



Using of a Graph of Action Dependencies for Plans Optimization

Lukáš Chrpa

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague
chrpa@kti.mff.cuni.cz

Abstract. There are a lot of planning techniques which prefer faster computation of plans instead of quality of these plans. However, in many cases we do not only need to have a plan which is computed quickly, but we need to have the plan of good quality as well. This paper presents a Graph of Action Dependencies as a tool which can be useful for plans optimization. This approach is very good for detecting pairs of inverse actions in plans or actions not needed to acquire a goal etc. Combination of this approach with existing planners should bring an improvement in acquiring more quality plans in a short time.

1 Introduction

Planning problems are an important branch of AI and they are very useful in many practical applications. However, many planning problems are hard to solve and many planners need to use heuristics for faster computation, but unfortunately at the cost of loss of the optimality of plans.

An early planner, called STRIPS [2], was very popular, because it generated plans very quickly. However, it was showed later that STRIPS did not always generate optimal plans [15]. Later, there were proposed many planning techniques for solving planning problems. In 1997, there was proposed an algorithm based on planning graph [1] which was much more efficient than other planners and brought a little revolution into planning. In 1998, there was introduced an International Planning Competition [10] which evoked an expansion of many planning techniques. Lot of these planning techniques were suboptimal which means that plans generated by these techniques were not optimal (for example FF [8] and LPG [5]). The main advantage of suboptimal planning systems rests in much faster generations of plans than in optimal planning systems. However, these plans are far from being optimal. This is the reason why it is needed to study the area of plans optimization, but there is not much work done yet. Results, published in [17], are very interesting, but the technique for the plans optimization described there seems to consume a lot of time.

A possible approach which can help with plans optimizations rests in plans structure analysis, especially in analysis or description of actions dependencies.

There are many of these approaches. One of them [4] refers about action languages as a formalism for talking about the effects of actions. This work describes a couple of action languages which are divided into two main classes (action description languages for describing the system and action query languages for making queries to the system). Another work [12] defines a language for expressing causal knowledge (previously studied at [3, 11]) and gives an approach of formalizing actions. The contribution of both of the works is mainly theoretical. More practical work [9] comes with landmarks, facts that must be true in every valid plan. This approach can evoke an improvement to the planners giving them more constraints in searching. One of the newest approaches [16], which is based on plan space planning techniques (more in [6]), gives us very good results in parallel planning. The importance of these results rests in a fact that an under-mentioned structure of a Graph of Action Dependencies is partly related to plan space planning.

This paper presents a Graph of Action Dependencies which provides an information about plans structures. Usage of graphs in many kinds of optimization tasks seems to be very useful, because many problems can be solved much faster [14]. This paper also describes algorithms of plans optimization which run in a polynomial time and which can easily detect and remove many kinds of unnecessary actions (more precisely described in section 4). These algorithms can be very helpful with making plans more optimal without spending a lot of time. However, according to the complexity results [7], it is not possible to make the plan optimal in a polynomial time. This paper also presents a possibility of plans optimization through optimization of their subplans with a small length which in spite of hardness of this problem should not consume a lot of time (if the length on the subplans is 2 that the optimization can be computed in a polynomial time).

This paper is organized as follows. Next section will present some basic notions from planning problems. Section 3 will introduce a Graph of Action Dependencies, an algorithm of its construction and some basic theoretical aspects of it. Section 4 will present algorithms for plans optimization, especially removing of actions which are not needed to acquire a goal and pairs of inverse actions, and finally the ideas how to make plans optimization in general. Section 5 will present my future research plans and the last section will conclude.

2 Preliminaries

This section presents definitions and notions which are needed for better reading and understanding of this paper.

2.1 Planning problems

In this subsection, we introduce some basic definitions which are needed to understand basic notions frequently used in a connection to planning problems. To get deeper insight about planning problems, see [6].

Definition 1. Assume that $L = \{p_1, \dots, p_n\}$ is a finite set of predicates. Planning domain Σ over L is a 3-tuple (S, A, γ) where:

- $S \subseteq 2^L$ is a set of states. $s \in S$ is a state. If $p \in s$ then p is true in s and if $p \notin s$ then p is not true in s .
- A is a set of actions. Action $a \in A$ is a 3-tuple $(p(a), e^-(a), e^+(a))$ where $p(a)$ is a precondition of the action a , $e^-(a)$ is a set of negative effects of the action a and $e^+(a)$ is a set of positive effects of the action a and $e^-(a) \cap e^+(a) = \emptyset$.
- $\gamma : S \times A \rightarrow S$ is a transition function. $\gamma(s, a) = (s - e^-(a)) \cup e^+(a)$ if $p(a) \subseteq s$.

Definition 2. Planning problem P is a 3-tuple (Σ, s_0, g) such that:

- $\Sigma = (S, A, \gamma)$ is a planning domain over L .
- $s_0 \in S$ is an initial state.
- $g \subseteq L$ is a set of goal predicates.

Definition 3. Plan π is an ordered sequence of actions $\langle a_1, \dots, a_k \rangle$ such that, the plan π solves the planning problem P if and only if exist appropriate sequence of states $\langle s_0, \dots, s_k \rangle$ such that $s_i = \gamma(s_{i-1}, a_i)$ and $g \subseteq s_k$. Plan π is optimal if and only if for each $\pi' \mid \pi \leq \pi' \mid$ is valid.

2.2 Graph theory

In this paper, there are used some notions from the theory of graphs. These notions are usually well known and due to space reason the notions will not be defined here. To get deeper insight about the graph theory and graph algorithms, see [13].

3 Graph of action dependencies

This section describes what is the Graph of Action Dependencies and how can be constructed from the existing plan. There are also presented some theoretical aspects of the Graph of Action Dependencies.

3.1 Basic definitions

Before we describe the algorithm which constructs the Graph of Action Dependencies we must define notions which are needed to know what is the Graph of Action Dependencies.

Definition 4. Let $\langle a_1, \dots, a_n \rangle$ is an ordered sequence of actions. The action a_j is straightly dependent on the action a_i (denoted as $a_i \prec a_j$) if and only if $i < j$, $(e^+(a_i) \cup (p(a_i) \setminus e^-(a_i))) \cap p(a_j) \neq \emptyset$ and does not exist any k_1, \dots, k_l such that $i < k_1, \dots, k_l < j$ and $(e^+(a_i) \cup (p(a_i) \setminus e^-(a_i))) \cap p(a_j) \subseteq \bigcup_{t=1}^l (e^+(a_{k_t}) \cup (p(a_{k_t}) \setminus e^-(a_{k_t})))$.

Definition 4 told us that some action a_j is straightly dependent on some action a_i when the action a_i is performed before the action a_j and the action a_i is the last action which operates with some predicate (or predicates) which are in the precondition of a_j (the predicate or predicates must be true after performing the action a_i). It is clear that some action can be straightly dependent on more actions.

Definition 5. Let $\pi = \langle a_1, \dots, a_n \rangle$ is a plan which solves the planning problem $P = (\Sigma, s_0, g)$. Let $a_0 = (\emptyset, \emptyset, s_0)$ and $a_{n+1} = (g, \emptyset, \emptyset)$ are actions. Let \prec is the relation of straight dependency over the ordered sequence of actions $\langle a_0, a_1, \dots, a_n, a_{n+1} \rangle$. Graph of action dependencies with respect to π and P is a directed graph $G = (V, E)$ such that $V = \{v_0, v_1, \dots, v_n, v_{n+1}\}$ and $(v_i, v_j) \in E$ if and only if $a_i \prec a_j$.

The Graph of Action Dependencies can give us more information about structures of plans which will be explained later.

Definition 6. Let $\langle a_1, \dots, a_n \rangle$ is an ordered sequence of actions and a relation \prec^* is a transitive closure of the relation of straight dependency \prec . The action a_j is dependent on the action a_i if and only if $a_i \prec^* a_j$.

In the following text we will use the special actions a_0 and a_{n+1} defined in the same way as in definition 5. For illustration, the action a_0 is performed before the plan and the action a_{n+1} is performed after the plan.

3.2 Construction of the Graph of Action Dependencies

There will be presented an algorithm which constructs from some plan the Graph of Action Dependencies. Each predicate p gains an attribute $d(p)$ which refers to the last action which operates with them. Each edge (v_i, v_j) , should contain an assignment of a set of such predicates p where $d(p) = i$ and $p \in p(a_j)$.

Algorithm 1:

INPUT: A plan $\pi = \langle a_1, \dots, a_n \rangle$ which solves the planning problem $P = (\Sigma, s_0, g)$
 OUTPUT: A Graph of action dependencies with respect to π and P

```

foreach predicate  $p \in s_0$  do  $d(p) = 0$  endforeach
 $V = \{v_0\}; E = \emptyset$ 
for  $i = 1$  to  $n$  do
   $V = V \cup \{v_i\}$ 
  compute a set  $X = \{d(p) \mid \forall p \in s_{i-1} \cap p(a_i)\}$ 
  foreach  $x \in X$  do
     $E = E \cup \{(v_x, v_i)\}$ 
    create an assignment for  $(v_x, v_i)$  which contains all predicates  $p$  where
       $d(p) = x$  and  $p \in p(a_i)$ 
  endforeach
   $s_i = (s_{i-1} \setminus e^-(a_i)) \cup e^+(a_i)$ 
  foreach predicate  $p \in s_i \cap (e^+(a_i) \cup (p(a_i) \setminus e^-(a_i)))$  do  $d(p) = i$  endforeach
endfor

```

```

 $V = V \cup \{v_{n+1}\}$ 
compute a set  $X = \{d(p) | \forall p \in s_n \cap g\}$ 
foreach  $x \in X$  do
     $E = E \cup \{(v_x, v_{n+1})\}$ 
    create an assignment for  $(v_x, v_{n+1})$  which contains all predicates  $p$  where
         $d(p) = x$  and  $p \in g$ 
endforeach

```

Theorem 1 (correctness of algorithm 1). *Let $\pi = \langle a_1, \dots, a_n \rangle$ is a plan which solves the planning problem $P = (\Sigma, s_0, g)$. Algorithm 1 always constructs a graph $G = (V, E)$ which is the Graph of Action Dependencies with respect to π and P .*

Proof. It is clear that $V = \{v_0, v_1, \dots, v_n, v_{n+1}\}$. Now we must prove that $(v_j, v_i) \in E$ if and only if $a_j \prec a_i$. From the algorithm 1 can be seen that $(v_j, v_i) \in E$ if and only if there exists some predicate $p \in s_{i-1} \cap p(a_i)$ whose attribute $d(p) = j$. It is clear that $0 \leq j < i$ because in i -th step of cycle there is no such predicate p' where $0 \leq d(p') < i$. If exist such predicate $q \in (s_{i-1} \cap p(a_i))$ whose attribute $d(q) = j$ then this attribute must be set to j in j -th step of cycle and $q \in (s_j \cap (e^+(a_j) \cup (p(a_j) \setminus e^-(a_j))))$. Now it is clear that $(e^+(a_j) \cup (p(a_j) \setminus e^-(a_j))) \cap p(a_i)$ is not empty and contains at least one predicate. It is also clear that does not exist any k such that $j < k < i$ and $q \in (e^+(a_k) \cup (p(a_k) \setminus e^-(a_k)))$. Now we can see that $a_j \prec a_i$. \square

It is clear that the algorithm 1 runs at $O(n)$ steps. The algorithm 1 can also run simultaneously with a planner based on forward planning techniques.

3.3 Theoretical aspects of the Graph of Action Dependencies

This subsection is dedicated to basic theoretical aspects of the Graph of Action Dependencies which can help us finding out possible advantages of this approach.

Proposition 1. *A Graph of action dependencies $G = (V, E)$ with respect to $\pi = \langle a_1, \dots, a_n \rangle$ and $P = (\Sigma, s_0, g)$ is acyclic and has a topological sort $ord : V \rightarrow \{0, \dots, n+1\}$ such that $ord(v_i) = i, \forall i \in \{0, \dots, n+1\}$.*

Proof. From the definition of topological sort we know that for all edges $(v_i, v_j) \in E$ we have $ord(v_i) < ord(v_j)$. From the assumption we know that $ord(v_i) < ord(v_j)$ if and only if $i < j$. From the definition 5 we know that $(v_i, v_j) \in E$ if and only if $a_i \prec a_j$ for some actions a_i and a_j . From the definition 4 we know that if $a_i \prec a_j$ than $i < j$. Now it is clear that ord is the topological sort of G and from this can be immediately seen that G is acyclic. \square

Proposition 2. *Let $G = (V, E)$ is a Graph of Action Dependencies with respect to $\pi = \langle a_1, \dots, a_n \rangle$ and $P = (\Sigma, s_0, g)$. There is a path from v_i to v_j in G if and only if $a_i \prec^* a_j$.*

Proof. It can be seen immediately from the definition 5 (i.e. $(v_k, v_l) \in E$ if and only in $a_k \prec a_l$) and from the fact that \prec^* is a transitive closure of \prec . \square

Proposition 3. *Let $G = (V, E)$ is a Graph of Action Dependencies with respect to $\pi = \langle a_1, \dots, a_n \rangle$ and $P = (\Sigma, s_0, g)$. If there does not exist any action $a_i \in \pi$ such that $p(a_i) = \emptyset$ and also $g \neq \emptyset$ than $a_0 \prec^* a_j$ where $0 < j \leq n + 1$.*

Proof. From the definition 4 we know that if $p(a_i) \neq \emptyset$ than there must exist an action a_j such that $a_j \prec a_i$ and $j < i$. The proof can be done inductively. It is clear that a_1 must be straightly dependent (and also dependent) on a_0 . Assume that a_1, a_2, \dots, a_k are dependent on a_0 . It is clear a_{k+1} must be straightly dependent on at least one action a_0, a_1, \dots, a_k . Then it is clear that a_{k+1} is also dependent on a_0 . \square

4 Optimizing plans using Graph of Action Dependencies

This section shows how can be the Graph of Action Dependencies used for the optimization of plans.

4.1 Removing of actions which are not needed to acquire goal

Any action a where $a \not\prec^* a_{n+1}$ is not needed to acquire the goal. According to the proposition 2 $a_i \not\prec^* a_{n+1}$ is satisfied if and only if there is no path from v_i to v_{n+1} in the Graph of Action Dependencies. To detect and remove these actions, we must find and remove all vertices (and corresponding actions) which have not any successors until there are no such vertices without successors. Other kind of unnecessary actions are such actions on that is a_{n+1} dependent, but does not exist any action which is straightly dependent on them by any predicate from the set of positive effects. The following algorithm detects and removes all of previously mentioned kinds of ‘useless’ actions and also modifies the corresponding Graph of Action Dependencies.

Algorithm 2:

INPUT: Plan $\pi = \langle a_1, \dots, a_n \rangle$ which solves the planning problem $P = (\Sigma, s_0, g)$ and a Graph of Action Dependencies $G = (V, E)$ with respect to π and P .

OUTPUT: Modified plan π' where does not exist any action on which the goal is not dependent. Modified Graph of Action Dependencies with respect to π' and P

```

while exists  $v_i \neq v_{n+1}$  such that no edge  $(v_i, v_j)$  for any  $v_j$  contains an assign-
ment containing a predicate from  $e^+(a_i)$  do
  remove  $a_i$  from  $\pi$ 
  foreach  $(v_j, v_i) \in E$  do
    foreach predicate  $p$  from the assignment of  $(v_j, v_i)$  set  $d(p) = j$ 
  endforeach
  foreach  $(v_i, v_k) \in E$  do
    foreach predicate  $p$  from the assignment of  $(v_i, v_k)$  do  $E = E \cup$ 
       $(v_{d(p)}, v_k)$  if  $(v_{d(p)}, v_k) \notin E$ 
  endforeach
  remove  $v_i$  and all its input and output edges from  $G$ 
endwhile

```

Theorem 2. [correctness of algorithm 2] Let $\pi = \langle a_1, \dots, a_n \rangle$ is a plan which solves the planning problem $P = (\Sigma, s_0, g)$ and $G = (V, E)$ is the a Graph of Action Dependencies with respect to π and P . The algorithm 2 always computes modified plan π' which also solves the planning problem P and where $|\pi'| \leq |\pi|$. The algorithm 2 also computes the modified Graph of Action Dependencies with respect to π' and P .

Proof. It is clear that the algorithm 2 always terminates, because there is a finite number of vertices in the Graph of Action Dependencies and in each step of while cycle is one of them deleted. It is also clear that $|\pi'| \leq |\pi|$ must be satisfied, because the algorithm 2 does not add actions to the plan. Let v_i is a vertex from V and a_i is a corresponding action from π that satisfy the condition in the while cycle. Let s_i is a state which is obtained by performing of the sequence of actions $\langle a_1, \dots, a_i \rangle$ on s_0 . It is clear that $e^+(a_i) \subseteq s_i$. If there exists any action a_j for $i + 1 \leq j \leq n + 1$ such that $e^+(a_i) \cap p(a_j) \neq \emptyset$ than the while condition must be broken (it can be seen from algorithm 1). If a_i is removed from π , the sequence of actions $\langle a_{i+1}, \dots, a_{n+1} \rangle$ can be performed on the state $s_{i-1} = (s_i \setminus e^+(a_i)) \cup e^-(a_i)$, because $e^+(a_i) \cap p(a_j) = \emptyset$ for every $j \in \{i + 1, \dots, n + 1\}$ (remember that a_{n+1} represents the goal state), so the plan $\langle a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n \rangle$ is valid and solves the planning problem P . Now it is needed to prove that the algorithm 2 also modifies the Graph of Action Dependencies. The first foreach cycle computes attributes of each predicate which is in $p(a_i)$ in such a way that the attributes have the same values as in algorithm 1 before i -th step of the for cycle. The second foreach cycle adds edges to the Graph of Action Dependencies in the same way as in the algorithm 1. \square

The algorithm 2 removes all actions which is not needed to acquire the goal. The complexity of the algorithm 2 depends on a fact how many steps we need to find a vertex satisfying the condition of the while cycle and how many times the while cycle will run. The while cycle can run at most $O(n)$ times, because after n runs there will remain only the special actions a_0 and a_{n+1} . To find a vertex satisfying the condition of the while cycle is needed $O(n)$ steps in a worst case (when using a naive search algorithm). This estimation should be improved by creating a list of successors of each vertex such that each edge between the vertex and its successors will contain at least one predicate from a set of positive effects from the action which corresponds with the vertex. It is clear that every vertex whose list is empty satisfies the condition of the while cycle. These lists can be created during the construction of the Graph of Action Dependencies and modified during any modification of the Graph of Action Dependencies.

4.2 Removing of inverse actions

Inverse actions are a couple of actions such that their performance consequently on a state leads to reach the same state as before the performance. The inverse actions are usually pairs of actions a, b such that $e^+(a) = e^-(b)$ and $e^-(a) = e^+(b)$. To detect and remove pairs of inverse actions we can also use the Graph of

Action Dependencies, because the pairs of inverse actions which can be omitted do not appear usually in plans consecutive. The idea of following algorithm is quite similar to the idea of the algorithm 2 and is based on detecting and removing of pairs of inverse actions which can be omitted.

Algorithm 3:

INPUT: Plan $\pi = \langle a_1, \dots, a_n \rangle$ which solves the planning problem $P = (\Sigma, s_0, g)$ and a Graph of Action Dependencies $G = (V, E)$ with respect to π and P .

OUTPUT: Modified plan π' where does not exist any unnecessary pair of inverse actions. Modified Graph of Action Dependencies with respect to π' and P

```

while exists inverse actions  $a_i$  and  $a_j$ , where  $i, j \neq 0, n+1$  such that  $(v_i, v_j) \in E$ 
  and no edge  $(v_i, v_k)$  for any  $v_k \neq v_j$  contains an assignment containing a
  predicate from  $e^+(a_i)$  do
    remove  $a_i$  and  $a_j$  from  $\pi$ 
    foreach  $(v_k, v_i) \in E$  and  $(v_l, v_j) \in E$ , where  $l \neq i$  do
      foreach predicate  $p$  from the assignment of  $(v_k, v_i)$  set  $d(p) = k$ 
      foreach predicate  $p$  from the assignment of  $(v_l, v_j)$  set  $d(p) = l$ 
    endforeach
    foreach  $(v_i, v_k) \in E$ , where  $k \neq j$ , and  $(v_j, v_l) \in E$  do
      foreach predicate  $p$  from the assignment of  $(v_i, v_k)$  do  $E = E \cup$ 
         $(v_{d(p)}, v_k)$  if  $(v_{d(p)}, v_k) \notin E$ 
      foreach predicate  $p$  from the assignment of  $(v_j, v_l)$  do  $E = E \cup$ 
         $(v_{d(p)}, v_l)$  if  $(v_{d(p)}, v_l) \notin E$ 
    endforeach
    remove  $v_i, v_j$  and all their input and output edges from  $G$ 
  endwhile

```

Theorem 3 (correctness of algorithm 3). *Let $\pi = \langle a_1, \dots, a_n \rangle$ is a plan which solves the planning problem $P = (\Sigma, s_0, g)$ and $G = (V, E)$ is a Graph of Action Dependencies with respect to π and P . The algorithm 3 always computes modified plan π' which also solves the planning problem P and where $|\pi'| \leq |\pi|$. The algorithm 3 also computes the modified Graph of Action Dependencies with respect to π' and P .*

Proof. The proof of termination of the algorithm 3 and the proof of $|\pi'| \leq |\pi|$ can be done in a same way as the proof of theorem 2. Let a_i and a_j from π are inverse actions that satisfy the condition of the while cycle. Let v_i and v_j are corresponding vertices to the actions a_i and a_j . Let s_{i-1} is a state which is obtained by performing of the sequence of actions $\langle a_1, \dots, a_{i-1} \rangle$ on s_0 . It is clear that performance of the actions a_i and a_j consecutively on the state s_{i-1} will result in the same state s_{i-1} . If there exists any action a_k for $i + 1 \leq k \leq j - 1$ such that $e^+(a_i) \cap p(a_k) \neq \emptyset$ than the while condition must be broken (it can be seen from algorithm 1). If a_i and a_j is removed from π , the plan $\langle a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_{j-1}, a_{j+1}, \dots, a_{n+1} \rangle$ is also valid, because all actions which are straightly dependent on a_j can be straightly dependent on actions performed before a_i (it is clear that all predicates from $e^+(a_j)$ must

exist before a_i is performed). If there exists some action which is straightly dependent on a_i , it is dependent only on predicates from $p(a_i) \setminus e^-(a_i)$, these predicates are also available before a_i is performed. Proof that the algorithm 3 modifies the Graph of Action Dependencies correctly can be done in the same way as in proof of theorem 2. \square

The algorithm 3 removes all pairs of inverse actions which can be omitted. The complexity of the algorithm 3 (likewise as on the algorithm 2) depends on a fact how many steps we need to find a pair of inverse actions satisfying the condition of the while cycle and how many times the while cycle will run. The while cycle runs at most $\frac{n}{2}$ times, because in each run of the while cycle 2 actions are removed. To find a pair of inverse actions satisfying the condition of the while cycle is needed $O(m)$ steps in a worst case, where m is number of edges in Graph of Action Dependencies and $m \leq n^2$. This estimation should be improved by creation of a list of edges which were not tested.

4.3 Looking for plans optimization in general

Previous subsections showed us two main possibilities of making plans more optimal in polynomial time. However, the complexity results [7] showed that there are some planning domains where is much harder to find an optimal plan than a satisfying (not optimal) plan. Next lines will show possibilities of usage Graph of Action Dependencies for plans optimization in general.

Definition 7. *Let $G = (V, E)$ is a Graph of Action Dependencies. $G' = (V', E')$ is Subgraph of Action Dependencies of G if and only if the next conditions are satisfied. V' is a subset of V . There is at most one vertex $v \in V'$ such that there exists an edge $(v', v) \in E$, where $v' \notin V'$. There is at most one vertex $w \in V'$ such that there exists an edge $(w, w') \in E$, where $w' \notin V'$. $(v_k, v_l) \in E'$ if and only if $v_k, v_l \in V'$ and $(v_k, v_l) \in E$.*

Previous definition gives a description how can be a given plan split into subplans which are satisfying planning subproblems of a given planning problem.

Proposition 4. *Let $G = (V, E)$ is a Graph of Action Dependencies with respect to π and P , where π is a plan which solves the planning problem P . If every Subgraph of Action Dependencies of G cannot be replaced by another Graph of Action Dependencies representing the solution of the same planning subproblem and having less of vertices than π is an optimal plan for the planning problem P .*

Proof. It is clear that if some part of a plan can be replaced by a shorter one than the plan is not optimal. The meaning of definition 7 is in a fact that every Subgraph of Action Dependencies represents a part of a plan (or whole plan) and if the Subgraph of Action Dependencies can be replaced by another Graph of Action Dependencies representing the solution of the same planning subproblem and which has less of vertices, the part of the plan (or the whole plan) represented by the Subgraph of Action Dependencies is not optimal. \square

Proposition 4 gives an idea how to make an algorithm based on searching for the Subgraphs of Action Dependencies and their corresponding subplans which always computes from a plan an optimal plan. However, in many planning problems the algorithm would have to replan the whole plan which means that in these case the usage of an optimal planned will be better. Luckily, the algorithm can be used for searching for subplans of length 2 and replacing them by plans contains one single action. In this case, the algorithm can run in polynomial time, the idea of searching for subplans of length 2 is very similar to searching for pairs of inverse actions (described previous subsection). When the algorithm is used for searching for subplans of length bigger than 2, the time needed to run of the algorithm can rise exponentially which means that the algorithm can be used for subplans of small length (not much greater than 2).

5 Future research

My plans of future research in this area are quite large, because the Graph of Action Dependencies seems to be a useful feature in many areas of planning. In the following lines, I will briefly introduce main possibilities of the other usage of Graph of Action Dependencies.

The Graph of Action Dependencies seems to be a very useful tool which can be good for a plan analysis. The plan analysis can be very useful in looking for heuristics and for gathering knowledge of planning domains which can help with computation of solutions of more larger planning problems.

At last, it seems to be useful to extend the Graph of Action Dependencies into a temporal planning. This extension can be very helpful in the connection between planning and scheduling and also can be helpful in (temporal) plan optimization and in achieving better plans quality by matching more constraint preferences.

6 Conclusion

This paper presented the Graph of Action Dependencies which showed us that it is an useful tool for plans optimization. There were presented algorithms for removing unnecessary actions like actions which are not needed to acquire a goal or pairs of inverse actions in a polynomial time which means that these actions can be removed from a plan quickly and make this plan more optimal. However, due to previous complexity results is not possible to get an optimal plan in a polynomial time, but this paper showed a possibility of making more optimal plans through optimizing their subplans which have a small length (if the length is 2 the optimization can be computed in a polynomial time). Despite the fact that plans cannot be fully optimized in polynomial time, presented algorithms seems to be very useful in combination with existing planners, because it should provide an improvement in plans quality in a short time.

7 Acknowledgements

I thank the reviewer for the comments. The research is supported by the Czech Science Foundation under the contract no. 201/07/0205 and by the Grant Agency of Charles University (GAUK) under the contract no. 326/2006/A-INF/MFF.

References

1. Blum A., Furst M. Fast planning through planning graph analysis. *Artificial intelligence*. 90:281–300. 1997.
2. Fikes R., Nilsson L. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4): 189-208. 1971.
3. Geffner H. Causal theories of nonmonotonic reasoning. *In proceedings of AAAI*. 524-530. 1990.
4. Gelfond M., Lifschitz V. Action Languages. *Electronic Transactions on Artificial Intelligence* 2:193-210. 1998.
5. Gerevini A., Serina I. LPG: a planner based on local search for planning graphs. *In proceedings of IPC*. 2002.
6. Ghallab M., Nau D., Traverso P. *Automated planning, theory and practice*. Morgan Kaufmann Publishers. 2004.
7. Helmert M. New Complexity Results for Classical Planning Benchmarks. *In proceedings of ICAPS*. 52-61. 2006.
8. Hoffmann J., Nebel B. The FF planning system: Fast plan generation through heuristics search. *Journal of Artificial Intelligence Research* 14:253-302. 2001.
9. Hoffmann J., Porteous J., Sebastia L. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research* 22:215-278. 2004.
10. <http://ipc.icaps-conference.org>
11. Lin F. Embracing causality in specifying the indirect effects of actions. *In proceedings of IJCAI*. 1985-1991 1995.
12. McCain N., Turner H. Causal Theories of Action and Change. *In proceedings of AAAI*. 460-465. 1997.
13. Mehlhorn K. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag. 1984.
14. Surynek P.; Chrapa L.; and Vyskocil J. Solving difficult problems by viewing them as structured dense graphs. *To appear at IJCAI*. 2007.
15. Sussman G. *A Computational Model of Skill Acquisition* New York: Elsevier. 1975.
16. Vidal V., Geffner H. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170(3): 298-335. 2006.
17. Westerberg C. H., Levine J. Optimising plans using genetic programming *In proceedings of ECP* 2001.