Alexander Nareyek (Ed.)

LNAI 2148

Local Search for Planning and Scheduling

ECAI 2000 Workshop
Berlin, Germany, August 2000
Revised Papers

Springer

Lecture Notes in Artificial Intelligence 2148

Subseries of Lecture Notes in Computer Science
Edited by J. G. Carbonell and J. Siekmann

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editor

Alexander Nareyek
GMD FIRST
Kekuléstr. 7, 12489 Berlin, Germany
E-mail: alex@ai-center.com

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Local search for planning and scheduling : revised papers / ECAI 2000 Workshop, Berlin, Germany, August 21, 2000. Alexander Nareyek (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2148 : Lecture notes in artificial intelligence)
ISBN 3-540-42898-4

CR Subject Classification (1998): I.2.8, F.2.2, G.1.6, G.2.2

ISBN 3-540-42898-4 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10845478 06/3142 5 4 3 2 1 0

Preface

With the increasing deployment of planning and scheduling systems, developers often have to deal with very large search spaces, real-time performance demands, and dynamic environments. Complete refinement methods do not scale well, making local search methods the only practical alternative. A dynamic environment also promotes the application of local search, the search heuristics not normally being affected by modifications of the search space. Furthermore, local search is well suited for anytime requirements because the optimization goal is improved iteratively. Such advantages are offset by the incompleteness of most local search methods, which makes it impossible to prove the inconsistency or optimality of the solutions generated. Popular local search approaches include evolutionary algorithms, simulated annealing, tabu search, min-conflicts, GSAT, and Walksat. The first article in this book – an invited contribution by Stefan Voß – gives an overview of these methods.

The book is based on the contributions to the Workshop on Local Search for Planning & Scheduling, held on August 21, 2000 at the 14th European Conference on Artificial Intelligence (ECAI 2000) in Berlin, Germany. The workshop brought together researchers from the planning and scheduling communities to explore these topics with respect to local search procedures. After the workshop, a second review process resulted in the contributions to the present volume.

Voß's overview is followed by two articles, by Hamiez and Hao and Gerevini and Serina, on specific "classical" combinatorial search problems. The article by Hamiez and Hao addresses the problem of sports-league scheduling, presenting results achieved by a tabu search method based on a neighborhood of value swaps. Gerevini and Serina's article addresses the topic that dominates the rest of the book: action planning. It builds on their previous work on local search on planning graphs, presenting a new search guidance heuristic with dynamic parameter tuning.

The next set of articles deal with planning systems that are able to incorporate resource reasoning. The first article, of which I am the author, makes it clear why conventional planning systems cannot properly handle planning with resources and gives an overview of the constraint-based EXCALIBUR agent's planning system, which does not have these restrictions. The next three articles are about NASA JPL's ASPEN/CASPER system. The first one – by Chien, Knight, and Rabideau – focuses on the replanning capabilities of local search methods, presenting two empirical studies in which a continuous planning process clearly outperforms a restart strategy. The next article, by Engelhardt and Chien, shows how learning can be used to speed up the search for a plan. The goal is to find a set of search heuristics that guide the search as well as possible. The last article in this block – by Knight, Rabideau, and Chien – proposes and demonstrates, a technique for aggregating single search moves so that distant states can be reached more easily.

The last three articles in this book address topics that are not directly related to local search, but the described methods make very local decisions during the search. Refanidis and Vlahavas describe extensions to the GRT planner, e.g., a hill-climbing strategy for action selection. The extensions result in much better performance than with the original GRT planner. The second article – by Onaindia, Sebastia, and Marzal – presents a planning algorithm that successively refines a start graph by different phases, e.g., a phase to guarantee completeness. In the last article, Hiraishi and Mizoguchi present a search method for constructing a route map. Constraints with respect to memory and time can be incorporated into the search process.

I wish to express my gratitude to the members of the program committee, who acted as reviewers for the workshop and this volume. I would also like to thank all those who helped to make this workshop a success – including, of course, the participants and the authors of papers in this volume.

June 2001

Alexander Nareyek
Workshop Chair

Table of Contents

Invited Paper

Combinatorial Optimization

Planning with Resources

Related Approaches

Incremental Local Search for Planning Problems

Eva Onaindia, Laura Sebastia, and Eliseo Marzal

Dpto. Sistemas Informaticos y Computacion
Universidad Politecnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
{onaindia,lstarin,emarzal}@dsic.upv.es

Abstract. We introduce a new approach to planning in STRIPS-like domains based on an incremental local search process. This approach arises as an attempt to combine the advantages of a graph-based analysis and a partial-order planner. The search process is carried out by a four-stage algorithm. The starting point is a graph, which totally or partially encodes the planning problem. The aim of the second phase is to obtain a first set of actions of a solution plan, the third stage guarantees the completeness and optimality of the generated solution and the fourth stage, a partial-order planner, completes the process by finding the missing actions of the final solution plan, if any.

1 Introduction

Graphplan-like or SATPLAN-like planners have shown to outperform classical planners for most of the standard planning domains. However, these two propositional approaches do not exhibit good results for large-sized problems due to the size of the graph they have to deal with. We tested STAN [3] and Blackbox [6] on large problems from the *blocksworld* domain and noticed that none of them were able to solve problems involving more than fifteen blocks.

Our motivation is to develop a new planning approach, which also offers a good performance for large-sized problems. In order to tackle this issue, we introduce a search method, which integrates a technique that incrementally exploits the problem knowledge and a Partial-Order Planner (POP). Our method is executed in four stages:

- The aim of the first stage is to generate a graph containing a set of actions. This graph may include all actions of a solution plan.
- The result of the second stage is a more refined graph which only contains a subset of actions which will necessarily appear in a correct solution.
- The third stage takes the solution obtained from the previous stage and returns a new improved partial plan. The purpose of this stage is to guarantee the completeness and optimality of the generated solution, i.e., to ensure that (a) this partial solution will eventually lead to a final plan, if a solution exists for the given problem, and (b) the plan will be optimal. This is achieved by finding a partial consistency order between the action nodes in the graph.

A. Nareyek (Ed.): Local Search for Planning and Scheduling, LNAI 2148, pp. 139–157, 2001.

At this stage unsolvable problems are detected and optimal solutions are found. In some cases, the graph resulting from this phase will comprise all the actions of a final solution plan.
- The fourth stage implements a POP which is aimed at adding the missing actions for the final plan and finding a total ordering relation among all the actions in the plan.

We present here a local search approach to planning which, starting from an initial graph generated from the problem, iteratively improves the partial plan contained in this graph. The second stage obtains a plan which may be a final valid solution or which possibly contains some unsatisfied preconditions and/or inconsistent ordering relations between actions. Then the third and fourth stages repair the flaws in the plan by choosing the "best" choice for each flaw. This is done by applying different criteria based on the graph properties and heuristics to measure the appropriateness of one choice with respect to another.

2 Creating the Problem Graph

The first phase of the algorithm creates a graph inspired in a Graphplan-like expansion. This graph, named *Problem Graph* (PG), may partially or totally encode the planning problem. The PG is a directed, layered graph with two kinds of nodes (literals and actions) and two kinds of edges (precondition-edges and add-edges). The levels alternate action levels containing action nodes and literal levels containing literal nodes.

- An action level A_j consists of all action instantiations a_{jk} which satisfy these two requirements:
  - all the preconditions of a_{jk} are present in the previous literal level L_{j-1} and
  - a_{jk} does not occur in any previous action level
- A literal level L_j is a set of propositions implictly representing the different world states reachable after executing the actions in A_j. More specifically, the set of literals in L_j is defined as $L_{j-1} \cup \mathsf{AddEff}(a)^1 \;\; \forall a \in A_j$, being a an action instantiation in A_j.

The first level in the PG is the literal level L_0 and it is formed by all the literals in the initial situation. A_1 consists of all action instantiations which are applicable in L_0. L_1 is the set of literals in L_0 plus the add effects of each action in A_1 and so forth. The PG creation terminates when a literal level containing all the literals from the goal situation is reached in the graph or when no new actions can be applied.

It must be noticed that a PG is neither a state-space graph nor a Planning Graph [1]. There are two main differences with respect to a Planning Graph:

---

[1] AddEff, DelEff and Pre stand for the add effects, delete effects and preconditions of an action respectively.

a) Levels in the PG do not stand for time steps but for instantiation steps which can comprise more than one execution step. An action level A_j denotes that all the actions in A_j will be executed at a time step $t \geq j$, and at least one action from A_{j-1} must be executed firstly.
b) Our PG does not take into account mutual exclusion relations between actions, so two actions in a level may interfere with each other, be complementary or independent actions. The PG is created by a forward-chaining process which simply adds the positive effects of actions.

For all the tested domains (see Section 5), except the *hanoi* problem, all the necessary actions of a valid solution already appeared in the PG. This cannot be always guaranteed because, as it was said above, the PG generation terminates when all the literals from the goal situation are present in a literal level, even though additional actions could be applied in this final level. The way of creating the PG ensures that the majority of propositions that would be generated with a systematic search method are obtained once the final literal level is reached.

The advantage of the PG is that its size is much smaller than the Planning Graph and the cost of creating this graph is hardly appreciable even when dealing with large-sized problems.

In the following, we illustrate the process of creating the PG for the *Sussman* anomaly problem in the *blocksworld* domain. Table 1 shows the initial literal level L_0 which consists of one node for each proposition in the initial situation. Action level A_1 contains the two applicable actions in the initial situation, unstack C A and pickup B, and L_1 all the literals at L_0 plus the add effects of the two actions at A_1. The column to the right of A_1 shows the numbers given to the preconditions and add effects of each action (P stands for preconditions and E stands for effects).

The process carries on with action level A_2 (Table 2). At this level, seven action instantiations, different from those in A_1, are found. The literal level L_2 is created by adding the add effects of the actions at A_2 (literals numbered from 10 to 14). This table also shows the final action and literal levels. The two literals

Table 1. Problem graph for the *Sussman* anomaly (1)

L_0		A_1		L_1	

on A table	1	unstack C A	P={4,5,6}
			E={7,8}
				on A table	1

clear B	2	pickup B	P={2,3,6}
			E={9}
				clear B	2

on B table	3			on B table	3
on C A	4			on C A	4
clear C	5			clear C	5
arm-empty	6			arm-empty	6
				holding C	7
				clear A	8
				holding B	9

Table 2. Problem graph for the *Sussman* anomaly (2)

A_2		L_2		A_3		L_3	

pickup A	P={1,6,9}
	E={10}
		on A table	1	unstack C B	P={5,6,12}
					E={2,7}
						on A table	1

putdown B	P={8}
	E={2,3,6}
		clear B	2	unstack B C	P={2,6,13}
					E={5,8}
						clear B	2

putdown C	P={7}
	E={5,6,11}
		on B table	3	unstack B A	P={2,6,14}
					E={8,9}
						on B table	3

stack C B	P={2,7}
	E={5,6,12}
		on C A	4	stack A B	P={2,10}
					E={6,9,15}
						on C A	4

stack C A	P={7,9}
	E={4,5,6}
		clear C	5	stack A C	P={5,10}
					E={6,9,16}
						clear C	5

stack B C	P={5,8}
	E={2,6,13}
		arm-empty	6	putdown A	P={10}
					E={1,6,9}
						arm-empty	6

stack B A	P={8,9}
	E={2,6,14}
		holding C	7			holding C	7

		holding B	8			holding B	8
		clear A	9			clear A	9
		holding A	10			holding A	10
		on C table	11			on C table	11
		on C B	12			on C B	12
		on B C	**13**			**on B C**	**13**
		on B A	14			on B A	14
						on A B	**15**
						on A C	16

from the goal situation, on B C and on A B, are present at literal level L_3 and the PG creation is completed.

3 Description of the Search Method

The search process itself is carried out in two stages. The output from both stages is a graph, each representing an improved plan.

3.1 The Basic Graph

The objective of this phase is to find the appropriate actions to satisfy the goal literals, then the actions to satisfy the subgoals (preconditions) of the former actions and so on. The result is a directed, layered graph with only action nodes named *Basic Graph* (BG). The number of levels in the BG is the number of action levels in the PG plus two additional levels, an initial and a final action level. The former contains one action with effects and no preconditions and the latter contains one action with preconditions and no effects. The effects of the initial action a_0 are the literals in the initial situation and the preconditions of the final action a_n are the goal literals. An edge or causal link $a_i \rightarrow a_j$ denotes that a precondition of a_j is solved by means of an add effect of a_i.

The process starts with the preconditions of a_n and attempts to find a set of actions in the same or any previous action level having these goals as add effects. The preconditions of these actions form a new set of subgoals and this operation is repeated until each literal has been processed.

In order to find a consistent causal link for each subgoal, the search method applies the following property:

Property 1 (literal consistency). *A literal p required by an action a_m ($p \in \mathsf{Pre}(a_m)$) is said to be consistent if these two requirements hold:*

1. *there is a sequence of actions $a_i \rightarrow a_{i+1} \dots a_{m-1} \rightarrow a_m$ such that $p \in \mathsf{AddEff}(a_i)$ and $p \notin \mathsf{DelEff}(a_j)\ \forall j \in [i+1, m-1]$.*
2. *for each action a_k such that $p \in \mathsf{DelEff}(a_k)$ there is a sequence $a_k \rightarrow a_{k+1} \dots a_{m-1} \rightarrow a_m$ with an action a_l, $l \in [k+1, m-1]$, such that $p \in \mathsf{AddEff}(a_l)$.*

The first part of the property states that there must exist an action with p as an add effect and no actions deleting p must appear after that action (Figure 1(1)). The second part states that for each action which deletes p there must exist an action ordered after it which produces p (Figure 1(2)).

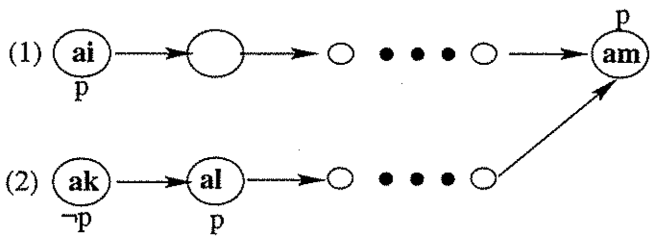

Fig. 1. Literal consistency property. Literals above each node represent the preconditions of actions and literals below the nodes denote add and delete effects.

In order to check literal consistency it is necessary to propagate effects of an action a_i each time a causal link $a_i \rightarrow a_j$ is asserted. The propagated effects of an action a_j are computed by means of the following procedure:

1. $\mathsf{PDelEff}(a_0) = \mathsf{DelEff}(a_0)$
   $\mathsf{PAddEff}(a_0) = \mathsf{AddEff}(a_0)$
2. Let $p_0, p_1, \dots, p_n$ be the paths in the graph that have a_j as destination node. Let $A = \{a_{0,j-1}, a_{1,j-1}, \dots, a_{n,j-1}\}$ be the set of predecessor actions of a_j, each corresponding to a path.

a) $\mathsf{PAddEff}(a_j) = \{x \in \mathsf{PAddEff}(a_i) : a_i \in A/(\exists a_k \in A \wedge x \in \mathsf{PDelEff}(a_k)) \rightarrow a_k < a_i$ [2]$\}$

$\mathsf{PDelEff}(a_j) = \{x \in \mathsf{PDelEff}(a_i) : a_i \in A/(\exists a_k \in A \wedge x \in \mathsf{PAddEff}(a_k)) \rightarrow a_k < a_i\}$

b) $\mathsf{PAddEff}(a_j) = \mathsf{PAddEff}(a_j) - \mathsf{DelEff}(a_j) \cup \mathsf{AddEff}(a_j)$
$\mathsf{PDelEff}(a_j) = \mathsf{PDelEff}(a_i) - \mathsf{AddEff}(a_j) \cup \mathsf{DelEff}(a_j)$

The literal consistency property determines that a precondition p of an action a_i is consistent if the propagation of effects from the initial action a_0 to a_i returns that $p \in \mathsf{PAddEff}(a_i)$ and $p \notin \mathsf{PDelEff}(a_i)$ at step 2(a) from the above procedure. Figure 2(a) shows an example of a consistent precondition, p, for the action a_k and Figure 2(b) shows that the precondition p of action a_k is still an inconsistent literal.

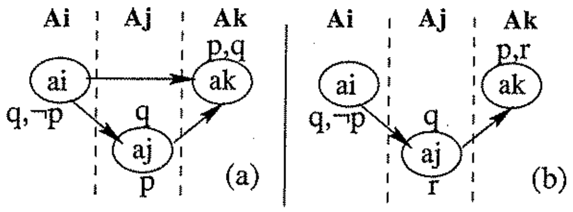

Fig. 2. Example of a consistent and an inconsistent literal

The literal consistency property guarantees that, when a problem is solvable, all the actions in the BG form a part of a correct solution, although all necessary actions may not appear in the BG. However, it is not possible to prove that such a solution exists or that the set of actions leads to an optimal solution. This will be the objective of the next stage.

Basically, the BG represents a plan which contains optimal sequences of actions to achieve each subgoal literal independently. If property 1 does not hold for all action preconditions, then it means that some of the subgoal literals cannot be satisfied and consequently the BG is not created. Let's take the example in Figure 2(b). If p stands for *arm-empty* and the only action having p as an add effect is the initial action a_0 (a_i in the example) and there is not an operator *putdown*, then the BG cannot be created.

The conflicts detected at this stage are only those which appear in a same sequence of actions. Conflict between actions of different sequences will be discovered at the next stage. Finally, it must also be noticed that there might be actions in the PG which belong to a correct solution and are not discovered during the generation of the BG.

---

[2] $a_k < a_i$ denotes an ordering relation between a_k and a_i such that a_k is executed before a_i

Figure 3 shows the BG for the *Sussman* anomaly problem. Action stack A B is introduced in the BG at A_3 to satisfy the precondition 15 of the final action a_n, and action stack B C is included in the BG at A_2 to satisfy literal 13 of a_n. Notice that these are the only actions in the PG which satisfy literals 13 and 15 respectively.

The algorithm proceeds now with the preconditions of the two actions inserted in the BG. Action stack A B requires literals 2 and 10. Literal 10 is only achieved by action pickup A whereas literal 2 can be produced by several actions in the PG in action level A_2 or by action a_0. The link pickup A → stack A B is inserted to satisfy literal 10, and the resolution of literal 2 is postponed until more information is available. Similarly, the action to satisfy the precondition 8 of stack B C is clearly identified whereas there are more than one choice for literal 5 in the PG (several actions in action level A_2 or action a_0).

The introduction of action pickup B to satisfy the precondition 8 of action stack B C causes precondition 2 of action stack A B to become an inconsistent literal, as the propagation of effects deletes this literal. Then it is necessary to find an action between A_1 and A_3 to restore literal 2; since there is already an action at A_2 in the BG which has literal 2 as an add effect (action stack B C), this action is chosen to solve precondition 2 of stack A B.

Following the same process, action unstack C A is introduced to satisfy literal 9 of action pickup A. As literal 5 of stack B C is not removed by any action (the propagation of delete effects from unstack C A does not cause any conflict because the action stack B C is not reached), the algorithm selects a_0 to satisfy this literal.

Finally, literal 6 of pickupA is solved by means of action stack B C; the remaining preconditions of unstack C A and pickup B are all solved with a_0. It is important to notice that the algorithm always selects at first place actions already contained in the BG rather than introducing a new action in the BG from the PG. The criteria the algorithm applies to select an action for a literal when there are several choices are explained in [11] in more detail.

3.2 The Optimal Graph

This stage performs two different tasks taking the BG as input:

- To discover whether the problem is solvable.
- To verify that the plan comprised in the BG leads to an optimal solution plan.

In order to accomplish these two tasks, the search method applies the following property.

Property 2 (partial consistency). *A BG is partially consistent if it is possible to set a total-order relation between each pair of actions in the same action level of the BG.*

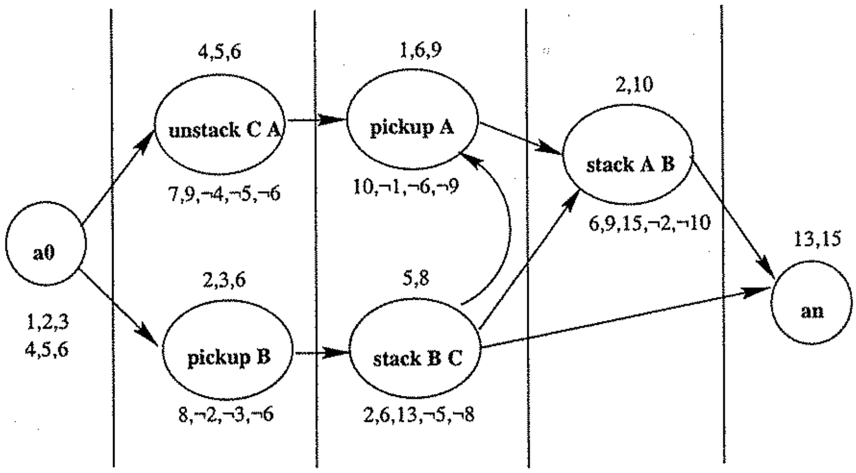

Fig. 3. BG for the sussman anomaly problem

Definition 1 (mutual exclusion). *Two actions a_i, a_j in the same level are mutually exclusive between each other if a_i deletes a precondition of a_j and a_j in turn deletes a precondition of a_i.*

We will show that the application of property 2 guarantees the completeness of the search method (it can find a graph if a solution exists for the given problem and returns "no graph" otherwise) and optimality of the obtained solution (optimality refers to the total number of actions in the final plan). The task of ensuring the total consistency of the graph is carried out by the POP at the fourth stage.

Unsolvable problems. Let a_1 and a_2 be two mutually exclusive actions, $\mathsf{Pre}(a_1) = \{x_1, y\}$, $\mathsf{Eff}(a_1) = \{z_1, \neg y\}$, $\mathsf{Pre}(a_2) = \{x_2, y\}$, $\mathsf{Eff}(a_2) = \{z_2, \neg y\}$.

If the only possible way to satisfy z_1 and z_2 is by means of actions a_1 and a_2 respectively, then we say this is a *precondition conflict*. The name comes from the fact that the literals involved in the conflict are the preconditions of actions (in the example, a_1 needs literal y and deletes y and likewise for a_2). In this case, the literal in conflict has to be achieved again by a new action (from the PG) or an existing action (from the BG). The only additional checking is to discover the correct ordering for the new producer action a_3 ($a_1 \rightarrow a_3 \rightarrow a_2$ or $a_2 \rightarrow a_3 \rightarrow a_1$).

Let's assume the precondition conflict cannot be solved with any of the actions in the PG or BG, that is neither Property 1 nor 2 hold after the resolution process. As there may be missing actions in the PG, the algorithm extends the PG one additional level to search for new actions to solve the conflict. This process is repeated until the PG cannot be further extended, that is, until no new action instantiations can be applied. Notice the PG only adds positive effects whose producer actions have not been previously asserted in the graph and therefore it is usually the case that only a few additional levels will be generated. Once the PG cannot be further expanded we can ensure all different action instantations are already included in the PG. Thus when a precondition conflict cannot be solved by any means the problem is unsolvable.

Theorem 1 (unsolvable problem). *If a problem is unsolvable then one of the following choices occurs:*

1. *any of the goal literals never appear in the PG or*
2. *the BG cannot be created or*
3. *there exists an irresoluble precondition conflict in the BG.*

Proof. A problem is unsolvable if there does not exist a sequence of actions which being applied to the initial situation gives rise to the goal state. Let O be the set of operators of a problem, $L = \{p_1, p_2, \dots, p_n\}$ the set of positive literals which are generated by all the instantiations of each $o \in O$ and G the set of goal literals.

(1) If such a sequence of actions does not exist then it might be the case that $\exists p_i \in G$ and $p_i \notin L$, in whose case p_i will never appear in the PG.

(2) If such a sequence of actions does not exist then it might be the case that it is not possible to find an independent sequence of actions for each $g \in G$, in whose case the BG will not be created because property 1 fails.

(3) If such a sequence of actions does not exist then it might be the case that the BG does contain a sequence of actions for each $g \in G$ but it is not feasible to combine all of them to form a global solution. In this case an irresoluble precondition conflict will be found in the BG as property 2 will eventually fail. Notice that establishing an ordering constraint between a pair of actions is equivalent to apply a resolution method such as promotion or demotion, and the procedure to solve mutually exclusive actions is equivalent to the process to restore a deleted literal in a POP. Thus, all resolution methods to solve a conflict are considered.

The above theorem does not state that these three cases are the only situations under which a problem turns out to be unsolvable. However, from our experience with this first prototype of the search method we can conclude that, if one of the above choices holds, then the problem is unsolvable. Let's explain this in an informal way.

If the **PG cannot be created** is because a given goal $g1$ never appears in the PG. Then the problem is unsolvable because:

- either there is no operator having $g1$ as an add effect in the domain definition, in whose case the problem has no solution or
- there exists an operator whose instantiation would generate $g1$ but it is never applicable during the PG creation. It is important to notice that the PG is a relaxed, unrestricted graph where the delete effects of actions are not taken into account. Consequently, if the conditions necessary for the operator to be applicable do not hold in the PG they will not hold in a plan either.

The situations under which the **BG cannot be created** do not all necessarily lead to an unsolvable problem. If the BG cannot be created it means that at least one inconsistent literal is found, i.e. property 1 fails because it is not possible to find a sequence of actions for achieving that literal. There are two different explanations for this situation:

- If p only appears at L_0 (initial literal level) ($p \notin L_h \,/ h \in [1, k]$) then there is no applicable operator having p as an add effect, in whose case p cannot be satisfied and property 1 fails. In this case it is easy to see the problem is unsolvable.
- On the other hand, if ($p \notin L_h \,/ h \in [1, k]$) it might occur there exists an applicable operator but this appears at a level A_l where $l > k$. Since our algorithm only considers action levels equal or lower than the one of the needer action, it would not find such an add effect to achieve the precondition p. In this case, the algorithm would return the problem is unsolvable although there might be a solution for it. This inconvenient can be easily tackled by considering all action levels when creating the BG, that is when solving the action preconditions.

The last case is when an **irresoluble precondition conflict** is found in the BG. Property 2 attempts to find a consistent ordering for each pair of actions a_i, a_j in the same level. If such an ordering exists (for example, $a_j < a_i$), then it means that a_i must be executed after a_j - one time step further than a_j at earliest. Then a_i is moved forward to the next action level and the same operation is repeated again to search for new mutually exclusive actions. This is an indirect way to verify that the sequences of actions that achieve each subgoal separately can be properly combined. Therefore, when an irresoluble precondition conflict is found it means there is a sequence of actions for achieving each subgoal but not a correct ordering for the entire set of actions. Since all the actions in the PG, BG and extended PG (if necessary) are taken into account when solving a conflict, we can conclude that an irresoluble precondition conflict entails an unsolvable problem.

In summary, if one of the three conditions holds then the problem is usually unsolvable. However, in some cases the algorithm could report that no solution exists for a solvable problem. All this indicates the algorithm is able to discover all unsolvable problems and might fail at finding the solution for some solvable problems.

Theorem 1 states that, when dealing with an unsolvable problem, one of the three conditions will hold. So, if the BG is created for a problem which is known to be unsolvable, then the lack of a solution will be discovered at the time of solving precondition conflicts. In order to illustrate this, let's take the following example from the *blocksworld* domain. The initial situation is the same as for the *Sussman* anomaly, that is, `on C A`, `on A table` and `on B table`. The goal state consists of the two contradictory literals: `on B C` and `on C B`.

The PG for this unsolvable problem is the same as for the *Sussman* anomaly problem until literal level L_2 (Table 1). The two literals from the goal situation

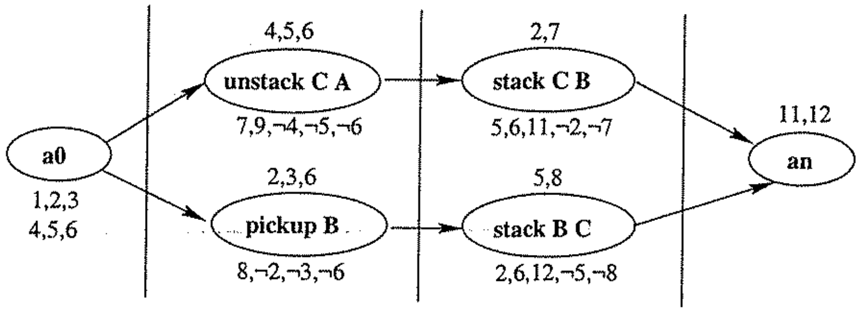

Fig. 4. BG for the unsolvable problem

are found in L_2 and the process for creating the PG terminates. The corresponding BG is shown in Figure 4.

As it can be seen in Figure 4, there exists a sequence of actions for each action precondition. However, we find a conflict between the two actions in A_1 as they both require and delete literal 6 (arm – empty).

When solving the precondition conflict we find six potential actions in the PG in action level A_2 to solve literal 6. Actions stack C B and stack B C cannot be used to restore literal 6 because they would both make property 1 fail (action stack C B would delete literal 2 which is required by pickup B, and action stack B C would delete literal 5 which is required by unstack C A). The other four actions make property 2 fail as all of them require and delete literals 7 or 8 which are also required by the actions in the BG (they are all mutually exclusive with the two actions stack C B and stack B C in the BG in action level $A2$).

The PG is then extended one level further to find new actions to solve the conflict (Table 2). Three of the actions in A_3 have literal 6 as an add effect. Action stack A B cannot be used because it requires and deletes literal 2 and therefore it cannot be ordered between unstack C A and pickup B. The same happens with action stack A C, which needs and deletes literal 5. The remaining choice is to use action putdown A which requires literal 10. As this literal is only generated by action pickup A, which in turn deletes literal 6, property 2 would fail because the same conflict is found again in the graph. The PG can be extended one more level (A_4 and L_4) and no actions for solving literal 6 are found at this level. We can conclude that the precondition conflict is irresoluble and consequently the problem has no solution.

Optimal solutions. During the process of creating the BG and verifying whether the problem is solvable, the selection of an action to solve a literal tends to obtain the minimal set of actions [11].

Property 3 (minimal graph). *Let A be the set of actions in a graph G. G is a minimal graph if for each precondition p of an action* $a_k \in A$ *there exists only one sequence of actions* $a_i \rightarrow a_{i+1} \ldots a_{k-1} \rightarrow a_k$ *such that* $p \in \mathsf{PAddEff}(a_{k-1})$

This property is also extended as follows: when there are several actions to solve a literal, the algorithm selects first an action whose preconditions are all already solved with the nodes in the graph instead of selecting an action which in turn needs additional actions to satify its preconditions. The application of property 3 is a first step to obtain an optimal plan. However, some additional checking must be done in order to guarantee the optimality.

Let a_1 and a_2 be two mutually exclusive actions, $\mathsf{Pre}(a_1) = \{x_1, y\}$, $\mathsf{Eff}(a_1) = \{z_1, \neg y\}$, $\mathsf{Pre}(a_2) = \{x_2, y\}$, $\mathsf{Eff}(a_2) = \{z_2, \neg y\}$. If there are other choices to generate $z1$ or/and $z2$ then we say this is an *effect conflict.* This problem usually arises due to a lack of information during the creation of the BG. The algorithm will then replace the producer action of $z1$ (or $z2$) by another action in the BG or PG which has this literal as an add effect. It is important to notice that the first operation carried out by the search method is to verify the effect conflict rather than the precondition conflict. This is because the resolution of a precondition conflict always inserts a new action instead of replacing one of the two mutually exclusive actions.

Proposition 1 (optimal graph). *Let G be a basic graph and A be the set of actions in G. If property 2 holds for the set A then G is an Optimal Graph (OG), that is A is a subset of the actions of an optimal solution.*

Proof. (1) The PG consists of the minimum number of levels as possible. (2) The process of creating the BG always tends to select the minimal set of actions (given two goals g_1 and g_2, if action a_1 achieves g_1 and action a_2 achieves both g_1 and g_2 then only a_2 is selected). (3) Actions existing in the BG are preferred to actions in the PG to solve conflicts caused by mutually exclusive actions. All this means that the algorithm is aimed at obtaining the optimal sequence of actions for each subgoal. (4) If there are no mutually exclusive actions, then it is feasible to succesfully combine the sequences of actions for each subgoal to produce an optimal plan.

Two remarks must be commented about this process:

- An OG leads to an optimal solution plan provided that the POP is known to be admissible.

- The third stage only verifies a partial consistency in the BG, which is very helpful to discover whether the problem is unsolvable and to check the optimality of the generated partial solution. However some other conflicts, as negative threats, can still be present in the OG (because they are not detected at the third stage). Thus the task of the POP will be to solve these conflicts and ensure the total consistency of the final solution plan.

The application of property 2 does not only allow to discover unsolvable problems but also to find and repair some conflicts in the BG. Figure 5 shows the OG for the *Sussman* anomaly problem after applying property 2. In the BG of Figure 3 we can see there are two mutually exclusive actions at A_1: both actions unstack C A and pickup B require and delete literal 6 (arm – empty). On solving this conflict, the algorithm searches for actions in the PG at the next action level A_2, to restore literal 6. There are several choices:

1. action putdown B is mutually exclusive with action stack B C, which is already in the BG
2. stack C B would make literal 2 of action stack A B become inconsistent
3. stack C A is mutually exclusive with action pickup A, which is already in the BG
4. stack B C cannot be used to restore literal 6 because it would delete literal 5 of unstack C A
5. stack B A is mutually exclusive with stack B C
6. putdown C does not cause any conflict

The algorithm selects putdown C to solve the conflict and the result is the OG shown in Figure 5.

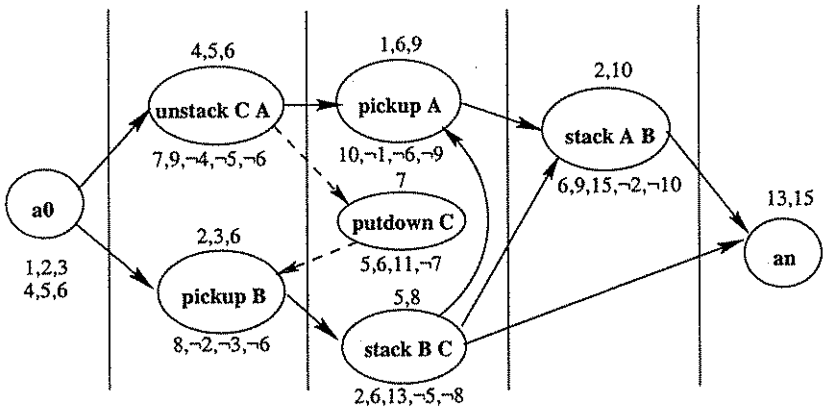

Fig. 5. OG for the *sussman* anomaly problem

4 The Partial-Order Planner

Our partial-order planner [9] is based on the UCPOP planner [8] and therefore completeness is guaranteed when starting from an empty initial plan. However, when the input to the POP is not an empty plan, additional operations are necessary to guarantee the completeness of the planner. In this section we show how to achieve this task. On the other hand, the planner uses an admissible heuristic search [4] which guarantees obtaining an optimal plan.

It must be pointed out that a POP cannot guarantee terminating on unsolvable problems although the complete search space is explored. Since this task has already been performed by the third stage we can ensure that the POP is only executed when it is confirmed a solution exists for the problem. The only remaining task is then to guarantee the POP is able to find such a solution.

4.1 From a Graph to an Input Graph

This section presents the process to generate an input plan from the BG or OG. This plan will be the initial plan for the Partial-Order Causal-Link (POCL) planner.

Let $\mathcal{G}(\mathcal{N},\mathcal{E})$ be a graph where $\mathcal{N}$ is the set of graph nodes so that $\mathcal{N}$ is a subset of the set of actions that constitute a solution plan for the given problem, and $\mathcal{E}$ is the set of edges that represent causal links between the actions in $\mathcal{N}$.

Definition 2 (Input plan). *An input plan is a tuple* $\Pi(\Lambda, \Sigma, \Theta, \Phi, \Gamma)$, *where:*

- $\Lambda = \mathcal{N}$ *is a set of actions that belong to the input plan.*
- $\Sigma = \mathcal{E}$ *is a set of causal links between the actions in* Λ.
- Θ *is a set of ordering constraints between the actions in* Λ, *resulted from the causal links in* Σ.
- Φ *is a set of non-satisfied preconditions of the actions in* Λ *(agenda) which in the case of an input plan is an empty set.*
- Γ *is a set of conflicts between the actions in* Λ.

There are two important features of our search method that must be enhanced:

- The input plan may not comprise all the actions of a solution plan because either the PG does not contain all the actions of the problem or the process of creating the BG has not included every action which might belong to the solution. The POCL planner is then responsible for finding these missing actions.
- In the process of creating the BG, when finding an action to solve the precondition of another action a_j, only action levels A_t where $t \in [0, j]$ are analyzed. This is because, according to the PG creation, it is more likely to find the correct action in a lower action level than in an upper level. However, since action levels do not stand for execution steps it might be the case that the correct action to solve a precondition of a_j were in an action level A_t where $t > j$. This issue is partially overcome at the OG stage since at this phase all action levels are taken into account.

These two features make the search method focus on a very restricted search space at the expense of introducing non-correct causal links for some literals (that is, the correct producer action to satisfy the literal is not the one denoted in the causal link).

When the POP input is an empty plan, a complete search space is generated and all choices to solve an open precondition or a conflict are considered in the resolution process. However, when the input is not an empty plan, completeness is not guaranteed because this non-empty plan is just the result of one branching line of the search space which would have been generated by a complete search method.

A way to recover completeness in the POP is by means of the White Knight (WK) concept [2]. This technique simply provides a combination of promotion, demotion, separation and precondition establishment methods.

Our WK technique [10] consists in deleting the threatened causal link and insert the precondicion associated to the causal link into the agenda (which it will be satisfied in turn by another existing or new action). This technique can also be used when a threat cannot be solved by promotion or demotion. Let a_i, a_j, a_k be three actions in the input plan such that $a_i \rightarrow a_j$, $a_i \rightarrow a_k$, $p \in \mathsf{AddEff}(a_i)$, $p \in \mathsf{Pre}(a_j)$, $p \in \mathsf{DelEff}(a_j)$, $p \in \mathsf{Pre}(a_k)$ and $p \in \mathsf{DelEff}(a_k)$. The following process is used to restore the precondition p of a_j or a_k (symmetrical threats):

1. select one of the symmetrical threats $\gamma(a_k, a_i, a_j)$ and solve it by promotion and/or demotion
2. select the other threat $\gamma(a_j, a_i, a_k)$ and use the WK technique
3. repeat this process inverting the order at which the threats have been selected

Proposition 2 (POP completeness). *If the plan obtained from the BG or OG is input to the POP as the initial plan, then the POP will find a solution.*

Proof. It is easy to check that all choices to restore the non-correct causal link are considered in this process. If the correct action for a literal is an action in the input plan, this will be discovered at step 2 of the above process when selecting an existing action. If the correct action does not appear in the PG, and consequently neither in the BG nor in the OG, this will be discovered at step 2 when selecting a new action. Thus, completeness is recovered during the planning process carried out by the POP by using a WK-based technique.

4.2 Main Features of the POP

It is important to distinguish between the two different types of plans that are input to the POP. The WK technique is mainly used when a precondition p of an action a_j is deleted by one action $a_i, i < j$ which belongs to another sequence of actions. However, in other graphs it is possible to solve interactions among actions by just finding a consistent ordering among them. This gives rise to two different types of graphs and, consequently, to two different types of plans. Let $\mathcal{A}$ be the set of actions in a graph and $\mathcal{S}$ the set of actions that constitute a solution plan for a given problem:

- *complete graphs*, when $\mathcal{A} = \mathcal{S}$. In this case, it is possible to set a total-order relation among all the actions in $\mathcal{A}$ without restoring any literal.
- *incomplete basic graphs*, when $\mathcal{A} \subset \mathcal{S}$. In this case, the WK technique must be used to solve the conflicts and new actions will have to be added in the input plan.

It is important to remark the similarity of the process undertaken at the third and fourth stage. In both cases the goal is to restore non-correct causal links in the graph. Unlike the third stage, the POP uses a WK technique when the appropriate actions to solve the conflicts are missing actions that do not appear in the PG; that is the reason why the third stage is not able to detect such conflicts.

The goal of the POP is to find a final plan, which involves two main tasks: detecting and solving conflicts among the actions in the OG and adding the necessary actions to complete the plan. These missing actions belong to any of the following three types:

- actions which do not appear in the PG or
- actions which appear in the PG but they are not discovered during the creation of the BG or
- actions which do appear in the PG and more than one execution of the same action is required (repeated actions).

5 Experimental Results

The experiments shown in table 3[3] correspond to a previous prototype of our method where the third stage is not implemented. Therefore, the algorithm is not obtaining the optimal solution for problems such as monkey test 1, nor detecting unsolvable problems.

Problems were taken from the UCPOP suite and Blackbox software distribution. All tests were run on a Sun Ultra 10 machine and results are given in seconds. We solved each problem ten times with Blackbox v3.6 [6] - using the Graphplan search engine-, STAN [3] and our search method. The results are classified into two groups, those for complete graphs and those for incomplete graphs (Table 3).

In most of the problems where the search method was able to obtain a complete graph, the CPU time was reduced by more than 50% compared to STAN and Blackbox. For example, in the *blocksworld* domain, as the number of blocks increased, this difference was greater. This is specially noteworthy in *TowerLarge* problems: our method solved the *TowerLargeD* problem that neither STAN nor Blackbox were able to solve.

For those problems with an incomplete graph, the search method behaves slightly worse that STAN and Blackbox, although this difference was not as significant as in the previous case. As the average and standard deviation results show (Table 4), our method behaviour was much more stable.

6 Related Work

Recently, local search techniques have been applied to planning problems. In particular, Kautz and Selman developed a local-search based method (Walksat)

---

[3] GT stands for the time used in the graph creation and TT for the total time. We have used a blocksworld domain with 3 operators.

Table 3. Performance of Blackbox, STAN and our method on different problems

Problem	**Blackbox**	**STAN**	**Our method**	
Complete graphs	TT	TT	GT	TT
Sussman	0.02	0.028	0.005	0.006
Tw_rever4	0.03	0.03	0.011	0.021
Tw_rever5	0.06	0.03	0.023	0.04
Tower4	0.07	0.032	0.013	0.013
Tower5	0.21	0.07	0.024	0.025
Tower6	0.6	0.16	0.046	0.047
Tower9	111	10.53	0.225	0.225
T_largeA	0.82	0.53	0.079	0.08
T_largeB	4.34	2.63	0.289	0.3
T_largeC	—	82.143	1.792	1.8
T_largeD	—	—	4.508	4.52
Incomplete Graphs	TT	TT	GT	TT
Hanoi3d	0.11	0.039	0.019	0.176
Hanoi4d	1.41	0.061	0.035	1.81
Ferry	0.04	0.012	0.005	0.073
Monkeyt1	0.11	0.022	0.009	0.08
Monkeyt2	0.26	0.037	0.014	0.212

Table 4. Average and Standard Deviation of our method, STAN and Blackbox of solved problems

	Our method	**STAN**	**Blackbox**
Average	0.454	6.025	7.034
Standard Deviation	1.017	20.469	26.812

for solving planning problems as satisfiability problems [7]. Gerevini and Serina propose a new method for local search [5] in the context of the "planning through planning graph analysis" approach introduced by Blum and Furst [1]. In this latter work, the general search scheme is based on an iterative improvement process, which, starting from an initial subgraph of the planning graph, greedily improves the "quality" of the current plan according to some evaluation functions.

Our approach keeps some similarity with this latter work in the sense that it also uses an initial subgraph of the planning graph which is iteratively refined in three consecutive stages. The mechanisms used for the plan improvement are a combination of formal properties on graphs and heuristics to evaluate the best choice to restore unsolved preconditions. In this way, our approach exploits the advantages of a local search while keeping the theoretical completeness of systematic search techniques.

Local search techniques are incomplete in the sense that they cannot detect that a search problem has no solution [5]. However, one remarkable aspect of our approach is that it is able to detect unsolvable problems, while efficiently solving hard search problems.

7 Conclusions

In this paper we have presented a search method consisting in a four-process execution. Each process obtains a partial solution which is incrementally refined by the subsequent processes. The first stage simply obtains a graph from the problem which may comprise all the actions of a final solution. The basic graph selects the actions which form a part of a correct solution. The aim of the third stage is to guarantee completeness and optimality of the generated solution. And the POP is in charge of obtaining the final solution by finding a correct ordering among all the actions in the plan.

Our objective was to develop a new planning approach by taking advantage of the partial-order planning properties and reducing the inefficiency caused by the large search spaces generated by these planners. We have also shown that our method average outperforms other planning approaches as Graphplan or SATPLAN planners. This is a first prototype of the search method. The obtained results confirm that the POP is still a bottleneck mainly for those problems which give rise to an incomplete graph. For this reason we suggest that the introduction of the third stage will significantly reduce the amount of work done by the fourth stage.

Acknowledgments. This work has been partially funded by the Spanish Government CICYT-FEDER project 1FD7-0887.

References

1. Blum A. L., Furst M.L.: Fast Planning Through Planning Graph Analysis. Artificial Intelligence 90:281–300 (1997).
2. Chapman D.: Planning for Conjuntive Goals. Artificial Intelligence 32(3):333–377 (1987).
3. Fox M., Long D.: STAN public source code. `http://www.dur.ac.uk/CompSci/research/stanstuff/` (1999)
4. Gerevini A., Schubert L.: Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. Journal of Artificial Intelligence Research 5:95–137 (1996)
5. Gerevini A., Serina I.: Fast Planning through Greedy Action Graphs. In Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99), 503–510 (1999)
6. Kautz H., Selman B.: Blackbox Planner 3.6. `http://www.research.att.com/ kautz/blackbox/` (1999)

7. Kautz H., Selman B.: The role of domain-specific knowledge in the planning as satisfiability framework. In Proceedings of the 4th International Conference on AI Planning Systems (AIPS-98), 181–189 (1998)
8. Penberthy J.S., Weld, D.S.: UCPOP: A Sound, Complete, Partial Order Planner for ADL. In Proceedings of the 1992 International Conference on Principles of Knowledge Representation and Reasoning, 103–114 (1992). Morgan Kaufmann, Los Altos, CA
9. Sebastia L., Onaindia E., Marzal E.: Improving expressivity and efficiency in Partial-Order Causal Link Planners. In Proceedings of the 18th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG-99), 124–136 (1999)
10. Sebastia L., Onaindia E., Marzal E.: A Graph-Based Approach for POCL Planning. In Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00), 531–535 (2000)
11. Onaindia E., Sebastia L., Marzal E.: 4SP: A four-stage planning process. In Proceedings of the ECAI-00 Workshop on New Results on Planning and Scheduling (PuK 2000), 115–129 (2000)