Ignoring Irrelevant Facts and Operators in Plan Generation

Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler

Institut für Informatik,
Albert-Ludwigs-Universität,
D-79100 Freiburg, Germany
E-mail: ⟨*last name*⟩@informatik.uni-freiburg.de

Abstract. It is traditional wisdom that one should start from the goals when generating a plan in order to focus the plan generation process on potentially relevant actions. The GRAPHPLAN system, however, which is the most efficient planning system nowadays, builds a "planning graph" in a forward-chaining manner. Although this strategy seems to work well, it may possibly lead to problems if the planning task description contains irrelevant information. Although some irrelevant information can be filtered out by GRAPHPLAN, most cases of irrelevance are not noticed.
In this paper, we analyze the effects arising from "irrelevant" information to planning task descriptions for different types of planners. Based on that, we propose a family of heuristics that select relevant information by minimizing the number of initial facts that are used when approximating a plan by backchaining from the goals ignoring any conflicts. These heuristics, although not solution-preserving, turn out to be very useful for guiding the planning process, as shown by applying the heuristics to a large number of examples from the literature.

1 Introduction

It is traditional wisdom that one should start from the goals when generating a plan in order to focus the plan generation process on potentially relevant actions. The GRAPHPLAN system [2], however, which is the most efficient planning system nowadays, builds a "planning graph" in a forward-chaining manner, applying all actions that are possible. While GRAPHPLAN works well on most of the examples known from the literature, one might suspect that larger examples containing information that is irrelevant for a particular task lead to performance problems. In fact, irrelevant information can lead to huge planning graphs even when GRAPHPLAN's feature for filtering out irrelevant facts is used.

Although, at first sight, it might seem to be pathological to have task specifications that contain irrelevant information, this situation occurs naturally when one wants to handle larger domains with varying and diverse goals. For instance, in a robotics domain one may want the robot to transport things, to guide people, to clean rooms, etc. In this case, one wants a domain description with all

the necessary operators and the static domain specification. For a given goal and initial state, most of the domain might be irrelevant, however. Further, even the toy examples from the literature sometimes contain some form of "irrelevant information," which, if removed, leads to better performance of the planning process.

For traditional planning systems that perform a backward-chaining search from the goal to the initial state, irrelevance is supposedly not a problem because such planning systems consider only actions that are relevant for solving the planning problem. Planning systems that do forward-chaining from the initial state may, however, run into performance problems. They probably explore possibilities that can never contribute to achieve the goal. However, when comparing GRAPHPLAN with a traditional backward-chaining planner, such as UCPOP [11], GRAPHPLAN is so much faster than UCPOP that GRAPHPLAN's larger sensitivity to irrelevance does not matter much. Nevertheless, "irrelevant information" can be a serious problem for GRAPHPLAN, as is shown below.

In order to bring the best of the two worlds together, one might think of mixing the backward-chaining with the forward-chaining approach. The basic idea for creating top-down expectations in a planning system is quite simple: do backward-chaining from the goals to the initial facts using the operators and ignoring any conflicts between operator applications. This creates an AND-OR tree, where the AND-nodes are the goals or the preconditions of an (instantiated) operator and the OR-nodes are single ground atoms which can be generated by different (instantiated) operators. However, how long should we grow this tree and what paths are relevant in this tree? Further, if we select paths in the AND-OR tree, what should we do with the nodes on the selected paths? Finally, how can we make sure that growing this tree does not result in an exponential explosion?

In the GRAPHPLAN system, these questions are answered as follows. Grow the tree as long as new ground atoms are produced, consider all paths as relevant and take all ground atoms on these paths as relevant. Finally, since a fixpoint computation is used instead of creating the tree explicitly, the computational costs are bounded polynomially in the number of ground atoms. While this is an efficient and safe strategy, since it does not exclude any solution, it is also very weak. In almost all cases we considered, it does not lead to any reduction in computational costs.

A heuristic guiding the planning process should be *efficient* and *solution preserving*, i.e., it should not exclude possible solutions. Since determining relevant information in planning is usually as hard as planning itself, satisfying both requirements probably leads to quite weak heuristics as in the case of the GRAPHPLAN system. We propose a heuristic based on minimizing the use of initial facts when backchaining from the goals to the initial facts, which is similar to McDermott's *greedy regression graph* heuristic. Based on that, we determine the information which is most likely relevant for the planning process. This heuristic is not solution preserving, but it is computationally very efficient and turns out to be quite effective for a large number of domains. Further, for McDer-

mott's [9,10] *grid world* examples and for a number of examples used by Kautz and Selman [7], our heuristics proved to be very effective, reducing GRAPHPLAN's planning time significantly.

The rest of the paper is structured as follows. The next section discusses the notion of "relevance" and "irrelevance" in plan generation. In Section 3, we then show the effects "irrelevant information" can have on different planning systems. A family of heuristics to determine potentially useful information from the description of a planning task is described in Section 4, and the empirical results of applying these heuristics to a large number of examples are given in Section 5. In Section 6 we discuss the results and conclude.

2 What is "Irrelevant" in Plan Generation?

Sometimes it is intuitively obvious that a planning task description contains irrelevant information. For example, if in a STRIPS [5] blocks-worlds planning task the blocks can have a color, then the colors are irrelevant – provided the goal description does not mention colors or if there are no operators to paint blocks.

There are more subtle cases, however. For example, if a block can be painted while the robot holds it and if the goal description contains information that the color of the block should be the same as in the initial state. Of course, the painting operation is irrelevant. However, it is not obvious how to detect this in a domain-independent way.

An even more subtle case is a blocks-world planning task with a number of additional blocks sitting on the table which are not mentioned in the goal description. These blocks are clearly irrelevant for finding a plan of stacking the relevant blocks, but they are considered by plan generation systems, leading to serious performance problems.

Trying to make the notions of *relevance* and *irrelevance* more concrete, one notes that there are at least two levels on which we can discuss these notions. First of all, there is the *external level*. A planning task description may contain type information, initial facts, and operators that are not needed for a solution.[1] Secondly, we can consider the *internal level* of the planning system. Here, we can consider ground operators or ground facts as relevant or irrelevant. In the following, we refer to operators, type information, initial facts, other ground facts, and instantiated operators as **pieces of information**.

Regardless of the level and the type of information, one can distinguish between different degrees of irrelevance. First of all, a piece of information may never be part of any solution for a given planning task, i.e., it is **completely irrelevant**. Secondly, a piece of information may appear in some plans, but it is not necessary for generating a plan. We call this **solution irrelevance**. If plan length is an issue, one can define the notion of **solution-length irrelevance** in a similar way. Unfortunately, however, these "semantic" notions of irrelevance are

[1] The goal description is, of course, never irrelevant!

computationally as hard as planning itself, at least for the case of propositional STRIPS, which is PSPACE-complete [3,1].

Theorem 1. *For propositional* STRIPS*, deciding* complete irrelevance *and* solution irrelevance *of a piece of information in a planning task description is* PSPACE*-complete under polynomial Turing-reductions.*[2]

Proof sketch. Membership in PSPACE follows from the following simple facts. A fact or operator is *solution-irrelevant*, if its removal does not change the plan existence property, which is in PSPACE, i.e., two calls to an oracle deciding plan existence suffice to decide solution irrelevance. Further, a fact or operator is *completely irrelevant* if adding something to the planning task that makes the fact or operator necessary for achieving the goals changes the plan existence property.

PSPACE-hardness of *solution irrelevance* follows, since by adding an operator or fact to a planning task description that must be used in any plan, we can decide plan existence of the original problem by deciding solution irrelevance of the new piece of information in the modified problem.

PSPACE-hardness of complete irrelevance follows because plan existence can be decided by deciding complete irrelevance of all facts or operators. ■

Approaching the notion of irrelevance from a more syntactic point of view, we consider the process of *chaining* between ground facts. We say that a fact φ **generates** ψ iff there is an operator o that can be instantiated to a (type-consistent) ground operator g in a way such that φ is a precondition of g and ψ is an add-effect of g. Based on this definition we say that φ_0 is **reachable by backchaining** from φ_n iff there exists a sequence of facts $\varphi_0, \varphi_1, \ldots, \varphi_n$ such that φ_i generates φ_{i+1}. Similarly, φ_n is **reachable by forward-chaining** from φ_0 iff φ_0 is reachable by backchaining from φ_n.

Using these notions, one can distinguish between *goal irrelevance* and *initial-state irrelevance*. A piece of information is **goal irrelevant** if it cannot be reached by backchaining from the goals, and a piece of information is **initial-state irrelevant** if it cannot be reached by forward-chaining from the initial state. Both notions imply complete irrelevance and are easily computable. GRAPHPLAN can filter out both kinds of irrelevant information. Initial-state irrelevant information is filtered out in building the planning graph, goal irrelevant information can be filtered out by using an option that leads to computing the ground facts reachable by backchaining from the goals. However, as pointed out above, these notions are also very weak and do not cover the more subtle cases mentioned in the beginning of this section.

3 Empirical Effects of Irrelevance

As mentioned in the Introduction, one would expect that traditional backward-chaining planners will have less difficulties with solution-irrelevant pieces of in-

[2] Using an appropriate definition for solution-length, one could prove the same for solution-length irrelevance.

formation than the GRAPHPLAN system. In order to test this hypothesis, we set up two sets of modified blocks-world planning tasks.

In the first set, we added colors and a `paint` operation to the blocks world, where the `paint` operation has three parameters, namely the block, its previous color, and its new color. Further, painting can only be performed if the robot holds the block. Finally, we specify in the initial conditions that all blocks are white and require in the goal description that they are still white. In the second set of examples, we took ordinary blocks-world planning tasks and added superfluous blocks that sit on the table and are clear. In both sets we varied the number of relevant blocks and the number of irrelevant details, i.e., colors and superfluous blocks.

The results[3] shown in Fig. 1 seem to confirm the hypothesis that backward-chaining planners such as UCPOP[4] are not as much affected as GRAPHPLAN by solution-irrelevant information.[5] The number of colors in a planning task description does not seem to affect the search process of UCPOP at all, while the colors seem to present a problem for GRAPHPLAN. Further, the effect of superfluous blocks for UCPOP is much less dramatic than for the GRAPHPLAN-system.

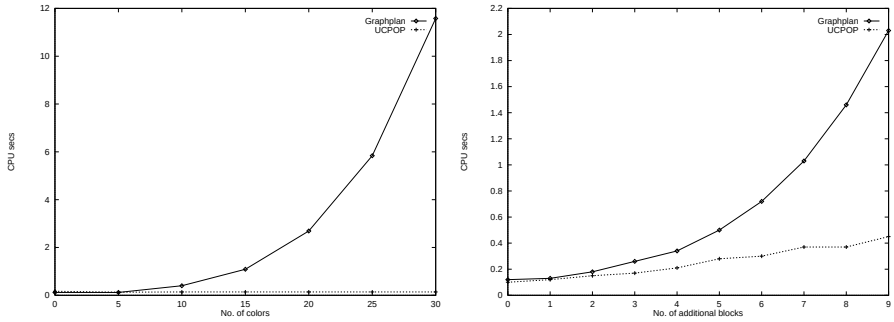

Fig. 1. CPU time for stacking 2 blocks when colors or other blocks are present

These results are quite interesting in themselves because they demonstrate that there are cases where UCPOP can outperform GRAPHPLAN. Having a closer look at GRAPHPLAN, one notes that the increase of runtime is mainly caused by the effort to generate the planning graph. In other words, the conjecture that creating a planning graph in a forward-chaining manner can lead to performance problems in the presence of irrelevant information seems to be justified. However, if we vary the number of relevant pieces of information as well, the picture changes. In Fig. 2, the x-axis measures the number of relevant blocks (note the log-scale for the CPU time).

[3] These and the following results were obtained by a single trial for each data point on a SPARCstation 4/110 with 64MB main memory.

[4] In all our experiments, we used the *zlifo* search strategy [6].

[5] Note that the colors are not goal-irrelevant because colors are mentioned in the goal description. For this reason, GRAPHPLAN cannot detect their irrelevance.

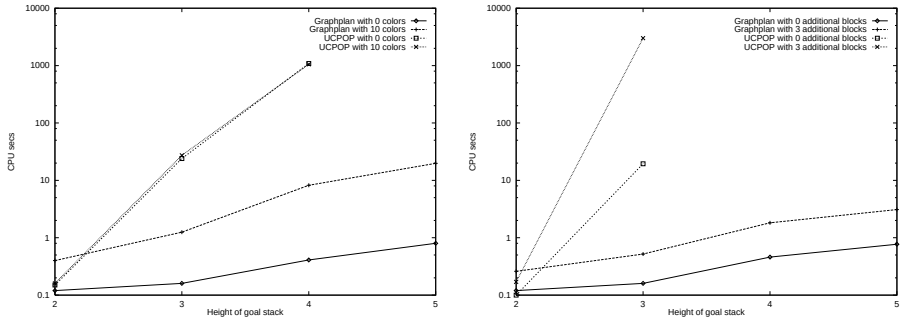

Fig. 2. CPU time for stacking n blocks when colors or other blocks are present

Although GRAPHPLAN seems to be affected by irrelevance in our examples much more than UCPOP, GRAPHPLAN is so much faster that this does not matter. Nevertheless, GRAPHPLAN's sensitivity to irrelevance can be quite serious. For example, a blocks-world planning task with eight blocks and ten colors cannot be solved by GRAPHPLAN without recompiling the planning system using a larger data structure for storing nodes in the planning graph. Further, in the examples with irrelevant blocks, the memory requirements grow with number of blocks regardless of whether the blocks are relevant or not. Since those memory requirements are so severe that they can fill up the memory of ordinary workstations with 64 MB even for planning tasks containing only 15 blocks, any attempt to reduce memory consumption is worthwhile. Finally, the runtime requirements grow with the total number of blocks. In fact, 15 blocks seemed to be the maximum number of blocks GRAPHPLAN could handle in one hour for our example set. So, removing any single irrelevant block can help in making a planning task practically solvable.

4 Selecting Relevant Information by Backchaining and Minimizing the Use of Initial Facts

As sketched above, one can straight-forwardly create top-down expectations in the planning process by backchaining from the goals creating an AND-OR-tree, where the AND-nodes are sets of ground facts (goals or the preconditions of an action) and the OR-nodes are single ground facts that can be generated by different operators. We call such a tree a **fact-generation tree**.

An OR-node of a fact-generation tree is considered to be **solved** if it is an initial fact or if one of its immediate children is solved. An AND-node is solved if all of its children are solved. The entire fact-generation tree is solved if the AND-node corresponding to the goals is solved. The fact-generation tree can be grown until it is solved or a preset depth is reached without a solution.

This is a very crude approximation to the planning process because the state of the world and delete-effects of operators are completely ignored. However, it provides us with a straight-forward and sound method for deciding that a

planning task cannot be solved in d steps (where non-conflicting steps can be executed in parallel as in GRAPHPLAN). In fact, we used this method to debug our planning task specifications, which often turned out to be unsolvable because of trivial typing errors.

Proposition 2. *If a fact-generation tree of depth d does not have a solution, then there does not exist any $\lfloor (d-1)/2 \rfloor$-step plan to solve the corresponding planning task.*

Although the method is straight-forward, it is not by itself computationally efficient. Since the tree grows exponentially with its depth, memory and runtime costs can be quite high even for moderate branching and depth. *Memoizing* the results for each ground fact together with the level in the tree and reusing the result if the new search node is as close to the leave nodes as the memoized results makes the process more efficient.[6] With memoizing, every ground fact will be created at most once per OR-level in the search tree, restricting the number of explored nodes polynomially in the depth of the tree and the size of the instantiated planning task description.

Proposition 3. *Creating a fact-generation tree of depth d can be done in time polynomial in $d \times n \times m$, where n is the number of ground facts and m is the number of ground operators.*

While it suffices to propagate the values *true* and *false* in a fact-generation tree in order to detect unsolvability, we need more sophisticated ways to compute a solution if we want to compute the pieces of information that are probably relevant for the planning process. The idea is to determine a **minimum set of initial facts** (i.e., a set with a minimal number of elements) that are necessary to solve the fact-generation tree. Since there are in general different ways to generate a fact, we determine for every AND- and OR-node the set of sets of initial facts that could be used to generate this particular node. We call these sets of sets **possibility sets**.

In order to illustrate this idea, we present a small artificial example. STRIPS-operators will be written as

$$N : P \rightsquigarrow A/D,$$

where N is the name of the operator, P is the set of preconditions, A is the set of add-effects, and D is the set of delete-effects. We use the following set of operators:

$$\begin{array}{ll} O1\colon & \{a,b\} \rightsquigarrow \{x,z\} \;/\; \{a\} \\ O2\colon & \{b,c\} \rightsquigarrow \{x\} \;\;\;/\; \emptyset \\ O3\colon & \{c\} \rightsquigarrow \{y\} \;\;\;/\; \{c\} \end{array}$$

Further, we assume that $\{a,b,c\}$ is the set of initial facts and $\{x,y\}$ is the set of goals. Then, as demonstrated in Fig. 3, the possibility set for the goals is $\{\{a,b,c\},\{b,c\}\}$, and the minimum set of initial facts that solves the fact-generation tree is $\{b,c\}$.

[6] This means, we effectively create a directed graph instead of a tree.

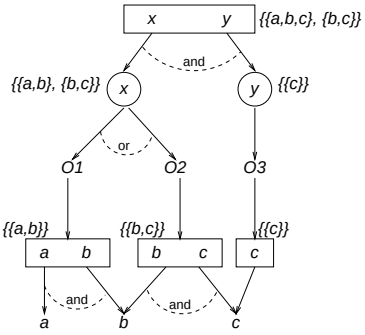

Fig. 3. Example for computing a minimum set of initial facts that can be used to solve the fact-generation tree

The computation of such possibility sets is computationally expensive, however. At each AND-node, the elements of the possibility set result from computing the union over all choices of picking elements from the possibility sets of the OR-nodes immediately below the AND-node. For example, the possibility set of the AND-node $\boxed{x\ y}$ in Fig. 3 is obtained by computing $\{a, b\} \cup \{c\}$ and $\{b, c\} \cup \{c\}$. In general, we may get a possibility set at an AND-node with a number of elements that is the product of the cardinalities of the sets at the OR-nodes immediately below the AND-node. This implies that the number of elements of these sets can grow exponentially with the breadth and depth of the fact-generation tree. Further, there does not seem to be an easy way out here. The problem of computing a minimum set of initial facts solving the fact-generation tree is a computationally hard problem.

Theorem 4. *Computing a minimum set of initial facts that solves a fact-generation tree is* NP*-hard for propositional* STRIPS.

Proof sketch. Follows by a straight-forward reduction of the *hitting set* problem to the problem at hand. ∎

Since the computation of minimum sets of initial facts that are necessary for solving the fact-generation tree is intended to be a heuristic for determining the relevance of pieces of information, we are not forced to compute all elements of a possibility set, however. Further, even non-optimal sets of initial facts can be useful. For this reason, we use a crude approximation. At every node we store only the 10 smallest sets computed so far.

Having computed the (approximation of a) possibility set for the goals, there is the question of what to do with the result. First of all, we must decide what to do in case the possibility set has more than one element. Different methods are conceivable:

1. use the union over all elements in the possibility set,
2. use the union over all set-inclusion minimal sets,
3. use the union over all minimum sets, or

4. pick a minimum set.

Running the heuristic on all the GRAPHPLAN examples and some other examples revealed that most of the time the fourth method, which we call **one-best-set** method, is a good choice. Sometimes, however, this led to longer plans than necessary, because resources that could have been used were removed. Worse yet, in the *rocket* examples it unfortunately removes one rocket so that the planning task becomes unsolvable. In those cases, the third method, which we call **all-best-sets** method, proved to be a safe way out. Finally, for some of McDermott's *grid world* examples, we had to use the first method, called **all-sets** method.

Secondly, we must decide what we want to do with this **set of probably relevant initial facts**. Since the creation of a fact-generation tree is only a very crude approximation of the planning process, initial facts that are necessary for solving the planning task might have been missed. We used three methods of selecting "probably relevant" pieces of information with an increasing degree of restriction:

1. Consider all *objects* mentioned in the set of probably relevant initial facts as relevant and filter out initial facts that contain irrelevant objects.
2. Consider only the *initial facts* in the set of probably relevant initial facts as relevant.
3. Consider only those *ground operators* as relevant that appear in the fact-generation tree and can be generated by the set of probably relevant initial facts.

These methods can be justified as follows. In domains containing a graph that has to be traversed (e.g., McDermott's [9] *Manhattan world*), where undirected edges are represented by pairs of initial facts, often only the first method is useful. The second method is useful if the fact-generation tree is similar to a final plan in that it makes use of all the relevant initial facts. The third method is useful if in building the fact-generation tree all ground operators are used that are necessary for the real plan.

It turned out that for different planning domains different strategies are effective. In the blocks-world domain, for example, a solved fact-generation tree often contains all the necessary ground operators to solve the planning task. In general, it seems to be a good *meta-heuristic* to try the third strategy first, since if it is unsuccessful, GRAPHPLAN fails fast. Then one should try the second and then the first strategy. Since determining the set of "probably relevant initial facts" is also only approximate, this sequence should be interleaved with the *one best set* and *all best sets* strategies described above. In case, we do not get a solution in this way, one may still run the planner on the original problem. In this way, we get (theoretical) completeness although the family of heuristics itself is not solution preserving (see also [4]).

5 Empirical Results

We implemented the family of heuristics described above as a C-program that can be used as a filter for GRAPHPLAN,[7] and tested them on a large set of examples. Applying our heuristics to the examples in Section 3 revealed that they effectively remove the irrelevant information in our blocks-world planning tasks. The CPU time of GRAPHPLAN on the problems and the CPU time of the heuristics combined with GRAPHPLAN are shown in Fig. 4.

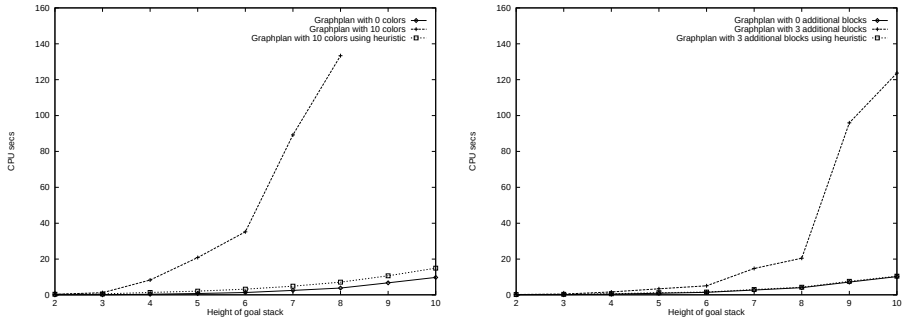

Fig. 4. Graphplan CPU time for stacking n blocks when colors or other blocks are present and when using the heuristic

In order to get an idea of how the heuristics behave on different domains, we ran them on the GRAPHPLAN examples. Runtime is not an issue here, because all problem are solved in a few seconds. More interesting is the question whether the heuristics are *solution-length preserving* or *solution preserving*. An answer to this question is given in Table 1.

In 55% of all examples, it was feasible to use the ground operators that appear in the fact-generation tree basing the selection on the one-best-set method. If the all-best-sets method is used, 86% of the examples could be solved. In case of the *logistics* and *mblocks* domains, however, basing the selection on *one best set* leads to longer plans, because useful resources are removed. In the *rockets* domain, the removal of one of the rockets leads to unsolvability.[8] Even worse is the *tsp* domain. Here, we have to base the selection of relevant information on the *objects* that appear in the set of probably relevant initial facts. The reason is that the undirected edges in the graph are represented by two ground facts, standing for two directed edges. In building the fact-generation tree, however, often only one of those facts is used, i.e., we only traverse the edge in one direction, while in the final plan we must also use the other direction.

More interesting planning tasks are, of course, those that are hard for GRAPHPLAN. One such set of hard tasks is the test set that has been used to demonstrate

[7] The program can be obtained from the authors.

[8] If in the planning task description one could distinguish between usable resources and the initial state information, the heuristic would have a much better way to decide what is relevant.

Table 1. Solution- and solution-length preserving properties of the different heuristics for different domains. An "S" means that the particular heuristic did not change the solution-existence property for all the examples considered, an "L" means that the plans have the same length. If the letter is in parentheses, it means that some examples did not have the property.

	select ground operators		select initial facts		select objects	
	one	all best	one	all best	one	all best
Domain	best set	sets	best set	sets	best set	sets
blocks	(L)	(L)	L	L	L	L
fixit	L	L	L	L	L	L
fridge	L	L	L	L	L	L
link	S	S	S	S	S	S
logistics	S	L	S	L	S	L
mblocks		S	S	L	S	L
monkey	L	L	L	L	L	L
rocket		(L)		(L)		L
tsp					L	L

the performance of SATPLAN [7]. The results for applying our heuristics are given in Table 2.

Table 2. Time steps and actions of generated plan and CPU time on a Sun Ultra 1/170 needed for plan generation on SATPLAN examples. "–" indicates that no solution was found.

	GRAPHPLAN		GRAPHPLAN with heuristic selecting ground operators				GRAPHPLAN with heuristic selecting initial facts			
			one best set		all best sets		one best set		all best sets	
	time	CPU	time	CPU	time	CPU	time	CPU	time	CPU
Task	/act.	secs	/act.	secs	/act.	secs	/act.	secs	/act.	secs
rocket_ext.a	7/34	107	–	0.2	–	0.2	8/27	0.4	7/34	108
rocket_ext.b	7/30	498	–	0.2	–	0.3	10/29	17	7/30	499
logistics.a	11/54	1954	13/51	928	11/54	1654	13/51	3355	11/54	1955
logistics.b	13/45	767	15/42	218	13/45	655	15/42	707	13/45	768
bw_large.a	12/12	1.8	12/12	0.8	12/12	0.8	12/12	0.9	12/12	0.9
bw_large.b	18/18	319	18/18	26	18/18	26	18/18	44	18/18	44

Although the planning tasks descriptions were not intentionally designed to have irrelevant information in it, it turned out that the *bw_large* examples contain irrelevant facts. Further, also the *logistics* and *rocket_ext* examples contain more initial facts than necessary to generate a solution. However, in these cases the facts correspond to resources which, if removed, lead to longer plans. In any case, applying the heuristics seems to pay off on this set of examples. If too

much information is removed, GRAPHPLAN is very fast in detecting this. Further, overall planning time is reduced in almost all cases.

Another interesting planning task is McDermott's [9] *Manhattan world* example.[9] GRAPHPLAN cannot handle it because the memory requirements are too high. Even on an SunUltra I workstation with 1 GB memory, GRAPHPLAN could not find a plan and failed with a memory overflow after 24 CPU hours. Applying our heuristic to the *Manhattan world* example (using *one-best-sets* with *relevant objects* for filtering) returned a reduced planning task after 14 CPU seconds on a Sun Ultra 1/170 . Running GRAPHPLAN on this reduced task resulted in a plan with length 40 after another 12 CPU seconds.[10] McDermott [10] reported that his UNPOP planner needs 16 CPU minutes on average to find a plan with an average length of 52.[11] We also ran our heuristics on the *Sokoban* examples suggested by J. Eckerle and adapted by McDermott [10] with similar results.

6 Discussion and Conclusion

Starting from the observation that the GRAPHPLAN system is highly sensitive to irrelevant information in the planning task description, we developed a family of heuristics aimed at identifying relevant information. These heuristics are very similar to McDermott's [9] *greedy regression graph* heuristic. Minor differences are that AND-nodes are completely instantiated in our case while they may contain variables in McDermott's system. Further, McDermott uses best-first search or discrepancy-search, while we used a simple iterative deepening search. One main difference is that we base our selection on minimum sets of initial facts necessary to solve the fact-generation tree, while McDermott bases his selection on the minimal number of actions. The most striking difference is, however, that we run our heuristic once before planning starts, while McDermott uses his heuristic each time before extending the plan by an action. Nevertheless, our approach results in quite reliable predictions most of the time.

Despite its simplicity, the heuristics turned out to be very useful. In many domains, the most constraining heuristic gives solution-preserving results and reduces planning time considerably. Further, the heuristic enables GRAPHPLAN to find plans for McDermott's *grid world* examples.

Although we used the heuristic only in combination with GRAPHPLAN, it can, of course, be combined with any other planner based on the STRIPS formalism such as SATPLAN [7]. We already extended the heuristics to deal with conditional

[9] We used a representation of the domain tailored to GRAPHPLAN, e.g., the conditional operators are transformed to a set of unconditional operators. Further the grid is encoded as a graph so that our heuristic can identify probably relevant objects.

[10] McDermott [9] claimed that 43 steps is the optimum for the example, but in fact only 40 steps are necessary.

[11] McDermott's results were obtained on a SPARCstation 2, which is approximately 15 times slower than a Sun Ultra 1/170.

operators and integrated it in our extension of the GRAPHPLAN-planner [8] with promising results.

Acknowledgements

We would like to thank Alfonso Gerevini and the anonymous reviewers for comments on an earlier version of this paper.

References

1. C. Bäckström. Equivalence and tractability results for SAS^+ planning. In B. Nebel, W. Swartout, and C. Rich, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference (KR-92)*, pages 126–137, Cambridge, MA, Oct. 1992. Morgan Kaufmann.
2. A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):279–298, 1997.
3. T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.
4. T. A. Estlin and R. J. Mooney. Multi-strategy learning of search control for partial order planning. In *Proceedings of the 13th National Conference of the American Association for Artificial Intelligence (AAAI-96)*, Portland, OR, July 1996. MIT Press.
5. R. E. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
6. A. Gerevini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5:95–137, 1996.
7. H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference of the American Association for Artificial Intelligence (AAAI-96)*, pages 1194–1201, Portland, OR, July 1996. MIT Press.
8. J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proc. European Conference on Planning 1997*, Toulouse, France, September 1997.
9. D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 142–149. AAAI Press, Menlo Park, 1996.
10. D. McDermott. Using regression-match graphs to control search in planning. Manuscript submitted for publication, 1997.
11. J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, W. Swartout, and C. Rich, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference (KR-92)*, pages 103–114, Cambridge, MA, Oct. 1992. Morgan Kaufmann.