Flexible Abstraction Heuristics for Optimal Sequential Planning

Malte Helmert
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee 52
79110 Freiburg, Germany
helmert@informatik.uni-freiburg.de

Patrik Haslum*
NICTA & Australian National University
Locked Bag 8001
Canberra ACT 2601, Australia
Patrik.Haslum@nicta.com.au

Jörg Hoffmann
DERI
University of Innsbruck
Technikerstraße 21a
6020 Innsbruck, Austria
joerg.hoffmann@deri.at

Abstract

We describe an approach to deriving consistent heuristics for automated planning, based on explicit search in abstract state spaces. The key to managing complexity is interleaving composition of abstractions over different sets of state variables with abstraction of the partial composites.

The approach is very general and can be instantiated in many different ways by following different *abstraction strategies*. In particular, the technique subsumes *planning with pattern databases* as a special case. Moreover, with suitable abstraction strategies it is possible to derive perfect heuristics in a number of classical benchmark domains, thus allowing their optimal solution in polynomial time.

To evaluate the practical usefulness of the approach, we perform empirical experiments with one particular abstraction strategy. Our results show that the approach is competitive with the state of the art.

Introduction

In contrast to tremendous improvements in the scaling behaviour of satisficing planners (as evidenced by the results of recent planning competitions) and planning systems that minimize so-called "parallel length" (Kautz, Selman, & Hoffmann 2006), the problem of *optimal sequential planning*, i. e. finding a plan with a minimal number of actions, remains very challenging.

The only known approach to optimal planning that is generally viable is search, in one form or another. The most important general method for improving the efficiency of search for optimal solutions is the use of *admissible heuristics*, i. e. lower bound functions on the distance to the nearest solution in the search space. There are of course numerous other ways to enhance the efficiency of search procedures for optimal planning: Compacting the representation of sets of states, using data structures such as BDDs, and using external storage permits larger search spaces to be explored (Edelkamp 2005). Clever branching strategies, for example using a constraint representation, help focus search on the relevant choice points, thus reducing branching factor and/or solution depth (Grandcolas & Pain-Barre 2007; Vidal & Geffner 2006). These improvements, however, are not enough to replace heuristics as a corner-stone of optimal planning. Moreover, improvements to heuristics and to other aspects of search complement each other.

A useful heuristic function must be efficiently computable (at most low order polynomial time) as well as accurate. Improving the accuracy of a heuristic function, without worsening its computational properties, usually translates directly into faster search for optimal solutions. This is precisely the contribution of this paper: We describe a way of deriving more accurate admissible heuristics for forward search, at a reasonable computational cost.

Our heuristics are *abstraction heuristics*, meaning the heuristic value is the optimal cost of the solution to an abstraction of the planning task. An abstraction is a mapping that reduces the size of the state space, by "collapsing" several states into one. By making the abstract space small enough, it becomes feasible to find the optimal solution by blind search. Distances in the abstract space are computed in a preprocessing phase and stored in memory, so that heuristic evaluation during search can be done by a simple lookup.

A particular form of abstraction heuristics, known as *pattern databases* (PDBs) have been shown to be very useful in several hard search problems, including optimal planning (Culberson & Schaeffer 1998; Edelkamp 2001). The abstraction mappings underlying PDB heuristics for planning are *projections*, which ignore completely all but a subset of the state variables of the planning task (known as the "pattern"): states that do not differ on the chosen variables are identified in the abstract space. This limits the representational power of PDB heuristics: in some planning tasks the abstraction that would be most useful as a heuristic can not be represented as a PDB (of reasonable size). Heuristics based on abstractions more general than projections have been used in some applications, but so far not for planning (Hoffmann et al., 2006, consider a form of non-projection abstractions, but do not use them to derive heuristics).

Recently, Dräger, Finkbeiner & Podelski (2006), in the context of verification of systems of concurrent automata, presented a method to construct abstractions that combine information about all state variables. The computational feasibility of this approach rests on interleaving composition with abstraction of the partial composites. The greater flexibility offered by not restricting abstractions to be projections

*NICTA is funded through the Australian government's *backing Australia's ability* initiative.

is a mixed blessing. It does, as we show, allow very accurate heuristics to be obtained from relatively small abstractions. However, the problem of selecting from the vast number of possible abstractions a good one, which is hard already for PDB heuristics, becomes even harder. A good *abstraction strategy* is vital for the approach to be practical.

We generalize the method of Dräger et al. to the problem of optimal planning, using the SAS$^+$ representation, and investigate various refinements of their abstraction strategy. In particular, we show that good – in some domains even perfect – heuristics can be obtained using a simpler strategy for the order of composition but a more refined strategy for simplifying the intermediate abstractions. We also present the first evidence that, with an appropriate abstraction strategy, this approach does in fact lead to better heuristics than using only projections, as done in PDBs. We prove that the framework subsumes PDBs, including additive PDBs, and that in certain domains, for which planning is known to be tractable, it can construct and succinctly represent perfect heuristics, i. e., heuristics that give exact estimates of goal distance for all reachable states. Experimentally, we show that our abstraction strategy frequently results in better heuristics than the currently best known method for automatically constructing PDB heuristics.

The next two sections formally introduce planning problems and their abstractions. We then discuss the computation of abstractions in general and the specific strategy we use. Finally, we provide theoretical and experimental results, demonstrating the effectiveness of the approach.

Background

We consider optimal planning in the classical setting, using the SAS$^+$ representation of planning tasks. A task specified in STRIPS or PDDL can be converted to SAS$^+$ representation automatically (Helmert 2006a). In this paper, we assume that the objective is to minimize plan length. We remark that our methods generalize easily to the case of minimizing the sum of non-negative action costs.

Definition 1 ***SAS$^+$ planning task.***
*A **SAS$^+$ planning task** or **SAS$^+$ task** for short is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ with the following components:*

- *$\mathcal{V} = \{v_1, \ldots, v_n\}$ is a set of **state variables**, each with an associated finite domain $\mathcal{D}_v$.*
  *A **partial variable assignment** over $\mathcal{V}$ is a function s on some subset of $\mathcal{V}$ such that $s(v) \in \mathcal{D}_v$ wherever $s(v)$ is defined. If $s(v)$ is defined for all $v \in \mathcal{V}$, s is called a **state**.*
- *$\mathcal{O}$ is a set of **operators**, where an operator is a pair $\langle pre, eff \rangle$ of partial variable assignments called **preconditions** and **effects**, respectively.*
- *s_0 is a state called the **initial state**, and $s_\star$ is a partial variable assignment called the **goal**.*

The semantics of a planning task are given by mapping it to a *transition graph*. (Transition graphs are often called transition *systems*; we call them graphs to emphasize their interpretation as (labelled) digraphs.) Searching in this transition graph corresponds to forward state space search.

Definition 2 ***Transition graphs and plans.***
*A **transition graph** is a 5-tuple $\mathcal{T} = \langle S, L, A, s_0, S_\star \rangle$, where S is a finite set of **states**, L is a finite set of **transition labels**, $A \subseteq S \times L \times S$ is a set of (labelled) **transitions**, $s_0 \in S$ is the **initial state**, and $S_\star \subseteq S$ is the set of **goal states**.*

*A path from s_0 to any $s_\star \in S_\star$ following the transitions of $\mathcal{T}$ is a **plan** for $\mathcal{T}$. A plan is **optimal** iff the length of the path is minimal.*

We denote the transition graph associated with a SAS$^+$ planning task Π by $\mathcal{T}(\Pi)$. Its states are the states of the task, i. e. complete assignments of values to variables, and the graph has an edge, labelled by $o \in \mathcal{O}$, from s to s' if o is applicable in s and applying the operator to s leads to s'. A solution to the planning task is a path through this graph.

The transition graph of Π can also be viewed as the synchronized product of transition graphs defined over variables in the planning task. With this in mind, the similarity with the concurrent automata model used by Dräger et al. is easy to see. However, SAS$^+$ operators allow for more general forms of synchronization. Some care must be taken in defining the transition graphs of individual variables to ensure that their product really is isomorphic to $\mathcal{T}(\Pi)$. Exactly how this is done is described in the next section.

Abstractions

Abstractions of transition graphs are at the core of our approach to constructing heuristics. Abstracting means ignoring some information or some constraints to obtain a more "coarse grained", and typically smaller, version of the transition graph. Abstraction has the important property that it preserves paths in the graph. This is what makes it suitable as a way to define admissible heuristics. Formally, abstraction of transition graphs is defined as follows:

Definition 3 ***Abstraction.***
*An **abstraction** of a transition graph $\mathcal{T} = \langle S, L, A, s_0, S_\star \rangle$ is a pair $\mathcal{A} = \langle \mathcal{T}', \alpha \rangle$ where $\mathcal{T}' = \langle S', L', A', s'_0, S'_\star \rangle$ is a transition graph called the **abstract transition graph** and $\alpha : S \to S'$ is a function called the **abstraction mapping**, such that $L' = L$, $\langle \alpha(s), l, \alpha(s') \rangle \in A'$ for all $\langle s, l, s' \rangle \in A$, $\alpha(s_0) = s'_0$, and $\alpha(s_\star) \in S'_\star$ for all $s_\star \in S_\star$.*

*If the abstract transition graph contains no transitions or goal states in addition to those required by the above definition, $\mathcal{A}$ is called a **homomorphism**.*

Note that abstraction is transitive: if $\langle \mathcal{T}', \alpha' \rangle$ is an abstraction of a transition graph $\mathcal{T}$, and $\langle \mathcal{T}'', \alpha'' \rangle$ is an abstraction of $\mathcal{T}'$, then $\langle T'', \alpha'' \circ \alpha' \rangle$ is also an abstraction of $\mathcal{T}$.

When the abstraction mapping is not relevant, we identify an abstraction with its abstract transition graph. In the case of homomorphisms, we may also identify an abstraction with its abstraction mapping, since the mapping completely determines the abstract graph.

Definition 4 ***Abstraction heuristic.***
Let Π be a SAS$^+$ task with state set $\mathcal{S}$, and let $\mathcal{A} = \langle \mathcal{T}, \alpha \rangle$ be an abstraction of its transition graph.

*The **abstraction heuristic** $h^{\mathcal{A}}$ is the function which assigns to each state $s \in \mathcal{S}$ the length of the shortest path, in $\mathcal{T}$, from $\alpha(s)$ to any goal state of $\mathcal{T}$.*

The value of $h^{\mathcal{A}}(s)$ is a lower bound on the length of the shortest path from s to any goal state in $\mathcal{T}(\Pi)$. Thus, $h^{\mathcal{A}}$ is an admissible and consistent heuristic for forward state space search. This is true of abstraction heuristics in general, not only abstractions of planning tasks. It follows from the simple fact that abstractions preserve edges in the transition graph: every path in $\mathcal{T}(\Pi)$ is also a path in $\mathcal{A}$, so the shortest path in the latter can not be longer than the shortest path in the former. The converse is not necessarily true, as there may be a shorter path to a goal state in $\mathcal{A}$ than in $\mathcal{T}(\Pi)$, so $h^{\mathcal{A}}$ will typically underestimate the real solution length.

The above definition does not specify how, for a given state, the value of $h^{\mathcal{A}}$ is computed. In the general setting of domain-independent planning, the only realistic possibility is by searching the transition graph of the abstraction. For pattern database heuristics, an exhaustive search of each abstraction is done as a preprocessing step, and goal distances for all abstract states are stored in memory. We follow the same approach. As mentioned in the introduction, the abstraction heuristics previously considered in planning, i. e. PDBs, are based on a particular type of abstractions, namely projections. Formally, these are defined as follows.

Definition 5 ***Projection.***
Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a SAS$^+$ task with state set $\mathcal{S}$, and let $V \subseteq \mathcal{V}$ be a subset of its variables.

A homomorphism on $\mathcal{T}(\Pi)$ defined by a mapping α such that $\alpha(s) = \alpha(s')$ iff $s(v) = s'(v)$ for all $v \in V$ is called a ***projection*** *onto variable set V, written as π_V.*

If V is a singleton set, $\pi_{\{v\}}$ is called an ***atomic projection****, also written π_v.*

The relationship between PDBs and our more general form of abstraction heuristics is further discussed later on.

For readers familiar with the concept of the *domain transition graph* (DTG) of a SAS$^+$ variable, we remark that the transition graph of an atomic projection onto a variable v is not identical to the DTG of v. This is because the transition graph of π_v has edges representing the effect of *any* operator on v, including operators that have no effect on v.

The abstractions we base our heuristics on are constructed by interleaving composition of abstractions with further abstraction of the composites. Composing here means the standard operation of forming the synchronized product, formally defined as follows.

Definition 6 ***Synchronized product.***
Let $\mathcal{A}' = \langle \langle S', L, A', s'_0, S'_\star \rangle, \alpha' \rangle$ and $\mathcal{A}'' = \langle \langle S'', L, A'', s''_0, S''_\star \rangle, \alpha'' \rangle$ be abstractions of a transition graph $\mathcal{T}$.

The ***synchronized product*** *of $\mathcal{A}'$ and $\mathcal{A}''$ is defined as $\mathcal{A}' \otimes \mathcal{A}'' = \langle \langle S, L, A, s_0, S_\star \rangle, \alpha \rangle$, where $S = S' \times S''$, $\langle (s', s''), l, (t', t'') \rangle \in A$ iff $\langle s', l, t' \rangle \in A'$ and $\langle s'', l, t'' \rangle \in A''$, $s_0 = (s'_0, s''_0)$, $S_\star = S'_\star \times S''_\star$, and $\alpha : \mathcal{S} \to S$ is defined by $\alpha(s) = (\alpha'(s), \alpha''(s))$.*

The synchronized product of two abstractions of a transition graph $\mathcal{T}$ is again an abstraction of $\mathcal{T}$. This is an immediate consequence of the definitions. Forming the product is an associative and commutative operation, modulo isomorphism of transition graphs.

As we earlier hinted, the synchronized product of all atomic projections of a SAS$^+$ task Π is equal to the full transition graph $\mathcal{T}(\Pi)$. In other words, the atomic projections are a complete representation of Π, from which $\mathcal{T}(\Pi)$ can be reconstructed by synchronized product operations. To show this, we first need to introduce the following concept of independence for abstractions.

Definition 7 ***Relevant variables, orthogonal abstractions.***
Let Π be a planning task with variable set $\mathcal{V}$, and let $\mathcal{A} = \langle \mathcal{T}, \alpha \rangle$ be an abstraction of $\mathcal{T}(\Pi)$.

We say that $\mathcal{A}$ ***depends on variable*** *$v \in \mathcal{V}$ iff there exist states s and s' with $\alpha(s) \neq \alpha(s')$ and $s(v') = s'(v')$ for all $v' \in \mathcal{V} \setminus \{v\}$. The set of* ***relevant variables*** *for $\mathcal{A}$, written varset($\mathcal{A}$), is the set of variables in $\mathcal{V}$ on which $\mathcal{A}$ depends.*

Abstractions $\mathcal{A}$ and $\mathcal{A}'$ are ***orthogonal*** *iff varset($\mathcal{A}$) $\cap$ varset($\mathcal{A}'$) $= \emptyset$.*

Clearly, projections satisfy *varset*$(\pi_V) = V$, so projections onto disjoint variable sets are orthogonal. Moreover, *varset*$(\mathcal{A} \otimes \mathcal{A}') =$ *varset*$(\mathcal{A}) \cup$ *varset*$(\mathcal{A}')$.

Theorem 8 ***Products of orthogonal homomorphisms.***
Let Π be a SAS$^+$ task, and let $\mathcal{A}$ and $\mathcal{A}'$ be orthogonal homomorphisms of $\mathcal{T}(\Pi)$. Then $\mathcal{A} \otimes \mathcal{A}'$ is a homomorphism of $\mathcal{T}(\Pi)$.

For space reasons, we omit the proof. We remark that the theorem does *not* hold if $\mathcal{A}$ and $\mathcal{A}'$ are not required to be orthogonal, and that it does *not* hold if the SAS$^+$ task contains conditional effects.[1]

It is now easy to prove that the transition graph of a SAS$^+$ task is isomorphic to the synchronized product of the atomic abstractions of all its variables, i. e. that $\mathcal{T}(\Pi) = \bigotimes_{v \in \mathcal{V}} \pi_v$. By definition, the abstraction mapping of the product is a bijection. By Theorem 8, the abstraction is a homomorphism. These two facts together imply the desired equivalence.

A Generic Abstraction Algorithm

Atomic projections and synchronized products can fully capture the state transition semantics of a SAS$^+$ task. However, for all but trivial tasks we cannot explicitly compute the product of all atomic projections: At some point, the abstract transition graphs become too large to be represented in memory. In the case of PDB heuristics, the memory limit translates into an effective limit on the number of variables that can be included in any single projection.

In contrast, the approach we pursue in this paper computes abstractions based on the *full* variable set. To do so, we maintain a pool of (orthogonal) abstractions, which initially consists of all atomic projections. We then repeatedly perform one of two possible operations until only a single abstraction remains:

- Two abstractions can be *merged* (i.e., composed) by replacing them with their synchronized product.

[1] The latter is a consequence of how we have defined the transition graph of a SAS$^+$ task, in particular the fact that edges are labelled only by operator names. In a task with conditional effects, transition labels must be extended with information about which effects are active in the transition.

generic algorithm compute-abstraction(Π, N):
  abs := $\{\pi_v \mid v$ is a variable of $\Pi\}$
  while $|$*abs*$| > 1$:
    Select $\mathcal{A}_1, \mathcal{A}_2 \in$ *abs*.
    Shrink $\mathcal{A}_1$ and/or $\mathcal{A}_2$ until *size*$(\mathcal{A}_1)\cdot$*size*$(\mathcal{A}_2) \leq N$.
    abs := (*abs* $\setminus \{\mathcal{A}_1, \mathcal{A}_2\}) \cup \{\mathcal{A}_1 \otimes \mathcal{A}_2\}$
  return the only element of *abs*

Figure 1: Algorithm for computing an abstraction for planning task Π, with abstraction size bound N. (The size of an abstraction is the number of abstract states.)

- An abstraction can be *shrunk* (i.e., abstracted) by replacing it with a homomorphism of itself.

To keep time and space requirements under control, we enforce a limit on the size of the computed abstractions, specified as an input parameter N. Each computed abstraction, including the final result, contains at most N states. If there are more than N states in the product of two abstractions $\mathcal{A}_1$ and $\mathcal{A}_2$ (i. e., if the product of their state counts exceeds N), either or both of the abstractions must be shrunk by a sufficient amount before they are merged. The general procedure is shown in Fig. 1. Note that the procedure has two important choice points:

- *merging strategy*: the decision which abstractions $\mathcal{A}_1$ and $\mathcal{A}_2$ to select from the current pool of abstractions.
- *shrinking strategy*: the decision which abstractions to shrink and how to shrink them, i. e. which homomorphism to apply to them.

We refer to the combination of a particular merging and shrinking strategy as an *abstraction strategy*. To obtain a concrete algorithm, we must augment the *generic abstraction algorithm* in Fig. 1 with an abstraction strategy.

Assuming that N is polynomially bounded by the input size (e. g., a constant) and that the abstraction strategy is efficiently computable, computing the abstraction requires only polynomial time and space. As argued previously, the resulting abstraction heuristic is admissible and consistent.

Even though these important properties hold independently of the choice of abstraction strategy, selecting a suitable strategy is of paramount importance, as it determines the quality of the resulting heuristic. In the following section, we introduce one particular abstraction strategy, which we use for experimentally evaluating the approach.

An Abstraction Strategy

The abstraction strategy we consider in this section makes use of two pieces of information: the merging strategy is based on connectivity properties of the causal graph of the SAS$^+$ task, and the shrinking strategy relies on shortest distances in the abstract transition graphs. We will now explain both components in detail.

Merging Strategy

The merging strategy we use is a special case of what we call a *linear* merging strategy. A linear strategy always maintains a single non-atomic abstraction called the *current abstraction*. Initially, the current abstraction is an atomic projection. In each iteration, the current abstraction is merged with an atomic projection to form the new current abstraction, until all atomic projections have been considered. At any point throughout the merging process, there is thus only one abstraction which is not an atomic projection, and a set of atomic projections which still need to be merged into it.

Dräger et al. use a non-linear merging strategy, but one that still has the property that each atomic abstraction is considered only once. A linear strategy is simpler and, we believe, achieves equally good results. In particular, in domains for which we prove the existence of a strategy yielding a perfect heuristic, linear strategies suffice.

A linear strategy is defined by the order in which atomic projections are merged into the current abstraction. We determine this order by following two simple rules:

1. If possible, choose a variable from which there is an arc in the causal graph to one of the previously added variables.
2. If there is no such variable, add a variable for which a goal value is defined. (In particular, choose a goal variable initially, when there are no previously added variables.)

Both rules are fairly intuitive: Goal variables are clearly important because they are the only ones for which π_v has non-zero goal distance, and causally connected variables are the only ones which can increase the heuristic estimates of the current abstraction. (We also experimented with the opposite order of rules, i. e. first adding all goal variables and then causally connected variables, with slightly worse results.)

When neither rule applies, there are no variables left that either have defined goal values or are relevant for the achievement of a goal. In other words, only irrelevant variables are left, and the remaining variables can be ignored. In our case, this situation does not arise because the PDDL-to-SAS$^+$ translator removes irrelevant variables.

In many situations, the two rules are not sufficient for deciding which variable to add next, because there are several matching candidates. In this case, we choose the candidate with the "highest level" according to the ordering criterion used by Fast Downward (Helmert 2006a).

Shrinking Strategy

To keep the size of the synchronized product $\mathcal{A} \otimes \mathcal{A}'$ below the bound N, we may need to shrink $\mathcal{A}$ or $\mathcal{A}'$ before computing the product. The shrinking strategy decides *which* of $\mathcal{A}$ and $\mathcal{A}'$ to shrink and *how* to shrink them.

Regarding the *which* question, we take the simple approach of always shrinking the current (non-atomic) abstraction. Our intuition is that atomic abstractions are typically much smaller and thus contain fewer states that can be abstracted without losing significant information. Clearly, more elaborate strategies are possible.

As for *how*, we need an algorithm that accepts an abstraction $\mathcal{A}$ and target size M and computes a homomorphism of $\mathcal{A}$ with M abstract states. In our case, we set $M = \lfloor \frac{N}{\mathit{size}(\mathcal{A}')} \rfloor$ to ensure that the following synchronized product operation respects the size limit N.

Shrinking an abstraction of size M' to size M can be understood as a sequence of $M' - M$ individual simplification steps, combining two abstract states in each step. Formally, this is equivalent to applying $M' - M$ homomorphisms in sequence, each of which maps two abstract states s and s' to a new abstract state $\{s, s'\}$ while leaving all other abstract states intact, mapping them to themselves. Adopting this view, we can describe a shrinking strategy by stating which pair of abstract states s and s' it chooses to combine whenever shrinking is necessary.

A first, very simple idea is to combine states arbitrarily, i. e., to select s and s' by uniform random choice. We did a few experiments with this approach, which led to terrible performance. A key reason for this is that combining two randomly selected nodes frequently leads to short-cuts in the abstract transition graph: If s is close to the initial state and s' is close to a goal state, then combining s and s' introduces a short path from the initial state to the goal which did not previously exist.

This problem can be avoided by only combining states with identical goal distance. We say that the *h-value* of an abstract state s is the length of a shortest path from s to some abstract goal state, in the abstract transition graph. Similarly, the *g-value* of s is the length of a shortest path from the abstract initial state to s, and the *f-value* is the sum of the g- and h-values. A shrinking strategy is called *h-preserving* if it only combines vertices with identical h-values, *g-preserving* if it only combines vertices with identical g-values, and *f-preserving* if it is both h-preserving and g-preserving.

It is easy to prove that if $\mathcal{A}'$ is an abstraction of $\mathcal{A}$ produced by an h-preserving shrinking strategy, then $h^{\mathcal{A}'} = h^{\mathcal{A}}$, i. e., both abstractions represent the same heuristic. (Of course, $\mathcal{A}$ may nevertheless contain relevant information not contained in $\mathcal{A}'$, leading to a difference in heuristic quality when computing the synchronized product of $\mathcal{A}$ or $\mathcal{A}'$ with another abstraction.) Similarly, g-preserving abstractions preserve distances from the abstract initial state to any abstract state s, which are lower bounds for the distances from the concrete initial state to any concrete state mapped to s by the abstraction.

We are interested in preserving h- and g-values (and thus f-values) because the f-value of an abstract state is a lower bound on the f-values associated with the corresponding nodes in A* search. The A* algorithm expands all search nodes n with $f(n) < L^*$ and *no* search node with $f(n) > L^*$, where L^* is the optimal solution length for the task. Thus, abstract states with high f-values are expected to be encountered less often during search, so combining them is less likely to lead to a loss of important information.

In summary, we are interested in preserving h- and g-values, and we prefer to combine states with high f-values. To achieve these goals, we use the following strategy for selecting the states to combine:

1. Partition all abstract states into *buckets*. Two states are placed in the same bucket iff their g- and h-values are identical. We say that bucket B is *more important* than bucket B' iff the states in B have a lower f-value than the states in B', or if the f-values are equal and the states in B have a higher h-value.
2. If there is any bucket which contains more than one state, select the least important such bucket and combine two of its states, chosen uniformly at random.
3. Otherwise, all buckets contain exactly one state. Combine the states in the two least important buckets.

The third rule usually comes into play only if the target abstraction size M is very low. As long as it does not trigger, the strategy is f-preserving. We remark that the ordering by f-values in the first rule has a very beneficial impact on performance, while the tie-breaking criterion (considering high h-value states more important) is less critical. We have observed similar, but slightly worse performance with the opposite tie-breaking criterion. This concludes the discussion of our abstraction strategy.

Representational Power

We identified some interesting theoretical properties of our framework, taking the form of *representational power* results: Which heuristic quality can be achieved in principle, when suitable abstraction strategies are chosen? Our results reinforce the motivation for linear and f-preserving abstraction strategies, introduced in the previous section. For lack of space, we describe the results informally. Formal proofs will be made available in a long version of the paper.

Comparison to Additive Pattern Databases

We have previously observed that our abstraction heuristics are a generalization of pattern database heuristics: The PDB heuristic h^P of a pattern $P \subseteq \mathcal{V}$ is identical to the heuristic h^{π_P} of the projection π_P.

However, PDB heuristics are not limited to the use of a single pattern P. To achieve better heuristic quality, many patterns are typically considered, which can be combined by taking the maximum of individual heuristic estimates, or, in some cases, by taking their sum. Taking the maximum of individual estimates is of course also possible in our approach if we compute several abstractions (as we will see in the experimental evaluation, this is sometimes beneficial). Considering sums, two patterns P and P' are *additive* iff $h^P + h^{P'} \leq h^*$, i. e., their sum is admissible. In most planning domains, exploiting additive patterns is critical for the succinct representation of high-quality heuristics.

A simple sufficient condition for additivity of patterns, identified by Edelkamp (2001), is that no operator affects variables in both patterns. Under the same condition, any abstractions $\mathcal{A}$ and $\mathcal{A}'$ with *varset*$(\mathcal{A}) \subseteq P$ and *varset*$(\mathcal{A}') \subseteq P'$ are additive, i. e., $h^{\mathcal{A}} + h^{\mathcal{A}'} \leq h^*$ – simply because any abstraction $\mathcal{A}$ is an abstraction of $\pi_{\mathit{varset}(\mathcal{A})}$.

More interestingly, additivity is *automatically captured* by our abstraction approach, provided that we limit ourselves to h-preserving abstractions. Let P and P' be additive patterns, let $\mathcal{A}$ be any h-preserving abstraction of π_P, let $\mathcal{A}'$ be any h-preserving abstraction of $\pi_{P'}$, and let $\mathcal{B}$ be any h-preserving abstraction of $\mathcal{A} \otimes \mathcal{A}'$. *Then $h^{\mathcal{B}}$ dominates the sum of the pattern database heuristics, $h^P + h^{P'}$.* Hence

additivity is captured "automatically" in the sense that, for every sum of concisely representable additive pattern heuristics, there exists a dominating abstraction heuristic that also has a concise representation. The reason is that every operator sequence defining a goal path from some abstract state of $\mathcal{B}$ can be partitioned into two disjoint operator sequences representing corresponding goal paths in $\mathcal{A}$ and $\mathcal{A}'$.

Domain-Specific Quality Bounds

Another way of studying the representational power of a class of heuristics is by considering *domain-specific* quality bounds. In particular, we are interested in domains for which the *perfect heuristic* h^* can be represented as a polynomial-time computable abstraction heuristic.

Clearly, this can only be possible in planning domains which admit polynomial-time optimal solution algorithms. Helmert (2006b) identifies six such domains in the IPC benchmark set, namely GRIPPER, MOVIE, PSR, SCHEDULE, and two variants of PROMELA. *Of these six domains, all but* PSR *have polynomial-time abstraction strategies for computing perfect abstraction heuristics.* (We do not have a positive or negative result for PSR, but believe that no perfect polynomial-sized abstraction heuristics exist.) Moreover, the results still hold when only allowing linear merging and f-preserving shrinking strategies, reinforcing our intuition that these classes of abstraction strategies are useful.

To give an example, a perfect abstraction heuristic for GRIPPER can be obtained as follows. Start with the variables for the robot position and for the gripper hands. Merge these without any shrinking. Then include all ball variables, in an arbitrary order. Combine any two states iff they agree on the status of robot and gripper hands, as well as on the numbers of balls in each room.

In contrast, PDB heuristics (with maximization and summing) cannot succinctly represent perfect heuristics in any of these domains apart from the trivial MOVIE. In the GRIPPER domain, for example, there exists no polynomial-sized family of PDBs which can guarantee a heuristic value of at least $(\frac{2}{3} + \varepsilon)h^*$, for any $\varepsilon > 0$.[2]

Experimental Evaluation

Encouraging theoretical results do not necessarily imply that an algorithm can be made to work well in practice. To evaluate the practical usefulness of our abstraction heuristics, we conducted an empirical study on six domains from the IPC benchmark suite which are known to be hard for optimal planners. We use the abstraction strategy introduced earlier, which we refer to as "LFPA" (for *linear, f-preserving abstraction*) throughout this section. We implemented LFPA within a standard heuristic forward search framework, using the A* algorithm with full duplicate elimination.

There are two guiding questions for our study. Firstly, is a heuristic planner based on our flexible abstraction heuristics competitive with the state of the art in sequentially optimal planning? To answer this question, we compare to two baseline approaches, namely *blind search* (A* with a heuristic function which is 0 for goal states and 1 otherwise) and the $h^{\max}$ heuristic (Bonet & Geffner 2001). These two heuristics were both implemented within the same planning system as LFPA, to allow a fairly unbiased comparison. We also compare to the BFHSP planner (Zhou & Hansen 2006), which was the best-performing sequentially optimal planner at IPC4. Finally, for those benchmarks in our collection which were part of the IPC5 benchmark suite (PIPESWORLD-TANKAGE and TPP), we compare to the official competition results for the participating sequentially optimal planners, FDP (Grandcolas & Pain-Barre 2007) and MIPS-BDD (Edelkamp 2005).

Considering the close relationship of our approach to pattern databases, the second guiding question is: Can our flexible abstraction heuristics result in better planner performance than heuristics based on a carefully selected set of pattern databases? To answer this question, we compare to the automatic pattern selection approach of Haslum et al. (2007), which together with Edelkamp's approach (Edelkamp 2006) defines the state of the art in planning with pattern databases. Here, we are interested in more detailed results, so we do not just compare total runtime, but also preprocessing time for computing the heuristic, number of node expansions during search, and search time.

Both our and the pattern database approach require – or at least significantly benefit from – some parameter tuning. In our case, we manually chose an appropriate value for the abstraction size bound N on a per-domain basis. In two of the six domains, it also proved slightly beneficial to compute several abstractions and use the maximum of their heuristic estimates. In these cases, we computed three abstraction heuristics. (For the second and third abstraction, the merging strategy randomizes the variable order so as to obtain substantially different abstractions in each pass.) The parameter tuning for the pattern selection approach we compare to was also performed on a per-domain basis.

Full experimental results are shown in Table 1.[3] In each domain except LOGISTICS and PSR, all tasks that can be solved by any of the planners are included in the table. For LOGISTICS, we omit four easy tasks solved by all approaches, and for PSR, we only report the five hardest tasks. Empty entries in the table denote tasks that were not solved by the respective technique except for the FDP and MIPS planners in the PIPESWORLD-NOTANKAGE, SATELLITE, LOGISTICS and PSR domains, where no data was available.

Our first observation is that the overall performance of LFPA (left half of Table 1) is excellent. There is only a single instance which can be solved by another planner but not by LFPA (TPP-08 solved by MIPS-BDD), and not a single instance which can be solved by another *heuristic search* planner but not by LFPA, as MIPS-BDD uses symbolic breadth-first search. Many tasks are only solved by LFPA; to the best of our knowledge, this is the first time these tasks are solved by a domain-independent sequentially optimal plan-

[2] Any single pattern may contain only a logarithmic number of balls; and the patterns are not additive if more than one of them contains the robot position variable. Hence, the robot moves are considered for only a logarithmic number of balls.

[3] The experiments were conducted on a machine with a 3.066 GHz CPU, using a 1.5 GB memory limit and 30 minute timeout.

inst.	L^*	LFPA	PDB	blind	$h^{\max}$	BFHSP	FDP	MIPS
PIPESWORLD-NOTANKAGE: $N = 2500$								
01	5	0.07	5.34	0.00	0.00	0.05		
02	12	0.55	6.83	0.02	0.02	0.10		
03	8	1.47	10.44	0.03	0.03	0.15		
04	11	1.24	16.52	0.18	0.19	0.20		
05	8	5.59	31.72	0.19	0.18	0.31		
06	10	5.30	38.56	0.82	0.96	0.58		
07	8	23.37	85.90	1.02	0.61	1.21		
08	10	5.23	194.04	5.97	4.77	1.86		
09	13	10.89	215.89	145.11	88.91	102.33		
10	18	113.18	617.12			1486.97		
11	20	7.22	380.44	7.12	18.46	147.34		
12	24	13.09	568.38	25.50	72.41			
13	16	10.51	600.05	6.13	8.08			
14	30	231.49	1239.93					
15	26	32.15	1433.73	63.10	131.06			
17	22	107.21						
21	14	12.92		2.41	3.05			
23	18	41.76			303.25			
24	24	313.44						
PIPESWORLD-TANKAGE: $N = 1000$								
01	5	1.00	10.59	0.00	0.00	0.15	0.05	1.09
02	12	1.91	20.48	0.01	0.04	0.20	0.41	1.32
03	8	3.99	99.27	1.45	1.60	5.21	2.17	7.98
04	11	9.98	244.80	5.66	17.56	20.44	27.75	17.22
05	8	8.68	162.32	0.34	1.97	5.04	1.27	10.09
06	10	17.43	409.57	1.63	8.17	16.35	8.57	22.78
07	8	25.84		379.66	1295.05		12.41	142.11
08	11	43.70					1092.63	
11	22	26.22	795.34	55.91	436.65			
13	16	70.14						
15	30	161.29						
21	14	52.79		121.27	736.92		375.14	
31	39	50.36		24.53	673.47			
SATELLITE: $N = 10000$ (3 abstractions)								
01	9	0.01	0.27	0.00	0.00	0.03		
02	13	0.08	0.32	0.01	0.01	0.08		
03	11	3.16	2.10	0.30	0.18	0.16		
04	17	6.92	11.54	9.34	6.26	3.08		
05	15	47.74	110.33			119.45		
06	20	21.19	634.77		707.85	265.45		
LOGISTICS: $N = 200000$								
4-0	20	0.10	3.21	0.09	0.06	0.11		
4-1	19	0.09	5.48	0.08	0.04	0.16		
5-0	27	0.87	16.75	1.12	1.03	2.41		
5-1	17	0.88	4.58	0.22	0.09	0.18		
6-0	25	3.65	16.48	5.96	3.34	3.62		
6-1	14	3.85	3.16	0.28	0.06	0.11		
7-0	36	24.56	91.12					
7-1	44	26.84	156.28					
8-0	31	37.09	79.49					
8-1	44	40.77	160.02					
9-0	36	55.47	127.52					
9-1	30	53.42	92.85					
10-0	45	117.46	497.51					
10-1	42	129.97	405.24					
11-0	48	129.82	377.28					
11-1	60	284.91						
12-0	42	185.90	545.76					
12-1	68	221.51						
PSR: $N = 200000$								
29	21	3.47	255.47	2.30	3.66	1.43		
36	22	67.94	1026.39	16.82	27.49	25.15		
40	20	38.29	1309.36	11.91	15.11	7.94		
48	37	36.16	787.37		457.08			
49	47	67.75						
TPP: $N = 50000$ (3 abstractions)								
01	5	0.00	0.01	0.00	0.00	0.02	0.00	0.86
02	8	0.00	0.33	0.00	0.00	0.04	0.00	0.86
03	11	0.01	0.52	0.00	0.00	0.04	0.00	0.86
04	14	0.04	1.74	0.00	0.00	0.07	0.03	0.90
05	19	5.83	3.44	0.24	0.24	0.74	1.59	0.99
06	25	106.45	173.66		350.84			3.93
07	34	602.66						22.11
08	40							35.07

inst.	L^*	LFPA			PDB		
		T_h	nodes	T_s	T_h	nodes	T_s
PIPESWORLD-NOTANKAGE: $N = 2500$							
01	5	0.07	0	0.00	5.32	0	0.02
02	12	0.54	598	0.01	6.81	0	0.02
03	8	1.47	7	0.00	10.40	10	0.04
04	11	1.20	2093	0.04	16.46	158	0.06
05	8	5.58	88	0.01	31.64	9	0.08
06	10	5.25	1483	0.05	38.48	16	0.08
07	8	23.36	203	0.01	85.78	0	0.12
08	10	5.20	475	0.03	193.92	0	0.12
09	13	6.12	128236	4.77	215.69	0	0.20
10	18	8.09	3002505	105.09	616.82	457	0.30
11	20	4.49	186040	2.73	379.14	9360	1.30
12	24	4.27	638241	8.82	564.35	40470	4.03
13	16	10.19	18946	0.32	599.19	862	0.86
14	30	6.04	13554766	225.45	1215.31	192184	24.62
15	26	18.95	723424	13.20	1427.60	36387	6.13
17	22	16.90	3960903	90.31			
21	14	12.75	7607	0.17			
23	18	31.19	351982	10.57			
24	24	28.53	9399079	284.91			
PIPESWORLD-TANKAGE: $N = 1000$							
01	5	1.00	0	0.00	10.55	0	0.04
02	12	1.89	695	0.02	20.44	0	0.04
03	8	3.84	1522	0.15	98.92	15	0.35
04	11	8.59	27933	1.39	244.31	791	0.49
05	8	8.52	584	0.16	161.94	13	0.38
06	10	17.03	2458	0.40	409.17	27	0.40
07	8	24.99	2066	0.85			
08	11	23.41	113187	20.29			
11	22	6.21	894491	20.01	771.20	69689	24.14
13	16	24.78	991522	45.36			
15	30	18.14	6177384	143.15			
21	14	49.58	51569	3.21			
31	39	21.54	1358979	28.82			
SATELLITE: $N = 10000$ (3 abstractions)							
01	9	0.01	0	0.00	0.26	28	0.01
02	13	0.08	0	0.00	0.30	324	0.02
03	11	3.16	0	0.00	1.87	2243	0.23
04	17	6.92	0	0.00	8.47	19523	3.07
05	15	32.14	86958	15.60	14.28	264551	96.05
06	20	15.20	45809	5.99	14.57	1435962	620.20
LOGISTICS: $N = 200000$							
4-0	20	0.10	0	0.00	3.20	0	0.01
4-1	19	0.09	0	0.00	5.47	0	0.01
5-0	27	0.87	0	0.00	16.74	0	0.01
5-1	17	0.88	0	0.00	4.56	0	0.02
6-0	25	3.65	0	0.00	16.47	0	0.01
6-1	14	3.85	0	0.00	3.14	0	0.02
7-0	36	24.56	0	0.00	90.97	832	0.15
7-1	44	26.84	0	0.00	153.09	17258	3.19
8-0	31	37.08	0	0.01	79.33	832	0.16
8-1	44	40.77	0	0.00	158.83	4160	1.19
9-0	36	55.47	0	0.00	127.28	1120	0.24
9-1	30	53.41	0	0.01	92.81	0	0.04
10-0	45	108.57	0	8.89	480.88	40288	16.63
10-1	42	105.66	0	24.31	384.78	58720	20.46
11-0	48	128.95	0	0.87	354.02	55072	23.26
11-1	60	128.55	2608466	156.36			
12-0	42	169.61	0	16.29	522.71	53552	23.05
12-1	68	164.63	0	56.88			
PSR: $N = 200000$							
29	21	3.47	0	0.00	255.43	124	0.04
36	22	67.80	1061	0.14	1026.20	139	0.19
40	20	37.87	3171	0.42	1308.62	1709	0.74
48	37	36.16	0	0.00	787.09	1623	0.28
49	47	62.13	233779	5.62			
TPP: $N = 50000$ (3 abstractions)							
01	5	0.00	0	0.00	0.00	5	0.01
02	8	0.00	0	0.00	0.33	8	0.00
03	11	0.01	0	0.00	0.52	11	0.00
04	14	0.04	0	0.00	1.73	18	0.01
05	19	5.83	0	0.00	3.10	6790	0.34
06	25	98.79	12121	7.66	11.96	1288743	161.70
07	34	150.46	3610376	452.20			
08	40						

Table 1: Left: Runtimes of optimal planners across the test domains. Column *inst.* denotes problem instance, column L^* optimal solution length. Other columns denote runtimes of different sequentially optimal heuristics and planning systems, including our *LFPA* heuristic. (For FDP and MIPS, results are only available for PIPESWORLD-TANKAGE and TPP.)
Right: Detailed comparison between abstraction and pattern database (PDB) heuristics. T_h is the time to compute the heuristic; *nodes* is number of nodes expanded to prove the optimal lower bound; T_s is search time.
Each domain is shown with the abstraction size bound N and the number of abstractions computed, if larger than 1.

ner. The only approach that appears generally competitive is the pattern database heuristic.

For the pattern database approach, it is worth looking at the data in a bit more detail (right half of Table 1). In general, LFPA tends to outperform PDB with respect to runtime and number of instances solved.

Interestingly, we can identify two groups of domains exhibiting different behaviour. In the PIPESWORLD domains, LFPA solves significantly more instances, but on the set of commonly solved instances, PDB tends to require much fewer node expansions. In contrast, in SATELLITE, LOGISTICS, and TPP, LFPA solves the same or only a few more instances, but expands significantly fewer nodes (in PSR the behaviour is somewhat mixed).

While these observations appear somewhat contradictory, they have a common explanation. Using only projections, PDBs allow abstract spaces to be more compactly represented, and therefore they allow the use of larger abstractions. But there comes a point when this compactness is not enough to capture even the smallest projection that would be required to obtain a useful heuristic. The LFPA abstractions, on the other hand, have to be smaller (much smaller, in some cases), but are better suited to capture "some but not all" of the relevant information.

Depending on the nature of the domain, the above has different effects. In the PIPESWORLD domains, the LFPA abstractions have to be very small to be feasible (note in Table 1 that $N = 2500$ and $N = 1000$ for these two domains). Hence PDB does much better on small problems, exploiting large abstractions giving near-perfect heuristics. In larger problems, however, the relevant projections of the state space become too large to be adequately captured by PDBs, so that search performance quickly deteriorates. This deterioration happens a bit more slowly for the LFPA approach, which can still capture some relevant information despite its small abstractions.[4]

In SATELLITE, LOGISTICS, and TPP, it appears that good PDB heuristics require huge abstractions even in small problems. This is easiest to see in SATELLITE, where each goal ("having a picture") corresponds to a large disjunction of options ("take picture with camera A on satellite X", "take picture with camera B on satellite Y", ...): unless a pattern contains all these options, there is always a short-cut in the corresponding PDB (an option whose critical preconditions have been abstracted away). LFPA can ameliorate these difficulties by combining options that are equivalent (cf. the perfect abstraction heuristic for GRIPPER); but this is not enough to scale much further than PDB heuristics.

We would overstate our results if we claimed that LFPA is *superior* to a well-chosen PDB heuristic. But we can certainly observe that our approach is competitive – or more than that – with the current state of the art for automatically derived PDBs, and optimal sequential planning in general.

[4]It should be noted that, when the PDB approach fails, it is often due to the pattern selection technique taking too long. If we extend the timeout to two hours, the PDB algorithm can solve four additional instances in PIPESWORLD-NOTANKAGE and five additional instances in PIPESWORLD-TANKAGE, closely approaching the number of tasks solved by LFPA in these domains.

Conclusion

Planning heuristics based on abstract transition graphs are a powerful tool for optimal sequential planning. They share the strengths of pattern database heuristics while alleviating the combinatorial explosion problem that arises as the number of variables in a pattern increases.

In the context of this general framework, we described a concrete strategy, LFPA, which generates abstractions of a particular kind. We showed that the representational power of the framework in general exceeds that of PDB heuristics, and that an implementation of LFPA is very competitive with the state of the art in sequentially optimal planning.

Significant improvements are still possible. In particular, our current shrinking strategy only takes into account graph *distances*, completely ignoring graph *labels* in the decision of how to simplify an abstraction. Developing more elaborate abstraction strategies thus remains an exciting topic for future research.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed model checking with distance-preserving abstractions. In *Proc. SPIN-2006*, 19–34.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. ECP 2001*, 13–24.

Edelkamp, S. 2005. External symbolic heuristic search with pattern databases. In *Proc. ICAPS 2005*, 51–60.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In *Proc. MoChArt-2006*, 121–135.

Grandcolas, S., and Pain-Barre, C. 2007. Filtering, decomposition and search space reduction for optimal sequential planning. In *Proc. AAAI 2007*.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI 2007*, 1007–1012.

Helmert, M. 2006a. The Fast Downward planning system. *JAIR* 26:191–246.

Helmert, M. 2006b. *Solving Planning Tasks in Theory and Practice*. Ph.D. Dissertation, Albert-Ludwigs-Universität Freiburg.

Hoffmann, J.; Sabharwal, A.; and Domshlak, C. 2006. Friends or foes? An AI planning perspective on abstraction and search. In *Proc. ICAPS 2006*, 294–303.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. Satplan: Planning as satisfiability. In *IPC-5 planner abstracts*.

Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.

Zhou, R., and Hansen, E. A. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170(4–5):385–408.