**	**24.91**	**24.95**
Elevators11 (20)	**19.00**	**18.97**	**18.97**	**18.92**	**18.92**	18.81
Floortile11 (20)	6.01	8.11	*13.69	*13.72	***14.00**	*13.41
Freecell (80)	16.10	15.98	23.63	24.12	***26.07**	10.04
Grid (5)	**2.00**	**2.00**	1.68	1.67	1.52	0.95
Gripper (20)	17.38	17.47	17.26	17.21	19.68	**19.93**
Logistics 00 (28)	**19.87**	**19.98**	**19.93**	**19.94**	**19.95**	**19.96**
Logistics 98 (35)	**5.00**	**4.99**	**5.00**	**5.00**	**5.00**	**5.00**
Miconic (150)	**112.70**	**112.81**	**112.87**	**112.90**	**112.52**	**112.97**
MPrime (35)	***21.25**	20.64	***21.14**	***21.12**	20.08	18.69
Mystery (30)	*12.20	*12.88	***13.39**	***13.37**	11.52	9.16
NoMystery11 (20)	15.17	**15.62**	**15.70**	**15.69**	15.46	15.19
Openstacks08 (30)	29.13	29.13	29.41	29.45	**29.97**	**29.76**
Openstacks11 (20)	19.38	19.31	19.54	19.54	**20.00**	19.76
Openstacks06 (30)	*18.33	*18.14	*18.13	***19.21**	*18.56	11.70
ParcPrinter08 (30)	10.03	***23.35**	21.25	21.58	19.59	20.78
ParcPrinter11 (20)	5.99	***17.12**	*16.65	*16.42	*16.10	14.85
Parking11 (20)	0.00	0.00	*0.98	***1.00**	*0.98	0.00
Pathways-noneg (30)	4.69	4.69	**4.96**	**5.00**	4.68	4.80
Peg-Solitaire08 (30)	29.44	**29.98**	**30.00**	**30.00**	**29.77**	24.34
Peg-Solitaire11 (20)	**19.90**	**19.98**	**19.99**	**20.00**	19.65	15.17
Pipesworld-nt (50)	13.66	13.01	14.42	14.05	***15.76**	6.46
Pipesworld-t (50)	*15.68	*15.76	*16.07	*15.70	***16.38**	9.66
PSR-small (50)	**49.67**	**49.59**	**49.51**	**49.77**	**49.36**	49.22
Rovers (40)	13.59	13.61	13.67	13.68	13.59	***14.45**
Satellite (36)	8.09	8.09	10.81	10.80	10.64	***12.70**
Scanalyzer08 (30)	10.75	11.26	10.72	10.77	**11.38**	11.20
Scanalyzer11 (20)	7.97	8.27	7.95	8.00	**8.61**	8.44
Sokoban08 (30)	16.19	*22.48	*26.85	*27.32	***27.73**	22.37
Sokoban11 (20)	*12.58	*16.50	*19.36	*19.45	***19.73**	15.70
Tidybot11 (20)	6.68	9.73	*15.77	*15.80	***16.95**	10.29
TPP (30)	**8.00**	**8.00**	**8.00**	**8.00**	**8.00**	7.66
Transport08 (30)	**13.99**	**14.00**	**13.97**	**14.00**	**14.00**	**14.00**
Transport11 (20)	**9.99**	**9.99**	**9.99**	9.67	**10.00**	9.89
Trucks (30)	9.17	9.46	*12.54	*12.53	11.02	***12.85**
VisitAll (20)	**12.00**	**12.00**	11.74	**11.97**	11.74	11.75
Woodworking08 (30)	15.97	19.02	*24.68	*24.58	***25.93**	***25.69**
Woodworking11 (20)	10.51	13.27	*17.64	*17.65	***18.93**	*18.65
Zenotravel (20)	10.98	10.92	10.97	***11.97**	10.99	***12.00**
Total (1396)	708.85	763.78	827.48	832.22	**840.78**	742.36
Score (36)	16.95	18.15	20.06	20.26	**20.47**	17.82
Total Cov (1396)	787	814	856	858	**862**	799
Score Cov (36)	19.01	19.61	**20.94**	**21.04**	**21.08**	19.38

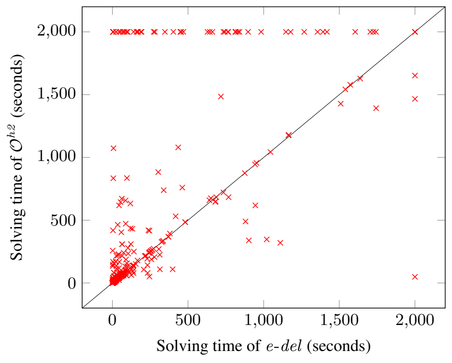
Table 4.2: Time score of bidirectional uniform-cost search with different methods to exploit state invariants. The best configurations per domain and those deviating in only 1% are highlighted in bold and, in domains where there are differences in coverage, the configurations with best coverage are marked with *.



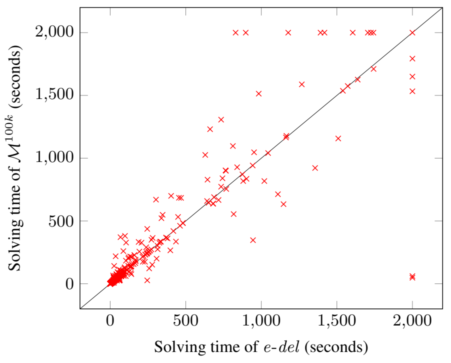
(a) $\mathcal{O}^{h2}$ versus $\mathcal{O}$.



(b) $\mathcal{M}^{100k}$ versus $\mathcal{O}^{h2}$.



(c) Edeletion versus $\mathcal{O}^{h2}$.



(d) Edeletion versus mutex BDDs.

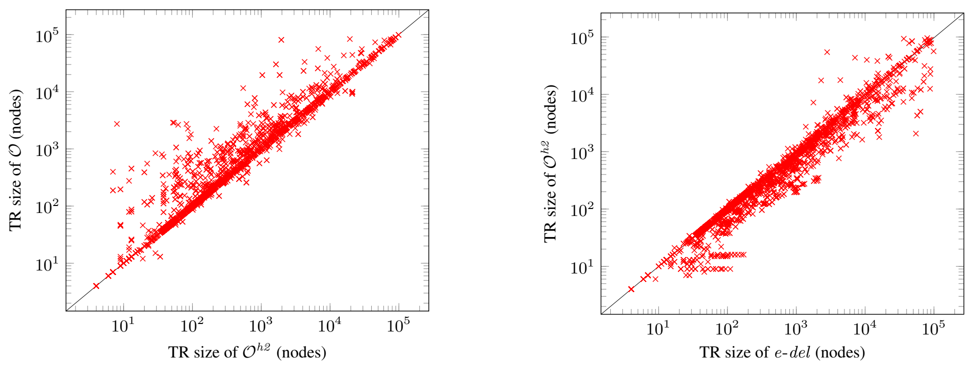
Figure 4.4: Comparison of time to solve problems with approaches using state-invariant constraints. Unsolved problems are assigned a time of 2000 seconds.

Using mutexes to prune the search has even a bigger impact in performance, both in coverage and solution time. Even though there is no theoretical guarantee that pruning states increases the performance of BDD-based search, this seems to be the case for the set of benchmarks considered in our experiments. The only cases where the search performance slightly decreases when pruning invalid states are GRID, PARCPRINTER and VISITALL. On the other hand, performance increases dramatically in many cases, such as BARMAN, FLOORTILE, FREECELL, etc. As shown by Figures 4.4b and 4.4c the use of state invariants is almost always better than the configuration without invalid operators and the speed up is of up to an order of magnitude.

Comparing the different encodings of the mutexes also yields interesting conclusions. Whenever we use the encoding of constraints in separated BDDs of different sizes, $\mathcal{M}^{1k}$ and $\mathcal{M}^{100k}$, we obtain similar results, though slightly favoring the version with a larger limit of 100,000 nodes. As concluded in the experiments regarding the TR representation (see Section 3.5.1 on page 49), using larger bounds for the BDD size leads to faster computation at the expense of using more memory. Encoding constraints in the TRs, e-del, is a more efficient way to use the constraints in the search, though there are some domains where the performance decreases like MYSTERY or ZENOTRAVEL. The advantage of e-deletion is especially noticeable in BARMAN, FREECELL or PIPESWORLD, in which there are a great number of mutexes. The plot of Figure 4.4d shows that the advantage of e-del over $\mathcal{M}^{100k}$ is more moderate than our other comparisons. In this case, the BDDs involved in the search are not affected, but $e\text{-}del$ reduces the overhead in pruning the invalid states and even gets speed ups in the image computation.

$e\text{-}del$ encodes the necessary constraints, according to our theoretical results in Section 4.4. In order to show the relevance of identifying which constraints are relevant, we compare the results with $e\text{-}del^{+}$, which encodes all the constraints in the TRs. Even though in ROVERS and SATELLITE encoding all the constraints simplifies the search, in most cases it is unfeasible to do so, such that the overall results are worse than the configuration not performing pruning at all. This highlights the importance of not including constraints in the operators where they are completely unrelated to the variables affected by the operator.

In order to understand the effect of state invariants in the image computation, we analyze how the constraints affect the size of the TRs. Figure 4.5 plots the relative number of BDD nodes used to represent each transition relation. We only consider the cases were a monolithic TR has been generated, since other cases are not really comparable. Removing invalid operators significantly decreases the size of the TRs, with very few cases in which the size increases. On the other hand, the size of the TRs is expected to increase when encoding state-invariant constraints and, indeed, that is the general trend. However, there are a number of cases where the constraints simplify the TRs. This explains how $e\text{-}del$ is able to provide an advantage over representing the constraints in separated BDDs.



(a) $\mathcal{O}^{h2}$ versus $\mathcal{O}$.

(b) $e\text{-}del$ versus $\mathcal{O}^{h2}$.

Figure 4.5: Number of BDD nodes employed in representing the transition relation of different configurations. Only those cases where a single TR could be created under 100,000 nodes were taken into account.

4.7.3 BDD Minimization Approaches

The results in the previous section show that the performance of symbolic bidirectional uniform-cost search is better whenever we use state invariants. However, as previously explained, there are no guarantees regarding how the constraints affect the performance of BDD-based search. Pruning states could arbitrarily increase the size of the BDDs used in the search. In order to ensure that the size of the BDDs is reduced we proposed the use of BDD-minimization methods that, instead of removing all invalid states from the BDD, add or remove invalid states in order to minimize the BDD size.

Table 4.3 shows the result of symbolic bidirectional uniform-cost search when using BDD-

	$\mathcal{M}_{\emptyset}$	$\mathcal{M}^{100k}$	$\mathcal{M}^{res}$	$\mathcal{M}^{con}$	$\mathcal{M}^{np\&}$	$\mathcal{M}^{lic}$
TOTAL (1396)	767.12	**841.51**	779.74	780.98	822.06	780.04
SCORE (36)	18.30	**20.60**	18.75	18.83	19.96	18.79
TOTAL COV (1396)	814	**858**	821	819	841	823
SCORE COV (36)	19.61	**21.04**	19.84	19.83	20.46	19.95

Table 4.3: Time score and coverage of bidirectional uniform-cost search without pruning ($\mathcal{M}_{\emptyset}$), pruning all invalid states ($\mathcal{M}^{100k}$) and different BDD-minimization methods: restrict ($\mathcal{M}^{res}$), constrain ($\mathcal{M}^{con}$), non-polluting and ($\mathcal{M}^{np\&}$) and leaf-identifying compaction ($\mathcal{M}^{lic}$).

minimization methods to take advantage of state-invariant constraints. The results depict a clear advantage of pruning all invalid states over the configurations using BDD-minimization. Some of those methods are even worse than not using the state-invariant constraints at all. Among BDD-minimization methods $\mathcal{M}^{np\&}$ is the one closest to $\mathcal{M}^{100k}$. This is not surprising since $\mathcal{M}^{np\&}$ is the most conservative of all the BDD-minimization methods. If considering the results in each domain separately, BDD-minimization methods do not improve the coverage in any domain with respect to $\mathcal{M}^{100k}$.

The results of our experiment show that using BDD-minimization to reduce the size of the BDDs involved in the search by including invalid states is not a good idea. This is somehow surprising since most of our analysis depicts a huge correlation between the size of the search BDDs and overall performance of the algorithms. However, even if the BDDs after the minimization are smaller, the result after the image computation contains more states and requires more BDD nodes to be represented.

4.7.4 State Invariants in Unidirectional Uniform-Cost Search

Our previous results show that state invariant pruning greatly enhances the performance of bidirectional search. However, according to the data of Table 4.1, there are certainly more forward than backward mutexes. Thus, invariant constraints can be expected to have a greater impact in backward search than forward search. Table 4.4 reports the results of different configurations in forward and backward search.

As in the bidirectional case, computing h^2 in order to remove invalid operators always pays off. Even in those cases where no operators are removed the time spent computing h^2 is negligible compared to the total search time. When invalid-state pruning is disabled, the benefits of removing invalid operators are larger for forward search because the performance of regression search is still significantly reduced by the presence of spurious states.

The impact of pruning invalid states during the search is much larger in backward search than in forward search. While invariant pruning increases the coverage of forward search in 8 problems, it allows backward search to solve 165 more instances. This is mainly due to the number of mutexes found in each direction.

In forward search, not enough mutexes are found to improve the performance. Mutexes are reliably found for all the problem instances only in FLOORTILE, MYSTERY, PARCPRINTER and TRUCKS. The results in these domains vary a lot. Mutex pruning helps greatly in FLOORTILE, moderately in TRUCKS, does not have an impact in MYSTERY and decreases performance in PARCPRINTER. The case of PARCPRINTER is an exception caused by the structure of the domain, since the search performs many cheap steps and the overhead of mutex pruning might decrease performance. This overhead is almost completely eliminated with e-deletion, though. On the other hand,

	fw				*bw*			
	$\mathcal{O}$	$\mathcal{O}^{h2}$	$\mathcal{M}^{100k}$	$e\text{-}del$	$\mathcal{O}$	$\mathcal{O}^{h2}$	$\mathcal{M}^{100k}$	$e\text{-}del$
AIRPORT (50)	15.75	***23.58**	*23.18	*22.80	2.72	9.29	21.90	21.73
BARMAN (20)	7.04	6.76	6.76	6.88	0.00	0.00	6.48	***9.96**
BLOCKSWORLD (35)	19.08	19.02	18.93	20.05	10.34	10.34	19.43	***22.75**
DEPOT (22)	*4.56	3.79	3.79	***5.00**	1.43	1.51	1.45	2.91
DRIVERLOG (20)	*10.77	***10.95**	*10.73	*10.77	7.24	7.24	7.45	7.48
ELEVATORS08 (30)	***19.00**	***18.98**	***19.00**	***19.00**	4.95	4.95	4.95	4.95
ELEVATORS11 (20)	***15.91**	***15.99**	***16.00**	***15.99**	3.51	3.51	3.51	3.51
FLOORTILE11 (20)	0.86	5.48	*13.58	***14.00**	5.19	5.64	*13.03	*13.11
FREECELL (80)	***19.33**	*18.75	*18.49	*18.85	4.89	4.86	16.46	***19.19**
GRID (5)	***2.00**	***2.00**	***2.00**	***1.99**	0.85	0.32	0.45	0.41
GRIPPER (20)	*16.99	*16.92	*16.94	***20.00**	8.72	8.72	10.95	*18.44
LOGISTICS 00 (28)	**16.00**	**16.00**	**16.00**	**16.00**	13.82	13.80	13.80	13.80
LOGISTICS 98 (35)	***5.00**	*3.09	*4.64	***5.00**	2.57	2.57	2.57	2.57
MICONIC (150)	79.58	79.54	79.40	79.62	***113.99**	***113.61**	***113.34**	***113.43**
MPRIME (35)	*25.24	***25.49**	***25.52**	*24.92	8.16	8.13	8.86	8.82
MYSTERY (30)	*14.11	***14.98**	***14.87**	*14.41	6.08	6.08	6.73	6.58
NOMYSTERY11 (20)	10.36	10.07	10.08	10.10	10.02	11.21	10.85	**11.66**
OPENSTACKS08 (30)	***29.68**	***29.45**	***29.68**	***29.70**	25.39	25.16	*26.21	*27.09
OPENSTACKS11 (20)	**19.91**	**19.91**	**19.91**	**20.00**	17.87	17.88	18.38	18.98
OPENSTACKS06 (30)	*19.49	*19.47	*19.37	***19.82**	6.75	6.70	10.47	13.22
PARCPRINTER08 (30)	9.22	***23.73**	19.66	*23.27	8.56	10.96	18.67	19.47
PARCPRINTER11 (20)	5.45	***17.80**	14.45	***17.72**	4.79	7.18	14.61	*15.18
PARKING11 (20)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PATHWAYS-NONEG (30)	***5.00**	***5.00**	***5.00**	*4.89	3.40	3.39	3.65	3.72
PEG-SOLITAIRE08 (30)	*28.34	***28.99**	***28.81**	*28.34	9.41	10.93	13.83	13.56
PEG-SOLITAIRE11 (20)	*18.78	***18.98**	*18.65	*18.38	2.82	2.83	5.55	5.50
PIPESWORLD-NT (50)	14.21	14.20	13.94	***16.10**	2.06	1.75	5.80	6.63
PIPESWORLD-T (50)	*16.46	*16.45	*16.44	***16.76**	3.82	3.86	4.74	4.83
PSR-SMALL (50)	**49.39**	**49.74**	49.13	49.05	47.18	47.41	48.77	48.21
ROVERS (40)	***12.77**	***12.82**	***12.82**	***12.82**	10.92	10.92	10.93	10.94
SATELLITE (36)	6.57	6.57	6.34	6.57	7.22	7.22	*9.68	***10.00**
SCANALYZER08 (30)	10.62	11.34	10.56	10.94	10.96	**11.52**	**11.63**	11.25
SCANALYZER11 (20)	8.12	8.31	8.06	8.41	8.43	8.52	8.49	**8.71**
SOKOBAN08 (30)	18.19	*26.16	***26.85**	***27.12**	2.00	3.67	22.83	23.65
SOKOBAN11 (20)	*13.78	*19.11	***19.44**	***19.51**	0.30	0.82	16.50	16.93
TIDYBOT11 (20)	9.09	*14.89	*14.81	***15.92**	0.00	1.00	6.70	7.06
TPP (30)	7.85	**8.00**	**8.00**	**8.00**	6.68	6.75	6.83	6.88
TRANSPORT08 (30)	***12.00**	***11.90**	*11.76	***11.90**	7.03	7.04	6.99	6.99
TRANSPORT11 (20)	*6.90	***7.00**	*6.89	*6.90	2.04	2.04	2.04	2.04
TRUCKS (30)	8.79	*9.74	***10.75**	***10.74**	6.31	6.84	8.77	9.50
VISITALL (20)	**9.00**	**9.00**	**9.00**	**9.00**	8.73	8.73	8.74	8.79
WOODWORKING08 (30)	11.88	16.37	16.41	15.98	12.59	15.08	22.08	***25.00**
WOODWORKING11 (20)	6.90	10.96	10.94	10.67	7.21	9.71	16.38	***19.00**
ZENOTRAVEL (20)	***9.00**	*8.89	*8.88	***9.00**	7.24	7.24	7.23	7.23
TOTAL (1396)	648.97	716.17	716.46	**732.89**	434.19	456.93	598.71	631.66
SCORE (36)	15.81	17.38	17.67	**18.17**	9.68	10.14	13.64	14.73
TOTAL COV (1396)	725	768	771	**776**	518	541	687	706
SCORE COV (36)	17.48	18.58	**18.79**	**18.92**	11.86	12.34	15.96	16.63

Table 4.4: Time score of unidirectional uniform-cost search with different configurations of constraints encoding. The best configurations per domain and those deviating in only 1% are highlighted in bold and, in domains where there are differences in coverage, the configurations with best coverage are marked with *.

it is remarkable that e-deletion may increase the performance of forward search, even in domains where constraints provide no pruning such as BLOCKSWORLD, DEPOT, PIPESWORLD or TIDYBOT. In those domains, encoding the constraints in the TRs helps to simplify image computation and to moderately speed up search.

In backward search, however, there are many available constraints in most domains and they can be successfully exploited to improve performance in most of them. In several domains where regression without pruning fails completely such as BARMAN or BLOCKSWORLD, using the constraints allows the planner to solve more problems.

Another interesting conclusion is that state invariants may affect the directionality of the domains, i. e., whether it is easier to perform forward or backward search. In general, the results depict a great advantage of forward over backward search. However, the use of state-invariant constraints helps to reduce the gap. Without the use of invariants, backward search is only clearly better in MICONIC, where as using state-invariants, it is also better in BARMAN, BLOCKSWORLD, NOMYSTERY, SATELLITE, and WOODWORKING. Closing the gap between forward and backward search also helps to increase the performance of the bidirectional search, as we have analyzed in Section 4.7.2.

4.7.5 State Invariants in Symbolic A* Search

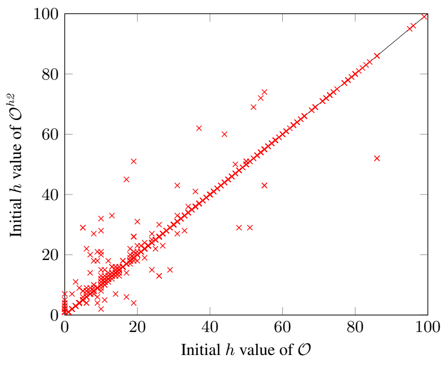
Table 4.5 shows the results of symbolic A* search with symbolic PDBs. Our different constraint encoding methods are not only used in the A* search, but also in the abstract searches to precompute the PDBs. The overall results again show that the use of state invariants helps to improve results of A* as well. However, the benefits are not as stable as in the case of uniform-cost search. Even though pruning invalid states ($\mathcal{M}^{100k}$) increases total coverage, it decreases the coverage in 9 domains with respect to the version without invalid state pruning ($\mathcal{O}^{h2}$). In order to find out whether the performance loss is caused by the abstract searches that generate the heuristic or the A* search, we analyze the quality of the heuristic used. One typical way to measure such quality is the heuristic value of the initial state.

	$\mathcal{O}$	$\mathcal{O}^{h2}$	$\mathcal{M}^{1k}$	$\mathcal{M}^{100k}$
TOTAL COV (1396)	765	792	807	**810**
SCORE COV (36)	19.06	19.64	**20.12**	**20.16**

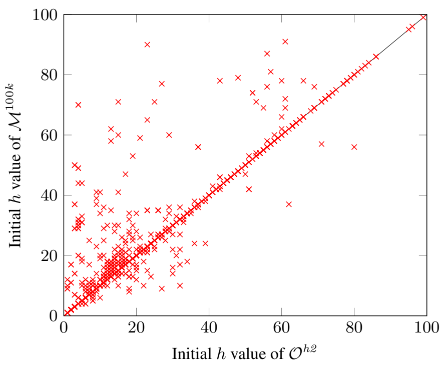
Table 4.5: Time score of BDDA* search with different methods to exploit state invariants. The best configurations and those deviating in only 1% are highlighted in bold.

Figure 4.6 plots the heuristic value of the initial state for different instances. For clarity purposes, we show only values lower or equal than 100 since there are few domains with larger heuristic values. The trend is slightly favorable to the variants with state-invariant constraints, but without a strict dominance. Unlike in our evaluation of state-invariant constraints in blind search, here the use of constraints decreases performance in many cases, obtaining worse symbolic PDBs. Obviously, using state invariants cannot decrease the values of a given abstraction so the reason is that different abstractions are being explored.

In some domains, the difference can be due to alterations in the evaluation function of patterns, misleading the hill-climbing search in the space of possible patterns. This is certainly the case when $\mathcal{O}$ works better than $\mathcal{O}^{h2}$. With $\mathcal{O}^{h2}$ pruning some operators or fluents, the patterns selected are different and, in some cases, they obtain lower heuristic estimates.



(a) $\mathcal{O}$ versus $\mathcal{O}^{h2}$



(b) $\mathcal{O}^{h2}$ versus $\mathcal{M}^{100k}$

Figure 4.6: Heuristic value of the initial state in each instance

However, among the cases where the heuristic value of $\mathcal{M}^{100k}$ is worse than that of $\mathcal{O}^{h2}$, there are cases where the difference is not in the patterns being searched but in the performance of the abstract search itself. While the abstract search without invariant pruning is trivial, pruning makes the search much harder.

We argued before that there were no theoretical guarantees of the search being simpler by pruning constraints. However, it may be surprising that this happens only in the abstract state spaces and not in the whole search space. The reason is that the abstract symbolic search may have lower complexity than the original search, especially if the abstract problem consists of independent subproblems, i. e., the causal graph of the abstract task has separated components. Then, the symbolic abstract search complexity depends on the size of the components instead of the number of variables in the abstraction. Introducing the mutex constraints of the original state space breaks the independence of the variables, increasing the complexity of the abstract search.

4.8 Summary

State-invariants are properties that hold in every reachable state. Any state violating the invariants is an spurious state, i. e., a dead end in regression search. Thus, state-invariant constraints can be used to prune the search. However, even though the importance of state-invariant pruning for regression search in planning is well-known, state-of-the-art symbolic search planners like GAMER do not use them.

In this chapter we used state-invariant constraints in order to prune symbolic search. We interpret state invariants as properties that must hold in any state that is part of a plan for the task, so that they can be used to prune forward and backward search. In order to generate state-invariants, we used mutexes computed with h^2 in progression and regression enhanced with operator disambiguation and *"exactly-1"* invariants derived with monotonicity analysis.

The main contribution of this chapter is to study different methods to use the state-invariant constraints to prune the symbolic search. Our methods require encoding the constraints as BDDs and can be divided into three methods:

- Constraint BDDs: Encode the constraints as BDDs and prune the states in the search using a conjunction to select the subset of states that do not violate the constraints.

- E-deletion: Encode the constraints as additional conditions in the operators, i. e., in the transition relation. We proved which subset of constraints must be encoded in each operator in order to ensure that no invalid states are generated during the search.
- BDD minimization: Constraints are encoded as BDDs, as in the constraint BDD method. However, the aim is to reduce the size of the BDDs representing the sets of states, keeping some spurious states instead of pruning them all.

The overall results show that state-invariant pruning is tremendously effective in symbolic search, especially in regression. Our enhanced version of GAMER, cGAMER, outperforms GAMER in most domains with the exception of the few domains where no state-invariants are found. cGAMER also beats other state-of-the-art planners, as we analyze in more detail in the next chapter. According to our empirical evaluation, e-deletion is the more efficient way to take advantage of constraints in symbolic search. On the other hand, BDD minimization successfully reduces the size of the BDDs involved in the search, but keeping spurious operators makes image computation harder anyway. We also considered the use of constraints in abstraction heuristics that ignore a subset of variables, such as Pattern Databases.

This chapter is an extended version of a previously published work in collaboration with Vidal Alcázar (Torralba and Alcázar, 2013).

Chapter 5

Symbolic versus Explicit-State Search Planning

In the previous chapters we analyzed two aspects of symbolic search, image computation and state-invariant pruning, with the aim of improving the performance of symbolic planners. We conclude with an empirical evaluation of symbolic search techniques, comparing them to explicit-state search.

5.1 Symbolic versus Explicit Unidirectional Uniform-Cost Search

In this section, we compare symbolic against explicit unidirectional uniform-cost search both in progression and regression. Even though neither of these techniques is part of the state of the art, comparing symbolic and explicit search without any heuristic is interesting because it provides a view on the inherent advantages of the symbolic representation. In this comparison, we emphasize the relevance of the improvements for symbolic search techniques that we presented in Chapters 3 and 4. For symbolic search we use the GAMER planner (Kissmann, 2012), which we denote cGAMER when using our improvements.

As the explicit counterpart, we compare against explicit-state forward uniform-cost search and partial-state regression search. For forward search, we used the implementation of A* of the Fast Downward planning system (Helmert, 2006b) using the *blind* heuristic (FD), which assigns a value of 0 to goal states and the minimum action cost to other states. For backward search, we used the FDR planner (Alcázar et al., 2013), implemented on top of Fast Downward. FDR implements partial-state regression using state-invariant pruning as well as other improvements originally made for forward search planners such as successor generators. All planners use the preprocessor described in Section 4.1.3 on page 56 to remove invalid operators and simplify the task.

Table 5.1 shows the comparison of symbolic and explicit uniform-cost search both in forward and backward directions. The results show the benefits of the symbolic representation, especially when using our improvements. In the forward search case, symbolic representation beats explicit state search in most domains, getting a total coverage of $+154$ problems even without our improvements. With cGAMER, the advantage gets increased to $+208$ problems, making symbolic search at least as good as the explicit version in all domains.

In backward search the performance of symbolic search without our improvements, GAMER, is

	Forward			Backward		
	FD	Gamer	cGamer	*FDR*	Gamer	cGamer
Airport (50)	23	23	**24**	22	11	23
Barman (20)	4	8	8	4	0	**10**
Blocksworld (35)	18	21	21	18	12	**23**
Depot (22)	4	4	**5**	2	2	4
Driverlog (20)	7	**11**	**11**	6	7	9
Elevators08 (30)	13	**19**	**19**	4	6	10
Elevators11 (20)	11	**16**	**16**	2	4	8
Floortile11 (20)	8	9	**14**	**14**	8	**14**
Freecell (80)	16	14	**20**	8	6	**20**
Grid (5)	1	**2**	**2**	0	1	1
Gripper (20)	8	**20**	**20**	8	11	**20**
Logistics 00 (28)	10	**16**	**16**	10	**16**	**16**
Logistics 98 (35)	2	**5**	**5**	2	2	3
Miconic (150)	50	82	96	65	85	**114**
MPrime (35)	19	23	**26**	10	9	10
Mystery (30)	**15**	**15**	**15**	11	7	8
NoMystery11 (20)	8	**13**	12	9	9	12
Openstacks08 (30)	21	**30**	**30**	8	20	**30**
Openstacks11 (20)	16	**20**	**20**	3	15	**20**
Openstacks06 (30)	7	11	**20**	7	7	16
ParcPrinter08 (30)	20	22	**24**	20	15	23
ParcPrinter11 (20)	15	17	**18**	15	11	**18**
Parking11 (20)	0	0	0	0	0	0
Pathways-noneg (30)	4	**5**	**5**	4	4	4
Peg-Solitaire08 (30)	27	**29**	**29**	11	12	24
Peg-Solitaire11 (20)	17	**19**	**19**	3	2	14
Pipesworld-nt (50)	14	15	**17**	8	4	9
Pipesworld-t (50)	11	15	**17**	2	4	6
PSR-small (50)	49	**50**	**50**	46	49	**50**
Rovers (40)	5	**14**	13	6	11	12
Satellite (36)	5	7	7	6	7	**10**
Scanalyzer08 (30)	**12**	**12**	**12**	**12**	9	**12**
Scanalyzer11 (20)	**9**	**9**	**9**	**9**	6	**9**
Sokoban08 (30)	27	**28**	**28**	17	4	25
Sokoban11 (20)	**20**	**20**	**20**	14	1	18
Tidybot11 (20)	12	12	**16**	3	1	9
TPP (30)	6	**8**	**8**	5	**8**	**8**
Transport08 (30)	11	11	**12**	8	8	9
Transport11 (20)	6	6	**7**	3	3	4
Trucks (30)	7	10	**11**	9	6	10
VisitAll (20)	9	9	9	9	9	9
Woodworking08 (30)	9	20	21	9	20	**25**
Woodworking11 (20)	4	14	15	4	14	**19**
Zenotravel (20)	8	8	**9**	7	7	8
Total Cov (1396)	568	722	**776**	443	453	706
Score Cov (36)	13.73	17.63	**18.92**	10.89	10.71	16.63

Table 5.1: Coverage of symbolic (Gamer and cGamer) versus explicit unidirectional uniform-cost search (*FD* and *FDR*). Gamer is the baseline symbolic planner and cGamer uses the improvements proposed in this thesis.

similar to that of partial-state regression, FDR. GAMER has better total coverage but the explicit variant has better coverage score. Moreover FDR solves problems faster when the solving times are considered. Nonetheless, note that FDR already uses state-invariant constraints to prune the search. Thus, when symbolic search is enhanced with our state-invariant and image-computation improvements, it consistently dominates explicit search again in all domains except MYSTERY.

In total, the advantage of the symbolic search is notable, obtaining better results than explicit search in all but 6 domains in which they are tied. Forward search is usually better than backward search both in the explicit and symbolic versions. However, backward search should not be abandoned since it obtains good results in a number of domains.

The empirical dominance of symbolic uniform-cost search over the explicit-state variant is supported by theoretical results regarding the size of BDDs involved in the search. The number of BDD nodes required to represent a given set of states is at most linear in the number of states and the number of variables of the task. On the other hand, the symbolic representation can sometimes represent exponentially many states in the size of the planning task with only a polynomial number of nodes. However, symbolic search is not guaranteed to dominate explicit-state search since image computation may be of exponential complexity in the size of BDDs. Moreover, in backward search the number of BDD nodes can be exponentially large in the number of partial states represented.

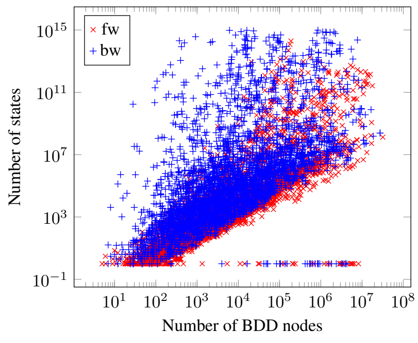
In order to obtain a better picture of the behavior of symbolic uniform-cost search, Figure 5.1 shows data relative to the layers expanded by symbolic forward and backward search. We exclude data from PARCPRINTER since the large action costs in that domain difficult the visualization. For each step in the search (i. e., the expansion of a set of states with a given g-value) we measure the number of states in the set, the number of BDD nodes used to represent the states and the time spent in the step, including image/pre-image computation, duplicate elimination and invalid state pruning.

Figure 5.1a shows the relation between the number of nodes in a BDD and the number of states it represents in each layer of uniform-cost search. Even though there are some cases where the number of BDD nodes is orders of magnitude smaller than the number of states represented and in the majority of cases they follow a linear relation, especially in forward search. To compare symbolic and explicit search, the relation between states and BDD nodes is more significant in forward search, since backward search uses a partial-state representation.

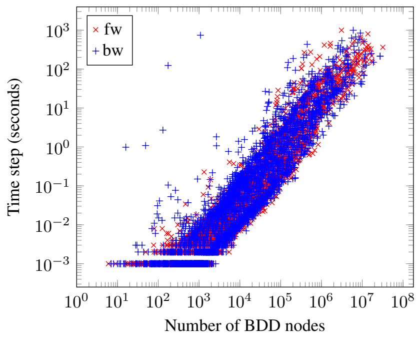
Figure 5.1b compares the time spent in one step of the algorithm with the number of nodes in the BDD that represents the set of expanded states. There is a strong linear relation between the number of nodes of a BDD and the time it takes to expand it. However, for a given BDD size, the time can vary in around two orders of magnitude. In this case, there is no significant difference between the time taken by forward and backward search with respect to the BDD size.

In Figures 5.1c and 5.1d, we show the number of nodes and time required for every g layer, respectively. Even though the number of nodes and time usually grows with the number of steps, this is not obvious when visualizing many problems of different domains at the same time. In this case, we can appreciate the advantage of forward over backward search, since there are some cases where backward search requires more nodes for a given g-layer. This can be observed in the first layers whereas in later layers there are more points associated with forward search due to earlier failures of backward search.

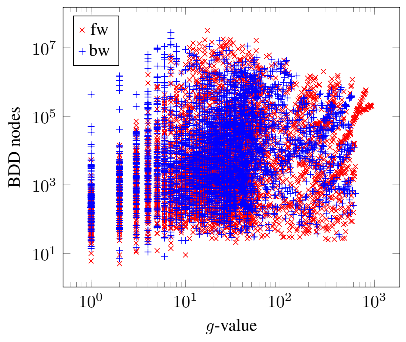
The comparison of forward and backward search shows that the difference between both algorithms is not very remarkable in the time needed for image/pre-image computation on a BDD of a given size, although there is a small difference in the number of BDD nodes needed to represent the sets of states of each g-layer. This confirms that the gap between forward and backward search is not large when using invalid state pruning.



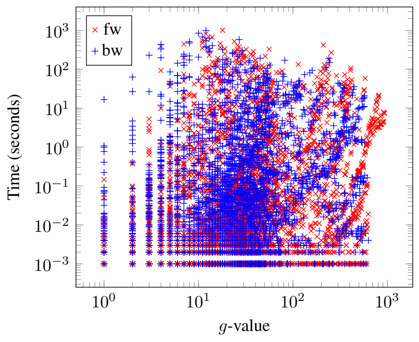
(a) States versus BDD nodes.



(b) Time spent versus BDD nodes.



(c) BDD nodes in each g-layer.



(d) Time spent in each g-layer.

Figure 5.1: Symbolic search layers in unidirectional uniform-cost search.

5.2 State-of-the-art in Symbolic and Explicit Optimal Planning

In the previous section, we have compared symbolic and explicit-state search in uniform-cost search, without using any heuristic. However, state-of-the-art optimal planners use heuristics and other pruning techniques in order to enhance search performance. In this section we compare the performance of symbolic search against heuristic search planners. As characteristic planners, we take the planners that performed best in the IPC-2011: LM-CUT and two configurations of the M&S heuristic. All planners use the preprocessor described in Section 4.1.3 on page 56 to remove spurious operators and simplify the task. FAST DOWNWARD STONE SOUP, the winner of IPC-2011, was a portfolio running these three planners by a fixed amount of time. In order to compare against the best possible results obtained by such portfolio, we compare against the maximum possible coverage where we consider a problem solved if it was solved by any of the individual planners. Thus, *best* is an optimistic approximation of the best possible results using the three planners. We use two symbolic algorithms, bidirectional uniform-cost search and BDDA* with symbolic PDBs, as they were implemented in GAMER and the CGAMER planner that uses our improvements.

Table 5.2 compares the coverage of heuristic search and symbolic planners with time and memory limits of 30 minutes and 4GB. The results show that symbolic algorithms outperform heuristic search planners across a variety of domains. Regarding total performance, heuristic search planners beat both variants of GAMER, mainly due to the accuracy of the LM-CUT heuristic. However, the

	A^*				Gamer		cGamer	
	M&S^b	M&S^g	LM-cut	*best*	*bd*	A^*	*bd*	A^*
Airport (50)	23	23	**29**	**29**	23	20	27	21
Barman (20)	4	4	4	4	8	4	**12**	8
Blocksworld (35)	20	28	28	28	21	27	**33**	26
Depot (22)	6	6	7	7	5	7	**8**	**8**
Driverlog (20)	13	12	13	13	12	**14**	**14**	**14**
Elevators08 (30)	14	1	22	22	24	20	**25**	18
Elevators11 (20)	12	0	18	18	**19**	17	**19**	15
Floortile11 (20)	8	4	**14**	**14**	10	12	**14**	**14**
Freecell (80)	4	19	15	19	14	20	**27**	25
Grid (5)	**3**	2	2	**3**	2	2	2	**3**
Gripper (20)	**20**	8	7	**20**	**20**	**20**	**20**	18
Logistics 00 (28)	20	16	20	20	20	18	20	**22**
Logistics 98 (35)	5	4	**6**	**6**	5	**6**	5	**6**
Miconic (150)	77	53	**141**	**141**	86	79	113	108
MPrime (35)	12	23	23	24	19	25	21	**28**
Mystery (30)	8	**17**	**17**	**17**	14	**17**	13	**17**
NoMystery11 (20)	**19**	14	14	**19**	14	14	16	15
Openstacks08 (30)	21	9	21	21	**30**	29	**30**	26
Openstacks11 (20)	16	4	16	16	**20**	19	**20**	19
Openstacks06 (30)	7	7	7	7	11	7	**20**	15
ParcPrinter08 (30)	20	20	22	**23**	21	13	22	19
ParcPrinter11 (20)	15	15	17	**18**	16	9	**18**	15
Parking11 (20)	0	0	**3**	**3**	0	1	1	0
Pathways-noneg (30)	4	4	**5**	**5**	**5**	**5**	**5**	**5**
Peg-Solitaire08 (30)	29	6	29	29	29	28	**30**	29
Peg-Solitaire11 (20)	19	0	19	19	19	18	**20**	19
Pipesworld-nt (50)	9	16	**18**	**18**	15	15	17	16
Pipesworld-t (50)	8	17	12	17	16	16	17	**18**
PSR-small (50)	**50**	**50**	49	**50**	**50**	**50**	**50**	**50**
Rovers (40)	8	6	7	8	**14**	13	**14**	13
Satellite (36)	7	6	7	7	7	7	**11**	10
Scanalyzer08 (30)	14	9	15	**17**	12	10	12	12
Scanalyzer11 (20)	11	6	12	**14**	9	7	9	9
Sokoban08 (30)	29	3	**30**	**30**	28	**30**	28	**30**
Sokoban11 (20)	**20**	1	**20**	**20**	**20**	**20**	**20**	19
Tidybot11 (20)	1	13	**17**	**17**	12	15	**17**	**17**
TPP (30)	7	6	7	7	**8**	**8**	**8**	**8**
Transport08 (30)	11	11	11	11	12	11	**14**	11
Transport11 (20)	7	6	6	7	8	6	**10**	6
Trucks (30)	8	8	10	10	10	11	**12**	**12**
VisitAll (20)	9	**16**	10	**16**	12	11	12	11
Woodworking08 (30)	13	14	22	22	22	22	**26**	25
Woodworking11 (20)	8	9	15	15	16	16	**19**	18
Zenotravel (20)	12	10	**13**	**13**	10	11	11	12
Total Cov (1396)	631	506	800	844	748	730	**862**	810
Score Cov (36)	15.86	13.65	18.65	20.42	18.38	18.38	**21.08**	20.16

Table 5.2: Coverage of symbolic versus explicit search. Explicit A^* planners use M&S with full (M&S^b) and greedy (M&S^g) bisimulation and LM-cut. *best* gets the best results of the three planners in each problem. Gamer and cGamer use symbolic bidirectional uniform-cost search (*bd*) and BDDA* with symbolic PDBs.

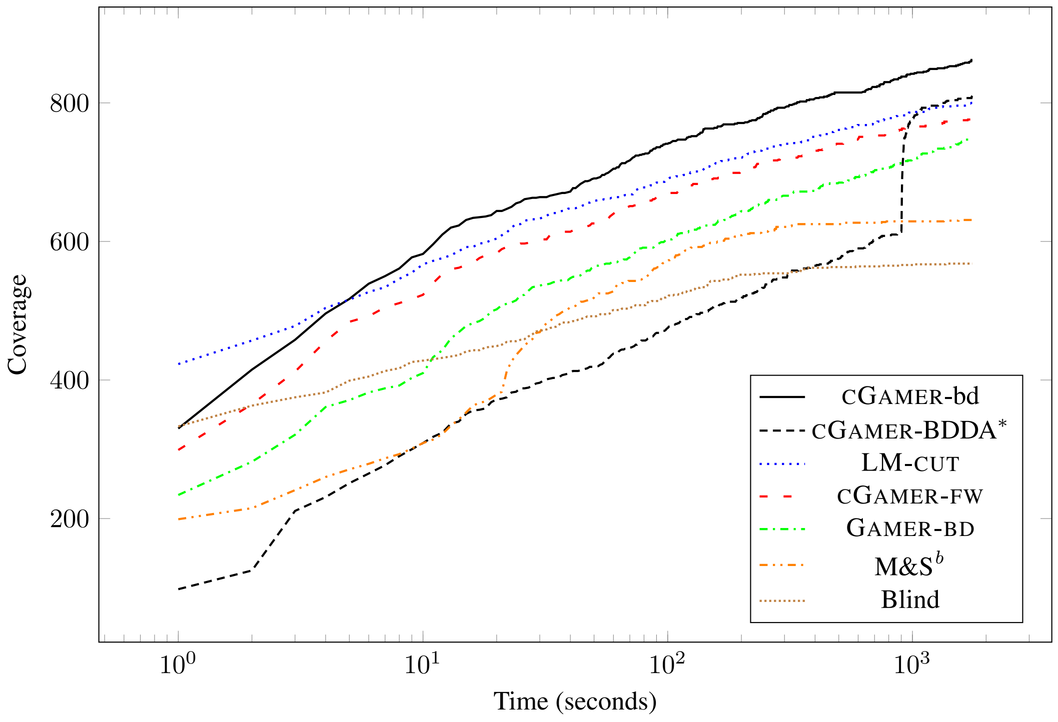


Figure 5.2: Cumulative coverage of symbolic and heuristic search planners. Total coverage of each planner at each time in logscale.

total performance of cGAMER is superior to heuristic search planners in this set of benchmarks. The case of bidirectional uniform-cost search is clear, beating even the portfolio approaches. BDDA* is also superior when considering standalone planners, but it is still behind the optimistic results of the heuristic search portfolio.

The results highlight the importance of symbolic planners not only in terms of average performance but also in a per-domain basis. Symbolic planners are required to obtain the best results in 23 domains. Comparatively, heuristic planners only get better results than symbolic search algorithms in 11 cases: AIRPORT, MICONIC, NOMYSTERY, PARCPRINTER (08 and 11), PARKING, PIPESWORLD-NT, SCANALYZER (08 and 11), VISITALL and ZENOTRAVEL.

Figure 5.2 shows the evolution of the coverage of different planners over time. We plot the cumulative number of instances solved by every planner at every second, in logarithmic scale. In general, the final score of the planners after 30 minutes is representative of the results with other time limits. However, there are some remarkable conclusions related to the behavior of these planners.

Blind search starts faster than other planners, but it converges quickly because the memory limit is exceeded after approximately 5 minutes. On the other hand, configurations that depend on a preprocessing step, such as M&S^b and BDDA* start solving less problems and only outperform blind search if enough time is spent in the search. M&S^b has a similar convergence to that of blind search because the heuristic is fast to compute and the memory limit is reached as well.

In the case of BDDA*, some problems are solved during the first 15 minutes employed by the preprocessing step in case that the original state space is completely traversed by symbolic regression. Afterwards, forward search is performed using the precomputed heuristic, which explains the increase of the coverage at that time.

The comparison of explicit-state and symbolic blind forward search complements our results of Section 5.1. Symbolic search is not only faster than explicit-state search (solving more problems in the first seconds), but also has a much better convergence rate due to the memory savings of BDDs.

Our symbolic search enhancements from Chapters 3 and 4, increase the score of GAMER uniformly over time. This speedup makes CGAMER-FW faster than GAMER, even though it does not perform bidirectional search. However, the lack of regression search causes a faster convergence and it is not clear whether CGAMER-FW would beat GAMER for larger timeouts.

Well-informed heuristics, such as LM-CUT, increase the performance of explicit-state search and the total performance is slightly better to that of symbolic forward search and similar to symbolic BDDA* with PDB heuristics. However, symbolic bidirectional search with our improvements is able to beat LM-CUT except for very short timeouts, probably due to the time spent in BDD initialization and the problem instances for which LM-CUT is perfectly informed.

5.3 Summary

In this chapter, we have empirically compared symbolic planners against explicit-state search planners. Our results show that the advantages of the symbolic representation are clear both in forward and backward search, especially when using the symbolic search improvements we proposed in previous chapters of this thesis. This advantages are then combined in a bidirectional symbolic planner, CGAMER.

The main conclusion is that CGAMER currently is one of the best state-of-the-art cost-optimal planners, beating by a large margin heuristic search planners such as LM-CUT or M&S and even portfolio approaches. Nevertheless, there remain domains where heuristic approaches are better than CGAMER.

Part II

Abstraction Heuristics in Symbolic Search Planning

Chapter 6

Abstraction Heuristics

Abstraction heuristics transform the problem state space to a smaller one and use the cost of the optimal solution as a heuristic estimation for the original problem. In order for an abstraction to be useful, the abstract instance should be easier to solve and the total time spent should be less than without using the abstraction. This is a reasonable requirement, yet it has turned out very difficult to achieve in practice. The key is how to derive "*good*" abstractions for a given problem in a domain-independent way.

In this chapter, we review the state of the art in abstraction heuristics for domain-independent planning. We explain the definition of *abstraction* and *abstraction heuristics* that we adopt, as well as the two methods to automatically derive abstractions in planning that we shall use in the rest of the thesis: Pattern Databases (PDBs) and Merge-and-Shrink (M&S).

6.1 Introduction

According to the empirical evaluation performed in Chapter 5, symbolic blind search outperforms explicit-state blind search. However, while the dominance of symbolic blind search was almost total over explicit-state blind search, heuristic explicit-state search is better in some domains. This suggests that the increased quality of search heuristics sometimes exceeds the structural savings for representing and exploring large state sets in advanced data structures. Therefore, it is natural to question whether heuristics may be used to improve performance in the symbolic setting as well.

The second part of this thesis attempts to address that question. The simplest approach to use heuristics in symbolic search is to use standard state-of-the-art heuristics like LM-CUT (Helmert and Domshlak, 2009) to evaluate each state independently. Unfortunately, that approach does not seem very promising since computing the heuristic is usually the bottleneck in heuristic search. While BDD-based search demonstrated that it may efficiently represent sets of states and compute their successors, the benefit would be negligible if the heuristic has to be computed for each state in the set. Thus, in order to obtain benefits over heuristic search it is necessary to take advantage of the BDD representation of state sets in the computation of the heuristic as well.

In this thesis we will focus our attention on abstraction heuristics. Abstraction heuristics transform the problem state space to a smaller one and use the cost of the optimal solution as a heuristic estimation for the original problem. Abstraction heuristics are a good fit for symbolic search because:

1. **Abstraction heuristics can be used to inform symbolic search.** Abstraction heuristics usually perform a precomputation phase before starting the search. In this phase, the heuristic value for every abstract state is computed and stored for its posterior use in the search. If the heuristic values are represented with BDDs, it is possible to efficiently perform heuristic evaluations in symbolic search, as explained in Chapter 2.

2. **Symbolic search can be used to derive stronger abstraction heuristics**. The use of symbolic search has benefits both in terms of memory and time. Using decision diagrams to represent the heuristic has additional benefits in terms of memory (Ball and Holte, 2008). Moreover, symbolic search can be used to traverse abstract state spaces more efficiently (Edelkamp, 2002).

Hence, it is not surprising that state-of-the-art symbolic planners, like GAMER, use symbolic pattern databases as admissible heuristics. In the following chapters, we shall consider new combinations of abstraction heuristics and symbolic search. In this chapter we review the state-of-the-art in abstraction heuristics. In Section 6.2, we explain which definition of abstraction we adopt and describe some useful properties of abstractions. Then, we review state-of-the-art methods to derive abstractions in a domain-independent way that shall be used in the following chapters, like pattern databases (Section 6.3) and merge-and-shrink (Section 6.4). Finally, Section 6.5 concludes with a summary of the most relevant points for the rest of the thesis.

6.2 Abstractions

Abstractions are simplifications of the problem, transforming the state space into a smaller abstract state space. The abstract instance is optimally solved and the cost of the solution is used to guide the search in the original instance. Multiple abstraction-based algorithms can be defined depending on two aspects: (a) what transformations we apply to the original problem and (b) how we take advantage of the abstract problem solution to solve the original problem. Regarding the way abstractions are used to solve the original problem, the abstract solution plan can be used as a guideline to solve the original problem. For example, abstraction hierarchies can be used to generate an abstract solution plan and refine it until it is valid for the original task (Knoblock, 1994). However, for most problems it is not trivial to automatically generate abstractions suitable for these purposes (Bäckström and Jonsson, 1995; Domshlak et al., 2009). We focus instead on abstraction heuristics, that use the cost of the optimal solution of the abstract problem as an admissible estimation of the optimal cost in the original problem. In this regard, we may freely discard all the information about plans in the abstract state space but their cost.

The question is reduced to what transformations should be applied to the problem. Different definitions of abstractions enable different transformations with diverse properties (Bäckström and Jonsson, 2012). In this thesis, we adopt the definition of homomorphism abstractions used by (Helmert et al., 2014) to define merge-and-shrink abstractions and that have been typically considered for abstractions in planning.

Definition 6.1 (Abstraction (Helmert et al., 2014))**.** *Let* Π *be a planning task with state space* $\Theta = (\mathcal{S}, L, T, s_0, S_\star)$*. An* abstraction *of* Θ *is a surjective function* $\alpha : \mathcal{S} \to \mathcal{S}^\alpha$ *mapping* $\mathcal{S}$ *to a set of abstract states,* $\mathcal{S}^\alpha$*.*

The abstraction defines an abstract state space Θ^α *from the state space* Θ*. The* abstract state space *of* α *is defined as a tuple* $\Theta^\alpha = \langle \mathcal{S}^\alpha, L, T^\alpha, s_0^\alpha, \mathcal{S}_\star^\alpha \rangle$ *where* $\mathcal{S}^\alpha$ *is the set of abstract states,* L *is a set of labels,* $T^\alpha = \{(\alpha(s), l, \alpha(t)) \mid (s, l, t) \in T\}$*,* $s_0^\alpha = \alpha(s_0)$ *and* $\mathcal{S}_\star^\alpha = \{s^\alpha \mid \exists s \in \mathcal{S}_\star, s^\alpha = \alpha(s)\}$*. The size of* α*, written* $|\alpha|$*, is the number of abstract states,* $|\mathcal{S}^\alpha|$*.*

The transformation $\Theta \to \Theta^\alpha$ is a homomorphism*, i. e., a structure preserving mapping such that for all $s, t \in \mathcal{S}$, $(s, l, t) \in T$, implies $(\alpha(s), l, \alpha(t)) \in T^\alpha$.*

Other definitions generalize the one we have adopted in that they allow us to perform more types of problem transformation other than homomorphisms. For example the abstractions considered by Bäckström and Jonsson (2012) or multi-mapping abstractions (Pang and Holte, 2012) that use non-surjective functions in which a state is mapped to more than one abstract state. We chose to stick with Definition 6.1 because restricting ourselves to homomorphism abstractions allows us to rely on their useful properties. Each abstraction function α is mapped to a unique abstract state space, Θ^α. Thus, given any mapping function α, the corresponding heuristic h^α is automatically defined, as in Definition 6.2.

Definition 6.2 (Abstraction heuristic (Helmert et al., 2014))**.** *Let α be an abstraction with an associated abstract state space Θ^α. The induced* abstraction heuristic *h^α uses the cost of the cheapest path from $\alpha(s)$ to $\mathcal{S}^\alpha_\star$ in the abstract state space, $h^*(\alpha(s), \mathcal{S}^\alpha_\star)$ as an estimation of the cost of the original problem, $h^*(s, \mathcal{S}_\star)$. The heuristic is admissible and consistent, since Θ^α is a homomorphism and paths in the original state space are preserved in the abstract state space.*

The transformation induced by an abstraction may be interpreted in several ways. On the one hand, every abstraction α induces an *equivalence relation* on $\mathcal{S}$, $\sim^\alpha$, defined as $s \sim^\alpha t$ if and only if $\alpha(s) = \alpha(t)$, i. e., they are mapped to the same abstract state. If $s \sim^\alpha t$, s and t are considered equivalent by the abstract state space. Hence, each abstract state s^α_i may be interpreted as a set of states S^α_i such that it contains every state mapped to s^α_i, i. e., $S^\alpha_i = \{s \mid \alpha(s) = s^\alpha_i\}$. Therefore, we will freely refer to an abstract state as an equivalence class or a set of states of the original state space.

Another useful definition is that of relevant variables of an abstraction:

Definition 6.3 (Relevant variables of an abstraction (Helmert et al., 2007; Helmert et al., 2014))**.** *Let Π be a planning task with variable set $\mathcal{V}$, and let α be an abstraction of Θ. We say that α depends on variable $v \in \mathcal{V}$ if and only if there exist states s and t such that $\alpha(s) \neq \alpha(t)$ and $s[v'] = t[v']$ for all $v' \in \mathcal{V} \setminus \{v\}$. The set of relevant variables for α, written $\mathcal{V}(\alpha)$, is the set of variables in $\mathcal{V}$ on which α depends, i. e., variables needed to describe its abstract states.*

Note that abstraction is *transitive*: if α is an abstraction of a transition graph, Θ, and α' is an abstraction of Θ^α, then their composition, $\alpha' \circ \alpha$, is also an abstraction of Θ. Let α_1 and α_2 be two abstractions of a given state space Θ. We say that α_2 is an abstraction of Θ^{α_1} if and only if for every pair of states $s, t \in \mathcal{S}$, $\alpha_1(s) = \alpha_1(t) \implies \alpha_2(s) = \alpha_2(t)$.

Definition 6.4 (Abstraction Hierarchy)**.** *An abstraction hierarchy is a directed acyclic graph in which each node corresponds to an abstraction. There is an arc from node α_i to node α_j if and only if α_j is an abstraction of Θ^{α_i}.*

We will mostly use hierarchies in which each node has only one child, i. e., lists of an arbitrary number of abstractions, $\alpha_0, \alpha_1, \ldots, \alpha_k$ such that each α_i is an abstraction of $\Theta^{\alpha_{i-1}}$ for all $i > 0$. Note that the original problem, Θ, may be the root of a hierarchy if we set α_0 to be an injective function (in that case Θ^{α_0} is isomorphic to Θ).

The previous definitions allow us to define an enormous number of different abstractions to compute heuristic estimates. However, for a heuristic to be useful, it has to be efficiently computable. From a practical point of view, this means that the benefits in terms of node expansions have to compensate the time spent in computing the heuristic for every node. Settling this theoretically is way too complicated, even for particular cases, given that the answer depends on many factors and even implementation details. Nevertheless, a common requirement for domain-independent abstractions is that they can be computed and solved in polynomial time in the size of the planning task (and preferably in at most quadratic time). Abstractions may be classified according to how we guarantee that the optimal costs in the abstract state space are computable in polynomial time for each state s. We identify three methods in the literature: explicit, implicit and symbolic abstraction heuristics.

Explicit-abstraction heuristics Explicit-abstraction heuristics construct a small state space that can be completely traversed. Before starting the search they perform a precomputation phase in which the optimal cost to reach an abstract goal state from every abstract state is computed and stored in a *lookup table*. The optimal costs are usually computed by traversing the state space with a best-first search, such as uniform-cost search, that traverses the entire abstract state space in polynomial time in the number of abstract states. The resulting lookup table has one entry per abstract state, so that the corresponding cost can be retrieved with minimum overhead. During the search, to evaluate the heuristic value $h^{\alpha}(s)$ of any state s, we follow two steps: (1) get the corresponding abstract state, $\alpha(s)$, and (2) perform a *lookup* in the table where all the abstract distances have been stored.

Explicit abstraction heuristics are tractable whenever they keep the number of abstract states, $|\mathcal{S}^{\alpha}|$, polynomially bounded in the problem size and given an abstraction function that can be computed in polynomial time. In practice, the number of states is bounded from above by a constant C, $|\mathcal{S}^{\alpha}| < C$. Also, the abstraction methods we study in the following sections, pattern databases and merge-and-shrink, have mappings computable in polynomial time.

Implicit-abstraction heuristics The main drawback of explicit-abstraction heuristics is that they have to keep the abstract state space small. Implicit abstraction heuristics, on the other hand, do not explicitly represent nor traverse the abstract state space, allowing us to use arbitrarily large abstract state spaces. Implicit-abstraction heuristics generate abstractions in a way that they belong to tractable fragments of planning problems (Katz and Domshlak, 2010a). For problems in these tractable fragments, one can take advantage of their structure (usually identified by special properties of the causal graph) to solve them in polynomial time. Some examples are *forks* and *inverted forks*.

Symbolic-abstraction heuristics Symbolic-abstraction heuristics proceed in a similar way to explicit-abstraction heuristics, precomputing the heuristic values of every state in the state space. The difference is that symbolic uniform-cost search is used to traverse the abstract state space more efficiently and the heuristic values are stored in decision diagrams. Symbolic search may lead to exponential gains in both time and memory, enabling the use of larger state spaces. Ideally, we should choose abstractions as informed as possible while keeping the search tractable. Unfortunately, proving that a symbolic search exploration of a given problem is tractable is not a trivial task. Kissmann and Hoffmann (2013); (2014), analyze in which cases it is possible to prove that a given SAS^{+} problem has efficient BDD explorations under some variable orderings, by looking at its causal graph. The answer is mostly negative: even for simple causal graphs one cannot prove the existence of variable orderings that make the symbolic search tractable. According to Kissmann and Hoffmann, "the evidence speaks against a strong connection between causal graph dependencies, and dependencies as relevant for BDD size".

Therefore, symbolic abstraction heuristics rely, as their explicit counterpart, on "small enough" abstract state spaces to guarantee that they can be completely traversed. Symbolic search is still useful to search larger state spaces than explicit abstractions.

6.3 Pattern Databases

Pattern Databases (PDBs) were originally defined as a selection of tiles in the sliding-tiles puzzle (Culberson and Schaeffer, 1998) and later extended to other domains. More general definitions have been applied, shifting the focus from the mere selection of care variables to different state-space abstractions (Holte et al., 2004). In automated planning, the *pattern* is usually defined as a selection of state variables, while the value of other variables is ignored (Edelkamp, 2001). A distinction must be made between the meaning of the term *pattern database* in the heuristic search and planning communities. While in the heuristic search community the term PDB is usually applied to any kind of abstraction heuristic whose values are precomputed and stored in a lookup table, in planning those are considered explicit-abstraction heuristics. Thus, in the planning community, the term is more specific regarding the type of abstractions considered. Other types of abstractions in planning like Cartesian abstractions (Seipp and Helmert, 2013) or merge-and-shrink abstractions could be considered more complex patterns for PDBs in the heuristic search community. In this thesis, we consider the definition of PDB commonly adapted in the planning community, that is, a projection of the original planning task over a subset of variables.

Definition 6.5 (Projection (Helmert et al., 2007; Helmert et al., 2014))**.** *A* projection *of the planning task, Π, over a subset of variables $V \in \mathcal{V}$ is defined by restricting the initial state, goals and preconditions/effects of the operators to V. In other words, a projection is an abstraction, α, so that two states s and t are equivalent if and only if they agree on the value of variables in $\mathcal{V}$, i. e., $s \sim^{\alpha} t$ if and only if $s[v] = t[v]$ for all $v \in V$.*

Different extensions to improve the performance of PDBs have been proposed, such as PDB compression (Felner et al., 2007; Ball and Holte, 2008). Other extensions relevant for this thesis that can also be used with other types of abstraction are:

- **Constrained Pattern Databases** (Haslum et al., 2005) enhance PDBs by using problem invariants. Since some invariants might be lost in the abstracted state space, we can safely prune abstract states in which the invariants do not hold. Admissibility is still preserved, since pruned states are also unreachable in the original state space. In particular, mutex and monotonicity invariant groups forbid two mutually exclusive propositions to hold in the same state. Thus, abstract states in which two mutex propositions appear are automatically pruned.

- **Partial Pattern Databases** (Anderson et al., 2007) aim at searching larger abstract state spaces at the expense of not fully traversing these abstract spaces. The backward search is truncated at goal distance d so that the distance of any expanded abstract state is known. States that were not expanded are assigned the next value larger than d. If the partial PDB takes into account all the problem variables, it searches a perimeter around the goal (Dillenburg and Nelson, 1994; Manzini, 1995).

- **Perimeter Pattern Databases** (Felner and Ofek, 2007) combine perimeter search and PDBs by storing the distance to states in the perimeter instead of the distance to the goal, though they were reported not to be better than the maximum between the PDB and the perimeter heuristic. More recently, perimeter PDBs have been applied in the context of automated planning (Eyerich and Helmert, 2013) showing that perimeter search can enhance the performance of standard PDBs contradicting the impressions of Felner and Ofek. A complete explanation of Perimeter PDBs is given in Chapter 8. Another extension considered storing different entries in the PDB for the distance to each state in the perimeter (Linares López, 2008) which was also shown to be beneficial.

6.3.1 Pattern Selection

The performance of PDBs greatly depends on the *patterns* chosen to make the abstractions. How to select PDB patterns has been an open question for a long time, though some progress has been made. For example, there have been studies of which patterns are best for particular combinatorial domains like sliding-tiles puzzle (Felner et al., 2004). The intuition is that the PDBs must select variables that are coupled. In the sliding-tiles puzzle this means to choose tiles that are close together in the goal state and preferably near to a corner.

In domain-independent planning, the pattern selection must be done for each problem without domain-specific knowledge. As we only consider projection abstractions, this is simplified to selecting a subset of SAS$^+$ variables. State-of-the-art methods for selecting patterns are based on a search in the space of possible patterns. Different types of search can be used like hill climbing (Haslum et al., 2007) or genetic algorithms (Edelkamp, 2006).

Some patterns may be skipped if they do not fit the following conditions:

- All patterns must include at least one goal variable, or else the heuristic value of every abstract state is zero.
- Every variable in the pattern is relevant. A variable v is relevant if and only if v is a goal variable or there exist another relevant variable in the pattern, v', and operator in the planning task, $o = (pre(o), \mathit{eff}(o), c(o))$, such that $v \in \mathcal{V}_{pre(o)}$ and $v' \in \mathcal{V}_{\mathit{eff}(o)}$. If a variable is not relevant, it enlarges the size of the abstract state space for no reason and can be removed.

To select which patterns are better, they are evaluated by searching the abstract state space and comparing which patterns derive larger heuristic estimates.

6.3.2 Limitations of Pattern Databases

Even assuming an oracle that is able to predict the best patterns for a given problem, the capabilities of PDBs have been proved to be quite restricted for some typical benchmarks (Helmert et al., 2007; Helmert et al., 2014). As an example, take a logistics task with m trucks, where one or more packages have to be picked up in location A and transported to location B. The plan consists of moving a truck to location A, loading the packages, moving the truck to location B and unloading the packages. If one single truck location is not included in the PDB, the truck is considered to be anywhere. In that case, the omitted truck may tele-transport all the packages without making a single move action. Therefore, in a domain like the one presented, some actions cannot be captured by PDBs unless they consider every truck — which means having an exponentially sized abstract state space in the number of truck variables.

6.4 Merge-and-Shrink

Merge-and-shrink (M&S) is an algorithm that derives abstractions that take into account all the problem variables, thus overcoming the major limitation of PDB abstractions. M&S was originally proposed in the context of directed model checking (Dräger et al., 2006; Dräger et al., 2009) and later adapted to planning (Helmert et al., 2007; Helmert et al., 2014).

Formally, M&S abstractions are constructed using the following rules:

(A) *Atomic projections*: For $v \in \mathcal{V}$, the *atomic projection* π_v of the task over a single variable v is an M&S abstraction over $\{v\}$.

Algorithm 6.1: Merge-and-shrink (Helmert et al., 2014)

```
Input: Planning task $\Pi$, size bound $M$
Output: M&S abstraction $\alpha$
1 $\mathcal{A} := \{\pi_v \mid v \in \mathcal{V}\}$
2 while $|\mathcal{A}| > 1$ do
3     Select $\alpha_1, \alpha_2 \in \mathcal{A}$
4     Shrink $\alpha_1$ and/or $\alpha_2$ until $|\alpha_1| \cdot |\alpha_2| < M$
5     $\alpha' := \alpha_1 \otimes \alpha_2$
6     Shrink $\alpha'$
7     $\mathcal{A} := (\mathcal{A} \setminus \{\alpha_1, \alpha_2\}) \cup \{\alpha'\}$
8 return the only abstraction in $\mathcal{A}$
```

(S) *Shrinking*: If β is an M&S abstraction over a set of variables $W \subseteq \mathcal{V}$ and γ is a function on S^β, then $\gamma \circ \beta$ is an M&S abstraction over W.

(M) *Merging*: If α_1 and α_2 are M&S abstractions over disjoint sets of variables $W_1, W_2 \subset \mathcal{V}$, $W_1 \cap W_2 = \emptyset$, then $\alpha_1 \otimes \alpha_2$ is an M&S abstraction over $W_1 \cup W_2$.

Rule (A) allows the algorithm to start from *atomic projections*, one for each variable. *Rule (S)* further abstracts an abstraction β by aggregating an arbitrary number of abstract states into the same abstract state. This reduces the total number of abstract states in the abstraction. Formally, this simply means to apply an additional abstraction γ to Θ^β. In *rule (M)*, the *merging step*, the merged abstraction $\alpha_1 \otimes \alpha_2$ is defined by $(\alpha_1 \otimes \alpha_2)(s) := (\alpha_1(s), \alpha_2(s))$. In other words, the new abstraction has $|\alpha_1| \times |\alpha_2|$ states, one per each pair $s_{\alpha_1} \in \Theta^{\alpha_1}$, $s_{\alpha_2} \in \Theta^{\alpha_2}$. The state space i n rule (M) is obtained with the *synchronized product* $\Theta^{\alpha_1} \otimes \Theta^{\alpha_2}$.

The *synchronized product* of two abstractions α_1 and α_2 is a standard operation deriving a new state space $\Theta^{\alpha_1 \otimes \alpha_2} = (\mathcal{S}', L, T', s'_0, \mathcal{S}'_\star)$ where $\mathcal{S}' = \mathcal{S}^{\alpha_1} \times \mathcal{S}^{\alpha_2}$, $T' = \{((s_1, s_2), l, (s'_1, s'_2)) \mid (s_1, l, s'_1) \in T^{\alpha_1} \wedge (s_2, l, s'_2) \in T^{\alpha_2}\}$, $s'_0 = (s_0^{\alpha_1}, s_0^{\alpha_2})$ and $\mathcal{S}'_\star = \{(s_1, s_2) \mid s_1 \in \mathcal{S}_\star^{\alpha_1} \wedge s_2 \in \mathcal{S}_\star^{\alpha_2}\}$. The constraint $W_1 \cap W_2 = \emptyset$ ensures that this is correct, i. e., that $\Theta^{\alpha_1} \otimes \Theta^{\alpha_2} = \Theta^{\alpha_1 \otimes \alpha_2}$.

Algorithm 6.1 shows the pseudocode of the M&S algorithm. It takes as input a planning task and a parameter M that imposes a bound for the abstraction size, i. e., no abstraction in the process will have more than M abstract states. The algorithm initializes a pool of abstractions with the atomic projection with respect to every variable $v \in \mathcal{V}$. While there is more than one abstraction left in the pool, the algorithm selects two abstractions and *merges* them, replacing them by their combination. Prior to every merging step, a shrinking step is applied to both selected abstractions, if necessary for the merged abstraction to satisfy the size bound M. Also, the resulting abstraction may be shrunk with an *exact strategy* such as the bisimulation strategy that we describe in detail in Section 6.4.2. This shrinking step aims to reduce the abstraction size without losing any information about the original problem.

M&S is a generalization of PDBs, since for any PDB we can construct an equivalent M&S abstraction just by merging the atomic abstractions that correspond to variables in the pattern. Variables not in the pattern are shrunk to a single abstract state, so that the abstraction does not distinguish their value, just as PDBs. However, using different shrink strategies M&S can derive abstract state spaces that PDBs cannot, e. g., considering all the variables of the problem while keeping the abstract state space small enough.

To implement M&S in practice, we need a *merging strategy* deciding which abstractions to merge in rule (M), and a *shrinking strategy* deciding which (and how many) states to aggregate in rule (S).

The performance of M&S greatly depends on the policies chosen for these steps.

6.4.1 Merge Policies

A *merge policy* decides which two abstractions to merge next. We say it is a *linear merging strategy* if, at each merge step, at least one of the two abstractions to be merged is an atomic abstraction (single variable abstraction). The rough idea is that SAS$^+$ variables are greedily chosen to construct a larger state space by computing the (synchronized) product of the existing state space and the one induced by the next SAS$^+$ variable. In this case, the merge policy is characterized by the order by which the variables are merged, $v_1, \ldots, v_n$ (hence "linear"). Even though the original work of Dräger et al. in Model Checking used a non-linear strategy, in planning, non-linear merge strategies have been introduced only recently (Sievers et al., 2014). Hence, all the merging strategies we will consider are linear.

Merging strategies usually determine the variable ordering a priori, selecting variables according to different criteria. The criteria used to select the variable ordering are based on the causal graph relationships.

- *cg*: prefer variables that are causally connected in the causal graph with the already merged variables. This criterion selects variables that are relevant for previously selected variables.
- *goal*: prefer goal variables over non-goal variables. This criterion attempts to consider the goals of the problem as soon as possible.

Ties are broken either randomly or according to *level* and *reverse-level*, i. e., the internal variable order of the Fast Downward planning system (Helmert, 2006b) or the reverse order. The best results reported with M&S used a *reverse level* strategy (Nissim et al., 2011; Helmert et al., 2014).

6.4.2 Shrink Policies

A *shrinking policy* takes as input a transition system (an abstract state space) and decides which abstract states must be aggregated in order to reduce the number of abstract states. All aggregated states are considered equivalent by the resulting abstraction. Thus, the output of the shrinking policy is an equivalence relation $\sim$ on abstract states. Optionally, the shrinking policy may take a parameter, M, that determines the maximum number of abstract states after the shrinking.

Several properties may be defined depending on which abstract states are aggregated. A shrinking policy is *locally h-preserving* with respect to a state space if it only aggregates states with the same goal distance in the abstract state space, Θ^α, i. e., $s \sim t$ only if $h^\alpha(s) = h^\alpha(t)$. A shrinking policy is *globally h-preserving* if it only aggregates states that have the same goal distance in the original state space, Θ, i. e., for every pair of abstract states s, t and assignment to non-relevant variables $d^{\mathcal{V}\setminus\mathcal{V}_\alpha}$, $s \sim t$ if and only if $h^{\alpha\otimes\Pi_{\mathcal{V}\setminus\mathcal{V}_\alpha}}(s \cup d^{\mathcal{V}\setminus\mathcal{V}_\alpha}) = h^{\alpha\otimes\Pi_{\mathcal{V}\setminus\mathcal{V}_\alpha}}(t \cup d^{\mathcal{V}\setminus\mathcal{V}_\alpha})$. When using a *globally h-preserving* strategy M&S always derives the perfect heuristic, h^*, since the shrinking keeps all the information that is relevant for the heuristic.

Similar properties of *locally/globally g-preserving* can be defined with respect to the g-value of abstract states. Furthermore, we say that a policy is *locally/globally f-preserving* if and only if it is both *locally/globally g-preserving* and *locally/globally h-preserving*.

The first shrinking strategy for planning, *fh*-shrinking, was based on these properties (Helmert et al., 2007). *fh*-shrinking aggregates states having the same g and h-values in the abstract state space until only M abstract states are left. It prefers aggregating states far away from the initial or goal states, i. e., those with highest f value, breaking ties in favor of states far away from the goal (with highest h-value). Thus, *fh*-shrinking is a locally f-preserving strategy. However, it was clearly outperformed by bisimulation-based shrinking (Nissim et al., 2011; Helmert et al., 2014).

Definition 6.6 (Bisimulation)**.** *Let $\Theta = \langle S, L, T, s_0, S_\star \rangle$ be a labeled transition system. An equivalence relation $\sim$ on S is a* bisimulation *for Θ if, whenever $s \sim t$ (in words: s* and *t* are bisimilar*), a transition $s \xrightarrow{l} s'$ exists if and only if there exists another transition $t \xrightarrow{l} t'$ such that $s' \sim t'$. We say that $\sim$ is* goal-respecting *for Θ if $\sim \subseteq \sim^G$, where $s \sim^G t$ if and only if either $s, t \in S_\star$ or $s, t \notin S_\star$. We say that $\sim$ is a* coarsest goal-respecting bisimulation *if, for every goal-respecting bisimulation $\sim'$, we have $\sim' \subseteq \sim$.*

In words, two (abstract) states s and t are *bisimilar* if they agree on whether or not the goal is true and every planning operator applied on s and t leads to states that are bisimilar. Even though the definition is recursive, a *coarsest goal-respecting bisimulation* always exists and can be computed in time polynomial in the size of the problem. The bisimulation shrinking strategy computes the *coarsest bisimulation*, i. e., it aggregates all states that are bisimilar. Moreover, bisimulation is globally h-preserving, so if only bisimilar states are aggregated, then M&S is guaranteed to derive the perfect heuristic.

Furthermore, label reduction may be applied to consider some operators equivalent, preserving the heuristic global optimality while potentially reducing the size of the abstraction. This reduction can be exponential in the problem size, allowing M&S to derive perfect heuristics in polynomial time in more cases. In this thesis we consider the non-exact label reduction (Nissim et al., 2011; Helmert et al., 2014) though a stronger *exact* label reduction was later proposed (Sievers et al., 2014).

In most benchmark domains, however, coarsest bisimulations are still large even under label reduction. In those cases it is possible to further reduce the size by applying another shrinking policy or to relax the bisimulation property. Relaxed variants of bisimulation only take into account a subset of transitions. *Greedy bisimulation* ignores all the transitions going to states further from the goal (i. e., (s, l, s') such that $h^\alpha(s) < h^\alpha(s') + c(l)$). Since all abstract optimal paths are preserved, greedy bisimulation is locally h-preserving. *Label-catching bisimulation* only takes into account transitions labeled with a previously selected set of relevant labels (Katz et al., 2012). Even though it has been shown that for some label subsets label-catching bisimulation is globally h-preserving, in general, it is intractable to compute such sets of labels. Instead, heuristics criteria approximate sets of label, without guarantees of being locally or globally preserving.

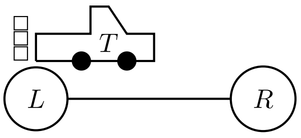
6.4.3 Cascading-Tables Representation of M&S

The *cascading-tables* representation is the data structure commonly used to represent M&S abstractions. It was first used by (Dräger et al., 2006) and presented in detail by (Helmert et al., 2014). It provides efficient operations to perform any of the three M&S abstraction rules defined in Section 6.4.

The cascading tables represent every intermediate M&S abstraction by means of a table, as shown in the example of Figure 6.1. This example shows a logistics task in which there are two locations: the left location, L, and the right one, R. A number of packages are initially located at L and must be delivered to R, by using the truck T. Thus, the task has one variable per package to identify its position (L, T, or R) and a variable identifying the location of the truck (L or R).

Each table represents an intermediate abstraction with relevant variables $W \subseteq \mathcal{V}$. These tables recursively define the mapping between states in Θ^W to abstract states. Each abstract state is identified with a number, from 0 to $|\alpha| - 1$, where $|\alpha|$ the number of abstract states in the abstraction.

Atomic abstraction tables (rule A) associate every possible value of the variable with an abstract state. The algorithm always starts with the atomic abstractions respective to each variable (rule A),



(a) Task of the example. Packages must be transported from location L to location R with the truck T.

v_i	
L	0
T	1
R	2

(A) π_i

π_1 \ π_2	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8

(M) $\pi_1 \otimes \pi_2$

π_1 \ π_2	0	1	2
0	0	1	3
1	1	2	4
2	3	4	5

(S) $\alpha_1 \coloneqq \gamma_1 \circ (\pi_1 \otimes \pi_2)$

Figure 6.1: Example of *cascading-tables* M&S representation. Subfigure (a) shows the task for our example while subfigures (A), (M) and (S) correspond to the application of M&S rules in the *cascading-tables* representation.

π_i. In our example, we just consider two abstractions describing the position of two packages: π_1 and π_2. The atomic abstraction maps each value of the variable (L, R, and T) to an abstract state. In our example, each value is mapped to a different abstract state, without any shrinking. For example, abstract state 1 of π_1 corresponds to having package-1 in the truck.

In each iteration, the M&S algorithm merges two abstractions (rule M). The resulting abstraction is the synchronized product ($\otimes$) of both abstractions, i. e., it has an abstract state for each pair of states of the input abstractions. Non-atomic abstractions are represented with a two-dimensional table, in which rows and columns correspond to abstract states of the merged abstractions. In our example, table (M) shows the product of atomic abstractions π_1 and π_2. This table associates each pair of abstract states in π_1 and π_2, $\langle s_i^{\pi_1}, s_j^{\pi_2} \rangle$, to the ID of an abstract state. The synchronized product just needs to set a different id for each cell in the table, so that all the pairs $\langle s_i^{\pi_1}, s_j^{\pi_2} \rangle$ correspond to different abstract states. For example, abstract state 1 of $\pi_1 \otimes \pi_2$ represents states in which package-2 is in the truck and package-1 is at L.

The abstraction in Figure 6.1M can be merged again with a new atomic abstraction π_3. The result would be a new table with one column for each abstract state in π_3 and one row for each abstract state in $\pi_1 \otimes \pi_2$, for a total of 27 cells. As the number of abstract states grows exponentially with the number of merge steps, the size of the abstractions must be reduced at some point. This is done with the shrinking operation (rule S). The shrinking operation reduces the number of abstract states by applying a function γ that maps the abstract states to a new set of abstract states, producing a new abstraction, α_1. In our example, the mapping γ_1 corresponds to consider that packages are interchangeable. It is equivalent to have package-1 at L and package-2 at R or vice versa. Thus, $\gamma_1(2) = \gamma_1(6) = 3$ and abstract state $s_3^{\alpha_1}$ correspond to states in which $(v_1 = L \wedge v_2 = R) \vee (v_1 = R \wedge v_2 = L)$. Applying a mapping γ in the cascading tables is again immediate, changing the value of each cell in the table accordingly.

An additional lookup table stores the precomputed optimal cost for each abstract state in the final M&S abstraction α. During the search, the heuristic value of a particular state s is retrieved in two steps. The first step gets the abstract state $\alpha(s)$ from the *cascading-tables* representation and step 2 gets the heuristic value from the lookup table. To retrieve the abstract state associated with any state s, we recursively look up the tables, returning the abstract state ID associated with the values of s. For example, consider the partial state $\langle L, R\rangle$ in which package-1 is at L and package-2 is at R. The retrieval algorithm first checks the tables related to π_1 and π_2, obtaining abstract states 0 and 2, respectively. Then, these values are used to perform a lookup in the α_1 table, to retrieve the value 3. An additional lookup on a table of heuristic values, which is not shown in our example, is needed in order to retrieve the heuristic value of abstract state number 3.

6.5 Summary

In this chapter we have reviewed state-of-the-art abstraction heuristics for domain-independent planning: pattern databases (PDBs) and merge-and-shrink (M&S). While PDBs have had successful results in the past, they only use a subset of variables, which is an important limitation for some planning domains. M&S is a generalization of PDBs that derives flexible heuristics, addressing the main limitation of PDBs. In the following chapters we will make use of PDBs and M&S abstractions in combination with symbolic search.

Chapter 7

Symbolic Representation of Merge-and-Shrink Heuristics

Recent advances in heuristics for explicit-state search planning have not been extrapolated to the case of symbolic search. While symbolic search planners use symbolic PDBs, explicit-state search planners have benefited from alternatives such as M&S abstractions, a generalization of PDBs.

In this chapter, we consider the use of M&S heuristics in symbolic search planning. M&S heuristics precompute and store the heuristic values in a data structure called *cascading tables*. In order to evaluate sets of states at once, we represent the heuristic by means of decision diagrams. For these purposes, we study how the *cascading-tables* representation can be automatically transformed to a symbolic representation and prove that this is tractable for the M&S heuristics that use a linear merging strategy.

7.1 Introduction

According to the results of the IPC-2011, the most successful heuristics for cost-optimal planning are LM-CUT (Helmert and Domshlak, 2009) and M&S (Helmert et al., 2014). However, none of these heuristics have been used in symbolic heuristic search. Current state-of-the-art symbolic planners like GAMER make use of Symbolic PDB heuristics. As introduced in Chapter 6, using a heuristic in symbolic search is not straightforward since sets of states must be evaluated at once. An evaluation per state is possible, though the benefits of the symbolic search may be hindered. Abstraction heuristics are a good fit for symbolic search because they can precompute the heuristic value of every state and encode them in a symbolic representation, as a list of BDDs. In this chapter, we consider using M&S heuristics in symbolic search. The potential of M&S abstractions is clear not only because they achieved impressive results, but also because they are a generalization of the PDB heuristics currently used in symbolic search planning (Kissmann and Edelkamp, 2011; Kissmann, 2012).

In order to use M&S heuristics in symbolic search, we have to symbolically represent them with decision diagrams. We consider two symbolic representations for M&S heuristics: using an Algebraic Decision Diagram (ADD) or a list of BDDs. Both representations have equivalent complexity and there are polynomial transformations between them (see Proposition 2.1 on page 22). As we introduced in Chapter 2, symbolic A* uses the representation with a list of BDDs because it eases the evaluation of sets of states. On the other hand, ADDs are more appropriate for explicit A*, because only one lookup is necessary to evaluate a single state. Thus, using a symbolic representation of M&S heuristics is not only useful for symbolic search, but may also have advantages in the explicit-search case. The advantages of using a symbolic representation in explicit-state search have already been studied in the case of Pattern Database abstractions. Ball and Holte showed that ADDs may be used to compress PDBs in different planning domains with very good compression ratios up to four orders of magnitude, being only worse than a hash-table approach in a few cases (Ball and Holte, 2008).

However, the potential of the symbolic representation has not been studied in the case of M&S abstractions yet.[1] In this chapter we perform several contributions on this topic, divided into several sections:

- In Section 7.2, we present an algorithm to automatically transform the *cascading-tables* representation explained in Section 6.4.3 into a symbolic representation. We analyze under which circumstances we can ensure that the symbolic representation of M&S abstractions is tractable.
- In Section 7.3, we theoretically compare the ability to generate perfect heuristics of M&S heuristics and symbolic regression search.
- In Section 7.4, we empirically evaluate the symbolic representation of M&S abstractions. We also use M&S heuristics in symbolic heuristic planning and compare their performance to that of symbolic PDBs.

7.2 Symbolic Representation of Merge-and-Shrink

The *cascading-tables* representation of M&S is a data structure that represents the M&S abstraction heuristics. As explained in Section 6.4.3, the representation involves two mappings, the abstraction function $\alpha : \mathcal{S} \to \mathcal{S}^\alpha$ mapping states in $\mathcal{S}$ to *abstract states* in $\mathcal{S}^\alpha$ and the precomputed cost for the abstract states $h^*_\alpha : \mathcal{S}^\alpha \to \mathbb{R}^+_0 \cup \{\infty\}$. Combining both mappings results in the desired heuristic function, $h : \mathcal{S} \to \mathbb{R}^+_0 \cup \{\infty\}$.

The key observation is that, when a linear merging strategy is used, the *cascading-tables* representation of an M&S abstraction can be cast as an ADD. This means that it is possible to construct an ADD that represents the same function as the *cascading-tables* representation in polynomial time. We first develop the intuition behind this correspondence with two different examples. Then, we present a simple algorithm that computes the ADD representation of any given M&S heuristic, assuming a linear merging strategy. We conclude this section with our main result that the size of the resulting ADD is polynomially bounded with respect to the size of the *cascading-tables* representation and that our transformation algorithm runs in polynomial time.

Figure 7.1 illustrates the correspondence between the *cascading-tables* representation of a linear M&S heuristic and the ADD representation with a simple example: a simplified VISITALL task that ignores the location of the robot. In our example, there are five independent Boolean variables, v_1 to v_5. The goal is to make all variables true and each operator o_i makes variable v_i true. The initial state is not relevant for us, since we want to obtain a heuristic function that estimates the distance to the goal from any state in the state space. We run M&S with a linear merging strategy and bisimulation shrinking. M&S with bisimulation derives the perfect heuristic, which consists of counting the number of variables that remain false in the state.

[1] In an independent research, Helmert et al. (2014) also considered the relationship between the cascading-tables and the ADD representation.

α_1

v_1	
$\bot$	0
$\top$	1

α_2

v_2	$s_0^{\alpha_1}$	$s_1^{\alpha_1}$
$\bot$	0	1
$\top$	1	2

α_3

v_3	$s_0^{\alpha_2}$	$s_1^{\alpha_2}$	$s_2^{\alpha_2}$
$\bot$	0	1	2
$\top$	1	2	3

α_4

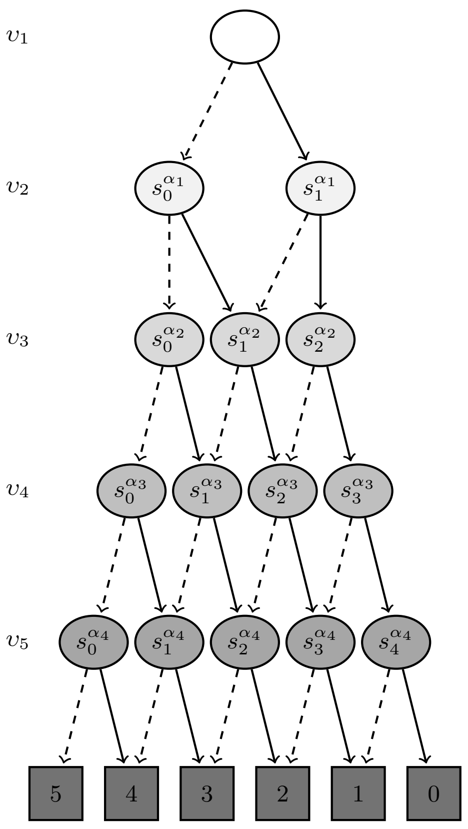
v_4	$s_0^{\alpha_3}$	$s_1^{\alpha_3}$	$s_2^{\alpha_3}$	$s_3^{\alpha_3}$
$\bot$	0	1	2	3
$\top$	1	2	3	4

α_5

v_5	$s_0^{\alpha_4}$	$s_1^{\alpha_4}$	$s_2^{\alpha_4}$	$s_3^{\alpha_4}$	$s_4^{\alpha_4}$
$\bot$	0	1	2	3	4
$\top$	1	2	3	4	5

	$s_0^{\alpha_5}$	$s_1^{\alpha_5}$	$s_2^{\alpha_5}$	$s_3^{\alpha_5}$	$s_4^{\alpha_5}$	$s_5^{\alpha_5}$
h	5	4	3	2	1	0

(a) Cascading tables M&S



(b) ADD M&S

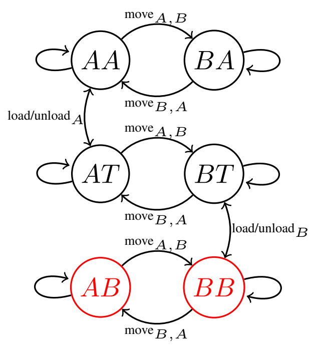
Figure 7.1: Relation between *cascading-tables* and ADD representation for M&S with a linear merging strategy. Each ADD layer corresponds to one cascading table and each ADD node corresponds to a column of the table associated with its layer.

Figure 7.1a depicts the *cascading-tables* representation of the M&S heuristic. As we are considering a linear merge ordering, at each step we add a new variable v_i. Each table corresponds to an abstraction α_i that considers i variables. Rows of the table directly correspond to different values of v_i, $\top$ and $\bot$ (note that for simplicity we are omitting the tables related to atomic abstractions). Each column of a table corresponds to abstract states of previous abstractions. For example, the value 1 in the table that represents α_2 corresponds to the column $s_1^{\alpha_3}$ of the table that represents the next abstraction, α_3.

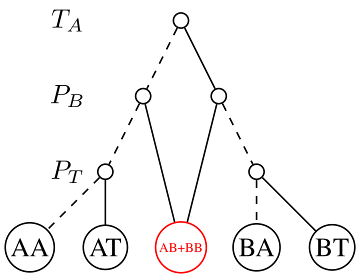
The table that represents each α_i, encodes equivalences between pairs $\langle s_{i-1}^{\alpha}, v_i \rangle$. For example, in α_2, $\langle s_1^{\alpha_1}, v_2 = \bot \rangle \equiv \langle s_0^{\alpha_1}, v_2 = \top \rangle \equiv s_1^{\alpha_2}$. Thus, both combinations are considered equivalent. In our example, each state $s_i^{\alpha_j}$ corresponds to having i variables true and the remaining $j-i$ variables false. In the last layer, the heuristic values of each abstract state $s_i^{\alpha_5}$ correspond to counting the number of variables that remain false: $5 - i$.

Figure 7.1b depicts the corresponding ADD representing the heuristic h^{α_5}. The figure is organized to highlight the similarity between both representations. Every level in the ADD corresponds to an intermediate abstraction α_i. Each abstract state of the intermediate abstraction is represented with an ADD node on the corresponding ADD level. To highlight this correspondence, ADD nodes are labeled in the figure with the abstract state they correspond to. In the last level, terminal ADD nodes correspond to the heuristic value of abstract states of the final M&S abstraction, $s_i^{\alpha_5}$. The tables in the *cascading-tables* representation are just an alternative to represent the ADD edges, i. e., the mapping from ADD nodes in one layer to the next one. $s_1^{\alpha_2}$ has two incoming edges: a 1-edge from $s_0^{\alpha_1}$ and a 0-edge from $s_1^{\alpha_1}$. These two edges correspond to the two cells with value 1 in the table representing α_2, since 0-edges correspond to $v_i = \bot$ and 1-edges to $v_i = \top$.

To build the analogy between the M&S construction process and an ADD, we can relate the two operations in M&S, merge and shrink, with the modifications that they imply in the ADD representation. As reflected in Figure 7.1, merging a new variable to a M&S abstraction corresponds to adding a new layer to the ADD. The shrinking operation that aggregates several abstract states into one, corresponds to the application of ADD reduction rules. Figure 7.2 shows an example where two abstract states, AA and AB, are shrunk. After the shrinking both AA and AB become equivalent in the abstract state space, so that they will get assigned the same heuristic value. In the corresponding ADD representation, this means that the nodes that represented AA and AB will become equivalent according to the reduction rules, because they represent exactly the same function. In Figure 7.2b the assignments $T_A \wedge P_B$ and $\neg T_A \wedge P_B$ that correspond to the package being at B point both to the same node.



(a) Shrinking in M&S.



(b) Reduction rule in ADD.

Figure 7.2: Correspondence between shrinking in M&S and ADD reduction rules. A task with a package, a truck, and two locations, A and B. The two abstract states in which the package is at B, AB and BB, are shrunk and get represented by the same ADD node, $AB + BB$.

The example of Figure 7.1b is very convenient to highlight the correspondence between columns/cells of each cascading table and ADD nodes/edges. Thanks to the simplicity of our example, there is a total correspondence between abstract states and ADD nodes. However, in a more general case, the correspondence may be slightly more complicated because other aspects must be contemplated:

1. SAS$^+$ variables may have more than two values. As explained in Chapter 2, a logarithmic number of BDD variables is used to represent each SAS$^+$ variable in those cases.

2. M&S may prune some abstract states if they are detected as unreachable or dead-ends in the abstract state space.

3. In the example of Figure 7.1b, bisimulation shrinking finds all the equivalences between abstract states, and there is a one-to-one correspondence between abstract states and ADD nodes. In a more general case, there may be more abstract states than ADD nodes. In those cases, ADD reduction rules determine that two or more abstract states are equivalent even though they were not shrunk. We will analyze the impact of ADD reduction rules in shrinking policies in Section 7.3.

Figure 7.3 shows a more general example of the mapping between the *cascading-tables* representation and an ADD. Our second example, based on one by Helmert et al. (2014), is a logistics task with two packages and a truck. There are two locations, left L and right R. The task has three different variables: p_1 and p_2 represent the location of the two packages, which can be at left L, at right R or in the truck T. The variable t represents the position of the truck, which can be placed at left L or at right R. Initially, the packages and the truck are at L and the goal is to carry both packages to the right.

The entire state space of the task is shown in Figure 7.3a and highlights the mapping induced by the M&S abstraction. In this case, we have chosen the abstraction mapping by hand to make an informative example. The abstract state spaces of the intermediate M&S abstractions are shown in Figure 7.3b.

The *cascading-tables* representation of the M&S heuristic shown in Figure 7.3c represents the mapping from states in the original state space to abstract states through three tables, α_1, α_2, and α_3. Figure 7.3c depicts the ADD representation of this M&S heuristic, following the same variable ordering as the linear merging strategy used by M&S. As before, each cascading table corresponds to an ADD layer, and each abstract state in the abstractions is represented through an ADD node. However, the picture is not as clear as in our first example due to the aforementioned aspects.

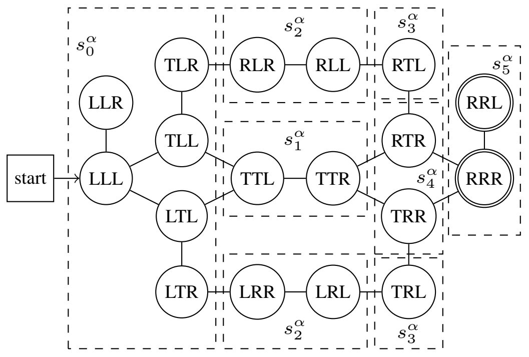
First, there are ADD nodes that are not associated with any abstract state. Those nodes are auxiliary nodes specifically created to cope with SAS$^+$ variables with more than two values. The root node needs to differentiate among values L, T and R, but ADD nodes only have two edges. Therefore, an auxiliary node is added in order to represent three outgoing edges from the root node.

Secondly, even though each abstract state is associated with an ADD node, some of them may be removed or aggregated by the ADD reduction rules. In our example, $s_0^{\alpha_2}$, $s_1^{\alpha_2}$, $s_2^{\alpha_2}$ and $s_5^{\alpha_2}$ are not necessary because, according to the cascading table, both edges point to the same result. Also, several abstract states are represented by a single ADD node, whenever they represent the same function. This is the case of $s_3^{\alpha_2}$ and $s_4^{\alpha_2}$, which are equivalent in the cascading tables. In summary, the ADD may be in a more compressed form because it takes advantage of knowing the full diagram, while M&S abstractions are constructed incrementally.

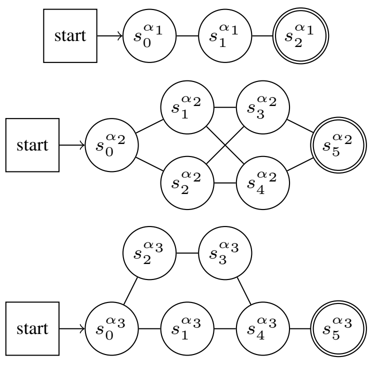
Finally, our example does not show any unreachable or dead end abstract state, but they can easily be managed as well. Every abstract state pruned by the M&S heuristic is associated with a terminal ADD node that corresponds to the value ∞.

7.2.1 M&S to ADD Algorithm

Given the one-to-one correspondence between M&S abstract states and ADD nodes, the algorithm to obtain the ADD representation of an M&S heuristic with linear merge is straightforward. Algorithm 7.1 computes the ADD from the *cascading-tables* representation of an M&S heuristic. The algorithm computes an ADD node for each abstract state of all intermediate abstractions in a bottom-up approach (as decision diagrams are usually built). ADD nodes are built with two functions: `ADD-constant` and `ADD-value`. `ADD-constant` generates a terminal node associated with an integer constant. `ADD-value` receives a variable v_i and a value in the domain of the variable



(a) Original state space, partitioned according to α_3.



(b) Abstract state spaces for α_1, α_2, and α_3.

α_1

v_1	
L	0
T	1
R	2

α_2

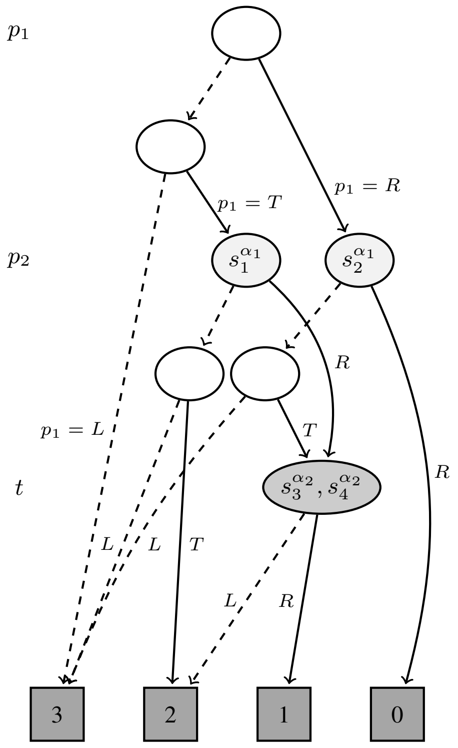
v_2	$s_0^{\alpha_1}$	$s_1^{\alpha_1}$	$s_2^{\alpha_1}$
L	0	0	2
T	0	1	3
R	2	4	5

α_3

v_3	$s_0^{\alpha_2}$	$s_1^{\alpha_2}$	$s_2^{\alpha_2}$	$s_3^{\alpha_2}$	$s_4^{\alpha_2}$	$s_5^{\alpha_2}$
L	0	1	2	3	3	5
R	0	1	2	4	4	5

	$s_0^{\alpha_3}$	$s_1^{\alpha_3}$	$s_2^{\alpha_3}$	$s_3^{\alpha_3}$	$s_4^{\alpha_3}$	$s_5^{\alpha_3}$
h	3	2	3	2	1	0

(c) Cascading tables M&S.



(d) ADD M&S.

Figure 7.3: Relation between *cascading-tables* and ADD representation for M&S. The linear merging strategy corresponds to the ordering p_1, p_2, and t.

Algorithm 7.1: M&S Heuristic to ADD

```
   Input: A list of cascading tables $T_1, \dots, T_n$ and heuristic values of $\alpha_n$, $h^*_{\alpha_n}$
   Output: The root node of an ADD that represents $h^*_{\alpha_n}$
1  for abstract state $s_i^{\alpha_n} \in \alpha_n$ do
2      $\text{ADD}_{n,i} \leftarrow$ ADD-constant($h^*_{\alpha_n}(s_i^{\alpha_n})$)
3  for layer $l \in [n-1 \dots 0]$ do
4      for abstract state $s_i \in \alpha_l$ do
5          if $T_l[s_i] = \mathit{pruned}$ then
6              $\text{ADD}_{l,i} \leftarrow$ ADD-constant($\infty$)
7          else
8              $\text{ADD}_{l,i} \leftarrow \sum_{d \in v_l} \big($ADD-value($v_l, d$) $\times \text{ADD}_{l+1,T_l[i,d]}\big)$
9  return $\text{ADD}_{0,0}$
```

$d \in D_{v_i}$ and returns an ADD that represents a function f such that $f(d) = 1$ and $f(d') = 0$ for all $d' \in D_{v_i}, d' \neq d$.

The algorithm maintains a matrix named ADD with one entry per abstract state in every intermediate abstraction. Let $\text{ADD}_{i,j}$ denote the ADD node that corresponds to the abstract state $s_j^{\alpha_i}$. Note that we do not explicitly check the ADD reduction rules, since the CUDD BDD library (Somenzi, 2012) automatically applies them (e. g., all the calls to `ADD-constant(∞)` will always return a reference to the same node). Since several abstract states may be represented with the same node, the ADD matrix stores only a reference to the nodes.

First, the algorithm generates one terminal ADD node per abstract state in the last layer, with their heuristic value (line 2). Again, the same node will be used for states with the same heuristic value (their entries in the ADD matrix reference to the same ADD node).

Then, nodes in the previous layer can be constructed pointing to nodes in layers already computed. A loop iterates over all layers in a bottom-up fashion (line 3). To construct the next layer of the ADD, the algorithm iterates over all abstract states in the corresponding abstraction (line 4). For every abstract state in the current layer, the cascading table is used to know which nodes must be pointed to from the newly generated node. Nodes corresponding to states pruned by M&S are directly assigned a constant heuristic value of ∞. Otherwise, the node representing the state corresponds to the function $\sum_{d \in v_l}$ ADD-value(v_l, d) $\times \text{ADD}_{l+1,T_l[i,d]}$. This function just returns a node that points to the node indicated by the cascading table ($T_l[i,d]$) for every value d of variable v_l.

As an intuition to understand how this expression works, sum ($+$) and multiplication ($\times$) operations over ADDs that represents 0-1 functions are equivalent to disjunction ($\vee$) and conjunction ($\wedge$) over BDDs, respectively. Thus, the multiplication ADD-value(v_l, d) $\times \text{ADD}_{l+1,T_l[i,d]}$ sets the node that represent the abstract state indicated by the cascading table, $T_l[i,d]$ as the value assigned to value d. The sum can be interpreted as a disjunction over all values of the variable. The simpler example is when v_l is a binary variable, since the result of `ADD-node` is a node whose 1-edge points to $\text{ADD}_{l+1,T_l[i,\top]}$ and whose 0-edge points to $\text{ADD}_{l+1,T_l[i,\perp]}$. When v_l has more than two possible variables intermediate auxiliary nodes are needed, as shown in Figure 7.3.When all layers have been constructed, the algorithm ends returning the reference to the root node of the ADD, $\text{ADD}_{0,0}$.

Adapting algorithm 7.1 to compute the ADD representation of the M&S abstraction function (that maps states in S to abstract states in S_n^{α}) is straightforward, just assigning terminal nodes the

ID of each abstract state instead of their heuristic value.

7.2.2 Computational Complexity Analysis

After presenting an algorithm to automatically compute the ADD representation of any M&S heuristic constructed with a linear merging strategy, we analyze the theoretical computational complexity of the representation and the algorithm to compute it.

First, we compare the size of both representations. Theorem 7.1 bounds the size of the ADD representation with respect to the *cascading-tables* representation, proving that it has, in the worst case, a linear overhead. Later, in Section 7.3 we prove that the ADD representation might be exponentially smaller than the *cascading-tables* representation generated by M&S with bisimulation.

Theorem 7.1. *Let $\alpha_1, \ldots, \alpha_n$ be a list of M&S abstractions generated with a linear merging strategy $v_1, \ldots, v_n$ such that $\alpha_{i+1} = \gamma(\alpha_i \otimes \pi_{i+1})$. Then, the symbolic ADD representation of the function $h^{\alpha_n} : S \to \mathbb{R}_0^+ \cup \{\infty\}$ under variable ordering $v_1, \ldots, v_n$ has at most $\sum_{i=0}^{n-1} |\alpha_i|(|dom(v_{i+1})| - 1)$ nodes.*

Proof. Our bound comes from the sum of the nodes of each layer l, which is bounded by $|\alpha_l|(|dom(v_{l+1})| - 1)$. Layer l has at most one node for each abstract state in $|\alpha_l|$, plus intermediate nodes to connect with layer $l + 1$, if the variable v_{l+1} has more than two values.

To discern between all the values of a variable v_i, we need to build a binary tree with at most $|dom(v_i)| - 1$ nodes, so in the worst case the number of nodes employed in layer l is $|\alpha_l|(|dom(v_{l+1})| - 1)$. □

Theorem 7.1 is our main result in this section because it bounds the size of the ADD that represents the M&S heuristic with respect to the size of the abstractions involved. In the following, we extend this result with several corollaries that simply extend the bound to other related cases. Corollary 7.3 ensures that the ADD representation of an M&S heuristic with a linear merge strategy can be computed in polynomial time. Note that all these bounds are only guaranteed if the variable ordering of the ordered decision diagrams corresponds to the M&S linear merging strategy. Indeed, since the variable ordering may have exponential impact on the size of the representation, if the variable orders differ no bound is established at all by Theorem 7.1.

Corollary 7.2. *Let $\alpha_1, \ldots, \alpha_n$ be a list of M&S abstractions generated with a linear merging strategy $v_1, \ldots, v_n$ such that $\alpha_{i+1} = \gamma(\alpha_i \otimes \pi_{i+1})$. Let B be the bound for the number of ADD nodes established by Theorem 7.1, $B = \sum_{i=0}^{n-1} |\alpha_i|(|dom(v_{i+1})| - 1)$. Then Algorithm 7.1 computes the ADD representation of the M&S heuristic, h^{α_n}, in time $\mathrm{O}(B \log(B))$.*

Proof. Algorithm 7.1 computes the ADD representation of the M&S heuristic by creating the nodes iteratively, so that it suffices to sum the time in generating each node. Every ADD node is constructed with a call to `ADD-constant` (which is performed in constant time) or with the standard ADD sum and product operations in the expression $\sum_{d \in v_l} \left(\texttt{ADD-value}(v_l, d) \times \text{ADD}_{l+1, T_l[i,d]}\right)$. Since the ADDs describing values d of a variable, $\texttt{ADD-value}(v_l, d)$, are all disjoint, this operations can be performed in time proportional to the number of created nodes. Thus, a constant number of operations is needed for each node, plus a lookup in the table of nodes to ensure the ADD reduction rules. Therefore, at most B nodes have to be constructed and each node takes $\mathrm{O}(\log(B))$ time. □

In order to guarantee the correct termination of M&S, shrinking strategies usually take a parameter M, which forces the abstractions to be small enough so that $|\alpha_i||dom(v_{i+1})| \leq M$. In those cases, the size of the ADD can be bounded by Mn, where n is the number of variables and M is the pre-defined maximum number of states.

Corollary 7.3. *Let $\alpha_1, \dots, \alpha_n$ be a list of M&S abstractions generated with a linear merging strategy $v_1, \dots, v_n$ such that $\alpha_{i+1} = \gamma(\alpha_i \otimes \pi_{i+1})$. Then, the symbolic ADD representation of α_n under variable ordering $v_1, \dots, v_n$ has at most nM nodes.*

Proof. The statement is directly derived from Theorem 7.1 and the definition of M. □

Corollary 7.3 ensures that we can use the ADD representation in practice, fixing a predefined constant to keep the complexity of generating the heuristic under a linear factor.

Corollary 7.4. *The size of the ADD representation of an M&S heuristic has, at most, a linear overhead with respect to its* cascading-tables *representation.*

Proof. This is directly derived from Theorem 7.1. The *cascading-tables* representation has one table for each layer with $|\alpha_l||dom(v_{l+1})|$ entries. The ADD representation uses $|\alpha_l|(|dom(v_{l+1})| - 1)$ nodes. Even if the number of ADD nodes is smaller than the number of table entries, there may be a linear overhead due to the memory needed for each ADD node with respect to each table entry. □

Corollary 7.4 is an important result because it guarantees that substituting the *cascading-tables* representation with an ADD has, at most, a linear overhead.[2] In Section 7.3 we will see that the ADD reduction rules might exponentially reduce the size of the *cascading-tables* representation obtained with state-of-the-art shrinking strategies. In this case, the reduction rules can be applied over the *cascading-tables* representation as well, but only after the whole heuristic has been generated — as opposed to symbolic search, which generates BDDs in compact form.

Corollary 7.5. *Let $\alpha_1, \dots, \alpha_n$ be a list of M&S abstractions generated with a linear merging strategy $v_1, \dots, v_n$ such that $\alpha_{i+1} = \gamma(\alpha_i \otimes \pi_{i+1})$.*

Then, the symbolic ADD representation of the function $\alpha_n : S \to S^{\alpha_n}$ under variable ordering $v_1, \dots, v_n$ has at most $\sum_{i=0}^{n-1} |\alpha_i|(|dom(v_{i+1})| - 1)$ nodes.

Proof. The proof of Theorem 7.1 applies in this case as well, since the *cascading-tables* representation represents the mapping from states to abstract states. □

Corollary 7.5 implies that M&S heuristics and M&S abstraction mappings can be efficiently represented with ADDs. This fact will be exploited later in Chapter 8.

Corollary 7.6. *Let an M&S heuristic run with a bound M for the maximum size of an abstraction, $|\alpha_i \otimes \pi_{i+1}| \leq M$. The representation of an M&S heuristic, h^{α_n}, as a sequence of BDDs associated with heuristic values $H = \langle h_0, \dots, h_{\max} \rangle$ can be computed in time and space $\mathrm{O}\left(|H| nM \log(nM)\right)$.*

Proof. It immediately follows from Corollary 7.3 and the linear transformation from an ADD a to a sequence of k BDDs in time and space $\mathrm{O}\left(k|a|\right)$ (see Proposition 2.1 in Section 2.7). □

Finally, Corollary 7.6 proves that there exists an efficient transformation of the M&S heuristic to a sequence of BDDs. This is a suitable encoding for heuristics in symbolic A* search, so we have enabled the use of M&S heuristics in symbolic A* search.

[2]Posterior research by Helmert et al. studied the bounds for non-linear merge strategies, proving that M&S with non-linear merge strategies is more expressive and cannot be transformed to an ADD with polynomial overhead (Helmert et al., 2015).

7.3 Complexity of Computing Perfect Heuristics

As we introduced in Chapter 6, running M&S with any merging strategy and bisimulation shrinking is a domain-independent method to obtain the perfect heuristic for every state in the domain. One of the major theoretical strengths of M&S abstractions is that they are capable of computing perfect heuristics in polynomial time for some domains where PDBs or other planning heuristics cannot. A distinguishing example is the GRIPPER domain, where one robot has to carry balls from one room to another. The state space of the task is exponential in the number of balls the robot must carry, since all balls can be used interchangeably. Bisimulation is able to exploit these symmetries to construct the perfect heuristic for the domain (Nissim et al., 2011).

Another method to derive the perfect heuristic for a planning task is symbolic regression search. Again, previous work proved that the perfect heuristic for the GRIPPER domain can be derived in polynomial time and space using symbolic search (Edelkamp and Kissmann, 2008a).

In the previous section, we have shown that symbolic representations can represent M&S heuristics with at most a linear overhead with respect to the *cascading-tables* representation. The fact that any M&S heuristic can be represented as a sequence of BDDs of polynomial size triggers the question of whether symbolic search can be used instead. In this section, with the aim of improving our understanding of the relationship between both techniques, we compare the ability of both approaches to represent and compute perfect heuristics.

7.3.1 Representation of Perfect Heuristics

In this section, we compare the size of the symbolic representation with that of M&S abstractions derived with bisimulation shrinking. While the overhead to symbolically representing any M&S heuristic is at most linear in the size of the planning task, we show that the opposite does not hold, i. e., there exist planning tasks in which the abstraction derived by M&S with bisimulation shrinking is exponentially larger than the corresponding symbolic representation.

First, we observe that for any (e. g., the perfect) heuristic —no matter how it is computed— the ADD representation under a given variable ordering has to be the same by the uniqueness property of reduced ordered decision diagrams. Given the one-to-one correspondence between intermediate M&S abstractions and ADD layers that we established in the previous section, we can focus on one intermediate abstraction and compare the abstraction size after applying bisimulation shrinking with the number of ADD nodes in that layer.

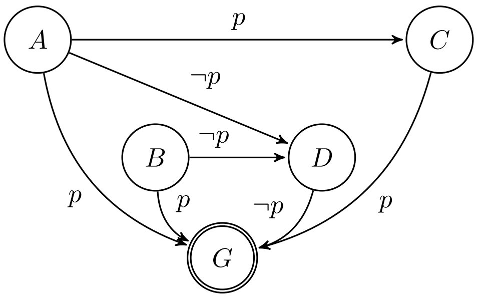
To ease the comparison, we may define the ADD shrinking strategy as: aggregate every state that will be represented by the same ADD node. Of course, this strategy is not practical because we would need to construct the ADD that represents the perfect heuristic first. Nevertheless, the ADD shrinking strategy guarantees to derive the coarsest abstraction that preserves the perfect heuristic of the original problem. Now, our question has been reduced to compare the size of an abstract state space after performing bisimulation or ADD shrinking.

To determine whether some abstract states will be aggregated by ADD shrinking (i. e., represented by the same ADD node), we rely on the definition of equivalent states (Definition 7.1). Informally, we say that two abstract states are *equivalent*, if for all their completions, the resulting states have the same heuristic value.

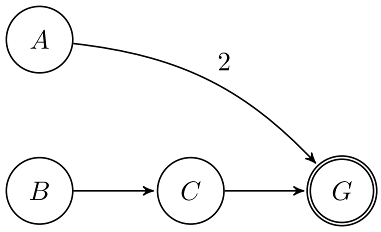
Definition 7.1 (Equivalent abstract states)**.** *Let α be an abstraction with relevant variables $\mathcal{V}^\alpha$. Let s^α and t^α be two abstract states of α. We say that s^α and t^α are equivalent, $s^\alpha \equiv t^\alpha$, if and only if for every assignment d to non-relevant variables $\mathcal{V} \setminus \mathcal{V}^\alpha$ the goal distance is the same: $h^*(s^\alpha \cup d) = h^*(t^\alpha \cup d)$.*

The shrinking strategy obtaining the coarsest abstraction that preserves the solution costs to derive the perfect heuristic is just to shrink all equivalent abstract states. Aggregating a pair of non-equivalent states would induce some error in the heuristic. The exponential gap depends on whether the shrinking strategy can predict without error if two abstract states will become ADD equivalent after merging all the remaining variables and apply the corresponding shrinking.

Next, we construct a couple of intuitive examples where bisimulation shrinking is not able to compute the most reduced form of the heuristic. Figure 7.4 shows two examples that highlight the limitations of bisimulation shrinking. Both examples contain abstract states that are equivalent but not bisimilar. The transition labels have already been reduced so that they refer to variables that have not yet been merged. In the example of Figure 7.4a, these labels correspond to preconditions over a binary variable p and all the transitions have unit costs. In Figure 7.4b actions can be performed independently of other parts of the problem but have different action costs.



(a) Irrelevant transitions. A and B are equivalent but not bisimilar due to the irrelevant transition $A \to C$.



(b) Alternative paths. A and B are equivalent but not bisimilar because their paths to the goal differ.

Figure 7.4: Examples of bisimulation versus ADD equivalence. A, B, C, D, and G are abstract states in intermediate M&S abstractions. In (a), p and $\neg p$ are variable assignments that serve as precondition of the operators that support the transitions. In (b), the transition $A \to G$ has cost 2.

If two abstract states are equivalent, their corresponding ADD nodes can be unified according to the ADD reduction rule (2) (see the definition of ROBDD on page 19). It is easy to see that states A and B in the example are equivalent because in case that p holds both have a cost of 1, while if $\neg p$ holds both have a cost of two. However, they are not bisimilar because B does not have any transition to C. Obviously, states C and D are not bisimilar, given that their transitions have different labels. Hence, no pair of states is reduced by bisimulation.

Bisimulation cannot detect all pairs of equivalent abstract states because of two reasons:

- Irrelevant transitions: when the abstract state incorporates transitions that are not in the optimal plan of any concrete state. Since in the end only the distance to the goal matters, those transitions that are not part of an optimal path should not be taken into account by bisimulation. In the example of Figure 7.4a, A and B are equivalent but not bisimilar due to the irrelevant transition $A \to C$. Checking if a transition is necessary in any optimal path (e. g., with path relevance analysis (Scholz, 2004; Haslum et al., 2013)) is not known to be tractable as it needs to consider the exponential number of combinations of values of the variables that have not been merged. Some approaches try to consider only a subset of transitions when computing the bisimulation (Katz et al., 2012) but either they do not guarantee perfect heuristics or do not reduce all the equivalent states.

- Alternative paths: when two abstract states have solutions of the same cost but through different states. In this case, the real costs are unknown to bisimulation, so that it is not able to prove the states to be equivalent. Label reduction improves on this by considering more transitions as being equivalent, but it is unable to reason about complete paths. In the example of Figure 7.4b, abstract states A and B are equivalent but not bisimilar. In this case all the transitions are relevant (i. e., used in the optimal plan from a concrete state). What bisimulation does ignore in this case is that the paths $A \xrightarrow{2} G$ and $B \xrightarrow{1} C \xrightarrow{1} G$ have the same cost.

Finally, we show that for a given task whose perfect heuristic may be represented in a polynomial ADD, the M&S heuristic with bisimulation shrinking requires exponential memory. It is possible to extend the example by adding an exponential number of equivalent states that are not bisimilar because they have different transitions that are not needed by any of their optimal paths. Therefore, this may cause an exponential gap between the size of the intermediate abstraction and the final reduced heuristic.

Theorem 7.7. *There exist families of planning tasks $\{\Pi_n\}$, merging strategies, and variable subsets $\{V_n\}$, so that the ADD representation of the perfect heuristic is exponentially smaller than the* cascading-tables *representation of the M&S heuristic with bisimulation with exact label reduction.*

Proof. Consider the family of planning tasks with variables $\{v_1, \dots, v_n, g\}$, where $v_1, \dots, v_n$ are Boolean ($\{0, 1\}$) and g has domain $\{1, \dots, n+1\}$. The initial state sets all variables to 0, the goal is $v_i = 1$ for all $i \in \{1, \dots, n\}$, $g = n + 1$ and the actions are:

- a_i: empty precondition; effect $\{v_i = 1\}$.
- $a_{g,i}$: precondition $\{v_i = 1, g = i\}$; effect $\{g = i + 1\}$.

Say our variable order is $v_1, \dots, v_n, g$.

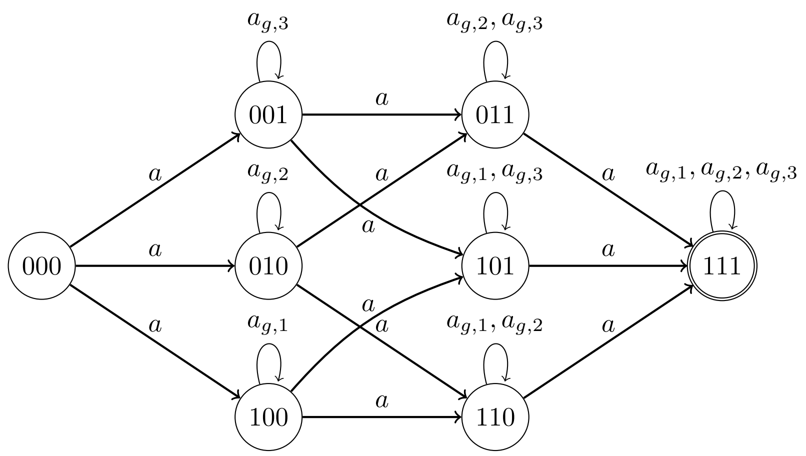
We first show that bisimulation shrinking with exact label reduction must yield exponential-size abstractions at some point during the abstraction process. Consider the set of states S^{V_k} after k merging steps, i. e., over variables $V_k := \{v_1, \dots, v_k\}$. Figure 7.5 depicts an example of the abstract state space after merging the first three variables, v_1, v_2, and v_3 and applying exact label reduction. Labels a_1, a_2, and a_3 have been reduced into label a, since they do not depend on other variables. However, labels $a_{g,i}$ cannot be reduced by exact label reduction because they are not equivalent in two different abstract state spaces and do not subsume each other (Sievers et al., 2014).

Running bisimulation, any two states differ on the subset of v_i with value 1. As each v_i with value 1 yields an outgoing (self-loop) label of the form $(\{g = i\}, \{g = i+1\})$, and as all these labels are different, no two states are bisimilar. Therefore, no pair of states is aggregated by bisimulation even with label reduction and the size of the abstract state space is exponential in k, 2^k.

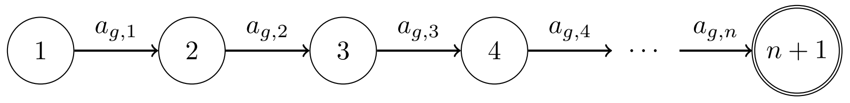
Now, we show that the ADD representation of the heuristic under the same variable ordering is of polynomial size. The key here is that the optimal plan from any state has to make all variables v_i true because they are part of the goal. The order in which the variables v_i are made true does not matter and they do not have any interaction with variable g. Thus, any valid plan from an arbitrary state with $g = i$ requires to perform exactly one move for every v_i variable that remains false. Therefore, at any given point with k variables, there are only $k + 1$ different abstract states, which correspond to the number of variables that are true in the state. □

7.3.2 Computing Perfect Heuristics

The results of Theorems 7.1 and 7.7 imply that the symbolic representation of perfect heuristics dominates the *cascading-tables* representation derived with M&S and bisimulation, i. e., it may be



(a) Projection over $v_1 \otimes v_2 \otimes v_3$.



(b) Atomic projection over g.

Figure 7.5: Example of planning task with exponential bisimulation and polynomial ADD representation. Bisimulation does not shrink any pair of states, even though states with the same number of 1s could be aggregated without any loss.

exponentially more succinct in some cases and, in the worst case, there is at most a linear overhead. Therefore, for any task in which M&S can derive the perfect heuristic in polynomial time, all the BDDs in the symbolic regression search are of polynomial size.

However, this argument does not guarantee that they can be derived in polynomial time and space in the same cases as M&S. Symbolic regression search iteratively constructs the reduced BDDs for every heuristic value through image operations. Even if all the BDDs are polynomial, the image operation that generates a BDD from another could take exponential time or memory. In conclusion, no technique dominates the other, though the advantages in the symbolic representation suggest that symbolic regression search could be stronger in practice.

7.4 Empirical Evaluation

Despite the theoretical results about the symbolic representation of the M&S heuristic, several questions remain unanswered about the practical usefulness of such a representation and the potential of the M&S framework compared to symbolic PDBs. This section provides experimental results of these techniques in benchmark domains in order to shed some light on these questions. We divided our evaluation in three different parts:

1. An evaluation of the memory used by the ADD representation of M&S compared to the *cascading-tables* representation obtained with different typical merging and shrinking strategies. We compare the memory used by both representations, as well as the time needed to perform the conversion.

2. An evaluation of the power of M&S with bisimulation and symbolic regression search to derive the perfect heuristics in our benchmark domains.

3. An evaluation of the performance of M&S and symbolic PDB heuristics to inform explicit and symbolic versions of A* search.

In our experiments we considered different parameters for the merge and shrink strategies. We used the default merge strategies in the FAST DOWNWARD repository.[3] These strategies correspond to a variable ordering that might also be used in symbolic search:

- *Level* (*lev*): Selects variables in FAST DOWNWARD or GAMER ordering. In combination with FAST DOWNWARD ordering, this means that goal variables are selected first.

- *Reverse level* (*rev*): Level ordering reversed.

- *Random*: random ordering as baseline for comparison.

- *Cggoal-lev* (*cgl*): Skips variables that are not connected in the causal graph to previously selected variables. If none exists, a goal variable is selected. Ties are broken by the level criterion.

- *Goalcg-lev* (*gcl*): Selects all the goal variables first, then variables connected in the causal graph to previously selected variables. Ties are broken by the level criterion.

- *Cggoal-rnd* (*cgr*): Skips variables that are not connected in the causal graph to previously selected variables. If none exists, a goal variable is selected. Ties are broken randomly.

All the merging strategies except *random* can be used with the FAST DOWNWARD and GAMER orderings, which amounts to a total of 11 merging strategies. Both orderings are usually contradictory, since FAST DOWNWARD places the goal variables at the end of the ordering and GAMER pretends to place close together the variables that are highly related, so that goal variables are usually placed in the middle of the ordering. o Regarding the shrinking strategies, we considered three different strategies, *fh-shrinking* (fhs), *bisimulation* (bop) and *greedy-bisimulation* (Nissim et al., 2011) (gop), presented in detail in Section 6.4.2 on page 98. We use different bounds for the maximum number of abstract states in any intermediate abstraction, M. The M&S planner that participated in IPC-2011 used a limit of M =200,000 with bop and gop without any limit, $M = \infty$. We denote these strategies as bop200k and gop, respectively, assuming no limit on M whenever it is not specified.

7.4.1 Empirical ADD Size of M&S Heuristics

First, we analyze the size of the ADD representation of different M&S heuristics. Table 7.1 shows the results of the two M&S configurations that took part in the IPC-2011, bop200k and gop with the *FD-rev* and GAMER orderings. For each domain, we report the number of instances in which the M&S heuristic was successfully generated as well as the size of the largest ADD for each domain. Surprisingly the ADDs are small, especially for the gop shrinking strategy, showing that not much memory is spent once the ADD has been computed. Unsurprisingly, the variable ordering also plays an important role in the ADD size.

[3] FAST DOWNWARD version 3288 of December 2013. Later versions include non-linear merging strategies (Sievers et al., 2014).

	FD-rev				*Gam-lev*			
	bop200k		gop		bop200k		gop	
	i	n	i	n	i	n	i	n
Airport (50)	22	706432	50	806	21	715013	49	10576
Barman (20)	20	143080	20	55	19	75275	20	95
Blocksworld (35)	35	164671	35	2766	35	57569	24	899
Depot (22)	10	217734	13	6476	11	215447	11	6237
Driverlog (20)	17	303660	15	2174	19	245025	11	18125
Elevators08 (30)	30	41372	1	60326	30	91342	6	232059
Elevators11 (20)	20	41372	0	–	20	91342	4	232059
Floortile11 (20)	20	552901	4	6815	20	1292520	6	64185
Freecell (80)	4	338281	80	270	13	194784	80	1152
Grid (5)	5	145870	3	305	0	–	2	511
Gripper (20)	20	962803	20	1860	20	113384	18	1760
Logistics 00 (28)	28	821417	28	7337	28	350206	21	3914
Logistics 98 (35)	9	438357	28	54939	12	85431	5	1488
Miconic (150)	150	836742	150	500	150	373233	150	560
Mystery (30)	8	168185	19	275	12	173622	19	754
NoMystery11 (20)	20	216066	20	1270	9	113875	14	862
Openstacks08 (30)	30	182310	9	152647	30	19428	10	1684
Openstacks11 (20)	20	182310	4	152647	20	14506	5	1684
Openstacks06 (30)	12	556673	30	30201	16	847680	20	2761
ParcPrinter08 (30)	26	2966769	30	74321	0	–	30	94558
ParcPrinter11 (20)	19	2966769	20	9929	0	–	20	11128
Parking11 (20)	0	–	0	–	6	112835	0	–
Pathways-noneg (30)	30	1025527	30	864	23	119140	30	981
Peg-Solitaire08 (30)	30	46974	6	949	30	147271	6	1129
Peg-Solitaire11 (20)	20	36491	0	–	20	197746	0	–
Pipesworld-nt (50)	9	95772	50	53	13	29905	50	92
Pipesworld-t (50)	9	54466	32	213080	15	174846	20	20109
PSR-small (50)	50	40124	50	20791	50	108614	50	2118
Rovers (40)	30	264481	40	9187	26	114840	33	5991
Satellite (40)	27	1609421	35	17918	25	185108	18	12326
Scanalyzer08 (30)	19	326022	9	29400	15	101179	6	6046
Scanalyzer11 (20)	14	326022	6	29400	9	101179	3	6046
Sokoban08 (30)	30	3749	3	4802	29	66898	2	13440
Sokoban11 (20)	20	3749	1	4802	20	66898	1	13440
Tidybot11 (20)	1	250	20	18	7	15097	20	27
TPP (30)	16	420898	30	20028	16	213965	26	14965
Transport08 (30)	27	196439	30	1752	27	59763	17	781
Transport11 (20)	20	295059	20	944	20	72647	20	1111
Trucks (30)	24	266430	30	277	23	319845	30	731
VisitAll (20)	20	3276750	20	7388	20	1137626	12	738
Woodworking08 (20)	30	630795	27	533038	30	174217	26	10846
Woodworking11 (20)	20	407281	20	533038	20	153405	18	8201
Zenotravel (40)	15	231883	20	8636	15	196317	13	1595

Table 7.1: Number i of instances with M&S heuristic and maximum number n of ADD nodes over all instances for each domain.

In order to measure the benefits of using an ADD to represent the heuristic, Figure 7.6 compares the memory used by the ADDs against the *cascading-tables* representation of M&S heuristics. The results show an advantage for the ADD representation, which in some cases is up to four orders of magnitude more succinct than the *cascading-tables* representation. This difference is not due to the data-structure being used, but due to the application of ADD reduction rules, which could be applied as well in the *cascading-tables* representation. Thus, the main conclusions of this comparison are not about the relation between a tabular and a decision diagram representation, but about the precision of the shrinking strategies. In the cases where the ADD representation uses much less memory than the tabular representation it means that more shrinking could have been applied without any loss in the heuristic accuracy.

Nevertheless, as M&S abstract state spaces are explicitly represented, all successfully generated M&S abstractions are quite small and can be represented by any of the two representations in less than 10 MB. Even though the ADD compression does not have practical advantages, the fact that the ADD representation is several orders of magnitude smaller implies that bisimulation shrinking is too conservative for computing M&S heuristics even with label-reduction techniques. This suggests that there might still be opportunities for designing better shrinking strategies. Finally, comparing the variable orderings, GAMER ordering allows ADDs to obtain more compression, especially in combination with greedy bisimulation.

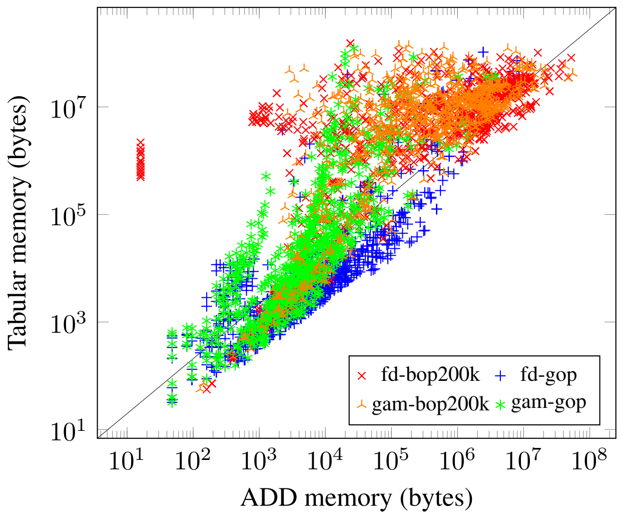


Figure 7.6: Memory spent in bytes by the ADD and tabular representations of M&S heuristics.

7.4.2 Computing Perfect Heuristics: Comparing M&S with Bisimulation and Symbolic Regression

As argued in Section 7.3.2, there are theoretical reasons that support the hypothesis that symbolic regression dominates M&S with bisimulation and label reduction to generate perfect heuristics. Generating the perfect heuristic is at least as hard as solving the problem in domains without 0-cost actions and often much harder. Nevertheless, even if this is not the most efficient way to optimally solve planning problems, it is interesting to compare the strengths of symbolic search and M&S.

In this section, we directly compare both approaches experimentally. We ran M&S with bisimulation and non-exact label reduction (Helmert et al., 2014) (M&S^b) and symbolic regression (sym_{bw}) on the complete set of benchmarks and measured the coverage and time spent in solving each problem. The comparison is not completely fair for M&S for two reasons. On the one hand, M&S computes the perfect heuristic value, $h^*(s)$, for every reachable state from s_0 and symbolic regression stops as soon as the initial state has been found. In practice, however, both solve the problem with no additional search effort in domains without zero cost actions. On the other hand, M&S does not take advantage of state invariants, while we use the state-invariant constraints in symbolic regression as described in Chapter 4. For a fairer comparison, we included a restrained version of symbolic regression (sym^-_{bw}) that does not make use of state-invariant constraints to prune the search nor stops when finding the initial state. In this case, the comparison is advantageous for M&S because it can avoid computing the heuristic for some unreachable states.

Table 7.2 shows the total coverage of the three approaches under different variable orderings. Experimentally, symbolic regression outperforms M&S^b in most domains, even when considering the restricted version sym^-_{bw}. While M&S^b runs out of memory in most cases, the performance of sym_{bw} is not too far away from state-of-the-art cost-optimal planners.

Even though the advantage of sym_{bw} over M&S^b is very stable across all the variable orderings, both techniques disagree in which are the best orderings. In general, the best orderings for sym_{bw} are GAMER variants, *Gam-lev* and *Gam-rev*. GAMER ordering optimization does not distinguish between top and bottom variables, so *Gam-lev* and *Gam-rev* are very similar except for a random tie-breaking. GAMER orderings work well with M&S^b, but they are worse than *FD-lev*.

	sym^-_{bw}	M&S^b	sym_{bw}
Random	9.56 (394)	6.96 (278)	12.85 (521)
FD-cggoal-lev	10.87 (498)	8.77 (381)	13.74 (611)
FD-cggoal-rnd	10.24 (469)	7.85 (325)	13.71 (609)
FD-goalcg-lev	9.95 (420)	8.23 (350)	12.83 (531)
FD-lev	10.03 (425)	7.55 (322)	13.15 (542)
FD-rev	10.65 (454)	8.30 (328)	13.81 (593)
Gam-cggoal-lev	11.56 (525)	8.41 (354)	15.51 (**675**)
Gam-cggoal-rnd	10.48 (472)	7.36 (297)	13.86 (601)
Gam-goalcg-lev	10.00 (426)	7.35 (305)	14.06 (569)
Gam-lev	11.60 (517)	7.45 (300)	15.53 (667)
Gam-rev	11.57 (512)	7.23 (290)	**15.71** (669)

Table 7.2: Total coverage of symbolic regression search and M&S with bisimulation. sym^-_{bw} disables state invariant pruning and explores the entire state space, i. e., does not stop when finding the initial state.

In order to measure the relative solving time of both techniques, Figure 7.7 compares the time spent by both techniques to generate the perfect heuristic for each of the instances. As mentioned before, M&S^b runs out of memory long before reaching the time limit, so beyond 100 seconds it solves very few instances. The plot shows that there are a few domains where M&S with bisimulation outperforms symbolic regression search under certain variable orderings. For example, with *FD-lev* variable ordering, M&S generates perfect heuristics in PEG-SOLITAIRE in less than 100 seconds while symbolic regression is unable to solve most instances. However symbolic regression outperforms M&S in most domains, except PEG-SOLITAIRE, TRANSPORT and ELEVATORS. On the other hand, the dominance of sym_{bw} over M&S^b is almost total when using *Gam-lev* ordering, including those domains. This reinforces the idea that variable ordering has a huge impact on the performance of both techniques.

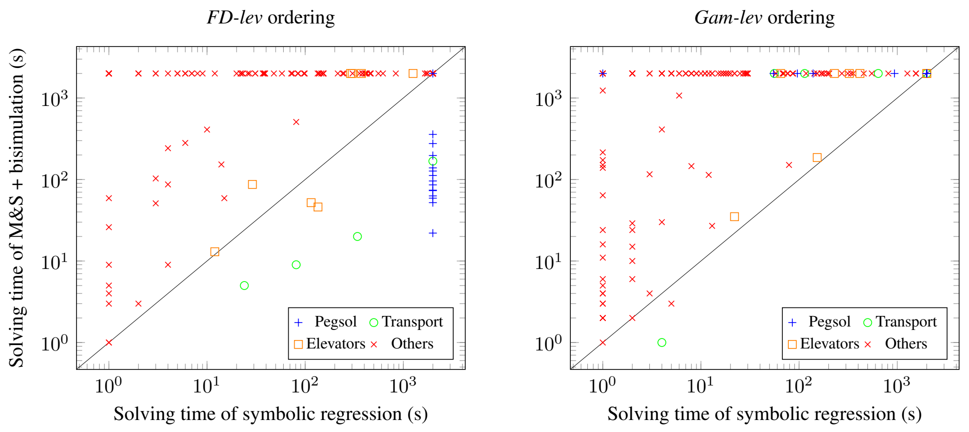


Figure 7.7: Solving time of symbolic regression search versus M&S with bisimulation in the IPC-11 benchmarks using *FD-lev* and *Gam-lev* variable orderings. Timeouts are assigned a value of 2000.

7.4.3 Symbolic and Explicit-State Search with M&S Heuristics

One important contribution of this chapter is that using the BDD representation of M&S heuristics with a linear merging strategy enables an efficient evaluation in symbolic search. In this section, we evaluate the performance when using M&S heuristics in symbolic search compared to blind symbolic search and explicit-search.

In order to guarantee a compact symbolic representation, M&S always uses a linear merging strategy with the same variable ordering as the symbolic search. We use three different variable ordering strategies: *Gam-lev* and *FD-rev* and *cgr*. For the shrinking strategies we use the two configurations used in IPC-2011: bisimulation with a limit of 200,000 abstract states and greedy bisimulation without any limit.

The results are shown in Table 7.3. When both use the same heuristic BDDA* beats A* in most domains, though there are a few counter-examples. A remarkable case is VISITALL with gop strategies, where the explicit-state A* consistently solves more problems than BDDA* due to the difference in tie-breaking criteria (the heuristic is perfect for the initial state but there is an exponential number of states with the same f-value, explaining the huge importance of tie-breaking in that case). The other few cases where A* outperforms BDDA* can be explained by the various overheads in symbolic search, such as transforming the M&S heuristic to the symbolic representation. Nevertheless, symbolic search configurations obtain better total coverage and obtain the best results in most domains.

While the M&S heuristic significantly improves the coverage of explicit-state search, in symbolic search it is less useful. In fact, with all the variable orderings, the best coverage of symbolic search is obtained without any heuristic, including the total best coverage with the GAMER ordering. The results are slightly biased because of the MICONIC domain, and the coverage scores show that the M&S heuristic can help in some cases in symbolic search as well. More importantly, in some domains such as DRIVERLOG, LOGISTICS98, and TRUCKS the combination of M&S heuristics and symbolic search solves more problems than symbolic blind search and A* with M&S. This shows

	Lazy BDDA*															A*													
	GAMER					*FD-rev*					*FD-cggoal-rnd*					∅	GAMER				*FD-rev*				*FD-cggoal-rnd*			
	∅	bop		gop		∅	bop		gop		∅	bop		gop			bop		gop		bop		gop		bop		gop	
		100k	200k	100k	∞		100k	200k	100k	∞		100k	200k	100k	∞		100k	200k	100k	∞	100k	200k	100k	∞	100k	200k	100k	∞
AIRPORT (50)	**24**	22	16	23	**24**	21	21	17	22	22	21	11	10	11	21	23	21	20	23	23	22	21	23	23	21	18	21	**24**
BARMAN (20)	**8**	4	5	4	**8**	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
BLOCKSWORLD (35)	21	25	21	23	26	20	25	22	**28**	**28**	21	25	23	**28**	**28**	18	19	19	**28**	26	22	20	**28**	**28**	21	21	21	24
DEPOT (22)	4	**7**	6	**7**	**7**	4	6	6	6	6	4	6	5	4	6	4	**7**	**7**	6	6	6	6	6	6	6	6	6	6
DRIVERLOG (20)	11	13	13	**14**	**14**	11	13	13	**14**	**14**	11	13	13	**14**	**14**	7	13	13	13	13	13	13	12	12	13	13	13	13
ELEVATORS08 (30)	19	20	**21**	20	5	19	19	19	19	1	19	19	19	19	1	13	15	16	17	6	14	14	14	1	18	18	18	4
ELEVATORS11 (20)	16	17	**18**	17	3	16	16	16	16	0	16	16	16	16	0	11	13	14	14	4	12	12	12	0	15	15	15	2
FLOORTILE11 (20)	**14**	**14**	**14**	**14**	8	**14**	**14**	**14**	**14**	4	**14**	10	10	8	4	8	9	10	8	6	8	8	8	4	8	8	8	4
FREECELL (80)	16	15	14	**20**	**20**	15	9	4	9	**20**	14	7	3	9	14	16	15	14	19	19	8	4	19	19	19	16	**20**	19
GRID (5)	1	**3**	**3**	1	2	1	**3**	**3**	1	2	1	**3**	**3**	1	2	1	2	**3**	2	2	**3**	**3**	2	2	**3**	**3**	2	2
GRIPPER (20)	**20**	**20**	**20**	**20**	18	**20**	**20**	**20**	**20**	**20**	15	15	15	12	15	8	8	8	8	8	**20**	**20**	8	8	8	8	8	8
LOGISTICS 00 (28)	16	21	21	20	19	21	**22**	**22**	**22**	21	16	21	19	21	20	10	19	19	16	16	20	20	16	16	20	20	17	16
LOGISTICS 98 (35)	4	5	**6**	**6**	5	3	5	5	5	5	2	5	5	5	5	2	5	5	4	3	5	5	4	4	5	5	5	4
MICONIC (150)	96	79	80	82	81	**108**	86	88	81	88	**108**	83	84	80	85	50	65	67	52	52	74	77	52	52	77	77	61	52
MPRIME (35)	22	20	16	20	26	23	15	12	12	**27**	21	15	12	12	**27**	19	18	16	23	23	15	12	23	23	16	16	19	23
MYSTERY (30)	15	12	11	13	16	15	9	8	11	16	15	9	8	11	16	15	12	11	**17**	**17**	10	8	**17**	**17**	10	10	12	**17**
NOMYSTERY11 (20)	11	**20**	18	**20**	14	12	**20**	**20**	18	16	11	19	**20**	18	14	8	16	16	14	14	18	18	14	14	**20**	**20**	15	14
OPENSTACKS08 (30)	**30**	**30**	**30**	**30**	9	24	25	23	25	9	27	29	24	28	9	21	21	21	21	9	21	21	21	9	21	21	21	9
OPENSTACKS11 (20)	**20**	**20**	**20**	**20**	4	19	19	18	19	4	19	**20**	17	**20**	4	16	16	16	16	4	16	16	16	4	16	16	16	4
OPENSTACKS06 (30)	**20**	16	10	13	13	9	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
PARCPRINTER08 (30)	**23**	16	16	16	20	**23**	17	16	20	20	**23**	15	15	18	20	20	19	19	20	19	20	20	20	19	19	19	20	19
PARCPRINTER11 (20)	**18**	12	12	12	15	**18**	12	11	15	15	**18**	11	11	14	15	15	14	14	15	14	15	15	15	14	14	14	15	14
PARKING11 (20)	0	**1**	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0	0	0
PATHWAYS-NONEG (30)	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	4	**5**	**5**	4	4	4	4	4	4	4	4	4	4	4	4	4
PEG-SOLITAIRE08 (30)	28	**29**	**29**	**29**	6	28	**29**	**29**	**29**	6	27	**29**	**29**	**29**	6	27	**29**	**29**	**29**	6	**29**	**29**	**29**	6	28	27	28	6
PEG-SOLITAIRE11 (20)	18	**19**	**19**	**19**	0	18	**19**	**19**	**19**	0	17	**19**	**19**	**19**	0	17	**19**	**19**	**19**	0	**19**	**19**	**19**	0	18	17	18	0
PIPESWORLD-NT (50)	15	15	11	14	15	13	13	9	14	14	12	8	4	11	12	14	**16**	12	15	15	14	9	15	15	13	7	11	15
PIPESWORLD-T (50)	**17**	15	13	14	16	16	14	8	15	15	10	10	8	13	13	11	13	12	**17**	16	16	8	**17**	**17**	10	8	9	**17**
PSR-SMALL (50)	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	49	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**
ROVERS (40)	12	11	11	12	**13**	11	10	9	10	10	10	10	10	11	11	5	8	8	6	6	8	8	6	6	8	8	7	6
SATELLITE (36)	7	**9**	8	8	8	6	7	7	7	7	6	7	8	7	7	5	6	6	6	6	6	7	6	6	7	6	6	6
SCANALYZER08 (30)	12	12	11	12	6	12	13	13	13	9	12	13	12	13	9	12	12	12	12	6	13	**14**	13	9	12	12	13	9
SCANALYZER11 (20)	9	9	8	9	3	9	10	10	10	6	9	10	9	10	6	9	9	9	9	3	10	**11**	10	6	9	9	10	6
SOKOBAN08 (30)	28	28	27	28	2	28	28	28	28	3	27	27	27	27	3	27	**29**	28	**29**	3	**29**	**29**	**29**	3	**29**	27	27	2
SOKOBAN11 (20)	**20**	**20**	19	**20**	1	**20**	**20**	**20**	**20**	1	**20**	**20**	**20**	**20**	1	**20**	**20**	**20**	**20**	1	**20**	**20**	**20**	1	**20**	**20**	**20**	1
TIDYBOT11 (20)	**14**	9	3	11	11	12	8	1	8	13	11	8	1	8	9	12	6	5	13	13	8	1	13	13	8	3	8	13
TPP (30)	9	8	8	8	8	**11**	8	8	10	9	10	8	8	10	8	6	6	6	6	6	7	7	6	6	7	7	8	6
TRANSPORT08 (30)	11	11	11	11	11	**12**	11	11	11	11	**12**	11	11	11	11	11	11	11	11	11	11	11	11	11	**12**	**12**	11	11
TRANSPORT11 (20)	6	6	6	6	6	**8**	7	7	7	6	7	7	7	6	6	6	6	6	6	6	7	7	6	6	7	7	6	6
TRUCKS (30)	10	10	10	**12**	11	10	10	10	10	10	10	10	10	10	10	7	8	9	8	8	8	8	8	8	8	8	8	8
VISITALL (20)	10	11	11	11	12	12	12	12	12	13	9	10	10	10	11	9	9	9	**16**	11	9	9	**16**	**16**	10	10	10	**16**
WOODWORKING08 (30)	22	22	20	**25**	23	13	18	17	22	22	14	15	15	17	16	9	16	16	14	14	12	13	14	14	14	14	16	14
WOODWORKING11 (20)	16	16	15	**18**	17	7	12	12	16	16	9	9	9	11	10	4	11	11	9	9	7	8	9	9	9	9	11	9
ZENOTRAVEL (20)	8	**12**	**12**	**12**	**12**	8	**12**	**12**	11	11	8	**12**	**12**	11	11	8	11	11	10	10	**12**	**12**	10	10	**12**	**12**	11	10
TOTAL COV (1396)	**756**	739	698	749	593	729	704	665	715	586	702	660	626	669	546	568	638	632	657	499	653	628	653	502	657	632	636	504
SCORE COV (36)	18.22	**18.83**	17.58	**18.64**	15.71	17.37	17.84	16.73	17.67	15.35	16.40	16.62	15.67	16.27	14.26	13.73	15.68	15.66	16.63	13.44	16.59	15.77	16.51	13.57	16.26	15.58	15.63	13.60

Table 7.3: Coverage of A* and Lazy BDDA* with M&S heuristics, compared against symbolic blind search (∅). Shrinking strategies are bisimulation (bop) and greedy bisimulation (gop), with limits of 100,000, 200,000 and ∞ abstract states.

that the combination of M&S heuristics and symbolic search may be useful in combination with other planners in portfolio approaches.

The results are highly influenced by the choice of the variable ordering. Overall the FAST DOWNWARD ordering is better for the M&S heuristic, while GAMER's ordering helps the symbolic exploration. Due to this, the integration of symbolic search and the M&S heuristic is difficult because both must use the same ordering in order to guarantee a compact representation of the heuristic.

7.5 Summary

Current state-of-the-art symbolic planners like GAMER use symbolic PDBs that ignore a subset of variables of the problem. However, in recent years more elaborate approaches have been developed for explicit-state search, like merge-and-shrink abstractions. Merge-and-shrink abstractions are a generalization of PDBs that have been shown to be very competitive with the state-of-the-art. For example, in the 2011 edition of the IPC, M&S was the runner-up and part of the portfolio winner of the competition. In this chapter, we have addressed the question of whether is it possible to use M&S in symbolic heuristic planners.

The M&S heuristic is precomputed and stored in a *cascading-tables* representation that provides efficient operations to build the abstraction and retrieve the precomputed heuristic values. However, in order to efficiently use a heuristic in symbolic search, it must be encoded in form of BDDs. In this chapter we provided an algorithm to construct the ADD representation of an M&S heuristic from the *cascading tables*. Then, the ADD representation can be converted to a vector of BDDs and plugged into an optimal symbolic heuristic search planner to exploit this expressive estimate.

We proved that the resulting ADD can be computed in polynomial time and has at most polynomial overhead with respect to the *cascading tables* under the assumption of a linear merging strategy and keeping the same variable ordering in the merging strategy and the ADD. Therefore, our approach exactly matches the quality of the explicit-search variants and is general to all M&S variants using a linear merge strategy. Moreover, we show that ADD reduction can yield smaller structures than those derived with state-of-the-art shrinking strategies, also providing advantages to its use in explicit A^* search. On the other hand, posterior research has studied the relation between M&S with non-linear merge strategies and the ADD representation (Helmert et al., 2015). Helmert et al. proved the conjecture stated by Helmert et al. (2014) that M&S with non-linear merge strategies is more expressive than ADDs, so that the transformation is not tractable with non-linear merge strategies.

Empirical results also yield interesting observations:

- The variable ordering for the M&S heuristic influences both the quality of the estimate and the symbolic exploration. The heuristic choice applied in FD pleases the M&S heuristic, while the optimization applied in GAMER pleases symbolic exploration. Future work is needed to combine the two for a competitive $BDDA^*$ exploration with the M&S heuristic.

- The small ADD sizes for the M&S heuristic suggest that there is sufficient memory for computing the maximum of more than one heuristic (in ADD representation). This results in a consistent, strictly more informed heuristic for the $(BDD)A^*$ exploration and provides a way of combining the accuracy of PDBs and M&S heuristics.

- Our comparison of symbolic regression and M&S with bisimulation for computing perfect heuristics shows that M&S is not a competitive approach for that matter, even though it can compute the perfect heuristic of some domains in time polynomial in the task size. We have shown that whenever M&S derives the perfect heuristic in polynomial time, the set of states involved in symbolic regression is of polynomial size. However, the opposite does not hold, i. e., the cascading tables as generated by M&S with bisimulation can be exponentially larger than the corresponding ADD. Even though the structure can be compressed after the heuristic is computed, memory can be exceeded before M&S terminates. Moreover, no known shrinking strategy guarantees a polynomial overhead with respect to the optimal compression achieved by ADD reduction rules. The dominance of regression search over M&S is also reflected in practice, since symbolic search yields the perfect heuristic in more instances than M&S.

- We have evaluated the empirical performance of M&S heuristics to guide explicit-state and symbolic BDDA* search. Surprisingly, our analysis shows that, though M&S heuristics can be useful for some domains, they do not always pay off in symbolic search.

This chapter is an extended version of a previously published work (Edelkamp et al., 2012), a collaboration with Stefan Edelkamp and Peter Kissmann.

Chapter 8

Symbolic Perimeter Abstraction Heuristics

Symbolic PDBs and M&S are two abstraction-based approaches to derive admissible heuristics for optimal planning. Symbolic PDBs use symbolic search to traverse PDB abstract state spaces. The use of symbolic search reduces the memory overhead and speeds up the search, enabling the use of more informed PDB abstractions. On the other hand, M&S abstractions are a generalization of PDBs and have been shown to overcome some of their limitations. In the previous chapter we proved that M&S abstractions can be represented symbolically with at most a linear overhead. In this chapter, we question whether symbolic search can be used to search the state space of M&S abstractions.

On the other hand, perimeter PDBs (PPDBs) are another improvement of PDBs to obtain better heuristics. Instead of starting the abstract state space search from the goals, they create a perimeter around the goals in the original state space and initialize the abstract search with the perimeter.

In this chapter, we present a new algorithm to derive admissible heuristics for cost optimal planning: Symbolic Perimeter Merge-and-Shrink (SPM&S). SPM&S is a perimeter abstraction heuristic that uses symbolic search to traverse the state space of several M&S abstractions. Empirical evaluation shows that SPM&S has the strengths of both symbolic search and M&S abstractions, deriving heuristics at least as good as the best of them for most domains.

8.1 Introduction

Symbolic PDBs use symbolic search to traverse the abstract state space and precompute the state space. Previous work on Symbolic PDBs has shown their ability to search larger state spaces than their explicit counterpart and, consequently, derive stronger heuristics. On the other hand, M&S abstractions generalize PDB abstractions, opening new ways to relax the problem and overcoming some limitations of PDBs. In Chapter 7 we analyzed the symbolic representation of M&S heuristics, concluding that M&S heuristics are efficiently representable with decision diagrams. Thus, in this chapter, we discern whether symbolic search can be used to obtain better M&S heuristics as Symbolic PDBs do with respect to PDBs.

Perimeter PDBs are another orthogonal improvement over PDBs. Perimeter PDBs were initially proposed in the context of heuristic search for combinatorial puzzles (Felner and Ofek, 2007) and later adapted to automated planning in a work parallel with ours (Eyerich and Helmert, 2013).

As a summary, in this chapter we make several separate contributions:

1. We define a hierarchy of M&S abstractions, which we call Symbolic M&S (SM&S). We take advantage of our results in Chapter 7 to consider how to apply symbolic search over SM&S state spaces. We will show that, under certain circumstances, our BDD representation of SM&S is guaranteed to be tractable.

2. We present a definition of perimeter abstraction heuristics. Our definition generalizes previous definitions of Perimeter PDBs. While previous work was restricted to unit-cost domains and only performs two phases, our definition can be used in domains with non-uniform costs and considers an arbitrary number of abstract state spaces. Moreover, we analyze how the closed lists of abstract searches should be populated in order to obtain more informed heuristics.

3. Our main contribution is a new algorithm to derive admissible heuristics for cost-optimal planning: Symbolic Perimeter M&S (SPM&S).[1] SPM&S combines our contributions by using our definition of perimeter abstraction heuristics on the SM&S hierarchy. Empirical evaluation shows that SPM&S successfully combines symbolic search and M&S abstractions to derive more accurate heuristics.

This chapter is structured as follows. First, in Section 8.2, we define the hierarchy of SM&S abstract state spaces and analyze its properties. Then, we define in detail perimeter abstraction heuristics: we summarize related work on perimeter PDBs in Section 8.3 and present our own definition of perimeter abstraction heuristics in Section 8.4. Then, Section 8.5 describes how to perform the mapping between states in different state spaces. The overall SPM&S algorithm that puts together the different parts described in previous sections is presented in Section 8.6. The new SPM&S heuristic is evaluated in Section 8.7, where we compare its performance against other state-of-the-art abstraction heuristics such as M&S and symbolic PDBs. As usual, the chapter concludes in Section 8.8 with a summary of the main contributions and conclusions.

8.2 Symbolic Merge-and-Shrink

In this section we propose Symbolic M&S, an improvement of M&S based on Symbolic PDBs (Edelkamp, 2002). Just as Symbolic PDBs, Symbolic M&S uses symbolic search to traverse the abstract state spaces. The difference is that, in our case, the abstract state space is defined as M&S abstractions. Using symbolic search, we aim to derive more accurate heuristics than M&S by searching larger abstract state spaces that do not need to be explicitly represented.

However, these larger state spaces cannot be directly obtained with the M&S framework, because the size of M&S abstract state spaces must remain small enough to explicitly represent them. Instead, our SM&S state spaces are derived from M&S abstractions without requiring their explicit representation. SM&S abstractions result of merging several intermediate M&S abstractions. In particular, we focus on linear merging strategies, where at any point during the M&S procedure there is a single non-atomic M&S abstraction plus the atomic-abstractions of the remaining variables.

Definition 8.1 (SM&S abstraction)**.** *Let α be an abstraction with relevant variables $\mathcal{V}_\alpha \subseteq \mathcal{V}$. Its associated SM&S abstraction $\alpha^{SM\&S}$ is defined as the synchronized product of α and the projection of the planning task over its non-relevant variables, $\mathcal{V} \setminus \mathcal{V}_\alpha$: $\alpha^{SM\&S} = \alpha \otimes \Pi_{\mathcal{V}\setminus\mathcal{V}_\alpha}$.*

Intuitively, M&S abstractions partially consider a subset of variables (the *relevant* variables) and completely ignore the rest. The SM&S abstraction derived from an M&S abstraction considers

[1]We originally called our technique Symbolic M&S (Torralba et al., 2013b). We rename it to Symbolic Perimeter M&S to highlight that it is a perimeter abstraction heuristic.

the same information about the relevant variables, but instead of ignoring all the other variables, they are fully considered (no abstraction is made over those variables). For example, consider a typical logistics task with several packages and trucks. an M&S abstraction, α, that considers only two packages may reduce the abstraction size by considering some combinations equivalent (for example, $p_1 = G \wedge p_2 = T \equiv p_1 = T \wedge p_2 = G$ or $p_1 = A \equiv p_1 = B$). However, α completely ignores all the information regarding the location of the trucks or other packages. $\alpha^{\text{SM\&S}}$, on the other hand, fully considers the location of all the trucks. $\alpha^{\text{SM\&S}}$ is still an abstraction of the original problem since it also applies the same equivalences than α for packages p_1 and p_2.

Definition 8.1 can be applied for any intermediate abstraction in the M&S algorithm. Next, we consider the abstraction hierarchy that results from computing the SM&S abstraction of every intermediate M&S abstraction.

Definition 8.2 (SM&S abstraction hierarchy)**.** *Let $\alpha_0, \ldots, \alpha_{n-1}$ be all the intermediate abstractions generated by the M&S procedure, where α_0 is the empty abstraction without any variable and α_{n-1} is the final result of M&S. We define the corresponding SM&S hierarchy as the list of SM&S abstractions: $\alpha_0^{SM\&S}, \ldots, \alpha_{n-1}^{SM\&S}$.*

Figure 8.1 shows an example of how the SM&S hierarchy is obtained from an M&S procedure. The left part shows the intermediate abstractions generated during the M&S algorithm. The M&S algorithm is initialized with the atomic abstraction of each variable, π_i. In our example, there are five variables and, consequently, there are five atomic abstractions, $\pi_1, \ldots, \pi_5$. The M&S algorithm iteratively merges two abstractions by computing their synchronized product and, if necessary, applies shrinking. In our example, a linear merging strategy is used, so that variables are iteratively merged into the main abstraction. It starts merging variables π_1 and π_2 into α_1 and iteratively includes more variables into each intermediate abstraction α_i. The induced SM&S abstractions always consider all the variables, and are the result of merging all variables into each α_i without applying any additional shrinking.

The SM&S hierarchy defines n state spaces, producing a trade-off between the size of the abstract state space and heuristic accuracy. $\alpha_0^{\text{SM\&S}}$ corresponds to the original non-abstracted state space, which has exponential size and is intractable to represent but produces the perfect heuristic. On the other hand, $\alpha_{n-1}^{\text{SM\&S}}$ corresponds to the final M&S abstract state space, $\alpha_{n-1}^{\text{SM\&S}} \equiv \alpha_{n-1}$, whose state space can be explicitly represented (assuming a suitable value of the maximum number of M&S abstract states) but with a possible great loss of information. This is reflected in the example of Figure 8.1 since $\alpha_0^{\text{SM\&S}}$ is the merge of all the variables and $\alpha_4^{\text{SM\&S}}$ is α_4. All the rest of SM&S abstractions in the hierarchy, $\alpha_i^{\text{SM\&S}}, 0 < i < n-1$, enable the desired trade-off between $\alpha_0^{\text{SM\&S}}$ and $\alpha_{n-1}^{\text{SM\&S}}$. Each $\alpha_i^{\text{SM\&S}}$ is strictly more relaxed than the previous $\alpha_{i-1}^{\text{SM\&S}}$, so that the size of the abstract state spaces decreases at expense of producing less informed heuristics $h^{\alpha_{i-1}^{\text{SM\&S}}}(s) \geq h^{\alpha_i^{\text{SM\&S}}}(s)$.

Even though similar hierarchies could be defined for PDB abstractions, all the advantages of M&S abstractions over PDBs carry over to the SM&S hierarchies. M&S abstractions are a generalization of PDBs, so every PDB hierarchy can be derived with SM&S as well. The main advantage of SM&S hierarchies is that every abstraction in the hierarchy takes into account all the variables in the problem, overcoming the theoretical limitations of PDBs.

Note that the size of SM&S abstract state spaces may be of exponential size, so that they cannot be explicitly represented. This is not a problem, since we will directly traverse those state spaces using symbolic search. In order to perform symbolic search over the abstract state spaces, we need to represent sets of abstract states and to be able to perform the successor generation. Below, we describe the encoding that we use and analyze its theoretical properties.

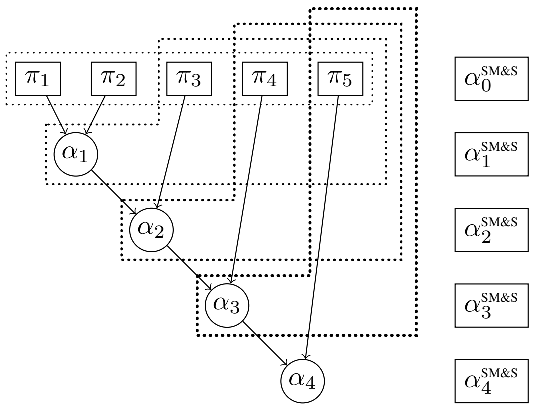
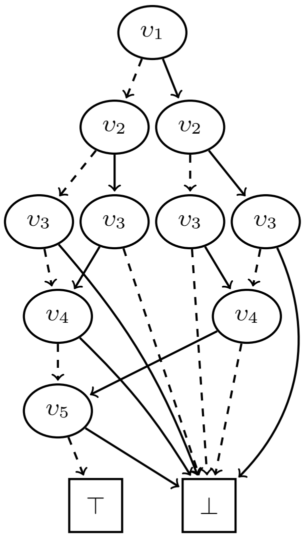

π_1
π_2
π_3
π_4
π_5
α_1
α_2
α_3
α_4
$\alpha_0^{\text{SM\&S}}$
$\alpha_1^{\text{SM\&S}}$
$\alpha_2^{\text{SM\&S}}$
$\alpha_3^{\text{SM\&S}}$
$\alpha_4^{\text{SM\&S}}$

Figure 8.1: Hierarchy of symbolic M&S state spaces. Atomic abstractions (π_i) are merged in a linear ordering to derive different M&S heuristics (α_i). Dotted lines show the combination of several abstractions that result in each $\alpha_i^{\text{SM\&S}}$.

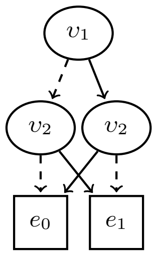
8.2.1 SM&S Abstract State Representation

In this section we study the symbolic representation of SM&S abstractions, i. e., how to represent sets of abstract states by means of BDDs. As explained in Chapter 2, sets of states in the original state space are represented as functions $f : \mathcal{S} \to \{\top, \bot\}$, where states are described in terms of binary variables $x = x_1, \ldots, x_n$. In the same way, sets of abstract states in $\alpha_i^{\text{SM\&S}} = \alpha_i \otimes \Pi_{v_{i+1},\ldots,v_n}$ are represented as characteristic functions $f : \mathcal{S}^{\alpha_i^{\text{SM\&S}}} \to \{\top, \bot\}$. An important decision is what variables (and variable ordering) should be used to represent sets of abstract states in $\mathcal{S}^{\alpha_i^{\text{SM\&S}}}$. We decided to use the same set of variables x that is used to represent the original state space. An alternative could be to design a new set of auxiliary variables y, optimized to represent abstract states, replacing variables $v_1, \ldots, v_i$ for another set of variables $y_1, \ldots, y_k$ specifically designed to represent the abstraction. While this alternative could help to perform the abstract search more efficiently, encoding abstract states with the same variables lets us use the same representation for searches in any state space. Thus, using the same set of variables for all the abstract states makes the conversion between original states and abstract states easier. Moreover, the same BDD can be interpreted as a set of abstract states or a set of complete states. This means that sets of abstract states can be interpreted as a set of total states, i. e., the set of all states mapped to some abstract state in the set. This has some practical implications. For example, the BDDs describing the abstract search to precompute the heuristic are directly the heuristic BDDs used to guide the symbolic A* search.

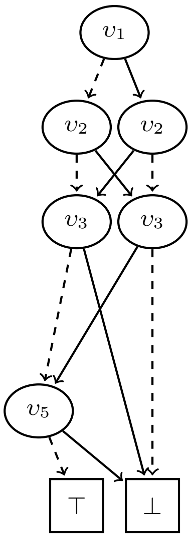
Figure 8.2 shows an example of the symbolic M&S relaxation of a search and how a BDD can be interpreted both as a set of states and a set of abstract states. Given a set of states in the original state space, depicted in Figure 8.2a, and an intermediate M&S abstraction α_2, depicted in Figure 8.2b, we derive the corresponding set of states $\alpha_2^{\text{SM\&S}}(S)$. S is a set of four states, namely 00000, 01100, 10110 and 11010. α_2 is an M&S abstraction over variables v_1 and v_2 with two abstract states e_0 and e_1. In a general case, our M&S abstractions will have a larger (but bounded) number of abstract states. The ADD depicted in Figure 8.2b represents the mapping from partial states $\langle v_1, v_2 \rangle$ to one of the abstract states, e_0 or e_1. Each abstract state corresponds to an equivalence relation over variables v_1 and v_2. e_0 makes starting with 00 equivalent to starting with 11, so that if a set of abstract states contains state $00abc$, it automatically contains $11abc$ as well. Whenever any state starting with 00 is reached in the abstract search, the corresponding state starting with 11 will be automatically reached



(a) Original BDD S



(b) Abstraction α_2 ADD



(c) $\alpha_2^{\text{SM\&S}}(S)$ BDD

Figure 8.2: Symbolic M&S relaxation that takes an input BDD, S, and an ADD that represents a M&S heuristic and computes the BDD that represents the abstract states in S. The top part of $\alpha_2^{\text{SM\&S}}(S)$ BDD coincides with α_2 ADD.

and vice versa. e_1 makes 10 and 01 equivalent in a similar way.

Figure 8.2c shows the BDD that represents the set of abstract states $\alpha_2^{\text{SM\&S}}(S)$, containing four abstract states or eight complete states, depending on our interpretation. An important point is that BDD nodes pointed to by the paths 00 and 11 are equivalent. A similar reasoning can be made for every abstract state e_i making the top part of any BDD describing abstract states at most as large as the ADD representation of the M&S abstraction.

The example shown in Figure 8.2 is an ideal case, where not only the layers corresponding to α_2 are reduced, but also other layers get simplified. Opposite examples can be defined where the number of BDD nodes required to represent $\alpha_i^{\text{SM\&S}}(S)$ is even exponentially larger than the number of nodes to describe S. However, the correspondence between the ADD that describes the M&S abstraction α_2 and the upper levels of any BDD describing a set of abstract states lets us prove some bounds for the size of BDDs describing any set of abstract states.

Proposition 8.1. *Let α_k be an M&S abstraction over relevant variables $v_1, \dots, v_k$ with M abstract states. Let $S^{\alpha_k^{SM\&S}}$ be a set of abstract states in $\Theta^{\alpha_k^{SM\&S}}$. Then, the layer $k{+}1$ of the BDD representation of $S^{\alpha_k^{SM\&S}}$, under variable ordering starting by $v_1, v_2, \dots, v_k$, has at most M BDD nodes.*

Proof. Each node in layer $k+1$ corresponds to one or more abstract states in α_k. Suppose that there is a node, n' that does not correspond to any abstract state. Let $d_1, \dots d_k$ be an assignment to variables $v_1, \dots, v_k$ such that leads from the root of the BDD to n'. By definition, all assignments are related to a unique abstract state, so let $s^{\alpha_k^{\text{SM\&S}}}$ be the abstract state corresponding to partial state $d_1, \dots d_k$.

By definition of the abstract state space, $d_1, \dots d_k$ is indistinguishable of all the other assignments mapped to $s^{\alpha_k^{\text{SM\&S}}}$. Therefore, node n' represents not only $d_1, \dots d_k$ but also all the other assignments of $s^{\alpha_k^{\text{SM\&S}}}$ and, hence, $s^{\alpha_k^{\text{SM\&S}}}$ itself is associated with n'. □

Proposition 8.1 bounds the size of a single layer of the BDD describing a set of abstract states in $\Theta^{\alpha_k^{\text{SM\&S}}}$. However, this result can easily be extended to all the upper layers in the BDD. As we are assuming a linear merging strategy, all the M&S abstractions are built as the merge of an M&S abstraction and the atomic abstraction of another variable. Therefore, the bound of Proposition 8.1 is valid for all the upper levels of the BDD.

Theorem 8.2. *Let $M_1, \dots, M_k$ be the number of abstract states of k intermediate abstractions $\alpha_1, \dots, \alpha_k$ of an M&S algorithm with a linear merging strategy, such that $\alpha_i = \gamma_i(\alpha_{i-1} \otimes \pi_i)$. Let $S^{\alpha_k^{SM\&S}}$ be a set of abstract states in $\Theta^{\alpha_k^{SM\&S}}$. Then, the BDD representation of $S^{\alpha_k^{SM\&S}}$, under variable ordering starting by $\upsilon_1, \upsilon_2, \dots, \upsilon_{k+1}$, is represented with at most $(\sum_{i \in [1,\dots,k]} M_i) + M_k 2^{n-k+1}$ BDD nodes.*

Proof. The BDD that represents $S^{\alpha_k^{\text{SM\&S}}}$ may be divided in its top and bottom parts. The top part includes layers 1 to k and the bottom part the remaining $(n-k)$ layers. From Proposition 8.1, we can bound the size of each top layer to M_i nodes, so that the top part of the BDD uses, at most, $\sum_{i \in [1,\dots,k]} M_i$ nodes. The bottom layers correspond to functions over the remaining $(n-k)$ variables. Each of these functions is described as a BDD with $n-k$ levels. In the worst case, they do not share any node, and thus each one has 2^{n-k+1} nodes. Since there are M_k different functions, the size of the bottom part of S_B is bounded by $M_k 2^{n-k+1}$. □

8.2.2 SM&S Transition Representation

Once we have defined the symbolic representation of sets of abstract states, in order to perform a symbolic search, we need a reliable way to perform successor generation. In Chapter 3 we studied in detail how to perform successor generation on the original state space on PDB abstract state spaces. In summary, planning operators are represented by means of one or more transition relations (TRs). Then, the *image* and *pre-image* operations use the TRs to compute the set of successor or predecessor states. One of the conclusions we came to is that how TRs are represented may have a huge impact on the search performance.

As a brief recap, each TR is a function $f : S, S' \rightarrow \{\top, \bot\}$ that relates predecessor to successor states, i. e., $f(s, s')$ is true if and only if there is a transition from s to s'. Predecessor states are represented with the standard set of variables x, and an auxiliary set of variables x' represents the successor states. The variable ordering is also very important. Usually, variables in x and x' are interleaved in the following manner: $x_1, x'_1, x_2, x'_2, \dots x_n, x'_n$. This variable ordering exploits the fact that operators usually affect a few variables so that most variables preserve their previous value.

In SM&S abstract state spaces, however, the transitions are different than in the original state space. Moreover, in a given $\alpha_i^{\text{SM\&S}}$, variables $\upsilon_1, \dots, \upsilon_i$ are highly related and the TR representation can be very complex. Therefore, contrary to our state representation, it is not possible to preserve the same variable ordering as for the TRs of the original state space.

We avoid the problem of representing SM&S TRs by using the TRs of the original state space. This is possible since the set of abstract states may be interpreted as a set of non-abstracted states. Of course, the result of the image operation is a set of states of the original state space. Fortunately, we can apply the abstraction function to obtain the set of abstract states associated with them. The *image* and *pre-image* operations are applied as $image(S^{\alpha_i^{\text{SM\&S}}}, TR^{\alpha_i^{\text{SM\&S}}}) = \alpha_i^{\text{SM\&S}}(image(S^{\alpha_i^{\text{SM\&S}}}, T))$. The details of this operation are described in Section 8.5.

Thus, we do not need to compute a new set of TRs for every abstract state space we traverse. However, using the original TRs also has relevant drawbacks. In particular, the intermediate BDDs may induce a large overhead. Even in the case where the sets of abstract states are guaranteed to be efficiently representable, the intermediate set of original states does not have any guarantee with respect to its size.

8.3 Perimeter Pattern Databases

Perimeter search is a form of bidirectional search independently devised by Manzini (1995) and Dillenburg and Nelson (1994) that operates in two phases: the backward phase and the forward phase, as represented in Figure 8.3. The backward phase generates a perimeter of radius r around the goal with a uniform-cost search,[2] so that the perfect heuristic value from each state in the perimeter to the goal is known and for any other state outside the perimeter its distance to the goal is strictly greater than r. Then, the forward phase performs a forward search from the initial state to the perimeter using any algorithm like A* or IDA*. Since the goal of the forward search is any state in the perimeter P, the heuristic evaluation estimates the distance to the closest state in the perimeter so that $h(s) = \min_{s' \in P} h(s, s')$.

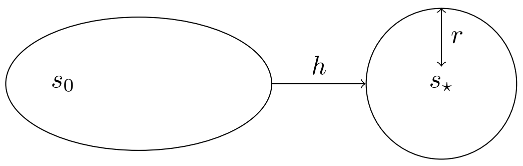


Figure 8.3: High-level diagram of Perimeter PDBs. First, the perimeter around the goal of radius r contains every state that can reach the goal with cost r or less. Then, the forward search is performed guided with heuristics that estimate the cost to reach the closest state in the perimeter frontier.

The major drawback of perimeter search is that heuristic evaluation may become too expensive when the perimeter is large. In practice, some optimizations avoid to evaluate $h(s, s')$ for every state in the perimeter (Manzini, 1995), but it is often not enough to deal with large perimeters. Using a model to predict the optimal radius r beforehand may be useful to avoid large overheads with respect to unidirectional search (Linares López and Junghanns, 2002). There are other ways to use the perimeter while avoiding a large number of heuristics evaluations, such as taking the estimation to the goal and correct it by considering the error made in the perimeter nodes (Kaindl and Kainz, 1997).

Abstraction heuristics are a candidate to overcome this problem, because they precompute the heuristic value of every abstract state. Perimeter Pattern Databases (PPDBs) were first studied by Felner and Ofek in (2007). PPDBs store for each abstract state the minimum distance towards any abstract state in the perimeter. Then, a single PDB lookup suffices to perform the heuristic evaluation independently of the number of states in the perimeter.

A perimeter PDB is constructed in two phases that perform a backward search:

1. As in perimeter search, use a backward search to build a perimeter P of radius r around the goal states in the original search space.

2. Perform a second backward search in the abstract state space seeded with abstract states corresponding to states in the perimeter $\{\alpha(p) : p \in P\}$ with initial cost r. As noted by Felner

[2]Other works in the literature consider breadth-first search because they deal with unit-cost domains. In our case we always consider domains with diverse action costs.

and Ofek, seeding the abstract search with cost r is equivalent to add r to each heuristic estimation. This means that the heuristic value of a state is an admissible estimation to the closest state in the perimeter plus the perimeter radius.[3]

The PPDB is used as a heuristic for a forward search. The heuristic is only admissible for nodes outside the perimeter. This is not inconvenient, though, because the search halts whenever a state in the perimeter is chosen for expansion. The potential of PPDBs is easily recognizable. They easily evaluate states during the search, which is the main drawback of perimeter search. Moreover, compared to standard PDBs, PPDBs produce heuristics at least as informed as PDBs. However, after performing a theoretical and empirical analysis, Felner and Ofek report that PPDBs are not better than just taking the maximum between the PDB and the perimeter heuristic (correcting values in the PDB under $r + 1$ to take $r + 1$).

Linares López (2008) proposed multi-valued PPDBs that store different entries in the PDB for the distance to each state in the perimeter. When a single PPDB is used, the heuristic value of a state to the minimum of all such entries $\min_{p \in P} h(\alpha(s), \alpha(p))$, so there is no advantage with respect to PPDBs. Nevertheless, when maximizing over multiple PPDBs it is possible to reason about which perimeter state is producing the estimates to get more informed estimations.

More recently, in a parallel work with ours (Torralba et al., 2013b), Eyerich and Helmert applied perimeter PDBs in the context of automated planning (Eyerich and Helmert, 2013). They show that perimeter search can enhance the performance of standard PDBs contradicting the conclusions reached by Felner and Ofek and suggest that perimeter PDBs should be revisited. Indeed, the analysis of Felner and Ofek only considered permutation domains with no spurious paths in the abstract state space, which is not a common case in planning domains. Eyerich and Helmert used a breadth-first implementation so that it only works in unit-cost domains, though, it could be extended to handle non-uniform action costs. They identify three challenges to extrapolate perimeter databases from heuristic search combinatorial domains to planning (C1 to C3), plus one additional challenge related to the collision of frontiers (C4):

C1 Perimeter search had been previously applied to domains with one goal state and invertible operators. In that case, the backward search to generate the perimeter is no harder than the forward search. In planning, one has to deal with the complexity of regression, i. e., an exponential number of goal states and operators that generate more than one state in backward search.

C2 In the heuristic search literature, PDBs are usually constructed offline. Their construction time is irrelevant since the same PDB is used to solve many different problems. In planning, however, the time spent in generating the abstraction heuristic is part of the total time. This applies to all kind of abstraction heuristics, but in the case of perimeter abstractions may be aggravated by the time spent in matching states in the perimeter to abstract states.

C3 The perimeter radius must be automatically set by the planner, without instance-specific parameter tuning.

C4 In order to terminate the forward search as soon as a state in the perimeter is expanded, membership in the perimeter must be efficiently performed. This can be challenging, especially for large perimeters.

[3]This initialization assumes unit-cost domains. For domains with multiple action costs, each abstract state is initialized with the h^*-value of the corresponding state in the perimeter. The general initialization of abstract searches is described in Definition 8.5 on page 136.

They successfully overcome these challenges. To address challenge 1, they propose the use of regression search using partial states as studied in Section1.2.3. This involves a number of optimizations, like using a *match tree* to detect (most) duplicate states. In order to ease Challenge 2, they propose an optimization to efficiently seed the abstract frontier with the abstract states associated to the perimeter. They iterate over states in the perimeter and get the corresponding abstract state by means of a hash function, avoiding the iteration over all abstract states. Challenge 3 requires to ensure that the perimeter is constructed without manually deciding the radius. To ensure the termination of their procedure, they set different parameters that limit the time and memory of the algorithm, like maximal radius, runtime limit and memory bound. Finally, Challenge 4 is addressed by checking membership in the perimeter only for those states with heuristic value equal to the perimeter radius, avoiding the computation overhead for most states in the search.

In this chapter, we address PPDB challenges in a different way, through the use of symbolic search in Symbolic Perimeter PDBs (SP-PDBs) and Symbolic Perimeter M&S (SPM&S). As pointed out by Kissmann, the extrapolation of perimeter pattern databases to the symbolic case is straightforward (Kissmann, 2012). Indeed, the use of symbolic search is a possible answer to the challenges identified by Eyerich and Helmert:

C1 Symbolic search is an effective method to perform regression. Applicability of actions in regression is performed with the *pre-image* operation, which is symmetric to the *image* operation used in forward search. Subsumption of partial states is not a problem for duplicate detection in symbolic search, because all partial states are represented as a single state set in symbolic representation. Moreover, the empirical evaluation in Section 5.1 on page 81 shows a clear advantage over partial-state based regression.

C2 In symbolic search, the states in the perimeter frontier are mapped into abstract states (to seed the open list of abstract searches) through symbolic operations. The particular operations for PDBs and SM&S abstractions are presented on Section 8.6.

C3 As Eyerich and Helmert (2013), we also rely on parameters that bound the time and memory resources invested in constructing the PPDBs. As in the symbolic bidirectional blind search, we use the number of BDD nodes in the search frontier to limit the memory and estimate the time needed for the next step.

C4 In symbolic search, if the perimeter is encoded as a BDD, a conjunction suffices to detect the collision with the perimeter. A single conjunction has quadratic complexity in the size of the BDDs — as opposed to quadratic in the number of states. In practice, this is efficient when the size of the perimeter BDD is succinct with respect to the list of partial-states it represents. Even though there are no theoretical guarantees, the good empirical results of regression search shown in Section 5.1 on page 81 suggest that this is often the case.

8.4 Perimeter Abstraction Heuristics Revisited

In this section, we present several contributions to current state-of-the-art PPDBs previously analyzed. We redefine PPDBs to generalize the definition previously given in several ways:

- We take into account action-costs, so that our definition is valid and admissible for domains with non-uniform costs.
- Previous work on perimeter abstraction heuristics have considered only two different state-spaces: first a search is conducted in the original state space to define the perimeter. In a

second step, the search is completed in an abstract state space. We generalize the two-phases of PPDBs to an arbitrary number of searches in different state spaces. Our definition is based on abstraction hierarchies and can handle more than one abstract state space.

Moreover, while previous definitions always initialize the relaxed search with an empty closed list, we consider the inclusion of some abstract states in the closed list to obtain more accurate heuristics (see Definition 8.5 on page 136).

Perimeter abstractions (PA) are defined over an abstraction hierarchy $\alpha_0, \dots, \alpha_k$ so that they perform $k+1$ phases, one for each abstract state space Θ^{α_i}. Previous definitions of PPDBs were limited to two phases, one in the original state space and another in an abstract state space. Instead, PAs perform a phase for each α_i in the hierarchy. Note that, even though our definition specifies that all the phases are performed over abstract state spaces, it can also search the original state space. To conduct a perimeter search in the original state space it suffices to define α_0 as the identity function, $s = \alpha_0(s)$, so that $\Theta^{\alpha_0} \equiv \Theta$.

Figure 8.4 illustrates the idea of perimeter abstraction heuristics. After constructing a perimeter around the goal of radius r, the search frontier is relaxed into an abstract state space. Abstract searches are used to explore the state space outside the perimeter, though the information is not perfect anymore, as reflected in Figure 8.4 by the irregular form of the perimeters. Besides, not only one perimeter PDB is built, whenever the abstract search becomes intractable, it is relaxed into another abstract state space. In the end, all the abstract perimeters are used as heuristics, to estimate the distance from states in the forward search to the perimeter.

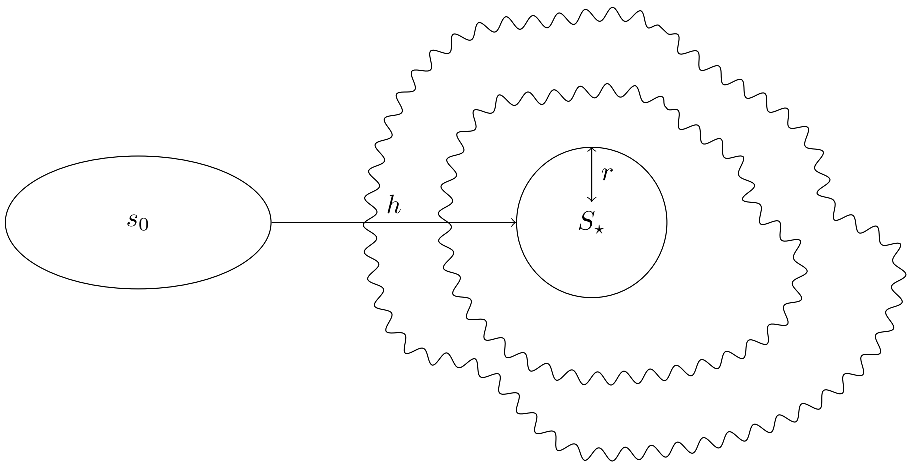


Figure 8.4: Example of a perimeter abstraction heuristic. Several abstract searches, each one more relaxed, provide the distance estimations to states in the goal perimeter.

Algorithm 8.1 shows the precomputation phases of a PA heuristic. The first phase performs a search in Θ^{α_0} (that usually corresponds to the original problem) initialized with the problem goals. Then, each phase i consists of a search in the abstract state space Θ^{α_i} initialized with the perimeter frontier of search $i-1$.

In order to obtain the heuristic values in a given state space with non-uniform action costs, we perform a backward uniform-cost search, DS.[4] We characterize the current status of a uniform-cost

[4] In our experiments we use symbolic uniform-cost search but all the results of this section are valid for the explicit-state search case as well.

Algorithm 8.1: Perimeter Abstraction Heuristic

```
Input: Planning problem $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$.
Input: Abstraction Hierarchy $\alpha_0, \dots, \alpha_k$.
Output: List of heuristics $h_{PA} = h_{DS0}, \dots, h_{DSk}$.
1  $open_0 \leftarrow \alpha_0(s_\star)$
2  $closed \leftarrow \perp$
3  $r \leftarrow 0$
4  foreach $\alpha_i \in \{\alpha_0, \dots, \alpha_k\}$ do          /* for each precomputation phase */
5      Uniform-Cost-Search $(open, closed, r, \alpha_i)$      /* perform uniform-cost search */
6      $h_{DSi} \leftarrow$ get-h$(closed, open, r)$                /* gather heuristic */
7      if $open = \emptyset$ then                  /* if the search has sucessfully finished */
8          return $h_{PA}$
9      if $i < k$ then                         /* if there are more abstractions left */
10         $open, closed \leftarrow$ re-seed$(open, closed, \alpha_{i+1})$
11 return $h_{PA}$
```

search as a tuple $DS = \langle open, closed, r \rangle$. $open$ represents the states generated but not expanded. $closed$ represents the states that have already been expanded. Both $open$ and $closed$ can be split into buckets $open_g$ and $closed_g$ that represent the states generated with cost g and expanded with cost g or less, respectively. r is the current radius of the perimeter, i. e., the current g-value of DS. In this chapter, we do not describe in detail how uniform-cost search is performed (details about uniform-cost algorithm and our implementation of symbolic uniform-cost search can be found in Section 2.4). Here, we only care about the status in which we initialize each search and after the search has been stopped. A search is stopped either when the state space has been completely traversed or the search was truncated at some radius r.

The algorithm initializes the first search in Θ^{α_0} with the goal of the problem at cost 0. The loop in line 4 iterates over all the precomputing phases, each associated with a different abstraction α_i. Each phase performs a uniform-cost search and gathers the related heuristic (lines 5 and 6). If there are more state spaces to explore, we re-seed $open$ and $closed$ with abstract states relative to the next abstraction (line 10). The algorithm terminates returning the heuristic when all the phases have been completed or if at some point there are no states left in $open$.

Before describing in detail how to re-seed the abstract searches, we define the heuristic that results from the PA algorithm. Each uniform-cost search performed in the algorithm produces an estimate h_{DS_i}. If the search DS_i on Θ^{α_i} was initialized with the tuple $\langle open, closed, r \rangle$, the corresponding heuristic, $h_{DS_i}(s)$, estimates the cost of reaching the goal from a state s passing by the perimeter $open$. r is an admissible estimation of the cost of reaching the goal from any node in the perimeter. Therefore, the estimation corresponds to $h_{DS_i}(s) = r + \min_{s_p \in open} h^\alpha(s, s_p)$. Since the searches are not always finished, the heuristic estimates follow the definition of partial abstraction heuristics (Anderson et al., 2007) detailed in Definition 8.3.

Definition 8.3 (Partial abstraction heuristic)**.** *Given a uniform-cost search $DS = \langle open, closed, r \rangle$ over the state space Θ^α. Let $g' > r$ be the cost of the next bucket in open, $g' = \arg\min_i open_i \neq \emptyset \wedge i > r$. We define the* next frontier cost *r' as the minimum between the cost of the next bucket in the open list and the current frontier plus the cost of the cheapest operator. $r' = \min(g', r +$

$\min_{o \in \mathcal{O}} c(o))$.[5]

We define the partial abstraction heuristic *of DS, h_{DS} as shown in Equation 8.1. States expanded by DS take the value with which they were expanded. Unexpanded states take either r' or ∞, depending on whether the exploration successfully finished or not:*

$$h_{DS}(s) = \begin{cases} c & \alpha(s) \in closed_c \\ r' & \alpha(s) \notin closed \text{ and } open \neq \emptyset \\ \infty & \alpha(s) \notin closed \text{ and } open = \emptyset \end{cases} \tag{8.1}$$

During the search, the perimeter abstraction heuristic, h_{PA}, combines the estimations of each search, h_{DS_i}. One must be careful to preserve admissibility when combining the heuristic estimates coming from searches initialized with different perimeters. The usual method to combine estimations of the same problem is just taking the maximum. However, in this case the maximum of all the estimates is not an admissible heuristic, because each estimate h_{DS_i} may be inadmissible in the inner part of the perimeter that was used to initialize DS_i. The previous definitions of PPDBs reviewed in Section 8.3 avoided the problem by considering only one abstract state space and stopping the forward search when colliding with the perimeter. The heuristic is admissible because states in the inner part of the perimeter are never evaluated.

Since we have several heuristics whose searches are initialized with different perimeters, the evaluation of states in the inner part of some perimeter cannot be avoided. In order to combine the individual estimations in an admissible way, we take the maximum of the subset of estimations that are ensured to be admissible. Following Definition 8.4 we rule out all the estimations coming from searches initialized with a perimeter in which the evaluated state had been expanded.

Definition 8.4 (Perimeter abstraction heuristic)**.** *Given the perimeter abstraction phases $DS_0, \ldots, DS_n$ performed over an abstraction hierarchy α_0 to α_n. Let s be a state, and DS_k be the first exploration in which $\alpha_k(s)$ was expanded, i. e., $k = \min\{i \mid \alpha_i(s) \in closed_{DS_i}\}$. Then, we define the perimeter abstraction heuristic as:*

$$h_{PA}(s) = \max_{i=[0,\ldots,k]} h_{DS_i}(s)$$

Before proving that h_{PA} is an admissible heuristic, we define how abstract searches are initialized with the $open$ and $closed$ lists of the previous search. Each search i continues from the perimeter generated by search $i-1$. Definition 8.5 describes how each search is initialized.

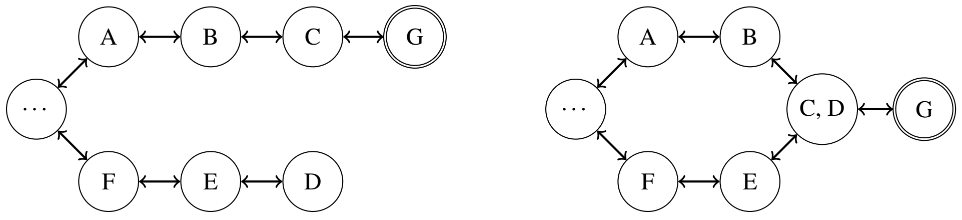
Definition 8.5 (Perimeter search initialization)**.** *Let a search $DS_i = \langle open', closed', r' \rangle$ on α_i initialized with perimeter of search $DS_{i-1} = \langle open, closed, r \rangle$. For every abstract state on α_i, $s_j^{\alpha_i}$, we denote S_j to the set of all states mapped to $s_j^{\alpha_i}$, i. e., $S_j = \{s : \alpha_i(s) = s_j^{\alpha_i}\}$. We initialize search DS_i as:*

$$open'_g = \{s_j^{\alpha} : \exists_{s \in S_j} s \in open_g\}$$

$$closed'_g = \{s_j^{\alpha} : \forall_{s \in S_j} s \in closed_g\}$$

$$r' = r$$

[5]Other works consider $r' = r + 1$, which is the special case for unit-cost domains. Our definition of r' takes into account generic action costs. For example, in domains with operators of cost 0, our definition takes the right value, $r' = r$.



(a) Original state space. (b) Abstract state space.

Figure 8.5: Example of initialization of the closed list. B was already expanded by the perimeter search, so there is no need to reopen it, pruning the spurious path in the abstract state space between A and D or E.

The open list is seeded with the abstract states of all the states in the perimeter, as in the case of common PPDBs. On the other hand, our initialization of the closed list differs from definitions used by previous work. In these previous works, the closed list is initialized as the empty list. Our proof of Theorem 8.3 shows that, in order to prove perimeter abstraction heuristics to be consistent and admissible, it is strictly forbidden to remove states from $open$ or add new states into $closed$. In other words, if we consider the sets of states associated with all $open$, $open$', $closed$ and $closed$' (by interpreting the abstract states as sets of states), they must meet the following constraints in the initialization of DS_i: $open^{i-1} \subseteq open^i$ and $closed^i \subseteq closed^{i-1}$.

Both the initialization of $closed$ to the empty list and the one used in Definition 8.5 are valid. The advantage of our initialization of $closed$ is that it includes more states into the closed lists. This allows us to detect as duplicates all the states that were included into $closed$. Hence, the number of states to be expanded (and therefore the search effort) is reduced. Moreover, pruning these states may also prune spurious paths that do not exist in the original state space, making the heuristics more informed. Thus, our initialization of the closed list derives more informed heuristics and still guarantees them to be admissible.

It may not be clear from the previous explanation in which cases the initialization of the closed list may help to prune spurious paths, improving the heuristic estimates. Figure 8.5 shows an example that shows how preserving some states in the closed list may produce a more informed heuristic than leaving it empty. Consider the part of the original state space depicted in Figure 8.5a. There are seven states, including the goal state G, three states that form a path to the goal, A, B and C, and another three states that are in another part of the state space D, E and F. From our figure, one can infer the distance to the goal of A, B, C and G, but nothing can be said about the distance to the goal of D, E and F: they are arbitrarily far from the goal state and even may lead to dead-ends.

A perimeter of depth 3 is constructed in the original state space, in which case states G, C and B are expanded — assuming unitary costs. The search frontier only contains state A. This frontier is used to initialize an abstract search in the abstract state space of Figure 8.5b. In this abstract state, states C and D are mapped to the same abstract state. On the other hand, all the other states stay as in the original state space (they have been mapped to themselves). If the closed list of the search in the abstract state space is initialized empty, there is a path from A to D of cost 2 and the heuristic value of D will be $5 = 2 + 3$. However, if the closed list is initialized with states expanded by the perimeter search, states B and G are detected as duplicates (note that the state C, D cannot be included in closed because D was not expanded). As B is not expanded in the abstract search, the spurious path between A and D is pruned and the heuristic value of D, E, F and other states will be higher.

Theorem 8.3. *Perimeter abstraction heuristics are admissible and consistent.*

Proof. A heuristic h is consistent if and only if $h(s) \leq h(s') + c(l)\ \forall (s, l, s')$. Given the definition of h_{PA}, we divide the proof in two cases, depending on whether s' was expanded by a search DS_i (case 2) or not (case 1).

In case 1, if s' was not expanded in any search, then its h-value equals ∞ or the larger frontier cost (depending on whether the last search was finished or truncated). In either case, the heuristic value for s' is the maximum and the inequality holds, $h_{PA}(s) \leq h_{PA}(s')\ \forall s$.

In case 2, let DS_i be the first search in which s' was expanded, and doing so with cost c. Then $h_{DS_i}(s') = c$. As no previous search expanded s', $h_{PA}(s') = \max_{0,\dots,i} h_{DS_i}(s') \geq c$. Thus, we need to prove the inequality $h_{PA}(s) \leq c + c(l)$. As s' was expanded in regression search DS_i, necessarily s was inserted in $open_{c+c(l)}$. Again, we divide the proof in two cases, depending on whether some search $DS_j, j > i$ expanded the bucket $open_{c+c(l)}$ (case 2.1) or not (case 2.2).

In case 2.1, s was first expanded by a search $DS_j, j \geq i$. As s was inserted in $open_{c+c(l)}$, the value with which s was expanded is at most $c + c(l)$, since s remains in $open_{c+c(l)}$ and either is not in *closed* or has been expanded with a lower cost. Note the relevance of Definition 8.5, where $open'$ preserves all the states in the open list and $closed'$ prevents states from being closed if they are not expanded. Therefore $h_{PA}(s) \leq c + c(l) = h_{PA}(s') + c(l)$.

In case 2.2, all the perimeter searches are truncated before reaching $c + c(l)$. $h_{PA}(s)$ will be the next frontier cost $d' \leq c + c(l) = h_{PA}(s') + c(l)$.

Admissibility is derived from consistency, given that the heuristic is perfect for goal states: $\forall s_\star \in S_\star,\ h_{PA}(s_\star) = 0$. □

8.5 Frontier Shrinking

One key aspect of perimeter abstraction heuristics is that, instead of restarting abstract searches from scratch, they restart the search from the current frontier by relaxing the open and closed lists. Thus, in order to initialize the abstract search, they have to perform the mapping from states in the original state space to abstract states, according to Definition 8.5. In this section, we consider how to initialize abstract searches efficiently when using a symbolic representation of the sets of states involved in the searches. We call this operation "shrinking the frontier" because we expect the frontier of the abstract state space to be more compact according to Theorem 8.2.

According with Definition 8.5, there are two different types of frontier shrinking operations. *Existential shrinking* is used to shrink the open list and *universal shrinking* is the operation applied to the closed list. Both take as input a set of states S_B and an abstraction function α and compute the set of abstract states that will be used to initialize the open and closed list, $S_B^{\exists\alpha}$ and $S_B^{\forall\alpha}$, respectively. Next, we describe how to implement both shrinking operations when the sets of states are described with BDDs and our abstraction is a PDB or a SM&S abstraction.

8.5.1 Frontier Shrinking in PDBs

PDB abstractions are characterized by a set of variables $\mathcal{V}_\alpha$, so that the value of the remaining variables is completely ignored. Implementing the existential and universal shrinking operations in this setting, reduces to applying the standard existential/universal quantification over variables that are not in $\mathcal{V}_\alpha$, as formulated in Equations 8.2 and 8.3.

$$S_B^{\exists\alpha} = \exists_{\mathcal{V}\setminus\mathcal{V}_\alpha} S_B \tag{8.2}$$

$$S_B^{\forall\alpha} = \forall_{\mathcal{V}\setminus\mathcal{V}_\alpha} S_B \tag{8.3}$$

8.5.2 Frontier Shrinking in SM&S

In SM&S abstractions, like in PDBs, we can distinguish between relaxed and non-relaxed variables. However, instead of completely ignoring the relaxed variables, an M&S abstraction is used to keep some information about the relaxed variables. Therefore, we describe the SM&S abstraction in terms of the set of relaxed variables $\mathcal{V}_\alpha$ (the relevant variables of the M&S abstraction) and the set of equivalences induced by the M&S abstraction, $\sim^\alpha$. Recall that each abstract state s^α is a set of states such that every pair of states $s_1, s_2 \in s^\alpha$ are equivalent, $s_1 \sim^\alpha s_2$. Equations 8.4 and 8.5 show how to compute the existential and universal shrinking of a set of states, S_B, given a BDD representation of S_B and each s^α:

$$S_B^{\exists\alpha} = \bigvee_{s^\alpha \in S^\alpha} \left((\exists \mathcal{V}_\alpha \, (S_B \wedge s^\alpha)) \wedge s^\alpha \right) \tag{8.4}$$

$$S_B^{\forall\alpha} = \bigvee_{s^\alpha \in S^\alpha} \left((\forall \mathcal{V}_\alpha \, ((S_B \wedge s^\alpha) \vee \neg s^\alpha)) \wedge s^\alpha \right) \tag{8.5}$$

As in the case of PDBs, both operations use the existential/universal BDD quantification of the abstracted variables. However, in this case, we must iterate over all the abstract states in S^α to keep their information. For each abstract state, s^α, we compute its existential shrinking in three consecutive steps:

1. $S_B \wedge s^\alpha$ is the subset of S_B which corresponds to any state in s^α. However, in our final result all partial states mapped to s^α must be indistinguishable.

2. Existential quantification of $\mathcal{V}_\alpha$ gets all the assignments to variables $\mathcal{V} \setminus \mathcal{V}_\alpha$ that have been reached for any $s \in s^\alpha$. That way, we keep the values of other variables while ignoring the relaxed variables, just like in the existential shrinking of PDB abstractions.

3. Finally, we compute the conjunction with s^α to reset the value of $\mathcal{V}_\alpha$ to only states in s^α.

As an example, take Figure 8.2 on page 129. Abstract state e_0, represents the partial states 00 and 11. Our first step, gets all the states in S_B that fit that description: 00000 and 11011. Then, the existential quantification forgets about the value of the relaxed variables, obtaining the set of states $xx000$ and $xx011$. Finally, those assignments are valid for e_0, so we end with four states after our relaxation: 00000, 00011, 11000 and 11011.

The case of universal shrinking is similar, though in this case the second step uses a universal quantification to get only those assignments to non-relaxed variables that are true for all partial assignments mapped to s^α. Moreover, a disjunction with $\neg s^\alpha$ is necessary to ignore the values reached with other abstract states in the universal quantification.

8.6 Symbolic Perimeter Merge-and-Shrink

In the previous sections, we have presented an abstraction hierarchy based on M&S abstractions and a new model of perimeter abstraction heuristics that can exploit it. In this section, we mix those ingredients up and propose the Symbolic Perimeter M&S heuristic, SPM&S. The main purpose of this section is to describe the design and implementation decisions that remain unsettled and are needed in order to have a practical implementation of the theoretical results of previous sections.

8.6.1 SPM&S Algorithm

First, we present a high-level description of the SPM&S procedure. SPM&S is a perimeter abstraction heuristic using symbolic search to explore SM&S abstractions. Thus, SPM&S performs symbolic regression and uses M&S abstraction to gradually relax the search whenever it becomes unfeasible. Figure 8.6 depicts a high level view of the interaction between symbolic search and M&S abstractions in the SPM&S algorithm. The diagram is divided into four parts: the upper part depicts the M&S abstractions that are used to relax the search; the middle part depicts the BDDs involved in the search; the x-axis represents the search progress with the different layers labeled with their g-value; and below the axis we show the resulting heuristics of each search.

SPM&S starts computing a symbolic perimeter, $\mathit{Exp}(\alpha_0)$. This perimeter corresponds to a regression search in the original state space, so that no M&S abstraction is being used to relax the search. The first BDD with $g = 0$ contains the goal states and by successive $\mathit{pre\text{-}image}$ operations SPM&S generates the sets of states with $g = 1$, $g = 2$, etc. This search proceeds as a standard symbolic regression search. As symbolic regression is intractable for general planning domains, the search will surpass in most instances the predetermined memory or time bounds given for the precomputation of the heuristic. Then, the search is truncated (at $g = 2$ in Figure 8.6). The minimum distance to the goal of the expanded state sets is stored, transforming the list of BDDs representing the search to an ADD representing the heuristic, $h_{\mathit{Exp}(\alpha_0)}$. Up to this point, we have just generated a symbolic perimeter with symbolic regression search, and no abstractions have been used yet.

Then, M&S is used to derive an abstraction, α_1. M&S merges variables, applying shrinking if needed to fit the maximum number of abstract states (in Figure 8.6, with a limit of six abstract states, α_1 must have at most three abstract states before merging the next variable). SPM&S initializes $\mathit{Exp}(\alpha_1)$ by relaxing the open and closed lists of the previous search, using the existential and universal shrinking operations described in Section 8.5. The top levels of the BDDs in the open or closed list of the new relaxed search, $\mathit{Exp}(\alpha_1)$, are at most as large as the ADD that represents the M&S abstraction, α_1, depicted in the upper part of the Figure. This is due to the fact that all partial states related to the same abstract state are considered equivalent so that when one is reached, all of them are, as explained in Section 8.2.1. For example, in Figure 8.6, abstract state e_1 represents partial states 00 and 10. During the exploration, if state $10010\ldots$ is reached, then state $00010\ldots$ is also reached and vice versa. Hence, BDD nodes pointed to by 00 and 10 are equivalent, making the top part of any BDD in the exploration equal to the ADD representation of α_1. Also, M&S abstractions are accumulative, so the top levels of α_2 coincide with those of α_1. With the relaxation induced by the M&S abstraction the search continues, but, after some steps, it may become unfeasible again (according with the time/memory bounds set as parameters). SPM&S continues interleaving symbolic explorations and M&S iterations until an exploration is completed or time/memory bounds are violated. When finished, it returns the list of ADDs representing the heuristic.

Algorithm 8.2 shows the SPM&S algorithm. It receives as input a planning task Π and some parameters to bound the memory and time resources. The output is a heuristic H represented as a list of ADDs. Each ADD is the result of a backward symbolic exploration over an M&S abstraction α. If SPM&S exceeds the time limit $T_{\text{SPM\&S}}$ a last ADD is included (line 16). This last ADD is the standard M&S heuristic, computed as usual with an explicit traversal of the abstract state space. SPM&S starts initializing the symbolic backward search as usual to $(\mathit{open} = S_\star, \mathit{closed} = \emptyset, d = 0)$ and α is empty. Symbolic search progresses following the relaxation imposed by α (line 7).

The `Explore` procedure performs a symbolic search of $\alpha^{\text{SM\&S}}$, updating the open and closed lists and the frontier cost, d. It works as symbolic uniform-cost search presented in Section 2.4, with additional parameters that affect the stop condition. At each step, the current frontier S_e is

Algorithm 8.2: Symbolic Perimeter Merge-and-Shrink

```
Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
Input: Memory bounds: $N$, $N_F$
Input: Time bounds: $T_{SM\&S}$, $T_{Sym}$, $T_{Exp}$, $T_I$
Output: List of ADDs: $H$
1  $H \leftarrow \emptyset$
2  $abs \leftarrow \{\pi_v \mid v \in \mathcal{V}\}$
3  $\alpha \leftarrow \emptyset$
4  $(open, closed, d) \leftarrow (S_\star, \emptyset, 0)$
5  while $open \neq \emptyset$ and $t_s < T_{SM\&S}$ do
6      if $|open_d| \leq N_F$ and $t_s < T_{Sym}$ then
7          Explore($\alpha, open, closed, d, N_F, T_{Exp}, T_I$)      /* Search abstract state space */
8          $H \leftarrow H \cup$ ADD($closed, d$)      /* Insert heuristic ADD in $H$ */
9      Select( $\pi_v \in abs$)      /* Select next atomic abstraction */
10     $abs \leftarrow abs \setminus \pi_v$      /* Remove atomic abstraction from $abs$ */
11     $E \leftarrow$ Shrink($\alpha, \frac{N}{size(\pi_v)}$)      /* Get shrinking equivalence */
12     $open \leftarrow open^{\exists E}$      /* Shrink search frontier */
13     $closed \leftarrow closed^{\forall E}$      /* Shrink closed list */
14     $\alpha \leftarrow \alpha^E \otimes \pi_v$      /* Merge next variable */
15 if $open \neq \emptyset$ then
16     $H \leftarrow H \cup$ Explicit-Search($\alpha$)
17 return $H$

18 Procedure Explore($\alpha, open, closed, d, N_F, T_{Exp}, T_I$)
19     while $open \neq \emptyset$ and $t_{exp} < T_{Exp}$ do
20         $S_e \leftarrow open_d \wedge \neg closed$      /* Expand $open_d$ */
21         $open_d \leftarrow \emptyset$
22         $closed_d \leftarrow S_e$
23         while $S_e \neq \emptyset$ and $|S_e| < N_F$ and $t < T_{Exp}$ do      /* BFS with 0-cost operators */
24             $S_e \leftarrow pre{-}image_0(S_e, \alpha, T_I) \wedge \neg closed$
25             $closed_d \leftarrow closed_d \vee S_e$
26         if $|closed_d| < N_F$ and $t_{exp} < T_{Exp}$ then      /* Apply cost operators */
27             for all $i > 0 \mid \exists o \in \mathcal{O}, c(o) = i$ do
28                 $open_{d+i} \leftarrow open_{d+i} \vee pre{-}image_i(closed_d, \alpha, T_I)$
29             $d \leftarrow \min_c open_c \neq \emptyset$
```

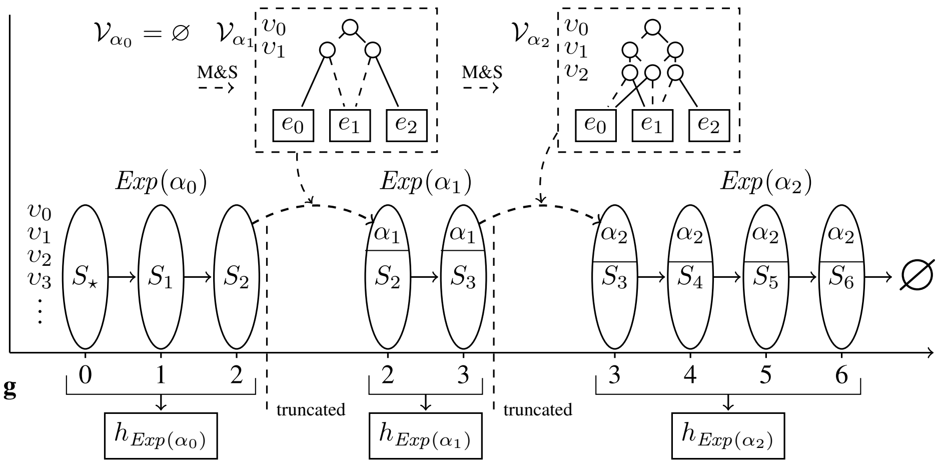


Figure 8.6: SPM&S example with binary variables, unary cost operators and a limit of six abstract states for M&S. Ellipses represent sets of states in the symbolic regression searches. In abstract searches, the top part of each BDD corresponds to the ADD that represents the M&S used to relax the search.

extracted from $open_d$, removing already expanded states (line 20). Before applying non-zero cost operators, a breadth-first search applying only 0-cost operators is performed in order to obtain all states reachable with cost d (lines 23 - 25). Those states are stored in the closed list and their successors are generated and inserted in the corresponding bucket of the open list. $pre\text{-}image_c$ computes the set of predecessor states in the state space $\alpha \otimes \Pi_{\mathcal{V}\setminus\mathcal{V}_\alpha}$ with operators of cost c. The exploration finishes when the open list is empty or the search is truncated. The heuristic derived from its closed list is stored in H as an ADD (line 8).

Several parameters bound the memory and time used by the algorithm. Memory is controlled by the maximum number of M&S abstract states N and the maximum number of nodes N_F to represent the search frontier. Four different parameters limit the time employed by the algorithm t_s or by the exploration t_{exp}. $T_{\text{M\&S}}$ aborts the heuristic generation to guarantee termination. T_{Sym} prevents SPM&S from performing more symbolic explorations to focus on completing the M&S abstraction. T_{Exp} fixes a maximum time for each individual exploration to avoid consuming all the time in one single exploration. Finally, T_I limits the maximum time employed in one $pre\text{-}image$. If T_I is exceeded, not only the $image$ but the whole exploration is halted. In order to avoid starting another $image$ as hard as the halted one, the maximum number of nodes in the frontier search is reduced to $N_F = \frac{|S_e|}{2}$. All these parameters are set independently of the domain and only depend upon the memory and time resources available to the planner.

When the search is truncated because one of the bounds was reached, the current abstraction is substituted, merging a new variable and shrinking if needed. After shrinking the abstraction, we also shrink the search frontier, $open$ and $closed$, in order to continue the search. Note that, after shrinking the search frontier with the new abstraction, the search only continues if the relaxed frontier has an acceptable size. SPM&S keeps including variables into the abstraction until the search is small enough, i. e., the relaxed frontier is smaller than the parameter N_F. The use of perimeter search prevents SPM&S from starting a new exploration per each merged variable, which could cause redundant work in case that most searches were truncated at the same frontier cost.

Once the heuristic is computed as a list of ADDs, it can be used in the search to solve the planning problem. The heuristic of SPM&S is computed just as any other perimeter abstraction heuristic, following Definition 8.4.

8.6.2 Theoretical Properties

We conclude with a recapitulation of the theoretical results that can be applied to the $h_{\text{SPM\&S}}$ heuristic.

Corollary 8.4. *$h_{SPM\&S}$ heuristic is admissible and consistent.*

Proof sketch. It follows from Theorem 8.3, since $h_{\text{SPM\&S}}$ is a particular instance of perimeter abstraction heuristic. □

One of the advantages of M&S is that it is possible to control its time and memory usage by appropriately setting the maximum number of abstract states, M. Assuming that M is polynomially bounded, computing the abstraction has polynomial complexity (Helmert et al., 2007). SPM&S does not ensure that the full symbolic exploration over an abstraction has a more concise BDD representation than the original problem. Also, relaxing the search frontier with existential shrinking could actually enlarge it. Fortunately, Theorem 8.2 in Section 8.2.1 derived a bound for the maximum number of BDD nodes needed to represent any set of states in the exploration.

Corollary 8.5. *Let α be an M&S abstraction with relevant variables $\mathcal{V}_\alpha$, generated with a maximum number of abstract states M. Let S_B be a BDD describing a set of states on $\Theta^\alpha \otimes \Pi_{\mathcal{V}\setminus\mathcal{V}_\alpha}$ using a variable order whose first/top variables correspond to $\mathcal{V}_\alpha$. Then, the size of S_B is bounded by:*

$$|S_B| \leq M \left(|\mathcal{V}_\alpha| + 2^{|\mathcal{V}\setminus\mathcal{V}_\alpha|+1} \right)$$

Proof. It follows from Theorem 8.2, just assigning the bound M for the number of abstract states in every intermediate M&S abstraction. □

Corollary 8.5 ensures that, as variables are merged into the abstraction, the complexity of its full symbolic exploration decreases. The bound for the top part of the BDD grows linearly on M, while the bound for its bottom part is exponentially reduced. Thus, eventually the full symbolic exploration of the abstraction will be tractable. In the limit, the exploration is performed over a linear sized state space and can be explicitly explored, just as the original M&S algorithm does. Theorem 8.5 requires relevant variables for the abstraction to be placed in the top levels of exploration BDDs. For this to hold in every symbolic exploration, the symbolic search must use the same variable ordering as the M&S merging strategy. Since changing the variable ordering of a BDD may cause an exponential blow-up, we use the same variable ordering in order to guarantee an efficient computation of SPM&S.

8.7 Empirical Evaluation

In this section, we evaluate the empirical performance of our new heuristic, SPM&S, and compare it against the two methods it combines: M&S and a symbolic perimeter, as well as to other state-of-the-art heuristics for cost-optimal planning. We also evaluate the influence of different parameters of our heuristic, in particular the combination of SPM&S with several PDB and SM&S abstraction hierarchies.

SPM&S has been integrated in the FAST DOWNWARD planning system (Helmert, 2006b). All the symbolic searches, for the construction of the perimeter and the exploration of the abstract state spaces, use the enhancements for image computation and invalid-state pruning presented in Part I of this thesis.

8.7.1 SPM&S Parameters

The SPM&S algorithm takes as input different parameters that can be classified into three categories:

Memory and time bounds SPM&S heuristics, as other abstraction heuristics like PDBs or M&S, rely on a heavy precomputation phase in order to efficiently compute the heuristic value for each state in the search. We defined parameters that limit the time and memory invested in the preprocessing phase in order to guarantee that it successfully terminates. Larger bounds will likely result in more informed heuristics at expenses of investing more preprocessing time. Of course, the optimal value of these parameters directly depends on the total resources that are available to the planner. SPM&S parameters were manually set to fit the competition setting: the maximum number of nodes to represent the state set to expand is $N_F = 10{,}000{,}000$. A maximum time is set to each individual $image$ ($T_I = 30s$), exploration ($T_{Exp} = 300s$) and symbolic search ($T_{Sym} = 900s$). Finally, the heuristic preprocessing is interrupted after $T_{SPM\&S} = 1200s$ seconds to ensure that the search is always started.

Variable ordering All the symbolic searches use a static variable ordering. We consider two different variable orderings: GAMER and FAST DOWNWARD orderings. As we previously analyzed in Section 7.4 on page 115, the variable ordering used by GAMER is optimized for symbolic search and the one used in FAST DOWNWARD is suitable for the merging strategy of M&S. However, we always use the same ordering for the symbolic search and merging variables in M&S since otherwise there is no guarantee about the BDD sizes.

Abstraction hierarchy SPM&S can be used in combination with different abstraction strategies that select the abstract state spaces that will be traversed in order to compute the heuristic estimates. In our experiments we used two different types of abstraction hierarchies: the SM&S hierarchies we presented in Section 8.2 on page 126 and PDB hierarchies.

We define PDB hierarchies by starting with the pattern that contains all the variables (equivalent to the original problem) and abstracting one variable at a time. Thus, the PDB hierarchies are defined in terms of a variable ordering. In order to select variable orderings for the construction of the PDB hierarchies, we use the preexisting merge linear strategies for M&S:

- *Level* (*lev*): Follows the BDD variable ordering, so it is the same abstraction layers than SM&S.

- *Reverse level* (*rev*): Reversed version of level. When used in combination with GAMER ordering, level and reverse level are equivalent up to the arbitrary tie-breaking, though *lev* abstracts variables from the top of the BDD and *rev* from the bottom. In combination with FAST DOWNWARD ordering, however, the difference might be more relevant.

- *Cggoal-lev* (*cgl*): Always selects a variable related in the causal graph to an already selected variable. If there is none, it selects a goal variable. Ties are broken according to the level criterion.

- *Cggoal-rnd* (*cgr*): As CGGoalLevel but breaking ties randomly.
- *Goalcg-lev* (*gcl*): First it selects all goal variables and then variables related to them in the causal graph, breaking ties according to the level criterion.
- *Random* (*rnd*): selects a random variable ordering. Useful as a baseline approach.

SM&S abstraction hierarchies are defined with the merge and shrink strategies. The merging strategy is always set to exactly the same linear ordering than the one used for the BDDs. Only this ordering guarantees that the resulting BDDs will eventually be tractable after abstracting away enough variables, as explained in Section 8.2.1 on page 128. Regarding shrinking strategies, we use again the ones that have been previously defined in the literature explained in Section 6.4.2 on page 98: bisimulation (b), greedy bisimulation (g) and f-preserving shrinking (*fh*). By default, all the shrinking strategies reduce the abstract state space enough to ensure that no M&S abstraction has more than 10,000 abstract states.

8.7.2 Coverage of A* Search with SPM&S

Table 8.1 shows the coverage of different configurations of SPM&S against the two combined approaches: symbolic perimeter and M&S. Additionally, we report the coverage of LM-CUT for comparison.

As we anticipated, M&S results are better with the FAST DOWNWARD ordering while the symbolic search (SP) benefits from the GAMER ordering. In total, SPM&S performs better with the GAMER ordering, though it may vary depending on the domain (e. g., FREECELL, LOGISTICS, or NOMYSTERY).

The dominance of SPM&S over SP and M&S is quite clear. Even though there are a few cases where SPM&S preprocessing does not terminate and is outperformed by the symbolic perimeter alone, this is compensated by combining the strengths of both algorithms. Moreover, in some cases SPM&S obtains better results than any of the two techniques as in GRID or ROVERS, for example.

The use of M&S abstractions does not payoff in general over using simpler abstractions such as PDBs. There are some domains, however, were the M&S heuristics help to improve performance over PDBs. This contradicts previously published results (Torralba et al., 2013b) that presented SM&S hierarchies as superior candidates than PDBs. The main reason for this difference is the usage of state-invariants as we will analyze in Section 8.7.5. The impact of using different abstraction hierarchies is analyzed in depth in Section 8.7.4.

Regarding the comparison with LM-CUT, both heuristics exhibit complementary strengths. In total, a symbolic perimeter with GAMER ordering is sufficient to obtain similar coverage results than LM-CUT, especially considering the coverage score that adjusts the weight given to MICONIC. Using abstraction strategies on top of symbolic perimeter further improves the results of SPM&S allowing it to outperform LM-CUT in general. Nevertheless, there are some domains were LM-CUT clearly performs best, such as AIRPORT, MICONIC or SCANALYZER.

8.7.3 Informativeness of SPM&S Heuristics

Our coverage comparisons show that SPM&S is a state-of-the-art heuristic, outperforming M&S and being competitive with LM-CUT heuristics. However, there are two main factors that influence the performance of heuristics: informativeness and computational effort. In this subsection we analyze how well informed the SPM&S heuristics are with respect to the two combined techniques and other

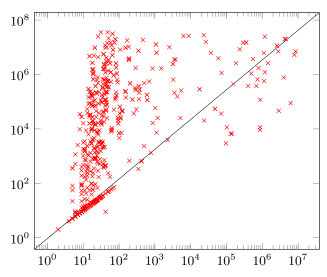
	GAMER ORDERING						FD ORDERING						
	SP		SPM&S		M&S		SP		SPM&S		M&S		LM-
	–	PDB	b_{10k}	g_{10k}	b_{10k}	g_{10k}	–	PDB	b_{10k}	g_{10k}	b_{10k}	g_{10k}	cut
AIRPORT (50)	25	25	25	25	23	23	25	26	25	25	23	23	**29**
BARMAN (20)	**8**	**8**	**8**	**8**	4	4	**8**	**8**	**8**	**8**	4	4	4
BLOCKSWORLD (35)	30	**32**	30	30	25	28	30	31	30	30	26	28	28
DEPOT (22)	**7**	**7**	**7**	**7**	**7**	6	**7**	**7**	**7**	**7**	**7**	6	**7**
DRIVERLOG (20)	12	12	**13**	**13**	**13**	**13**	12	12	**13**	**13**	**13**	12	**13**
ELEVATORS08 (30)	23	**24**	23	23	14	15	21	**24**	21	21	13	13	22
ELEVATORS11 (20)	18	**19**	18	18	12	13	18	**19**	18	18	11	11	18
FLOORTILE11 (20)	**14**	**14**	**14**	**14**	8	8	**14**	**14**	**14**	**14**	8	8	**14**
FREECELL (80)	22	30	21	21	19	19	22	**35**	23	23	19	19	15
GRID (5)	2	**3**	**3**	**3**	2	2	2	2	2	2	2	2	2
GRIPPER (20)	**20**	**20**	**20**	**20**	8	8	**20**	**20**	**20**	**20**	11	8	7
LOGISTICS 00 (28)	16	16	16	18	16	16	16	17	**20**	17	**20**	16	**20**
LOGISTICS 98 (35)	4	4	5	5	5	4	3	4	5	5	4	4	**6**
MICONIC (150)	107	107	107	107	60	52	117	117	117	117	64	52	**141**
MPRIME (35)	22	22	22	22	22	**23**	**23**	22	20	20	21	**23**	**23**
MYSTERY (30)	15	15	15	15	**17**	**17**	15	15	15	15	14	**17**	**17**
NOMYSTERY11 (20)	13	14	14	14	16	14	13	16	**18**	16	**18**	14	14
OPENSTACKS08 (30)	**30**	**30**	**30**	**30**	21	21	20	20	20	20	21	21	21
OPENSTACKS11 (20)	**20**	**20**	**20**	**20**	16	16	15	15	15	15	16	16	16
OPENSTACKS06 (30)	**11**	**11**	10	**11**	7	7	7	7	7	7	7	7	7
PARCPRINTER08 (30)	**23**	**23**	**23**	**23**	20	20	22	22	22	22	20	20	22
PARCPRINTER11 (20)	**18**	**18**	**18**	**18**	15	15	17	17	17	17	15	15	17
PARKING11 (20)	1	1	1	1	1	1	1	1	1	1	1	1	**2**
PATHWAYS-NONEG (30)	4	4	4	4	4	4	4	4	4	4	4	4	**5**
PEG-SOLITAIRE08 (30)	29	29	29	29	27	27	29	**30**	29	29	29	29	28
PEG-SOLITAIRE11 (20)	19	19	19	19	17	17	19	**20**	19	19	19	19	18
PIPESWORLD-NT (50)	15	16	14	15	16	15	15	15	15	15	**19**	15	17
PIPESWORLD-T (50)	12	14	13	13	14	**17**	12	11	13	15	14	**17**	12
PSR-SMALL (50)	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	49	**50**	49
ROVERS (40)	12	**13**	**13**	**13**	8	6	11	11	11	11	7	6	7
SATELLITE (36)	**9**	**9**	**9**	**9**	6	6	7	7	7	7	6	6	7
SCANALYZER08 (30)	12	12	12	12	12	12	12	12	13	13	12	13	**15**
SCANALYZER11 (20)	9	9	9	9	9	9	9	9	10	10	9	10	**12**
SOKOBAN08 (30)	28	28	28	28	29	29	28	28	28	28	29	29	**30**
SOKOBAN11 (20)	20	20	20	20	20	20	20	20	20	20	20	20	20
TIDYBOT11 (20)	14	14	15	16	12	13	9	9	9	9	12	13	**17**
TPP (30)	**8**	**8**	**8**	**8**	6	6	**8**	**8**	**8**	**8**	6	6	7
TRANSPORT08 (30)	12	**13**	12	12	11	11	12	**13**	12	12	11	11	11
TRANSPORT11 (20)	7	**9**	8	8	6	6	8	8	8	8	6	6	6
TRUCKS (30)	**10**	**10**	**10**	**10**	8	8	**10**	**10**	**10**	**10**	8	8	**10**
VISITALL (20)	11	12	12	10	10	**16**	12	12	12	11	9	**16**	10
WOODWORKING08 (30)	**28**	**28**	**28**	27	14	14	15	16	16	16	13	14	22
WOODWORKING11 (20)	**20**	**20**	**20**	**20**	9	9	9	10	10	10	8	9	15
ZENOTRAVEL (20)	10	10	10	11	10	10	9	9	12	11	12	10	**13**
TOTAL COV (1396)	800	**822**	806	809	649	650	756	783	774	769	660	651	796
SCORE COV (36)	19.32	**19.93**	**19.74**	**19.83**	16.25	16.51	18.08	18.63	18.74	18.48	16.59	16.48	18.56

Table 8.1: Coverage of SPM&S against other approaches. M&S heuristic, symbolic perimeter (SP), symbolic perimeter PDBs (SPPDB) and SPM&S with bisimulation and greedy bisimulation and a limit of 10,000 abstract states.

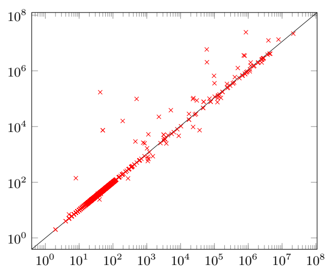
state-of-the-art heuristics. From the heuristics in the literature, LM-CUT is a good choice because it is known to be an expensive but very well-informed heuristic (Helmert and Domshlak, 2009).

In order to compare the informativeness of the heuristics, we report two different metrics: expanded nodes and initial h-value. The initial h-value is a direct comparison of the value of each heuristic for particular states, while the number of expansions measure the ability of the heuristic to reduce the overall search effort. Figures 8.7 and 8.8 show the comparisons in both metrics of SPM&S against different competitors in all commonly solved problems. For the heuristic value comparison, we omit PARCPRINTER since the large heuristic values of that domain difficult the visualization.

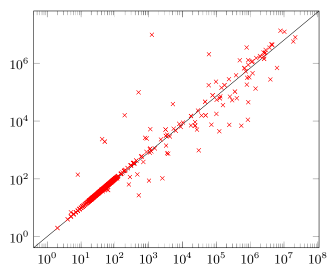
The comparison of SPM&S against SP and SPPDB reveals that the main strength of the heuristic



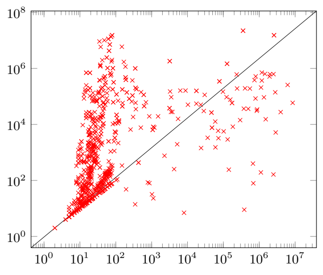
(a) M&S



(b) SP

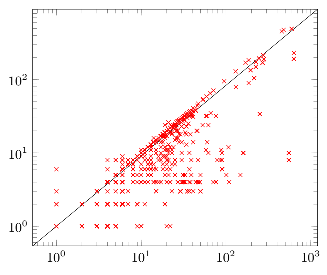


(c) SPPDB

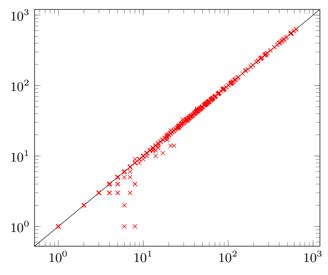


(d) LM-CUT

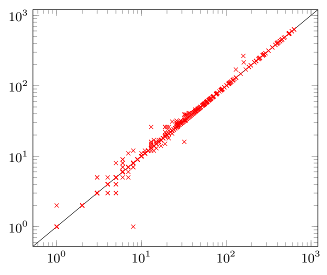
Figure 8.7: Number of expanded nodes by SPM&S (x-axis) against other heuristics (y-axis) in commonly solved instances. Points above the main diagonal are problems solved faster by SPM&S.



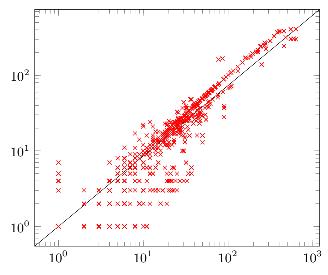
(a) M&S



(b) SP



(c) SPPDB



(d) LM-CUT

Figure 8.8: Initial state heuristic value of SPM&S (x-axis) against other heuristics (y-axis) in commonly solved instances of all domains except PARCPRINTER. Points below the main diagonal are problems in which SPM&S is more informed.

is due to the perimeter search. Using abstraction heuristics to extend the information of the perimeter derives strictly more informed heuristics. Therefore, both SPPDB and SPM&S obtain more accurate heuristics than SP and, in some cases, significantly reduce the number of expanded nodes by several orders of magnitude. However, in most cases, SPM&S and SPPDB obtain the same heuristic value than SP, showing that the construction of a symbolic perimeter is already a strong heuristic.

Perhaps more surprisingly, the comparison with LM-CUT shows that SPM&S is a well-informed heuristic compared to the current state of the art in cost-optimal planning. It does not only derive the perfect heuristic for many problems (for most of the problems that were solved with less than 1,000 expansions), but also derives a heuristic competitive with LM-CUT in harder instances. The difference in heuristic value of the initial state shows that SPM&S is more precise in many tasks, showing that it is a well-informed heuristic across the entire state space and not only close to the goal.

8.7.4 Analysis of Abstraction Hierarchies

Our analysis in the previous sections have used only the default configuration of SPPDB. However, there is a wide variety of strategies that result in different heuristics. Moreover, the initialization of the abstract searches with the perimeter can be disabled, in order to compare with standard symbolic PDB approaches.

	GAMER ORDERING		FD ORDERING	
	$\mathcal{P}$	$\neg\mathcal{P}$	$\mathcal{P}$	$\neg\mathcal{P}$
SP	19.32 (800)	–	18.08 (756)	–
PDB-rev	20.04 (825)	19.78 (812)	18.69 (784)	18.50 (769)
PDB-lev	**20.21** (826)	19.99 (818)	18.91 (783)	18.81 (779)
PDB-cggoal-lev	20.03 (823)	19.76 (813)	18.60 (775)	18.37 (765)
PDB-cggoal-rnd	19.77 (813)	19.77 (810)	18.53 (772)	18.69 (770)
PDB-goalcg-lev	**20.28 (828)**	**20.23 (828)**	19.09 (785)	19.15 (785)
PDB-rnd	19.74 (814)	19.60 (812)	18.61 (775)	18.48 (768)
SM&S$_{bop10k}$	19.74 (806)	19.85 (810)	18.74 (774)	18.90 (773)
SM&S$_{gop10k}$	19.83 (809)	19.88 (812)	18.48 (769)	18.36 (766)
SM&S$_{fh10k}$	18.75 (762)	18.81 (764)	17.34 (710)	17.34 (709)

Table 8.2: Coverage score and total coverage of SPM&S with different abstraction hierarchies. The symbolic perimeter, SP, that does not make use of any abstraction, is compared against PDB and SM&S hierarchies. All the versions are ran with perimeter abstractions ($\mathcal{P}$) and with standard abstraction heuristics ($\neg\mathcal{P}$). The best configurations and those deviating in only 1% are highlighted in bold.

Table 8.2 shows the results of the different abstraction hierarchies. The first observation is that most abstraction strategies help to improve the symbolic perimeter, SP. The only strategy that does not produce better results than a simple perimeter is the SM&S abstractions with *fh* shrinking. Of course, the bad performance is not due to having a less informed heuristic, but because of exceeding memory in the M&S heuristic for some domains.

Overall, the best strategy is PDBs with *gcl*, with a total coverage of 828. Other PDB strategies such as *lev* or *rev* have a similar performance and the difference is not significant. Interestingly, the performance of PDBs with a random selection of variables is a very competitive approach, even obtaining better results than SM&S approaches or other PDB strategies. This shows the lack of good abstraction strategies for our setting, where finding good abstractions is not straightforward.

Perimeter abstractions are useful in combination with some strategies, though $\neg\mathcal{P}$ in combination with *gcl* obtains the same results as the best version using perimeter abstractions. In combination with SM&S hierarchies, the results with and without perimeter are similar, so SPM&S is not taking advantage of the perimeter initialization in those cases. Note, however, that even though $\neg\mathcal{P}$ ignores the perimeter for the initialization of abstract searches, it uses the perimeter for other purposes such as selecting one abstraction from the hierarchy (the one in which the perimeter is reduced enough) and to return the maximum between the heuristics derived from all the searches (including the perimeter and one or more abstract searches).

Interestingly, using the perimeter to initialize abstract searches is especially useful in combination with strategies that do not distinguish goal from non-goal variables such as *lev* or *rev*. In those cases the perimeter derives information from the goal variables, so that they can be abstracted away without a huge information loss. Similarly, if all the goal variables are included in the abstraction (*gcl*), using the perimeter is not important.

8.7.5 Removing Spurious States

The results shown in Table 8.1 on page 146 contradict previously published results, which reported SPM&S being better with SM&S hierarchies than when using only PDBs (Torralba et al., 2013b). One important difference in the experimental setting is that in this thesis we are using the improvements presented in Chapter 4: removing invalid operators and using the state-invariant constraints with e-deletion to enhance symbolic search. As we analyzed in Section 4.7.4 on page 76, the use of mutexes greatly improves the performance of symbolic regression search. This is no different in SPM&S, were symbolic regression is used both in the original and the abstract state spaces. In the following, we take a closer look to the use of state-invariant constraints in SPM&S.

	GAMER ORDERING		FD ORDERING	
	$\mathcal{M}_\emptyset$	$\mathcal{M}$	$\mathcal{M}_\emptyset$	$\mathcal{M}$
SP	16.68 (715)	19.32 (800)	16.24 (689)	18.08 (756)
pdbs	17.08 (730)	**19.93 (822)**	16.82 (708)	18.63 (783)
b_{10k}	17.48 (735)	**19.74** (806)	17.35 (717)	18.74 (774)
g_{10k}	17.41 (732)	**19.83** (809)	17.02 (719)	18.48 (769)

Table 8.3: Coverage of different configurations of SPM&S without state-invariant pruning ($\mathcal{M}_\emptyset$) and using it ($\mathcal{M}$). The best configurations and those deviating in only 1% are highlighted in bold.

Table 8.3 compares the performance of our approaches when disabling the removal of spurious states during the search. When spurious states are not removed from the search ($\mathcal{M}_\emptyset$), the performance of all our symbolic perimeter approaches decreases significantly. However, the benefits of removing spurious states are greater in the case of SP and SPPDB than for any SPM&S version. Interestingly, M&S abstractions obtain better results than PDBs to relax a symbolic perimeter only when the symbolic search performs worse. This can be partially explained by the ability of M&S heuristics to prune spurious states, whose impact is heavily reduced when those states are being removed anyway thanks to the state-invariants.

8.8 Summary

In this chapter we have presented SPM&S, a novel admissible heuristic for cost-optimal planning that combines symbolic search, M&S abstractions and perimeter abstraction heuristics. A preliminary version of this chapter was previously published (Torralba et al., 2013b). SPM&S has several contributions with respect to other state-of-the-art heuristics:

1. Using symbolic search to search M&S abstract state spaces. M&S represents abstract state spaces explicitly. For that reason, explicit search is an efficient method to search the abstract state spaces. Thus, at a first glance, symbolic search cannot enhance M&S. However, we defined SM&S abstractions as larger M&S abstractions based on the smaller explicitly represented abstractions. SM&S abstractions can be searched with symbolic search, in an attempt to combine the strengths of symbolic PDBs and the flexibility of M&S abstractions.

2. A more general schema of perimeter abstraction heuristics than the one used by previous works. The main difference is that our perimeter abstraction heuristics consider multiple perimeters in different abstract state spaces, instead of only one. Moreover, we considered action costs and improved the initialization of the closed list to derive better heuristics.

3. We proposed the use of symbolic search in a perimeter abstraction setting. We analyzed how to symbolically represent sets of abstract states and how to efficiently perform the mapping between states in different state spaces.

In summary, SPM&S uses M&S abstractions to relax a symbolic backward search, keeping only partial information about some variables of the problem. The SM&S hierarchy leaves the choice of different abstract state spaces to search, ranging from the original non-abstracted state space to the one explored by the M&S heuristic. Every SM&S abstraction reduces the abstract state space size, so that there is a trade-off between the abstract search complexity and the heuristic informativeness. However, searching all these abstractions is clearly redundant work, since heuristics of less relaxed abstractions are strictly more informed than more relaxed ones. Predicting which SM&S abstraction has the best benefit/effort relation is a complex task. SPM&S solves this problem using perimeter abstraction heuristics, that search in the less relaxed state spaces while it is feasible and use more abstracted state spaces to continue the search afterwards.

Experimental results show that SPM&S successfully obtains results at least as good as the best of either one of the two techniques it combines: symbolic search and M&S. Even though M&S abstractions are highly dependent of the shrinking policy, SPM&S improves the performance of a symbolic perimeter even with a simple shrinking strategy, SPM&S-all. Moreover, it improves M&S results for all shrinking policies used. On the other hand, bisimulation shrinking derives good abstractions for SPM&S on some domains but not in others. This suggests there is still room for improvement with new shrinking policies specific for SPM&S (i. e., taking into account the symbolic search frontier and not preserving information about states already expanded by previous explorations).

Chapter 9

Symbolic Bidirectional Heuristic Search

While symbolic bidirectional uniform-cost search has a remarkable performance without using any heuristic, heuristics still play a central role in cost-optimal planning. Thus, it is of major interest to combine both approaches by using heuristics in a symbolic bidirectional search.

In this chapter, we propose to use of perimeter abstraction heuristics in symbolic bidirectional search. Perimeter abstractions are a good fit for our setting because they take advantage of the ability of symbolic uniform-cost search to efficiently generate perimeters. We analyze how symbolic bidirectional searches can be used to explore abstract state spaces in order to compute heuristic estimates.

Our empirical evaluation shows that, while diverse abstraction strategies may improve the performance of symbolic bidirectional search, none of the abstraction strategies we used provides good performance across different domains.

9.1 Introduction

The results of Part I of this thesis have shown that symbolic bidirectional blind search has become one of the best alternatives for cost-optimal planning, outperforming not only A^*-based planners but also BDDA*, the symbolic search variant of A^*. Nevertheless, most cost-optimal planners are still based on A^* guided with an admissible heuristic. Different kinds of heuristics have been proven very useful to lessen the search effort in forward search.

In this Chapter, our aim is to combine heuristics and symbolic bidirectional search into a symbolic bidirectional heuristic planner. Although bidirectional heuristic search (BHS) has a long history (Pohl, 1969; Champeaux and Sint, 1977), it has not been extensively explored for domain-independent optimal planning. This can be attributed to two different types of reasons. On the one hand, backward search in planning has not been widely used in the last years due to the inherent difficulties of regression in planning and the computational cost of detecting collision between frontiers (Alcázar et al., 2014). On the other hand, most bidirectional heuristic search algorithms have not been shown to be superior to regular A^* and bidirectional brute-force search (Kaindl and Kainz, 1997; Barker and Korf, 2015). Due to these difficulties, BHS has fallen out of favor, with the majority of the search and planning community working mostly with regular A^* and similar techniques.

Nevertheless, a successful application might be possible in our setting. On the one hand, symbolic search has already helped us to overcome the problems related to regression and frontier-collision detection in domain-independent planning, as described in detail in Section 5.1 on page 81. On the other hand, the fact that bidirectional blind search already outperforms A* suggests that considering heuristics cannot be too harmful if done carefully. As the use of heuristics can reduce the effectiveness of bidirectional brute-force search, we will use heuristics only in cases where bidirectional brute-force search is unfeasible.

The challenge that we face is to introduce heuristics in the symbolic bidirectional search, taking advantage of them whenever they are informative and avoiding a huge performance loss whenever the heuristics are less useful.

Our approach consists of starting a bidirectional blind search and introduce heuristics afterwards, only when the search without heuristics has failed to solve the problem in a reasonable amount of time. Thus, our algorithm will mimic the behavior of bidirectional blind search in domains where it performs best. This approach may look unfeasible from an explicit-search perspective, where it is better to use heuristics in the first place, since changing the heuristic requires to evaluate all the states in the current frontier and there might be too many at that point. If the heuristic is useful, it would be best to use it in the first place. However, as we will detail in Section 9.4, our symbolic Lazy-A* implementation presented in Section 2.6.2 on page 33 is able to reevaluate the entire search frontier in a reasonable time.

Referring to heuristics, we use SPM&S, the symbolic perimeter abstraction heuristics proposed in Chapter 8. They are adequate for our purposes, since they can be used in our symbolic setting. Moreover, there is a good synergy between bidirectional search and perimeter-based abstraction heuristics. Whenever blind search does not completely solve the problem, our perimeter heuristic takes advantage of the effort made by the blind search by using the current search perimeter. This synergy somehow alleviates some of the problems associated with other previous BHS approaches, as we will study in Section 9.2.1.

Nevertheless, in order to use SPM&S in a BHS, we have to extend it to the bidirectional case, where heuristics must provide estimates to both search frontiers. In order to generate estimations in both search directions, we perform a bidirectional search in the abstract state space, instead of the common unidirectional search — usually in the backwards direction. This is a novel approach since, up to our knowledge, there is not any literature referring to bidirectional search in abstract state spaces. On the other hand, if the abstract search is completed, the effort is doubled for performing it in both search directions. To address that, we will consider partial abstractions, where the abstract search can provide useful heuristic estimates even if it has not been completed.

We call our algorithm SymBA*, an acronym of symbolic bidirectional A*. SymBA* performs bidirectional searches on different state spaces. It starts in the original search space and, when the search becomes too hard, it derives an abstraction heuristic enhanced by the frontier of the opposite direction. The planner decides at any point whether to advance the search in the original state space, enlarging the perimeter, or search in an abstract state to provide better heuristic estimations for the original state space search. In this case the novelty lies on the dynamism of the approach, in the sense that SymBA* computes and refines heuristics when needed and can advance in either direction, using the frontier of one direction to enhance the heuristic for the opposite direction.

In summary, the main contributions of this Chapter are:

- Present SymBA*, a bidirectional heuristic search algorithm for domain-independent optimal planning.

- Introduce bidirectional perimeter abstraction heuristics, that perform bidirectional search in the abstract state spaces in order to provide heuristic distance estimations to the initial state and the goal.

- A theoretical analysis of how to perform partial and perimeter abstraction heuristics when heuristics are used in the abstract state space searches.
- A comparison of our approach and other related works in the literature, such as hierarchical heuristic search.
- An empirical analysis of SymBA*'s performance with respect to our base algorithms, symbolic bidirectional blind search and A* with SPM&S.

The remainder of this chapter is organized as follows. First, we analyze the literature regarding BHS and hierarchical heuristic search in Section 9.2. We introduce our notation in Section 9.3. Then, we present our BHS algorithm, SymBA*, in Section 9.4. In Section 9.5 we describe in detail how heuristic values are computed and prove that the overall algorithm is optimal. In Section 9.6 we report our extensive empirical evaluation of SymBA* performance. The chapter concludes with a brief summary of the main conclusions in Section 9.7.

9.2 Related Work

In this section we review the related work in the literature, that was not already covered by our analysis of the state-of-the-art symbolic search algorithms in Chapter 2 and abstraction heuristics in Chapter 6. In particular, we review works of two different areas. On the one hand, we describe algorithms that use heuristics in a bidirectional search. On the other hand, we analyze hierarchical heuristic search algorithms that use abstraction heuristics for a single problem. These hierarchical heuristic search algorithms skip the expensive precomputation phase of abstraction heuristics and compute the abstract distances lazily, avoiding searching unnecessary parts of the abstract state space. Both areas are relevant for our work because they have not only served as inspiration, but also form a theoretical basis to build our algorithm.

9.2.1 Bidirectional Heuristic Search

In bidirectional search, two different searches are performed in opposite directions, i. e., from the initial state to the goal and vice versa. The promise is to reduce the search depth by a factor of two, potentially reducing the search effort by an exponential factor with respect to unidirectional search. Bidirectional search has been able to live up to its promises in the blind case, with the bidirectional version of breadth-first search obtaining exponential speed ups with respect to the unidirectional one. Thus, it is not surprising that bidirectional heuristic search is a well-studied topic, with different approaches that attempt to take advantage of heuristics in the bidirectional setting. However, empirical results have never been too favorable whenever combining bidirectional search and heuristic estimates. A good general overview of BHS approaches is presented in the heuristic search book (Edelkamp and Schrödl, 2012). Bidirectional heuristics algorithms can be broadly split into three categories (Kaindl and Kainz, 1997): front-to-end, front-to-front and perimeter search.

Front-to-End Bidirectional Heuristic Search

Front-to-end algorithms were the first approaches to BHS. In front-to-end BHS, algorithms make use of two consistent heuristics, $h^{fw}(s)$ and $h^{bw}(s)$, which estimate the distance from a state s to the goal and initial state, respectively. The first approach to introduce heuristics in bidirectional search was the Bidirectional Heuristic Path Algorithm (BHPA) (Pohl, 1969; Pohl, 1971). BHPA performs two opposite A* searches, one forward search from the initial state to the goal making use of $h^{fw}(s)$ and one backward search from the goal towards the initial state guided with $h^{bw}(s)$. Whenever the two frontiers meet, a solution plan has been found. However, in order for proving optimality the searches must continue until one of them has expanded all nodes with an f-value lower than the optimal solution. Unfortunately, in the worst case, the search effort is the same than performing two A* searches independently, so that the performance is worse than unidirectional A*.

BS* (Kwa, 1989) is an improved version of BHPA that applies several enhancements to prune the searches by interchanging information between them:

- **Trimming**: Discard any state s in open whose $f^*(s)$ is larger than $c(\pi)$, the cost of the current best plan.
- **Screening**: Avoid inserting into the open list any state s whose $f^*(s)$ is larger than $c(\pi)$, the cost of the current best plan.
- **Nipping**: Avoid expanding any state that has already been expanded by the search in the opposite direction.
- **Pruning**: Whenever a node is closed by nipping, its descendants can be removed from the open list of the opposite search.

Trimming and screening are general optimizations that are usually used with unidirectional A* too. Nipping and pruning are specific to the bidirectional case since they take advantage of one frontier to prune the opposite. However, even though all these improvements enhance the performance of BS* with respect to BHPA, BS* was never shown to be more efficient than A*.

The reasons why front-to-end BHS was not better than standard A* were well studied (Kaindl and Kainz, 1997). They disproved the long-believed hypothesis that the two frontiers were passing each other in BHPA. They provided evidence that even though the frontiers collision in a reasonable time and BHPA is able to find a solution quickly, most effort of BHPA is spent on proving the solution to be optimal. Further improvements of front-to-end search consist of using the real-cost values of states to determine the heuristic error and improve the heuristic values of other states accordingly (Kaindl and Kainz, 1997). However, these algorithms were never shown to outperform unidirectional A* search.

Recent research by Barker and Korf (2015) provides evidence against the hypothesis of (Kaindl and Kainz, 1997) that BHPA spends most of the time in proving optimality. Nevertheless, Barker and Korf argue that any front-to-end BHS will be outperformed by either bidirectional brute-force search (when the heuristic is weak) or A* (when the heuristic is strong) except in pathological cases.

Front-to-Front Bidirectional Heuristic Search

Due to the difficulties in outperforming A* by using front-to-end BHS, researchers considered other ways to introduce heuristics in the bidirectional setting. Even though it was later shown that the frontiers passing each other was not really the cause for the limitation of front-to-end BHS, some algorithms were developed with the goal of redirecting both frontiers towards each other. These algorithms are usually called wave-shaping or front-to-front algorithms.

Front-to-front algorithm require a heuristic, $h(s, s')$, that is able to estimate the distance between any two nodes in the state space, s and s'. The bidirectional heuristic front-to-front algorithm (BHFFA) (Champeaux and Sint, 1977; Champeaux, 1983) is the first front-to-front approach.

BHFFA computes the heuristic of a node as the minimum distance to any state in the opposite frontier. Even though the increased heuristic accuracy significantly decreases the number of node expansions, the time spent on computing the heuristic value from the state to every state in the opposite frontier is simply too large. It is possible to reduce the number of evaluations by using d-node retargeting (Politowski and Pohl, 1984) that avoids to consider irrelevant parts of the opposite frontier. However, even with such optimizations, the cost of computing the distance of each node with respect to the opposite frontier is too large.

Other Approaches

Other alternative approaches to traditional bidirectional search include:

- Perimeter search (Dillenburg and Nelson, 1994; Manzini, 1995), that we already covered in Section 8.3 on page 131. A relevant extension for our work is the heuristic perimeter search (Linares López, 2005). Heuristic perimeter search is a variant of perimeter search that constructs a perimeter biased towards the initial state, according to a given heuristic. The perimeter search is performed with an IDA* search, instead of the usual uniform-cost search, generating a non-uniform perimeter.

- Single-frontier bidirectional search (Felner et al., 2010; Lippi et al., 2012) performs a single search where every node is qualified with a pair of states s and t which stand for states reached in the forward and backward directions respectively. When expanding a node, SFBS determines what state to expand or, in other words, in what direction to continue the search. By deciding locally the direction of the search the algorithm actually performs bidirectional search.

- Approaches that combine bidirectional with unidirectional search by considering several phases (Kaindl et al., 1999; Pulido et al., 2012).

9.2.2 Hierarchical Heuristic Search

Abstraction heuristics use the optimal distance in an abstract state space as an estimation for the original problem. These optimal distances are usually precomputed before starting the search, by traversing the entire abstract state space. This approach is valid in heuristic search problems such as the N-puzzle or Rubik's cube because the precomputed heuristic is reused to solve an arbitrarily large number of problem instances. However, traversing the entire abstract state space might be prohibitive for solving a single task.

Hierarchical heuristic search focuses on using abstraction heuristics in cases where the abstraction is generated for a particular problem, so that the computational cost of computing the abstraction should be amortized. In order to do so, hierarchical heuristic search explores the abstract state space lazily avoiding searching parts that are irrelevant for the problem at hand. Whenever a state is evaluated, a search in the abstract state space is conducted to retrieve the optimal distance from the corresponding abstract state. These techniques are called "hierarchical" because searches in the abstract state space may use an abstraction heuristic as well, leading to a well-defined hierarchy of abstractions, $\alpha_0, \alpha_1, \ldots$. Different hierarchical heuristic search algorithms differ on how they compute the optimal cost for abstract states.

Hierarchical A* (Holte et al., 1996) conducts an A* search in the abstract state space, whereas Hierarchical IDA* (Holte et al., 2005) uses IDA* (Korf, 1985) to reduce the memory used by the algorithm. In both cases, the abstract state search may be guided with another heuristic, which can be another hierarchical heuristic. However, these algorithms have an important drawback, a lot of search effort is made to compute the cost of a few abstract states so that many abstract states are re-expanded.

Several caching techniques can be used in order to avoid recomputing the same distances again and again (Holte et al., 1996):

- h^*-caching: When computing the optimal distance $h(\alpha(s), \alpha(s_\star))$ store the known optimal abstract distances of $\alpha(s)$ and all the abstract states lying in the optimal abstract plan from $\alpha(s)$ to $\alpha(s_\star)$. This causes the resulting heuristic to be inconsistent, though there is no need to reopen nodes since the properties of the heuristic guarantee that all nodes prematurely closed with a wrong g-value do not lie on an optimal plan from s_0 to $s_\star$.
- Optimal-path caching: Whenever there is a state for which $h^*(s)$ is already known, include a goal with an additional cost of $h^*(s)$ instead of expanding s.
- P-g caching: After computing the optimal distance $h(\alpha(s), \alpha(s_\star))$, assign each state in the search s' a heuristic value equal to $h(\alpha(s), \alpha(s_\star)) - g(s')$. P-g caching subsumes h^*-caching but can only be used with consistent heuristics and in domains where operators have inverses.

Despite the use of these caching techniques, both hierarchical A* and hierarchical IDA* must generate the same abstract nodes several times, because thanks to h^*-caching and optimal-path caching we can avoid reexpanding those nodes in the optimal path from $\alpha(s)$ to $\alpha(g)$ but all other nodes in the abstract searches must be expanded again if they appear in future abstract searches.

Switchback (Larsen et al., 2010) (or its version with enhanced stop criterion, Short-Circuit (Leighton et al., 2011)) proposes to use an abstract search in the opposite direction to determine the value of abstract states. This is the same method used by approaches that precompute the heuristic values, though in this case the search is stopped when the abstract state is found. Again, the search can be guided with a heuristic, which in this case will be an estimation of the cost to reach the abstract initial state. The advantage with respect to other hierarchical heuristic search algorithms is immediate: each abstract state is expanded only once. In order to prove optimality of the algorithm, Switchback relies on the fact that the g-value of any expanded node in A* with a consistent and admissible heuristic is guaranteed to be optimal (Nilsson, 1982).

In automated planning, problems in a domain may differ in the number of objects or even in the goals of the problem. Thus, using the same abstraction heuristics in multiple instances is not possible. This makes hierarchical heuristic search a perfect fit for planning, since a new abstraction heuristic is generated for every problem. However, state-of-the-art planners do not use hierarchical heuristic search. Despite the fact that hierarchical heuristic search was specifically designed for this case, planners that use abstraction heuristics always precompute the entire abstract state space distances before starting the A* search.

9.3 Notation

In the following sections, we describe SymBA*, a BHS algorithm that performs searches on different state spaces. We denote a bidirectional search tree on a state space, Θ_i, as T^{Θ_i}. A bidirectional search is composed of two unidirectional searches in opposite directions: a forward search, $T^{\Theta_i}_{fw}$, and a backward search, $T^{\Theta_i}_{bw}$. We will use $T^{\Theta_i}_u$ to denote a unidirectional search in an unspecified direction and $T^{\Theta_i}_{\neg u}$ to denote the search in the opposite direction.

Each search tree T_u consists of an open list, $open(T_u)$, and a closed list, $closed(T_u)$. The g-value of a search state s is denoted as $g(s)$ in cases where there is a single search and $g_{fw}(s)$ or

$g_{bw}(s)$ whenever we want to differentiate between the forward and the backward search. As usual, we denote the optimal g-value of s, $g^*(s)$. Also, we will write $g(\alpha(s))$ or $g(s^\alpha)$ to denote the g-value of an abstract state. We follow similar notation for the h and f-values. Finally, we denote by $g(T)$ and $f(T)$ the minimum g and f-values of any state in the open list of the search T.

9.4 SymBA*: Symbolic Bidirectional A*

SymBA* performs several symbolic bidirectional A* searches on different state spaces. First, SymBA* starts a bidirectional search in the original state space. Since no abstraction heuristic has been derived yet, it behaves like symbolic bidirectional blind search. At each iteration, the algorithm performs a step in a selected direction, i. e., expands the set of states with minimum g-value in that frontier. This search continues until the next step in both directions is deemed as unfeasible, because SymBA* estimates that it will take too much time or memory. Only then, a new bidirectional search is initialized in an abstract state space. Both the forward and backward abstract searches are initialized with the frontiers of the current original search. The abstract searches provide heuristic estimations for the original search, increasing the f-value of states in the search frontiers. Eventually, the search in the original state space will be simplified (as the number of states with minimum f-value will be smaller)[1] and SymBA* will continue expanding states in the original search space.

Figure 9.1 shows a diagram of the bidirectional searches performed by an execution of SymBA*. After generating a perimeter around s_0 and $s_\star$, a new search is started in an abstract state space, starting from the perimeter frontier. The abstract state space is explored in a bidirectional uniform-cost fashion, exploiting the strengths of symbolic bidirectional search. After both abstract frontiers meet, the algorithm continues expanding only the intersection of both frontiers, until it reaches one of the perimeters (the forward perimeter in the example of Figure 9.1). These abstract states correspond to real states in the perimeter frontier, which are selected for expansion. Other states in the perimeter have a larger f-value, so that they do not need to be immediately expanded. After generating the next successor states (the ellipse in the inferior part of the figure), no additional search is required to compute the abstract distances. The states that must be expanded correspond to the abstract states in the next abstract bucket.

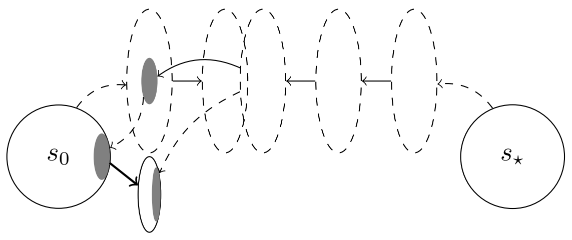


Figure 9.1: Diagram of SymBA* algorithm that shows how the algorithm computes the heuristic to decide which states in the perimeter frontier should be expanded. Dashed ellipses represent sets of states in an abstract symbolic bidirectional search.

One important feature of the algorithm is the lazy evaluation of the heuristics. The search in abstract state spaces is delayed until strictly needed to simplify the original search. We model this

[1]This is not entirely true in the symbolic case, as having fewer states does not mean that the BDD that represents them is smaller, but in most cases there is a positive correlation.

Algorithm 9.1: SymBA*

```
Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$
Output: Cost-optimal plan or "no plan"
1  $T^{\Theta}_{fw} \leftarrow \langle s_0, \Theta \rangle$
2  $T^{\Theta}_{bw} \leftarrow \langle s_\star, \Theta \rangle$
3  $SearchPool \leftarrow \{T^{\Theta}_{fw}, T^{\Theta}_{bw}\}$
4  $\pi \leftarrow$ "$noplan$"
5  while $\max(f(T^{\Theta}_{fw}), f(T^{\Theta}_{bw})) < Cost(\pi)$ do
6      if $\exists T \in SearchPool$ s.t. Is-Candidate(T) then
7          $T^{\Theta_i}_u \leftarrow$ Select-Search($SearchPool$)
8          $\pi' \leftarrow$ Expand-frontier($T^{\Theta_i}_u$)
9          if $\Theta_i = \Theta \land \pi' \neq \emptyset \land Cost(\pi') < Cost(\pi)$ then
10             $\pi \leftarrow \pi'$
11         Notify-h($S^{\Theta_i}, S^{\Theta}$)
12     else
13         $\alpha \leftarrow$Select-abstraction($T^{\Theta}_{fw}, T^{\Theta}_{bw}$)
14         $\left\langle T^{\Theta_\alpha}_{fw}, T^{\Theta_\alpha}_{bw} \right\rangle \leftarrow$Apply($\alpha, T^{\Theta}_{fw}, T^{\Theta}_{bw}$)
15         $SearchPool \leftarrow SearchPool \cup \{T^{\Theta_\alpha}_{fw}, T^{\Theta_\alpha}_{bw}\}$
16     return $\pi$
```

by considering a pool of active searches and letting the algorithm decide which search should be advanced at any step.

Algorithm 9.1 shows the pseudocode of SymBA*, which decides whether to advance the search in the original state space or in one of the abstract state spaces. SymBA* maintains a pool of all the current active searches. The pool is initialized with a bidirectional search in the original state space. The algorithm proceeds while the current best solution so far has not been proven optimal (line 5). The current plan, π, is optimal if it has a cost smaller or equal than the current f-value of any of the searches in the original state space, $f(T^{\Theta}_{fw})$ or $f(T^{\Theta}_{bw})$. At each iteration, the algorithm filters the searches that are valid candidates from the pool and selects the most promising ones.

A search is a valid candidate if and only if it is both *useful* and *feasible*. The search in the original search space is always useful. A search in an abstract search space is useful if and only if there are still states from the opposite frontier that do not correspond to a state already expanded in the abstract space. The main intuition behind this is that non-useful searches cannot possibly simplify the next step in the original search space. A search is *feasible* if the estimated time and number of nodes needed to perform the next step does not surpass the bounds imposed as parameters: *maximum step time and number of nodes*. Among all the searches that are *valid candidates*, we select those that have a greater minimum f-value, because they are closer to proving that the current solution is optimal. If more than one search has the same minimum f-value, we select the one whose next step is estimated to take less time.

Once a search has been selected, the procedure `ExpandFrontier` expands the set of states that have a minimum g-value among those that have a minimum f-value, like in the standard implementation of BDDA*. If we progress in the original state space search, a new plan of lower cost than the best so far may be found. If an abstract state space is selected, we update the heuristic value of the other searches in the opposite direction in the pool, both abstract and original.

If there are no valid search candidates (line 12), a new bidirectional search is added to the pool (which amounts to two new searches). First, we select a new abstraction strategy (line 13). Using the strategy, we relax the current frontiers of the original state space search, until the frontier size is small enough to continue the search and there is no previous equivalent search (line 14). Finally, the new search is included in the pool to be selected in subsequent iterations.

Implementation with Symbolic Lazy BDDA* Even though the SymBA* algorithm could in principle be implemented with explicit search, using symbolic search has several advantages that can dramatically improve the performance. This is hardly a surprise, because across this Thesis we have already highlighted the advantages of symbolic search when performing bidirectional search and perimeter abstractions. But the role of symbolic search in SymBA* goes even further.

One of the main characteristics of SymBA* is that the heuristics change dynamically during the search procedure. Not only the algorithm may decide to initialize a new abstract search at any point, but also every time that an abstract search performs a step, the heuristic value of states in the original search may increase. Re-evaluating the entire search frontier repeatedly may be too costly if done naïvely, becoming a bottleneck and making the entire algorithm unfeasible.

In our case, we use our Lazy implementation of symbolic A* presented in Section 2.6.2 on page 33. Lazy BDDA* keeps the states organized by g-value and only evaluates them before an expansion to obtain the subset of states with an f-value lower or equal than the current f-value. Since the states are not organized by their h-value, changing the heuristic in the middle of the search does only require to re-evaluate the set of states currently selected for expansion, without any additional computation in the open list. Therefore, only when the heuristic values of all the states selected for expansion have been raised, the algorithm evaluates other states in the open list in order to select the ones with minimum f-value.

Removing a heuristic is also possible, though one must be careful to preserve consistency since the heuristic value of all the states in the search frontier may decrease. Instead of re-evaluating every state in the open list, it suffices to keep track of the previous minimum f-value, which can be interpreted as applying pathmax propagation (Méro, 1984). However, the set of states previously selected for expansion are not necessarily the best ones according to the new heuristic. Hence, to preserve consistency we re-evaluate the states in ascending order of g, i. e., breaking ties in favor of those states with lower g-value as usual in symbolic BDDA*. Again, the advantage of Lazy BDDA* is that any evaluated state whose bound is lower or equal than the current f-value can be expanded before evaluating other states in the search.

9.5 Partial Bidirectional Abstractions

In SymBA*, abstract searches are used to compute heuristic estimations that can be used in the original state space, just as in many other heuristic search algorithms. However, the algorithm has some particular characteristics, such as being bidirectional or interleaving the search in the original and abstract state spaces. Our aim is to obtain heuristic estimations as informed as possible while proving that our algorithm is admissible. In particular, we will consider three different aspects of our abstract searches:

- **Bidirectional**: In bidirectional A* search, in order to inform both search frontiers, the heuristic has to estimate goal and initial state distances. To obtain those distances in the abstract state space, two searches must be performed: the abstract forward search computes the initial-state distances to inform the backward search in the original state space, while the abstract backward search computes goal-distance estimations to inform the forward search. Moreover, since these two searches are performed in the same state space they can exchange information to improve their performance.

- **Partial**: Partial abstractions do not search the entire abstract state space completely, but stop the search at some point. For every abstract state that has not been expanded, the heuristic establishes a lower bound for the optimal cost in the abstract state space.
- **Non-uniform perimeter**: Instead of starting the abstract regression search from the goal, perimeter abstractions initialize abstract searches with a search frontier. Usually the search frontier is just a perimeter around the goal state, in which all states in the frontier have the same goal-distance. In our case, however, a heuristic search is used to initialize the perimeter so the goal-distance of states in the perimeter can be much wider. This does not affect the admissibility of the approach, though. As we will prove later, any search which only closes states with their optimal g-value may be used as a perimeter to seed an abstract search. On the other hand, having two perimeters to initialize the abstract bidirectional search provides additional information that can be used to enhance the initialization of the abstract search.

Perimeter and partial abstraction heuristics are not new and the conditions for consistency and admissibility are well-known for them. As an example, the SPM&S heuristic presented in Chapter 8 uses partial perimeter abstractions. More detailed definitions of partial and perimeter abstractions were already presented in Section 8.4 on page 133. However, the use of bidirectional abstract searches requires us to reconsider partial and perimeter abstractions, in order to understand the interactions between these techniques and prove that our algorithm is admissible. Next, we describe our abstraction heuristics in depth, explaining the relationship between the different techniques applied and providing the reasons that make them necessary. We extend the definition of partial and perimeter abstractions to the case when the abstract state spaces are explored by A* instead of uniform-cost.

9.5.1 Bidirectional Abstractions

As SymBA* requires to derive heuristics in both directions, it has to perform forward and backward searches in the abstract state space. However, it is dubious that performing both searches independently is worth it, since the effort to generate the heuristic is doubled without clear compensation. In order to really take advantage of the bidirectional nature of searches in SymBA*, one has to consider the relations between these two searches. Therefore, a bidirectional search is performed in the abstract state space in which the forward search represents a “perfect” heuristic for the backward search and vice versa.

A distinction must be made between bidirectional search in the original state space and the ones performed in abstract state spaces. The first type of search aims to find a plan, so whenever the two frontiers meet a plan is retrieved and techniques like nipping and pruning (Kwa, 1989) should remain activated to prune both search frontiers avoiding redundant work. On the other hand, searches on abstract state spaces are used to derive heuristic estimates for the original search. In this case, we cannot apply all pruning techniques that work in the original state space. Pruning techniques like trimming and screening, that are based on the length of the best current plan found, can still be applied. However, pruning based on the intersection of both frontiers, such as nipping and pruning must be disabled in order for the estimations to be admissible.

Thus, the interaction between both searches is reduced to use each other as a heuristic, but otherwise they do not directly interact to detect the collision of their frontiers. Nevertheless, using each search as heuristic for the opposite is enough to reduce the search effort of both directions. Figure 9.2 reflects the interaction between both searches by drawing in different colors parts performed by different searches.

While the original search performs nipping, avoiding the searches to explore the space already traversed in the opposite direction, abstract searches continue after the collision of both frontiers (highlighted in green). They must redundantly explore the space in order to provide admissible estimations to the original search. However, as soon as the intersection of both frontiers is not empty, only the states in the intersection need to be expanded, greatly reducing the search effort in both directions. This can be interpreted as retrieving the optimal abstract plans and checking whether there exists a corresponding "real plan". In the worst case, if the abstract searches are completed and they traverse the entire abstract state space, the search effort is doubled plus an overhead for using heuristics.

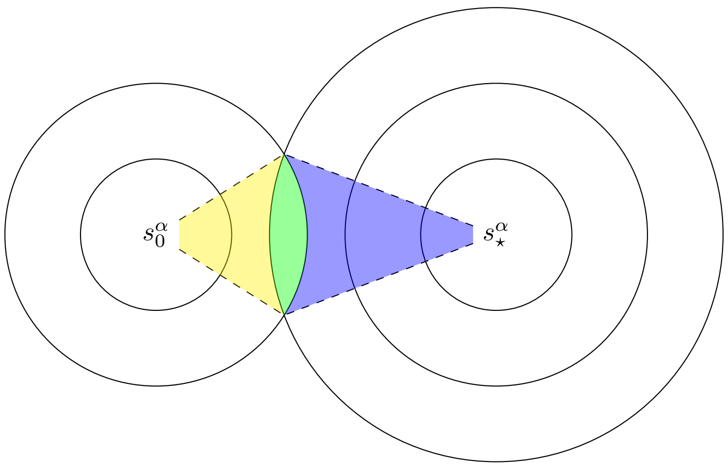


Figure 9.2: Interaction of abstract bidirectional searches. The intersection of both searches is colored green. The continuation of forward and backward search after the frontier collision is colored blue and yellow, respectively.

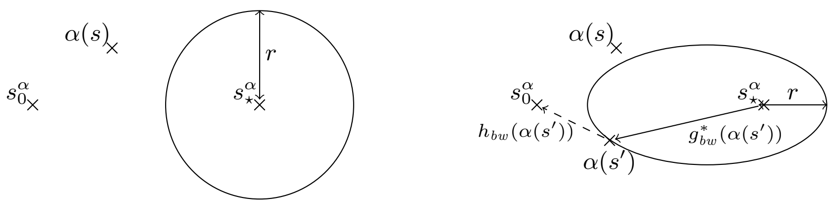
However, making the abstract searches bidirectional may reduce the search effort invested in the abstract state space. Intuitively, what our algorithm does is to compute all optimal abstract paths in the abstract state space by means of a bidirectional search. Once those paths have been identified, we may expand the corresponding paths in the original state space that correspond to those optimal abstract paths of cost $f^*(\alpha(s_0))$. If we succeed in finding a plan of cost equal to the optimal abstract cost, the plan must be optimal since the cost in the original state space cannot be smaller than in the abstract state space.

In case that no corresponding plan with that cost exists in the original state space, the search in the abstract state space can be continued, to retrieve all paths of cost $f^*(\alpha(s_0)) + 1$. This process continues until an optimal solution is found in the original state space.

9.5.2 Partial Abstractions

Partial abstractions are needed in SymBA* because we aim to search large abstract state spaces that cannot be entirely explored. If we try to continue the search until all the states in the opposite frontier are expanded, we will likely need to expand most parts of the abstract state space. For example, in Switchback (Larsen et al., 2010), the abstract search continues until the optimal abstract distance is known for every state in the current frontier of the original search. This means that, if one single state is a dead-end both in the original and the abstract state space, the abstract state space must be completely traversed before continuing the search. Most parts of that computation are irrelevant, since the termination criterion of an A* search to guarantee that the solution π is optimal is that, for every not expanded state s, $f^*(s) \geq c(\pi)$. In other words, the heuristic value of a state does not matter as long as it is large enough to guarantee that its f-value is not the minimum among the current f-value of other states in our search. Thus, the advantage of partial abstractions in SymBA* is double: when the abstract searches are intractable it helps the algorithm to avoid getting stuck in an abstract search and when the abstract searches are tractable it reduces the amount of abstract state space that needs to be searched in order to prove that the solution is optimal.

In partial abstractions, abstract states can be classified depending on whether they have been expanded or not. For those abstract states that have been already expanded, their optimal distance is already known and can be used as a heuristic estimate. The question is which heuristic value can we assign to abstract states that have not been expanded yet. The usual answer in unidirectional search is to use the minimum g-value in the open list of the search (see Definition 8.3 on page 136). In that case, the next minimum g-value in the search is a satisfactory lower bound for the cost of non-expanded states because the abstract state space is explored with a uniform-cost search and the g-value is regularly increased. However, if the abstract state space is explored with an A* search, this bound is no longer useful because as the search continues the bound may remain constant. Consider the example of Figure 9.3, in which we explore the abstract state space with uniform-cost or A* search and want to evaluate a non-expanded state, s. In uniform-cost search, the explored part of the state space is represented as a circle because nodes are expanded in ascending order of g_{bw}. In that case, $h(s) = r$. If the abstract exploration uses a heuristic in A*, the exploration is biased towards the goal, i. e., s_0^α. While the lower bound $h(s) = r$ may still be used as heuristic, we say that it is not satisfactory because all the effort wasted in expanding nodes closer to s_0^α does not help to increase r.



(a) Uniform-cost search (b) A*

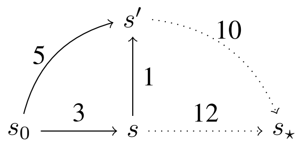
Figure 9.3: Bound for partial heuristic abstractions. The bound is usually $h(s) \geq r$. In A* we use $f(s) \geq \min\limits_{\alpha(s')} f^\alpha(s') \geq \min\limits_{\alpha(s')} g^*_{bw}(\alpha(s')) + h_{bw}(\alpha(s'))$.

Therefore, when the abstract state space is traversed using A*, the estimation for not expanded states must be based on the minimum f-value of the search, instead of the g-value. Thus, instead of setting a bound for the h-value of the states, we use a bound for their f-value, based on the following inequality: $f(T^\alpha) \leq f^*(\alpha(s)) \leq f^*(s)$. As $f(s) = g(s) + h(s)$, we can translate the bound for $f(s)$ to a heuristic value $h(s) = f(T^\alpha) - g(s)$.

Proposition 9.1. *Let T^α be an admissible abstract search and s be a state such that $\alpha(s)$ has not been expanded by T^α. Then $f(T^\alpha) - g(s)$ is an admissible estimation.*

Proof. By definition $g(s) \geq g^*(s)$. Moreover, $f(T^\alpha) \leq f^*(\alpha(s)) \leq f^*(s) = g^*(s) + h^*(s)$. Therefore, the inequality holds: $h^*(s) = f^*(s) - g^*(s) \geq f^*(s) - g(s) \geq f(T^\alpha) - g(s)$. □

The problem is that such heuristic may be inconsistent and expand states with suboptimal g-value, as shown in the example of Figure 9.4. In the example, only the initial state has been expanded so there are two states in the open list: s with $g(s) = 3$ and s' with $g(s') = 5$. In the abstract state space, a shortcut has been introduced in the path through $\alpha(s)$ such that $g(T^\alpha)(s) = 1$ and $h^\alpha(s) = 4$. If the abstract search has been explored until $f(T^\alpha) = 6$, clearly α (s) will have been expanded with a cost of $h^\alpha(s) = 4$ but $\alpha(s')$ will have not been expanded. In this case, $f(s) = 5 + 5 = 10 > f(T^\alpha) = 6$. However, $f(s') = f(T^\alpha)$. Thus, s' gets expanded before s with a suboptimal value of $g(s') = 3 > g^*(s') = 2$.


(a) Original state space

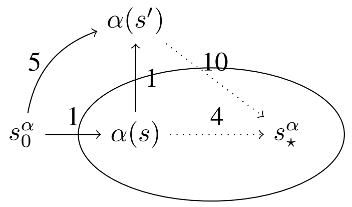

(b) Abstract state space

Figure 9.4: Example of inconsistency of bidirectional partial abstraction. Dashed edges represent paths of a given cost.

Inconsistency of heuristics is not necessarily a problem and, in some domains, it is possible to take advantage of such inconsistencies to improve the heuristic estimates (Felner et al., 2011). However, inconsistent estimates are undesirable in SymBA* because A* may close some states with a suboptimal value, requiring their later re-expansion. The problem is that SymBA* relies on the optimality of the values in the closed lists in order to use perimeter abstractions, so closing a state with a suboptimal g-value must be avoided at any cost. In the following, we define how to use the f-value of the A* search in the abstract state space of a partial abstraction heuristic and, at the same time, guarantee that all closed states will have the optimal g-value. To further improve the heuristic estimates, we rely on the fact that our symbolic BDDA* algorithm breaks ties in favor of states with smaller g-values.

The key is not to close any state with an unknown abstract optimal cost. The abstraction heuristic is still partial because there is no need to compute the exact h-value of those states that can be proven to have a higher f-value than the current minimum f in the original search. However, the abstract search cannot be stopped just at any point, but it should explore all the values with $f(\alpha(s)) \leq f$.

Definition 9.1 (Partial abstraction heuristic)**.** *Let $T^\alpha = \langle open, closed \rangle$ be an A* search over the state space Θ^α informed with an admissible and consistent heuristic h and let $f(T^\alpha)$ be the minimum f-value of any state in open $\min_{s^\alpha \in open} g(s^\alpha) + h(s^\alpha)$, or ∞ if $open(T^\alpha)$ is empty.*

Let $g^T(s)$ be the optimal g-value for those states closed by T^α and ∞ for other states whose optimal g-value is still unknown:

$$g^T(s) = \begin{cases} c & \text{if } \alpha(s) \in \mathit{closed}_c^{bw} \\ \infty & \text{otherwise} \end{cases}$$

We define the partial abstraction heuristic *of T as:*

$$h_T(s) = \min\left\{\max\left\{f(T^\alpha) - g(s), g(T^\alpha) + 1\right\}, g^T(s)\right\} \tag{9.1}$$

Intuitively, the bidirectional partial abstraction heuristic sets the f-value of every state in the search whose corresponding abstract state has not yet been expanded to the current f-value in the abstract search. For expanded states, it returns their real value, but only when the abstract search has an f-value large enough. This suffices to guarantee that the heuristic is consistent.

Lemma 9.2. *Let $T = \langle open, closed\rangle$ be an A^* search that always expands the state with lower g-value among those with lower f-value, informed with a partial abstraction heuristic. Then, for any state s such that $s \in closed(T), g(s) = g^*(s)$.*

Proof. Suppose that $g(s) > g^*(s)$. By Lemma 1 in (Hart et al., 1968), there exists a state $r \in open(T)$, $r \neq s$, which is in the optimal path from s_0 to s such that $g(r) = g^*(r)$. Since r lies on the optimal path from s_0, $g^*(r) = g(r) \leq g^*(s) < g(s)$. On the other hand, since s was selected for expansion by T, $f(s) < f(r)$. We prove that this leads to contradiction for the possible heuristic values of r and s. There are four cases, depending on the part that dominates the minimum of Equation 9.1 for states r and s:

i, ii) $h(s) = h^\alpha(s)$. Since $h(r)$ takes the minimum with $h^\alpha(r)$ and h^α is a consistent heuristic, we have a consistent estimation: $h(r) \leq h^\alpha(r) \leq h(s) + c(r, s)$. We get a contradiction as in Lemma 2 in (Hart et al., 1968).

iii) $h(s) = \max\{g(T^\alpha) + 1, f(T^\alpha) - g(s)\}$ and $h(r) = h^\alpha(r)$.

From the value of $h(s)$, we get that $f(s)$ should be at least as $f(T^\alpha)$. On the other hand, since the dominant part in the minimum expression is $h^\alpha(r)$, $f(r) \leq f(s)$. Therefore, $f(s) \geq f(T^\alpha) \geq f(r)$. We get a contradiction because $f(r)$ should be strictly greater than $f(s)$ due to the tie-breaking in favor of lower g-values.

iv) $h(s) = \max\{g(T^\alpha) + 1, f(T^\alpha) - g(s)\}$ and $h(r) = \max\{g(T^\alpha) + 1, f(T^\alpha) - g(r)\}$. Again, we consider which part dominates the maximum expression in $h(r)$:

- If $h(r) = g(T^\alpha) + 1, h(s) > h(r)$ and the estimate is consistent.
- If $h(r) = f(T^\alpha) - g(r)$, $f(r) = f(T^\alpha) \leq f(s)$, contradiction because $f(r)$ should be strictly greater than $f(s)$ due to the tie-breaking in favor of lower g-values.

□

As a side note, the requirement of the tie-breaking criterion in favor of lower g-values is only required whenever a state with minimum f could raise the heuristic value in case the search is continued. The same heuristic could work for different tie-breaking criteria, if the abstract search is always continued until $f(T^\alpha) > f$.

9.5.3 Perimeter Bidirectional Abstractions

Perimeter abstractions, that initialize the abstract search with another search frontier, are useful to obtain better heuristic estimations. The main drawback is that computing the perimeter may be expensive and could not be worth the effort. However, as SymBA* performs bidirectional search in the original state space, it does not need to perform any additional effort to compute the forward and backward perimeters. Moreover, in Chapter 8 we analyzed the synergy between symbolic search and perimeter abstractions. Operations such as mapping states in the frontier to a new abstract state space can be done symbolically without iterating over all the states in the search frontier. Thus, the use of the perimeter to initialize the abstract searches improves the heuristics without imposing a large overhead.

However, as in the case of partial abstractions, perimeter abstractions need to be redefined in order to handle bidirectional and heuristic searches. There are two main differences with respect to the definition of perimeter abstractions used in Chapter 8.

First, the perimeter search is carried out with an A* search instead of a uniform-cost search. Using uniform-cost search guarantees the perimeter to be uniform, such that all non-expanded states have a minimum distance of r, the radius of the perimeter. However, abstraction heuristics do not require a perimeter of a fixed radius to obtain admissible estimates — any frontier in the original space can be used as a seed to improve the heuristic as long as the g-value of the expanded states is optimal. Because of this, we propose the use of the frontier in one direction in a bidirectional search algorithm to enhance an abstraction heuristic used by the search in the opposite direction.

The second difference is that in the bidirectional case we have two different perimeters: the forward and the backward one. Thus, when initializing the abstract search, it is possible to use the additional information of the opposite perimeter to improve the heuristic estimates. In our definition of the initialization of perimeter abstraction searches (see Definition 8.5 on page 136), we contemplated including in the closed list all abstract states that had already been completely expanded by the perimeter search. In this case, we can take advantage of the bidirectional perimeter by ignoring all those abstract states that were completely explored by both searches, as summarized in Definition 9.2.

Definition 9.2 (Bidirectional perimeter search initialization)**.** *Let* $T_\alpha = \left\langle \left\langle open'^{fw}, closed'^{fw} \right\rangle, \left\langle open'^{bw}, closed'^{bw} \right\rangle \right\rangle$ *be a bidirectional heuristic search in* Θ^α *initialized with the perimeter of a bidirectional search* $T = \left\langle \left\langle open^{fw}, closed^{fw} \right\rangle, \left\langle open^{bw}, closed^{bw} \right\rangle \right\rangle$*. For every abstract state on* α*,* s_i^α*, we denote* S_i *to the set of all states mapped to* s_i^α*, i. e.,* $S_i = \{s : \alpha(s) = s_i^\alpha\}$*. We initialize search* A^*_α *as:*

$$open'^{fw}_g = \left\{ s_j^\alpha : \exists_{s \in S_j} s \in open^{fw}_g \right\}$$

$$open'^{bw}_g = \left\{ s_j^\alpha : \exists_{s \in S_j} s \in open^{bw}_g \right\}$$

$$closed'^{fw}_g = closed'^{bw}_g = \left\{ s_j^\alpha : \forall_{s \in S_j} s \in closed^{fw}_g \cup closed^{bw}_g \right\}$$

States closed by any of the two frontiers can be ignored in the abstract search since their heuristic value will never be computed again in any direction — due to nipping, states expanded in the opposite direction are never considered. Lemma 9.3 proves that the heuristic estimations are still admissible and consistent for all relevant states, i. e., those that have not yet been expanded in any of the perimeter searches.

Lemma 9.3. *Let h be a bidirectional perimeter abstraction heuristic explored with a backward search in Θ^α initialized with perimeters $\left\langle open^{fw}, closed^{fw} \right\rangle$ and $\left\langle open^{bw}, closed^{bw} \right\rangle$. Then, h is admissible and consistent for any state $s \notin closed_{fw}$.*

Proof. A heuristic h is consistent if and only if $h(s) \leq h(s') + c(l)\ \forall (s, l, s')$. In our case, we only contemplate states $s, s' \notin closed^{fw}$, so $\alpha(s)$ or $\alpha(s')$ will not be introduced in $closed^\alpha$ when initializing the abstract search unless s or s' have already been expanded in the perimeter search.

We divide the proof in two cases, depending on whether s' was expanded by the backward perimeter, by the abstract search or by none of them:

i) If $s' \in closed^{bw}$, then s was inserted in $open_{bw}$ with value $h(s') + c(l)$. s will be expanded in the perimeter search or in the abstract search with a cost lower or equal than $h(s') + c(l)$. Thus, $h(s) \leq h(s') + c(l)$ and the estimations are consistent.

ii) If $s' \notin closed^{bw}$, then $\alpha(s')$ may be generated in A^*_α or not. If it is not, $h(s') = \infty$ and consistency follows. If $\alpha(s')$ is expanded in α, $\alpha(s)$ will be generated and inserted in $open^\alpha$ with value $h(s') + c(l)$. Therefore, $h(s) \leq h(s') + c(l)$.

□

Theorem 9.4. *Bidirectional A* guided with perimeter abstraction heuristics returns the optimal solution.*

Proof. We show that any state s in the optimal solution is selected from expansion with its optimal value, so the heuristic in that path must necessarily be admissible. Let s be a state selected from $open_{fw}$ for expansion. If s belongs to $closed_{bw}$ it is pruned due to nipping, so we may assume that $s \notin closed_{bw}$.

Suppose that s has suboptimal g-value, $g(s) > g^*(s)$, then by Lemma 1 in (Hart et al., 1968) there exist a $r \in open_{fw}$ such that $g(r) = g^*(r)$ and r is in the optimal path from s_0 to s. Now, r may have been closed by the backward perimeter or not:

1. Suppose that r was closed in the backward perimeter, $r \in closed_{bw}$. Then, r was closed with the perfect value, $h(r) = h^*(r)$ and a solution of minimum cost passing by r is already known. Since r is in the optimal path from s_0 to s, s cannot be part of a better solution.

2. Suppose that r was not closed in the backward perimeter. Then, neither r and s were in $closed_{bw}$ when the perimeter was initialized and, by Lemma 9.3, $h(r) \leq h(s) + c(r, s)$. As the heuristic is consistent, we reach a contradiction.

□

9.5.4 Useful Searches

After defining how our heuristic estimates are computed, we may define what makes an abstract search useful. Intuitively, an abstract search is useful if it has the potential of simplifying the original search, i. e., if it may change the set of states that are going to be selected for expansion.

Definition 9.3 (Useful abstract search)**.** *Let T_u be a best-first search over a state space Θ from s_0 to $s_\star$. Let α be an abstraction and $T^\alpha_{\neg u}$ be an abstract search over Θ^α in the opposite direction, from $s^\alpha_\star$ to s^α_0.*

Let S be the set of states currently selected for expansion in T_u, i. e., a subset of those that minimize the f-value according to any given tie-breaking criteria.

We say that $T^{\alpha}_{\neg u}$ is useful *for T_u if and only if $f(T^{\alpha}_{\neg u}) \leq f(T_u)$ or $S \wedge closed(T^{\alpha}_{\neg u})$ is not empty.*

Lemma 9.5. *Let T_u be an A* search informed with a heuristic generated by an abstract search $T^{\alpha}_{\neg u}$. Let S be the set of states selected for expansion in T_u, i. e., those with minimum f-value in $open(T_u)$ with any tie-breaking criterion. Then, if $T^{\alpha}_{\neg u}$ is not useful for T_u continuing the abstract search cannot possible alter the set of states selected for expansion, S.*

Proof. Continuing the abstract search can only raise the h-value of the states. To show that, consider the definition of $h(s)$. On one hand, $f(T^{\alpha}_{\neg u})$ and $g(T^{\alpha}_{\neg u})$ monotonically increase as the search is performed. On the other hand, $g^{T^{\alpha}_{\neg u}}(s)$ does decrease as the search advances: first it takes the value ∞ until $\alpha(s)$ is expanded and then it takes the value of $g^*_{bw}(\alpha(s))$. However, just before the expansion of $\alpha(s)$, $f(T^{\alpha}_{\neg u}) - g(s)$ cannot be greater than $g^*_{bw}(\alpha(s))$:

$$f(T^{\alpha}_{\neg u}) - g(s) > g^*_{bw}(\alpha(s))$$

$$g^*_{bw}(\alpha(s)) + h_{bw}(\alpha(s)) - g(s) > g^*_{bw}(\alpha(s))$$

$$g^*(s) \leq g(s) < h_{bw}(\alpha(s)) \leq h^*_{bw}(\alpha(s)) \leq g^*(s)$$

Contradiction.

As the heuristic value of the states can only be increased, the only way to change the states selected for expansion is to increase the heuristic value of the currently selected states in S. Suppose that the heuristic value of a state $s \in S$ is raised by advancing a non-useful search. According to Definition 9.1 $h(s) = \min\left\{\max\left\{f(T^{\alpha}) - g(s), g((T^{\alpha}) + 1\right\}, g^T(s)\right\}$

Thus, to raise $h(s)$, we have to raise the minimum between $g^T(s)$ and either $f(T^{\alpha})$ or $g(T^{\alpha})$. As the search is not useful, $\alpha(s)$ has already been expanded, so the value of $g^T(s)$ is fixed. The only way to increase $h(s)$ is to increase $f(T^{\alpha})$ or $g(T^{\alpha})$.

Therefore, $h(s)$ can only be increased whenever $\alpha(s)$ has not been expanded yet or $\max\left\{f(T^{\alpha}) - g(s), g(T^{\alpha})\right\} < g^T(s)$. The latter condition can be simplified because if $\alpha(s)$ was expanded, $g(T^{\alpha}) > g^T(s)$. □

Thus, in order to prove optimality, it is enough to expand abstract searches until every abstract state has an f-value greater or equal than the optimal solution cost. All the states whose abstract counterparts have not been expanded do not need to be explored because their f-value is not optimal.

9.6 Empirical Evaluation

In this section we evaluate the SymBA* algorithm and the impact that different abstraction hierarchies have in the overall performance of the algorithm. As a baseline we take the implementation of symbolic bidirectional uniform-cost search. All the symbolic searches use the enhancements for image computation and invalid-state pruning presented in Part I of this thesis.

9.6.1 SymBA* Configuration

SymBA* performs a blind search until both search frontiers surpass the default limit of 10,000,000 nodes or 45 seconds for image computation. At that point, the algorithm initiates a bidirectional perimeter abstraction heuristic to inform the search. We use the same abstraction hierarchies than in Chapter 8, described in detail in Section 8.7.1 on page 144.

The algorithm relies on the assumption that the effort for images computation is proportional to the frontier BDD size. This assumption is already made by the bidirectional blind search in order to decide which search direction is preferred and is well supported by empirical data (see Figure 5.1 in page 84). In this algorithm, image computation is truncated after 60 seconds. In case that image computation gets truncated or takes more than 45 seconds, the maximum number of nodes, N_F, which is initially set to 10,000,000, is updated to the current frontier linearly scaled by the difference in time. Therefore, the maximum number of nodes is a good bound not only of the memory spent by the algorithm but also for the time needed to perform the next step.

The search frontier is relaxed according to each abstraction in the hierarchy until the size of the search frontier is "small enough" for the algorithm to estimate that performing some steps in the abstract state space is feasible. A new parameter controls the maximum size that the frontier is allowed to have in order to start the search in the abstract state space. The ratio after relax is $r_\alpha = 0.8$, so that the new frontier must have at most $r_\alpha \cdot N_F$ nodes.

One common problem of abstraction heuristics is that they have expensive precomputation phases. Usually, this precomputation is allotted a fixed amount of memory and time in order to ensure that the search is initiated. Even though SymBA* does not perform the abstract state space search at the beginning, we impose several conditions under which the planner completely stops using abstractions in order to continue the search in an attempt to leverage the heuristic that has been generated. No more abstract searches are selected when:

1. A limit of $1{,}500$ seconds is surpassed.
2. The algorithm has used more than 3GB of memory.
3. 500 seconds have been spent in selecting the abstraction, i. e., generating of the abstraction and relaxing the frontier.

Also, the planner might fail the attempt to abstract the frontier, in case that the operation exceeds $t_{relax} = 10$ seconds or $N_{relax} = 10{,}000{,}000$ nodes.

9.6.2 Coverage of SymBA*

In order to evaluate the SymBA* algorithm, we compare its performance against symbolic uniform-cost bidirectional search (sym_{bd}). A careful reader may notice that the results of sym_{bd} do not match the ones that were presented in Chapter 4. This is no strange since, as detailed in Section 1.5 on page 10, we are using a different hardware to run the experiments as well as another implementation of the algorithm, integrated in FAST DOWNWARD instead of CGAMER. More importantly, all the experiments reported here are based in the same implementation for a comparison as fair as possible. We evaluate the performance of SymBA* with four different abstraction hierarchies: two PDB strategies and two SM&S with different shrinking. A detailed description can be found in Section 8.7.1 on page 144. All the configurations are ran under two different variable orderings: FAST DOWNWARD and GAMER.

The results in Table 9.1 show that all configurations are close in terms of coverage, which is not surprising given that all of them start running bidirectional blind search with a limited amount of resources. The use of abstraction heuristics do not completely pays off in terms of total coverage, with the bidirectional blind search with GAMER ordering having the overall best coverage. However, it is remarkable that abstraction heuristics help to improve bidirectional blind search in 12 different

	GAMER ORDERING					FD ORDERING				
	∅	*PDB*		SM&S		∅	*PDB*		SM&S	
		rev	*lev*	bop 10k	gop 10k		*rev*	*lev*	bop 10k	gop 10k
AIRPORT (50)	**27**	**27**	**27**	**27**	**27**	22	22	22	22	22
BARMAN (20)	**11**	9	9	9	9	8	8	8	8	8
BLOCKSWORLD (35)	31	**33**	31	31	31	30	32	31	30	30
DEPOT (22)	7	**8**	7	7	7	7	7	7	7	7
DRIVERLOG (20)	12	13	12	13	13	13	**14**	13	**14**	**14**
ELEVATORS08 (30)	**25**	**25**	**25**	**25**	**25**	24	**25**	23	24	24
ELEVATORS11 (20)	19	19	19	19	19	19	19	19	19	19
FLOORTILE11 (20)	14	14	14	14	14	14	14	14	14	14
FREECELL (80)	23	24	**26**	22	22	21	**26**	23	21	20
GRID (5)	**3**	**3**	**3**	**3**	**3**	2	2	2	2	2
GRIPPER (20)	20	20	20	20	20	20	20	20	20	20
LOGISTICS 00 (28)	20	20	20	19	20	**22**	20	21	**22**	21
LOGISTICS 98 (35)	5	5	**6**	5	5	5	5	**6**	5	5
MICONIC (150)	111	107	107	108	109	**122**	120	121	**122**	120
MPRIME (35)	24	23	24	23	23	24	25	**26**	25	25
MYSTERY (30)	15	15	15	15	15	15	15	15	15	15
NOMYSTERY11 (20)	14	15	**16**	15	14	14	**16**	14	**16**	15
OPENSTACKS08 (30)	**30**	**30**	**30**	**30**	**30**	26	26	27	26	26
OPENSTACKS11 (20)	**20**	**20**	**20**	**20**	**20**	18	18	18	18	18
OPENSTACKS06 (30)	**20**	**20**	**20**	**20**	**20**	11	9	9	9	9
PARCPRINTER08 (30)	21	21	**22**	**22**	21	21	21	21	21	21
PARCPRINTER11 (20)	16	16	**17**	**17**	**17**	16	16	16	16	16
PARKING11 (20)	1	1	1	1	1	1	1	1	1	1
PATHWAYS-NONEG (30)	5	5	5	5	5	5	5	5	5	5
PEG-SOLITAIRE08 (30)	29	29	29	29	29	29	29	29	29	29
PEG-SOLITAIRE11 (20)	19	19	19	19	19	19	19	19	19	19
PIPESWORLD-NT (50)	**15**	**15**	**15**	**15**	**15**	13	13	13	14	14
PIPESWORLD-T (50)	**16**	**16**	**16**	**16**	**16**	15	15	15	15	15
PSR-SMALL (50)	50	50	50	50	50	50	50	50	50	50
ROVERS (40)	**14**	**14**	13	**14**	**14**	11	12	12	11	11
SATELLITE (36)	**9**	**9**	**9**	**9**	**9**	7	7	7	7	7
SCANALYZER08 (30)	12	12	12	12	12	12	12	**13**	12	**13**
SCANALYZER11 (20)	9	9	9	9	9	9	9	**10**	9	**10**
SOKOBAN08 (30)	28	28	28	28	28	28	28	28	28	28
SOKOBAN11 (20)	20	20	20	20	20	20	20	20	20	20
TIDYBOT11 (20)	15	13	13	**17**	**17**	12	9	9	9	9
TPP (30)	9	8	8	8	8	**11**	**11**	**11**	**11**	**11**
TRANSPORT08 (30)	**14**	13	13	13	13	**14**	**14**	13	**14**	**14**
TRANSPORT11 (20)	**10**	9	9	9	9	**10**	9	9	**10**	**10**
TRUCKS (30)	**12**	**12**	**12**	**12**	**12**	10	11	10	10	10
VISITALL (20)	12	12	12	12	12	12	12	12	12	12
WOODWORKING08 (30)	26	27	**28**	**28**	**28**	19	16	16	17	17
WOODWORKING11 (20)	19	**20**	**20**	**20**	19	13	10	10	11	11
ZENOTRAVEL (20)	10	**12**	11	10	11	9	10	11	9	11
TOTAL COV (1396)	**842**	840	**842**	840	840	803	802	799	799	798
SCORE COV (36)	**20.71**	**20.75**	**20.71**	**20.75**	**20.76**	19.31	19.34	19.21	19.23	19.25

Table 9.1: Coverage of SymBA* in combination with different abstraction heuristics.

domains with GAMER ordering and 14 domains with FAST DOWNWARD ordering, being necessary to obtain the best results in 15 different domains. The main conclusion is that, as already observed in the SPM&S heuristic presented in Chapter 8, none of the abstraction strategies is capable of selecting good abstractions in a domain-independent way.

9.6.3 Abstraction Hierarchies

The results from Table 9.1 suggest that different abstraction hierarchies obtain good results in different cases. In order to evaluate the impact of abstraction hierarchies in the overall performance, we used the different strategies to generate PDB and SM&S hierarchies that we already presented in Chapter 8 (see page 144). In order to evaluate the impact of perimeter and bidirectional abstractions we perform an ablation analysis, where we disable those features one at a time. Table 9.2 shows the result of the full SymBA* algorithm against a version using non-perimeter abstractions ($\neg\mathcal{P}$) and another one using only backward search in the abstract state spaces (rbw).

Moreover, we include some hypothetical configurations that combine the best results of several configurations, considering a problem solved if any of the individual planners solved it. $best$-PDB combines all the PDB based planners. $best$-abs considers the same planners as $best$-PDB plus the configurations using SM&S hierarchies. Finally, $best$ includes the configuration without any heuristic as well. While the results of this configurations is not "real", they provide an optimistic estimation of the maximum capabilities of SymBA* when a good strategy for the problem at hand is selected.

	GAMER ORDERING			FD ORDERING		
	full	$\neg\mathcal{P}$	rbw	*full*	$\neg\mathcal{P}$	rbw
SP	20.71 (842)	20.71 (842)	20.71 (842)	19.31 (803)	19.31 (803)	19.31 (803)
PDB-rev	20.75 (840)	20.54 (833)	20.80 (846)	19.34 (802)	19.16 (797)	19.52 (809)
PDB-lev	20.71 (842)	20.72 (839)	20.64 (838)	19.21 (799)	19.35 (801)	19.39 (804)
PDB-rnd	20.91 (844)	20.67 (836)	20.72 (840)	19.08 (791)	19.36 (794)	19.58 (804)
PDB-cggoal-lev	20.61 (837)	20.58 (835)	20.66 (838)	19.32 (802)	19.07 (795)	19.36 (802)
PDB-cggoal-rnd	20.68 (840)	20.65 (838)	20.50 (838)	19.29 (798)	19.01 (790)	19.39 (801)
PDB-goalcg-lev	20.82 (843)	20.82 (844)	20.65 (837)	19.32 (803)	19.35 (801)	19.55 (810)
SM&S$_{bop10k}$	20.75 (840)	20.81 (840)	20.83 (839)	19.23 (799)	19.34 (801)	19.55 (808)
SM&S$_{gop10k}$	20.76 (840)	20.78 (842)	20.88 (842)	19.25 (798)	19.37 (801)	19.39 (806)
SM&S$_{fh10k}$	20.51 (837)	20.69 (836)	20.63 (837)	19.20 (792)	19.28 (795)	19.40 (802)
SM&S + PDBs	20.82 (838)	20.71 (839)	20.71 (835)	19.48 (801)	19.46 (796)	19.40 (805)
$best$-PDB	21.15 (855)	21.14 (854)	21.14 (856)	19.84 (822)	19.85 (816)	20.10 (825)
$best$-abs	**21.41** (864)	**21.43** (863)	**21.43** (866)	20.06 (828)	20.20 (826)	20.31 (831)
$best$	**21.62 (873)**	**21.62** (870)	**21.45** (868)	20.33 (837)	20.47 (835)	20.34 (832)

Table 9.2: Coverage of SymBA* in combination with different abstraction heuristics. SM&S + PDBs uses a combination of different strategies. The $best$ configurations are not real planners, but a selection of the best results obtained with any abstraction hierarchy for each problem. Three different configurations: full (*full*), disabling perimeter abstractions ($\neg\mathcal{P}$) and disabling bidirectional abstractions (rbw)

The results of Table 9.2 suggest that abstraction heuristics have limited impact in symbolic bidirectional search. There are no large differences between the different abstraction hierarchies and most of them obtain a coverage similar to the uniform-cost approach, SP. However, this can be attributed to the fact that the abstraction strategies do not reliably select useful abstractions. Indeed, the *PDB-rnd* strategy that abstracts variables at random is mostly indistinguishable from other strategies. The results of $best$ suggest that the SymBA* algorithm has potential to combine abstraction heuristics and symbolic bidirectional search if good abstraction strategies are used. However, the comparison of the full algorithm against the versions disabling perimeter and bidirectional abstractions suggests that none of this features is needed, at least with the abstraction heuristics being used.

The SM&S + PDBs configuration uses a combination of SM&S$_{bop10k}$, *PDB-cggoal-rnd*, *PDB-goalcg-lev*, and *PDB-rev*, trying to get closer to the $best$ configuration. Whenever one of the abstraction strategies does not generate useful abstractions it attempts a different abstraction strategy. However, the overhead is not negligible and in the end it has the same performance than other configurations. Thus, for SymBA* to be a useful algorithm, new domain-independent abstraction strategies are required.

9.7 Summary

Symbolic bidirectional uniform-cost search is a state-of-the-art method for cost-optimal planning. The question remains whether is it possible to use heuristic estimates to further improve its results. However, introducing heuristics in a bidirectional setting is not a simple task. Not in vain, multiple bidirectional heuristic search algorithms have been proposed in the past failing to outperform good-old A* search.

In this chapter we have introduced a new algorithm, SymBA*, that uses abstraction heuristics to inform a bidirectional heuristic search. SymBA* takes advantage of symbolic bidirectional uniform-cost search by deferring the use of heuristics until a blind search seems unfeasible. However, instead of discarding the previous search, the current frontier is used as input to construct perimeter abstraction heuristics, similar to the ones used in Chapter 8.

In order to generate heuristics for both search frontiers, a bidirectional search is carried out in an abstract state space. This requires extending the definition of partial and perimeter abstraction heuristics to the bidirectional case, in order to produce more informed estimates while ensuring optimality of the solutions.

Our experimental results show that this is a promising idea, though finding the right abstraction strategies in a domain-independent way is not a trivial task.

Part III

Conclusions and Future Work

Chapter 10

Experimental Analysis

In this chapter, we summarize the experimental results of this thesis, comparing the overall performance of the symbolic search planners we developed with other state-of-the-art optimal planners. While in other chapters the empirical evaluation was focused in the comparison of the proposed techniques against the closest baseline approach in each case, the goal of this chapter is to describe the big picture: how the improvements made in this thesis affect to the state of the art in optimal planning. We conclude the chapter with a brief summary of the IPC-14, where our planners were prominent participants.

10.1 Summary Evaluation

In this section, we perform a final comparison between the different techniques presented in this thesis and some relevant competitors. Table 10.1 shows the coverage results of the following configurations:

- FAST DOWNWARD A*: explicit-state search using different state-of-the-art heuristics, including LM-CUT and M&S. FAST DOWNWARD STONE SOUP, the winner of IPC-2011, was a portfolio running these three planners by a fixed amount of time. In order to compare against the best possible results obtained by such portfolio, we compare against the maximum possible coverage where we consider a problem solved if it was solved by any of the individual planners. Thus, *best* is an optimistic approximation of the best possible results using the three planners.
- GAMER: symbolic search planner using bidirectional uniform-cost search and BDDA* with symbolic PDBs. This is our symbolic planner baseline to compare against our symbolic search improvements of Part I of this thesis.
- cGAMER: Version of GAMER with the improvements presented in Part I of this thesis.
- BDDA* with M&S: Symbolic A* search with the M&S heuristic. The symbolic representation of M&S abstractions with linear merge strategies enables an efficient evaluation for symbolic search, as shown in Chapter 7.
- A*-SPM&S: heuristic search planner using our SPM&S heuristic presented in Chapter 8. This heuristic is based on a symbolic perimeter and perimeter abstraction heuristics: PDBs

and M&S. We ran different configurations. SP uses only a symbolic perimeter around the goal. The other configurations extend the perimeter using different abstraction hierarchies, including PDBs, SM&S, and multiple PDB strategies (SPM&S-k) with or without SM&S.

- SymBA*: Symbolic bidirectional A* search. The *bd* configuration is similar to CGAMER-BD though there are some differences in the implementation details. The version with SPPDB and SM&S uses the bidirectional abstraction heuristics described in Chapter 9.

All planners use the preprocessor described in Section 4.1.3 on page 56 to remove invalid operators and simplify the task. For other details about configuration parameters, check the default parameters used in the empirical evaluation of the chapter where each technique was presented.

Table 10.1 shows the coverage of all approaches, summarizing the empirical contributions of this thesis. In Part I of this thesis, we proposed some improvements for the symbolic search planner GAMER and we called the resulting configuration CGAMER. CGAMER does not only consistently beat GAMER, but also is a state-of-the-art planner that outperforms even heuristic portfolio approaches, as studied in Section 5.2 on page 84. The comparison in this case is less favorable to CGAMER mainly due to hardware settings. The experiments in Part I were ran in a cluster using a single processor per machine, while the experiments of the rest of the thesis were ran in parallel, using 16 processors per machine. Executing several processes in the same computer can decrease the performance because the increased time in memory accesses. Since the BDD manipulation in symbolic search requires an extensive number of memory accesses, it is not surprising that the performance of symbolic search planners is significantly reduced with respect to heuristic search planners.

In Chapter 7 we considered the use of M&S heuristics in BDDA*. While it outperforms explicit-state search with the same heuristics, it is not a competitive approach in general. Nevertheless, it was a step towards the development of new symbolic heuristics. In Chapter 8 we proposed the SPM&S heuristic, which combined M&S abstractions and symbolic perimeter search. The SPM&S heuristic does not only beat the simple perimeter (SP) and M&S heuristics, but also is a highly competitive heuristic outperforming even the state-of-the-art heuristic LM-CUT. Results are even better when multiple abstraction hierarchies are used, taking the maximum estimation from each of their results (SPM&S-k).

Finally, in Chapter 9 we developed the SymBA* algorithm that aims to increase the performance of symbolic bidirectional uniform-cost search with perimeter abstraction heuristics. The different configurations of SymBA* obtain quite similar results to the bidirectional blind search. Nevertheless, of all the planners considered in this thesis, SymBA* has the best overall coverage score.

Figure 10.1 shows the cumulative coverage of several planners over time. As expected, SP and SPPDB are closely related. Both planners start constructing a symbolic perimeter, so their performance is indistinguishable during the first 100 seconds. Afterwards, SP starts the search and starts solving more problems. SPPDB continues precomputing perimeter abstraction heuristics and starts the A* search later, but in the end outperforms SP. A similar observation can be made in the case of SPPDBmulti, which performs an even longer precomputation phase to obtain better results at the end.

As we analyzed in Chapter 8, using a symbolic perimeter as heuristic is competitive against state-of-the-art heuristics such as LM-CUT. Even though SP solves less problems than LM-CUT at the beginning, due to the precomputation phase of the heuristic, the performance after 1800 seconds is the same and even better when the perimeter is extended with abstraction heuristics (SPPDB).

Regarding bidirectional search, as we already analyzed in more detail in Chapter 5 the improvements we made in Part I of the thesis to the bidirectional uniform-cost search of GAMER, help to

	A*				Gamer		cGamer		BDDA* + M&S			A* + SPM&S					SymBA*		
	M&S		LM-	*best*	*bd*	BDDA*	*bd* (*)	BDDA*	Gamer ordering	FD ordering		SP	SPPDB	SM&S	SPM&S-k		*bd*	SPPDB (*)	SM&S (*)
	bop	gop	cut						bop	bop	gop			bop	bop + PDB (*)	PDB			
Airport (50)	21	23	**29**	**29**	23	22	25	20	22	17	22	25	27	25	26	25	27	27	27
Barman (20)	4	4	4	4	6	4	8	4	4	4	4	8	8	8	9	9	**11**	10	**11**
Blocksworld (35)	20	28	28	28	21	27	31	25	25	22	28	30	32	23	31	**33**	31	30	30
Depot (22)	6	6	7	7	5	6	5	**8**	7	6	6	7	**8**	3	7	7	7	7	7
Driverlog (20)	13	12	13	13	12	**14**	**14**	**14**	13	13	**14**	12	13	13	**14**	13	12	**14**	**14**
Elevators08 (30)	14	1	22	22	24	20	**25**	22	20	19	1	23	23	23	24	24	**25**	24	24
Elevators11 (20)	12	0	18	18	**19**	17	**19**	16	17	16	0	18	18	18	**19**	**19**	**19**	**19**	**19**
Floortile11 (20)	8	4	**14**	**14**	9	12	**14**	**14**	**14**	**14**	4	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**
Freecell (80)	4	19	15	19	14	17	20	23	15	4	20	22	31	17	34	**37**	23	24	23
Grid (5)	**3**	2	2	**3**	2	2	2	2	**3**	**3**	2	2	**3**	**3**	**3**	**3**	**3**	2	**3**
Gripper (20)	**20**	8	7	**20**	**20**	**20**	**20**	17	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Logistics 00 (28)	20	16	20	20	18	18	20	**22**	21	**22**	21	16	17	16	19	19	20	20	20
Logistics 98 (35)	5	4	**6**	**6**	5	5	5	**6**	5	5	5	4	5	5	5	5	5	5	5
Miconic (150)	77	54	**141**	**141**	84	0	110	106	79	88	88	107	107	107	108	108	111	108	107
MPrime (35)	12	23	23	24	22	24	22	26	20	12	**27**	22	22	19	23	23	24	25	25
Mystery (30)	8	**17**	**17**	**17**	14	16	14	15	12	8	16	15	15	15	15	15	15	15	15
NoMystery11 (20)	19	14	14	19	13	14	16	13	**20**	**20**	16	13	16	16	17	18	14	14	14
Openstacks08 (30)	21	9	21	21	**30**	28	**30**	24	**30**	23	9	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
Openstacks11 (20)	16	4	16	16	**20**	18	**20**	18	**20**	18	4	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Openstacks06 (30)	7	7	7	7	10	7	**20**	12	16	7	7	11	10	11	10	10	**20**	**20**	**20**
ParcPrinter08 (30)	20	20	22	**23**	19	13	22	18	16	16	20	**23**	**23**	**23**	**23**	**23**	21	21	21
ParcPrinter11 (20)	15	15	17	**18**	15	9	17	14	12	11	15	**18**	**18**	**18**	**18**	**18**	16	16	16
Parking11 (20)	0	0	2	2	0	1	1	1	1	0	0	1	**4**	0	**4**	**4**	1	1	1
Pathways-noneg (30)	4	4	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	4	4	4	4	**5**	**5**	**5**
Peg-Solitaire08 (30)	29	6	28	29	29	28	**30**	28	29	29	6	29	29	29	**30**	**30**	29	29	29
Peg-Solitaire11 (20)	19	0	18	19	19	18	**20**	18	19	19	0	19	19	19	**20**	**20**	19	19	19
Pipesworld-nt (50)	9	16	**17**	**17**	14	12	15	14	15	9	14	15	15	15	14	14	15	15	15
Pipesworld-t (50)	8	17	12	17	13	14	16	**19**	15	8	15	12	16	8	12	15	16	16	16
PSR-small (50)	**50**	**50**	49	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**
Rovers (40)	8	6	7	8	**14**	12	**14**	13	11	9	10	12	13	12	13	12	**14**	13	13
Satellite (36)	7	6	7	7	7	7	**9**	**9**	**9**	7	7	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**
Scanalyzer08 (30)	14	9	15	**17**	11	11	12	12	12	13	9	12	12	12	12	12	12	12	12
Scanalyzer11 (20)	11	6	12	**14**	8	8	9	9	9	10	6	9	9	9	9	9	9	9	9
Sokoban08 (30)	29	3	**30**	**30**	27	29	28	28	28	28	3	28	29	25	28	28	28	28	28
Sokoban11 (20)	**20**	1	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	1	**20**	**20**	19	**20**	**20**	**20**	**20**	**20**
Tidybot11 (20)	1	13	**17**	**17**	11	10	10	**17**	9	1	13	14	12	10	15	11	15	12	**17**
TPP (30)	7	6	7	7	8	8	8	8	8	8	**9**	8	8	8	8	8	**9**	8	**9**
Transport08 (30)	11	11	11	11	11	11	13	11	11	11	11	12	12	12	13	12	**14**	**14**	13
Transport11 (20)	7	6	6	7	7	6	9	6	6	7	6	7	8	8	9	9	**10**	9	9
Trucks (30)	8	8	10	10	9	10	12	12	10	10	10	10	10	10	11	**13**	12	12	12
VisitAll (20)	9	**16**	10	**16**	12	10	12	10	11	12	13	11	12	11	12	12	12	12	12
Woodworking08 (30)	13	14	22	22	22	22	23	26	22	17	22	**28**	25	**28**	25	25	26	25	25
Woodworking11 (20)	8	9	15	15	16	15	17	15	16	12	16	**20**	**20**	**20**	19	19	19	19	19
Zenotravel (20)	12	10	**13**	**13**	10	11	11	11	12	12	11	10	12	11	12	12	10	11	11
Total Cov (1396)	629	507	796	**842**	728	631	823	781	739	665	586	800	828	776	838	841	**842**	833	838
Score Cov (36)	15.82	13.66	18.56	20.35	17.86	17.21	19.99	19.12	18.83	16.73	15.35	19.32	20.28	18.87	20.59	20.56	**20.71**	20.32	**20.82**

Table 10.1: Summary of experimental results of the thesis. Planners marked with (*) denote that they participated in IPC-2014, as mentioned in Section 10.2.

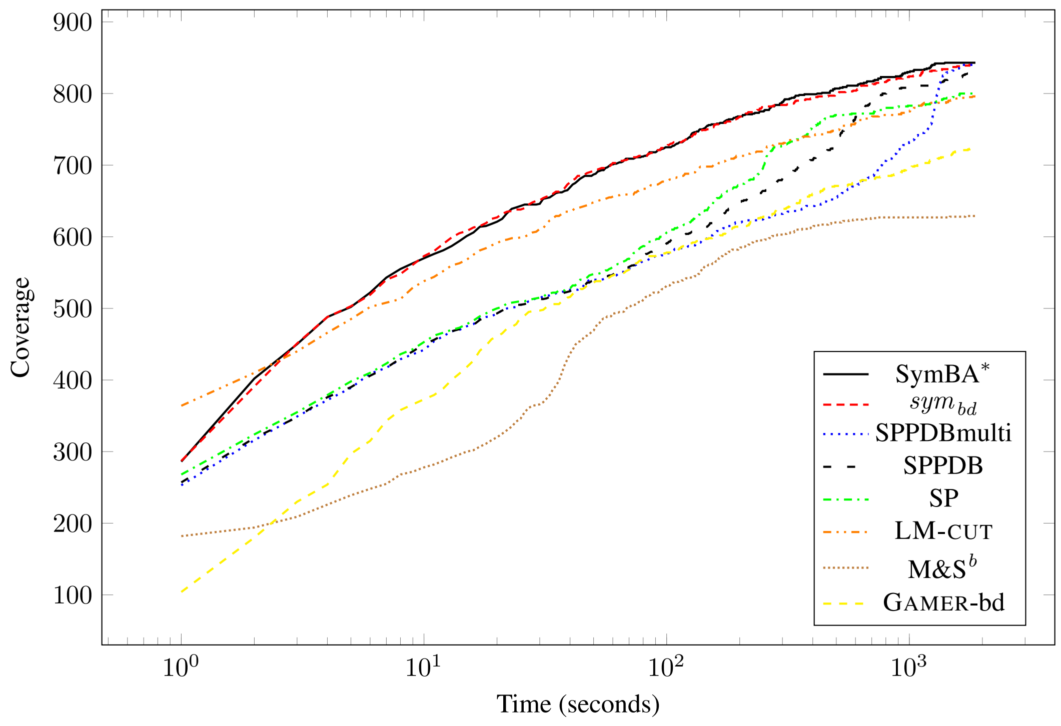


Figure 10.1: Cumulative coverage of planners. Total coverage of each planner at each time in logscale.

make a stronger planner whose performance is remarkable in our set of benchmarks. After few seconds, symbolic bidirectional search outperforms the other competitors. The SymBA* planner introduces heuristics but, as analyzed in Chapter 9, its performance is close to sym_{bd}. Abstraction heuristics are useful in some domains but the overhead reduces the performance in others and, in total, both algorithms tie in total coverage.

10.2 The 2014 International Planning Competition

We report the results of IPC-14, where our planners competed against other state-of-the-art planners. IPC results are not completely adequate for the evaluation of techniques, since results can be biased due to bugs and execution errors present in most planners. Nevertheless, some trends may be observed when comparing the results of different planners.

We submitted three different planners to the competition:

- cGAMER-BD (Torralba et al., 2014b): Our planner using symbolic bidirectional uniform-cost search with the enhancements of Part I of this thesis.

- SPM&S (Torralba et al., 2014c): Uses A* with the symbolic perimeter M&S heuristic that we presented in Chapter 8 of this thesis. It uses multiple SM&S and PDB abstraction hierarchies.

- SymBA* (Torralba et al., 2014a): The symbolic bidirectional search informed with bidirectional abstraction heuristics proposed in Chapter 9. We submitted two different parameter configurations that differ on the abstraction hierarchies used: SYMBA-1and SYMBA-2. They both use PDB abstractions, based on the variable orderings *cgr*, *gcl* and *rev*. SYMBA-2, additionally, uses SM&S abstraction hierarchies using bisimulation shrinking with a maximum number of 10,000 abstract states.

All the planners use the improvements on image computation from Chapter 3 and h^2 constraints for symbolic search described in Chapter 4. These planners correspond to the configurations included in Table 10.1 that are marked with (*). Some differences can be found due to some bugfixes or small optimizations. Also, our planners were adapted for the IPC to support conditional effects, who were not considered in the empirical evaluation of this thesis.

Other participants in the IPC were:

- GAMER and DYNAMIC-GAMER (Kissmann et al., 2014): The vanilla version of GAMER uses symbolic bidirectional uniform-cost search without the enhancements proposed in this thesis. DYNAMIC-GAMER is an extended version using dynamic variable reordering (Kissmann and Hoffmann, 2014), an enhancement of symbolic search that is orthogonal to the work in this thesis.

- NUCELAR (Núñez et al., 2014b), MIPLAN, DPMPLAN (Núñez et al., 2014a), CEDALION (Seipp et al., 2014), ALLPACA (Malitsky et al., 2014) are portfolio approaches that use several planners. MIPLAN, DPMPLAN and CEDALION are sequential static portfolios that execute a given set of planners for a fixed amount of time. The former two use a mixed-integer programming approach to divide the available time among a set of candidate planners (Núñez et al., 2012). The latter focuses on optimizing the parameter configuration of the FAST DOWNWARD planner for different domains to construct the set of candidate planners. NUCELAR and ALLPACA decide which planner/s to use for each instance using a machine learning approach, with features automatically extracted from the problem description (Cenamor et al., 2013). For more details about the planners considered by the portfolios and the methods to decide which planners execute, we refer the reader to the original articles.

- RIDA (Franco et al., 2014), RLAZYA* (Karpas et al., 2014): Combine different heuristics during a single search process, deciding which heuristics to use in a state-per-state basis (Barley et al., 2014; Tolpin et al., 2013).

- METIS (Alkhazraji et al., 2014): Combine the incremental version of the LM-CUT heuristic (Pommerening and Helmert, 2013) with symmetry and partial order reduction pruning (Domshlak et al., 2012; Wehrle and Helmert, 2012).

- HFLOW (Bonet and van den Briel, 2014b) uses the flow heuristics.

- h^{++} (Haslum, 2014): An incremental lower bound procedure that iteratively introduces constraints into the delete-relaxation heuristic until the relaxed plan is executable in the original problem.

The IPC-2014 featured 14 different domains, with 20 problem instances each though some were unsolvable. The domains selected for the IPC can be classified in several categories:

- Domains from previous IPCs. Including BARMAN, FLOORTILE, OPENSTACKS, TIDYBOT, TRANSPORT and VISITALL. All this domains have been used in the empirical evaluation thorough this thesis, though new problem instances were used in IPC-14.

	BARMAN	CAVE	CHILDSNACK	CITYCAR	FLOORTILE	GED	HIKING	MAINTENANCE	OPENSTACKS	PARKING	TETRIS	TIDYBOT	TRANSPORT	VISITALL	TOTAL
SYMBA-2	**6**	3	4	**18**	**20**	**20**	**20**	4	**20**	0	10	10	**9**	7	**151**
SYMBA-1	**6**	3	4	**18**	**20**	19	**20**	4	**20**	0	10	4	**9**	6	143
CGAMER-BD	**6**	0	1	**18**	**20**	0	15	0	19	3	**11**	**13**	8	6	120
SPMAS	5	3	2	1	**20**	18	12	4	14	4	7	8	**9**	7	114
RIDA	0	3	0	16	5	19	17	**5**	3	**6**	8	8	8	**15**	113
DYNAM-GAMER	3	3	**10**	15	14	0	17	3	19	0	2	0	7	6	99
ALL-PACA	0	**7**	0	17	6	15	13	**5**	8	**6**	3	1	5	12	98
CEDALION	0	**7**	0	14	5	15	13	**5**	1	2	5	7	6	13	93
METIS	3	**7**	6	0	8	15	13	**5**	3	4	8	7	6	6	91
NUCELAR	0	**7**	0	13	0	15	13	**5**	3	5	9	0	7	13	90
RLAZYA	0	**7**	0	17	5	15	9	**5**	2	4	6	7	6	5	88
GAMER	3	3	2	**18**	13	0	14	0	16	0	3	0	6	5	83
HFLOW	0	3	0	0	3	7	4	**5**	1	0	10	0	5	**15**	53
MIPLAN	0	**7**	0	11	0	0	10	**5**	0	1	0	0	0	13	47
DPMPLAN	0	**7**	0	8	0	0	0	**5**	0	5	0	0	6	12	43
HPP-CE	0	0	0	7	0	3	0	**5**	0	0	0	0	0	0	15
HPP	0	0	0	6	0	3	0	**5**	0	0	0	0	0	0	14

Table 10.2: Results of the 2014th International Planning Competition. The results of our planners are highlighted in gray.

- New domains without conditional effects. Including CHILDSNACK, GED, HIKING and TETRIS.
- New domains with conditional effects. Including CAVE, CITYCAR and MAINTENANCE.

Table 10.2 shows the results of IPC-2014, with planners sorted according to their total coverage. The results of our four submissions were remarkable, getting the first places of the competition. SymBA* was the IPC winner, solving more problems than other participants and being the best planner in 7 out of 14 domains.

CGAMER-BD was the runner up of the competition. A direct comparison of SymBA* and CGAMER-BD shows advantages for the former, contradicting the results from Section 9.6 on page 167. However, the advantage of SymBA* over the symbolic bidirectional uniform-cost search used by CGAMER-BD is not only due to the use of abstraction heuristics. CGAMER-BD PDDL parser and conditional-effect support had some errors that caused the planner to fail in three domains: MAINTENANCE, GED and CAVE. Ignoring those domains, CGAMER-BD performance is similar to the versions of SymBA*.

Our last planner, SPM&S finished in third position, not only on top of other heuristic search planners, but also outperforming portfolios that use multiple heuristics. This remarks the potential of symbolic search, and symbolic regression in particular, for deriving admissible heuristics.

The other symbolic planners in the competition also obtained good results. GAMER, the baseline approach that we have compared against in this thesis, solved 83 problems. DYNAMIC-GAMER was ranked the fifth position being the best planner in CHILDSNACK.

Regarding the comparison of symbolic versus explicit search, symbolic planners outperformed explicit-search planners in 10 out of 14 domains. The results in domains from previous competitions are not very surprising. Symbolic search planners are better in BARMAN, FLOORTILE, OPEN-

STACKS, TIDYBOT and TRANSPORT and explicit-search planners in PARKING and VISITALL. In the new domains, we see a similar trend with symbolic planners being better in 5 out of 7 domains.

10.3 Summary

In this chapter we have highlighted the performance of the planners we developed in this thesis. We performed an empirical comparison of all the approaches we considered and highlighted our contributions to the state of the art in cost-optimal planning and, in particular, to symbolic planning.

The results of the 2014 International Planning Competition confirm what we showed in our evaluation: symbolic bidirectional search performs incredibly well, beating all other planners in around half of the domains.

Chapter 11

Conclusions

In this chapter, we summarize the contributions of the dissertation, discuss the main conclusions we drew and possible future work.

11.1 Contributions

The main contributions of this thesis are:

Advances on symbolic search planning The use of symbolic search planning has been a promising avenue for a long time. We analyzed the state-of-the-art symbolic planner, GAMER, and implemented two orthogonal improvements:

1. Analysis of the image computation in symbolic search planning. The image computation, used to perform the successor generation, is probably the main bottleneck in symbolic search. We studied in detail different methods to perform the image computation and empirically compared them. Our new methods increase the performance of GAMER. This work is part of a collaboration with Stefan Edelkamp and Peter Kissmann (Torralba et al., 2013a) and it was presented in Chapter 3.

2. State-invariant constraints for symbolic search planning. State-invariants are properties that hold in every reachable state and it is well-known that they are very useful to prune regression search. However, they were not used by symbolic search state-of-the-art planners. We studied how to efficiently encode state-invariant constraints as BDDs for their use to prune symbolic bidirectional search. This work is part of a collaboration with Vidal Alcázar (Torralba and Alcázar, 2013) and it was presented in Chapter 4.

Both contributions helped to raise the performance of symbolic search planning, making symbolic bidirectional uniform-cost search one of the best state-of-the-art algorithms for cost-optimal planning. This is reflected by the performance of our planner, CGAMER, runner-up in the 2014 edition of the International Planning Competition.

Symbolic abstraction heuristics Abstraction heuristics are a promising avenue to generate domain-independent heuristics. Starting from the more classical Pattern Databases, in the last decade numerous domain-independent methods to generate abstractions have been developed for planning, such as merge-and-shrink or counter-example guided abstractions. While symbolic search had been previously used in combination with PDBs, in this thesis we have further explored the relationship of symbolic search and abstraction heuristics:

1. Symbolic representation of merge-and-shrink heuristics. We theoretically and empirically studied the representation of linear merge-and-shrink heuristics in the form of Algebraic and Binary Decision Diagrams. We proved that the resulting data-structures are guaranteed to have a polynomial size under a certain BDD variable orderings. This enables the use of merge-and-shrink heuristics in symbolic BDDA*. However, our comparison with Symbolic PDBs shows that the restrictions with respect to the variable ordering limit the performance of merge-and-shrink in symbolic search. This work is part of a collaboration with Stefan Edelkamp and Peter Kissmann (Edelkamp et al., 2012) and it was presented in Chapter 7.

2. We presented symbolic perimeter merge-and-shrink, a new heuristic that uses symbolic regression search and merge-and-shrink abstractions in a similar way to perimeter abstraction heuristics. Empirical results show that the resulting heuristic is competitive with state-of-the-art planning. This work was presented in Chapter 4 and (Torralba et al., 2013b).

Bidirectional search with abstraction heuristics The final part of the thesis attempts to join the results of symbolic bidirectional uniform-cost search achieved in Part I of the thesis and the symbolic abstraction heuristics studied in Part II. We present the SymBA* algorithm that uses abstraction heuristics lazily, only computing them when needed. We extend the definition of perimeter and partial abstraction heuristics to a bidirectional heuristic setting and use the resulting estimates in a symbolic bidirectional search. This work was presented in Chapter 9.

Overall results The outcome of the empirical evaluation performed in this thesis is very positive. Our enhancements help symbolic search to consistently outperform explicit-state uniform-cost search, and make symbolic bidirectional uniform-cost search one of the best approaches to cost-optimal planning, as shown in Chapter 5. SPM&S, our heuristic produced as part of the research on the relation of symbolic search and M&S abstractions, is also one state-of-the-art heuristic for cost-optimal planning. Finally, the combination of bidirectional search and perimeter abstraction heuristics is still a promising approach, despite the results show that new abstractions strategies are required for such setting. Our final comparison in Chapter 10 highlights the relevance of the techniques developed during this thesis for the current state of the art.

11.1.1 Publications

List of publications related to this thesis:

- Stefan Edelkamp, Peter Kissmann, and Álvaro Torralba (2012). “Symbolic A* Search with Pattern Databases and the Merge-and-Shrink Abstraction”. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. ed. by Luc De Raedt, Christian Bessière, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 306–311

- Álvaro Torralba, Stefan Edelkamp, and Peter Kissmann (2013a). “Transition Trees for Cost-Optimal Symbolic Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo

Oddi, and Simone Fratini. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 206–214

- Álvaro Torralba, Carlos Linares López, and Daniel Borrajo (2013b). "Symbolic Merge-and-Shrink for Cost-Optimal Planning". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. ed. by Francesca Rossi. IJCAI/AAAI, pp. 2394–2400

- Álvaro Torralba and Vidal Alcázar (2013). "Constrained Symbolic Search: On Mutexes, BDD Minimization and More". In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. ed. by Malte Helmert and Gabriele Röger. AAAI Press, pp. 175–183

- Javier García, José E. Flórez, Álvaro Torralba Arias de Reyna, Daniel Borrajo, Carlos Linares López, Angel García Olaya, and Juan Sáenz (2013). "Combining linear programming and automated planning to solve intermodal transportation problems". In: *European Journal of Operational Research* 227.1, pp. 216–226

11.2 Future Work

In this thesis, we have developed techniques based on symbolic search for cost-optimal planning. Below we list some of the directions in which our work could be further developed.

11.2.1 Strategies for Abstraction Heuristics

In this thesis, we have explored the relation of abstraction heuristics and symbolic search. However, in this thesis, we have limited ourselves to reuse abstraction strategies that were developed in previous works. Both, the pattern selection in PDBs and the merge or shrinking strategies in M&S have a huge influence in the performance of all the techniques explored in the second part of this thesis. However, those strategies were not specifically designed to be used in our setting. An interesting question is whether is it possible to develop strategies better suited for the combination with the symbolic searches. In particular:

- Combined strategies to optimize the abstraction selection and the BDD variable ordering simultaneously.

- Perimeter abstraction strategies that take into account the current perimeter frontier in order to obtain more succinct or informed abstractions.

11.2.2 Other Heuristics in Symbolic Search

From our experiments, it is clear that symbolic search provides significant advantages over explicit-state search. The main limitation of symbolic search is the difficulty of combining it with most domain-independent heuristics. In this thesis, we have restricted our analysis to merge-and-shrink abstractions and, to a lesser degree, pattern databases. However, we were limited to use a few heuristics, combining them by taking the maximum of such heuristics.

The current trend in explicit-search cost-optimal planning goes in a very different direction, though. Most heuristics are based on an additive cost-partitioning among a relatively large amount of heuristics. Recently, some promising approaches based on determining the cost-partitioning in a per-state basis have emerged. This detailed per-state analysis seems a bit contrary to the spirit of symbolic search, based on looking at whole sets of states at once. Thus, despite the efforts that we made in this thesis, the gap between symbolic search and heuristics continues increasing. Understanding that gap and proposing new ways to reduce it seems a very promising line of research for the future of cost-optimal planning.

Bibliography

Vidal Alcázar (2014). "Generation and Exploitation of Intermediate Goals in Automated Planning". PhD thesis. Universidad Carlos III de Madrid (pages 54–56).

Vidal Alcázar, Daniel Borrajo, Susana Fernández, and Raquel Fuentetaja (2013). "Revisiting Regression in Planning". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Francesca Rossi. IJCAI/AAAI, pp. 2254–2260 (pages 9, 53–56, 58, 69, 81).

Vidal Alcázar, Susana Fernández, and Daniel Borrajo (2014). "Analyzing the Impact of Partial States on Duplicate Detection and Collision of Frontiers." In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Steve Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml. AAAI, pp. 350–354 (pages 9, 151).

Vidal Alcázar and Álvaro Torralba (2015). "A Reminder about the Importance of Computing and Exploiting Invariants in Planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press (page 56).

Yusra Alkhazraji, Michael Katz, Robert Matmüller, Florian Pommerening, Alexander Shleyfman, and Martin Wehrle (2014). "Metis: Arming Fast Downward with Pruning and Incremental Computation". In: *International Planning Competition (IPC)*, pp. 88–92 (page 179).

Kenneth Anderson, Robert Holte, and Jonathan Schaeffer (2007). "Partial Pattern Databases". In: *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*. Ed. by Ian Miguel and Wheeler Ruml. Vol. 4612. Lecture Notes in Computer Science. Springer, pp. 20–34 (pages 35, 95, 135).

Fahiem Bacchus and Qiang Yang (1994). "Downward Refinement and the Efficiency of Hierarchical Problem Solving". In: *Artificial Intelligence Journal* 71.1, pp. 43–100 (page 4).

Christer Bäckström and Peter Jonsson (1995). "Planning with Abstraction Hierarchies can be Exponentially Less Efficient". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, pp. 1599–1605 (page 92).

Christer Bäckström and Peter Jonsson (2012). "Abstracting Abstraction in Search with Applications to Planning". In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. Ed. by Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith. AAAI Press, pp. 446–456 (pages 92, 93).

Christer Bäckström and Bernhard Nebel (1995). “Complexity Results for SAS+ Planning”. In: *Computational Intelligence* 11, pp. 625–656 (pages 2, 3).

R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi (1997). “Algebraic Decision Diagrams and Their Applications”. In: *Formal Methods in System Design* 10.2/3, pp. 171–206 (page 22).

Marcel Ball and Robert C. Holte (2008). “The Compression Power of Symbolic Pattern Databases”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 2–11 (pages 35, 92, 95, 104).

Joseph Kelly Barker and Richard E. Korf (2015). “Limitations of Front-To-End Bidirectional Heuristic Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Blai Bonet and Sven Koenig. AAAI Press, pp. 1086–1092 (pages 151, 154).

Michael W. Barley, Santiago Franco, and Patricia J. Riddle (2014). “Overcoming the Utility Problem in Heuristic Generation: Why Time Matters”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Steve Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml. AAAI, pp. 38–46 (page 179).

Christoph Betz and Malte Helmert (2009). “Planning with h^+ in Theory and Practice”. In: *Proceedings of the German Conference on Artificial Intelligence (KI)*. Ed. by Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz. Vol. 5803. Lecture Notes in Computer Science. Springer, pp. 9–16 (page 6).

Beate Bollig (2014). “A simpler counterexample to a long-standing conjecture on the complexity of Bryant’s apply algorithm”. In: *Information Processing Letters* 114.3, pp. 124–129 (page 20).

Beate Bollig and Ingo Wegener (1996). “Improving the Variable Ordering of OBDDs Is NP-Complete”. In: *IEEE Transactions on Computers* 45.9, pp. 993–1002 (page 20).

Blai Bonet (2013). “An Admissible Heuristic for SAS+ Planning Obtained from the State Equation”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Francesca Rossi. IJCAI/AAAI, pp. 2268–2274 (pages 6, 7).

Blai Bonet and Hector Geffner (2001). “Planning as heuristic search”. In: *Artificial Intelligence Journal* 129.1-2, pp. 5–33 (pages 5, 6, 8, 53, 54, 58).

Blai Bonet and Hector Geffner (2008). “Heuristics for planning with penalties and rewards formulated in logic and computed through circuits”. In: *Artificial Intelligence Journal* 172.12-13, pp. 1579–1604 (page 23).

Blai Bonet and Malte Helmert (2010). “Strengthening Landmark Heuristics via Hitting Sets”. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Ed. by Helder Coelho, Rudi Studer, and Michael Wooldridge. Vol. 215. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 329–334 (page 7).

Blai Bonet and Menkes van den Briel (2014a). “Flow-Based Heuristics for Optimal Planning: Landmarks and Merges”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Steve Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml. AAAI, pp. 47–55 (pages 6, 7).

Blai Bonet and Menkes van den Briel (2014b). “Optimal Planning using Flow-Based Heuristics”. In: *International Planning Competition (IPC)*, pp. 85–86 (page 179).

Ronen I. Brafman and Carmel Domshlak (2003). “Structure and Complexity in Planning with Unary Operators”. In: *Journal of Artificial Intelligence Research (JAIR)* 18, pp. 315–349 (page 4).

Randal E. Bryant (1986). “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* 35.8, pp. 677–691 (pages 9, 18–20).

Randal E. Bryant (1992). “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”. In: *ACM Computing Surveys* 24.3, pp. 293–318 (page 20).

Jerry R. Burch, Edmund M. Clarke, and David E. Long (1991a). “Representing Circuits More Efficiently in Symbolic Model Checking”. In: *Design Automation Conference (DAC)*, pp. 403–407 (page 38).

Jerry R. Burch, Edmund M. Clarke, and David E. Long (1991b). “Symbolic Model Checking with Partitioned Transistion Relations”. In: *Proceedings of the International Conference on Very Large Scale Integration (VLSI)*, pp. 49–58 (page 38).

Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill (1994). “Symbolic model checking for sequential circuit verification”. In: *IEEE Transactions on CAD of Integrated Circuits and Systems* 13.4, pp. 401–424 (page 38).

Tom Bylander (1994). “The Computational Complexity of Propositional STRIPS Planning”. In: *Artificial Intelligence Journal* 69.1-2, pp. 165–204 (page 2).

Isabel Cenamor, Tomás De La Rosa, and Fernando Fernández (2013). “Learning predictive models to configure planning portfolios”. In: *Proceedings of the 4th workshop on Planning and Learning (ICAPS-PAL 2013)*, pp. 14–22 (page 179).

Dennis de Champeaux (1983). “Bidirectional Heuristic Search Again”. In: *Journal of the ACM* 30.1, pp. 22–32 (page 154).

Dennis de Champeaux and Lenie Sint (1977). “An Improved Bidirectional Heuristic Search Algorithm”. In: *Journal of the ACM* 24.2, pp. 177–191 (pages 151, 154).

Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, James H. Kukula, Thomas R. Shiple, Helmut Veith, and Dong Wang (2001). “Non-linear Quantification Scheduling in Image Computation”. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE Press. San Jose, California, pp. 293–298 (page 38).

Olivier Coudert and Jean Christophe Madre (1990). “A Unified Framework for the Formal Verification of Sequential Circuits”. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 126–129 (page 66).

Joseph C. Culberson and Jonathan Schaeffer (1998). “Pattern Databases”. In: *Computational Intelligence* 14.3, pp. 318–334 (pages 6, 7, 35, 68, 95).

William Cushing, J. Benton, and Subbarao Kambhampati (2011). “Cost Based Satisficing Search Considered Harmful”. In: *CoRR* abs/1103.3687 (page 11).

Adnan Darwiche (2001). “On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision”. In: *Journal of Applied Non-Classical Logics* 11.1-2, pp. 11–34 (page 23).

Adnan Darwiche (2011). “SDD: A New Canonical Representation of Propositional Knowledge Bases”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Toby Walsh. IJCAI/AAAI, pp. 819–826 (page 24).

Adnan Darwiche and Pierre Marquis (2002). “A Knowledge Compilation Map”. In: *Journal of Artificial Intelligence Research (JAIR)* 17, pp. 229–264 (page 23).

Rina Dechter and Judea Pearl (1985). “Generalized Best-First Search Strategies and the Optimality of A*”. In: *Journal of the ACM* 32.3, pp. 505–536 (pages 6, 31).

Edsger W. Dijkstra (1959). “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1.1, pp. 269–271 (pages 5, 26).

John F. Dillenburg and Peter C. Nelson (1994). “Perimeter Search”. In: *Artificial Intelligence Journal* 65.1, pp. 165–178 (pages 95, 131, 155).

Carmel Domshlak, Jörg Hoffmann, and Ashish Sabharwal (2009). “Friends or Foes? On Planning as Satisfiability and Abstract CNF Encodings”. In: *Journal of Artificial Intelligence Research (JAIR)* 36, pp. 415–469 (page 92).

Carmel Domshlak, Michael Katz, and Alexander Shleyfman (2012). “Enhanced Symmetry Breaking in Cost-Optimal Planning as Forward Search”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet. AAAI, pp. 343–347 (page 179).

Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski (2006). “Directed Model Checking with Distance-Preserving Abstractions”. In: *Proceedings of the Workshop on Model Checking Software (SPIN)*. Ed. by Antti Valmari. Vol. 3925. Lecture Notes in Computer Science. Springer, pp. 19–34 (pages 96, 99).

Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski (2009). “Directed model checking with distance-preserving abstractions”. In: *Journal on Software Tools for Technology Transfer* 11.1, pp. 27–37 (pages 96, 98).

Stefan Edelkamp (2001). “Planning with Pattern Databases”. In: *Proceedings of the JoEuropean Conference on Planning (ECP)*. Ed. by Amedeo Cesta and Daniel Borrajo. Lecture Notes in Computer Science. Volume lost due to September 11th. Springer, pp. 13–34 (pages 7, 9, 95).

Stefan Edelkamp (2002). “Symbolic Pattern Databases in Heuristic Search Planning”. In: *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. Ed. by Malik Ghallab, Joachim Hertzberg, and Paolo Traverso. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 274–283 (pages 9, 35, 68, 92, 126).

Stefan Edelkamp (2005). “External Symbolic Heuristic Search with Pattern Databases”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Susanne Biundo, Karen L. Myers, and Kanna Rajan. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 51–60 (pages 9, 35).

Stefan Edelkamp (2006). “Automated Creation of Pattern Database Search Heuristics”. In: *Proceedings of the Workshop on Model Checking and Artificial Intelligence (MoChArt)*. Ed. by Stefan Edelkamp and Alessio Lomuscio. Vol. 4428. Lecture Notes in Computer Science. Springer, pp. 35–50 (page 96).

Stefan Edelkamp and Malte Helmert (2001). “MIPS: The Model-Checking Integrated Planning System”. In: *AI Magazine* 22.3, pp. 67–72 (pages 9, 31, 66).

Stefan Edelkamp and Peter Kissmann (2008a). “Limits and Possibilities of BDDs in State Space Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, pp. 1452–1453 (pages 22, 112).

Stefan Edelkamp and Peter Kissmann (2008b). “Partial Symbolic Pattern Databases for Optimal Sequential Planning”. In: *Proceedings of the German Conference on Artificial Intelligence (KI)*. Ed. by Andreas Dengel, Karsten Berns, Thomas M. Breuel, Frank Bomarius, and Thomas Roth-Berghofer. Vol. 5243. Lecture Notes in Computer Science. Springer, pp. 193–200 (pages 9, 14).

Stefan Edelkamp and Peter Kissmann (2009). “Optimal Symbolic Planning with Action Costs and Preferences”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Craig Boutilier, pp. 1690–1695 (pages 9, 14).

Stefan Edelkamp and Peter Kissmann (2011). “On the Complexity of BDDs for State Space Search: A Case Study in Connect Four”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Wolfram Burgard and Dan Roth. AAAI Press, pp. 18–23 (pages 22, 59).

Stefan Edelkamp, Peter Kissmann, and Álvaro Torralba (2012). “Symbolic A* Search with Pattern Databases and the Merge-and-Shrink Abstraction”. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Ed. by Luc De Raedt, Christian Bessière, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 306–311 (pages 34, 123, 184).

Stefan Edelkamp and Frank Reffel (1998). “OBDDs in Heuristic Search”. In: *Proceedings of the German Conference on Artificial Intelligence (KI)*. Ed. by Otthein Herzog and Andreas Günter. Vol. 1504. Lecture Notes in Computer Science. Springer, pp. 81–92 (pages 9, 31, 53).

Stefan Edelkamp and Stefan Schrödl (2012). *Heuristic Search – Theory and Applications*, pp. I–XXIV, 1–836 (pages 35, 153).

Patrick Eyerich and Malte Helmert (2013). “Stronger Abstraction Heuristics Through Perimeter Search”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 303–307 (pages 53, 95, 125, 132, 133).

Ariel Felner (2011). “Position Paper: Dijkstra’s Algorithm versus Uniform Cost Search or a Case Against Dijkstra’s Algorithm”. In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. Ed. by Daniel Borrajo, Maxim Likhachev, and Carlos Linares López. AAAI Press, pp. 47–51 (page 5).

Ariel Felner, Richard E. Korf, and Sarit Hanan (2004). “Additive Pattern Database Heuristics”. In: *Journal of Artificial Intelligence Research (JAIR)* 22, pp. 279–318 (pages 8, 96).

Ariel Felner, Richard E. Korf, Ram Meshulam, and Robert C. Holte (2007). “Compressed Pattern Databases”. In: *Journal of Artificial Intelligence Research (JAIR)* 30, pp. 213–247 (page 95).

Ariel Felner, Carsten Moldenhauer, Nathan R. Sturtevant, and Jonathan Schaeffer (2010). “Single-Frontier Bidirectional Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Maria Fox and David Poole. AAAI Press, pp. 59–64 (pages 9, 155).

Ariel Felner and Nir Ofek (2007). “Combining Perimeter Search and Pattern Database Abstractions”. In: *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*. Ed. by Ian Miguel and Wheeler Ruml. Vol. 4612. Lecture Notes in Computer Science. Springer, pp. 155–168 (pages 95, 125, 131, 132).

Ariel Felner, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan R. Sturtevant, and Zhifu Zhang (2011). “Inconsistent heuristics in theory and practice”. In: *Artificial Intelligence Journal* 175.9-10, pp. 1570–1603 (page 163).

Richard Fikes and Nils J. Nilsson (1971). “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artificial Intelligence Journal* 2.3/4, pp. 189–208 (page 2).

Charles Forgy (1982). “RETE: A Fast Algorithm for the Many Patterns/Many Objects Match Problem”. In: *Artificial Intelligence Journal* 19.1, pp. 17–37 (page 41).

Santiago Franco, Mike Barley, and Pat Riddle (2014). “RIDA: In Situ Selection of Heuristic Subsets”. In: *International Planning Competition (IPC)*, pp. 93–96 (page 179).

Javier García, José E. Flórez, Álvaro Torralba Arias de Reyna, Daniel Borrajo, Carlos Linares López, Angel García Olaya, and Juan Sáenz (2013). “Combining linear programming and automated planning to solve intermodal transportation problems”. In: *European Journal of Operational Research* 227.1, pp. 216–226 (pages 1, 185).

Hector Geffner (2014). "Artificial Intelligence: From programs to solvers". In: *AI Communications* 27.1, pp. 45–51 (page 1).

Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins (1998). *PDDL — The Planning Domain Definition Language Version 1.2*. Tech. rep. Yale Center for Computational Vision and Control (page 2).

Malik Ghallab, Dana S. Nau, and Paolo Traverso (2004). *Automated planning - theory and practice*. Elsevier, pp. I–XXVIII, 1–635 (page 1).

Andrew V. Goldberg and Renato Fonseca F. Werneck (2005). "Computing Point-to-Point Shortest Paths from External Memory". In: *Proceedings of the Workshop on Algorithm Engineering and Experiments and the Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO)*. Ed. by Camil Demetrescu, Robert Sedgewick, and Roberto Tamassia. SIAM, pp. 26–40 (page 28).

Eric A. Hansen, Rong Zhou, and Zhengzhu Feng (2002). "Symbolic Heuristic Search Using Decision Diagrams". In: *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*. Ed. by Sven Koenig and Robert C. Holte. Vol. 2371. Lecture Notes in Computer Science. Springer, pp. 83–98 (pages 9, 32).

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107 (pages 5, 6, 30, 31, 164, 166).

Patrik Haslum (2008). "Additive and reversed relaxed reachability heuristics revisited". In: *International Planning Competition (IPC)* (page 55).

Patrik Haslum (2009). "$h^m(P) = h^1(P^m)$: Alternative Characterisations of the Generalisation From $h^{\max}$ To h^m". In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 354–357 (pages 6, 55).

Patrik Haslum (2014). "h^{++} and h^{++}_{ce} (IPC 2014 Planner Abstract)". In: *International Planning Competition (IPC)*, p. 87 (page 179).

Patrik Haslum, Blai Bonet, and Hector Geffner (2005). "New Admissible Heuristics for Domain-Independent Planning". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Manuela M. Veloso and Subbarao Kambhampati. AAAI Press / The MIT Press, pp. 1163–1168 (pages 53, 54, 68, 69, 95).

Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig (2007). "Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, pp. 1007–1012 (pages 7, 14, 35, 96).

Patrik Haslum and Hector Geffner (2000). “Admissible Heuristics for Optimal Planning”. In: *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. Ed. by Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock. AAAI, pp. 140–149 (pages 4, 6, 55).

Patrik Haslum, Malte Helmert, and Anders Jonsson (2013). “Safe, Strong, and Tractable Relevance Analysis for Planning.” In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 317–321 (page 113).

Malte Helmert (2003). “Complexity results for standard benchmark domains in planning”. In: *Artificial Intelligence Journal* 143.2, pp. 219–262 (page 6).

Malte Helmert (2004). “A Planning Heuristic Based on Causal Graph Analysis”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Shlomo Zilberstein, Jana Koehler, and Sven Koenig. AAAI, pp. 161–170 (page 4).

Malte Helmert (2006a). “New Complexity Results for Classical Planning Benchmarks”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 52–62 (page 6).

Malte Helmert (2006b). “The Fast Downward Planning System”. In: *Journal of Artificial Intelligence Research (JAIR)* 26, pp. 191–246 (pages 14, 41, 56, 81, 98, 144).

Malte Helmert (2009). “Concise finite-domain representations for PDDL planning tasks”. In: *Artificial Intelligence Journal* 173.5-6, pp. 503–535 (pages 2–4, 55, 57).

Malte Helmert and Carmel Domshlak (2009). “Landmarks, Critical Paths and Abstractions: What’s the Difference Anyway?” In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 162–169 (pages 6, 7, 11, 14, 91, 103, 146).

Malte Helmert, Patrik Haslum, and Jörg Hoffmann (2007). “Flexible Abstraction Heuristics for Optimal Sequential Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Mark S. Boddy, Maria Fox, and Sylvie Thiébaux. AAAI, pp. 176–183 (pages 93, 95, 96, 98, 143).

Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim (2014). “Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces”. In: *Journal of the ACM* 61.3, 16:1–16:63 (pages xix, 7, 11, 14, 92, 93, 95–99, 103, 104, 107, 119, 122).

Malte Helmert and Hauke Lasinger (2010). “The Scanalyzer Domain: Greenhouse Logistics as a Planning Problem”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz. AAAI, pp. 234–237 (page 1).

Malte Helmert and Robert Mattmüller (2008). “Accuracy of Admissible Heuristic Functions in Selected Planning Domains”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, pp. 938–943 (page 6).

Malte Helmert and Silvia Richter (2004). “Fast downward-making use of causal dependencies in the problem representation”. In: *International Planning Competition (IPC)*, pp. 41–43 (page 14).

Malte Helmert, Gabriele Röger, and Silvan Sievers (2015). “On the Expressive Power of Non-Linear Merge-and-Shrink Representations”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press (pages 111, 122).

Jörg Hoffmann (2005). “Where ’Ignoring Delete Lists’ Works: Local Search Topology in Planning Benchmarks”. In: *Journal of Artificial Intelligence Research (JAIR)* 24, pp. 685–758 (page 6).

Jörg Hoffmann, Piergiorgio Bertoli, Malte Helmert, and Marco Pistore (2009). “Message-Based Web Service Composition, Integrity Constraints, and Planning under Uncertainty: A New Connection”. In: *Journal of Artificial Intelligence Research (JAIR)* 35, pp. 49–117 (page 1).

Jörg Hoffmann and Bernhard Nebel (2001). “The FF Planning System: Fast Plan Generation Through Heuristic Search”. In: *Journal of Artificial Intelligence Research (JAIR)* 14, pp. 253–302 (page 6).

Robert C. Holte, Ariel Felner, Jack Newton, Ram Meshulam, and David Furcy (2006). “Maximizing over multiple pattern databases speeds up heuristic search”. In: *Artificial Intelligence Journal* 170.16-17, pp. 1123–1136 (page 7).

Robert C. Holte, Jeffery Grajkowski, and Brian Tanner (2005). “Hierarchical Heuristic Search Revisited”. In: *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*. Ed. by Jean-Daniel Zucker and Lorenza Saitta. Vol. 3607. Lecture Notes in Computer Science. Springer, pp. 121–133 (page 155).

Robert C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy (2004). “Multiple Pattern Databases”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Shlomo Zilberstein, Jana Koehler, and Sven Koenig. AAAI, pp. 122–131 (page 95).

Robert C. Holte, M. B. Perez, Robert M. Zimmer, and Alan J. MacDonald (1996). “Hierarchical A*: Searching Abstraction Hierarchies Efficiently”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by William J. Clancey and Daniel S. Weld. AAAI Press / The MIT Press, pp. 530–535 (pages 155, 156).

Youpyo Hong, Peter A. Beerel, Jerry R. Burch, and Kenneth L. McMillan (1997). “Safe BDD Minimization Using Don’t Cares”. In: *Design Automation Conference (DAC)*, pp. 208–213 (page 66).

Youpyo Hong, Peter A. Beerel, Jerry R. Burch, and Kenneth L. McMillan (2000). “Sibling-substitution-based BDD minimization using don’t cares”. In: *IEEE Transactions on CAD of Integrated Circuits and Systems* 19.1, pp. 44–55 (page 67).

Richard Howey, Derek Long, and Maria Fox (2004). “VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL”. In: *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE Computer Society, pp. 294–301 (page 11).

William Hung (1997). “Exploiting Symmetry for Formal Verification”. MA thesis. University of Texas at Austin (page 22).

Rune M. Jensen, Randal E. Bryant, and Manuela M. Veloso (2002). “SetA*: An efficient BDD-Based Heuristic Search Algorithm”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Rina Dechter and Richard S. Sutton. AAAI Press / The MIT Press, pp. 668–673 (pages 9, 32).

Rune M. Jensen, Manuela M. Veloso, and Randal E. Bryant (2008). “State-set branching: Leveraging BDDs for heuristic search”. In: *Artificial Intelligence Journal* 172.2-3, pp. 103–139 (pages 9, 38).

Peter Jonsson and Christer Bäckström (1998). “State-Variable Planning Under Structural Restrictions: Algorithms and Complexity”. In: *Artificial Intelligence Journal* 100.1-2, pp. 125–176 (page 4).

Hermann Kaindl and Gerhard Kainz (1997). “Bidirectional Heuristic Search Reconsidered”. In: *Journal of Artificial Intelligence Research (JAIR)* 7, pp. 283–317 (pages 9, 131, 151, 153, 154).

Hermann Kaindl, Gerhard Kainz, Roland Steiner, Andreas Auer, and Klaus Radda (1999). “Switching from Bidirectional to Unidirectional Search.” In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Thomas Dean. Morgan Kaufmann, pp. 1178–1183 (page 155).

Erez Karpas and Carmel Domshlak (2009). “Cost-Optimal Planning with Landmarks”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Craig Boutilier, pp. 1728–1733 (page 7).

Erez Karpas, Michael Katz, and Shaul Markovitch (2011). “When Optimal Is Just Not Good Enough: Learning Fast Informative Action Cost Partitionings”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert. AAAI, pp. 122–129 (page 8).

Erez Karpas, David Tolpin, Tal Beja, Solomon Eyal Shimony, and Ariel Felner (2014). “The Rational Lazy A* Planner”. In: *International Planning Competition (IPC)*, pp. 97–100 (page 179).

Michael Katz and Carmel Domshlak (2010a). “Implicit Abstraction Heuristics”. In: *Journal of Artificial Intelligence Research (JAIR)* 39, pp. 51–126 (page 94).

Michael Katz and Carmel Domshlak (2010b). “Optimal admissible composition of abstraction heuristics”. In: *Artificial Intelligence Journal* 174.12-13, pp. 767–798 (page 8).

Michael Katz, Jörg Hoffmann, and Malte Helmert (2012). “How to Relax a Bisimulation?” In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet. AAAI, pp. 101–109 (pages 99, 113).

Peter Kissmann (2012). “Symbolic Search in Planning and General Game Playing”. PhD thesis. Universität Bremen (pages 10, 14, 21, 81, 103, 133).

Peter Kissmann and Stefan Edelkamp (2011). “Improving Cost-Optimal Domain-Independent Symbolic Planning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Wolfram Burgard and Dan Roth. AAAI Press, pp. 992–997 (pages 9, 14, 21, 33, 35, 51, 68, 103).

Peter Kissmann, Stefan Edelkamp, and Jörg Hoffmann (2014). “Gamer and Dynamic-Gamer – Symbolic Search at IPC 2014”. In: *International Planning Competition (IPC)*, pp. 77–84 (pages 22, 179).

Peter Kissmann and Jörg Hoffmann (2013). “What’s in It for My BDD? On Causal Graphs and Variable Orders in Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 327–331 (pages 21, 22, 94).

Peter Kissmann and Jörg Hoffmann (2014). “BDD Ordering Heuristics for Classical Planning”. In: *Journal of Artificial Intelligence Research (JAIR)* 51, pp. 779–804 (pages 21, 94, 179).

Craig A. Knoblock (1994). “Automatically Generating Abstractions for Planning”. In: *Artificial Intelligence Journal* 68.2, pp. 243–302 (pages 4, 92).

Alexander Koller and Jörg Hoffmann (2010). “Waking Up a Sleeping Rabbit: On Natural-Language Sentence Generation with FF”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz. AAAI, pp. 238–241 (page 1).

Richard E. Korf (1985). “Depth-First Iterative-Deepening: An Optimal Admissible Tree Search”. In: *Artificial Intelligence Journal* 27.1, pp. 97–109 (pages 5, 155).

Richard E. Korf (1997). “Finding Optimal Solutions to Rubik’s Cube Using Pattern Databases”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Benjamin Kuipers and Bonnie L. Webber. AAAI Press / The MIT Press, pp. 700–705 (page 6).

Richard E. Korf and Ariel Felner (2002). “Disjoint pattern database heuristics”. In: *Artificial Intelligence Journal* 134.1-2, pp. 9–22 (page 8).

James B. H. Kwa (1989). “BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm”. In: *Artificial Intelligence Journal* 38.1, pp. 95–109 (pages 9, 154, 160).

Bradford John Larsen, Ethan Burns, Wheeler Ruml, and Robert Holte (2010). “Searching Without a Heuristic: Efficient Use of Abstraction”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Maria Fox and David Poole. AAAI Press, pp. 114–120 (pages 156, 162).

Michael J. Leighton, Wheeler Ruml, and Robert C. Holte (2011). “Faster Optimal and Suboptimal Hierarchical Search”. In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. Ed. by Daniel Borrajo, Maxim Likhachev, and Carlos Linares López. AAAI Press, pp. 92–99 (page 156).

Carlos Linares López (2005). “Heuristic Perimeter Search: First Results”. In: *Proceedings of the Conference of the Spanish Association for Artificial Intelligence (CAEPIA)*. Ed. by Roque Marín, Eva Onaindia, Alberto Bugarín, and José Santos Reyes. Vol. 4177. Lecture Notes in Computer Science. Springer, pp. 251–260 (page 155).

Carlos Linares López (2008). “Multi-valued Pattern Databases”. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Ed. by Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris. Vol. 178. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 540–544 (pages 95, 132).

Carlos Linares López, Sergio Jiménez Celorrio, and Malte Helmert (2013). “Automating the evaluation of planning systems”. In: *AI Communications* 26.4, pp. 331–354 (page 11).

Carlos Linares López and Andreas Junghanns (2002). “Perimeter Search Performance”. In: *Proceedings of the Conference on Computers and Games*. Ed. by Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson. Vol. 2883. Lecture Notes in Computer Science. Springer, pp. 345–359 (page 131).

Marco Lippi, Marco Ernandes, and Ariel Felner (2012). “Efficient Single Frontier Bidirectional Search”. In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. Ed. by Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant. AAAI Press, pp. 49–56 (page 155).

Yuri Malitsky, David Wang, and Erez Karpas (2014). “The AllPACA Planner: All Planners Automatic Choice Algorithm”. In: *International Planning Competition (IPC)*, pp. 71–73 (page 179).

Giovanni Manzini (1995). “BIDA: An Improved Perimeter Search Algorithm”. In: *Artificial Intelligence Journal* 75.2, pp. 347–360 (pages 95, 131, 155).

Bart Massey (1999). “Directions In Planning: Understanding The Flow Of Time In Planning”. PhD thesis. Computational Intelligence Research Laboratory, University of Oregon (pages 9, 55).

Kenneth L. McMillan (1993). *Symbolic model checking*. Kluwer Academic publishers, pp. I–XV, 1–194 (pages 9, 17).

Kenneth L. McMillan (1996). “A Conjunctively Decomposed Boolean Representation for Symbolic Model Checking”. In: *Computer Aided Verification (CAV)*. Ed. by Rajeev Alur and Thomas A. Henzinger. Vol. 1102. Lecture Notes in Computer Science. Springer, pp. 13–25 (page 66).

Làszló Méro (1984). “A Heuristic Search Algorithm with Modifiable Estimate.” In: *Artificial Intelligence Journal* 23.1, pp. 13–27 (pages 33, 159).

Shin-ichi Minato (1993). “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems”. In: *Design Automation Conference (DAC)*, pp. 272–277 (page 23).

In-Ho Moon, James H. Kukula, Kavita Ravi, and Fabio Somenzi (2000). “To split or to conjoin: the question in image computation”. In: *Design Automation Conference (DAC)*, pp. 23–28 (page 38).

T. A. J. Nicholson (1966). “Finding the Shortest Route between Two Points in a Network”. In: *The Computer Journal* 9.3, pp. 275–280 (page 28).

Nils J. Nilsson (1982). *Principles of Artificial Intelligence*. Springer, pp. I–XV, 1–476 (page 156).

Raz Nissim, Jörg Hoffmann, and Malte Helmert (2011). “Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Toby Walsh. IJCAI/AAAI, pp. 1983–1990 (pages 98, 99, 112, 116).

Sergio Núñez, Daniel Borrajo, and Carlos Linares López (2012). “Performance Analysis of Planning Portfolios”. In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. Ed. by Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant. AAAI Press, pp. 65–71 (page 179).

Sergio Núñez, Daniel Borrajo, and Carlos Linares López (2014a). “MIPlan and DPMPlan”. In: *International Planning Competition (IPC)*, pp. 13–16 (page 179).

Sergio Núñez, Isabel Cenamor, and Jesús Virseda (2014b). “NuCeLaR”. In: *International Planning Competition (IPC)*, pp. 48–51 (page 179).

Armando Ordóñez, Vidal Alcázar, Juan Carlos Corrales, and Paolo Falcarin (2014). “Automated context aware composition of Advanced Telecom Services for environmental early warnings”. In: *Expert Systems with Applications* 41.13, pp. 5907–5916 (page 1).

Jörn Ossowski and Christel Baier (2006). “Symbolic Reasoning with Weighted and Normalized Decision Diagrams”. In: *Electronic Notes in Theoretical Computer Science* 151.1, pp. 39–56 (page 24).

Héctor Luis Palacios Verdes (2009). “Translation-based approaches to Conformant Planning”. PhD thesis. Universitat Pompeu Fabra (page 23).

Bo Pang and Robert C. Holte (2012). “Multimapping Abstractions and Hierarchical Heuristic Search”. In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. Ed. by Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant. AAAI Press, pp. 72–79 (page 93).

Edwin P. D. Pednault (1994). “ADL and the State-Transition Model of Action”. In: *Journal of Logic and Computation* 4.5, pp. 467–512 (page 2).

Ira Pohl (1969). *Bi-directional and heuristic search in path problems*. 104. Department of Computer Science, Stanford University. (pages 28, 151, 154).

Ira Pohl (1971). “Bi-directional search”. In: *Machine Intelligence*. Vol. 6. Edinburgh University Press., pp. 127–140 (page 154).

George Politowski and Ira Pohl (1984). “D-Node Retargeting in Bidirectional Heuristic Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Ronald J. Brachman. AAAI Press, pp. 274–277 (page 155).

Florian Pommerening and Malte Helmert (2013). “Incremental LM-Cut”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 162–170 (page 179).

Florian Pommerening, Gabriele Röger, and Malte Helmert (2013). “Getting the Most Out of Pattern Databases for Classical Planning”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Francesca Rossi. IJCAI/AAAI, pp. 2357–2364 (page 8).

Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet (2014). “LP-Based Heuristics for Cost-Optimal Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Steve Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml. AAAI, pp. 226–234 (page 8).

Julie Porteous, Laura Sebastia, and Jorg Hoffmann (2001). “On the extraction, ordering, and usage of landmarks in planning”. In: *Proceedings of the JoEuropean Conference on Planning (ECP)*, pp. 174–182 (page 7).

Francisco Javier Pulido, Lawrence Mandow, and José-Luis Pérez-de-la-Cruz (2012). “A two-phase bidirectional heuristic search algorithm”. In: *Proceedings of the Starting AI Researchers’ Symposium (STAIRS)*. Ed. by Kristian Kersting and Marc Toussaint. Vol. 241. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 240–251 (page 155).

Silvia Richter and Malte Helmert (2009). “Preferred Operators and Deferred Evaluation in Satisficing Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 273–280 (page 33).

Silvia Richter, Malte Helmert, and Matthias Westphal (2008). “Landmarks Revisited”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, pp. 975–982 (page 7).

Wheeler Ruml, Minh Binh Do, Rong Zhou, and Markus P. J. Fromherz (2011). “On-line Planning and Scheduling: An Application to Controlling Modular Printers”. In: *Journal of Artificial Intelligence Research (JAIR)* 40, pp. 415–468 (page 1).

Stuart J. Russell and Peter Norvig (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)* Pearson Education, pp. I–XVIII, 1–1132 (page 1).

Scott Sanner and David A. McAllester (2005). "Affine Algebraic Decision Diagrams (AADDs) and their Application to Structured Probabilistic Inference". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, pp. 1384–1390 (page 24).

Ulrich Scholz (2004). "Reducing planning problems by path reduction". PhD thesis. Darmstadt University of Technology (page 113).

Jendrik Seipp and Malte Helmert (2013). "Counterexample-guided Cartesian Abstraction Refinement". In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 347–351 (pages 7, 95).

Jendrik Seipp, Silvan Sievers, and Frank Hutter (2014). "Fast Downward Cedalion". In: *International Planning Competition (IPC)*, pp. 17–27 (page 179).

Claude E. Shannon (1938). "A Symbolic Analysis of Relay and Switching Circuits". In: *Transactions of the American Institute of Electrical Engineers (AIEE)* 57.12, pp. 713–723 (page 18).

Detlef Sieling (2002). "The Nonapproximability of OBDD Minimization". In: *Information and Computation* 172.2, pp. 103–138 (page 20).

Detlef Sieling and Ingo Wegener (1993). "Reduction of OBDDs in Linear Time". In: *Information Processing Letters* 48.3, pp. 139–144 (pages 19, 23).

Silvan Sievers, Martin Wehrle, and Malte Helmert (2014). "Generalized Label Reduction for Merge-and-Shrink Heuristics". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, pp. 2358–2366 (pages 98, 99, 114, 116).

Fabio Somenzi (2012). *CUDD: CU Decision Diagram Package Release 2.5.0* (pages 14, 109).

David Tolpin, Tal Beja, Solomon Eyal Shimony, Ariel Felner, and Erez Karpas (2013). "Toward Rational Deployment of Multiple Heuristics in A*". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Francesca Rossi. IJCAI/AAAI (pages 33, 34, 179).

Álvaro Torralba and Vidal Alcázar (2013). "Constrained Symbolic Search: On Mutexes, BDD Minimization and More". In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. Ed. by Malte Helmert and Gabriele Röger. AAAI Press, pp. 175–183 (pages 80, 183, 185).

Álvaro Torralba, Vidal Alcázar, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp (2014a). "SymBA*: A Symbolic Bidirectional A* Planner". In: *International Planning Competition (IPC)*, pp. 105–109 (page 179).

Álvaro Torralba, Vidal Alcázar, Peter Kissmann, and Stefan Edelkamp (2014b). “cGamer: Constrained Gamer”. In: *International Planning Competition (IPC)*, pp. 74–76 (page 178).

Álvaro Torralba, Vidal Alcázar, Carlos Linares López, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp (2014c). “SPM&S Planner: Symbolic Perimeter Merge-and-Shrink”. In: *International Planning Competition (IPC)*, pp. 101–104 (page 178).

Álvaro Torralba, Stefan Edelkamp, and Peter Kissmann (2013a). “Transition Trees for Cost-Optimal Symbolic Planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 206–214 (pages 28, 52, 183, 184).

Álvaro Torralba, Carlos Linares López, and Daniel Borrajo (2013b). “Symbolic Merge-and-Shrink for Cost-Optimal Planning”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Francesca Rossi. IJCAI/AAAI, pp. 2394–2400 (pages 53, 126, 132, 145, 149, 150, 184, 185).

Menkes van den Briel, J. Benton, Subbarao Kambhampati, and Thomas Vossen (2007). “An LP-Based Heuristic for Optimal Planning”. In: *Principles and Practice of Constraint Programming (CP)*. Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, pp. 651–665 (page 7).

Vincent Vidal and Hector Geffner (2005). “Solving Simple Planning Problems with More Inference and No Search”. In: *Principles and Practice of Constraint Programming (CP)*. Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, pp. 682–696 (page 58).

Martin Wehrle and Malte Helmert (2012). “About Partial Order Reduction in Planning and Computer Aided Verification”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet. AAAI, pp. 287–305 (page 179).

Christopher Wilt and Wheeler Ruml (2011). “Cost-Based Heuristic Search Is Sensitive to the Ratio of Operator Costs”. In: *Proceedings of the Symposium on Combinatorial Search (SoCS)*. Ed. by Daniel Borrajo, Maxim Likhachev, and Carlos Linares López. AAAI Press, pp. 172–179 (page 11).

Yexiang Xue, Arthur Choi, and Adnan Darwiche (2012). “Basing Decisions on Sentences in Decision Diagrams”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Jörg Hoffmann and Bart Selman. AAAI Press, pp. 842–849 (page 24).

Ryo Yoshinaka, Jun Kawahara, Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato (2012). “Counterexamples to the long-standing conjecture on the complexity of BDD binary operations”. In: *Information Processing Letters* 112.16, pp. 636–640 (page 20).

Sandra Zilles and Robert C. Holte (2010). “The computational complexity of avoiding spurious states in state space abstraction”. In: *Artificial Intelligence Journal* 174.14, pp. 1072–1092 (page 53).