Towards Getting Domain Knowledge: Plans Analysis through Investigation of Actions Dependencies

Lukáš Chrpa and Roman Barták
Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague
{chrpa,bartak}@kti.mff.cuni.cz

Abstract

There are a lot of approaches for solving planning problems. Many of these approaches are based on 'brute force' search methods and do not care about structures of plans previously computed in certain planning domains. By analyzing these structures we can obtain useful knowledge that can help in finding solutions for more complex planning problems. Methods described in this paper are based on analysis of action dependencies appearing in plans. This analysis provides new knowledge about the planning domain that can be passed directly to planning algorithms to improve their efficiency.

Introduction

Many automated planning algorithms are based on 'brute force' search techniques accommodated with efficient heuristics guiding the planner towards the solution (Bonet & Geffner 1999). Hence an important question is how to find such information or knowledge that can be transformed into efficient planning heuristics. Several heuristics are based on the structure of Planning Graph (Blum & Furst 1997). These heuristics provided good results on many problems but on the other hand the analysis of Planning Graph itself does not seem to reveal complete information hidden in the plans structures. An approach in (Hoffmann, Porteous, & Sebastia 2004), which is closely related to plans analysis discussed in this paper, describes Landmarks - facts that must be true in every valid plan. Another work (Knoblock 1994) presents a structure called Causal Graph which describes dependencies between state variables. Both the Landmarks and the Causal Graphs are tools based on analyzing literals, giving us useful information about the planning domain but almost no information about the dependencies between the actions. One of the most influential works from the area of actions dependencies (McCain & Turner 1997) defines a language for expressing causal knowledge (previously studied in (Geffner 1990; Lin 1995)) and formalizes actions in it. One of the newest approaches (Vidal & Geffner 2006) based on plan space planning techniques over temporal domains gives very good results, especially in parallel planning, because it handles better supports, precedences and causal links. There are other practical approaches such as (Wu, Yang, & Jiang 2005) where knowledge is gathered from plans stochastically and (Nejati, Langley, & Konik 2006) where learning from expert traces is adapted for acquiring classes of hierarchical task networks. Finally, the paper (Chrpa 2007) presents a structure called Graph of Action Dependencies which provides information about plans structures based on dependencies of actions.

Another way for improving efficiency of planners rests in using macro-actions (Korf 1985) that represent sequences of primitive actions. The advantage of using macro-actions is clear - shorter plans are explored to find a solution - and there are some techniques for finding macro-actions (Newton, Levine, & Fox 2005; Coles, Fox, & Smith 2007). However, an efficient procedure for finding macro-actions remains a hard problem and many planners are still unable to handle well macro-actions together with primitive actions.

In this paper, we study new methods for plans analysis by investigating actions dependencies. We spotted that every action in the plan depends on some previous actions (or on the initial state) and we propose to formally capture this dependency relation. We provide a theoretical background based on this relation mostly aimed on possibilities of finding such actions that can be assembled into macro-actions without loss of plans validity. We also extend this theoretical background for subplans (usually obtained by assemblage of actions). Based on this theoretical background, we propose methods for acquiring knowledge from plans over given domains. The first method looks for such actions that can be assembled and this assemblage can be helpful in finding more complex plans. The second method decomposes plans into subplans such that the actions in these subplans can be applied in any order. Both methods are aimed to providing useful information to automated planners in the form of heuristics guiding the action sequencing.

The paper is organized as follows. The next section introduces basic notions from the planning theory. Then, we provide the theoretical background of the problem of actions dependencies in plans. After that, we describe the methods for gathering knowledge from plans. Finally, we discuss exploitation of the proposed methods and possible future works.

Preliminaries

Traditionally, AI planning deals with the problem of finding a sequence of actions transforming the world from some initial state to a desired state. State s is a set of predicates that are true in s. Action a is a 3-tuple $(p(a), e^-(a), e^+(a))$ of sets of predicates such that $p(a)$ is a set of predicates representing the precondition of action a, $e^-(a)$ is a set of negative effects of action a, $e^+(a)$ is a set of positive effects of action a, and $e^-(a) \cap e^+(a) = \emptyset$. Action a is applicable to state s if $p(a) \subseteq s$. If action a is applicable to state s, then the new state s' obtained after applying the action is $s' = (s \setminus e^-(a)) \cup e^+(a)$. A planning domain is represented by a set of states and a set of actions. A planning problem is represented by a planning domain, an initial state and a set of goal predicates. A plan is an ordered sequence of actions which leads from the initial state to any goal state containing all of the goal predicates. For deeper insight in this area, see (Ghallab, Nau, & Traverso 2004).

Action dependencies in plans: A theoretical background

Planning is basically about action sequencing. Clearly, actions in a valid sequence forming the plan are dependent in the sense that one action provides predicates serving as preconditions for other actions. In this section, we formally describe this dependency relation and present some of its useful features.

Basic description of the problem

Every action needs some predicates to be true before the action is applicable. These predicates are provided by the initial state or by other actions that were performed before. If we have a plan solving a planning problem, we can identify which actions are providing these predicates to other actions that need them as their precondition. The following definition describes this relation formally.

Definition 1.1: Let $< a_1, \ldots, a_n >$ be an ordered sequence of actions. Action a_j is *straightly dependent on the effect of action* a_i (denoted as $a_i \rightarrow a_j$) if and only if $i < j$, $(e^+(a_i) \cap p(a_j)) \neq \emptyset$ and there does not exist any $k_1, \ldots, k_l$ such that $i < k_1, \ldots, k_l < j$ and $(e^+(a_i) \cap p(a_j)) \subseteq \bigcup_{t=1}^{l} e^+(a_{k_t})$.
Action a_j is *dependent on the effect of action* a_i if and only if $a_i \rightarrow^* a_j$ where $\rightarrow^*$ is a transitive closure of the relation $\rightarrow$. To describe negation of this relation, we will simply use $a_i \nrightarrow^* a_j$.

The relation of straight dependency on the effect of action (hereinafter $\rightarrow$ only) means that $a_i \rightarrow a_j$ holds if some precondition of action a_j is provided by action a_i and a_i is the last action providing the precondition before action a_j. Notice that an action may be in the relation $\rightarrow$ with more actions. The following definition formally describes the set of predicates shared by two actions.

Definition 1.2: Let $E(a_i, a_j)$ be a set of predicates defined in the following way:

- $E(a_i, a_j) = (e^+(a_i) \cap p(a_j)) \setminus \bigcup_{t=i+1}^{j-1} e^+(a_t)$ iff $a_i \rightarrow a_j$
- $E(a_i, a_j) = \emptyset$ otherwise

Clearly $E(a_i, a_j)$ is not empty if and only if $a_i \rightarrow a_j$. We can extend Definition 1.1 to describe that an action is straightly dependent on the initial state or that the goal is straightly dependent on some action. However, to model these dependencies we can also use two special actions in the style of plan-space planning: $a_0 = (\emptyset, \emptyset, s_0)$ (s_0 represents the initial state) and $a_{n+1} = (g, \emptyset, \emptyset)$ (g represents the set of goal predicates). Action a_0 is performed before the plan and action a_{n+1} is performed after the plan.

Let us now define the complementary notion of action independence. The motivation behind this notion is that two independent actions can be swapped in the action sequence without influencing the plan.

Definition 1.3: Let $< a_1, \ldots, a_n >$ be an ordered sequence of actions. Actions a_i and a_j (without loss of generality we assume that $i < j$) are *independent on the effects* (denoted as $a_i \nleftrightarrow a_j$) if and only if $a_i \nrightarrow^* a_j$, $p(a_i) \cap e^-(a_j) = \emptyset$ and $e^+(a_j) \cap e^-(a_i) = \emptyset$.

The following lemma shows formally how the relation $\nleftrightarrow$ can be used to modify the sequence of actions.

Lemma 1.4: Let $\pi =< a_1, \ldots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \ldots, a_n >$ be a plan for planning problem P and $a_i \nleftrightarrow a_{i+1}$. Then plan $\pi' =< a_1, \ldots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \ldots, a_n >$ also solves planning problem P.

Proof: Assume that π solves planning problem P. Let s_j (respectively s'_j) be a state obtained by performing the first j actions from π (respectively π'). It is clear that $s_k = s'_k$ for $k \leq i-1$. From the assumption we know that π is valid which means that s_n is a goal state. Now, we must prove that s'_n is also a goal state. From definition 1.3 we know that $e^+(a_i) \cap p(a_{i+1}) = \emptyset$ which means that action a_{i+1} can be performed on state s'_{i-1} which results in state $s'_i = (s'_{i-1} \setminus e^-(a_{i+1})) \cup e^+(a_{i+1})$. From definition 1.3 we also know that $e^-(a_{i+1}) \cap p(a_i) = \emptyset$ which means that a_i can be performed on state s'_i which results in state $s'_{i+1} = (((s'_{i-1} \setminus e^-(a_{i+1})) \cup e^+(a_{i+1})) \setminus e^-(a_i)) \cup e^+(a_i)$. Finally, we know that $e^+(a_{i+1}) \cap e^-(a_i) = \emptyset$ which implies in $s'_{i+1} = s_{i+1}$ ($s_{i+1} = (((s_{i-1} \setminus e^-(a_i)) \cup e^+(a_i)) \setminus e^-(a_{i+1})) \cup e^+(a_{i+1})$). It is clear that after performing the remaining actions from π' (beginning from the $(i+2)$-nd action) on state s'_{i+1} we obtain state $s'_n = s_n$ which is also the goal state. □

The previous lemma showed that any two adjacent actions independent on the effects can be swapped without loss of validity of plans. This feature can be easily generalized for longer subsequences of actions where all actions in the sequence are pairwise independent on the effects.

Corollary 1.5: Let $\pi =< a_1, \ldots, a_i, \ldots, a_{i+k}, \ldots, a_n >$ be a plan solving planning problem P and $a_{i+l} \nleftrightarrow a_{i+m}$

for every $0 \leq l < m \leq k$. Let λ be a permutation over sequence $< 0, \ldots, k >$. Then plan $\pi' =< a_1, \ldots, a_{i+\lambda(0)}, \ldots, a_{i+\lambda(k)}, \ldots, a_n >$ also solves planning problem P.

Proof: The proof can be done in the following way. Action $a_{i+\lambda(0)}$ is shifted to position i by repeated application of lemma 1.4. The relation $a_{i+l} \nleftrightarrow a_{i+m}$ remains valid during these shifts so we can then shift $a_{i+\lambda(1)}$ to position $i+1$ etc. □

Remark 1.6: By assembling two primitive actions we obtain a new macro-action which means that the result of applying the macro-action to some state is identical to the result of applying the primitive actions to the same state. An action which is obtained by assembling of actions a_i and a_j (in this order) will be denoted as $a_{i,j}$, formally:

- $p(a_{i,j}) = (p(a_i) \cup p(a_j)) \setminus e^+(a_i)$
- $e^-(a_{i,j}) = (e^-(a_i) \cup e^-(a_j)) \setminus e^+(a_j)$
- $e^+(a_{i,j}) = (e^+(a_i) \cup e^+(a_j)) \setminus e^-(a_j)$

This approach can be easily extended for more actions.

The next proposition gives conditions for assembling of actions (or creating macro-actions).

Proposition 1.7: Let $\pi =< a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n >$ be a plan solving planning problem P and $i < j$ be indexes of actions in π. Assume that following conditions hold:

- for every k such that $i < k < j$: $a_i \nleftrightarrow a_k \vee a_k \nleftrightarrow a_j$
- for every k and x such that $i < k < x < j$: $\neg(a_i \nleftrightarrow a_k) \wedge a_i \nleftrightarrow a_x$ implies $a_k \nleftrightarrow a_x$
- for every l and x such that $i < x < l < j$: $\neg(a_l \nleftrightarrow a_j) \wedge a_x \nleftrightarrow a_j$ implies $a_x \nleftrightarrow a_l$

Then, there exists a plan $\pi' =< a_1, \ldots, a_{i,j}, \ldots, a_n >$ that also solves planning problem P.

Proof: It is clear that when a_i and a_j are adjacent then these actions can be assembled into $a_{i,j}$ without loss of validity of the plan (see Remark 1.6). If the actions are not adjacent then we can move the intermediate actions either before a_i or after a_j so eventually a_i and a_j become adjacent. The intermediate actions are shifted by repeating the following steps:

1. let a_x be the action directly following a_i in the current plan such that $a_i \nleftrightarrow a_x$, then we can swap a_i and a_x according to Lemma 1.4
2. let a_y be the action directly preceding a_j in the current plan such that $a_y \nleftrightarrow a_j$, then we can swap a_y and a_j according to Lemma 1.4
3. let a_k be the action between a_i and a_j with the largest index k in the current plan such that $\neg(a_i \nleftrightarrow a_k)$ then this action can be moved after a_j by repeated application of Lemma 1.4 (action a_k can be swapped with the action directly following it until a_k is moved after a_j)
4. let a_l be the action between a_i and a_j with the smallest index l in the current plan such that $\neg(a_l \nleftrightarrow a_j)$ then this action can be moved before a_i by repeated application of Lemma 1.4

Recall that any swap operation does not change the result of the plan which implies that π' is also a valid plan for P. □

Decomposition of the problem

One of the planning techniques, which seems to be very contributive, is a decomposition of planning problems into smaller subproblems. In this subsection we describe the possibility of plans decomposition into subplans and we also define relationships between them.

Definition 2.1: Let π be a plan solving planning problem $P = (\Sigma, s_0, g)$. *Subplan* π_i is a subsequence of π (not necessarily continuous) which solves some planning problem $P_i = (\Sigma_i, s_{0_i}, g_i)$, where $\Sigma_i \subseteq \Sigma$.

Definition 2.1 says what a subplan is. The following lemma describes a condition that the action subsequence must satisfy to form a subplan.

Lemma 2.2: Let $\pi =< a_1, \ldots, a_n >$ be a plan solving planning problem $P = (\Sigma, s_0, g)$, $a_0 = (\emptyset, \emptyset, s_0)$ and $a_{n+1} = (g, \emptyset, \emptyset)$. Let $\pi_i =< a_{i_1}, \ldots, a_{i_m} >$ be a subsequence of π. If there does not exist any action a_k such that $a_{i_p} \rightarrow^* a_k$ and $a_k \rightarrow^* a_{i_q}$ for any p, q then π_i is a subplan which solves a planning problem $P_i = \Big(\Sigma, \bigcup_{r \in \{0,\ldots,n\} \setminus \{i_1,\ldots,i_m\}, u \in \{1,\ldots,m\}} E(a_r, a_{i_u}), \bigcup_{s \in \{1,\ldots,n+1\} \setminus \{i_1,\ldots,i_m\}, v \in \{1,\ldots,m\}} E(a_{i_v}, a_s)\Big)$.

Proof: It is clear that both π and π_i are plans over the planning domain Σ. Predicates needed for execution of the subplan π_i can be provided only by the actions (including the special action a_0) that are in relation $\rightarrow$ (on the left hand side) with any action from π_i. Analogically we know that predicates obtained by execution of subplan π_i are needed for the other actions from π (including the special action a_{n+1}) that are in relation $\rightarrow$ (on the right hand side) with any action from π_i. From the assumption we know that there is no action both providing some predicates to P_i and requiring some predicates from P_i. □

Remark 2.3: Let $\pi =< a_1, \ldots, a_n >$ be a plan and $\pi_i =< a_{i_1}, \ldots, a_{i_m} >$ be its subplan (defined according to lemma 2.2). Proposition 1.7 shows how pairs of actions can be assembled without loss of validity of the plan. By generalization of this approach we are able to assemble the whole subplan π_i into one action $a_{i_1,\ldots,i_m}$ without loss of validity of plan π. It is clear that the assemblage can be done if for each $1 \leq k < m$ actions a_{i_k}, a_{i_k+1} satisfy the assumption of proposition 1.7.

Definition 2.4: Let $\pi =< a_1, \ldots, a_n >$ be a plan and $\pi_i =< a_{i_1}, \ldots, a_{i_m} >$ be its subplan. An assemblage of the

subplan π_i into a single action is called *a condensation of the subplan*.

The condensation of subplans results in a creation of new actions that are representing these subplans. It is clear that the planning domain must be extended by such actions.

Remark 2.5: The condensation of subplans results in a simplification of complex plans and in a creation of new actions. The procedure of identifying action dependencies and subplans can now be applied to the extended planning domain so one can obtain a hierarchical structure of actions useful for HTN style of planning.

Algorithms for plans analysis

In the previous section, we introduced formally several notions and their properties, namely the relation of action dependency and composition of actions. Now, we shall present the algorithms for finding action dependencies and identifying actions that can be assembled.

Auxiliary algorithms

First, we need to compute the relation $\rightarrow$ that is used in other methods to be presented later. The idea of the algorithm for computing the relation $\rightarrow$ is straightforward. Each predicate p is annotated by $d(p)$ which refers to the last action that created it. We simulate execution of the plan and each time an action a_i is executed, we find the dependent actions by exploring $d(p)$ for all preconditions p of a_i. Formally:

1. For each predicate p from the initial state s_0 set $d(p) := 0$.
2. For $i := 1$ to n do
   (a) For each $j \in \{d(p) | p \in p(a_i)\}$ define $a_j \rightarrow a_i$ and compute $E(a_j, a_i) := \{p | d(p) = j \wedge p \in p(a_i)\}$.
   (b) Set $d(p) := i$ for every $p \in e^+(a_i)$.
3. Do step (2a) also for the special action a_{n+1}.

The relation $\rightarrow$ can be naturally represented as a directed acyclic graph so the relation $\rightarrow^*$ is obtained as a transitive closure of the graph, for example using the algorithm from (Mehlhorn 1984).

Using the relation $\rightarrow^*$ we can easily identify actions independent on effects. These actions must not be in relation $\rightarrow^*$ and they must satisfy some additional conditions that can be verified in a constant time.

Finally, we need to identify pairs of actions that can be assembled. Proposition 1.7 gives the conditions that these actions must satisfy. The pairs of actions to be assembled are identified from example plans given in the input.

It can be seen that all the algorithms presented in this section run in polynomial time (with respect to the number of actions).

Looking for operators that can be assembled

Planning domains include planning operators rather than ground actions (Ghallab, Nau, & Traverso 2004) that we assumed in previous sections. Planning operators contain variables and actions are obtained by substituting appropriate constants for the variables. We shall now present a method for assembling planning operators using the techniques from the previous sections. The idea is as follows. For a pair of planning operators we explore pairs of actions that are ground instances of these operators. Based on the number of such pairs in example plans, we propose whether and how the planning operators can be assembled.

Let M be a square matrix where both rows and columns represent all planning operators in the given planning domain. Field $M(i, j)$ contains a set of pairs $< N, C >$ such that:

- N is a number of such actions a_i, a_j that are instances of i-th and j-th planning operator (in order), $a_i \rightarrow a_j$ and both actions a_i and a_j satisfy the conditions from the proposition 1.7 in some example plan.
- C is a set of constraints between the actions a_i and a_j with respect to i-th and j-th planning operator (namely which variables are unified in the operators).

In other words, matrix M contains candidates for assembling (or becoming macro-actions). The following algorithm gives the procedure for selecting and processing of these candidates.

1. Compute matrix M by checking all such actions (except the special actions a_0 and a_{n+1}) that are in the relation $\rightarrow$ and satisfy the conditions from the proposition 1.7. (several plans over the same domain should be explored).
2. Select a 'proper' candidate. If there is not such candidate then exit.
3. Assemble the candidate (a pair of planning operators) with respect to the given constraints. Update the given plans as well (and also the given domain).
4. Update matrix M. Then continue with step 2.

Meaning of the 'proper' candidate seems to be quite colloquial. The best way how to find such a candidate is to detect if a ratio between the number of candidate actions (stored in $M(i, j)$) and the number of instances of the corresponding operators exceeds a defined bound. It is clear that if the bound is too small, many actions will be assembled that may lead to very large domains with many operators. In the other hand, if the bound is too large, almost no actions will be assembled which means that domains may remain unchanged.

Subplans detection

Detection of proper subplans according to lemma 2.2 can be a very hard problem, because each plan can have many such candidate subplans. In this subsection, we provide a method for subplans detection that splits a plan into strips in such a way that each strip will contain actions that are all independent and each strip is dependent on the previous strip.

Let $level : A \rightarrow N_0$ be a function which assigns each action a value (i.e. $level(a_i) = k$) in the following way:

- $level(a_0) = 0$
- For each $j < i$, where a_j not in relation $\nleftrightarrow$ with a_i $level(a_j) < level(a_i)$ holds.

- For each action a, $level(a)$ is as small as possible.

The computation of the function $level$ is not so difficult, because it is only needed to check the relation $\nleftrightarrow$ with previous actions, beginning from the highest $level$. It is clear that all actions with the same level are in the relation $\nleftrightarrow$ that results immediately from the definitions.

Now, we can decompose plans into subplans in such a way that every subplan contains such actions that have the same level. These subplans also have one interesting feature - all actions can be performed independently of their order (recall corollary 1.5). Computation of such subplans can be usually done very quickly (almost in a constant time).

The above decomposition can be contributive when we have a strip which grows with growing of plans complexity and the growth can be traced by some measurable values (for instance a number of objects). When such a strip is detected, we know that we are able to compute the strip for more complex planning problems more easily. Then the whole strip can be assembled and added as a single action to the given domain. This procedure can be applied on more strips satisfying mentioned conditions.

Let us now summarize the method:

- Detect strips of independent actions by analyzing previously computed plans in a given planning domain.
- Before start of solving of a new planning problem in the given domain, compute the strips (representing subplans), assemble them and put them as a new action into the given domain
- Solve the new planning problem with the updated domain.

The hope is that if the method can be used, it can significantly reduce the search space for more complex plans.

Preliminary evaluation

Depots domain

Depots domain is a well known planning domain from the third International Planning Competition (IPC). This domain was devised in order to see what would happen if two previously well-researched domains logistics and blocks were joined together. They are combined to form a domain in which trucks can transport crates around and then the crates must be stacked onto pallets at their destinations. The stacking is achieved using hoists, so the stacking problem is like a blocks-world problem with hands. Trucks can behave like 'tables', since the pallets on which crates are stacked are limited. We used the typed STRIPS version of the domain for evaluation of our method for looking for actions that can be assembled. We solved the first two problems normally with SGPlan (Hsu *et al.* 2007) and then we built the matrix as described in the previous text (see figure 1). We can see that the proper candidates for the assemblage are operators LIFT, LOAD and UNLOAD, DROP, because the ratio between the number of candidates and the number of instances equals 100%. It means that we can assemble these operators into LIFT-LOAD and UNLOAD-DROP and remove the original ones. When this approach was tested on more complex problems over this domain (SGPlan was used), the improvement was transparent (for instance, the problem 15 from IPC was solved in 0.5sec instead of unsuccessful try (more than 600sec) to solve the problem in the original domain). DRIVE, UNLOAD also seem to be a proper candidate (ratio 60%) for assemblage but in this case the original operators cannot be removed because they can be also used separately. We are not aware about any planner that can prioritize among the operators, in particular, to try the macro-operator first and if it does not lead to a solution then trying the primitive operators.

	LIFT(H,C,S,P)	LOAD(H,C,T,P)	DRIVE(T,P1,P2)	UNLOAD(H,C,T,P)	DROP(H,C,S,P)
LIFT(H,C,S,P) / 5		**5** (H,C,P)			
LOAD(H,C,T,P) / 5	**1** (H,P)			**2** (H,T,P)	
DRIVE(T,P1,P2) / 5		**2** (T,P2)		**3** (T,P2)	
UNLOAD(H,C,T,P) / 5					**5** (H,C,P)
DROP(H,C,S,P) / 5					

Figure 1: Matrix of candidates for assembling from Depots domain. Values in the left column after '/' represent the number of instances of the operators in all analyzed plans. The fields are filled with pairs representing the number of candidates for assemblage and the set of variables that must be equal.

PSR domain

PSR domain is another well known planning domain from the fourth IPC. In this domain, planners must resupply a number of lines in a faulty electricity network. The flow of electricity through the network, at any point in time, is given by a transitive closure over the network connections, subject to the states of the switches and electricity supply devices. The domain is therefore a good example of the usefulness of derived predicates in real-world applications. This domain is quite interesting, because the complexity of finding an optimal plan is polynomial (Helmert 2006). We analyzed several less complex problems from this domain by the method of the plans decomposition to the strips mentioned in the previous text. The analysis showed a very interesting result - despite the rising plan complexity the number of strips remained the same. In addition, each strip contained such actions that were instances of only one operator. The structure is following:

1. WAIT-CB(x)
2. WAIT-CB(x)-CONDEFF(y)-YES
3. WAIT-CB(x)-ENDOF-CONDEFFS
4. OPEN-SD(x)
5. CLOSE-CB(x)
6. WAIT-CB(x)
7. WAIT-CB(x)-CONDEF(y)-NO

8. WAIT-CB(x)-ENDOF-CONDEFFS

By exploring the structure we can see that the problem can be easily decomposed and we are able to guide the planner very well. Further analysis reveals that the problems are based on closing and updating CB(x) slots letting some SD(x) slots closed. Now, we know what CB(x) slots must be closed and what SD(x) slots cannot be opened. We are ready for computation of strips 4 and 5. Strips 1-3 serve for strips 4 and 5, because they provided updated CB(x) slots that are needed. Analogically, strips 6-8 provide updated CB(x) slots as well which can serve for the goal. Unfortunately again these results are mostly theoretical, because there is not such a planner that can handle the results of above analysis.

Conclusions

In this paper, we presented methods that are able to acquire some knowledge from plans in a given domain. The first method looks for such actions that can be assembled and this assemblage should be helpful while solving more complex plans. This method gained very good results when plans contained such planning operators that could not be only assembled but also fully replaced by the assembled operators. There is certainly a space for improvement, because the method can detect such operators that could be assembled but the original operators must remain in the domain. However, existing planners do not support a possibility for prioritizing operators which causes worse performance of these planners on updated domains because of a larger number of operators. The second method decomposes plans into subplans such that the actions in these subplans can be performed in any order. This approach takes advantage when we are able to trace these subplans and appoint some their aspect with respect to the growing complexity of planning problems in a given domain. When we are able to acquire such a knowledge we can significantly improve the planning procedure. However, there is not yet such a planner that can fully support this approach.

In future we can investigate more deeply the detection of subplans that appear frequently in plans but cannot be revealed with existing methods. As mentioned before it is a hard problem. Nevertheless we believe that some approaches for pattern recognition can be used, because when we visualize the relation $\rightarrow$ we might be able to recognize the subplans as patterns that appear frequently in plans. Another possibility of future research is an investigation of connection between the plans structures represented by the relation $\rightarrow$ and the complexity of the corresponding planning problems. This idea is motivated by the research in the area of causal graphs (Gimenez & Jonsson 2007; Katz & Domshlak 2007).

Acknowledgements

The research is supported by the Czech Science Foundation under the contracts no. 201/08/0509 and 201/05/H014 and by the Grant Agency of Charles University (GAUK) under the contract no. 326/2006/A-INF/MFF.

References

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proceedings of ECP*, 360–372.

Chrpa, L. 2007. Using of a graph of action dependencies for plans optimization. In *Proceedings of IMCSIT/AAIA*, volume 2, 213–223.

Coles, A.; Fox, M.; and Smith, A. 2007. Online identification of useful macro-actions for planning. In *Proceedings of ICAPS*, 97–104.

Geffner, H. 1990. Causal theories of nonmonotonic reasoning. In *Proceedings of AAAI*, 524–530.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning, theory and practice*. Morgan Kaufmann Publishers.

Gimenez, O., and Jonsson, A. 2007. On the hardeness of planning problems with simple causal graphs. In *Proceedings of ICAPS*, 152–159.

Helmert, M. 2006. New complexity results for classical planning benchmarks. In *Proceedings of ICAPS*, 52–61.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

Hsu, C.-W.; Wah, B. W.; Huang, R.; and Chen, Y. 2007. *SGPlan*. http://manip.crhc.uiuc.edu/programs/SGPlan/index.html.

Katz, M., and Domshlak, C. 2007. Structural patterns of tracable sequentialy-optimal planning. In *Proceedings of ICAPS*, 200–207.

Knoblock, C. 1994. Automatically generated abstractions for planning. *Artificial Intelligence* 68(2):243–302.

Korf, R. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–77.

Lin, F. 1995. Embracing causality in specifiing the indirect effects of actions. In *Proceedings of IJCAI*, 1985–1991.

McCain, N., and Turner, H. 1997. Causal theories of action and change. In *Proceedings of AAAI*, 460–465.

Mehlhorn, K. 1984. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *Proceedings of ICML*, 665–672.

Newton, M. H.; Levine, J.; and Fox, M. 2005. Genetically evolved macro-actions in ai planning. In *Proceedings of PLANSIG*, 47–54.

Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.

Wu, K.; Yang, Q.; and Jiang, Y. 2005. Arms: Action-relation modelling system for learning action models. In *Proceedings of ICKEPS*.