The Knowledge Engineering Review, Vol. 25:3, 281–297.
doi:10.1017/S0269888910000159

Generation of macro-operators via investigation of action dependencies in plans

LUKÁŠ CHRPA

Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, Charles University in Prague, Malostranské náměsti 25, 118 00, Prague 1, Czech Republic
e-mail: chrpa@kti.mff.cuni.cz

Abstract

There are many approaches for solving planning problems. Many of these approaches are based on 'brute force' search methods and they usually do not care about structures of plans previously computed in particular planning domains. By analyzing these structures, we can obtain useful knowledge that can help us find solutions to more complex planning problems. The method described in this paper is designed for gathering macro-operators by analyzing training plans. This sort of analysis is based on the investigation of action dependencies in training plans. Knowledge gained by our method can be passed directly to planning algorithms to improve their efficiency.

1 Introduction

Planning is an important branch of Artificial Intelligence (AI) research. In planning, we define states of 'worlds' described by logical facts or functions and actions (or operators) that can modify these states. The purpose of planning is to generate a sequence of actions that transform the 'worlds' from some initial state to the given goal state.

Despite significant improvement in planning systems in the last few years, many automated planning algorithms are still based on 'brute force' search techniques accommodated with heuristics guiding the planner toward the solution Bonet and Geffner (1999). Hence, an important question is how such knowledge transformable into efficient planning heuristics can be found. Several heuristics are based on the structure of a Planning Graph Blum and Furst (1997). While these heuristics provide good results, an analysis of the Planning Graph does not seem to reveal complete information hidden in the plan structures. One of the most significant works from the past was a solver called REFLECT (Dawson & Siklóssy, 1977), which uses a preprocessing phase to ease its own solving. Specifically, the preprocessing phase consists of detecting incompatible predicates (i.e. predicates that cannot be simultaneously true) and building macro-operators (described in greater depth in Section 3). System PRODIGY (Minton & Carbonell, 1987) focuses on learning search control rules (i.e. logical rules describing relationships between predicates or operators). Search control rules were also applied in a well-known planner called TALPlanner (Kvanström & Magnusson, 2003). A newer approach presented in Hoffmann *et al.* (2004) describes Landmarks—facts that must be true in every valid plan. Another work (Knoblock, 1994) presents a structure called Causal Graph that describes dependencies between state variables. The most recent studies (Gimenez & Jonsson, 2007; Katz & Domshlak, 2007) analyze the 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 planning domains, but almost no information about the dependencies between actions in plans. One of the

most influential works from the area of action dependencies (McCain & Turner, 1997) defines a language for expressing causal knowledge (previously studied in Geffner (1990) and Lin (1995)) and formalizes the actions in it. One of the newest approaches (Vidal & Geffner, 2006) is based on plan space planning techniques over temporal domains. It gained very good results, especially in parallel planning, because it handles supports, precedences and causal links in a better way. There are other more practically oriented approaches, such as those described by Wu *et al.* (2005), where knowledge is gathered from plans stochastically, and Nejati *et al.* (2006) where learning from expert traces is adapted for acquiring classes of hierarchical task networks (HTN). Finally, papers (Chrpa & Bartak, 2008a, 2008b) define relations describing action dependencies and present methods based on these relations.

Another way to improve the efficiency of planners rests in using macro-actions or macro-operators that represent sequences of primitive actions or operators (related works are discussed in Section 3). In this paper, we provide a method generating macro-operators by investigation of action dependencies in training plans (the method is an extension of the work presented in (Chrpa, 2008)). Our method is used for learning macro-operators from simpler training plans; the learned macro-operators are encoded back into the domains and the primitive operators replaced by the macro-operators are removed from the domains. Such domains can be passed to planners without modifying their code. It means that our method is designed as a supporting tool for arbitrary planners.

The paper is organized as follows. In the next section, we introduce basic notions from the planning theory. Then we discuss related works in the area of macro-operators. After that, we provide a brief theoretical background of the problem of action dependencies in plans and then we describe our method for gathering macro-operators from training plans. Then we present and discuss the formal soundness and time complexity of our method. Finally, we discuss the experimental results of our method, similarities and differences with existing approaches and possible directions of our future research.

2 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. Action a is applicable to state s if $p(a) \subseteq s$. If action a is applicable to state s, then new state s' obtained after applying action a 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 that lead from the initial state to any goal state containing all of the goal predicates. For a deeper insight in this area, see Ghallab *et al.* (2004).

In this paper, we consider the classical representation of planing problems. This representation allows the definition of planning operators, in which actions are their grounded instances. Our approach supports the Typed STRIPS representation of PDDL (Planning Domain Definition Language).

3 Related works

Macro-operators (macro-actions) represent sequences of primitive operators (actions), but behave as common planning operators (actions). The advantage of using macro-operators is clear—shorter plans are explored to find a solution. However, macro-operators usually have much more instances than primitive operators, which leads to an increased branching factor for search.

One of the oldest approaches, STRIPS (Fikes & Nilsson, 1971), generates macro-actions from all subsequences of plans. It leads to plenty of useless macro-actions. REFLECT (Dawson & Siklóssy, 1977) builds macro-operators from pairs of primitive operators that are applied successively

and share at least one argument. In this case, macro-operators are learned directly from a domain analysis. However, it may lead to the generation of useless macro-operators. FM (McCluskey, 1987) follows the ideas used by STRIPS, but instead of STRIPS, FM compiles learned sequences of operators into one single operator representing the whole sequence of primitive ones. In addition, FM learns b-chunks that help it with instantiating of macro-actions. Even though FM gained a significant improvement against STRIPS, still it produced many useless and too complex macro-operators. MORRIS (Minton, 1985) learns macro-operators for STRIPS from parts of plans appearing frequently or being potentially useful (but having low priority). Macro Problem Solver (MPS), presented in Korf (1985), learns macro-actions only for particular goals. It needs different macro-actions when the problem instances scale or goals are different. MACLEARN (Iba, 1989) generates macro-actions that can 'traverse' from one peak of a particular heuristic function to another peak. A domain-dependently oriented work (Iba, 1985) discusses the usability of macro-operators in puzzle worlds (for instance, Peg Solitaire).

One of the state-of-the-art approaches, MARVIN (Coles & Smith, 2007; Coles *et al*., 2007) learns macro-operators online from action sequences that help FF-based planners to escape plateaus. It also learns macro-operators from plans of reduced versions of the given problems. One of the most outstanding works in the area of macro-actions is Macro-FF (Botea *et al*., 2005), a system for generating macro-operators through the analysis of static predicates. In addition, Macro-FF can learn macro-operators from training plans by analyzing successive actions. Macro-FF is produced in two versions. CA-ED version is designed for arbitrary planners where changing their source code is not necessary and SOL-EP version (planner dependent) where a planner (in this case, FF) is enchanted for handling macro-operators. WIZARD (Newton *et al*., 2007) learns macro-actions from training plans by genetic algorithms. There are defined several genetic operators working over action sequences appearing in training plans. WIZARD is designed for arbitrary planners. DHG (Armano *et al*., 2003, 2005) is able to learn macro-operators from static domain analysis by exploring a graph of dependencies between operators.

Our method is designed for domain-independent planning and for arbitrary planners like the other systems. Macro-operators can be assembled only from operators that are dependent, in terms that one operator provides a predicate (or predicates) to the other operator. It is similar to existing approaches. Nevertheless, there are some differences between our method and the existing approaches. We are able to detect pairs of actions that can be assembled into macro-actions, but the actions do not have to be necessarily successive in training plans. In addition, we are able to update the training plans in such a way that the updated training plans consider generated macro-operators. Therefore, it is not necessary to run the planners again. This can help us with another issue, the removal of unnecessary primitive operators that can be replaced by generated macro-operators. Despite the potential loss of completeness of some planning problems, planners benefit from the removal of primitive operators and the experiments we made on International Planning Competition (IPC) domains did not reveal any problem that became unsolvable. In addition, our method can reveal a suitable set of macro-operators in very little time. A more thorough comparison of our method with the existing ones is done in Section 8.3 (last paragraph).

4 Action dependencies in plans

Action choice is the key part of planning. Plans often contain sequences with dependencies between actions in the sense that one action provides predicates serving as preconditions for the 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 $\langle a_1, \ldots, a_n \rangle$ *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* $(e^+(a_i) \cap p(a_j)) \not\subseteq \bigcup_{t=i+1}^{j-1} e^+(a_t)$.
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 $\rightarrow$.

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 before action a_j providing that predicate. Notice that an action may be straightly dependent on more actions (if it has more predicates in the precondition). The relation of dependency on the effects of action (hereinafter dependency only) is a transitive closure of the relation of straight dependency.

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

- $a_i \not\rightarrow a_j$ means that a_j is not straightly dependent on a_i (i.e. $\neg(a_i \rightarrow a_j)$).
- $a_i \not\rightarrow^* a_j$ means that a_j is not dependent on a_i (i.e. $\neg(a_i \rightarrow {}^*a_j)$).

Let us define the complementary notion of action independency. The motivation behind this notion is that two independent actions being adjacent can be swapped in the action sequence without influencing the plan (lemma 1.4 (see below) which has been formally proved in Chrpa and Bartak (2008b).

DEFINITION 1.3. *Let* $\langle a_1, \ldots, a_n \rangle$ *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* $a_i \not\rightarrow {}^*a_j$, $p(a_i) \cap e^-(a_j) = \emptyset$ *and* $e^+(a_j) \cap e^-(a_i) = \emptyset$.

LEMMA 1.4. *Let* $\pi = \langle a_1, \ldots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \ldots, a_n \rangle$ *be a plan solving planning problem P and* $a_i \nleftrightarrow a_{i+1}$. *Then plan* $\pi' = \langle a_1, \ldots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \ldots, a_n \rangle$ *also solves planning problem P.*

The symbol for relation of independency on the effects (hereinafter independency only) evokes a symmetrical relation even though, according to Definition 1.3, the relation of independency does not have to be necessarily symmetrical. The reason for using the symmetrical symbol is hidden in the previously mentioned property of the independency relation (lemma 1.4).

Remark 1.5. 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 (i.e. $\neg(a_i \nleftrightarrow a_j)$).

Computation of the relation of straight dependency is quite straightforward. The idea is based on storing of indices of the last actions that created the particular predicates. 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 when action a_i is executed, we find the dependent actions by exploring $d(p)$ for all predicates p in the precondition 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 Mehlhorn (1984). The relation of independency can be easily computed by checking every pair of actions a_i and a_j $(i < j)$ on satisfaction of the conditions from Definition 1.3. It is straightforward that the time complexity in the worst case is $O(n^2)$ where n represents the length of the sequence of actions (plan).

5 Identifying actions that can be assembled

We obtain a new macro-action by assembling two primitive actions. The result of applying a macro-action to some state is identical to the result of applying the primitive actions in the given

Figure 1 Four different situations for moving the intermediate actions (gray-filled) before or behind one of the boundary actions (black-filled)

order to the same state. A macro-action 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; see Chrpa *et al.* (2007).

It is clear that we have to decide which actions can be assembled. We can analyze several previously found plans (training plans), where we focus on actions (instances of operators) that are (or can be) often successive. We can analyze the plans by looking for successive actions only. However, in such a case, we may miss many pairs of actions that can be performed successively, but in the plans, there are some other actions placed between them. If the intermediate actions can be moved before or behind the chosen pair of actions without losing plan validity, then we can assemble even non-successive actions. We use the main property of independent actions (can be swapped if adjacent) for detection if a pair of actions can be assembled (we can make them adjacent). To get more insight regarding permutations in plans, see Fox and Long (1999). 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 behind one of the boundary actions (according to lemma 1.4). If the intermediate action is not independent on one of the boundary actions, then we have to move it only before or behind the other boundary action, which means that this intermediate action must be independent on all actions in between (including the boundary action).

The algorithm (Figure 2) is based on repeated application of the above steps. If all intermediate actions are moved before or behind the boundary actions, then the boundary actions can be assembled (become adjacent). If some intermediate actions remain and none of the steps can be performed, then the boundary actions cannot be assembled. Anyway, if the algorithm returns true (i.e. actions can be assembled), we also obtain lists of action indices representing (intermediate) actions that must be moved before (respectively behind) actions a_i and a_j. Usage of these lists will be explained in the following section.

6 Generating macro-operators

As mentioned earlier, 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 into more complex problems than macro-actions. The idea of detecting such operators, which can be assembled, is based on the investigation of training plans, where we explore pairs of actions (instances of operators) that can be assembled more times.

DEFINITION. 2.1. *Let M be a square matrix where both rows and columns represent all planning operators in the given planning domain. If field $M(k,l)$ contains a pair $\langle N, V \rangle$ such that:*

- N is a number of such pairs of actions a_i, a_j that are instances of k-th and l-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.

```
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 := ∅
    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) ∈ S then
            D := D \ {k}
            chg := true
            L := L ∪ {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) ∈ S then
            D := D \ {k}
            chg := true
            R := R ∪ {k}
        EndIf
        Z := {x|x ∈ D ∧ (i, x) ∉ S}
        k := max(Z)
        If k > 0,(k, j) ∈ S and ForEach l ∈ D ∧ l > k (k, l) ∈ S holds then
            D := D \ {k}
            chg := true
            R := R ∪ {k}
        EndIf
        Z := {x|x ∈ D ∧ (x, j) ∉ S}
        k := min(Z)
        If k > 0,(i, k) ∈ S and ForEach l ∈ D ∧ l < k (l, k) ∈ S holds then
            D := D \ {k}
            chg := true
            L := L ∪ {k}
        EndIf
    Until not chg
    If D = ∅ then Return true else Return false
EndFunction
```

Figure 2 Algorithm for detecting pairs of actions that can be assembled

In addition, a_i (resp. a_j) cannot be in such a pair with the other instances of l-th (resp. k-th) operator.

- V is a set of variables shared by k-th and l-th planning operators.

Then M is a matrix of candidates.

In other words, the matrix of candidates contains proper pairs of actions (instances of planning operators) for assembling (or becoming macro-actions). The algorithm (Figure 3) constructs the matrix of candidates from the given set of training plans solving the planning problems in the same domain. Computation of the sets of variables that operators share needs to be clarified. For example, in a variant of a well-known BlockWorld domain, there are operators PICK (box, hoist and surface) and DROP (box, hoist and surface). If we decide to make a macro-operator PICK-DROP (consisting of PICK and DROP operators in this order), then we can also see that the box and hoist are always the same (we are picking and dropping the same box with the same hoist in time), and only the 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 (instances of the given operators) that can be assembled.

Now, we explain the purpose of lists L and R that are generated in function DETECT-IF-CAN-ASSEMBLE. If we have to update plans by replacing selected actions by macro-actions (instances of generated macro-operators), then we must also reorder other actions to keep the

```
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) ∈ 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 arguments 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}
                    If a_i resp. a_j are not already selected as a candidate with l-th operator resp. k-th operator then
                        N1 := N + 1 else N1 := N
                    M_{l,k} := ⟨N1, OV ∩ V⟩
                EndIf
            EndIf
        EndForeach
    EndForeach
EndProcedure
```

Figure 3 Algorithm for creating the matrix of candidates for assemblage

```
(:action pickup_stack
    :parameters (?x ?y)
    :precondition (and (clear ?x)(ontable ?x)(handempty)(clear ?y))
    :effect (and (clear ?x)(on ?x ?y)(handempty)
            (not (ontable ?x))(not (holding ?x))(not (clear ?y)) )
  )
```

Figure 4 Example of PICKUP-STACK macro-operator

(training) plans valid. The following approach shows how to reorder actions in plan $\pi = \langle a_1, \ldots, a_n \rangle$, if a pair of selected actions a_i, a_j is assembled into macro-action $a_{i,j}$:

- actions $a_1, \ldots a_{i-1}$ remain in their positions
- actions listed in L are moved (in order) to positions $i, \ldots, +|L|-1$
- macro-action $a_{i,j}$ is added 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 (in the given domain), we can use the following approach (formally in Figure 5). We create macro-operators repeatedly until no other macro-operator can be created. At first, we have to compute the matrix of candidates from all the training plans (CREATE-MATRIX). Then we select a proper candidate for creating macro-operators (SELECT-CANDIDATE), which means that such a candidate must satisfy certain conditions (which will be explained later). To ensure the soundness of the generated macro-operators, we have to assign inequality constraints for macro-operator arguments. It prevents a possible instantiation of invalid macro-actions if these arguments are set as equal. In Figure 4, we can see an example of the PICKUP-STACK macro-operator. If the arguments are set as equal, we can simply see that such an instance is applicable, but invalid (when unfolded). Inequality constraints can be easily detected by simulation of performance of the operators that are going to be assembled. After a creation of the macro-operator from the selected candidate, we must update all training plans (UPDATE-PLANS), which means that we replace particular pairs

of actions by the corresponding instances 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.

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

$$max\left(\frac{N_{i,j}}{f(O_i)}, \frac{N_{i,j}}{f(O_j)}\right) \geqslant b \tag{6.1}$$

$$\frac{N_{i,j}}{\sum_k f(O_k)} \geqslant c \tag{6.2}$$

$$a(O_i) + a(O_j) - |V_{i,j}| \leqslant d \tag{6.3}$$

Condition 6.1 says that we are looking for such operators whose instances usually appear (or can appear) successively. Constant $b \in \langle 0; 1 \rangle$ represents a pre-defined bound that prevents selecting such operators whose instances do not appear successively so often. It is clear that if the bound is too small, many operators may be assembled. It usually causes that generated macro-operators are representing almost the whole training plans, which does not bring any contribution to planners. On the other hand, if the bound is too big, almost no operators may be assembled, which means that the domains may remain unchanged. However, in some cases we are not able to prevent the generation of such macro-operators representing a huge part of some training plan, even though b is set quite big. The reason for this rests in the fact that sometimes only one (or a very few) instances of some operator occur in all the 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 of candidates) and the frequency of the operator becomes 1. It means that the operator will be certainly selected for assemblage. To prevent this unwanted selection, we can add condition 6.2 allowing only the selection of such operators whose ratio between the number of instances being able to be assembled (stored in $N_{i,j}$) and the number of all actions from all the training plans reaches a predefined constant c.

Another problem we are facing rests 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 macro-operators may have much more instances than primitive operators. To avoid troubles with planners regarding grounding, we should limit the maximum number of parameters for each macro-operator by a pre-defined constant d (condition 6.3). If there are more candidates satisfying all the conditions, then we prefer the candidate with the maximum value of the expression listed in condition 6.1.

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 the updated training plans is non-zero. Similarly, we decided to remove every primitive operator whose frequency in the updated training plans becomes zero. It is clear that it may cause a possible failure when solving non-training problems. Fortunately, in IPC benchmarks, it does not happen (we did not experience any such problem during the experiments). If for some problem planners fail to find a solution, then it is possible to bring the removed primitive operators back to the domain and run the planners again.

7 Soundness and complexity discussion

We assume that all plans used for analysis by our algorithms are valid. To ensure the validity of the plans, we can simply extend the algorithm for computing the relation of straight dependency by checking the satisfiability of action preconditions. It is quite straightforward that the algorithms for computing the relations of (straight) dependency and independency (sketches of the algorithms are discussed at the end of Section 3) are sound and can be computed in $O(n^2)$ steps (in the worst case), where n represents the length of the input plan. Soundness and time complexity of the other presented algorithms are justified in more detail.

PROPOSITION 3.1. *The algorithm DETECT-IF-CAN-ASSEMBLE (Figure 2) is sound and can be computed in the worst case in $O(l^2)$ steps where l is the number of intermediate actions (actions between a_i and a_j).*

Proof. The idea of the algorithm is based on moving intermediate actions before or behind defined actions. It is clear that a pair of adjacent actions can be assembled into a macro-action (we must follow their order) without loss of validity of the examined plan. The moving of intermediate actions can be done in the four cases (Figure 1), where two of them are a mirror of the other two. Without loss of generality, we prove the soundness and complexity only in two cases (on the left-hand side on Figure 1), because the soundness and complexity of the other cases can be proved analogically. First, if $a_i \nleftrightarrow a_{i+1}$, then by applying lemma 1.4, we can move a_{i+1} before a_i without loss of the plan's validity and it takes a constant time (i.e. $O(1)$). Second, assume that $a_i \leftrightarrow a_k$, $k < j$ and k is the greatest possible value. If $a_k \nleftrightarrow a_l \; \forall l : k < l < j$, then by repetitively applying lemma 1.4, we can move a_k behind a_j also without loss of the plan's validity. It can take at most $O(l)$ steps. The algorithm always terminates because in each run of the loop we remove at least one intermediate action. When no intermediate action remains, the loop ends. It means that the cycle is performed at most l times. Hence, in the worst case the algorithm requires $O(l^2)$ steps to perform. □

PROPOSITION 3.2. *The algorithm CREATE-MATRIX (Figure 3) is sound and can be computed in the worst case in $O(n^4)$ steps where n is the total length of all the training plans.*

Proof. For each training plan, the algorithm explores each pair of actions being in the relation of straight dependency by the algorithm DETECT-IF-CAN-ASSEMBLE (Figure 2), which is sound (proposition 3.1). It is clear that by using this approach, we can build the matrix of candidates consistent with the previously stated conditions. It is also clear that in the worst case we can have $O(n^2)$ relations of straight dependency and the algorithm DETECT-IF-CAN-ASSEMBLE in the worst case can be performed in $O(n^2)$ steps (proposition 3.1—considering that l can be close to n). Summarized, it gives us the time complexity $O(n^4)$ in the worst case. □

THEOREM 3.3. *The algorithm GENERATE-MACRO (Figure 5) is sound and can be computed in the worst case in $O(n^5)$ steps, where n is the total length of all the training plans.*

Proof. From the soundness of the algorithms DETECT-IF-CAN-ASSEMBLE (proposition 3.1) and CREATE-MATRIX (proposition 3.2), we know that each candidate for assemblage represents a pair of actions that can be assembled without loss of the plan's validity. If we generalize it and consider the inequality constraints, then we can simply see that each macro-operator produced by this algorithm is valid. The algorithm also always terminates because in each step of the loop the length of the training plans decreases at least by one, which means that the loop can be performed in the worst case $n - 1$ times. Together with the complexity of the algorithm CREATE-MATRIX (proposition 3.2), it gives us the time complexity $O(n^5)$ in the worst case. □

It is well known that if we add a generated macro-operator into the domain, then the domain remains valid. We can also remove the primitive operators fully replaced by the generated macro-operators. It brings us an improvement of the performance of the planners. However, it may cause an insolvability of some problems that were solvable in original domains. Hopefully, in all tested

```
Procedure GENERATE-MACRO(IN set of plans P, OUT set of macro-operators O)
    O := ∅
    Repeat
        picked := false
        CREATE-MATRIX(P,M)
        If SELECT-CANDIDATE(M,C) then
            picked := true
            ASSIGN-INEQUALITY-CONSTRAINTS(C)
            O := O ∪ {C}
            UPDATE-PLANS(P,C)
        EndIf
    Until not picked
EndProcedure
```

Figure 5 Algorithm for generation of macro-operators

cases it did not happen as we can see in the experiments (Section 8). Despite the high time complexity of our method (in the worst case), the experiments showed that our method is fast (tenths of a second for one run of the GENERATE-MACRO procedure).

8 Experimental evaluation

In this section, we present the experimental evaluation of our method. We compare the performance of the given planners between the original domains and the domains updated by our method. The planning domains and planning problems that we used here are well known from the IPC. We have done the evaluation in the following steps:

- Generate several simpler training plans as an input for our method.
- Generate macro-operators by our method and add them to the domains. Remove such primitive operators that no longer appear in the updated training plans.
- Compare running times for more complex problems between the original domains and the updated domains. The time limit was set to 600 seconds.

We used SATPLAN 2006 (Kautz *et al.*, 2006) and SGPLAN 5.22 (Hsu *et al.*, 2007) both for the generation of the training plans (for the learning phase) and for the comparison of the running times and quality of plans. We also used LAMA (Richter & Westphal, 2008), Filtering and Decomposition for Planning (FDP) (Grandcolas & Pain-Barre, 2007) and LPG-td (Gerevini & Serina, 2002) for the comparison of running times and quality of plans (not for the learning phase).[1] The choice of the planners was motivated by great results that the planners achieved on the (several last) IPCs. Since SATPLAN and FDP cannot handle negative preconditions (which are necessary for representation of inequality constraints), we used a tool called ADL2STRIPS[2] that can produce grounded STRIPS domain from ADL domain.

For the evaluation, we used IPC domains *Blocks*, *Depots*, *Zenotravel*, *Rovers*, *Gripper*, *Satellite* and *Goldminer*.[3]

8.1 Generating macro-operators and updating the domains

As mentioned earlier, the generation of macro-operators depends on pre-defined bounds b, c and d (conditions 6.1, 6.2 and 6.3). The number of training plans for each domain differs from 3 to 6 with respect to their lengths. The average time taken by both SGPLAN and SATPLAN to

[1] The results of LPG are only briefly reported.

[2] Available on IPC4 website.

[3] Can be obtained on `http://ipc.icaps-conference.org`

Table 1 Suggestion of our method—the best results for the particular domains

Domain	Added macro-operators	Removed primitive operators
Blocks	PICKUP-STACK, UNSTACK-STACK, UNSTACK-PUTDOWN	PICKUP, PUTDOWN, STACK, UNSTACK
Depots	LIFT-LOAD, UNLOAD-DROP	LIFT, LOAD, UNLOAD, DROP
Zenotravel	REFUEL-FLY	REFUEL
Rovers	CALIBRATE-TAKE-IMAGE	CALIBRATE, TAKE-IMAGE
Gripper	PICK-MOVE-DROP, MOVE-PICK-MOVE-DROP	MOVE, PICK, DROP
Satellite	SWITCH-ON-CALIBRATE	SWITCH-ON, CALIBRATE, SWITCH-OFF
Gold miner	MOVE-PICKUP-LASER, MOVE-DETONATE-BOMB-MOVE-PICK-GOLD	PICKUP-LASER, PICK-GOLD, DETONATE-BOMB

generate a training plan was (mostly) within tenths of a second.[4] Despite the high (worst-case) time complexity $O(n^5)$ (Theorem 3.3), the average time taken by one run of our method (GENERATE-MACRO procedure) was within tenths of a second.[5]

We used different settings of bounds b, c and d and two different planners (SATPLAN 2006, SGPLAN 5.22) for the generation of the training plans. First, bound d was set to $N+1$ (except for the Satellite domain and Gripper domain for SATPLAN's training plans), where N represents the greatest number of arguments of operators in the particular domain, because we did not want to generate too complicated macro-operators. If bound b was set too low, then many useless macro-operators were generated. We found out that a reasonable value of bound b can be almost in all cases 0.8; only in the Gripper domain (for SATPLAN's training plans), we lowered it to 0.6. Setting bound c was not as definite as setting the other bounds. Usually, the reasonable value was between 0.1 and 0.05, but in the Gripper domain it was set to 0.03. The reason for keeping bound c low (0.03–0.05) rested in the fact that in the Blocks and Gripped domains, all the primitive operators were replaced by generated macro-operators. The choice of a planner for the generation of training plans brought several differences—only in the Blocks domain, it resulted in the same result. In the Depots domain, we were not able to remove some primitive operators when SATPLAN's training plans were used as we did when SGPLAN's training plans were used. In the Zenotravel and Rovers domains, we were not able to learn any suitable set of macro-operators when SATPLAN's training plans were used. Likewise, in the Satellite domain, when SGPLAN's training plans were used. In the Gripper domain, the results of learning differed with respect to planners' strategies—SATPLAN prefers to carry balls in both robotic hands, SGPLAN prefers to carry balls just in one robotic hand. In the Gold Miner domain, the planners preferred different operators, which resulted in slightly different results of learning.

The results of learning (best for the particular domains) are showed in Table 1. We stated only such alternatives that provided the best results in the running times and quality of plans comparison for the particular domains.

8.2 Comparison of running times and quality of plans

In this evaluation, we used SGPLAN 5.22, an absolute winner of the IPC 5, SATPLAN 2006, co-winner of optimal track in the IPC 5, LAMA, winner of the IPC 6 suboptimal track and FDP, participant of the IPC 5 and LPG-td, awarded on the IPC 4. The benchmarks ran on XEON 2.4 GHz, 1GB RAM and Ubuntu Linux. The results are presented in Tables 2 and 3. We chose such problems (in most domains) that were neither so easy nor so hard for the particular planners, because the evaluation of these problems usually tells us the most about the particular domains.

[4] Performed on XEON 2.4GHz, 1GB RAM, Ubuntu Linux.

[5] Performed on Core2Duo 2.66GHz, 4GB RAM, Win XP SP2.

Table 2 Comparison of running times and plans lengths (we assume that macro-actions are unfolded into primitive actions) for SGPLAN (left-hand side) and SATPLAN (right-hand side)

	SGPLAN							SATPLAN					
	Time (in seconds)			Plan length				Time (in seconds)			Plan length		
Problem	orig	upd-SG	upd-SAT	orig	upd-SG	upd-SAT	Problem	orig	upd-SAT	upd-SG	orig	upd-SAT	upd-SG
Blocks14-0	>600.00	0.03	0.03	NA	48	48	Blocks14-0	23.58	3.14	3.14	38	56	56
Blocks14-1	>600.00	0.03	0.03	NA	44	44	Blocks14-1	38.06	3.84	3.84	36	88	88
Blocks15-0	>600.00	0.32	0.32	NA	88	88	Blocks15-0	46.90	7.24	7.24	40	60	60
Blocks15-1	179.84	0.05	0.05	114	54	54	Blocks15-1	45.68	7.63	7.63	52	142	142
depots1817	24.56	15.52	20.71	100	104	94	depots4321	5.24	4.40	2.07	43	41	38
depots4534	>600.00	0.53	54.71	NA	112	110	depots5656	222.42	>600.00	143.33	70	NA	59
depots5656	410.94	0.32	7.70	133	132	82	depots6178	6.82	43.14	26.11	51	50	42
depots7615	8.48	1.88	2.14	98	102	91	depots7654	10.04	25.96	16.45	41	56	39
zeno-5-20a	0.88	0.75	–	98	101	–	depots8715	35.96	46.95	err	50	38	err
zeno-5-20b	1.07	0.77	–	92	97	–	zeno-3-10	3.77	–	4.17	31	–	35
zeno-5-25a	1.74	1.05	–	124	122	–	zeno-5-10	34.07	–	48.19	42	–	38
zeno-5-25b	0.57	0.58	–	117	125	–	zeno-5-15a	92.13	–	30.93	50	–	51
rovers4621	2.31	0.03	–	48	44	–	zeno-5-15b	err	–	err	err	–	err
rovers5624	0.10	0.02	–	52	52	–	rovers4621	182.20	–	>600.00	47	–	NA
rovers7182	4.32	0.12	–	90	91	–	rovers5624	4.30	–	>600.00	62	–	NA
rovers8327	3.53	0.06	–	78	71	–	rovers8327	1.17	–	>600.00	45	–	NA
gripper16	0.05	0.05	1.11	135	135	101	gripper8	>600.00	8.14	0.03	NA	53	71
gripper17	0.06	0.06	1.31	143	143	107	gripper9	>600.00	12.86	0.06	NA	59	79
gripper18	0.06	0.07	1.56	151	151	113	gripper10	>600.00	19.78	0.04	NA	65	87
gripper19	0.06	0.07	1.83	159	159	119	gripper11	>600.00	err	0.07	NA	err	95
gripper20	0.07	0.08	2.13	167	167	125	gripper12	>600.00	err	0.06	NA	err	103
satellite26	3.73	–	29.99	138	–	138	satellite15	82.79	88.25	–	68	70	–
satellite27	4.73	–	13.20	138	–	139	satellite16	>600.00	115.07	–	NA	69	–
satellite28	12.87	–	260.26	193	–	193	satellite17	129.39	127.62	–	74	73	–
satellite29	18.69	–	70.36	195	–	195	satellite18	25.05	24.40	–	44	43	–
satellite30	31.57	–	117.52	231	–	231	satellite19	>600.00	574.46	–	NA	66	–
satellite31	56.65	–	201.36	272	–	272	gminer7×7-06	6.00	5.07	6.34	33	35	34
gminer7×7-06	err	0.01	0.01	NA	33	30	gminer7×7-07	6.08	4.91	5.82	38	38	37
gminer7×7-07	err	0.02	0.01	NA	34	65	gminer7×7-08	3.06	2.08	2.81	25	25	25
gminer7×7-08	err	0.01	0.01	NA	25	26	gminer7×7-09	4.24	3.47	4.26	33	30	29
gminer7×7-09	err	0.01	0.01	NA	29	32	gminer7×7-10	5.96	4.83	6.05	35	35	35
gminer7×7-10	err	0.01	0.02	NA	33	43							

Table 3 Comparison of running times and plans lengths (we assume that macro-actions are unfolded into primitive actions) for LAMA (left-hand side) and FDP (right-hand side)

	LAMA							FDP					
	Time (in seconds)			Plan length				Time (in seconds)			Plan length		
Problem	orig	upd-SG	upd-SAT	orig	upd-SG	upd-SAT	Problem	orig	upd-SG	upd-SAT	orig	upd-SG	upd-SAT
Blocks14-0	0.12	0.08	0.08	84	84	84	Blocks10-1	>600.00	11.82	11.82	NA	34	34
Blocks14-1	0.13	0.06	0.06	52	44	44	Blocks10-2	>600.00	6.57	6.57	NA	34	34
Blocks15-0	0.44	0.10	0.10	144	52	52	Blocks11-0	>600.00	178.31	178.31	NA	36	36
Blocks15-1	0.27	0.14	0.14	112	62	62	Blocks11-1	>600.00	146.24	146.24	NA	34	34
depots1817	>600.00	93.68	>600.00	NA	122	NA	Blocks11-2	>600.00	93.02	93.02	NA	38	38
depots4534	243.61	1.39	9.81	122	67	107	depotprob7512	1.66	0.10	0.37	15	15	15
depots5656	>600.00	0.53	7.70	NA	70	98	depotprob1935	>600.00	11.41	44.18	NA	27	27
depots7615	>600.00	5.71	61.61	NA	77	78	depotprob6512	>600.00	70.24	288.27	NA	30	30
zeno-5-20a	1.22	0.87	–	91	91	–	depotprob1234	>600.00	29.18	147.31	NA	23	21
zeno-5-20b	1.55	0.75	–	83	91	–	zeno-2-4	5.34	7.88	–	11	11	–
zeno-5-25a	2.85	0.98	–	95	105	–	zeno-2-5	42.29	73.01	–	11	11	–
zeno-5-25b	7.18	1.42	–	100	115	–	zeno-2-6	58.18	7.71	–	15	15	–
rovers4621	0.06	0.06	–	47	47	–	zeno-3-6	551.53	>600.00	–	11	NA	–
rovers5624	0.08	0.04	–	50	50	–	gripper8	>600.00	0.04	8.98	NA	71	53
rovers7182	0.23	0.18	–	90	90	–	gripper9	>600.00	0.06	16.91	NA	79	59
rovers8327	0.15	0.10	–	71	77	–	gripper10	>600.00	0.08	32.66	NA	87	65
gripper16	0.05	0.06	3.76	101	135	101	gripper11	>600.00	0.10	55.57	NA	95	71
gripper17	0.05	0.07	4.49	107	143	107	gripper12	>600.00	0.13	92.14	NA	103	77
gripper18	0.06	0.08	5.26	113	151	113	satellite3	1.39	–	0.28	11	–	11
gripper19	0.07	0.08	6.13	122	159	119	satellite4	62.52	–	17.24	17	–	17
gripper20	0.07	0.10	7.02	128	167	125	satellite5	>600.00	–	264.06	NA	–	15
satellite26	3.85	–	7.37	139	–	139	satellite6	>600.00	–	>600.00	NA	–	NA
satellite27	2.80	–	3.63	135	–	139							
satellite28	>600.00	–	11.76	NA	–	194							
satellite29	16.82	–	19.88	190	–	191							
satellite30	72.18	–	32.63	229	–	229							
satellite31	40.46	–	67.47	269	–	272							
gminer7×7-06	0.22	0.04	0.03	170	31	31							
gminer7×7-07	0.04	0.04	0.03	65	34	65							
gminer7×7-08	>600.00	0.03	0.03	NA	25	26							
gminer7×7-09	0.14	0.04	0.03	130	29	32							
gminer7×7-10	0.30	0.04	0.03	176	31	43							

The less complex problems were solved in the updated domains almost as fast as or a bit slower than in the original ones (except for the Rovers domain in SATPLAN's evaluation). The hardest problems (both original and updated) were not solved within the time limit of 600 seconds.

SGPLAN performed well in the original domains on almost all the tested problems except Blocks problems, some Depots problems and Gold Miner problems. Running times in the updated domains were always better except in the Gripper domain, where the running times were slightly worse, and the Satellite domain where the results were significantly worse. The quality of the plans[6] generated in the updated domains was not much worse, however; sometimes, the quality was slightly better and, surprisingly, in one Blocks problem it was more than twice better. The best results SGPLAN were reached in the Blocks domain where the speed-up was quite impressive. The possible reason may rest in the fact that SGPLAN's heuristics (FF-based) do not handle well problems like Blocks or Depots, because the plan quality was significantly better in the updated problems as well. SGPLAN's behavior in the Gold Miner domain was weird, because for all more complex (original) problems, SGPLAN terminated without throwing any error message after about 3 minutes of running.

SATPLAN, unfortunately, did not benefit often from our method. In the Blocks domain, SATPLAN was able to generate plans faster, but at the price of significantly worse quality of plans. However, SATPLAN produced very good results in the updated Gripper domain, where the problems normally unsolvable (in 600 seconds) were solved in a couple of seconds (for the domain updated on the basis of SATPLAN's training plans) or in hundreds of a second (for the domain updated on the basis of SGPLAN's training plans). The reason for that rests in the fact that SATPLAN uses a Planning Graph and each tested problem in the updated Gripper domain can be solved in only two layers. SATPLAN also gained quite good results in the Satellite and Gold Miner domains. Errors thrown by SATPLAN were caused by insufficient memory or a large domain file (produced by the ADL2STRIPS tool).

FDP is a planner based on CSP techniques that guarantees optimal plans. Even though FDP seems to be an ideal candidate for generating training plans for the learning phase, it fails to find a reasonable number of training plans in reasonable time. However, the experiments showed that the performance on the updated domains significantly increased in most of the tested problems. Using macro-operators reduced the depth of the search, which expectedly increased FDP's performance. The worse results gained in the Zenotravel domain was caused by the fact that no macro-action was used in the solutions of the updated problems (except zeno-2-6). An absence of results on the Rovers and Gold Miner domains is caused by the inability of FDP to process the domains descriptions (both for the original and updated ones) correctly.

LAMA is a planner that combines the Causal Graph heuristic and FF-based heuristics. In the Depots, Blocks and Gold Miner domains, the quality of plans was significantly better in updated domains. In addition, the time comparison for the Depots domain showed a significant increase in performance. The results correlate a bit with the results achieved by SGPLAN, because SGPLAN uses FF-based heuristics as well.

We also made experiments with the LPG-td (Gerevini & Serina, 2002) planner. LPG is based on local search techniques. Our experiments showed significantly worse performance in the Blocks and Depots domains; in the other domains, LPG performed almost the same. However, the results of LPG had huge discrepancies (both the running times and the quality of plans) with respect to the selected random seed.

8.3 Additional remarks

The presented results showed an interesting improvement for more complex problems in the domains updated by our method. Even though we used only at most six training plans for each domain (depending on the length of the training plans), we usually gathered enough knowledge for

[6] A ratio of the length of the plans in the original domains and the length of the plans in the updated domains—macro-actions are unfolded into primitive actions.

updating the domains. Even though we removed primitive operators from the original domains, we were able to solve correctly each problem in the updated domains. The reason may be that planning problems from the IPCs usually differ by the number of objects and not by different types of initial states or goals. However, there exist domains (for instance, *Freecell*, *Pipesworld*, *N-puzzle* and *Sokoban*) where our method did not manage to find any reasonable set of macro-operators (in terms, that found macro-operators did not fully replace any primitive operator).

Generated macro-operators used in the comparison were in almost all cases combined only from two primitive operators, except in the Gripper and Gold-Miner domains. Although the construction of more complex macro-operators may reduce the depth of the search, such macro-operators may have much more instances that can cause troubles to planners (increased branching factor).

The success of our method depends on several factors. First, training plans should be optimal (shortest) or nearly optimal, because non-optimal plans may contain flaws (useless actions) that may prohibit the detection of useful macro-operators or useless primitive operators. Second, we have to decide what result of our method (generated macro-operators and removed primitive operators) is the best. We followed the strategy where the particular generated macro-operator replaces at least one primitive operator that is removed from the domain. The experiments showed that our strategy is reasonable and contributive in many cases. Of course, there is a possible improvement that considers planners' specifics and strategies. SGPLAN is a planner that decomposes a problem into subproblems and solves them by other planning techniques, mostly FF-based. LAMA also uses FF-based heuristics and, in addition, Causal Graph heuristics. FF-based planning techniques usually experience difficulties with plateaux. Therefore, if there are such macro-operators that help the FF-based planner to escape plateaux, then the performance of the planner should significantly increase. It has been already studied in Coles and Smith (2007). SATPLAN is a planner that translates Planning Graph into SAT and then uses a SAT solver to solve the problem. Potential success, in this case, mainly rests in the reduction of makespan (i.e. the numbers of layers of the planning graph that must be explored). However, if makespan is reduced only slightly, it may not result in speed-up, because the layers can be much more complex. It also depends on the first appearance of instances of particular macro-operators in the Planning Graph (the later the better).

For most of the older approaches (typically for STRIPS or MPS), it is quite common to generate more complex macro-operators to penetrate the depth of the search as much as possible. Our method is able to generate more complex macro-operators, if bounds b and c are kept lower and bound d is kept higher. However, such macro-operators are very problem-specific, which makes them unusable for a larger scale of problems in the given domain. Systems like PRODIGY or DHG use static domain analysis and do not require training plans for their learning. Some macro-operators learned by these systems may be unnecessary (i.e. instances of these macro-operators usually do not appear in solutions of most of the problems). State-of-the-art systems Marvin or Macro-FF (SOL-EP version) are built on the FF planner. These systems achieved very promising results, but they cannot be applied with other planners. WIZARD and Macro-FF (CA-ED version) are, like our method, designed as a supporting tool for arbitrary planners without changing their code. WIZARD learns macro-operators genetically from training plans, which follows quite a different policy than our method does. The usability of macro-operators is evaluated by the monitoring of running behavior of planners on updated training problems (by the macro-operators). WIZARD, in comparison to our method, reported better results in the Satellite domain or with the LPG planner, but WIZARD spends many hours on the learning phase, whereas our method spends seconds. Macro-FF (CA-ED version) generates macro-operators from an analysis of static predicates, then adds them into the domain and then generates training plans (with the macro-operators). Unlike that, our method generates macro-operators from training plans gathered from the original training problems and does not require to resolve them (by the planners) in their updated form (with macro-operators). The idea, how the usability of macro-operators is evaluated, is quite similar to our method, but a bit simpler—Macro-FF (CA-ED version) picks the n most frequent macro-operators (assembled from two primitive operators).

In addition, our method detects which primitive operators can be removed (with the risk of losing completeness). For example, in the Depots domain, our method and Macro-FF (CA-ED version) found the same macro-operators. Our method, in addition, removed four (resp. two) primitive operators by using SGPLAN (resp. SATPLAN) for generating the training plans. Removing the primitive operators brought much more benefit to the planners' performance and often to the quality of plans.

9 Conclusion

In this paper, we presented a method for generating macro-operators and removing useless primitive operators from the planning domain. The method explores pairs of actions (not necessarily adjacent) that can be assembled in the given training plans. It results both in the detection of suitable macro-operators and primitive operators that can be removed. The method is designed as a supporting tool for arbitrary planners. The presented evaluation showed that using our method is reasonable and can transparently improve the planning process, especially on more complex planning problems. Nevertheless, the results were obtained by evaluation of IPC benchmarks only. Probably, the main disadvantage of IPC benchmarks rests in similarities of the planning problems (the problems differ only in the number of objects), which makes the analysis of plans structures much easier. In real world applications, it may be more difficult to use our method properly (e.g. we need a set of good training plans, etc.). Classification of such problems where we can remove particular primitive operators without loss of the problems' completeness remains an open problem. Furthermore, more complex macro-operators may contain many parameters that may cause big difficulties to planners. We are also investigating possibilities of pruning potentially useless actions (operators' instances), see Chrpa and Bartak (2009).

In future, we should also focus on a possible extension of our method for generating HTNs. Then we can use some HTN planner, for example, SHOP2 (Nau *et al*., 2003). This idea partially follows the idea listed in Nejati *et al*. (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 should also study action dependencies more from the side of predicates, because it may reveal knowledge that can be used as heuristics for planners.

Acknowledgments

The research is supported by the Czech Science Foundation under contract nos 201/08/0509 and 201/05/H014.

References

Armano, G., Cherch, G. & Vargiu, E. 2005. DHG: a system for generating macro-operators from static domain analysis. In *Proceedings of Artificial Intelligence and Applications (AIA)*, Innsbruck, Austria, 18–23.

Armano, G., Cherch, G. & Vargiu, E. 2003. A parametric hierarchical planner for experimenting abstraction techniques. *Proceedings of IJCAI*, Acapulco, Mexico, 936–941.

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

Bonet, B. & Geffner, H. 1999. Planning as heuristic search: new results. In *Proceedings of ECP*, Durham, UK, Lecture Notes in Computer Science **1809**, 360–372. Springer.

Botea, A., Enzenberger, M., Muller, M. & Schaeffer, J. 2005. Macro-FF: improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* **24**, 581–621.

Chrpa, L. 2008. Generation of macro-operators via investigation of actions dependencies in plans. In *Proceedings of KEPS*, Sydney, Australia. `http://ktiml.mff.cuni.cz/~bartak/KEPS2008/`

Chrpa, L. & Bartak, R. 2008a. Looking for planning problems solvable in polynomial time via investigation of structures of action dependencies. In *Proceedings of SCAI*, Stockholm, Sweden. IOS Press, **173**, 175–180.

Chrpa, L. & Bartak, R. 2008b. Towards getting domain knowledge: plans analysis through investigation of actions dependencies. In *Proceedings of FLAIRS*, Coconut Grove, Florida, USA. AAAI Press, 531–536.

Chrpa, L. & Bartak, R. 2009. Reformulating planning problems by eliminating unpromising actions. In *Proceedings of SARA*, Lake Arrowhead, California, USA. AAAI Press, 50–57.

Chrpa, L., Surynek, P. & Vyskocil, J 2007. Encoding of planning problems and their optimizations in linear logic. In *Proceedings of INAP/WLP*. Technical Report 434, Bayerische Julius–Maximilians–Universität Würzburg, 47–58.

Coles, A., Fox, M. & Smith, K. A. 2007. Online identification of useful macro-actions for planning. In *Proceedings of ICAPS*, Providence, RI, USA. AAAI Press, 97–104.

Coles, A. & Smith, K. A. 2007. Marvin: a heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research* **28**, 119–156.

Dawson, C. & Siklóssy, L. 1977. The role of preprocessing in problem solving systems. In *Proceedings of IJCAI* **1**, Cambridge, MA, USA, 465–471.

Fikes, R. & Nilsson, L. 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Inteligence* **2**(3–4), 189–208.

Fox, M. & Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of IJCAI*, Stockholm, Sweden, 956–961.

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

Gerevini, A. & Serina, I. 2002. LPG: a planner based on local search for planning graphs with action costs. In *Proceedings of AIPS*, Toulouse, France. AAAI Press, 13–22.

Ghallab, M., Nau, D. & Traverso, P. 2004. *Automated Planning, Theory and Practice*. Morgan Kaufmann Publishers.

Gimenez, O. & Jonsson, A. 2007. On the hardeness of planning problems with simple causal graphs. In *Proceedings of ICAPS*, Providence, RI, USA. AAAI Press, 152–159.

Grandcolas, S. & Pain-Barre, C. 2007. Filtering, decomposition and search space reduction for optimal sequential planning. In *Proceedings of AAAI*, Vancouver, British Columbia, Canada. AAAI Press, 993–998.

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

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

Iba, G.A. 1989. A Heuristic approach to the discovery of macro-operators. *Machine Learning* **3**, 285–317.

Iba, G. A. 1985. Learning by discovering macros in puzzle solving. In *Proceedings of IJCAI*, Los Angeles, California, USA, 640–642.

Katz, M. & Domshlak, C. 2007. Structural patterns of tracable sequentialy-optimal planning. In *Proceedings of ICAPS*, Providence, RI, USA. AAAI Press, 200–207.

Kautz, H., Selman, B. & Hoffmann, J 2006. Satplan: planning as satisfiability. In *Proceedings of IPC*. `http://zeus.ing.unibs.it/ipc-5/booklet/deterministic11.pdf`

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.

Kvanström, J. & Magnusson, M. 2003. TALplanner in the third international planning competition: extensions and control rules. *Journal of Artificial Intelligence Research* **20**, 343–377.

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

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

McCluskey, T. L. 1987. Combining weak learning heuristics in general problem solvers. In *Proceedings of IJCAI*, Milan, Italy, 331–333.

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

Minton, S. 1985. Selectively generalizing plans for problem-solving. In *Proceedings of IJCAI*, Los Angeles, California, USA, 596–599.

Minton, S. & Carbonell, J. G. 1987. Strategies for learning search control rules: an explanation-based approach. In *Proceedings of IJCAI*, Milan, Italy, 228–235.

Nau, D., Au, T., Ilghami, O., Kuter, U., Mudrock, J., Wu, D. & Yaman, F. 2003. SHOP2: an HTN planning system. *Journal of Artificial Intelligence Research* **20**, 379–404.

Nejati, N., Langley, P. & Konik, T. 2006. Learning hierarchical task networks by observation. In *Proceedings of ICML*, Pittsburgh, Pennsylvania, USA, *ACM International Conference Proceeding Series 148*, 665–672.

Newton, M. H., Levine, J., Fox, M. & Long, D. 2007. Learning macro-actions for arbitrary planners and domains. In *Proceedings of ICAPS*, Providence, Rhode Island, USA, 256–263.

Richter, S. & Westphal, M 2008. The LAMA planner using landmark counting in heuristic search In *Proceedings of the 6th IPC*. `http://ipc.informatik.uni-freiburg.de/`

Vidal, V. & 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. & Jiang, Y. 2005. Arms: action-relation modelling system for learning action models. In *Proceedings of ICKEPS*. `http://scom.hud.ac.uk/scomtlm/competition/papers/paper6.pdf`