Generation of Macro-operators via Investigation of Actions Dependencies in Plans

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 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. The method described in this paper is designed for gathering macro-operators by analyzing of training plans. This analysis is based on investigation of action dependencies in the training plans. Knowledge gained by our method can be passed directly to planning algorithms to improve their efficiency.

Introduction

Despite significant improvement in planning systems in last years many automated planning algorithms are based on 'brute force' search techniques accommodated with 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) 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. The most recent studies (Gimenez & Jonsson 2007; Katz & Domshlak 2007) analyze Causal Graph with respect to complexity of planning problems. 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 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, papers (Chrpa & Bartak 2008b; 2008a) define relations describing action dependencies and present methods based on these relations.

Another way for improving efficiency of planners rests in using macro-actions (or macro-operators) (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). One of the most outstanding works in the area of macro-actions is Macro-FF (Botea *et al.* 2005), system for generating macro-operators through domain analysis. However, an efficient procedure for finding macro-actions (or macro-operators) remains a hard problem and many planners are still unable to handle well macro-actions together with primitive actions.

In this paper, we extend the work (Chrpa & Bartak 2008b) by providing a method generating macro-operators in terms of plans analysis by investigation of action dependencies. The method is used for learning macro-operators from training plans solving less complex planning problems to be helpful in solving more complex planning problems. In other words, the method provides knowledge which is encoded back to the domain. The method is designed to be used as a supporting tool for planners.

The paper is organized as follows. The next section introduces basic notions from the planning theory. Then, we provide brief theoretical background of the problem of actions dependencies in plans. After that, we describe the method for gathering macro-operators from training plans. Then, we present and discuss the evaluation of the method. Finally, we discuss exploitation of the proposed method and possible directions of future research.

Preliminaries

Traditionally, AI planning (in state space) 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).

In this paper we consider Typed STRIPS representation of planing problems. This representation allows definition of planning operators like 'templates' of actions. Clearly, actions are grounded instances of planning operators.

Action dependencies in plans

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.

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 effects 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 effects of action (hereinafter straight dependency only) means that $a_i \rightarrow a_j$ holds if some predicate from the precondition of action a_j is provided by action a_i and a_i is the last action providing the predicate before action a_j. Notice that an action may be straightly dependent on more actions. The relation of dependency on the effects of action (hereinafter dependency only) is a transitive closure of the relation of straight dependency. We can extend Definition 1.1 by describing 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.

Remark 1.2: Negation of the relations of straight dependency and dependency is marked in the following way:

- $a_i \nrightarrow a_j$ means that a_j is not straightly dependent on a_i.
- $a_i \nrightarrow^* a_j$ means that a_j is not dependent on a_i.

Let us now define the complementary notion of action independency. The motivation behind this notion is that two independent actions that are adjacent can be swapped in the action sequence without influencing the plan (it was formally proved in (Chrpa & Bartak 2008b)).

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 symbol for relation of independency on the effects (hereinafter independency only) looks symmetrical even though according Definition 1.3. it is not. The reason for using the symmetrical symbol is hidden in the previously mentioned property of the independency relation (recall that independent adjacent actions can be swapped without loss of plan validity).

Remark 1.4: Since the relations of dependency and independency are not complementary, we define the following symbol:

- $a_i \leftrightarrow a_j$ means that a_j is not independent on a_i.

Computation of the relation of straight dependency is quite straightforward. The idea is based on storing of the index of the last action which created the particular predicate. Concretely, 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. The relation of straight dependency can be naturally represented as a directed acyclic graph so the relation of dependency is obtained as a transitive closure of the graph, for example using the algorithm from (Mehlhorn 1984). The relation of independency can be easily computed by checking of every pair of actions a_i and a_j such that $i < j$ and the actions satisfy the conditions from Definition 1.3.

Identifying actions that can be assembled

By assembling of two primitive actions we obtain a new macro-action. The result of applying the macro-action to some state is identical to the result of applying the primitive actions to the same state. A macro-action which is obtained

Figure 1: Four different situations for moving the intermediate actions (grey-filled) before or after one of the boundary actions (black-filled).

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, see (Chrpa, Surynek, & Vyskocil 2007).

It is clear that we have to decide which actions can be assembled. We analyze several previously computed plans, where we focus on actions (instances of operators) that are (or can be) often consequent. We can analyze the plans by looking only for consequent actions. However in such a case we may miss many actions that can be performed consequently but in the plans there are some other actions placed between them. If such intermediate actions can be moved before or after them without loss of plan validity then we can assemble even non-consecutive actions. We use the main property of independent actions (can be swaped if adjacent) for detection if a pair of action can be assembled (we can make them adjacent). Figure 1 shows four different situations (actually two situations and their mirror alternatives) for moving the intermediate actions. Clearly, if the intermediate action is adjacent and independent on the boundary action we can move this action before or after the boundary action. If the intermediate action is not independent on one of the boundary actions we have to move it only before or after the other boundary action which means that this intermediate action must be independent on all actions in between (including the boundary action). The following algorithm is based on repeated application of these steps. If all intermediate actions are moved before or after the boundary actions then the boundary actions can be assembled. If some intermediate actions remain and no of the steps can be performed then the boundary actions cannot be assembled. Formally:

Function DETECT-IF-CAN-ASSEMBLE(`IN` index i, `IN` index j, `IN` independency relation S, `OUT` list of indices L, `OUT` list of indices R) : **returns** bool
  $D := \{k | i < k < j\}$
  $L := R := \emptyset$
  Repeat
  $chg := false$
    $k := min(D)$ // $min(D)$ returns the smallest element from D or 0 if D is empty
    If $k > 0$ and $(i, k) \in S$ **then**
      $D := D \setminus \{k\}$
      $chg := true$
      $L := L \cup \{k\}$
    EndIf
    $k := max(D)$ // $max(D)$ returns the greatest element from D or 0 if D is empty
    If $k > 0$ and $(k, j) \in S$ **then**
      $D := D \setminus \{k\}$
      $chg := true$
      $R := R \cup \{k\}$
    EndIf
    $Z := \{x | x \in D \wedge (i, x) \notin S\}$
    $k := max(Z)$
    If $k > 0$,$(k, j) \in S$ and **ForEach** $l \in D \wedge l > k$ $(k, l) \in S$ holds **then**
      $D := D \setminus \{k\}$
      $chg := true$
      $R := R \cup \{k\}$
    EndIf
    $Z := \{x | x \in D \wedge (x, j) \notin S\}$
    $k := min(Z)$
    If $k > 0$,$(i, k) \in S$ and **ForEach** $l \in D \wedge l < k$ $(l, k) \in S$ holds **then**
      $D := D \setminus \{k\}$
      $chg := true$
      $L := L \cup \{k\}$
    EndIf
  Until not chg
  If $D = \emptyset$ **then Return** true **else Return** false
EndFunction

Anyway, if the previous algorithm returns true (ie. actions can be assembled) we obtain also lists of action indices representing actions that must be moved before (respectively after) actions a_i and a_j. Usage of the lists will be explained in the following section.

Generating macro-operators

As mentioned before planning domains include planning operators rather than ground actions. Assembling operators rather than actions is more advantageous, because macro-operators can be more easily converted to more complex problems than macro-actions. The idea of detection such operators that can be assembled is based on investigation of training plans, where we explore pairs of actions (instances of operators) that can be assembled more times.

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 pair $< N, V >$ 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 can be assembled in some example plan.
- V is a set of variables shared by i-th and j-th planning operators.

In other words, matrix M contains candidates for assembling (or becoming macro-actions). The following algorithm constructs matrix M from a set of training plans (all plans must solve planning problems over the same domain):

Procedure CREATE-MATRIX(IN set of plans P, OUT matrix M)
Set M as empty square matrix
ForEach π **in** P **do**
Compute D as a relation of straight dependency on actions from π
Compute S as a relation of independency on actions from π
ForEach $(i, j) \in D$ **do**
If DETECT-IF-CAN-ASSEMBLE(i, j, S, L, R) **then**
Set k as the id of the operator whose a_i is an instance
Set l as the id of the operator whose a_j is an instance
Compute V as a set of variables that a_i and a_j share
If $M_{k,l}$ is empty **then**
$M_{k,l} :=< 1, V >$
Else
$< N, OV >:= M_{l,k}$
$M_{l,k} :=< N + 1, OV \cap V >$
EndIf
EndIf
EndForeach
EndForeach
EndProcedure

Computation of the sets of variables that operators share needs to be clarified. For example, in a variant of well known BlockWorld we can have operators PICK(box,hoist,surface) and DROP(box,hoist,surface). If we decide to make a macro-operator MOVE (consisting of PICK and DROP operators) we can also see that box and hoist are always the same (we are picking and dropping the same box with the same hoist in time), only surface may differ. Generally, we observe which parameters (objects) are shared by actions and select such parameters that are shared by all pairs of actions that can be assembled and that are instances of the particular operators.

Now, we explain the purpose of lists L and R that are generated in function DETECT-IF-CAN-ASSEMBLE. When we need to update plans by replacing selected actions by macro-actions (instances of generated macro-operators) we must also reorder other actions to keep the plans valid. The following approach shows how to reorder actions in plan $< a_1, \ldots, a_n >$ if a pair of selected actions a_i, a_j is assembled into macro-action $a_{i,j}$:

- actions $a_1, \ldots a_{i-1}$ remains in its positions
- actions listed in L are moved (in order) to positions $i, \ldots, i + |L| - 1$
- macro-action $a_{i,j}$ is moved to $i + |L|$-th position
- actions listed in R are moved (in order) to positions $i + |L| + 1, \ldots, j - 1$
- actions $a_{j+1}, \ldots, a_n$ are moved one position back (to positions $j, \ldots, n - 1$)

To generate macro-operators from training plans (over a given domain) we can use the following approach. We create repeatedly macro-operators until no other macro-operator can be created. At, first we have to compute a matrix of candidates for assembling from all training plans (CREATE-MATRIX). Then we select a proper candidate for creating macro-operators (SELECT-CANDIDATE) which means that such a candidate must follow pre-defined conditions (it will be explained later). After a creation of macro-operator from the selected candidate we must update all training plans (UPDATE-PLANS) which means that we replace particular pairs of actions by a particular instance of the new macro-operator. UPDATE-PLANS procedure can be easily implemented by application of the previously described approach (reordering actions after assembling) on every pair of actions (instances of the selected operators) in every plan.

Procedure GENERATE-MACRO(IN set of plans P, OUT set of macro-operators O)
$O := \emptyset$
Repeat
$picked := false$
CREATE-MATRIX(P,M)
If SELECT-CANDIDATE(M,C) **then**
$picked := true$
$O := O \cup \{C\}$
UPDATE-PLANS(P,C)
EndIf
Until not $picked$
EndProcedure

Last but not least the remaining unexplained issue is the function for selecting a proper candidate for assemblage (SELECT-CANDIDATE). We suggested to select such a candidate that satisfies the following conditions (let $f(O)$ represent frequency of operator O (how many instances of operator O occur in all training plans) and $N_{i,j}$ represents number N in field $M_{i,j}$):

- let $q = max(\{\frac{N_{i,j}}{f(O_i)} | M_{i,j}$ is not empty $\} \cup \{\frac{N_{i,j}}{f(O_j)} | M_{i,j}$ is not empty $\})$
- $q \geq b$

These conditions say that we are looking for such operators whose instances usually appear (or can appear) consecutively. Constant $b \in \langle 0; 1 \rangle$ represents a pre-defined bound which prevents selecting such operators whose instances do not appear consecutively so often. It is clear that if the bound is too small, many operators will be assembled. It may lead to very large domains with many operators which may cause many difficulties for planners. In the other hand, if the bound is too large, almost no actions will be assembled which means that domains may remain unchanged.

At last, we must decide which macro-operators can be added to the domain and which primitive operators can be

removed from the domain. Here, we decided to add every macro-operator whose frequency in updated plans is non-zero. On the other hand, we decided to remove every primitive operator whose frequency in updated plans become zero. It is clear that it may cause possible failure in generation of further plans, but usually in IPC (International Planning Competition) domains it does not happen. In case that some generation fails we can bring the primitive operators back to the domain and execute the generation again.

These conditions say that we are looking for such operators whose instances usually appear (or can appear) consecutively. Constant $b \in \langle 0; 1 \rangle$ represents a pre-defined bound which prevents selecting such operators whose instances do not appear consecutively so often. It is clear that if the bound is too small, many operators will be assembled. It usually causes that generated macro-operators are representing whole training plans that do not bring any contribution to planners. On the other hand, if the bound is too large, almost no actions may be assembled which means that domains may remain unchanged. However, in some cases we are not able to prevent generation of such macro-operators representing a huge part of some training plan even though b is quite large. The reason for this rests in the fact that sometimes only one (or very few) instance of some operator occurs in all training plans. Almost always we can find some other action that can be assembled with this instance, because the ratio between the number of candidates (stored in the matrix) and frequency of the operator becomes 1. It implies that the operator will be certainly selected for assemblage. To prevent this unwanted selection we should add the following condition:

- $\frac{N_{i,j}}{\sum_i f(O_i)} \geq c$

This conditions allows only to select such operators whose number of instances being able to be assembled (stored in $N_{i,j}$) divided by the number of all actions from the all training plans reaches predefined constant c. Another problem we are faced lies in the fact that many planners use grounding. It means that the planners generate all possible instances of operators that are used during planning. However, macro-operators usually have more parameters than primitive operators which means that the macro-operators may have much more instances than the primitive operators. To avoid troubles with planners regarding grounding we should limit a maximum number of parameters for each macro-operator (by pre-defined constant d).

We must also decide which macro-operators can be added to the domain and which primitive operators can be removed from the domain. Here, we decided to add every macro-operator whose frequency in updated plans is non-zero. On the other hand, we decided to remove every primitive operator whose frequency in updated plans become zero. It is clear that it may cause possible failure in generation of further plans. Luckily in IPC (International Planning Competition) benchmarks it almost does not happen. In case that some generation fails it is possible to bring the removed primitive operators back to the domain and execute the planner again.

Complexity discussion

The presented algorithms are designed to be performed in a polynomial time. It is almost clear that the presented algorithms run in polynomial times, but two issues need to be clarified. First, we take a look on the repeat cycle in function DETECT-IF-CAN-ASSEMBLE. This cycle can be performed at most as many times as the number of intermediate actions, because in every loop we remove at least one of the intermediate actions. Second, we take a look on the repeat cycle in procedure GENERATE-MACRO. We know that in every loop we generate some macro-operator and replace at least one pair of actions by the instance of the macro-operator. It means that the sum of plans lengths is decreased in every loop and therefore the repeat cycle can be performed at most as many times as the sum of the plans length.

Evaluation

Planning domains and planning problems used in the following evaluation are well known from IPC. We do the evaluation in the following steps:

- Generate several simpler plans as an input for our method.
- Generate macro-operators by our method and add them to the domain, remove such primitive operators that appear no longer in the updated plans.
- Compare the running time for more complex problems between the original domain and the updated domain.

We used SATPLAN 2006 (Kautz, Selman, & Hoffmann 2006) and SGPLAN 5.22 (Hsu *et al.* 2007) for generation of the training plans and SGPLAN 5.22 for comparison of the running time. The time comparison is made for such problems having their running time in the original domain greater than 2 seconds.

Tested domains

Blocks domain is well known from the second IPC. The domain consists of a table, a gripper and cubical blocks. Blocks are distributed in columns placed on the table. We can move only the topmost blocks to the table or to the other topmost blocks. The domain consists of 4 primitive operators (PICKUP, PUTDOWN, STACK, UNSTACK).

Depots domain is a well known planning domain from the third 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. The domain consists of 5 primitive operators (LIFT, LOAD, DRIVE, UNLOAD, DROP).

Zenotravel domain is a well known planning domain from the third IPC. The domain involves transporting people around in planes, using different modes of movement:

Domain	Planner	#p	b	c	d	#a	#r
Blocks	SGPLAN	3	0.3	0.05	4	5	4
Blocks	SGPLAN	3	0.8	0.1	4	2	2
Blocks	**SGPLAN**	**3**	**0.8**	**0.05**	**4**	**3**	**4**
Blocks	**SATPLAN**	**3**	**0.8**	**0.05**	**4**	**3**	**4**
Blocks	**SGPLAN**	**5**	**0.8**	**0.05**	**4**	**3**	**4**
Blocks	**SATPLAN**	**5**	**0.8**	**0.05**	**4**	**3**	**4**
Depots	SGPLAN	3	0.3	0.05	6	4	5
Depots	**SGPLAN**	**3**	**0.8**	**0.1**	**6**	**2**	**4**
Depots	SATPLAN	3	0.8	0.1	6	2	2
Depots	SGPLAN	5	0.3	0.05	6	4	2
Depots	SGPLAN	5	0.8	0.1	6	2	2
Depots	SATPLAN	5	0.8	0.1	6	2	2
Zenotr	SGPLAN	3	0.3	0.05	7	3	4
Zenotr	**SGPLAN**	**3**	**0.8**	**0.1**	**7**	**1**	**1**
Zenotr	SATPLAN	3	0.8	0.1	7	0	0
Zenotr	SGPLAN	6	0.3	0.05	7	4	2
Zenotr	**SGPLAN**	**6**	**0.8**	**0.1**	**7**	**1**	**1**
Zenotr	SATPLAN	6	0.8	0.1	7	2	0

Table 1: Results of the presented method for generating macro-operators using different settings and different planners for the generation of the training plans. #p represents the number of training plans. #a represents the number of added macro-operators. #r represents the number of removed primitive operators. Highlighted rows are representing such results that have the best performance in the further running times comparison test.

fast and slow. The domain consists of 5 primitive operators (BOARD, DEBARK, FLY, ZOOM, REFUEL).

Generating macro-operators and updating the domains

From the previous text we know that we can handle with macro-operators generation by pre-defined bounds b, c and d. Table 1 shows results of our method for generating macro-operators. We used different settings of bounds b,c and d, two different planners (SATPLAN 2006, SGPLAN 5.22) for generation of training plans and we also took into account a different number of training plans. The highlighted results in table 1 further used in the comparison (see the next subsection) are following. In Blocks domain we suggested:

- to add 3 new macro-operators (PICKUP-STACK, UNSTACK-STACK, UNSTACK-PUTDOWN)
- to remove all 4 primitive operators (PICKUP, PUTDOWN, STACK, UNSTACK)

In Depots domain we suggested:

- to add 2 new macro-operators (LIFT-LOAD, UNLOAD-DROP)
- to remove 4 primitive operators (LIFT, LOAD, UNLOAD, DROP)

In Zenotravel domain we suggested:

- to add 1 new macro-operator (REFUEL-FLY)
- to remove 1 primitive operator (REFUEL)

problem	original domain	updated domain
probBLOCKS-14-0	> 300s	0.38s
probBLOCKS-14-1	> 300s	0.36s
probBLOCKS-15-0	> 300s	1.27s
probBLOCKS-15-1	168.84s	0.44s
depotprob5656	> 300s	2.01s
depotprob4534	> 300s	0.51s
depotprob7615	8.81s	1.93s
depotprob1817	24.28s	15.65s
ZTRAVEL-5-20a	7.31s	5.00s
ZTRAVEL-5-20b	8.29s	6.69s
ZTRAVEL-5-25a	12.72s	8.31s
ZTRAVEL-5-25b	3.91s	3.21s

Table 2: Comparison of running times in tested domains.

It is no surprise that setting $b = 0.3$ caused generation of more macro-operators, but some of them were too complicated and almost useless for future usage. Bound d was set in conformity with the domains, because we did not want to generate much more complicated macro-operators (regarding the number of parameters). The most interesting bound was c, because in Block domain it can be lower than in the other domains. The reason of this rests in replacing of all primitive operators with macro-operators in Blocks domain. The choice of the planner and the number of training plans did not work upon the results in Block domain. The results in Depots domain was quite unexpected, because the best result was taken by SGPLAN with the less number of the training plans. The reason for this rests in the fact that SGPLAN and SATPLAN (even though it should be optimal) generates non-optimal plans which means that in more complex plans these planners generate flaws like inverse actions that are unnecessary etc. In Zenotravel domain the number of training plans did not affects the results, but SGPLAN produces better results than SATPLAN. The reason of this rests in the fact that Zenotravel domain contains two primitive operators that are very similar (ZOOM and FLY), ZOOM can be replaced by FLY, but it does not work backwards. SGPLAN prefers operator FLY more frequently than SATPLAN.

Comparison of running times

In this evaluation we used SGPLAN, an absolute winner of the last IPC. The results in table 2 are very interesting. We chose such problems from each domain that were solved (in the original domains) at least in 3 seconds, because our method for updating domains is designed for more complex problems. Despite this the less complex problems are solved in the updated domains almost as fast as in the original ones.

The best results was reached in Blocks domain. In the original Blocks domain SGPLAN failed three times to find a solution in 300 seconds and one problem was solved in more than 150 seconds but in the updated Blocks domain SGPLAN solved all problems in approximately 1 second.

In Depots domain we also reached very good results. In the original Depots domain SGPLAN failed two times to find a solution in 300 seconds but in the updated Depots do-

main SGPLAN solved these problems at most in 2 seconds. In the rest problems we can see also a good improvement, comparing the running time in the original and updated domains.

The results gained in Zenotravel domain also showed an improvement. Despite the improvement seems to be quite small in comparison with the other ones it showed that the running time was smaller in the updated domain in all problems by more than 20%.

Additional remarks

Presented results showed the improvement in solving more complex problems when domains are updated by using our method. Even though we used only at most 6 training plans for each domain, we gathered enough knowledge for updating the domains. Despite the removing of primitive operators from the domains every problem was solved in the updated domains. The reason may be that IPC planning problems usually differ by the number of objects and not by different types of initial states or goals.

Generated macro-operators used in the comparison were in all cases they were combined only from two primitive operators. Despite the construction of more complex macro-operators should reduce a depth of search, such macro-operators can have many more instances causing troubles to planners when using grounding, because in more complex problems grounding can consume much more time and memory.

As mentioned before we decide to evaluate only such problems whose running time in the original domain exceed 2s, because the method is designed for more complex problems. Anyway, the results of less complex problems (running time in the original domains less than 2s) were almost the same both in the original domains and the updated domains.

Conclusion

In this paper, we presented a method for generating macro-operators from training plans to be helpful on more complex plans. The method is looking for such actions that can be assembled which results in detection of such operators that can be assembled into a macro-operator. The actions that can be assembled are detected via investigation of plans structures based on the relations of action dependencies or independencies. The advantage of this approach rests in the fact that we are able to detect such actions that are not adjacent in plans, but actions lying between them can be moved before or after them. The method is designed to be used as a supporting tool for planners. The presented evaluation showed that using of our method in reasonable and can transparently improve the planning process on more complex plans. Nevertheless the results were obtained by evaluation of IPC benchmarks only. The main disadvantage of IPC benchmarks rests in a similarity of the problems (the problems differ only in the number of objects) which makes easier analyzing of plans structures for obtaining useful information (like macro-operators). In real world applications it may be more difficult to use the method properly (for example, we need a set of good training plans etc.). Another problem rests in the fact that existing planners do not support prioritizing of actions which should prefer macro-operators more easily. Furthermore, more complex macro-operators may contain many parameters which can cause a big difficulties to planners that use grounding (computing of all possible instances of all operators). The planners cannot handle well with a huge number of actions and their performance can be extremely low.

In future, we should focus on extension of our method to be able to generate Hierarchical Task Networks (HTN). Then we can use some HTN planner, for example SHOP2 (Nau *et al.* 2003). This idea partially follows the idea of (Nejati, Langley, & Konik 2006). In addition, we should investigate more deeply how stochastic data gathered during the execution of our method (like the number of operators in training plans etc.) can be efficiently used. We believe that the stochastic investigation can be very helpful in avoiding of obtaining many unnecessary instances of macro-operators during grounding. We should also study action dependencies more from the side of predicates, because it may reveal knowledge which can be used as heuristics for planners.

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.

Botea, A.; Enzenberger, M.; Muller, M.; and Schaeffer, J. 2005. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

Chrpa, L., and Bartak, R. 2008a. Looking for planning problems solvable in polynomial time via investigation of structures of action dependencies. In *Proceedings of SCAI*, 175–180.

Chrpa, L., and Bartak, R. 2008b. Towards getting domain knowledge: Plans analysis through investigation of actions dependencies. In *Proceedings of FLAIRS*, 531–536.

Chrpa, L.; Surynek, P.; and Vyskocil, J. 2007. Encoding of planning problems and their optimizations in linear logic. In *Proceedings of INAP/WLP*, 47–58.

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.

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.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. Satplan: Planning as satisfiability. In *Proceedings of IPC*.

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.

Nau, D.; Au, T.; Ilghami, O.; Kuter, U.; Mudrock, J.; Wu, D.; and Yaman, F. 2003. Shop2: An htn planning system. *Journal of Artificial Intelligence Research* 20:379–404.

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*.