Branching Matters: Alternative Branching in Graphplan

Jörg Hoffmann
Institut für Informatik
Georges-Köhler-Allee, Geb. 52
79110 Freiburg, Germany
hoffmann@informatik.uni-freiburg.de
+49 (761) 203-8229

Héctor Geffner
Departament de Tecnologia
Passeig de Circumval-lació, 8
08003 Barcelona, Spain
hector.geffner@tecn.upf.es
+34 93-542-2563

Abstract

We illustrate the importance of branching in planning by exploring alternative branching schemes in Graphplan. As argued elsewhere, Graphplan can be understood as a heuristic search planner that performs an IDA* regression search with a heuristic function encoded in the plan graph. Here, we study two alternatives to Graphplan where the IDA* search algorithm and the heuristic encoded in the plan graph are maintained but the *branching scheme* is changed: rather than constructing plans from the tail, commitments are allowed anywhere in the plan. These commitments force certain actions *in* or *out* of certain time steps. While the regression search allows Graphplan to build the plan graph only once, in the new branching scheme the plan graph must be computed anew for each state. This re-computation is expensive but is often compensated by a reduced number of states in the search. We show through a number of experiments that the resulting planner scales up much better than the original Graphplan in domains involving a higher degree of parallelism, and compares well to state-of-the-art domain-independent optimal parallel planners such as Blackbox using Chaff.

Introduction

Heuristic search has been shown to be an effective approach in AI Planning. In this approach, plans are searched for with the help of a heuristic function that is extracted automatically from the problem encoding (McDermott 1999; Bonet & Geffner 2001). Most current domain-independent planners make use of heuristic estimators of some sort, and even planners such as Graphplan, that have not been designed from this perspective, can be understood in those terms (Bonet & Geffner 1999).

Current heuristic search planners, however, are strong for optimal and non-optimal sequential planning (Haslum & Geffner 2000; Hoffmann & Nebel 2001) but weaker for parallel planning. In optimal parallel planning, the task is to compute a plan that involves a minimum number of time steps where in each time step many actions can be done in parallel. Optimal parallel planning is the basic type of planning supported by current Graphplan and SAT planners (Blum & Furst 1995; Kautz & Selman 1999) and a special case of *temporal planning* when all actions have the same duration.

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

An important obstacle faced by current heuristic search planners in the parallel setting is the *high branching factor*. Indeed, in heuristic regression based planners such as Graphplan and HSPr* (Haslum & Geffner 2000), the number of sets of concurrent actions that need to be considered in each state s grows *exponentially* with the degree of parallelism: one set for each combination of supports for the atoms in s. It is thus not surprising that the best current optimal parallel planners are based on either SAT or CSP formulations where branching is accomplished by trying the different values of selected boolean variables, and hence where the branching factor is always two (Kautz & Selman 1999; Rintanen 1998; Baioletti, Marcugini, & Milani 2000; Do & Kambhampati 2000). Of course, a small branching factor by itself is not sufficient for good performance, and indeed, these SAT and CSP approaches supplement the *branching strategy* with effective *pruning mechanisms* based on constraint propagation and consistency checking.

In this work, we deal with the problem of optimal parallel planning from the perspective of heuristic search. However, rather than searching for plans either forward from the initial state or backward from the goal, we consider a branching scheme that is closer to the one found in SAT and CSP approaches: we pick an action variable a and a time point i, and branch making the action a at time i either true or false. States σ in the search are thus no longer sets of *atoms* but sets of *commitments*. Yet, as before, we use a heuristic function $h(\sigma)$ to provide a lower bound on the cost of achieving the goals given the commitments in σ, and prune σ from the search when $h(\sigma)$ exceeds a bound in an IDA* search.

More precisely, we define a *branch and bound* planner, that we call BBG (Branch-and-Bound-Graphplan), on top of the IPP implementation (Koehler *et al.* 1997) of Graphplan. As argued in (Bonet & Geffner 1999) and then in more detail in (Haslum & Geffner 2000), Graphplan can be understood as a heuristic search planner that searches for parallel plans from the goal using the IDA* search strategy and a heuristic that is implicitly encoded in the plan graph. BBG retains from Graphplan the same *IDA* search strategy* and the same *plan graph heuristic function*, but changes the *branching strategy* allowing commitments anywhere in the plan to be made. In the resulting search space, the heuristic is obtained by recomputing the plan graph in each node. While this recomputation is expensive, we show that it is often compen-

sated by a reduced number of states in the search. Indeed, the resulting planner scales up much better than the original Graphplan in domains involving a higher degree of parallelism where it compares well against state-of-the-art domain independent parallel planners such as Blackbox (Kautz & Selman 1999) using Chaff (Moskewicz *et al.* 2001).

Two clarifications before we proceed. First, all planners based on algorithms such as IDA* such as Graphplan and HSPr* are ' branch and bound' ; namely, they branch in a certain way, and prune nodes by means of suitable bounds. We find it useful, however, to refer to such planners as heuristic search planners, leaving the term ' branch and bound' for planners that are not restricted to regression or progression search. Second, the notion of branching has been analyzed in AI planning but implicitly, in terms of the *space* searched during plan construction. Namely, linear planners (including progression and regression planners) are said to search in the space of *states,* while non-linear planners are said to search in the space of *plans* (e.g., (Weld 1994)). This is certainly a useful distinction, yet as discussed in more detail in (Geffner 2001), focusing explicitly on the notion of branching in planning sheds also some light on the relation between linear and non-linear planning on the one hand, and SAT/CSP formulations and other well-known combinatorial optimization problems on the other. For example, the best codes for the Traveling Salesman Problem do not branch by considering all possible next cities - as would correspond to a linear planner - but by selecting an edge in the graph, and forcing this edge *in* or or *out* of the solution (Balas & Toth 1985). Something similar occurs in the Job Shop Problem (Carlier & Pinson 1989), and as we will see, in the branching scheme introduced in this paper.

The paper is organized as follows. The next section outlines the Graphplan algorithm and its interpretation as a heuristic search planner. We then include two sections describing the new branching schemes and the branching selection criteria. Then, after providing an overview over the implementation, we evaluate the approach empirically, and discuss the results. A short appendix provides some more implementation details.

Graphplan and Heuristic Search

We provide a brief summary of Graphplan, see (Blum & Furst 1995) for details. We use the letter I and G to denote the initial and goal situations of a STRIPS planning task; actions a are the usual triples $pre(a)$, $add(a)$, $del(a)$, and for each proposition p, we assume the presence of a `dummy' action $a = NOOP(p)$ with $pre(a) = add(a) = \{p\}$ and $del(a) = \{\}$. Graphplan builds a *plan graph* which is a layered graph consisting of *fact layers* and *action layers*. Fact layer 0 is the initial state I, and action layer 0 is the set of actions (including NO-OPs) whose preconditions are in I. Fact layer i, for $i > 0$, contains the add effects of actions in layer $i - 1$, and action layer i, for $i > 0$, contains the actions whose preconditions are in fact layer i without a mutex. Mutex relations are defined as follows. A pair of actions in action layer 0 is *mutex* if they *interfere*, i.e., if one action deletes a precondition or an add effect of the other (a non-interfering pair of actions is called *compatible*). A pair of facts at layer $i > 0$ is mutex if there is no non-mutex pair of actions at layer $i - 1$ achieving the facts. A pair of actions at a layer $i > 0$ is mutex if the actions interfere, or if they have *competing needs*, i.e., mutex preconditions. A set of facts is *reachable* at layer i if all these facts are contained in layer i and no pair of them is mutex. From the first layer where G is reachable, Graphplan starts a regression search, where a set of (sub)goals s at layer i yields a new set of subgoals s' at layer $i-1$ for each set of pairwise non-mutex actions (possibly including NO-OPs) that achieve s. In that case, s' is the union of those action's preconditions. The search terminates successfully when the initial layer is reached, or negatively when all options have been explored unsuccessfully. In the latter case the graph is extended by another layer, and search continues. When the search beneath a subgoal set s at layer i fails, s is *memoized*, i.e., remembered as unsolvable at i.

As argued originally in (Bonet & Geffner 1999), and further elaborated in (Haslum & Geffner 2000), Graphplan can be understood as a heuristic search planner with the following features:

1. **Branching:** States are sets of atoms (subgoals), and the children s' of a state s are obtained by parallel regression with NO-OPs as described above. The initial state in the search is G and the goal states s are such that $s \subseteq I$.
2. **Bounds:** The plan graph encodes an *admissible heuristic* h_G for parallel planning, estimating the parallel cost of reaching s from I, where $h_G(s) = i$ iff i is the index of the first fact layer where s is reachable and not memoized (memoizations are updates on the heuristic function; see 4).[1]
3. **Search Algorithm:** Search is an enhanced version of IDA* (Korf 1985), where the sum of the accumulated cost $g(s)$ (distance from G), and the estimated cost $h_G(s)$ (distance estimate to I) is used to prune states s whose cost exceed the current bound (the index of the current top graph layer). Actually Graphplan never generates such nodes. The algorithm takes advantage of additional information stored in the plan graph to convert the search into a more efficient `solution extraction' procedure. For example, compatible but mutex (due to competing needs) pairs of actions are not used for regressing a state; the resulting state would indeed have a cost exceeding the IDA* bound.
4. **Memoization:** Memoizations are updates on the heuristic function h_G. If a state s is expanded and the search beneath it fails, then the optimal cost $h^*(s)$ must be strictly greater than $h_G(s)$ (otherwise a solution through s would have been found given the same bound), and hence the estimated cost $h_G(s)$ can be increased. This is a common extension in modern IDA* implementations which refer to the technique as *transposition tables*;

[1] The heuristic h_G is equivalent to the parallel h^m heuristic, for $m = 2$, defined in (Haslum & Geffner 2000) as the relaxation that approximates the cost of large sets of atoms by the cost of the most costly m-tuple in the set. Without the mutexes, the h_G heuristic corresponds to the h^1 heuristic, also known as 'max heuristic' (Bonet & Geffner 2001).

e.g., (Sen & Bagchi 1989; Reinfeld & Marsland 1994; Junghanns & Schaeffer 1999; Haslum & Geffner 2000).

The dramatic improvement in performance that Graphplan brought to AI planning in the middle 90's can be traced from this perspective to the use of an informative *heuristic function.* Indeed, previous partial order planners, unlike Graphplan, perform *blind* search (using a different branching scheme).

Branching

The search for optimal solutions in planning, and more generally, combinatorial optimization, is often the result of two main operations: *branching* and *pruning.*[2] Branching refers to the scheme used for generating search nodes, while pruning refers to the criterion used for eliminating some of them from consideration. In Graphplan, *branching* is done by parallel regression, and *pruning* is done when the lower bound $f(s) = g(s) + h_G(s)$ exceeds the current bound. SAT and CSP approaches rely on the same search strategy, IDA*, but branch and prune in a different way: *branching* is done by trying the different values of a selected variable, and *pruning* is done when the choices, along with the problem constraints and the bound condition, yield an inconsistency by constraint propagation. Branching and pruning, however, are largely two orthogonal operations, and in this paper we aim to explore alternative branching schemes to Graphplan's parallel regression, leaving its pruning technique, and in particular its plan graph heuristic estimator, untouched.

Branching on Actions

In the first alternative branching scheme we consider, *branching on actions*, an action a and time d (the decision layer) smaller than or equal to the current bound are selected, and a binary split is generated: one in which a is made true at time d, the other where it is made false. A state σ in this search is a set of choices or commitments of this type. For example in the blocks world, a state σ will contain a set of moves to be done at various time points, and a set of moves *not* to be done at possibly other time points.

We let $h^*(\sigma)$ refer to the cost of the best (parallel) plan that is compatible with the commitments contained in σ. Since computing the exact cost is intractable ($h^*(\emptyset)$ is the length of an optimal parallel plan), we compute a lower bound $h_G(\sigma) \leq h^*(\sigma)$. This lower bound (admissible heuristic) is obtained by building a plan graph as in Graphplan *but* respecting the commitments made in σ; namely, if $a[i] =$ **true** $\in \sigma$, then action a must be in action layer d of the plan graph for σ and all actions mutex with a at d must be excluded from that layer. Similarly, if $a[d] =$ **false** $\in \sigma$, action a is excluded from layer d in the plan graph for σ.

As for Graphplan, $h_G(\sigma)$ is given by the index of the first layer in the plan graph for σ that contains the goals without a mutex. The initial state $\sigma = \sigma_0$ is the empty set of commitments, and thus $h_G(\sigma_0)$ is computed from exactly the same plan graph that is constructed by Graphplan. For other states, the plan graph is constructed incrementally from the parent plan graph by retracting one or more actions in the corresponding layer, and propagating this retraction forward. A state σ in this search is *pruned* when $h_G(\sigma)$ exceeds the current bound or when the plan graph construction procedure cannot comply with the commitments made (the latter happens when a new commitment at layer d makes the preconditions of an action a previously committed to a layer i, $i > d$, either unachievable or mutex). The terminal states in this search are those for which the plan graph contains no 'flaws', and hence, where a valid plan can be extracted easily. This is explained more precisely below, when the selection criteria for choosing the action a and time d for branching are presented.

Branching on Mutexes

In the second branching scheme that we consider, *branching on mutexes*, a pair of actions a_1 and a_2 that are mutex at time d are selected, and a three way split is generated: one in which a_1 is made true at d, one in which a_2 is made true at d, and finally, one in which a_1 and a_2 are both false at d. As before, if a state σ contains a commitment of the form $a[d] =$ **true**, then the plan graph for σ will have all actions that are mutex with a at d excluded. The heuristic function, pruning, and initial state are the same as those for branching on actions.

The choice of the two branching schemes above is somewhat arbitrary: we could as well have done branching not on actions but on facts or both (as in SAT). In principle, any partition of the space of plans that could be reflected in the plan graph construction would do. Ours is thus an initial and not an exhaustive exploration of alternative branching schemes in the Graphplan setting.

Branching Selection Criteria

We refer to the choice of the action (mutex) and time step on which to branch as the branch point selection. Like the similar variable-selection heuristic criterion in constraint-based search, the branch point selection criterion does not affect the soundness or completeness of the branching scheme, but affects the size of the search tree and thus is critical for performance. After some limited empirical exploration, we settled on two goal-oriented criteria that select branching points in order to remove *flaws* in a *relaxed plan.* A state σ for which the relaxed plan contains no flaw is a *terminal state* from which a valid parallel plan can be easily extracted (with no search). The flaw identification and repair loop in BBG is reminiscent of a similar loop in partial-order planning (Weld 1994); yet the nature of the flaws and repairs, as well as the pruning criteria, are different.

A relaxed plan in a state σ is defined as in the FF planner (Hoffmann & Nebel 2001): as a solution extracted from the plan graph built in σ ignoring mutex relations between actions. Thus, construction of a relaxed plan from the plan graph in σ is backtrack free and fast. While there may be an exponential number of relaxed plans in σ, we care only

[2] This applies to linear-space search algorithms like IDA* and DFS Branch and Bound, but not to exponential-space algorithms like A* that, in any case, are not feasible over large problems. See (Korf 1993).

about the first relaxed plan found, that we call the relaxed plan for σ. Since we would like this relaxed plan to have as few flaws as possible, we extract it by ordering the atoms and their supporters in each layer so that 1) atoms with smaller number of supporters are regressed first, and 2) no-ops are chosen for support if possible, otherwise, a regular action is chosen greedily, minimizing the number of mutex relations with supporters already selected.

The relaxed plan, like a regular parallel plan, is a set of action occurrences (a, i) where a is an action and i is a time point. Unlike regular plans, however, the relaxed plan explicitly represents the occurrence of NO-OP actions as well.

A *flaw* in the relaxed plan is a triplet (a, a', i) where (a, i) and (a', i) are two action occurrences in the relaxed plan such that a and a' are mutex at i.

The first selection criterion for choosing branching points in both branching schemes focuses on flaws that occur *latest* in the relaxed plan, i.e. the flaws (a, a', d) with maximal time index d. In both branching schemes, an arbitrary latest flaw (a, a', d) is chosen (actually, the first flaw found in the relaxed plan extraction procedure). When branching on mutexes this flaw forms the new branching point; when branching on actions one of (a, d) or (a', d) is chosen arbitrarily (the action with the lower internal index). We have tried other simple static selection criteria (earliest flaws, middle flaws, etc) but the experiments didn't show improvements, and in certain cases showed a clear deterioration. The focus on the `latest flaws' does not mean that we are back to a regression search as these are the latest flaws in the relaxed plan and not in the plan graph itself. Indeed, criteria for choosing branching points that do not make use of the relaxed plan do not appear to do as well.

We consider also a dynamic selection criterion based on estimating the *criticality* of action occurrences in the relaxed plan. The *criticality measures* $c(a, i)$ are computed in a single backward pass over the plan graph and aim to approximate the relative number $c^*(a, i)$ of relaxed plans in which a occurs at time i, a number that ranges from $1/NR$ to 1, where NR is the number of relaxed plans.[3] Ideally, we would like to choose a branching point that removes highly critical actions in all its branches. For that, let b be a branching point, and let $b_1, \ldots, b_n$, refer to the commitments made in each of its branches (thus, n is equal to 2 resp. 3 in the branching on actions resp. mutexes scheme). We define the criticality of a commitment b_j, $c(b_j)$, as the measure $c(a, i)$ of the most critical action-time pair a, i forced out by b_j,[4] and the criticality of a branching point b, $c(b)$, as the product $c(b_1) \cdot \ldots \cdot c(b_n)$. When branching on mutexes, we select for branching the most critical relaxed plan flaw $b = (a, a', d)$, and when branching on actions, we select for branching the most critical branching point $b = (a, d)$ that occurs in a relaxed plan flaw (a, a', d). Ties, as before, are broken arbitrarily.

[3] Criticality measures are computed for both facts (p, i) and actions (a, i) as follows. The goal facts g are assigned a measure $c(g, m) = 1$ where m is the current IDA* search bound. The process then proceeds backwards from graph layer m to graph layer 0. If a fact f at layer i has achievers $a_1 \ldots a_n$ at layer $i - 1$ then it propagates a measure $c(f, i)/n$ into each of these actions. The measure $c(a, i)$ of an action a at layer i is then the max measure it gets from the facts at layer $i + 1$. The action propagates in turn its own measure $c(a, i)$ into its preconditions at i. The measure $c(f, i)$ of a fact at layer $i < m$ is the max measure it gets from the actions a at layer i.

[4] More precisely, if the commitment b_j in a state σ is $a[i] =$ **true**, then $c(b_j)$ is defined as $\max_{a'} c(a', i)$ over all actions a' that are mutex with a at time i in the plan graph corresponding to σ. Similarly, if the commitment b_j is $a[i] =$ **false** for one or more actions a, then $c(b_j)$ is defined as $\max_{a'} c(a', i)$ over all such actions a.

Implementation Overview

Seeing as in our approach the plan graph must be recomputed in every single search state, an efficient implementation of the re-computation is crucial for performance. Our implementation builds on the ideas introduced in STAN (Long & Fox 1999), which we re-implemented following their realization in IPP (due to some minor changes in the re-implementation our graph building procedure is in fact somewhat more efficient than IPP's in most of the cases we tried). Here, we provide a brief overview over the implementation; as said, some more details can be looked up in a short appendix.

The plan graph is represented as a single layer of facts and actions, where only the layer-dependent information is updated along the time steps. When initially building the graph, all the optimizations described in (Long & Fox 1999) are applied in order to avoid unnecessary re-computations of the same pieces of information (such unnecessary computations are surprisingly plentiful in the original Graphplan implementation). Some crucial informations that must frequently be accessed, namely when computing mutexes which is the most frequent computation performed while building the graph, are kept in bit vectors.

When a new constraint is introduced at a layer d, the graph must be updated by removing a set of actions at layer d, and propagating the changes. In general, we have a layer i, a set ops of actions to be removed at i, and a set $opmutexs$ of action mutexes to be introduced at i. These changes are propagated to the next fact layer, i.e., the actions ops are removed from the possible achievers for the respective facts at layer $i + 1$, and the mutexes $opmutexs$ are introduced (all these operations are done implicitly by updating bit vectors and setting flags). In consequence, facts can become unachievable or mutex. We thus get a set fts of facts to be removed at $i + 1$, and a set $ftmutexs$ of fact mutexes to be introduced at $i + 1$. Propagating these changes to action layer $i + 1$ amounts to removing actions which need a fact in fts, or a facts pair in $ftmutexs$, as precondition, and introducing mutexes between actions which now have competing needs due to $ftmutexs$. So we get a list ops of actions to be removed at $i + 1$, and a set $opmutexs$ of action mutexes to be introduced at $i + 1$. We are now back in the situation we started from, so the algorithmic loop is closed. The loop terminates when the current deadline layer is reached. If the goals are not reachable in the deadline layer then it is known that the heuristic value is above that deadline so this node

can be pruned.[5] All changes made during the graph update are kept in memory so they can easily be retracted when search backtracks over the decision that caused the update.

Results

The motivation for the introduction of our new branching schemes was to improve the performance over highly-parallel domains where the branching factor in Graphplan and similar parallel heuristic search planners grows exponentially (see (Haslum & Geffner 2000) for a discussion). We have thus compared BBG with IPP over two different suits of domains: one containing domains that are highly parallel, the other containing domains that are mostly serial. The results, as we will see, show a dramatic improvement in the former, and no significant improvement - in certain cases a clear deterioration - in the latter. We also wanted to evaluate how close the change in the branching scheme takes the 'new Graphplan', BBG, in relation to state of the art domain-independent parallel planners such as Blackbox using Chaff. At the end of this section we include such a comparison, where we will see that in around half of our domains Blackbox appears to dominate BBG, while in the other half BBG appears to dominate Blackbox.

All experiments were conducted on a notebook using a Pentium-III processor running at 850 MHz with 256 MBytes main memory. The source code and testing examples are available by the authors on request.

Parallel Domains

The first parallel domain we consider is the classical *Logistics* domain where packages must be transported between locations by means of trucks (within cities) and aeroplanes (between different cities). The parallelism lies in that all moving actions for different vehicles, as well as all loading / unloading actions for different packages, do not interfere with each other.

We obtained a suit of random *Logistics* instances as follows. We used the generator available from the FF homepage.[6] We generated instances scaling in 5 different sizes, 10 instances per size. The instances all had 4 cities with 2 locations each, 1 truck per city, and 2 aeroplanes. Size was scaled by increasing the number of packages to be transported from 7 to 15 in steps of 2 (we chose these specific size parameters after some experimentation in order to make the resulting instances challenging, but feasible for the planners in question; in comparison, the well-known *Logistics* instance "log-c", e.g., corresponds exactly to our smallest instance set). See the data in Table 1.

[5] As the reader may have noticed, stopping the graph update as soon as the heuristic value is known to be above the deadline is quite closely related to what constraint-based approaches (like (Rintanen 1998; Baioletti, Marcugini, & Milani 2000)) do when they find that the goal constraints can not be met. In difference to that, in our framework one could also do a full graph update until the goals are reached, and keep the resulting heuristic value in memory for use in later search iterations. Here, we exclusively focus on the effects of branching, and leave this possibility as a topic for future work.

[6] At http://www.informatik.uni-freiburg.de/∼hoffmann/ff.html.

	Logistics				
planner	size 1	size 2	size 3	size 4	size 5
BBG-BA-L	1.00	1.00	0.90	1.00	0.70
BBG-BM-L	1.00	1.00	0.90	0.70	0.70
BBG-BA-C	1.00	1.00	1.00	1.00	0.80
BBG-BM-C	1.00	1.00	1.00	1.00	0.80
IPP	1.00	0.80	0.20	0.00	0.10
IPP-N	0.90	0.40	0.20	0.00	0.00

Table 1: Fraction of solved instances in the *Logistics* domain. See explanation in the text.

The 6 planners tested here are the following. BBG-BA-* is BBG with branching on actions scheme, similarly BBG-BM-* branches on mutexes; BBG-*-L is with latest flaw selection, BBG-*-C is with criticality flaw selection. IPP is the IPP 4.0 implementation available from the IPP homepage;[7] IPP-N is the same implementation but with memoization turned off — we include this data as comparing it to BBG, in which no heuristic function update techniques or transposition tables are used, yields a more immediate picture of what the effects of our new branching techniques are.

Each planner was given maximum bounds of 500 seconds runtime, and 256 MBytes memory usage (to avoid swapping). Table 1 shows *the fractions of the 10 instances per size that were solved within these bounds by the respective planners.* Clearly, all BBG configurations scale up much better than the IPP configurations. We also measured the search time spent and the number of search states encountered by the planners. We do not include this data in the tables for two reasons. First, most of the time the fractions of solved instances are already sufficient to distinguish between planners in our experiments here so the additional data would unnecessarily blow up the discussion. Second, taking time averages in the presence of a significant number of unsolved instances is a problem (should unsolved instances be ignored? should they be weighted by the cut-off time/space? ...). For these reasons, we report only some of these data in the text, when directly comparing pairs of planners *where the time and states averages are taken over the instances that are solved by both planners (ignoring the rest).*

In the *Logistics* suit, all BBG configurations behave roughly similar with a slight advantage for BBG-BM-C. On those instances solved by both BBG-BM-C and IPP, the average search times per size are 0.19, 0.45, 0.34, -, 0.31 seconds for BBG-BM-C (in the second largest size class, no instance is solved by both planners), and 1.72, 117.16, 18.16, -, 447.11 seconds for IPP. These measurements are pure search time, i.e., they do *not* include the initial planning graph building phase which, as said before, is somewhat more efficient in BBG due to minor implementation differences. The average numbers of search states visited are 24, 77, 28, -, 22 for BBG-BM-C, and 5786, 104818, 33646, -, 239888 for IPP. The search states we count in BBG are the ones in which a commitment has been made success-

[7] At http://www.informatik.uni-freiburg.de/∼koehler/ipp.html.

fully (without exceeding the current bound on the heuristic function), while in IPP the states we count are the successfully created subgoal sets (the subgoal sets produced after successful backchaining with non-mutex actions, and that are not memoized at the respective graph layer). Clearly, the reduction in the number of states in the search is dramatic in this case (several orders of magnitude), while the reduction in search time is significant, although not as dramatic due to the additional overhead incurred by BBG in the more costly heuristic computation. In comparison to IPP-N, the data are about an order of magnitude more drastic: comparing BBG-BM-C to IPP-N, the search time values are 0.19, 0.22, 0.33, -, - for BBG-BM-C and 2.13, 27.97, 296.03, -, - for IPP-N. The search state numbers are 23, 20, 28, -, - respectively 37529, 387862, 1998143, -, -. We have made similar observations — i.e., that the data in direct comparison to IPP-N are around an order of magnitude more drastic than in direct comparison to IPP — in all the other parallel domains below; we do not report all this data as it seems redundant.

Our second parallel domain is a variant of the *Blocksworld* where blocks can be moved in a move-A-from-B-to-C fashion, thus enabling parallel moves of different blocks to different destinations. Our testing suit contains random instances (generated with the generator available from the FF homepage) with 6 to 14 blocks, increasing in steps of 2, 10 instances per size. See the data in Table 2.

Blocksworld					
planner	size 1	size 2	size 3	size 4	size 5
BBG-BA-L	1.00	1.00	1.00	1.00	0.90
BBG-BM-L	1.00	1.00	1.00	0.90	0.90
BBG-BA-C	1.00	1.00	1.00	0.60	0.70
BBG-BM-C	1.00	0.70	0.60	0.40	0.30
IPP	1.00	1.00	1.00	0.80	0.60
IPP-N	1.00	1.00	1.00	0.70	0.60

Table 2: Fraction of solved instances in the *Blocksworld* domain. See explanation in the text.

The tested planners, the time and memory bounds in the experiment, and our form of data presentation are the same as explained above for the *Logistics* experiment. As can be seen from Table 2, in the *Blocksworld* it does make quite a difference which branching scheme and branching point selection criterion are used. Those BBG configurations with latest flaw selection (-L) solve more instances than IPP and are more efficient. The average search times on those instances solved by both BBG-BA-L and IPP are 0.09, 0.20, 0.29, 1.47, 1.53 for BBG-BA-L and 0.00, 0.01, 0.05, 1.44, 7.12 for IPP. The average number of states visited are 5, 31, 17, 73, 52, and 6, 60, 80, 408, 3060 respectively. The reduction in the number of states is smaller in this domain but still significant. On the other hand, the overall reduction in time is moderate, due to larger overhead per node in BBG (once again, note that these figures are averages over the instances that IPP found relatively easy, ignoring the instances solved by BBG but not by IPP). As in *Logistics*, the data in comparison to IPP-N is an order of magnitude more drastic.

Our third test is over the *Satellite* domain as used in the 3rd International Planning Competition (IPC-3) (Fox & Long 2002). A number of satellites must take images of phenomena in space, which involves, amongst other things, turning the satellites into the right direction, and calibrating the instruments. Individual satellites can act in parallel. Our testing suit contains random instances (generated with the official competition generator) with 5 satellites each and 6 to 10 images to be taken, 10 instances per size (we will later also present results for random *Satellite* instances with only 1 satellite, where there is less parallelism). See the data in Table 3.

Satellite — parallel suit					
planner	size 1	size 2	size 3	size 4	size 5
BBG-BA-L	1.00	1.00	1.00	1.00	1.00
BBG-BM-L	1.00	1.00	1.00	1.00	1.00
BBG-BA-C	1.00	1.00	1.00	1.00	1.00
BBG-BM-C	1.00	1.00	1.00	1.00	1.00
IPP	1.00	1.00	0.00	0.00	0.00
IPP-N	1.00	1.00	0.00	0.00	0.00

Table 3: Fraction of solved instances in the *Satellite* domain, (more) parallel random suit. See explanation in the text.

The difference in performance that branching makes in this case is quite notorious (see Table 3). All instances where more than 7 images must be taken are solved by BBG while none is solved by IPP. The smallest instances, on the other hand, are simple and are solved by IPP with no search (BBG solves them as well). The branching scheme and branching point selection make a lot of difference in this case, with advantages for branching on actions (-BA) and latest flaw selection (-L): the average number of states is 9, 110, 1821, 3557, 4567 for BBG-BA-L while it is 8, 190, 12789, 58568, 100817 for BBG-BM-C.

The fourth example domain is *Depots*, which also originates from the IPC-3 suit. It is a combination of *Logistics* and *Blocksworld* where objects ("crates") must be transported in trucks and then be arranged in stacks using a number of robot arms, "hoists". The robot arms can act independently of each other. In all our random examples (generated with the official competition generator) there are 2 locations, and 2 trucks. One can use 7 hoists to arrange 6 to 10 crates, 10 instances per size (we will later also present results for random *Depots* instances with only 2 hoists, where there is less parallelism). See the data in Table 4.

Depots — parallel suit					
planner	size 1	size 2	size 3	size 4	size 5
BBG-BA-L	1.00	1.00	1.00	1.00	0.70
BBG-BM-L	1.00	1.00	1.00	1.00	0.60
BBG-BA-C	1.00	0.60	0.30	0.20	0.20
BBG-BM-C	1.00	0.60	0.30	0.00	0.10
IPP	0.80	1.00	1.00	1.00	0.10
IPP-N	0.80	1.00	0.90	0.80	0.10

Table 4: Fraction of solved instances in the *Depots* domain, (more) parallel random suit. See explanation in the text.

In Table 4, we again see a superiority of those BBG configurations that use latest flaw selection (-L). BBG-BA-L, like in the previous domains except *Logistics*, tends to be fastest. Comparing BBG-BA-L to IPP we obtain average search times 0.81, 1.22, 6.24, 5.06, 1.40 for BBG-BA-L and 0.39, 0.59, 19.07, 78.69, 11.08 for IPP (the single largest instance solved by IPP thus appears to be relatively easy). The search state numbers are 235, 199, 435, 662, 66 for BBG-BA-L and 1076, 860, 15432, 21719, 24422 for IPP.

We finally consider the *Mystery* and *Mprime* domains as introduced by Drew McDermott in the 1st International Planning Competition. These are *Logistics* variations where trucks consume one fuel unit per move. The fuel units are associated with the locations (rather than with the trucks), and moving away from a location decrements the amount of fuel available at that location. In *Mprime* there is an additional operator to transfer fuel between locations. To obtain instances that were within the range of solvability of the planners at hand but that still featured a significant amount of potential parallelism we had to restrict the number of locations to 2. Our random instances (generated with the generators available from the FF homepage) then all have 3 trucks (truck transport capacity is limited to a random number between 1 and 3 objects), and scale from 6 to 10 objects to be transported. See the data in Table 5.

Mystery					
planner	size 1	size 2	size 3	size 4	size 5
BBG-BA-L	1.00	1.00	1.00	0.40	0.40
BBG-BM-L	1.00	1.00	1.00	0.40	0.40
BBG-BA-C	1.00	1.00	1.00	0.60	0.50
BBG-BM-C	1.00	1.00	1.00	0.50	0.50
IPP	1.00	1.00	1.00	0.40	0.40
IPP-N	1.00	0.90	1.00	0.40	0.20
Mprime					
BBG-BA-L	0.80	1.00	0.60	1.00	0.50
BBG-BM-L	0.80	1.00	0.50	1.00	0.50
BBG-BA-C	1.00	1.00	1.00	1.00	0.90
BBG-BM-C	1.00	1.00	0.90	1.00	0.70
IPP	1.00	1.00	0.80	1.00	0.50
IPP-N	0.70	1.00	0.50	0.90	0.50

Table 5: Fraction of solved instances in the *Mystery* and *Mprime* domains. See explanation in the text.

In both domains it can be observed that the planners do not perform dramatically different in our example suit, but that there are advantages for the BBG configurations with criticality flaw selection (-C), which solve more instances than the other planners. BBG-BA-C generally performs best (of all the planners). The *Mystery* search times compared to IPP are 1.24, 5.55, 9.40, 7.77, 28.09 for BBG-BA-C and 6.09, 16.31, 17.29, 13.78, 140.40 for IPP; the number of states are 824, 4757, 5729, 5141, 14418 for BBG-BA-C and 17159, 63019, 71707, 55638, 468469 for IPP (as usual the data compared to IPP-N are an order of magnitude more drastic). Like in *Blocksworld*, the reduction in the number of states is quite significant while the reduction in search time is more moderate. Something similar occurs in *Mprime*.

Serial Domains

To test the hypothesis that the alternative branching schemes are suitable in particular for parallel domains where the branching factor of regression search is very high, we also ran experiments with more serial random suits. To outrule, as far as possible, the side effects of structural differences between individual planning domains, we used more serial versions of (four of) the parallel domains described above. In *Logistics* and *Blocksworld*, we introduced artifical preconditions and effects that force actions to be applied sequentially;[8] in *Satellite* and *Depots* we generated random suits with less potential parallelism. The *Logistics* and *Blocksworld* data are displayed in Table 6.

Logistics-serialized					
planner	size 1	size 2	size 3	size 4	size 5
BBG-BA-L	1.00	1.00	1.00	0.80	0.70
BBG-BM-L	1.00	1.00	1.00	0.80	0.50
BBG-BA-C	1.00	1.00	1.00	0.90	0.70
BBG-BM-C	1.00	1.00	1.00	0.80	0.70
IPP	1.00	1.00	1.00	1.00	1.00
IPP-N	1.00	1.00	1.00	0.80	0.70
Blocksworld-serialized					
BBG-BA-L	1.00	0.90	0.90	0.40	0.10
BBG-BM-L	1.00	1.00	0.90	0.30	0.10
BBG-BA-C	1.00	0.50	0.60	0.10	0.00
BBG-BM-C	1.00	0.30	0.60	0.00	0.00
IPP	1.00	1.00	1.00	0.90	0.60
IPP-N	1.00	1.00	1.00	0.40	0.10

Table 6: Fraction of solved instances in *Logistics-serialized* and *Blocksworld-serialized*. See explanation in the text.

To make the serialized *Logistics* instances feasible for our planners, we had to use only a single city with two locations and one truck. The number of objects scales from 5 to 9, like always we have 10 random instances per size. The best BBG planner is BBG-BA-C. Comparing it with IPP, we find the runtime data to be 0.01, 0.02, 0.06, 0.18, 0.39 for IPP, and 4.15, 0.51, 10.68, 68.18, 18.39 for BBG-BA-C. Even the number of search states visited is smaller for IPP, the precise data being 218, 284, 854, 1957, 3350 for IPP and 11305, 890, 19323, 99084, 19846 for BBG-BA-C. So in this case the new branching schemes search over more states, and the overall search takes more time. The comparison to IPP without memoization (IPP-N) is better for BBG: the runtime data are 1.57, 0.25, 3.39, 7.11, 26.18 for IPP-N and 4.15, 0.51, 10.68, 16.39, 18.29 for BBG-BA-C; the states data are 101676, 12467, 147355, 278770, 670861 for IPP-N and 11305, 890, 19323, 25912, 19846 for BBG-BA-C. This suggests that in the serialized form of *Logistics* the alternative forms of branching can reduce the search space, but not as drastically as memoization does; also, we see that the search space reduction in comparison to plain regression,

[8] One copy of the original operators require a new fact f; they delete f, and add f'; a second copy of the operators require f', delete f', and add f. This way all actions interfere with each other while the domain semantics remain the same.

i.e., IPP-N, is not sufficient to make up for the more costly computation.

In our serialized *Blocksworld* suit, we have instances with 8 to 12 blocks, 10 instances per size. The conclusion to be drawn is the same as in *Logistics-serialized*. IPP scales best and IPP-N scales similarly to the best BBG planner, BBG-BA-L. Looking at average runtimes and search state numbers in instances solved by both IPP and BBG-BA-L respectively IPP-N and BBG-BA-L, one finds that the data is somewhat inconclusive in that the runtimes for all the planners vary around a few seconds, the search states for IPP and BBG-BA-L vary around a few hundreds, and the search states for IPP-N vary around a few thousands; in particular, the values for BBG-BA-L become the better the larger the instances become. This phenomenon seems to be due to the fact that the larger instances solved by either BBG-BA-L or IPP-N are the "easy" ones (e.g., where the first graph layer in which the goals are reached already contains a plan). As in *Logistics*, the alternative branching schemes do not seem to help much in this serialized domain.

As said before, our serial *Satellite* (*Depots*) suit differs from the parallel suit in that there is (are) only 1 satellite (only 2 hoists). The resulting data are displayed in Table 7.

Satellite — serial suit					
planner	size 1	size 2	size 3	size 4	size 5
BBG-BA-L	1.00	1.00	1.00	1.00	1.00
BBG-BM-L	1.00	1.00	1.00	1.00	1.00
BBG-BA-C	1.00	1.00	0.80	1.00	0.00
BBG-BM-C	1.00	1.00	0.00	0.00	0.00
IPP	1.00	1.00	1.00	1.00	1.00
IPP-N	1.00	1.00	0.00	0.00	0.00
Depots — serial suit					
BBG-BA-L	1.00	1.00	1.00	1.00	1.00
BBG-BM-L	1.00	1.00	1.00	1.00	0.90
BBG-BA-C	1.00	1.00	0.70	0.50	0.20
BBG-BM-C	1.00	1.00	0.80	0.50	0.20
IPP	1.00	1.00	1.00	1.00	1.00
IPP-N	1.00	1.00	0.90	0.70	0.30

Table 7: Fraction of solved instances in *Satellite*, (more) serial random suit, and *Depots*, (more) serial random suit. See explanation in the text.

In the serial *Satellite* suit, again memoization is a more effective enhancement to basic Graphplan search than alternative branching: the runtimes IPP compared to the best BBG planner BBG-BA-L are 0.01, 0.07, 0.25, 0.36, 1.11 for IPP and 0.16, 0.90, 10.79, 10.51, 204.00 for BBG-BA-L; the search state data is 223, 752, 2164, 2397, 6194 for IPP and 264, 2107, 18698, 12875, 183273 for BBG-BA-L (the BBG-BA-L search space size is still a vast improvement over basic regression search: IPP-N searches 54812 respectively 9116650 states on average in the smallest two groups of instances). In the serial *Depots* suit, the best BBG planner BBG-BA-L is still better than IPP with memoization, but not as much as in the parallel suit: the runtime data are 0.00, 0.01, 0.47, 5.94, 49.46 for IPP and 0.12, 0.17, 0.87, 2.10, 28.86 for BBG-BA-L; the search state data are 39, 113, 1382, 9768, 59587 for IPP and 50, 50, 418, 488, 7793 for BBG-BA-L. The data for BBG-BA-L compared to IPP-N is, like it was in the parallel domains, about an order of magnitude more drastic.

Finally, we have also collected some data to directly support our hypothesis on what the reasons are for BBG's performance in difference to IPP. To assess the degree of parallelity in a planning task, and its effect on the branching factor during Graphplan-style backchaining, we have measured the following parameters: *normalized potential parallelism* — averaged over all actions a, the quotient between the number of actions that a does not interfere with and the total number of actions; *IPP branching factor* — averaged over all non-leaf nodes n in IPP's search space, the number of subgoal sets produced by successful backchaining from n (including memoized subgoal sets). See the data in Table 8.

domain	size 1	size 2	size 3	size 4	size 5
average normalized potential parallelism					
Logistics	0.96	0.97	0.97	0.98	0.98
Logistics-ser.	0.00	0.00	0.00	0.00	0.00
Blocksworld	0.42	0.52	0.60	0.65	0.69
Blocksworld-ser.	0.00	0.00	0.00	0.00	0.00
Satellite-par.	0.93	0.94	0.94	0.95	0.95
Satellite-ser.	0.68	0.71	0.73	0.76	0.77
Depots-par	0.68	0.69	0.70	0.71	0.72
Depots-ser.	0.47	0.47	0.48	0.49	0.49
Mystery	0.74	0.74	0.74	0.75	0.75
Mprime	0.74	0.74	0.75	0.76	0.76
average IPP branching factor					
Logistics	59.53	176.26	167.98	-	133.61
Logistics-ser.	2.52	2.50	3.59	3.79	5.56
Blocksworld	1.18	4.35	6.71	7.91	46.46
Blocksworld-ser.	3.65	4.89	4.28	6.68	5.32
Satellite-par.	1.00	1.17	-	-	-
Satellite-ser.	9.85	12.32	14.39	15.90	17.88
Depots-par.	13.14	24.78	55.07	84.71	54.79
Depots-ser.	2.06	4.57	10.17	13.59	17.44
Mystery	66.08	30.65	74.45	107.47	131.87
Mprime	47.50	48.51	50.68	46.22	67.28

Table 8: Parallelism and regression branching factor data across all domains. See explanation in the text.

The data in Table 8 are averaged over all instances in the respective sets; the IPP branching factor was only computed when we had a complete search space, i.e., when IPP solved the task. Clearly, in *Logistics*, *Blocksworld*, *Satellite*, and *Depots* the parallel suits contain a higher potential parallelism (in the artificially sequentialized versions of *Logistics* and *Blocksworld* there is no such parallelism at all); likewise clearly, except in *Satellite* where the only parallel instances solved by IPP get solved without any search at all, the IPP branching factor in the parallel suits is higher. This strongly supports our hypothesis about the effect of parallelism on Graphplan's backward search, and thus at least helps to explain the performance results we have seen earlier. Note that making sensible inter-domain comparisons here is difficult; there is certainly a correlation between the normalized po-

tential parallelism and the relative behavior of IPP and BBG that one can expect, but the exact form of the correlation appears to depend on the individual structure of the domain (some more on this later in the discussion of open questions).

Comparison with the State-of-the-art

In order to assess how far the alternative branching schemes in Graphplan approach the resulting planner to state-of-the-art parallel planners, we report some results comparing one of the options in BBG, BBG-BA-L (i.e., BBG with branching on actions schemes and latest flaw selection criterion) against Blackbox using Chaff. Considering the significant success of Chaff in the SAT community, we figure that Blackbox using Chaff is one of the strongest SAT-based optimal parallel planners around, perhaps the best one. We ran Blackbox on the same sets of example suits as described in the previous sections. The resulting data are displayed in Table 9.

domain	planner	size 1	size 2	size 3	size 4	size 5
Logistics	BBG	1.00	1.00	0.90	1.00	0.70
Logistics	BBOX	1.00	1.00	1.00	1.00	1.00
Blocksworld	BBG	1.00	1.00	1.00	1.00	0.90
Blocksworld	BBOX	1.00	1.00	0.70	0.30	0.00
Satellite-par.	BBG	1.00	1.00	1.00	1.00	1.00
Satellite-par.	BBOX	1.00	1.00	1.00	1.00	1.00
Depots-par.	BBG	1.00	1.00	1.00	1.00	0.70
Depots-par.	BBOX	0.80	0.50	0.40	0.10	0.00
Mystery	BBG	1.00	1.00	1.00	0.40	0.40
Mystery	BBOX	1.00	1.00	1.00	0.70	0.70
Mprime	BBG	0.80	1.00	0.60	1.00	0.50
Mprime	BBOX	1.00	1.00	1.00	1.00	1.00
Logistics-ser.	BBG	1.00	1.00	1.00	0.80	0.70
Logistics-ser.	BBOX	1.00	1.00	1.00	0.90	1.00
Blocksworld-ser.	BBG	1.00	0.90	0.90	0.40	0.10
Blocksworld-ser.	BBOX	0.40	0.00	0.00	0.00	0.00
Satellite-ser.	BBG	1.00	1.00	1.00	1.00	1.00
Satellite-ser.	BBOX	1.00	1.00	1.00	1.00	1.00
Depots-ser.	BBG	1.00	1.00	1.00	1.00	1.00
Depots-ser.	BBOX	1.00	1.00	1.00	0.80	0.20

Table 9: Fraction of solved instances, compared between BBG-BA-L and Blackbox using Chaff. See explanation in the text.

In the *Logistics* (both serial and parallel) as well as the *Mystery* and *Mprime* suits, Blackbox is superior to BBG-BA-L. In the *Satellite* suits the planner performances are similar modulo an average search time factor of 1.45 (in the parallel suit) and 1.69 (in the serial suit) in favor of Blackbox (like for the other planners, we only measured pure search time for Blackbox, i.e., the time spent in Chaff). In the *Blocksworld* and *Depots* suits BBG-BA-L is superior to Blackbox in our experiment. Note that the behavior of Blackbox in these latter 4 suits is quite bad — on the larger instances, the planner often ran out of memory; how much of that phenomenon is due to the Graphplan implementation in Blackbox, or to the size of the CNFs generated, or to the clause learning mechanism in Chaff, is hard to tell. In those examples that do get solved by both planners BBG-BA-L is superior to Blackbox in search time by a factor of 0.37 (in a single parallel *Depots* instance, Blackbox is faster) to 22.91, 7.25 on average (note that Chaff is a highly optimized implementation so this factor is due to *search*, not programming, advantages in BBG).

Discussion

We have introduced two non-directional branching schemes in Graphplan and showed that they significantly improve performance over a number of parallel domains. These results have been achieved by a *pure* modification to the branching scheme, leaving the bounding technique in Graphplan untouched. Indeed, we intentionally did not integrate a number of optimizations that could have accelerated the resulting planner. For example, when introducing a commitment $a[d] = \textbf{true}$ one can exclude all actions at layer $d-1$ that delete some precondition of a. In fact, this is one of the propagation mechanisms used in DPPlan (Baioletti, Marcugini, & Milani 2000). We did not integrate this and other possible propagation techniques in order to not obliterate our focus of research: our goal in this work has been to evaluate the impact of branching in a particular planning setting, not to develop a state-of-the-art optimal parallel planner. The fact that the resulting planner compares well with powerful parallel planners such as Blackbox using Chaff just underlines the importance of branching in planning.

Some relevant open issues are the following. It remains to be determined more conclusively how and when alternative branching helps. The results on our parallel testing suits in difference to those on our serial testing suits suggest that parallelism plays an important role, and for that there is a good explanation - the branching factor during regression - for whose support we have already presented some empirical data. The correlation between parallelism and performance is, however, not straightforward. On the positive side, the performance improvements obtained with BBG are most drastic in *Logistics* and *Satellite*, which are the two domains in our suit with highest potential parallelism. On the negative side, e.g. the improvements in *Depots* are more drastic than in *Mystery* and *Mprime* though our potential parallelism and IPP branching factor data would suggest that the improvements in the latter domains should be at least as high as in the former. It thus remains to investigate other domain characteristics that could play a role, e.g. the quality of the plan graph heuristic function, and to explore the interactions between the different characteristics. The same holds true for the comparison to Blackbox where the relative performance of the planners is yet even more inexplicable — it seems that Chaff is not well suited for the SAT formulae that arise in *Blocksworld*-related planning tasks.

On the algorithmic side, it remains to study other branching schemes and branching point selection heuristics in the setting we have introduced. Finally, our intention is to use these ideas for developing an efficient optimal *temporal* planner, i.e., to generalize from the special case we got here where action durations are uniform.

Appendix: Implementation Details

As said before, our implementation is oriented at the ideas introduced in STAN (Long & Fox 1999). The plan graph is represented as a single layer of facts and actions, where only the layer-dependent information is updated along the time steps. Some crucial informations that must frequently be accessed are kept in bit vectors. Most importantly this is the case for: the precondition-, add-, and delete-list of each action; the set of facts which are mutex with an action precondition at a graph layer; the set of actions which are mutex with all achievers for a fact at a layer; and the set of actions that can achieve a fact at a layer. Full details about the computations are in (Long & Fox 1999); for example, when f-M are the bits for the actions that are mutex with all achievers of a fact f, and f'-A are the bits for the actions that achieve f', then f is mutex with f' iff there is no bit b such that f'-$A[b] = 1$ and f-$M[b] = 0$.

When a new constraint is introduced at a layer d, the graph must be updated by removing a set of actions at layer d, and propagating the changes. In general, we have a layer i, a set ops of actions to be removed at i, and a set $opmutexs$ of action mutexes to be introduced at i. These changes are propagated to the next fact layer as follows. First the bits corresponding to ops are set to 0 in the bit vectors f-A that represent the achievers of the respective facts f, and the facts whose bit vectors become empty are removed — of course, in this and the following removal operations, the facts and actions are not physically removed from the graph but only marked as deleted. Simultaneously, we build a list $chfts$ of facts whose achievers information has changed. Then all facts that are achieved by an action participating in $opmutexs$ are included into $chfts$. For each fact f in $chfts$, we thereafter recompute f-M, i.e., the bit vector representing the actions that are mutex with all achievers of f. Finally for all pairs f and f' of facts such that at least one of them is in $chfts$ it is checked (by comparing the appropriate bit vectors, f-M and f'-A) whether they are now mutex; if so, the respective bit vectors are updated. All propagated changes are kept in memory, and retracted when search backtracks over the decision that caused the propagation. The result of the propagation is a set of facts fts to be removed at $i + 1$ (i.e., the next fact layer), and a set $ftmutexs$ of fact mutexes to be introduced at $i + 1$. Propagating these changes to action layer $i+1$ is relatively simple. First remove all actions that have a fact in fts as precondition. Then proceed over all pairs o and o' of actions such that a pair $f \in pre(o)$ and $f' \in pre(o')$ is in $ftmutexs$: if $o = o'$ then remove o; else, make o and o' mutex. Again, the changes are remembered for later retraction, and the output is a list ops of actions to be removed at $i + 1$, and a set $opmutexs$ of action mutexes to be introduced at $i+1$. From here, we continue as described above until the current deadline layer is reached.

References

Baioletti, M.; Marcugini, S.; and Milani, A. 2000. DPPlan: An algorithm for fast solution extraction from a planning graph. In *Proc. AIPS-00*.

Balas, E., and Toth, P. 1985. Branch and bound methods. In *The Traveling Salesman Problem*. 361–401.

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. IJCAI-95*, 1636–1642.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proc. ECP-99*, 359–371.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AI* 129(1–2):5–33.

Carlier, J., and Pinson, E. 1989. An algorithm for solving the job shop scheduling problem. *Management Science* 35(2).

Do, M. B., and Kambhampati, S. 2000. Solving planning-graph by compiling it into CSP. In *Proc. AIPS-00*, 82–91.

Fox, M., and Long, D. 2002. The third international planning competition: Temporal and metric planning. In *Proc. AIPS-02*, 333–335.

Geffner, H. 2001. Planning as branch and bound and its relation to constraint-based approaches. Technical report, Universidad Simón Bolívar. At `www.ldc.usb.ve/~hector`.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. AIPS-00*, 70–82.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 2001:253–302.

Junghanns, A., and Schaeffer, J. 1999. Domain-dependent single-agent search enhancements. In *Proc. IJCAI-99*.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-based planning. *Proc. IJCAI-99*, 318–327.

Koehler, J.; Nebel, B.; Hoffman, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an ADL subset. In *Proc. ECP-97*, 273–285.

Korf, R. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *AI* 27(1):97–109.

Korf, R. 1993. Linear-space best-first search. *AI* 62:41–78.

Long, D., and Fox, M. 1999. The efficient implementation of the plan-graph in STAN. *JAIR* 10:85–115.

McDermott, D. 1999. Using regression-match graphs to control search in planning. *AI* 109(1-2):111–159.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *DAC2001*.

Reinfeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Rintanen, J. 1998. A planning algorithm not based on directional search. In *Proc. KR'98*, 617–624. Morgan Kaufmann.

Sen, A., and Bagchi, A. 1989. Fast recursive formulations for BFS that allow controlled used of memory. In *Proc. IJCAI-89*, 297–302.

Weld, D. S. 1994. An introduction to least commitment planning. *AIMag* 15(4):27–61.