

Improving Planning Graph Analysis for Artificial Intelligence Planning

Martin G. Marchetta^{1,2} and Raymundo Q. Forradellas¹

¹ Logistics Studies and Applications Centre (CEAL),
School of Engineering, National University of Cuyo,
Centro Universitario, CC405 (M5500AAT) Mendoza, Argentina

mmarchetta@fing.uncu.edu.ar

kike@uncu.edu.ar

<http://fing.uncu.edu.ar/ceal/>

² CONICET, Argentina

Abstract. Among the most recently proposed approaches for planning, those based on HTN (Hierarchical Task Networks) planning have been the most promising ones. However, techniques based on planning graph analysis have some interesting properties and usually require less effort for modelling planning domains. Since the appearance of the Graphplan algorithm, several improvements and enhancements have been developed following this line. Despite this, there are some special cases for which both the performance of the algorithm and the expressiveness of the domain representation language are not good enough. In this paper some improvements on both performance and expressiveness of the planning language are presented. Results of experiments carried out on test cases more complex than usual ones are also presented, as a preliminary benchmark for the proposed improvements.

Key words: Planning, Artificial intelligence, Graphplan, Planning graph

1 Introduction

Planning has received much attention since several years ago, and it is still one of the most active research areas within artificial intelligence. The planning problem may be defined as follows: given an initial state I , a goal state G and a set of operator schemas O , find a set of actions A (possibly including ordering and causal constraints) such that G can be achieved by executing A from I .

Several kind of approaches have been proposed for solving planning problems. In recent years, planners based on HTN formalism have been the most promising and spread ones, specially for performance reasons [1]. However, HTN planners usually require more knowledge representation for modelling the planning domain, since not only operator schemas, but also decomposition methods must be specified. Planners based on planning graphs, on the other hand, have some interesting properties like soundness and completeness, and at the same time they only require the specification of operator schemas.

Since Graphplan [2] appeared, several improvements have been proposed. Weld et. al. [3] propose expressiveness extensions in order to support uncertainty handling, and sensing actions. Anderson et. al. [4] also proposed expressiveness enhancements in order to support operators with conditional effects.

Despite these improvements, the main drawback of these planners is that on some complex domains, the generated graph may have a big size. Besides, some realistic domains contain operators with many constraints which produce manageable graphs, but even on these cases if constraints are very coupled the construction of the planning graph may take too much time. Additionally, current implementations of these planners have restricted expressiveness in the domain modelling language, which makes some domains not representable.

In this paper, some improvements are proposed on the original graphplan algorithm, which address these two problems. Some optimisations on both planning graph generation and the plan search procedures are proposed, which significantly reduce the time required to find solutions in the special cases mentioned above. Additionally, some extensions to the planning language are proposed and implemented, in order to support the modelling of more complex domains.

The paper is organised as follows. Section 2 presents a brief summary of planning by means of planning graph analysis. Section 3 presents the improvements proposed for planning graph generation and plan search, in order to achieve better performance. Section 4 presents the enhancements on the planning language expressiveness proposed and implemented in this work. Experiments on test cases and performance comparisons are presented in section 5. Finally, section 6 presents conclusions of this work and the future work to be done.

2 Planning Graph Analysis

The Graphplan algorithm was first described in [2]. It is a generative system based on a data structure called *planning graph*. The planning graph is a special graph which represents the search space of the particular planning problem to be solved. Graphplan is an iterative algorithm, which alternates graph building stages with solution search stages.

The planning graph is generated by means of a forward-chaining procedure. It is divided in stages or *levels*. There are two type of levels in the planning graph: *proposition* levels and *action* levels. These two kind of levels alternate themselves within the graph.

Each node of the graph represents either an instance of an action schema (i.e. an action schema with all its variables instantiated), or a proposition representing a fact. Nodes in the graph are connected by three type of arcs: *precondition* arcs, *effect addition* arcs and *effect deletion* arcs. Precondition arcs relate an action with the facts that must be true for the action to be applicable, while effects addition and deletion arcs relate an action with its effects, i.e. with the facts that the action makes true and makes no longer true after its execution, respectively.

Additionally, there are mutual exclusion relationships between actions and propositions. Any two propositions or actions at a given graph level are mutually

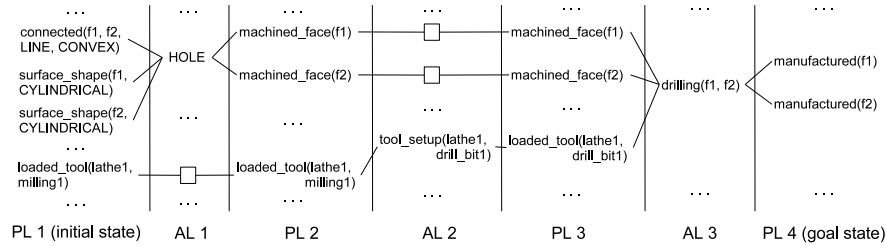


Fig. 1. Example of a planning graph for a manufacturing domain

exclusive, if no feasible plan may contain both of them at the same time in that level. Figure 1 shows an example planning graph.

At each iteration, the algorithm adds a new stage to the planning graph (i.e. adds an action level and the corresponding proposition level). The new stage is created by adding all possible action schema instances whose preconditions are true in the previous proposition level. New actions are analysed in order to find mutual exclusions present in the graph. Finally the effects of new actions are added to the next proposition level. Once the graph has been extended, a search is performed within it, in order to find a feasible solution. This graph-extension/plan-searching procedure is repeated until no new action schemas may be added or until a feasible solution is found.

3 Improving Performance

At a first sight, the generation of the planning graph before starting the search for a solution may seem a waste of resources. However, while building the graph no combinatorial of actions or links is considered, since in this phase the algorithm is not trying to find a solution but to materialise the search space in a compact data structure. This means that no backtracking is performed during graph creation. Besides, during this phase graphplan detects and stores mutual exclusion constraints related to facts and action instances within the graph, which later helps to prune unuseful branches during the search for a solution.

However there are situations on which this approach may have poor performance. One case arises when the generated planning graph has a big number of nodes, which has the double drawback of making both the graph generation and the search procedure very slow, and at the same time of consuming much space. This situation occurs when the following conditions hold:

- a. There are many facts and objects in the initial state
- b. Action schemas available in the domain have few preconditions

The problem here is that many operator instances may have their preconditions satisfied, and thus must be included in the planning graph. Hence the algorithm is not well suited for this kind of problems.

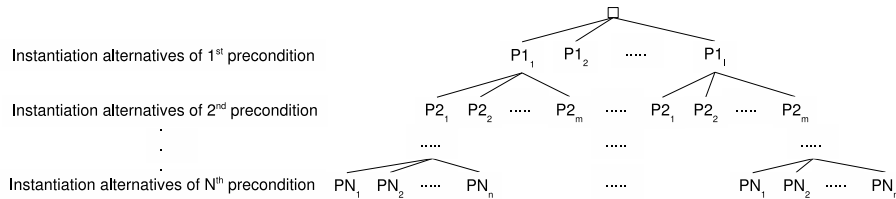


Fig. 2. Search tree for instantiation of preconditions

In addition to this, there is another interesting case on which planners based on planning graph analysis have poor performance. This case arises when the first of the above conditions hold, and at the same time action schemas have many coupled preconditions with many variables instead of having few preconditions. This case is interesting because many real problems require domain models with both complex operators and many initial facts and objects.

3.1 Planning Graph Generation

The performance problem in the situation mentioned above is related to the planning graph generation. Since in this scenario operators have many coupled preconditions, many operator instances are generated and discarded, resulting in much work being done in order to generate relatively small planning graphs.

Graphplan instantiates operator schemas by directly satisfying their preconditions in order, from the first one to the last one, and then checking for types. When the algorithm tries to satisfy a precondition, it looks up within a list containing the facts which are true in the current state of the world, searching for one that unifies with the precondition, considering both its functor and the values already assigned to its variables.

Whenever a precondition can not be satisfied, the algorithm backtracks the search procedure and tries a different instantiation alternative for the previous precondition. Thus, operator instantiation is in fact implemented in graphplan as a depth-first search procedure, as can be seen in figure 2.

Consider the operator schema shown in figure 3. The operator schema has six parameters whose values must be assigned in order to instantiate the operator. The example illustrates a common situation: preconditions are coupled (i.e. they share variables), and some variables which are instantiated while satisfying some

```
(operator DRILLING
  (params
    (<plant> PRODUCTION_PLANT) (<machine> LATHE) (<drill_bit> DRILL_BIT) (<product> PRODUCT) (<diameter> SIZE) (<depth> SIZE)
  )
  (preconds
    (at <machine> <plant>) (at <drill_bit> <plant>) (primary_shaped <product>) (diameter <drill_bit> <diameter>)
    (length <drill_bit> <depth>) (holes <product> <diameter> <depth>)
  )
  (effects
    (holes_drilled <product> <diameter> <depth>)
  )
)
```

Fig. 3. Example of a planning operator schema

precondition may appear in other ones that must be satisfied later. This situation is represented in figure 3 by the $\langle product \rangle$ variable, which is first instantiated in the third precondition, but it appears again in the last one.

In this scenario, poor performance may be obtained if there are many objects and facts in the current state of the world (e.g. if there are many production plants, drill bits, products, etc). Following the example in figure 3, suppose that the algorithm assigns the value *Product1* to $\langle product \rangle$ variable when it satisfies the third precondition, and suppose that *Product1* in fact does not have any holes at all. When the last precondition is reached, it will not be satisfied so the algorithm will backtrack to the previous decision point (i.e. the last instantiated precondition), it will select another alternative and try again to satisfy the last precondition. This procedure will be repeated until all combinations of alternative instantiations under that branch of the tree have been checked, and only then the algorithm will backtrack to the previous decision point and select another value for $\langle product \rangle$, which originally caused the problem.

The issue described above arises because the instantiation of a variable of a precondition affects the satisfiability of later ones. This situation occurs when preconditions are coupled, and the problem grows as coupling is higher. The impact of this problem on planning performance depends on the branching factor of the search tree: the bigger the branching factor the higher the impact is. Let b be the branching factor of the tree, and let d be the depth difference between two preconditions P_1 and P_2 , such that the instantiation of one of them affects the satisfiability of the other one. The number of unuseful nodes that potentially may be explored for incompatible values of P_1 and P_2 is $O(b^d)$.

This problem may be significantly reduced by taking advantage of the nature of coupled preconditions. Since a variable may have only one value, when it is instantiated it constrains possible values of variables on other coupled preconditions (and their satisfiability). Thus in this work an early consistency check was implemented during variable instantiation, similar to *arc consistency check* used in constraint satisfaction techniques [5, 6].

Hence, once values have been assigned to variables of a precondition, remaining preconditions are checked in order to determine if they are at least satisfiable. By applying this form of arc consistency check, the algorithm is capable of immediately pruning complete unuseful branches as soon as the inconsistency is found. As an example, in the case of the operator shown in figure 3 as soon as a $\langle product \rangle$ is selected, the algorithm checks if the remaining preconditions are *potentially* satisfied, that is, it checks for each remaining precondition if there is at least a fact in the current state that unifies with it, considering its functor and the values of variables instantiated so far. Even when an inconsistency may be found later, complete branches may be pruned at early stages.

In order to maximise the effect of this consistency check, preconditions are ordered according to the most-restrictive heuristic [6], as a precompilation step. Once operator schemas are read into the planner, preconditions are ordered according to a rank that is computed for each one as the sum of variables of other preconditions affected by its instantiation.

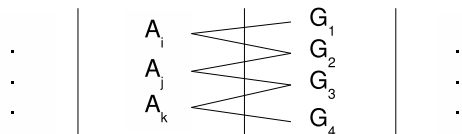


Fig. 4. Example of a non-minimal set of actions

3.2 Plan Search Within Planning Graphs

Some optimisations have also been implemented in the plan search procedure, once the planning graph has been generated. While searching for a solution within the planning graph, graphplan builds sets of actions whose combined effects satisfy all goals in the current level of the graph, starting from the last level (i.e. the level of goals), and going backwards to the initial state. After it has built a suitable set of actions, it determines the goals to be satisfied in the next level, which are the preconditions of actions in the set.

In order to limit the effect of goal orderings, and to avoid including redundant actions within the solution, the algorithm only considers *minimal sets of actions*, which are those for which (a) every goal of the corresponding level is satisfied by an action, and (b) no action may be removed from the set without making (a) no longer true. Figure 4 shows an example of a non-minimal set of actions. When a non-minimal set of actions is found, the search procedure is immediately backtracked and another combination is tried.

When the planning domain contains operator schemas with many effects and much overlapping (i.e. several operator schemas may satisfy the same goal), many non-minimal sets of actions may be created and discarded, making unnecessary slow the search procedure. This situation is similar to that described in the previous section (though it is not as common as it).

In this case, the optimisation implemented is to make an early check in order to detect non-minimal sets of actions as soon as possible. As actions are added to the partial solution, the algorithm updates the count of actions which satisfy each goal. After the achievement of each goal, the algorithm verifies that actions achieving it also achieve other goals not satisfied by other actions (i.e. every action achieving the goal must also achieve another one having a count of 1).

Following the example of figure 4, when A_k is selected to achieve G_4 it also achieves G_3 so all actions which satisfy G_3 are checked. Since the counter of all goals achieved by A_j are greater than 1, this action is redundant and the set is non-minimal, so the algorithm backtracks the search. By doing this, the algorithm does not try to satisfy the remaining goals, pruning all that (unuseful) branches within the search tree.

4 Enhancing domain language expressiveness

STRIPS-like planning languages are considered to be not expressive enough for many domains, specially for realistic ones. Since graphplan first appeared, sev-

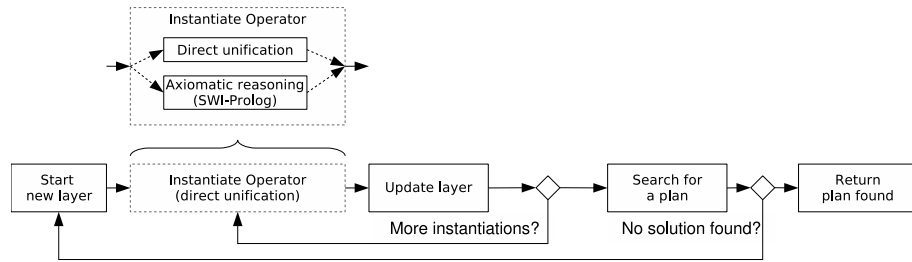


Fig. 5. Graphplan flow and its modification for supporting axiomatic reasoning

eral enhancements have been made to it with respect to its expressiveness, as mentioned in section 1. However there is a useful feature not tackled so far within planning graph analysis, which is indeed supported by modern planners (such as SHOP2 and JSHOP2 [7]), which is axiomatic reasoning.

By means of axiomatic reasoning, a planner can check whether a precondition is true or not by reasoning, instead of simply trying to unify it with a ground fact. In this work these reasoning capabilities have been incorporated into graphplan by integrating the planner with the SWI-Prolog open source prolog engine [8]. Figure 5 depicts the original flow of graphplan and the modification made in order to support axiomatic reasoning.

As can be seen on the figure, the modification made to the algorithm for supporting this kind of reasoning only affects the graph generation phase, since the planning graph is extended before starting each solution search. Prolog preconditions included in operators are distinguished from regular preconditions (satisfiable by direct unification) by a # mark preceding the predicate. Whenever a predicate with that mark is found, the algorithm tries to solve it by means of the integrated prolog engine rather than unifying it with the ground facts.

Each time the graph is extended by adding one more level, facts made true by the previous level are asserted into the prolog knowledge base, in order to allow the engine to satisfy the corresponding preconditions when the main algorithm requires it. In addition to this, values of already instantiated variables are also introduced within the prolog preconditions at the moment of their evaluation. Hence reasoning by means of the SWI-Prolog engine is completely integrated within the main flow of the algorithm, allowing the combination of both direct unification and prolog preconditions.

5 Experiments and test cases

A simple manufacturing domain was used for the tests. The goal is to manufacture a set of engines (products), each of which is composed of 12 parts (1 cylinder head, 4 pistons, 1 engine block, 4 connecting rods, 1 crankshaft and 1 crankcase). Each one of these components may have features of several types (slots, pockets and holes). Some components are manufactured only by machining from stock

Table 1. Performance results obtained from experiments

Problem size Prods. (comp.)	Avg. time (sec.)		Std. deviation		Variance		Time rate (G/G+)
	Graphplan	Graphplan+	Graphplan	Graphplan+	Graphplan	Graphplan+	
2 (24)	9,00	0,230	2,02	0,034	4,06	0,001	39,13
5 (60)	29,59	0,483	5,84	0,027	34,05	0,001	61,26
7 (84)	51,89	0,679	11,72	0,046	137,37	0,002	76,42
10 (120)	87,40	0,965	22,56	0,077	509,16	0,006	90,57
13 (156)	120,90	1,301	15,52	0,113	240,77	0,013	92,93
15 (180)	173,00	1,589	42,99	0,115	1847,78	0,013	108,87
17 (204)	184,10	1,752	24,35	0,096	592,77	0,009	105,08
20 (240)	257,90	2,158	58,32	0,147	3400,77	0,022	119,51

material, while others require a first moulding process. Each type of component feature is produced by a manufacturing process, modelled within the domain as planning operators. There are five operators in the domain: moulding, drilling, pocket rough milling, pocket finish milling and slot milling.

Each component may be manufactured in one of several available production facilities. Each facility has its own machines (moulds of different types, lathes and milling machines), and tools (drill bits and milling cutters, with different diameters and lengths). Features of components can be manufactured only by suitable tools and machines (product holes have diameters and depths that must be met by drill bits, the same applies to slots which have widths and depths, and so on). This yields a domain where operators have an average of 6 preconditions with many couplings. Additionally, many tools, machines, production plants, etc. (i.e. many domain objects) were introduced in problem instances, situation which corresponds to that described in 3.1.

A program was developed for automatically generating instances of the problem described above. 10 random instances of this problem were generated for different sizes of the problem, namely 2 products (24 components), 5 (60), 7 (84), 10 (120), 13 (156), 15 (180), 17 (204) and 20 products (240 components). The true number of goals is greater than the number of components to be manufactured, since each component may have more than one feature (which are the individual goals to achieve). Since the exact number of goals is also randomly generated, results are presented as a function of the number of components.

Table 1 shows the average results obtained from experiments. For each problem size, 10 random instances were solved with both planners and the average, standard deviation and variance of running times were computed and included in the table. From the data shown in the table it can be seen that the modified algorithm has much better performance than the original one, on this kind of problems. The last column of the table shows a rate between average running times of both algorithms, from which it can be seen that improvements range from 1 to 2 orders of magnitude, and the performance difference seems to grow without limit as the problem size grows.

Figure 6.a shows the comparison between both planners. Figure 6.b depicts the performance of the modified version of graphplan (labelled graphplan+). Another conclusion that can be drawn from these charts is that Graphplan

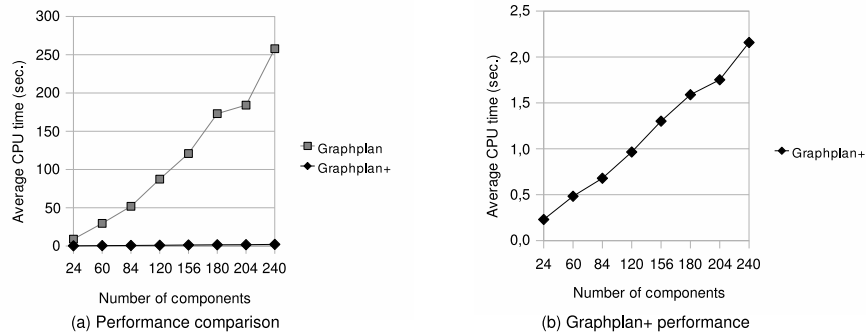


Fig. 6. Performance comparisons between Graphplan and Graphplan+

performance seems to be approximately exponential in the number of goals to be achieved, while Graphplan+ performance seems to be approximately linear.

In order to illustrate the kind of problems on which axiomatic reasoning may be useful, a slightly more complex version of the manufacturing domain described previously is presented here. In the new version of the problem, engines must be assembled as a final stage of production. Before engines can be assembled, all their components must be manufactured. In this case, the problem with STRIPS representations is that each component may have an arbitrary number of features, and hence some kind of universal quantification is needed.

Hence, a new operator named *assemble* (depicted in figure 7.a) was included in the domain. This operator requires as preconditions that there are enough components of each type, that these components have not been used on another assembly, and that all their features have been produced. The first two type of preconditions are solved by the original direct unification mechanism, while the last one is implemented with a prolog precondition. Figure 7.b shows part of the prolog implementation.

6 Conclusions and future work

Even when newer planning formalisms (such as HTN) have had much more popularity in recent years than planning graph analysis, these last have some interesting properties such as soundness, completeness and simple domain modelling requirements. In this paper several enhancements were proposed and implemented on the original graphplan algorithm, which significantly improve both performance and expressiveness of the planning language, enlarging the class of problems addressable by the algorithm.

From the test cases and experiments it follows that some realistic cases with which the original algorithm had problems are now solvable with the proposed enhancements, which is the most important conclusion obtained from this work. Future work will include further development on both performance and expressiveness issues, mostly related to current trends in planning [9]. From the perfor-

```

(operator ASSEMBLE
  params
    (<engine> product) (<cylinder_head> product)
    (<piston1> product) (<piston2> product)
    ...
  preconds
    (product_type <cylinder_head> cylinder_head)
    (product_type <engine_block> engine_block)
    ...
    (not_used <cylinder_head>)
    (not_used <crankshaft>) (not_used <piston1>)
    ...
    (#manufactured(<cylinder_head>))
    (#manufactured(<engine_block>))
    ...
  effects
    (assembled <engine>)
    (del not_used <cylinder_head>)
    (del not_used <engine_block>)
    ...
)

```

(a) Part of the ASSEMBLE operator

```

manufactured(Component) :-
  primary_shaped(Component),
  holesDimensions(Component, HolesDimensions),
  slotsDimensions(Component, SlotsDimensions),
  ...
  holesMachined(Component, HolesDimensions),
  ...
holesDimensions(Component, HolesDimensions) :-
  forall([Diameters, Depths],
    holes(Component, Diameters, Depths),
    HolesDimensions).
...
holesMachined(_Component, []).
holesMachined(Component, [[Diameter,Depth]]Rest) :-
  holes_drilled(Component, Diameter, Depth),
  holesMachined(Component, Rest).
...

```

(b) Partial Prolog implementation of manufactured/1 precondition

Fig. 7. Operator with axiomatic preconditions and their Prolog implementation

mance point of view, arc consistency checks during the search phases of planning will be explored, since on domains which generate graphs with many levels the search procedure may take too much time. Considering expressiveness, further features need to be developed, including list-type variables handling, number-type variables (whose existence in the current world state must not be checked), support of internal variables within operators, allow to disable unique value assignment checks for variables, support of operator priorities, and negated preconditions, among others. Finally, another important feature that should be developed is supporting the search for optimal plans.

References

1. Nau, D., Au, T., Ilghami, O., Kuter, U., Wu, D., Yaman, F., Muñoz-Avila, H., Murdock, J.W.: Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20 (2), 34–41 (2005).
2. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. *Artificial Intelligence* 90, 281–300 (1997).
3. Weld, D.S., Anderson, C.R., Smith, D.E.: Extending Graphplan to Handle Uncertainty and Sensing Actions. In *Proceedings of AAAI-98*, pages 897–904 (1998).
4. Anderson, C.R., Weld, D.S., Smith, D.E.: Conditional effects in Graphplan. In *Proceedings of the 4th Int. Conf. AI Planning Systems* (1998).
5. Van Hentenryck, P.: Constraint satisfaction using constraint logic programming. *Artificial Intelligence* 58, 113–159 (1992).
6. Russel, S., Norvig, P.: *Artificial Intelligence: A modern approach*. Prentice Hall, 2nd Ed. (2002).
7. Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F.: SHOP2: An HTN Planning System. *J. of Artificial Intelligence Research* 20, 379–404 (2003).
8. SWI-Prolog project, Human Computer-Studies Laboratory (HCS), University of Amsterdam, <http://www.swi-prolog.org/>.
9. Nau D.S.: Current Trends in Automated Planning. *AI Magazine* 28(4), 43–58 (2007).