Research Note

BDD Ordering Heuristics for Classical Planning

Peter Kissmann
KHSMANN@CS.UNI-Saarland.DE

Jörg Hoffmann
HOFFMANN@CS.UNI-Saarland.DE

Saarland University, Saarbrücken, Germany

Abstract

Symbolic search using binary decision diagrams (BDDs) can often save large amounts of memory due to its concise representation of state sets. A decisive factor for this method's success is the chosen variable ordering. Generally speaking, it is plausible that dependent variables should be brought close together in order to reduce BDD sizes. In planning, variable dependencies are typically captured by means of causal graphs, and in preceding work these were taken as the basis for finding BDD variable orderings. Starting from the observation that the two concepts of “dependency” are actually quite different, we introduce a framework for assessing the strength of variable ordering heuristics in sub-classes of planning. It turns out that, even for extremely simple planning tasks, causal graph based variable orders may be exponentially worse than optimal.

Experimental results on a wide range of variable ordering variants corroborate our theoretical findings. Furthermore, we show that dynamic reordering is much more effective at reducing BDD size, but it is not cost-effective due to a prohibitive runtime overhead. We exhibit the potential of middle-ground techniques, running dynamic reordering until simple stopping criteria hold.

1. Introduction

Finding good variable orderings is an important task in many areas of Artificial Intelligence, such as constraint satisfaction problems (CSPs), SAT, and planning (for some heuristic search approaches, but especially when applying symbolic search). In many cases, an efficient ordering is determined by evaluating a graphical representation of the underlying problem. For CSPs, for example, the constraint graph can be used to determine a variable ordering for backtracking-based approaches. Typical approaches take the minimum width (Freuder, 1982), maximum degree, or maximum cardinality (Dechter & Meiri, 1989) of nodes in the constraint graph into account. An alternative approach considers the bandwidth of the constraint graph under a given ordering, which is the maximum distance in that ordering of any two nodes that are adjacent in the graph; the idea is to find an ordering that minimizes the bandwidth (Zabih, 1990).

In SAT, a widely used approach to determine the variable order in conflict-driven clause learning (CDCL) is variable state independent decaying sum (VSIDS) (Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001). This is based on the weights of propositional variables, i.e., how often such a variable occurs in the clauses. Recently, Rintanen (2012) noted that for applying SAT solvers to planning tasks, a different ordering might be more efficient, giving better coverage in the typical benchmarks of the international planning competition (IPC). This ordering takes the structure of planning tasks into account, trying to support (sub)goals as early on as possible.

In planning, variable dependencies are typically represented by the causal graph (e.g., Knoblock, 1994; Jonsson & Bäckström, 1995; Brafman & Domshlak, 2003; Helmert, 2006), capturing variable dependencies in terms of co-occurrences in action descriptions. This kind of graph has turned out
to be useful for a great variety of purposes, including problem decomposition (Knoblock, 1994), system design (Williams & Nayak, 1997), complexity analysis (e.g. Jonsson & Bäckström, 1995; Domshlak & Dinitz, 2001; Brafman & Domshlak, 2003; Katz & Domshlak, 2008; Giménez & Jonsson, 2008; Chen & Giménez, 2010), derivation of heuristic functions (Helmert, 2004, 2006), and search topology analysis (Hoffmann, 2011b, 2011a). For our purposes here, the causal graph’s most relevant application is the derivation of variable orderings. That has been done for BDDs, to which we return in detail below, as well as for merge-and-shrink heuristics (Helmert, Haslum, & Hoffmann, 2007; Helmert, Haslum, Hoffmann, & Nissim, 2014). In merge-and-shrink, a complete variable ordering corresponds to a (linear) merging strategy, an order in which variables are merged into a global abstraction. In a recent extension to non-linear merging strategies (Sievers, Wehrle, & Helmert, 2014), the order of the merges is instead given by a tree. Such a merge tree bears some similarity to the concept of vtrees, which are used as a generalization of variable orderings for sentential decision diagram (SDDs) (Darwiche, 2011). Fan, Müller, and Holte (2014) have shown that efficient merge trees can be determined by means of the causal graph. To do so, they use Min-Cuts in the causal graph, putting the two resulting sets of variables into two different branches of the merge tree and recursively continue in the subgraphs.

In this paper, we are concerned with symbolic search based on binary decision diagrams (BDDs) (Bryant, 1986) for optimal planning. A variable ordering here refers to the order in which variables are queried within the BDDs, a key ingredient for the practical efficiency of the approach. In planning, not much work has been invested into finding good variable orderings, but in model checking, where symbolic search originated (McMillan, 1993), many different variable ordering schemes have been proposed in the past (e.g., Malik, Wang, Brayton, & Sangiovanni-Vincentelli, 1988; Minato, Ishiura, & Yajima, 1990). Again, many of those are based on the evaluation of a graphical representation of the problem. Often, bringing “dependent variables” close together results in smaller BDDs. This can be straightforwardly applied to planning, by defining “variable dependencies” via the causal graph. That is exactly how Gamer, a state-of-the-art symbolic search planner, determines its variable ordering (Kissmann & Edelkamp, 2011).

The starting point of our investigation is a feeling of discomfort with the double use of the word “dependency” in the above. In causal graphs, such a dependency means that the corresponding variables appear in at least one common action, so changing the value of one variable may require changing the other variable as well. BDDs, on the other hand, represent Boolean functions \( \varphi \). If many assignments to a subset \( P \) of all variables immediately determine the truth value of \( \varphi \), independently of the value of the other variables, then the variables in \( P \) should be grouped closely together. In planning, \( \varphi \) typically represents a layer of states sharing the same distance to the initial state (forward search) or the goal (backward search). So the concept of “dependence” here relates to determining whether or not a state is a member of such a layer. What, if anything, does this have to do with causal graph dependencies?

We do not have a conclusive answer to that question, but we contribute a number of insights suggesting that the two concepts of dependence do not have much in common. We consider the issue from both a theoretical and a practical perspective. On the theoretical side, we introduce a simple formal framework for assessing the strength of variable ordering heuristics in sub-classes of planning. Applying that framework to causal graph based variable orders, we show that these may be exponentially worse than optimal orderings, even for extremely simple planning tasks.

On the practical side, we experiment with a wide range of variable ordering schemes, several ones based on the causal graph, and also a range of techniques adapted from the model checking
literature. To get an idea of “how good” these ordering schemes are, on the grand scale of things, we use an upper and a lower “delimiter”. For the latter, we use random variable orderings. Not too surprisingly, most ordering schemes are better than random; surprisingly, not all of them are. Indeed, Fast Downward’s “level” heuristic (Helmert, 2006) turns out to be much worse than the average random BDD variable ordering.

As the upper delimiter, we employ dynamic reordering techniques that minimize BDD size online, during the construction process. Compared to static up-front variable ordering schemes, such reordering has a much better basis for taking decisions, but is much more time-consuming. It is thus expected for the BDD size results to be much better. The extent to which that happens in our experiments is remarkable, however: Static orderings are hardly ever even a tiny bit better, whereas the advantage of dynamic reordering easily and frequently goes up to three orders of magnitude.

While it has been successfully employed in at least one domain in non-deterministic planning (Cimatti, Pistore, Roveri, & Traverso, 2003), dynamic reordering usually is prohibitively slow and not cost-effective. Still, its prowess at reducing BDD size, combined with our pessimistic outlook on static ordering schemes, suggests that it may be the better alternative. An initial experiment indicates that this could, indeed, be the case: With very simple adaptive stopping criteria, running dynamic reordering only up to a certain point, we obtain better results than with any of the static ordering schemes.

The remainder of the paper is organized as follows. Section 2 gives the necessary background on our planning framework and the use of BDDs. Section 3 introduces our theoretical framework and investigates the properties of causal graph based ordering schemes in a range of well-known planning sub-classes. Section 4 presents our experiments regarding the quality of causal graph based ordering schemes, and Section 5 presents our experiments with adaptive stopping criteria for dynamic reordering. Section 6 concludes the paper with a brief discussion and outlook.

This research note is an extension of the authors’ previous short conference paper (Kissmann & Hoffmann, 2013). The present paper contains comprehensive details regarding the technical background and the variable orderings we implemented, and it includes full proofs. The experiments with adaptive stopping criteria for dynamic reordering, Section 5, are new.

2. Background

For BDD-based planning, as argued e.g. by Edelkamp and Helmert (1999), it is important to have a small encoding of the given planning task. So we use a finite-domain variable representation as the basis for our investigation. A finite-domain representation (FDR) planning task is a tuple \( \Pi = (V, A, I, G) \), where \( V \) is the set of state variables and each \( v \in V \) is associated with its finite domain \( D(v) \). \( A \) is a finite set of actions where each \( a \in A \) is a pair \( \langle \text{pre}_a, \text{eff}_a \rangle \) of partial assignments to \( V \) with \( \text{pre}_a \) being the precondition and \( \text{eff}_a \) the effect of action \( a \). The initial state \( I \) is a complete assignment to \( V \). The goal \( G \) is a partial assignment to \( V \). By \( V(pa) \), for a partial assignment \( pa \), we denote the variables \( v \in V \) where \( pa(v) \) is defined.

An action \( a \in A \) is applicable in a state \( s \) iff \( \text{pre}_a \subseteq s \). For the resulting successor state \( s' \) it holds that \( s'(v) = \text{eff}_a(v) \) for all \( v \in V(\text{eff}_a) \) and \( s'(v) = s(v) \) for all \( v \in V \setminus V(\text{eff}_a) \). A plan is a sequence of actions whose successive application starting in the initial state results in a state \( s_g \) with \( G \subseteq s_g \). A plan is optimal if no plan of shorter length exists.

Binary decision diagrams (BDDs) as introduced by Bryant (1986) represent Boolean functions \( \varphi \). A BDD \( \beta \) is a directed acyclic graph with one root and two terminal nodes, the 0-sink and the
1-sink. Each internal node corresponds to a binary variable \( p \) and has two successors, one following the high edge taken if \( p \) is true and one following the low edge taken if \( p \) is false. For any assignment to all variables the sink reached corresponds to the value of the function \( \varphi \) represented by \( \beta \).

As is common in practice, here we use reduced ordered BDDs. An ordered BDD (OBDD) is a BDD in which the ordering of the binary variables on any path is fixed. A reduced OBDD applies two reduction rules to result in a canonical representation: (i) remove any node with identical successor along the high and the low edge; (ii) merge nodes of the same variable that have the same successor along the high edge and the same successor along the low edge. Figure 1 illustrates example BDDs for the function \( \varphi = ((x_1 \land x_2) \lor \neg x_3) \) with the ordering \( \langle x_1, x_2, x_3 \rangle \). In Figure 1a we have the full OBDD without any reduction. When considering the nodes for \( x_3 \), we note that the rightmost one can be removed due to rule (i), and the other three can be merged due to rule (ii). Applying these rules to preceding layers as well, we end up with the reduced OBDD in Figure 1b.

We consider BDD-based planning in terms of symbolic search (McMillan, 1993) as implemented in Gamer (Kissmann & Edelkamp, 2011). The finite-domain variables \( V \) of the FDR task are encoded by replacing each \( v \in V \) with a binary counter \( \gamma(v) \) using \( \lceil \log_2 |D(v)| \rceil \) bits. For a task representable by \( n \) bits we need \( 2^n \) BDD variables in two sets, one set \( x \) representing the current state variables, another set \( x' \) representing the successor state variables. Each action \( a \in A \) is represented by a transition relation BDD, \( T_a(x, x') \), which captures the changes due to the application of \( a \) but also the frame, i.e., the variables that do not change:

\[
T_a(x, x') = \text{pre}_a(x) \land \text{eff}_a(x') \land \text{frame}(V \setminus V(\text{eff}_a), x, x')
\]

with \( \text{frame}(V', x, x') = \bigvee_{v \in V'} v(x) \leftrightarrow v(x') \) modeling the frame. It is possible to create a monolithic transition relation over all actions, i.e., \( T(x, x') = \bigvee_{a \in A} T_a(x, x') \). However, this typically is not feasible in terms of memory. Thus, we store the transition relations of all actions separately (Burch, Clarke, & Long, 1991).

In order to calculate the successors of a set of states \( S \), represented in the current state variables, we use the image function

\[
\text{image}(S) = \bigvee_{a \in A} \exists x. (S(x) \land T_a(x, x'))[x' \leftrightarrow x].
\]
The conjunction makes sure that only applicable actions are considered, and sets the corresponding successor state variables. The existential quantification removes the current state variables. The operator \( [x' \leftrightarrow x] \) stands for a swapping of the current and successor state variables, so that in the end the successor states are again represented in the current state variables, i.e., they are the new current states. Finally, the disjunction ensures that the successors based on all actions are calculated.

For the case of backward search, the pre-image calculating the predecessors of a set given in the successor variables looks similar, only that the successor state variables are quantified instead of the current state variables.

Using these two functions, symbolic breadth-first search is straightforward: Starting at the initial state (or the set of goal states), iterate the image (or the pre-image), until a goal (or the initial state) is reached. Storing the entire set of reached states we can ensure completeness. During search, each layer \( L \) of states – a subset of states with identical distance to the initial state (forward search) or the goal (backward search) – is then represented by a BDD for its characteristic function.

Based on a given variable ordering, the size of the BDD, i.e., the number of nodes needed to represent the corresponding function, can differ exponentially, so that finding good orderings is crucial in practice. As the size also has an influence on the runtime (e.g., the time and memory requirements for the conjunction of two BDDs is polynomial in the product of the sizes of the two BDDs), smaller size is important not only in terms of memory but also in terms of runtime. BDD packages typically contain dynamic reordering algorithms, which can reduce the BDD sizes based on the current situation. However, as previous work has argued (Kissmann & Edelkamp, 2011), and as our experiments here reconfirm, the runtime overhead of dynamic reordering is prohibitive in planning. The alternative is to use static variable ordering schemes instead. We define such schemes as functions \( \Omega \) mapping any planning task \( \Pi \) to a non-empty set \( \Omega(\Pi) \) of variable orderings, i.e., orderings of the planning task’s finite-domain variables \( V \). We use a set \( \Omega(\Pi) \) here, as opposed to a single ordering, because the variable ordering schemes we consider here contain ambiguity, i.e., they impose only some constraints on the final variable ordering as opposed to fixing a unique complete ordering.

Before the first BDD is created, the set of possible orderings is determined in a pre-processing step, and the actual ordering \( \langle v_1, \ldots, v_n \rangle = o \in \Omega(\Pi) \) is chosen arbitrarily (i.e., we do not consider this step here). The calculated ordering is defined over the set of multi-valued variables. Thus, to get the final BDD binary variable order we replace each finite-domain variable \( v_i \) in \( o \) with its binary counter \( \gamma(v_i) \). This means that the BDD treats these counters like inseparable fixed blocks. (Note that the bits of the counters are not represented at the level of the planning tasks \( \Pi \), so that it is impossible for \( \Omega \) to make an informed choice for a separation of such a block.) In addition to these blocks we store the current and successor state variables in an interleaved fashion (Burch, Clarke, Long, McMillan, & Dill, 1994).

For any layer \( L \) and ordering \( o \) of the planning task’s finite-domain variables, the ordered BDD is unique. We denote its size, i.e., the number of nodes, by \( BDDSize(o, L) \). By \( BDDSize^*(L) := \min_o BDDSize(o, L) \) we denote the size of the BDD for an optimal variable ordering. Finding such an optimal ordering is NP-hard (Bryant, 1986).

The state of the art ordering scheme in symbolic planning is based on the causal graph \( CG_{\Pi} \) of the planning task (Knoblock, 1994; Domshlak & Dinitz, 2001). \( CG_{\Pi} \) is a directed graph with nodes \( V \) and an arc \( (v, v') \) iff \( v \neq v' \) and there exists an action \( a \in A \) such that \( (v, v') \in V(\text{eff}_a) \cup V(\text{pre}_a) \times V(\text{eff}_a) \). In other words, we have an arc from \( v \) to \( v' \) if both appear as an effect of some action or \( v \) appears in the precondition of an action that has \( v' \) in its effect.
Gamer’s scheme, denoted $\Omega^g$, maps $\Pi$ to the set of orderings $o = \langle v_1, \ldots, v_n \rangle$ that minimize the expression $\sum_{(v_i, v_j) \in CG_\Pi} (i - j)^2$. The idea is that variables $v_i, v_j$ that are adjacent in the CG are dependent and should be brought close together in the ordering by minimizing their distance $|i - j|$. This bears some similarity to the minimal bandwidth variable ordering in CSPs (Zabih, 1990), though there the maximum of the distances is to be minimized, while we minimize the sum.

In practice, Gamer approximates $\Omega^g$ by a limited amount of local search in the space of orderings, as finding an optimal solution is NP-hard (Kissmann & Edelkamp, 2011). For this, it starts several searches with a random ordering, swaps two variables and checks if the sum decreased. If it did, the search continues with the new ordering, otherwise it will stick to the old one. In the end, the generated ordering with the smallest sum is used. The original hope was that there is a connection between the two notions of dependency. This was supported by the fact that the new ordering resulted in improved coverage in the used benchmark set compared to what was used before.

Apart from $\Omega^g$, we also consider the scheme $\Omega^c$, which is only defined for an acyclic $CG_\Pi$. It maps $\Pi$ to the set of topological orderings of the nodes in $CG_\Pi$. We consider this to be of theoretical interest since it is the straightforward way to “trust the causal graph completely”, i.e., to take the dependencies as derived from the causal graph and order the BDD variables accordingly.

3. What’s in a Causal Graph: Theory

As we have pointed out in the introduction, it is doubtful whether the concept of dependency in the causal graph has any real relation with the concept of dependency relevant to BDD size. We now frame this in terms of a classification of the guarantees offered, or rather, the guarantees not offered, by $\Omega^g$ and $\Omega^c$ in restricted classes of planning tasks.

We first introduce our theoretical framework, then outline our results for $\Omega^c$ and $\Omega^g$.

3.1 Classification Framework

We classify ordering schemes, relative to a given scalable family of planning tasks, as follows:

**Definition 1** (Classification of Ordering Schemes). Let $F = \{\Pi_n\}$ be an infinite family of FDR planning tasks parameterized by $n$, where the size of $\Pi_n$ is bounded by a polynomial in $n$. Let $d \in \{\text{forward, backward}\}$ be a search direction. A variable ordering scheme $\Omega$ is:

(i) **perfect** in $F$ for $d$ if for all $\Pi_n \in F$, all $d$-layers $L$ in $\Pi_n$, and all $o \in \Omega(\Pi_n)$, we have $BDDSize(o, L) = BDDSize^*(L)$.

(ii) **safe** in $F$ for $d$ if there exists a polynomial $p$ s.t. for all $\Pi_n \in F$, all $d$-layers $L$ in $\Pi_n$, and all $o \in \Omega(\Pi_n)$, we have $BDDSize(o, L) \leq p(BDDSize^*(L))$.

(iii) **viable** in $F$ for $d$ if there exists a polynomial $p$ s.t. for all $\Pi_n \in F$ and all $d$-layers $L$ in $\Pi_n$, there exists $o \in \Omega(\Pi_n)$ with $BDDSize(o, L) \leq p(BDDSize^*(L))$.

In other words, a perfect $\Omega$ guarantees to deliver only optimal orderings, a safe $\Omega$ guarantees at most polynomial overhead, and a viable $\Omega$ always delivers at least one good ordering but runs the risk of super-polynomial overhead. If $\Omega$ is not viable, then it actively deceives the planner, in the sense that all variable orderings suggested are super-polynomially bad in some task and layer. Note that our interpretation of “viability” is generous in that, while at least one good ordering must be delivered, that ordering may differ for different search directions and layers, so that the
disambiguation over $\Omega(\Pi_n)$ is left with the job of determining which ordering actually is the good one. One could define this notion more strictly, but as our results will be negative anyhow we stick to this optimistic version.

We extend the classification to arbitrary sub-classes $\mathcal{C}$ of FDR (whose sizes still are bounded by a polynomial) by the worst case over all families $\mathcal{F}$ contained in $\mathcal{C}$: if $\mathcal{C}$ contains at least one $\mathcal{F}$ where $\Omega$ is not perfect, then $\Omega$ is not perfect in $\mathcal{C}$; if $\mathcal{C}$ contains at least one $\mathcal{F}$ where $\Omega$ is not safe, then $\Omega$ is not safe in $\mathcal{C}$; if $\mathcal{C}$ contains at least one $\mathcal{F}$ where $\Omega$ is not viable, then $\Omega$ is not viable in $\mathcal{C}$.

As we are interested in variable orderings derived from the causal graph, it is natural to consider sub-classes of FDR characterized by their causal graphs. For a set of directed graphs $G$, by $FDR(G)$ we denote the class of FDR planning tasks whose causal graphs are elements of $G$. We investigate some widely considered causal graph special cases, namely:

- **Chains** ($G^{\text{chain}}$), where we can find an order $x_1, \ldots, x_n$ of the variables so that there are only arcs from each $x_i$ to $x_{i+1}$ for $1 \leq i \leq n - 1$ (cf. Figure 2a).
- **Forks** ($G^{\text{fork}}$), where we have one variable $x$, and a set of variables $g_i$, with an arc from $x$ to each $g_i$ (cf. Figure 2b).
- **Inverted forks** ($G^{\text{ifork}}$), where we have a set of variables $x_i$, and one variable $g$, with an arc from each $x_i$ to $g$ (cf. Figure 2c).
- **Directed acyclic graphs** (DAGs, $G^{\text{dag}}$).

As simple limiting cases, we also consider causal graphs without any arcs ($G^\emptyset$), as well as arbitrary causal graphs ($G^\forall$). Figure 2d illustrates the relations between the cases considered.

Bad cases are inherited in the hierarchy of Figure 2d: if $G \subseteq G'$, then for any ordering scheme the classification within $FDR(G')$ is at least as bad as that in $FDR(G)$, simply because the culprit worst-case (not-perfect/not-safe/not-viable) family $\mathcal{F}$ of FDR planning tasks in $FDR(G)$ will be contained in $FDR(G')$ as well.

### 3.2 Classification Results

We start our investigation with empty causal graphs, i.e., causal graphs with no arcs:
Theorem 1. For both search directions, any ordering scheme is safe in $FDR(G^0)$. $\Omega^{ga}$ and $\Omega^{cg}$ are not perfect.

Proof. If the causal graph has no arcs, then all variables move independently, i.e., each action may have only a single variable in the precondition, and the same variable in the effect. So any forward/backward layer with distance $d$ contains exactly the states in which the sum of individual distances (from a variable’s initial value to a variable’s goal value) equals $d$. For any variable $v$ of the task, the number of vertices (more precisely, of copies of its binary counter $\gamma(v)$) needed is thus bounded by the number of possible individual-distance sums of the variables preceding $v$. Hence BDD size is polynomially bounded regardless of the variable ordering.

To see that $\Omega^{ga}$ and $\Omega^{cg}$ are not perfect, consider the following simple example. We design an FDR task $\Pi_n$ that uses 2 variables $x$ and $y$, each with a domain of size 4, represented by the values $00$, $01$, $10$, and $11$. For forward search, initially $x = 00$ and $y = 00$ holds. For the $x$ variable we have an action setting it to $01$ if it is currently $00$, another setting it to $10$ from $00$, and two setting it to $11$ from $01$ or $10$, respectively. For the $y$ variable we have an action setting it to $01$ if it is currently $00$, another setting it from $01$ to $10$ and another setting it from $10$ to $11$. Thus, for the values of the $x$ variable we have distances of $0$, $1$, $1$, and $2$, respectively, from the initial value of $x$, and for the $y$ variable we have distances of $0$, $1$, $2$, and $3$, respectively, from $y$’s initial value. Figure 3 illustrates the domain transition graphs (DTGs) for variables $x$ and $y$. A similar task having the same distances to the goal values can be defined for backward search.

Each variable is represented by two BDD variables, $x_0$, $x_1$ and $y_0$, $y_1$. If we keep the order within the $x$ and $y$ variables fixed, we have two possible orderings: $x$ before $y$ or vice versa. For a distance of 1 from the initial (or goal) state, we get the BDDs illustrated in Figure 4: Ordering $x$ before $y$ results in a slightly larger BDD. Thus, $\Omega^{ga}$ and $\Omega^{cg}$, which correspond to all possible orderings, are not perfect, which concludes the proof.

Even though the schemes $\Omega^{ga}$ and $\Omega^{cg}$ do not constrain the set of possible orderings in any way, Theorem 1 can be seen as a “good case” for the connection of causal graphs and BDD orderings: Empty causal graphs entail that any ordering is safe. The connection doesn’t seem to carry any further than this trivial case, though: In all other sub-classes considered, the space of BDD orderings contains exponentially bad ones. Indeed, that is true not only for the set of all BDD orderings, but
also for the subsets delivered by $\Omega^{ga}$ and $\Omega^{cg}$. The classification of these schemes is very bad in almost all considered cases, with a little bit of hope only for chain causal graphs.

Our negative results employ Boolean functions in \textbf{quadratic form}. These have the variables $\{x_1, y_1, \ldots, x_n, y_n\}$, and take the form $\bigl( x_1 \oplow y_1 \bigr) \ophi \ldots \ophi \bigl( x_n \oplow y_n \bigr)$, where either $\ophi \in \{\lor, \oplus\}$ and $\oplow = \land$, or vice versa. We denote these functions by $Q(\ophi, \oplow)$. For each of these functions, the ordering $\langle x_1, y_1, \ldots, x_n, y_n \rangle$ (i.e., bringing pairs of $x_i$ and $y_i$ together) yields a BDD whose size is polynomial in $n$, while the ordering $\langle x_1, \ldots, x_n, y_1, \ldots, y_n \rangle$ (i.e., splitting the variables in two blocks, one with all $x$ and one with all $y$ variables) yields a BDD of exponential size. (Wegener, 2000, proves this for $Q(\lor, \land)$ as depicted in Figure 5; similar arguments apply for the other quadratic forms.)

\textbf{Theorem 2.} For both search directions, $\Omega^{ga}$ and $\Omega^{cg}$ are not safe in $\text{FDR}(G_{\text{fork}})$.

\textit{Proof.} To prove the claim for backward search, consider the function $Q(\lor, \land) = \lor_{i=1}^n (x_i \land y_i)$. We design an FDR task $\Pi_n$ that uses $2n + 1$ Boolean variables, $\{g, x_1, y_1, \ldots, x_n, y_n\}$ including an additional variable $g$ that the goal requires to be true. There are $n$ actions achieving $g$, each of which requires a pair $(x_i \land y_i)$ to be true as the precondition. Clearly, $\Pi_n \in \text{FDR}(G_{\text{fork}})$. The backward layer with a distance of 1 from the goal is characterized by $\neg g \land \lor_{i=1}^n (x_i \land y_i)$.

An optimal ordering for $Q(\lor, \land)$ consists of pairs of $(x_i, y_i)$ or $(y_i, x_i)$. Adding the $g$ variable, an optimal ordering places it either at the front (as depicted in Figure 6a) or at the end. These cases require exactly one node representing the $g$ variable. Placing the $g$ variable anywhere else requires as many nodes representing $g$ as there are nodes (different from the 0-sink) reached by edges passing through that layer. In this case, there are two $g$ nodes if $g$ is placed between two pairs, and up to three nodes if it is placed between two nodes constituting a pair (see Figure 6b for the latter case).

Any ordering following $\Omega^{ga}(\Pi_n)$ places $g$ in the middle and the $x$ and $y$ variables in an arbitrary order around it. Any ordering following $\Omega^{cg}(\Pi_n)$ places $g$ at the end and the $x$ and $y$ variables in an arbitrary order before it. In both cases, all $x$ variables may be placed before all $y$ variables, resulting in an exponential overhead which concludes the proof for backward search.
(a) Good variable ordering: \(\langle x_1, y_1, x_2, y_2, x_3, y_3 \rangle\)

(b) Bad variable ordering: \(\langle x_1, x_2, x_3, y_1, y_2, y_3 \rangle\)

Figure 5: BDDs with different variable orderings for \(Q(\lor, \land)\) with \(n = 3\): \((x_1 \land y_1) \lor (x_2 \land y_2) \lor (x_3 \land y_3)\). Solid arrows denote high edges, dashed ones low edges.

(a) \(g\) at the front.

(b) \(g\) within a pair.

Figure 6: BDDs representing \(\neg g \land Q(\lor, \land)\) with different positions of the \(g\) variable. Solid arrows represent high edges, dashed ones low edges.

For forward search, we consider the same function \(Q(\lor, \land)\), and construct \(\Pi_n\), which has the same variables \(\{g, x_1, y_1, \ldots, x_n, y_n\}\) but where the domains of \(\{x_1, y_1, \ldots, x_n, y_n\}\) are ternary: unknown, true (\(\top\)), or false (\(\bot\)). All \(x\) and \(y\) variables are initially unknown, and can be set to either true or false if they are currently unknown. There are \(n\) actions achieving \(g\), exactly as above. Then
in the states with initial state distance $2n + 1$ all $x$ and $y$ variables are either true or false and the states are exactly those that satisfy $g \land Q(\land, \lor) = g \land \bigwedge_{i=1}^{n} (x_i = \top) \land (y_i = \top)$. As the causal graph remains unchanged, the set of possible orderings following $\Omega^{ga}(\Pi_n)$ and $\Omega^{cg}(\Pi_n)$ remains the same as in backward search as well, so that again some orders result in exponential overhead which concludes the proof for forward search.

Note that, while the proof construction shows that some orders possible in $\Omega^{ga}$ and $\Omega^{cg}$ are super-polynomially bad, other possible orders are good. Hence, while as claimed we prove that $\Omega^{ga}$ and $\Omega^{cg}$ are not safe in $FDR(G_{fork})$, it might be the case that $\Omega^{ga}$ and $\Omega^{cg}$ are viable in $FDR(G_{fork})$. We leave this as an open question.

**Theorem 3.** For both search directions, $\Omega^{ga}$ and $\Omega^{cg}$ are not safe in $FDR(G_{fork})$.

**Proof.** For both search directions, we use the same function $Q(\land, \lor) = \bigwedge_{i=1}^{n} (x_i \oplus y_i)$, and the same FDR task $\Pi_n$ with Boolean variables $\{x_1, y_1, \ldots, x_n, y_n\}$ plus an additional variable $z$ with domain $\{d_1, dx_1, dy_1, d_2, dx_2, dy_2, \ldots, d_n, dx_n, dy_n, d_{n+1}\}$. The actions are such that, for $1 \leq i \leq n$, $z$ can move from $d_i$ to either $dx_i$ or $dy_i$, and from each of these to $d_{i+1}$ (see Figure 7). An action preconditioned on $dx_i$ achieves $x_i$, an action preconditioned on $dy_i$ achieves $y_i$. Initially, $z = d_1$ and all $x_i, y_i$ are false. The goal requires that $z = d_{n+1}$ and all $x_i, y_i$ are true. In forward search, the states with initial state distance $3n$ are exactly those where $z = d_{n+1}$ and $Q(\land, \lor)$ is true, and in backward search the states with goal state distance $3n$ are exactly those where $z = d_1$ and $Q(\land, \lor)$ is true.

Any ordering following $\Omega^{ga}(\Pi_n)$ places $z$ in the middle and the $x$ and $y$ variables arbitrarily around it; any ordering following $\Omega^{cg}(\Pi_n)$ places $z$ at the beginning and the $x$ and $y$ variables
arbitrarily after it. Thus, there is no constraint on the variables \( \{ x_1, y_1, \ldots, x_n, y_n \} \), so that placing all \( x \) variables before all \( y \) variables is an ordering compatible with both schemes, and results in exponential overhead.

Again, the proof shows that \( \Omega^{ga} \) and \( \Omega^{cg} \) are not safe, but makes no statement regarding viability. Note also that the task in the proof construction is unsolvable. It is easy to modify the task to be solvable without breaking the proof argument for the forward search direction. We did not investigate whether the same is true of the backward search direction as well. In practice, while proving unsolvability has not traditionally been a popular objective in planning, state space exhaustion is one of the traditional purposes BDDs are deemed to be good for.

For DAG causal graphs, we prove that there are cases where all orderings admitted by \( \Omega^{ga} \) and \( \Omega^{cg} \) are super-polynomially bad:

**Theorem 4.** For both search directions, \( \Omega^{ga} \) and \( \Omega^{cg} \) are not viable in \( FDR(G^{dag}) \).

**Proof.** For the backward search claim, we use the combination of a chain causal graph and an inverted fork as illustrated in Figure 8. We design an FDR task \( \Pi_n \) that uses \( 2n + 1 \) Boolean variables, \( \{ g, x_1, y_1, \ldots, x_n, y_n \} \), including a variable \( g \) that the goal requires to be true. There are \( n \) actions achieving \( g \), each of which requires a pair \( (x_i \land y_i) \) to be true as the precondition (this part of the task is the same as in the proof of Theorem 2). We add actions ensuring that in our two schemes all \( x \) variables will be placed before all \( y \) variables (or vice versa). One action has an empty precondition and sets \( x_1 \) to true in its effect, another one requires \( x_n \) to be true in the precondition and sets \( y_1 \) to true in its effect, the rest have \( x_{i-1} \) (or \( y_{i-1} \)) in the precondition and set \( x_i \) (or \( y_i \)) to true in the effect. All states with a goal distance of \( 1 \) are thus characterized by \( \neg g \land Q(\lor, \land) \).

Any order induced by \( \Omega^{ga} \) places \( g \) in the middle, and either places all \( x \) variables in increasing order before \( g \) and all \( y \) variables in increasing order after \( g \), or places all \( y \) variables in decreasing order before \( g \) and all \( x \) variables in decreasing order after \( g \). \( \Omega^{cg} \) induces an order starting with all \( x \) variables in increasing order, followed by all \( y \) variables in increasing order, followed by \( g \). Thus, in all cases, the \( x \) variables are placed separately from the \( y \) variables, resulting in exponential overhead which proves the claim for the backward search direction.

For forward search we use the same approach as in the proof of Theorem 2, namely to extend the domain of all \( x \) and \( y \) variables to \( \{ \text{true} (\top), \text{false} (\bot), \text{unknown} \} \). All \( x \) and \( y \) variables are initialized to the value unknown. There are \( n \) actions setting \( g \) to true, all requiring a pair of \( (x_i \land y_i) \) to be true. The additional actions are as follows. Two require \( x_1 \) to be unknown and set it to true or false, respectively. Two require \( x_n \) to be true and \( y_1 \) to be unknown and set \( y_1 \) to true or false, respectively. Two require \( x_n \) to be false and \( y_1 \) to be unknown and set \( y_1 \) to true or false, respectively. In the same manner we have four actions for each \( x_i \) and \( y_i \) \((2 \leq i \leq n)\), requiring \( x_{i-1} \) \((y_{i-1})\) to be true respectively false, and requiring \( x_i \) \((y_i)\) to be unknown, and setting \( x_i \) \((y_i)\) to true respectively false. Thus, all states with an initial state distance of \( 2n + 1 \) can be characterized by the function...
BDD ORDERING HEURISTICS FOR CLASSICAL PLANNING

Figure 9: Causal graph for the planning task used in the proof of Theorem 5.

Figure 10: BDDs representing $\bigoplus_{i=1}^{3} y_i \land \bigvee_{i=1}^{3} (x_i \land y_i)$, as used in the proof of Theorem 5. Solid arrows represent high edges, dashed ones low edges.

$$g \land Q(\lor, \land) = g \land \bigvee_{i=1}^{n} (x_i = \top) \land (y_i = \top).$$

The variable orders induced by $\Omega^{ga}$ and $\Omega^{cg}$ are the same as in backward search, resulting in exponential overhead, concluding the proof.

From this we immediately get (recall that $\Omega^{cg}$ is defined only for acyclic causal graphs):

**Corollary 1.** $\Omega^{ga}$ is not viable in $FDR(G^\forall)$.

We close our investigation with a somewhat positive result for chain causal graphs:

**Theorem 5.** For both search directions, $\Omega^{ga}$ and $\Omega^{cg}$ are not perfect in $FDR(G^{\text{chain}})$. There exists an ordering scheme that is not viable in $FDR(G^{\text{chain}})$.

**Proof.** The first part of the claim is inherited from $FDR(G^\emptyset)$, i.e., as a corollary of Theorem 1.

For the second part of the claim, existence of a non-viable ordering scheme, we consider first the backward search direction, using the function $Q(\lor, \land) = \bigvee_{i=1}^{n} (x_i \land y_i)$. We design an FDR task $\Pi_n$ that uses $2n$ Boolean variables, $\{x_1, y_1, \ldots, x_n, y_n\}$. The goal requires all $y$ variables to be false. We have an action without precondition to set $x_1$ to true, actions with preconditions requiring $y_{i-1}$ to be false setting $x_i$ to true, and actions preconditioned on $x_i$ being true setting $y_i$ to false. The causal graph is depicted in Figure 9. Clearly, $\Pi_n \in FDR(G^{\text{chain}})$.

The states with distance 1 from the goal are the ones where all except one $y_i$ are false, and for the single true $y_i$ we have $x_i$ true as well. This is characterized by the formula $\bigoplus_{i=1}^{n} y_i \land Q(\lor, \land) = \bigoplus_{i=1}^{n} y_i \land \bigvee_{i=1}^{n} (x_i \land y_i)$. It is easy to see that the exclusive or part of this formula does not change.
the relevant properties of BDDs for the quadratic form, i.e., we still have orderings with polynomial and other orderings with exponential number of nodes, e.g., those placing all $x$ variables before all $y$ variables (see Figure 10 for illustration). Any ordering scheme including only such latter orderings is not viable.

For the forward search direction case, we construct a planning task where all $x$ and $y$ variables are ternary (unknown, true ($\top$), false ($\bot$)), and are unknown initially. The value of $x_1$ can be set freely; $y_i$ can be set to true or false if $x_i$ is true, and can only be set to true if $x_i$ is false; $x_{i+1}$ can be set freely once $y_i$ has been set to either true or false. In $2n$ steps, we can reach exactly the states characterized by $Q(\land, \lor) = \bigwedge_{i=1}^n (x_i = \top) \lor (y_i = \top)$. A BDD representing $Q(\land, \lor)$ is of exponential size if, e.g., all $x$ variables are placed before all $y$ variables, and any ordering scheme including only such orderings is not viable.

Note that, for both planning task families $\{\Pi_n\}$ just described, both $\Omega^{\mathbb{E}_a}$ and $\Omega^{\mathbb{E}_g}$ and force the $x_i$ and $y_i$ variable to be ordered in pairs, resulting in BDDs of minimal size (see Figure 10a). In that sense, these two planning task families constitute our only truly positive result: Within them, the ordering information in the causal graph keeps us from making exponentially bad mistakes. That positive message would be much stronger if $\Omega^{\mathbb{E}_a}$ and $\Omega^{\mathbb{E}_g}$ were safe for all families of tasks with chain causal graphs. It remains an open question whether that is so.

Figure 11 gives an overview of our results. The evidence speaks rather clearly against a strong connection between causal graph dependencies and dependencies as relevant for BDD size. Note that the causal graph underlying Theorem 4 – non-viability for $FDR(G^{\text{dag}})$ – has a very simple form combining a chain with an inverted fork, and that Theorem 2 – non-safety for $FDR(G^{\text{ifork}})$ – relies on planning tasks which fall into a known syntactically identified tractable class for optimal planning (Katz & Domshlak, 2010). Note also that being “not safe” already is quite bad in practice, incurring an exponential risk unless we have a clever way of choosing an ordering within $\Omega(\Pi)$ (which, at the moment, we do not have).

4. What’s in a Causal Graph: Practice

While we have shown poor worst-case performance of causal graph based variable ordering schemes in theory, practice might be another matter. To assess the latter, we implemented a comprehensive set of causal graph based variable ordering schemes, comprising 12 such schemes in total, and ran them in comparison to practical “good”/“bad” delimiters. As the “bad” delimiter, we used random orderings. As the “good” delimiter, we used the off-the-shelf dynamic reordering algorithm of Gamer’s BDD package CUDD, which is based on sifting (Rudell, 1993).
A few words are in order regarding how sifting works. The variable with the greatest number of nodes in the current BDD is chosen. It is first moved towards the end of the ordering, then towards the beginning of the ordering, by iteratively swapping its position with the next variable in the corresponding direction. Once all positions have been tried, the variable is moved to the position where the BDD size was smallest. This done, the next variable is chosen, until all variables have been processed. For better comparability with our ordering schemes, we restrict the algorithm to keep the variables representing each $\gamma(v)$ together.

As previously indicated, dynamic reordering consumes too much runtime to be cost-effective. In the present experiments, where we are interested only in BDD size, we give dynamic reordering ample runtime. In Section 5, we will identify simple adaptive criteria for stopping dynamic reordering automatically during the search, taking advantage of its size reduction capacity without suffering too much from its runtime consumption.

We ran the benchmarks of the 2011 International Planning Competition (IPC’11), and we used Gamer as the base implementation for all planners, running them on one core of an Intel Xeon X5690 CPU with 3.47 GHz. Unless otherwise stated, we used the IPC’11 settings, namely a timeout of 30 minutes and a memory limit of 6 GB.

4.1 Ordering Schemes

We ran six schemes based directly on the causal graph:

- **Gamer** is Gamer’s original ordering scheme, which approximates $\Omega^{ga}$.
- **GamerPre** is like Gamer but on a causal graph extended by arcs between pairs of precondition variables. The idea here is to capture not only dependency for forward search, but also for backward search, i.e., when inverting the actions.
- **WGamer** is like Gamer but with arcs weighted by the number of relevant actions, i.e., the number of actions inducing the corresponding arcs.
- **WGamerPre** is like GamerPre with weighted arcs.
- **CGLevel** is Fast Downward’s (Helmert, 2006) level heuristic, which approximates $\Omega^{cg}$. It orders the variables by strongly connected components and, within these components, considers the weighted causal graph and orders variables with smallest incoming weight first. Similar to WGamer, the weights correspond to the number of actions that induce an arc.
- **CGSons** is another approximation of $\Omega^{cg}$. It always selects a variable $v$ all of whose parents have already been selected; or at least one of whose parents has already been selected; or an arbitrary variable if no such $v$ exists.

Additionally, we used six ordering schemes we adopted from the model checking literature, based on a structure called the abstract syntax tree (AST) (e.g., Maisonneuve, 2009). That is a directed graph containing a root node for the overall task and subtrees for all actions. Each subtree consists of nodes representing the subformulas of the specified action (i.e., subformulas for the action’s precondition and its effect). The variables of the task are the leaves of the AST. The leaves are merged, i.e., we have only one node for each variable of the task. Edges point from a node representing some function to all corresponding subtrees.
We construct the AST based on the PDDL input. Consider the following example actions, similar to those in the Floortile domain. We have predicates \( \text{at}(r,t) \), denoting the tile \( t \) robot \( r \) is currently on and \( \text{painted}(t) \) denoting whether tile \( t \) has already been painted. We have two actions \( a_1 = \text{paint}(r_1,t_1,t_2) \) with precondition \( (\text{at}(r_1,t_1) \land \neg \text{painted}(t_2)) \) and effect \( (\text{painted}(t_2)) \) denoting that a robot \( r_1 \) can paint tile \( t_2 \) if it currently is on \( t_1 \) and \( t_2 \) has not been painted. Similarly, action \( a_2 = \text{paint}(r_2,t_3,t_2) \) with precondition \( (\text{at}(r_2,t_3) \land \neg \text{painted}(t_2)) \) and effect \( (\text{painted}(t_2)) \) denotes that robot \( r_2 \) can paint tile \( t_2 \) if it currently is on \( t_3 \) and \( t_2 \) has not been painted.

Figure 12 illustrates the corresponding AST. We have the root for all actions \( A \), and one subtree for each of the two actions \( a_1 \) and \( a_2 \). For both actions the preconditions and effects are encoded but we retain only one copy of each variable in the leaves (here relevant only for \( \text{painted}(t_2) \)).

Using their first author’s names for reference, the additional ordering schemes are the following.

**Butler** (Butler, Ross, Kapur, & Mercer, 1991) is an extension of an approach by Fujita, Fujisawa, and Kawato (1988). The latter proposed to perform a depth-first search (DFS) in the AST, starting at the root node, and to order the variables in the order in which they are reached the first time. Butler et al. extended this to a setting with several roots (if we remove the overall root and retain only the subtrees for the various actions we arrive at exactly the same setting). Their approach starts the DFS at the action containing the highest number of variables. Within the tree it advances in a similar manner: It always continues with the subtree that contains the highest number of variables among all subtrees of the current node. The retrieved ordering is then again in the order in which the variables are reached the first time.

**Chung1** (Chung, Hajj, & Patel, 1993) is a two-step approach. In the first step it assigns values to all nodes of the AST. Starting at the leaves, assigning them a value of 0, it assigns each inner node the maximum of the values assigned to its successors plus 1. In the second step it performs a DFS starting at the root, which is guided by the values of the nodes, visiting those successors with highest value first. The order in which the variables are reached for the first time is then chosen as the variable ordering.

**Chung2** (Chung et al., 1993) determines the shortest distance between each pair of variables, which can be calculated by considering all edges in the AST as undirected. Additionally, the total distances, i.e., the sum of the minimal distances to all other variables, are stored for all vari-
ables. A variable with smallest total distance is chosen first. The next one is the one closest to the last variable inserted into the ordering. In case of a tie the distance to the preceding variables is also taken into account.

*Maisonneuve* (Maisonneuve, 2009) is a greedy approach starting with an empty sequence. In each step it temporarily extends the current sequence by a variable not yet in the sequence. For this variable, a weight is determined, which is the number of variables from this extended sequence that appear in an action, summed over all actions. The variable is then removed from the sequence and the next one added. When all weights are calculated the variable with highest weight is appended to the sequence, and the next iteration starts, calculating the new weights for the remaining variables. In the end, the last sequence contains all variables and thus corresponds to the variable ordering.

*Malik* (Malik et al., 1988) assigns a level value (the maximal level of all its predecessors plus 1) to each node within the AST. The root is assigned value 0. The variables are then ordered according to their level values, those with highest values coming first.

*Minato* (Minato et al., 1990) calculates weights for all nodes in the AST. The weight of the root node of each action is set to 1, that of all successors of a node to $w/m$ if $w$ is the node’s weight and $m$ the number of successors of that node. One of the variables with highest weight is then chosen first and all its nodes removed (along with all ingoing edges and – recursively – those nodes with no remaining successors). For the reduced graph the weights are recalculated and the procedure continues until finally all variables are in the ordering.

### 4.2 Bad Delimiter

To get the bad delimiter we ran 5000 “random orderings”, where each such ordering corresponds to one run of the IPC’11 benchmark tasks, using a random variable ordering for each instance. To make this feasible we used a time-out of one minute (our backward search implementation is not viable for such a short time-out, so that we use only forward search here). For comparison to this data, the same settings (1 minute time-out, only forward search) was used with the twelve static ordering schemes. Initially we ran all ordering schemes and the random orderings on all tasks; after 200 random runs we removed all tasks from the benchmark set that were not solved at least once during the previous (random or static ordering) runs, retaining 85 tasks. Figure 13 shows coverage, i.e., the number of solved planning tasks, on the $x$ axis, and the fraction of random orderings having that coverage on the $y$ axis. The coverage data for the ordering schemes are shown as vertical lines.

*Malik* and *CGLevel* lie in and below the middle of the Gaussian distribution, respectively. In other words, *Malik* is as bad as, and *CGLevel* is even worse than, the average random ordering. Matters are not as bleak for the other ten ordering schemes, which are close together and lie clearly above the Gaussian distribution. Compared to a best-of over the random orders, however, all the ordering schemes appear rather humble. Consider Table 1. In particular, consider $n_{-r}$, giving the number of instances solved by ordering scheme $\Omega$ but not by any random order, and consider $n_{+r}$, giving the number of instances not solved by the scheme but solved by some random order. As Table 1 shows, $n_{-r}$ is strictly smaller than $n_{+r}$ in all but three of the ordering schemes $\Omega$, and is strictly larger (by a single task) only for one of the schemes (namely *Butler*). The average over $n_{-r}$ is 2.92 while that over $n_{+r}$ is 9.08.
Figure 13: Coverage for random orders vs. ordering schemes. Schemes are ordered top-to-bottom from worst to best coverage. \( X + Y \) means that both schemes, \( X \) and \( Y \), result in the same coverage.

<table>
<thead>
<tr>
<th>Type</th>
<th>Butler</th>
<th>CGLevel</th>
<th>CGSons</th>
<th>Chung1</th>
<th>Chung2</th>
<th>Gamer</th>
<th>GamerPre</th>
<th>Maisonneuve</th>
<th>Malik</th>
<th>Minato</th>
<th>WGamer</th>
<th>WGamerPre</th>
<th>Best Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_{-r} )</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>( n_{+\Omega} )</td>
<td>2</td>
<td>26</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>11</td>
<td>22</td>
<td>8</td>
<td>7</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>( n_{++\Omega} )</td>
<td>77</td>
<td>53</td>
<td>75</td>
<td>75</td>
<td>77</td>
<td>73</td>
<td>73</td>
<td>68</td>
<td>57</td>
<td>71</td>
<td>72</td>
<td>68</td>
<td>78</td>
</tr>
<tr>
<td>( n_{-\Omega} )</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Differences in solved instances for the 85 IPC’11 tasks (1 minute timeout); \(-r\) means solved by no random ordering, \(+r\) by at least one random ordering, \(-\Omega\) not solved by the corresponding ordering scheme, \(+\Omega\) solved by the corresponding ordering scheme.

4.3 GoodDelimiter

Here we performed bidirectional blind search, i.e., the most competitive setup in general. Figure 14 contains one data point for every pair \((I, \Omega)\) of IPC’11 benchmark instance \(I\) and ordering scheme \(\Omega\) that were solved by both (a) Gamer using dynamic reordering starting from an arbitrary variable order (the one returned by Gamer’s grounding process), and (b) Gamer using ordering scheme \(\Omega\) (without dynamic reordering). The time-out is 6 hours for (a), and 30 minutes for (b). The \(x\)-value of each data point is the size of the largest BDD constructed for \(I\) by (a), the \(y\)-value is the size of the largest BDD constructed for \(I\) by (b). We allowed a much higher time-out for dynamic reordering because such reordering is not runtime effective: The question we are asking here is merely which of the two methods yields smaller BDDs. Figure 14 shows that dynamic reordering is universally much better at this, giving us sizes that are up to three orders of magnitude smaller than those of the schemes. For a total of 1911 instances (solved by both an ordering scheme and dynamic reordering), in 1431 cases the BDD sizes are smaller by a factor of up to 10 when using dynamic reordering, in
Table 2 shows the coverage of the different schemes on the IPC’11 tasks. We can make a similar observation to that of the one minute, only forward search results, namely that CGLevel and Malik are clearly behind the others. The last column shows the coverage of Gamer using dynamic reordering, and provides two numbers, first the coverage with the 6 hours timeout, second the coverage with the same timeout as the schemes, i.e., 30 minutes. From this it becomes clear that applying dynamic reordering for the entire search time is not feasible in practice when limiting the runtime.
Limited Dynamic Reordering

Given the much more memory-efficient behavior of dynamic reordering, a possible approach is to run dynamic reordering for a limited time only, hoping to get an ordering that is good enough for the remainder of the search. Reordering is automatically started when the number of allocated BDD nodes reaches a certain threshold (by default, the first threshold is 4000 nodes), which is dynamically adapted after each reordering (by default, the next threshold is set to 2 times the number of nodes after reordering). A simple way to control dynamic reordering is to limit the number of reordering steps, and to turn dynamic reordering off once the desired number of reorderings has been performed.

For different reordering limits, the total runtime for a task often looks similar to the situation depicted in Figure 15a. With too few reorderings it takes a long time to solve the task due to a bad initial ordering. Also, the first reorderings can sometimes hurt, as they are performed at the very beginning or during construction of the transition relation, before enough information on good orderings is available. However, with too many reorderings solving takes a long time due to an immense overhead in reordering time, which grows exponentially with each step.

An important different behavioral pattern is depicted in Figure 15b: In some domains, such as PegSol or Sokoban, the minimum of the curve is at the very beginning (without any reordering), and the total runtime only increases afterward (mainly based on the increase in reordering time). An explanation for this behavior might be that the initial ordering is already pretty good, so that dynamic reordering cannot improve much and its overhead is incurred in vain. The same also often happens in the easier tasks of a domain, so that learning a good setting based on the simpler tasks seems impossible.

Attempting to exploit these observations to design adaptive stopping criteria, geared at finding a good point for stopping dynamic reordering, given only the available observations (e.g., number of BDD nodes before/after reordering, reordering times, current total runtimes), we experimented with the following approaches.

First, we noticed that early on the reordering time increases from step to step by a small factor, but later on that factor increases. In some preliminary runs we saw that often the area of smallest runtime coincides with the situation when the increase in reordering time reaches some threshold, often between 1.25 and 1.75 (see, e.g., Figure 16a and compare it to the runtime minimum in Figure 15a for the same planning task). We call this the factor criterion.
Second, another observation from these preliminary runs is that the percentage of the time spent in the last reordering step on the total current runtime often follows a U-like curve, and the minimum of that curve often lies close to the same number of reorderings as the total runtime minimum (see, e.g., Figure 16b and compare to the runtime minimum in Figure 15a for the same task). We employ a percentage criterion, which stops reordering after a (possibly local) minimum has been reached, i.e., we compare the percentage of the current step with that of the previous step; when the current one is greater we stop reordering.

Finally, A simple combination of both criteria is to stop reordering as soon as one of them tells us to do so.

To evaluate these adaptive stopping criteria, we ran all tasks of the IPC’11 domains with different limits for the number of reorderings, ranging from 0 to 20. Based on those runs we calculated the results we would achieve with the adaptive stopping criteria. See Table 3. The best possible coverage, i.e., the number of tasks solved by at least one setting with limited number of reorderings, is 175, while without any reordering we found 167 solutions. The adaptive stopping criteria yield coverage between 158 and 171. Performance is reasonable for the factor criterion, but is quite bad for the percentage criterion and the combination of both criteria. Recall that the percentage criterion aims at stopping reordering before incurring prohibitive overhead. Indeed, with this criterion, reordering is often stopped earlier than with the factor criterion. In some cases this was detrimental, particularly in the Woodworking domain where this strategy happens to fall into a dramatic local peak of the total-runtime curve, resulting in 9 problem instances no longer being solved.

As, in several cases, dynamic reordering during the transition relation creation was counterproductive, we also ran delayed reordering, where dynamic reordering is started only after the BDDs for the transition relation have been created. The results are in Table 4. The case without reordering is unchanged with respect to Table 3. The best possible result is now slightly worse than before, with coverage 174. For the adaptive stopping criteria, the picture changes substantially: In contrast to Table 3, the percentage criterion now excels, delivering coverage just 1 short of the best possible. Regarding the factor criteria, overly small or large factors now are bad and the best behavior (2 short of best possible) is obtained in the middle.

In all those cases the highest number of reorderings we could observe was clearly below 20. Either the planner ran out of time or memory, or finished before the last reorderings could be performed. In all cases we used GamerPre as the initial ordering, which turned out to be among the best in a preliminary set of experiments.
Table 3: Coverage results for different stopping criteria. Immediate reordering.

<table>
<thead>
<tr>
<th>Domain</th>
<th>no reord</th>
<th>best possible</th>
<th>factor criterion</th>
<th>percentage criterion</th>
<th>both criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.25 1.5 1.75 2.0</td>
<td>1.25 1.5 1.75 2.0</td>
<td></td>
</tr>
<tr>
<td>Barman</td>
<td>7</td>
<td>8</td>
<td>8 8 8 8 8</td>
<td>8 8 8 8 8</td>
<td>8</td>
</tr>
<tr>
<td>Elevators</td>
<td>19</td>
<td>19</td>
<td>19 19 19 19 19</td>
<td>19 19 19 19 19</td>
<td>19</td>
</tr>
<tr>
<td>Floortile</td>
<td>8</td>
<td>8</td>
<td>7 8 8 8 8</td>
<td>8 7 8 8 8</td>
<td>8</td>
</tr>
<tr>
<td>NoMystery</td>
<td>14</td>
<td>16</td>
<td>14 14 14 14 14</td>
<td>14 14 14 14 14</td>
<td>14</td>
</tr>
<tr>
<td>Openstacks</td>
<td>20</td>
<td>20</td>
<td>20 20 20 20 20</td>
<td>20 20 20 20 20</td>
<td>20</td>
</tr>
<tr>
<td>PARC-Printer</td>
<td>6</td>
<td>7</td>
<td>6 6 7 7</td>
<td>6 6 6 6 6</td>
<td>6</td>
</tr>
<tr>
<td>PegSol</td>
<td>18</td>
<td>18</td>
<td>17 17 17 17 17</td>
<td>17 17 17 17 17</td>
<td>17</td>
</tr>
<tr>
<td>Scanalyzer</td>
<td>9</td>
<td>9</td>
<td>9 9 9 9</td>
<td>9 9 9 9 9</td>
<td>9</td>
</tr>
<tr>
<td>Sokoban</td>
<td>19</td>
<td>19</td>
<td>17 17 17 17 17</td>
<td>17 17 17 17 17</td>
<td>17</td>
</tr>
<tr>
<td>Tidybot</td>
<td>12</td>
<td>14</td>
<td>14 14 14 14 14</td>
<td>13 13 13 13 13</td>
<td>13</td>
</tr>
<tr>
<td>Transport</td>
<td>8</td>
<td>9</td>
<td>9 9 9 9</td>
<td>9 9 9 9 9</td>
<td>9</td>
</tr>
<tr>
<td>VisitAll</td>
<td>11</td>
<td>12</td>
<td>11 12 12 12 12</td>
<td>11 11 11 11 11</td>
<td>11</td>
</tr>
<tr>
<td>Woodworking</td>
<td>16</td>
<td>16</td>
<td>10 15 16 16</td>
<td>7 9 7 7 7</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>167</strong></td>
<td><strong>175</strong></td>
<td><strong>161 168 170 171</strong></td>
<td><strong>158 159 158 158 158</strong></td>
<td><strong>158</strong></td>
</tr>
</tbody>
</table>

Table 4: Coverage results for different stopping criteria. Delayed reordering, i.e., reordering started only after creation of the transition relation BDDs.

<table>
<thead>
<tr>
<th>Domain</th>
<th>no reord</th>
<th>best possible</th>
<th>factor criterion</th>
<th>percentage criterion</th>
<th>both criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.25 1.5 1.75 2.0</td>
<td>1.25 1.5 1.75 2.0</td>
<td></td>
</tr>
<tr>
<td>Barman</td>
<td>7</td>
<td>8</td>
<td>8 8 8 8 7</td>
<td>8 8 8 8 8</td>
<td>8</td>
</tr>
<tr>
<td>Elevators</td>
<td>19</td>
<td>19</td>
<td>19 19 19 19 19</td>
<td>19 19 19 19 19</td>
<td>19</td>
</tr>
<tr>
<td>Floortile</td>
<td>8</td>
<td>8</td>
<td>7 8 8 8 8</td>
<td>8 7 8 8 8</td>
<td>8</td>
</tr>
<tr>
<td>NoMystery</td>
<td>14</td>
<td>16</td>
<td>16 16 14 14 14</td>
<td>16 16 16 16 16</td>
<td>16</td>
</tr>
<tr>
<td>Openstacks</td>
<td>20</td>
<td>20</td>
<td>20 20 20 20 20</td>
<td>20 20 20 20 20</td>
<td>20</td>
</tr>
<tr>
<td>PARC-Printer</td>
<td>6</td>
<td>7</td>
<td>6 7 7 7</td>
<td>6 6 7 7 7</td>
<td>6</td>
</tr>
<tr>
<td>PegSol</td>
<td>18</td>
<td>18</td>
<td>17 17 17 17 17</td>
<td>17 17 17 17 17</td>
<td>17</td>
</tr>
<tr>
<td>Scanalyzer</td>
<td>9</td>
<td>9</td>
<td>9 9 9 9</td>
<td>9 9 9 9 9</td>
<td>9</td>
</tr>
<tr>
<td>Sokoban</td>
<td>19</td>
<td>19</td>
<td>19 19 19 19 19</td>
<td>19 19 19 19 19</td>
<td>19</td>
</tr>
<tr>
<td>Tidybot</td>
<td>12</td>
<td>13</td>
<td>13 13 13 13</td>
<td>13 13 13 13 13</td>
<td>13</td>
</tr>
<tr>
<td>Transport</td>
<td>8</td>
<td>9</td>
<td>9 9 9 9</td>
<td>9 9 9 9 9</td>
<td>9</td>
</tr>
<tr>
<td>VisitAll</td>
<td>11</td>
<td>12</td>
<td>12 12 12 12 12</td>
<td>12 12 12 12 12</td>
<td>12</td>
</tr>
<tr>
<td>Woodworking</td>
<td>16</td>
<td>16</td>
<td>10 15 16 16</td>
<td>16 10 15 16 16</td>
<td>16</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>167</strong></td>
<td><strong>174</strong></td>
<td><strong>165 172 171 169</strong></td>
<td><strong>173 165 172 173 173</strong></td>
<td><strong>173</strong></td>
</tr>
</tbody>
</table>

To shed some light on these observations, Figure 17 shows coverage as a function of more different factor values, for both the case of immediate reordering (Figure 17a) and that of delayed reordering (Figure 17b). In Figure 17a, we see that the percentage criterion stops reordering too early. Without it, the coverage resulting from stopping reordering based solely on the factor criterion can get as high as 173. However, before the ascend in the coverage actually starts the percentage criterion stops reordering, thus cutting off many solutions. In Figure 17b, up to a factor of 1.6 both curves are identical, both mainly increasing with increasing factor. After that, the combination criterion rises another little bit, while the factor criterion alone drops substantially. The combined criterion avoids that drop because, at some point (here, at a factor of roughly 2.0), the percentage criterion stops reordering at least as early as the factor criterion.
BDD Ordering Heuristics for Classical Planning

Figure 17: Coverage as a function of factor, for the factor criterion alone, and for its combination with the percentage criterion (denoted as “Both”).

6. Conclusion

It is tempting to equate the “variable dependencies” in BDD-based symbolic search with those identified in causal graphs, and previous research has done so unquestioningly. Looking a little more closely at this issue, we have shown that causal graph based variable orderings are exponentially bad even in severely restricted sub-classes of planning. Empirically, Fast Downward’s level heuristic is worse than random, and all ordering schemes lag far behind off-the-shelf reordering.

One may wonder about the meaning of the theoretical results: How could a static ordering scheme not incur exponential overhead in the worst case? We agree with that view in principle, but we did not expect this to happen in planning tasks so restricted as to be tractable for domain-independent optimal planning. It remains to be seen to what extent our classification framework is suitable to characterize the properties of other ordering schemes and/or planning fragments.

Our impression at this point is that static ordering schemes are so limited as to be hopeless. Prior to actually building the BDDs, it appears impossible to extract any reliable information about which form they will take. The way forward, then, is to use dynamic reordering techniques in a more targeted manner. Our initial experiments in that direction did not meet with an immediate breakthrough, but certainly they show promise, especially considering the primitive nature of the method and of the stopping criteria employed. Promising future directions include more flexible on/off strategies for dynamic reordering, machine learning for deciding when to toggle the switch, and planning-specific reordering techniques exploiting the particular structure of the BDDs at hand.

Acknowledgments

We thank the anonymous reviewers of the ICAPS 2013 short version and those of a previous version of this article, whose comments helped tremendously to improve the paper.

References


Wegener, I. (2000). *Branching Programs and Binary Decision Diagrams*. SIAM.
