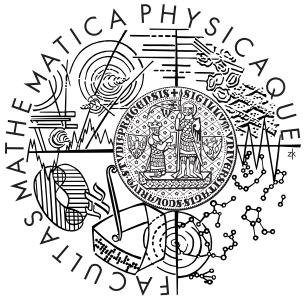
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL DISSERTATION



Mgr. Lukáš Chrpa

Learning for Classical Planning

Department of Theoretical Computer Science and
Mathematical Logic

Thesis advisor: doc. RNDr. Roman Barták, Ph.D.
Theoretical Computer Science

2009

Acknowledgements

I would like to thank my advisor doc. RNDr. Roman Barták, Ph.D. for his valuable guidance and support during my studies. I would like also to thank anonymous reviewers from various conferences or workshops for their comments and feedback. My thanks also go to my family and friends for their moral support during my studies. 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.

I declare that this doctoral thesis was composed by myself and all presented results are my own unless otherwise stated. I agree with making the thesis publicly available.

In Prague, June 28, 2009 Lukáš Chrpa

Contents

List of Tables

List of Figures

Abstract

This thesis is mainly about classical planning for artificial intelligence (AI).

In planning, we deal with searching for a sequence of actions that changes the environment from a given initial state to a goal state. Planning problems in general are ones of the hardest problems not only in the area of AI, but in the whole computer science. Even though classical planning problems do not consider many aspects from the real world, their complexity reaches EXPSPACE-completeness. Nevertheless, there exist many planning systems (not only for classical planning) that were developed in the past, mainly thanks to the International Planning Competitions (IPC).

Despite the current planning systems are very advanced, we have to boost these systems with additional knowledge provided by learning. In this thesis, we focused on developing learning techniques which produce additional knowledge from the training plans and transform it back into planning domains and problems. We do not have to modify the planners.

The contribution of this thesis is included in three areas. First, we provided theoretical background for plan analysis by investigating action dependencies or independencies. Second, we provided a method for generating macro-operators and removing unnecessary primitive operators. Experimental evaluation of this method brought promising results, because we were able in many cases to boost planners' performance at the cost of the little time spent on learning. Third, we provided a method for eliminating unnecessary actions. Like the previous method, this method gathered also very promising results, because we were also able to boost planners' performance at the cost of the little time spent on learning.

Chapter 1

Introduction

This thesis presents results achieved during my PhD study at Charles University in Prague. This thesis is about learning techniques used for classical planning for Artificial Intelligence (AI).

AI planning deals with the problem of finding a sequence of actions transforming the world from some initial state to a desired goal state. Generally, solving planning problems is about search, where we exhaustively test possible alternatives (action sequences in this case) whether they lead (or not) to the solution. Even for the simplest planning problems the number of such alternatives is very high which classifies planning problems as ones of the hardest problems in AI[1]. It means that we still require innovations which can improve techniques for solving planning problems. In addition, planning problems have utilization in practice, for example, in manufacture process, autonomous agent planning etc.

The research in the planning area is holding several decades. During this time, many promising planning techniques have been developed. Especially, since 1998, every two years the International Planning Competition (IPC) is being organized. It results in development of many advanced planning systems and significant improvement of the planning process. Despite this improvement, many planning problems still remain hard and challenging.

An opportunity to help the planners rests in gathering additional knowledge hidden in planning domains or problems, so called learning for planning. By this additional knowledge we can mean macro-operators, relations

[1]Classical planning problems, we are focusing on in this thesis, do not consider many aspects normally appearing in the real world, but still the complexity of classical planning remains very high (EXPSPACE-complete).

between predicates or operators etc. Despite the idea of learning for planning is not very new [25], there are not so many techniques supporting this. In 2008, the learning track was firstly organized on the IPC. It may cause that learning for planning become the one of the most growing research areas in near future.

1.1 Contributions

The contributions presented in this thesis are related to learning for classical planning. We focused on developing such learning techniques that produce additional knowledge from training plans and transform it back into planning domains or problems. It means that we do not have to modify the planners' source code.

First, we proposed a theoretical background describing action dependencies and independencies in plans and possibilities of plan decompositions [18]. In addition, we studied possibilities of making plans shorter [15] and we found out that if actions match specific criteria, then they can be removed from the plan without losing its validity.

Then, we proposed a method for generating macro-operators by investigation of action dependencies and independencies in training plans. It has been initially studied in [18] and more thoroughly studied in [16]. The method is used for learning macro-operators from simpler training plans (generated by the common planners), 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 the source code of planners.

Finally, we proposed a method that is able to detect connectivity (called entanglements) between initial or goal predicates and operators from given training plans and encode it back into planning domains and their related planning problems. Again the transformed domains and problems can be easily passed to existing planners. The advantage of the method rests in pruning unnecessary instances of operators.

All the contributions presented in this thesis were published in international conferences or workshops. All the papers were reviewed by at least two experts in the given areas.

1.2 Thesis overview

This thesis is organized as follows:

Chapter 2: Classical planning – This chapter is devoted to preliminaries for classical planning and give the readers an overview of the research status and existing approaches in this area.

Chapter 3: Learning for classical planning – This chapter is devoted to a brief introduction and overview of learning techniques that are currently used for classical planning.

Chapter 4: Action dependencies in plans – This chapter is devoted to a description of action dependencies and independencies in plans and discusses some theoretical results in this area.

Chapter 5: Learning macro-operators by plans investigation – This chapter is devoted to the method for learning macro-operators.

Chapter 6: Eliminating unpromising actions – This chapter is devoted to eliminating actions that are unnecessary for the planning process.

Chapter 7: Conclusion and future work – In this chapter, we conclude this thesis and give possible directions of future research.

Summarized, chapters 2 and 3 serve as an introduction of classical planning and learning techniques used in planning and discuss related works in these areas. Chapters 4-6 presents new results gathered during my PhD studies.

Chapter 2

Classical planning

This chapter is devoted to preliminaries for classical planning which are required to fully understand to the text of this thesis as well as to give the readers an overview of the research status and existing approaches in this area.

2.1 Motivation

Planning [34] is an important branch of artificial intelligence (AI). Despite significant improvements of the planning process during last decades many planning problems remain hard and challenging, because planning itself has been proved [12, 28] as one of the hardest problems in the whole AI.

Traditionally, AI planning deals with the problem of finding a sequence of actions transforming the world from some initial state to a desired goal state. In figure 2.1 there is showed a simple example of a classical planning problem. In this particular case we want to find a sequence of actions for a robotic hand for swapping blocks A and B. We consider that the hand is able to unstack a block from another block, pick up a block from the table, stack a block to another block and finally put down a block to the table. Intuitively, we know that every action can be performed if particular conditions are satisfied (for instance the hand must be empty before picking or unstacking the block). We also know that after the performance of some action the world will be modified. The feasible solution (plan) of our example is showed in figure 2.2. In addition, the presented solution is optimal (regarding the plan's length) because there does not exist any plan with a smaller length.

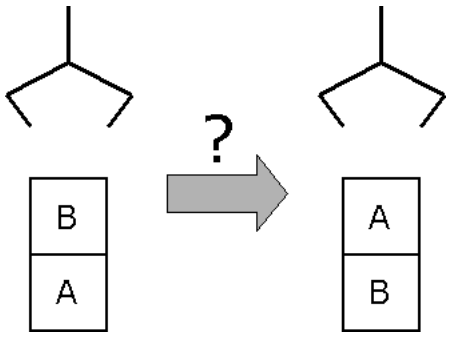


Figure 2.1: Example of a planning problem

Such kind of a problem we have just informally introduced is usually called a *classical planning* problem. In classical planning we consider such problems that do not care about time, resources and uncertainty. Nevertheless there exist extensions of classical planning that can handle those features.

Planning is also essential for many real world problems where an application of autonomous robots or agents is necessary. A good example of such an application is space exploring missions. One of the first successful applications of planning in space exploration was the Deep Space 1 mission [7]. During the mission there was applied the Autonomous Remote Agent system [60], software based on planning techniques. Another application of planning in space exploration was used during the Mars Rover mission [1, 53], where the autonomous vehicles (rovers) were successfully used for Mars exploration. The newest results in the area of planning usage in space exploration are presented for example in [13, 14]. Beside the space exploration there are many other applications of planning in practice, for example planning of manufacturing processes [61], planning routes for drivers with different preferences [71] or planning autonomous robots in rescue missions [24, 67].

2.2 Representation

In the previous section we informally introduced classical planning. This section is devoted to a formal description of classical planning. We discuss three different ways how to represent classical planning [34] where all the ways are equivalent in their expressive power. Classical planning is based on

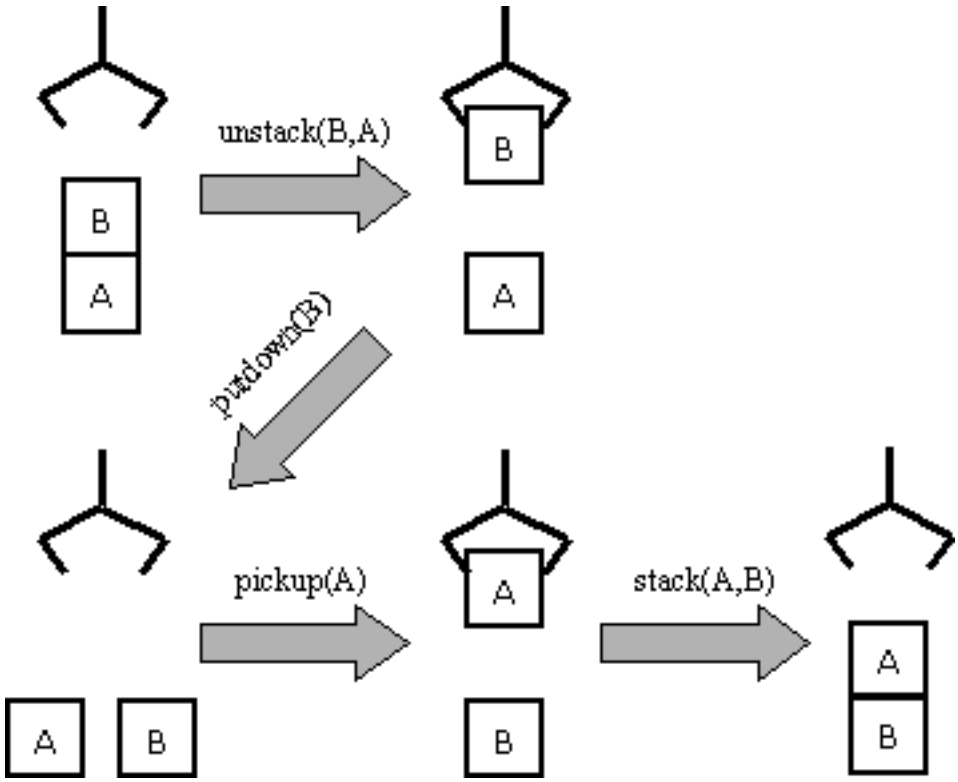


Figure 2.2: Example of a solution of a planning problem

a state-transition system[1] which is a deterministic, static, finite and fully observable with implicit time and restricted goals [34]. In addition, we assume that this system does not consider contingent events (i.e., events that may suddenly change the state).

2.2.1 Set-theoretic representation

In the *set-theoretic representation* there is a finite set of propositional symbols intended to represent various propositions about the world. For example, proposition $carrying A$ specifies that the crane is carrying block A and proposition $clear B$ specifies that no box is stacked on box B.

[1]State-transition system is an abstract machine that consists of a set of states and transitions between states. State transition systems with a finite number of states and transitions can be represented as directed graphs.

Definition 2.1: Let $L = \{p_1, \ldots p_n\}$ be a finite set of propositional symbols. A *(set-theoretic) planning domain* on L is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq 2^L$, i.e., each *state* $s \in S$ is a subset of L. If $p \in s$ then p holds in the state represented by s, and if $p \notin s$ then p does not hold in the state represented by s.

- Each *action* $a \in A$ is a triple of subsets of L, which we will write as $a = (p(a), e^-(a), e^+(a))$. The set $p(a)$ is called the *precondition* of a, the set $e^-(a)$ is called the *negative effects* of a and the set $e^+(a)$ is called the *positive effects* of a. We require these two sets of effects to be disjoint ($e^-(a) \cap e^+(a) = \emptyset$). Action a is *applicable* (or performable) to state s if $p(a) \subseteq s$.

- S has the property that if $s \in S$ then for every action a that is applicable to s, $(s \setminus e^-(a)) \cup e^+(a) \in S$. It means that if an action is applicable to a state then after the application of the action to the state, another state is produced.

- The state-transition function $\gamma : S \times A \rightarrow S$ is defined such that $\gamma(s, a) = s \setminus e^-(a)) \cup e^+(a)$ if $a \in A$ is applicable to $s \in S$, $\gamma(s, a)$ is undefined otherwise.

- We can extend the transition function for sequences of actions. $\gamma(s, \langle\rangle) = s$, $\gamma(s, \langle a_1, \ldots, a_k\rangle) = \gamma(\gamma(s, a_1), \langle a_2, \ldots, a_k\rangle)$ if a_1 is applicable to s and $\gamma(s, \langle a_1, \ldots, a_k\rangle)$ is undefined otherwise.

Definition 2.2: A *(set-theoretic) planning problem* is a triple $P = (\Sigma, s_0, g)$ where:

- $s_0 \in S$ is an *initial state.*

- $g \subseteq L$ is a set of *goal propositions* that give the requirements that a state must satisfy in order to be a goal state, i.e. $S_g = \{s \in S \mid g \subseteq s\}$ is a set of goal states.

Definition 2.3: A *plan* is a sequence of actions $\pi = \langle a_1, \ldots a_k\rangle$, where $k \geq 0$. The *length* of the plan π is $|\pi| = k$, where k represents the number of actions. Let $P = (\Sigma, s_0, g)$ be a planning problem. A plan π is a solution

for P if $g \subseteq \gamma(s_0, \pi)$. π is *minimal* if no other solution for P contains fewer actions than π.

Remark 2.4: In classical planning the minimality of a plan means the optimality of the plan.

We have formally defined what the planning domain, the planning problem and the plan mean (in the set-theoretic representation). Straightforwardly, the planning domain describes the world with its own rules, the planning problem describes the task we want to solve in the world and the plan represents the solution of the task. In practice the planning domain is defined only by a (finite) set of proposition symbols (L) and a (finite) set of actions (A). This is because the state space (S) can be very large (up to 2^L elements) even for small domains. If we recall the example from figure 2.1 then the (set-theoretic) planning domain can be defined as showed in figure 2.3 and the (set-theoretic) planning problem can be defined as showed in figure 2.4. Despite the set-theoretic representation is quite straightforward it has a big disadvantage. The set-theoretic planning domain depends on the exact number of objects in a particular planning problem because we have to define the propositions and the actions for every object apart.

2.2.2 Classical representation

The *classical representation* generalizes the set-theoretic representation by replacing the proposition symbols by the first-order logic predicates. It gives us a possibility to bypass the main disadvantage of the set-theoretic representation allowing us to define the planning domains independently of the exact number of objects in particular planning problems.

Definition 2.5: A *planning operator* is a 4-tuple $o = (n(o), p(o), e^-(o), e^+(o))$ whose elements are defined as follows:

- $n(o)$, the *name* of the operator o, is an expression of the form $name(x_1, \ldots, x_k)$ where *name* is called an *operator symbol (or name)*, $x_1, \ldots, x_k$ are all of the variable symbols that appear in the operator, and *name* is unique.
- $p(o), e^-(o)$ and $e^+(o)$ are generalizations of the preconditions, negative and positive effects of the set-theoretic action (instead of being sets of propositions, they are sets of predicates).

`Proposition symbols:` $onAB$, $onBA$, $ontableA$, $ontableB$, $clearA$, $clearB$, $carryingA$, $carryingB$, $handempty$.

`Actions:`

$$
\begin{array}{rlcl}
pickupA = \{ & p & = & \{ontableA, clearA, handempty\}, \\
& e^- & = & \{ontableA, clearA, handempty\}, \\
& e^+ & = & \{carryingA\} \quad \} \\
pickupB = \{ & p & = & \{ontableB, clearB, handempty\}, \\
& e^- & = & \{ontableB, clearB, handempty\}, \\
& e^+ & = & \{carryingB\} \quad \} \\
putdownA = \{ & p & = & \{carryingA\}, \\
& e^- & = & \{carryingA\}, \\
& e^+ & = & \{ontableA, clearA, handempty\} \quad \} \\
putdownB = \{ & p & = & \{carryingB\}, \\
& e^- & = & \{carryingB\}, \\
& e^+ & = & \{ontableB, clearB, handempty\} \quad \} \\
unstackAB = \{ & p & = & \{onAB, clearA, handempty\}, \\
& e^- & = & \{onAB, clearA, handempty\}, \\
& e^+ & = & \{carryingA, clearB\} \quad \} \\
unstackBA = \{ & p & = & \{onBA, clearB, handempty\}, \\
& e^- & = & \{onBA, clearB, handempty\}, \\
& e^+ & = & \{carryingB, clearA\} \quad \} \\
stackAB = \{ & p & = & \{carryingA, clearB\}, \\
& e^- & = & \{carryingA, clearB\}, \\
& e^+ & = & \{onAB, clearA, handempty\} \quad \} \\
stackBA = \{ & p & = & \{carryingB, clearA\}, \\
& e^- & = & \{carryingB, clearA\}, \\
& e^+ & = & \{onBA, clearB, handempty\} \quad \}
\end{array}
$$

Figure 2.3: Example of a set-theoretic planning domain

`Initial state:`

$$s_0 = \{ontableB, onAB, clearA, handempty\}$$

`Goal propositions:`

$$g = \{ontableA, onBA, clearB, handempty\}$$

Figure 2.4: Example of a set-theoretic planning problem

Remark 2.6: The states in the classical representation are represented by sets of grounded predicates. Actions are grounded instances of planning operators. Actions applicability and results of actions performance are similar to the set-theoretic representation (instead of the propositions we have the grounded predicates). We do not consider the function symbols in the classical representation.

Definition 2.7: Let L be a first-order language that has finitely many predicate symbols and constant symbols. A *(classical) planning domain* in L is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq 2^{\{\text{all ground atoms of } L\}}$
- $A = \{\text{all ground instances of operators in } O\}$, where O is a set of operators as defined before.
- $\gamma(s, a) = s \setminus e^-(a)) \cup e^+(a)$ if $a \in A$ is applicable to $s \in S$, and $\gamma(s, a)$ is undefined otherwise.
- S is closed under γ which means that if $s \in S$ then for every action $a \in A$ that is applicable to s, $\gamma(s, a) \in S$.

Definition 2.8: A *(classical) planning problem* is a triple $P = (\Sigma, s_0, g)$, where:

- $s_0 \in S$ is the *initial state*.
- g is any set of grounded predicates - the *goal*
- $S_g = \{s \in S \mid g \subseteq s\}$

`Predicate symbols:` $on(X,Y)$, $ontable(X)$, $clear(X)$, $carrying(X)$, $handempty$.

`Actions:`

$$
\begin{array}{rlcl}
pickup(X) = \{ & p & = & \{ontable(X), clear(X), handempty\}, \\
& e^- & = & \{ontable(X), clear(X), handempty\}, \\
& e^+ & = & \{carrying(X)\} \quad \} \\
putdown(X) = \{ & p & = & \{carrying(X)\}, \\
& e^- & = & \{carrying(X)\}, \\
& e^+ & = & \{ontable(X), clear(X), handempty\} \quad \} \\
unstack(X,Y) = \{ & p & = & \{on(X,Y), clear(X), handempty\}, \\
& e^- & = & \{on(X,Y), clear(X), handempty\}, \\
& e^+ & = & \{carrying(X), clear(Y)\} \quad \} \\
stack(X,Y) = \{ & p & = & \{carrying(X), clear(Y)\}, \\
& e^- & = & \{carrying(X), clear(Y)\}, \\
& e^+ & = & \{on(X,Y), clear(X), handempty\} \quad \}
\end{array}
$$

Figure 2.5: Example of a classical planning domain

`Initial state:`

$$s_0 = \{ontable(B), on(A,B), clear(A), handempty\}$$

`Goal predicates:`

$$g = \{ontable(A), on(B,A), clear(B), handempty\}$$

Figure 2.6: Example of a classical planning problem

We have formally defined the classical representation for classical planing, where we used the predicates instead of the propositions that we used in the set-theoretic representation. In practice the planning domain is defined (likewise in the set-theoretic representation) by a (finite) set of predicates (L) and a (finite) set of planning operators (O). If we recall again the example from figure 2.1 then the (classical) planning domain can be defined as showed in figure 2.5 and the (classical) planning problem can be defined as showed in figure 2.6. In the classical representation the planning domain does not depend on the exact number of objects in particular planning problems. This brings a big advantage against the set-theoretic representations, because we can use the same planning domain definition for all the planning problems that can be defined under this domain. In addition, the classical planning problems can be easily transformed to the set-theoretic planning problems by grounding.

2.2.3 State-variable representation

The *state-variable representation* differs from the set-theoretic and classical representations by using of functions instead of the logic-based propositions or predicates. Let D be a set of all constant symbols in a planning domain. D can be partitioned into various classes of constants (for instance trucks, boxes, cranes etc.). We will represent each constant by an *object symbol* (i.e., truck1, box2,crane1 etc.). To write the unground expressions, we will use *the object variables* that range over the sets of constants. Each object variable v has a range D^v that is the union of one or more classes. A *term* is either a constant or an object variable.

Definition 2.9: Let $f_o : S \rightarrow P$ be an explicit function, such that the value of $f_o(s)$ gives the unique property $p \in P$ of object o in state s. Here, the symbol f_o is a *state-variable symbol* that denotes a *state-variable function* whose values are characteristic attributes of the current state.

For example, state-variable function $on(A)$ specifies which block is stacked on the block A, i.e., if $on(A) = B$, then block B is stacked on the block A.

Definition 2.10: A k-ary *state variable* is an expression of the form $x(v_1, \ldots, v_k)$, where x is a *state-variable symbol*, and each v_i is either an object symbol (partitioned into disjoint classes, corresponding to the object

of the domain) or an object variable (ranges over a class or the union of classes of constants). A state variable denotes an element of the state-variable function:

$$x : D_1^x \times D_2^x \times \ldots \times D_k^x \times S \rightarrow D_{k+1}^x$$

where $D_i^x \in D$ is the union of one or more classes.

Definition 2.11: A *planning operator* is a 4-tuple $o = (n(o), p(o), e^-(o), e^+(o))$ whose elements are defined as follows:

- $n(o)$, the *name* of the operator o, is an expression of the form $name(u_1, \ldots, u_k)$ where *name* is called an *operator symbol (or name)*, $u_1, \ldots, u_k$ are all of the variable symbols, and *name* is unique.
- $p(o)$ is a set of expressions on the state variables and relations (for instance, $x(t_1, \ldots t_k) = y(t_1, \ldots t_l)$ or $x(t_1, \ldots t_k) > z(t_1, \ldots t_m)$)
- $e(o)$ is a set of assignments of values to the state variables of the form $x(t_1 \ldots, t_k) \leftarrow t_{k+1}$, where each t_i is a term in the appropriate range.

Definition 2.12: Let L be a planning language in the state variable representation defined by a finite set of state variables X and by a finite set of rigid relations R (properties that do not vary from one state to another). A *(state-variable) planning domain* in L is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq \prod_{x \in X} D_x$, where D_x is the range of the ground state variable x; a state s is denoted $s = \{(x = c) \mid x \in X\}$, where $c \in D_x$.
- $A = \{$all ground instances of operators in O that meet the relations in $R\}$, where O is a set of operators as defined before; an action a is applicable to a state s iff every expression $(x = c) \in p(a)$ is also in s.
- $\gamma(s, a) = \{(x = c) \mid x \in X\}$, where c is specified by an assignment $x \leftarrow c$ in $e(a)$ if there is such an assignment, otherwise $(x = c) \in s$.
- S is closed under γ which means that if $s \in S$ then for every action a applicable to s $\gamma(s, a) \in S$.

Definition 2.13: A *(state-variable) planning problem* is a triple $P = (\Sigma, s_0, g)$, where s_0 is an initial state in S and the goal g is a set of expressions on the state variables in X.

`State-variable functions:` $on(X)$, $ontable(X)$, $holding()$.

`Actions:`

$$
\begin{aligned}
pickup(X) = \{ \quad & p = \{ontable(X) = 1, on(X) = null, holding = null\}, \\
& e = \{ontable(X) \leftarrow 0, holding \leftarrow X\} \quad \}
\end{aligned}
$$

$$
\begin{aligned}
putdown(X) = \{ \quad & p = \{holding = X\}, \\
& e = \{ontable(X) \leftarrow 1, holding \leftarrow null\} \quad \}
\end{aligned}
$$

$$
\begin{aligned}
unstack(X,Y) = \{ \quad & p = \{on(Y) = X, on(X) = null, holding = null\}, \\
& e = \{on(Y) \leftarrow null, holding \leftarrow X\} \quad \}
\end{aligned}
$$

$$
\begin{aligned}
stack(X,Y) = \{ \quad & p = \{holding = X, on(Y) = null\}, \\
& e = \{on(Y) \leftarrow X, holding \leftarrow null\} \quad \}
\end{aligned}
$$

Figure 2.7: Example of a state-variable planning domain

`Initial state:`

$s_0 = \{ontable(B) \leftarrow 1, ontable(A) \leftarrow 0, on(A) \leftarrow B, on(B) \leftarrow null, holding \leftarrow null\}$

`Goal predicates:`

$$g = \{ontable(A) = 1, on(B) = A, on(B) = null, handempty = null\}$$

Figure 2.8: Example of a state-variable planning problem

We have formally defined the state-variable representation for classical planing, where, instead of the set-theoretic or the classical representations, we used the (state-variable) functions for representation of the properties of the world. Here, we do not have to distinguish between the positive and negative effects, because the effects in the state-variable representation mean changes of the values of state-variable functions. In practice the planning domain is defined (likewise in the other representations) by a (finite) set of state-variable functions and a (finite) set of planning operators. If we recall again the example from figure 2.1 then the (state-variable) planning domain can be defined as showed in figure 2.7 and the (state-variable) planning problem can be defined as showed in figure 2.8. Like in the classical representation the state-variable planning domain does not depend on the exact number of objects in particular planning problems. As we said before the state-variable representation and the classical representation have equivalent expressivity. The predicates from the classical representation can be easily represented by the state-variable functions with values $\{0, 1\}$. State-variable functions from the state-variable representation can be represented by predicates as follows:

- $x(t_1, \ldots, t_n) = v$ (appearing in the initial states, the goals and the preconditions) is replaced by $x(t_1, \ldots, t_n, v)$.
- $x(t_1, \ldots, t_n) \leftarrow v$ (appearing in the effects) is replaced by $x(t_1, \ldots, t_n, v)$ (positive effect) and $\neg x(t_1, \ldots, t_n, w)$ (negative) , and also put $x(t_1, \ldots, t_n, w)$ into the operator's precondition.

Classical and state-variable representations are more expressive than the set-theoretic representation even though all the representation schemes can still represent the same set of planning domains. A set-theoretic representation of a planning problem may take exponentially more space than the equivalent classical or state-variable representation.

2.3 Complexity of classical planning

As mentioned before, classical planning, even though it is decidable [28], is one of the hardest classes of AI problems. For the set-theoretic (or ground classical) representation of classical planning, it has been proven in [12] that the complexity of plan existence is PSPACE-complete depending on the length of the particular problem representation. For classical representation, it has been proven in [28] that the complexity of plan existence is

EXPSPACE-complete depending on the length of the particular problem representation. Similarly, for state-variable representation, it has been proven in [46] that the complexity of plan existence is EXPSPACE-complete depending on the length of the particular problem representation. The difference of the complexity between set-theoretic representation and the other representations rests in the fact that an equivalent set-theoretic representation may take exponentially more space than the other representations. These complexity results refer to classical planning in general, giving upper bounds to domain-independent planning systems. However, these bounds lie too high, making classical planning (recall that classical planning is quite restricted) very hard and challenging. On the other hand, it has been proven in [37, 40] that many planning domains and their related planning problems can be much easier (P or NP).

2.4 Extensions to classical planning

Classical planning itself is not expressive enough to be able to take into account different quantities such as time, resources, or uncertainty.

Planning with resources [6, 36] extends classical planning by adding of (consumable) resources like fuel, energy or containers capacity. Resources are usually represented by integer functions with value range $\langle 0, c \rangle$, where c represents the maximal capacity of a particular resource storage.

Temporal planning [2, 3, 52] extends classical planning by adding durations into actions. Instead of having immediate effect of action performance in classical planning, in temporal planning every action performance takes some time, i.e., effects of an action may be considered when the action performance is finished.

Planning under uncertainty [23, 30] extends classical planning by adding uncertainty to effects of actions and to initial states. Usually uncertainty is represented by probability values attached to predicates, telling us what a chance of a predicate to be presented in a particular state is. A special case of planning under uncertainty is *partially observable planning* [70], where we are not able to gather full information about the world.

Hierarchical task network (HTN) planning [27, 62] uses a different point of view to planning. In HTN planning the problem definition does not consist of primitive operators only like in the classical planning but also the non-primitive operators assembled from the primitive and non-primitive ones.

2.5 Planning domain definition language

Planning Domain Definition Language (PDDL) [33] is a LISP-based language for formalization of planning domains and planning problems. PDDL became very popular since it has been used in the *International Planning Competitions (IPC)*. Foundations of PDDL are related to *STRIPS* language [29]. It allows to define the planning operators and the planning problems analogous to the classical representation. PDDL, however, incorporates many other additional features. The most common one is *typing* which assigns objects their own types (like blocks, depots, trucks etc.). If we extend our example by allowing multiple number of robotic hands, then the planning domain can be represented in PDDL like in figure 2.9 and a sample planning problem can be represented in PDDL like in figure 2.10.

One of the well known extensions of PDDL is *ADL* [64]. ADL allows to encode negative preconditions, conditional effects, disjunctive goals etc. However, those features required more complex kind of representation, because none of the three representations discussed in this thesis do not fully support these features. Another extension of PDDL called *fluents* [56] allows to define the state-variable or numeric functions. Last but not least interesting extension of PDDL is *Probabilistic PDDL* [69] which is used for planning under uncertainty. PDDL, certainly, has many more extensions that can be used to represent, for example, durative actions, additional constraints and so on.

2.6 State-of-the-art planners

If we recall that the classical planning is based on a restricted state-transition system, then the first idea how to solve planning problems might refer to a problem of path finding in directed graphs. Despite it looks quite straightforwardly such an approach is, however, almost impossible even for simpler problems, because the number of states in such state-transition systems is extremely huge. Classical approaches [34] use classical forward (from an initial state to a goal state) or backward (from a goal state to an initial state) search, usually accommodated with heuristics [10]. Neo-classical approaches are based on a structure called *planning graph* [9] which can significantly improve the planning process by removing of symmetries (actions that can be performed in any order obtaining the same state) that may occur during

```
  (define (domain blocksworld)
(:requirements :typing) (:types block hand)
(:predicates (clear ?b-block)
             (on-table ?b - block)
             (empty ?h - hand)
             (holding ?h - hand ?b - block)
             (on ?b1 ?b2 - block))
(:action pickup
  :parameters (?h - hand ?b - block)
  :precondition (and (clear ?b) (on-table ?b) (empty ?h))
  :effect (and (holding ?h ?b) (not (clear ?b)) (not (on-table ?b))
               (not (empty ?h))))
(:action putdown
  :parameters  (?h - hand ?b - block)
  :precondition (holding ?h ?b)
  :effect (and (clear ?b) (empty ?h) (on-table ?b)
               (not (holding ?h ?b))))
(:action stack
  :parameters  (?h - hand ?b ?underb - block)
  :precondition (and (clear ?underb) (holding ?h ?b))
  :effect (and (empty ?h) (clear ?b) (on ?b ?underb)
               (not (clear ?underb)) (not (holding ?h ?b))))
(:action unstack
  :parameters  (?h - hand ?b ?underb - block)
  :precondition (and (on ?b ?underb) (clear ?b) (empty ?h))
  :effect (and (holding ?h ?b) (clear ?underb)
               (not (on ?b ?underb)) (not (clear ?b)) (not (empty ?h)))))
```

Figure 2.9: Example of a typed strips PDDL planning domain

```
(define (problem blocksworld-n10-1)
(:domain blocksworld)
(:requirements :typing)
(:objects h - hand b1 b2 b3 b4 b5 b6 b7 b8 b9 b10  - block)
(:init
 (empty h)
 (on b1 b7)
 (on-table b2)
 (on-table b3)
 (on-table b4)
 (on-table b5)
 (on b6 b9)
 (on-table b7)
 (on b8 b10)
 (on-table b9)
 (on b10 b6)
 (clear b1)
 (clear b2)
 (clear b3)
 (clear b4)
 (clear b5)
 (clear b8)
)
(:goal (and
 (on b4 b2)
 (on b5 b10)
 (on b7 b5)
 (on b8 b9)
 (on b10 b4))
) )
```

Figure 2.10: Example of a typed strips PDDL planning problem

classical backtrack search.

Since 1998, planning systems recorded a significant improvement thanks to the *International Planning Competition* (IPC)[2]. IPC is being held biannually, where many planners are challenging each other in a growing number of competition's tracks. One of the most famous planners is *Fast Forward* (FF) [42] which won the 2nd IPC and was a top performer in STRIPS track on the 3rd IPC. FF is a forward chaining heuristics state space planner, where the heuristics tries to estimate the solution length by relaxing of the original planning problem by removing of negative effects of planning operators. FF has many extensions - for example Metric-FF [41] which can handle with numeric state variables or Probabilistic-FF [26] for planning under uncertainty. The other famous planner which was awarded on the 3-rd and the 4-rd IPC is *LPG* [32]. LPG is based on local search and planning graph techniques and can handle numeric functions and durative actions. The best performance in optimal deterministic propositional planning track of the 5th IPC had *SATPLAN* [48] and *MAXPLAN* [68], planners based on the transformation of planning problems into boolean satisfiability [47]. SATPLAN also won the same track on the 4th IPC. An absolute winner of the 5th IPC in the all sub-optimal deterministic tracks was *SGPLAN* [44]. SGPLAN is a planner that decomposes problems into subproblems, then it solves them by FF-based techniques. *LAMA* [65] won the sub-optimal track in the 6th IPC. LAMA is based on FF-based heuristics and Causal Graph heuristics (similarly to Fast Downward planner).

[2]http://ipc.icaps-conference.org

Chapter 3

Learning for classical planning

This chapter is devoted to a brief introduction and overview of basic learning techniques that are used for classical planning. We discuss mainly techniques related to analysis of planning domains, problems or plans that can gather knowledge used for improving of solving efficiency for more complex planning problems.

3.1 Motivation

Despite a significant improvement of the planning systems (recall the previous section) during recent years many planning problems remain hard and challenging. This is, because the planners are using 'brute force' search techniques that often recompute results gathered earlier even though such a search is usually guided by good heuristics. To avoid (or lower) such recomputations it is required to use a knowledge base. One of the most significant works in this area [25] discusses a system called REFLECT which uses preprocessing for gathering additional knowledge (detecting incompatible predicates or generating macro-operators) required by the solver. Control rules [59] (i.e., logical rules describing dependencies between predicates or operators) are designed as an additional support for the planners. For instance, if we recall our example (BlocksWorld domain) we can (informally) define the following rules:

1. Take down the initial 'towers' of blocks, i.e., all blocks will be on the table.

2. Build the goal ‘towers‘.

Simply, we can see that the rules can significantly help the planner making every BlocksWorld problems. Using of (control) rules is not a new technique in planning. There are planners supporting control rules, for instance, well known TALplanner [51] which is based on temporal logic. However, TALplanner (and other similar based systems) are not able to learn the rules by themselves. It means that it is required to define the rules by domain experts or use some approach based on machine learning techniques. In the following text we focus on such learning techniques that can be used domain independently (in terms of generality - i.e. the techniques are applicable to an arbitrary domain even though they generate domain dependent knowledge).

3.2 Existing techniques for domain analysis

Different kinds of domain, problem or plan analysis can be useful for developing heuristics or gathering other additional knowledge for the planners. Work [31] refers about action languages, formalisms describing the relations between the effects of actions. This work describes a couple of action languages which are divided into two main classes (action description languages for describing the system and action query languages for making queries to the system). Another work [54] defines a language for expressing causal knowledge and gives an approach of formalizing actions. However, the contribution of both of the works is mainly theoretical. Opposite to these works we did not only focus on the theoretical framework, but we also took into account the possible practical usage (for example, generation of the macro-operators)

One of the useful tools for analysis of planning problems is called Landmarks [43] - facts that must be true in some point of every valid plan. Landmarks serve like ‘navigators‘ of the searching process. However, finding such Landmarks and their ordering is PSPACE-complete. Hopefully, there is a greedy algorithm for finding ordered Landmarks which is, however, not sound but anyway it gathered good results in connection with FF or LPG planner.

In [49] there is presented a structure called Causal Graph, useful for domain (and problem) analysis. Graph $G = (V, E)$ (for domain analysis) is a directed graph, where vertices (V) are representing propositions and edges (E) are defined in the following way:

- If there exists an action a such that $p \in e^+(a) \cup e^-(a) \wedge q \in e^+(a) \cup e^-(a)$, then $(p, q) \in E$ and $(q, p) \in E$
- If there exists an action a such that $p \in e^+(a) \cup e^-(a) \wedge q \in p(a)$, then $(p, q) \in E$

Causal Graph is obtained from G in such a way that all strongly connected components in G are reduced to single vertices. Applying topological sort on Causal Graph [49] brings us an abstraction hierarchy which can help us to decompose planning problems into easier ones. Causal Graph was an inspiration for developing heuristics presented in [38]. Work [35] discuss (time) complexity for problems having simple Causal Graph.

Both the Landmarks and the Causal Graph are techniques for domains or problems analysis. These techniques were used for developing heuristics that some modern planners use (for example, LAMA). Opposite to these techniques, we provided the framework for plan analysis that can be used for the analysis of the training plans.

Another kind of domain analysis rests in detecting of unnecessary actions. FF planner [42] generates only reachable actions (i.e. actions that can be applicable at some point of the planning process). In work [39] they go further, because they try to achieve goals of the planning problem consecutively focusing on actions that might be relevant for the particular goals. Opposite to these works, which eliminate only unreachable actions, we are trying to eliminate actions that are normally reachable, but unnecessary for the planning process (see Chapter 6). It means that we can use our method together with these techniques (modern planners usually include them). However, our method does not ensure that reformulated problems are solvable as the original ones, but experiments (Section 6.3) showed that it happens only occasionally.

3.3 Macro-operators and macro-actions

Macro-operators or macro-actions [50] behave like normal planning operators or actions but they consist of sequences of primitive operators or actions. Reasons, why macro-operators (or macro-actions) are used in planning process, are quite straightforward - reduction of the depth of the search space, however, for the cost of increase of the branching factor.

```
(:action unstack-putdown
    :parameters (?h - hand ?x - block ?y - block)
    :precondition (and (clear ?y)(on ?y ?x)(empty ?h))
    :effect (and (clear ?x)(clear ?y)(ontable ?y)(empty ?h)
            (not (on ?y ?x))(not (holding ?y)) )
  )
```

Figure 3.1: Example of a macro-operator

Recalling the BlocksWorld domain, we can define macro-operator UNSTACK-PUTDOWN (see figure 3.1) which is made by assembling UNSTACK and PUTDOWN operators (in order). Macro-operators in general can be assembled from two or more (primitive) operators. Formally, operators o_i and o_j can be assembled into a macro-operator $o_{i,j}$ in the following way:

- $p(o_{i,j}) = (p(o_i) \cup p(o_j)) \setminus e^+(o_i)$
- $e^-(o_{i,j}) = (e^-(o_i) \cup e^-(o_j)) \setminus e^+(o_j)$
- $e^+(o_{i,j}) = (e^+(o_i) \cup e^+(o_j)) \setminus e^-(o_j)$

It can be easily seen that this approach can be generalized for more operators by successive assembling of pairs of (macro-)operators. For example, if we would like to assemble operators o_i, o_j and o_k we assemble o_i and o_j into $o_{i,j}$ and then $o_{i,j}$ and o_k into $o_{i,j,k}$. Assembling operators into macro-operators is more deeply discussed in [20].

The idea of macro-operators (or macro-actions) is not very new. One of the oldest approaches, STRIPS [29], generates macro-actions from all subsequences of plans. This approach, however, generates many useless macro-actions. REFLECT [25] builds macro-operators from pairs of primitive operators that are applied successively and share at least one argument. In this case, macro-operators are learnt directly from the domain analysis. However, it may also lead to a generation of useless macro-operators. FM [55] follows the ideas used by STRIPS, but unlike STRIPS, FM compiles learnt sequences of operators into one single operator representing the whole sequence of primitive ones. In addition, FM learns additional knowledge that help it with instantiating of macro-actions. Even though FM gained a significant improvement against STRIPS, still it produces many useless and too

complex macro-operators. MORRIS [58] 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 [50], learns macro-actions only for particular goals. It needs different macro-actions when the problem instances scale or goals are different. MACLEARN [45] generates macro-actions that can 'traverse' from one peak of a particular heuristic function to another peak.

One of the state-of-the-art approaches, MARVIN [21, 22] learns macro-operators online from actions sequences that help FF-based planners to escape plateaus. It also learns the macro-operators from the plans of the reduced versions of the given problems. One of the most outstanding works in the area of macro-actions is Macro-FF [11], 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 - designed for arbitrary planners without changing their code and SOL-EP - planner (in this case FF) is enchanted for handling macro-operators. WIZARD [63] learns macro-actions from training plans by genetic algorithms. There are defined several genetic operators working over action sequences appearing in the training plans. WIZARD is designed for arbitrary planners. DHG [4, 5] is able to learn the macro-operators from the static domain analysis by exploring of the graph of dependencies between the operators. Even though these systems gathered good results there remains a space for improvements.

Our method for generation of the macro-operators (see Chapter 5) is designed for domain-independent planning and for arbitrary planners like the other systems (for, example WIZARD). The macro-operators can be built only from dependent operators, in term that one operator provide the predicate (or predicates) to the other one. It is similar to the 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 the macro-actions but the actions do not have to be necessarily successive in the 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. So, it is not necessary to run the planners again. It can help us with another issue, removing of the unnecessary primitive operators that can be replaced by the generated macro-operators. Despite the potential loss of completeness of some planning problems, the planners benefit from the removal of the primitive operators and the experiments we made on IPC

domains did not reveal any problem that became unsolvable. More thorough comparison of our method with the existing ones is done in Subsection 5.4.4 (last paragraph).

Chapter 4

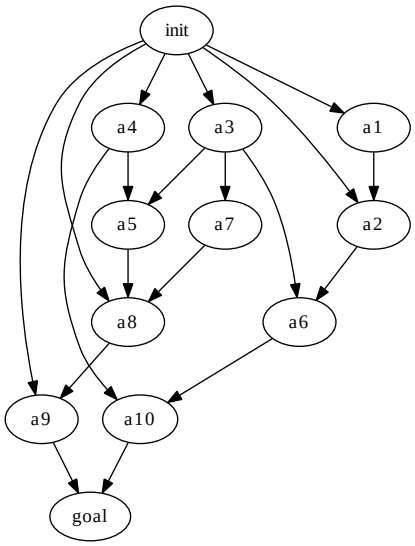
Action dependencies in plans

This chapter is devoted to a description of action dependencies or independencies in plans. Likewise the other approaches (like Causal Graph [49]) did, we focused on developing a theoretical framework as a supporting tool for learning methods (in our case, macro-operators - see Chapter 5). In addition, we provided a brief theoretical study regarding using of action dependencies for plan optimizations.

4.1 Basic aspects of action dependencies

(Sequential) planning is about looking for valid sequences of actions (plans) which solve the planning problems. The actions usually need certain predicates to be true before the action can be performed. These predicates can be provided by the initial state or by the other actions that were performed before. There are several existing techniques for the domain analysis, for example, previously mentioned Causal Graph [49] or analysis of action dependencies made in DHG [5]. Our approach studied in [18] is focused on analyzing action dependencies (and independencies) in plans. If we have a plan solving a planning problem, we can identify which actions are providing these predicates to other actions. The following definitions describes such relations formally.

Definition 4.1: Let $\langle a_1, \ldots, a_n \rangle$ be an ordered sequence of actions. Action a_j is *straightly dependent on 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 $(e^+(a_i) \cap p(a_j)) \not\subseteq \bigcup_{t=i+1}^{j-1} e^+(a_t)$.



a1: LIFT(hoist0,crate1,pallet0,depot0)
a2: LOAD(hoist0,crate1,truck1,depot0)
a3: DRIVE(truck1,depot0,distributor0)
a4: LIFT(hoist1,crate0,pallet1,distributor0)
a5: LOAD(hoist1,crate0,truck1,distributor0)
a6: UNLOAD(hoist1,crate1,truck1,distributor0)
a7: DRIVE(truck1,distributor0,distributor1)
a8: UNLOAD(hoist2,crate0,truck1,distributor1)
a9: DROP(hoist2,crate0,pallet2,distributor1)
a10: DROP(hoist1,crate1,pallet1,distributor0)

Figure 4.1: Example of the relation of straight dependency in Depots domain, problem depotprob1818

Definition 4.2: Action a_j is *dependent on action* a_i if and only if $a_i \rightarrow^* a_j$ where $\rightarrow^*$ is a transitive closure of the relation of straight dependency ($\rightarrow$).

Definition 4.3: Let $\langle a_1, \ldots, a_n \rangle$ be an ordered sequence of actions. Let $E(a_i, a_j)$ be a set of predicates defined in the following way:

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

Remark 4.4: Negation of the relations of straight dependency and dependency is denoted in the following way:

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

Clearly, it holds that if $a_i \rightarrow a_j$, then a_i is the last action in the sequence that provides a predicate (or predicates) essential for the action a_j. This predicate (or predicates) is stored in $E(a_i, a_j)$. Notice that an action may

be straightly dependent on more actions (if it has more predicates in the precondition). We can extend the definitions by describing that an action is straightly dependent on an initial state or that a goal is straightly dependent on some actions. However, to model these dependencies we have to use two special actions: $a_0 = (\emptyset, \emptyset, s_0)$ (s_0 represents the initial state) and $a_{n+1} = (g, \emptyset, \emptyset)$ (g represents a set of goal predicates). Action a_0 is performed before the plan and action a_{n+1} is performed after the plan. To get more familiar, see figure 4.1.

Let us now define the notion of action independency which not necessarily complementary to action dependency. The motivation behind this notion is that two independent actions can be swapped in the action sequence without influencing the plan.

Definition 4.5: Let $\langle a_1, \ldots, a_n \rangle$ be an ordered sequence of actions. Action a_j is *independent on action* a_i (denoted as $a_i \nleftrightarrow a_j$) (we assume that $i < j$) if and only if $a_i \nrightarrow^* a_j$, $p(a_i) \cap e^-(a_j) = \emptyset$ and $e^+(a_j) \cap e^-(a_i) = \emptyset$.

Remark 4.6 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 (ie. $\neg(a_i \nleftrightarrow a_j)$).

The symbol for relation of action independency evokes a symmetrical relation even though according to Definition 4.5 the relation is not necessarily symmetrical. The reason for using the symmetrical symbol is hidden in the property of the independency relation mentioned bellow (lemma 4.7) [18]. This property can be used to modify the plans without losing their validity.

Lemma 4.7: 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 $P = (\Sigma, s_0, g)$ 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.

Proof: Assume that π solves planning problem P. Let s_j (respectively s'_j) be a state obtained by performing the first j actions from π (respectively π'). It is clear that $s_k = s'_k$ for $k \leq i-1$. From the assumption we know that π is valid which means that s_n is a goal state. Now, we must prove that s'_n is also a goal state. From Definition 4.5 we know that $e^+(a_i) \cap p(a_{i+1}) = \emptyset$ (otherwise a_{i+1} must be dependent on a_i which breaks one of the independency conditions) which means that action a_{i+1} can be performed on state s'_{i-1} which

results in state $s'_i = (s'_{i-1} \setminus e^-(a_{i+1})) \cup e^+(a_{i+1})$. From definition 4.5 we also know that $e^-(a_{i+1}) \cap p(a_i) = \emptyset$ which means that a_i can be performed on state s'_i which results in state $s'_{i+1} = (((s'_{i-1} \setminus e^-(a_{i+1})) \cup e^+(a_{i+1})) \setminus e^-(a_i)) \cup e^+(a_i)$. Finally, we know that $e^+(a_{i+1}) \cap e^-(a_i) = \emptyset$ which implies in $s'_{i+1} \supseteq s_{i+1}$ ($s_{i+1} = (((s_{i-1} \setminus e^-(a_i)) \cup e^+(a_i)) \setminus e^-(a_{i+1})) \cup e^+(a_{i+1}))$. It is clear that after performing the remaining actions from π' (beginning from the $(i+2)$-nd action) on state s'_{i+1} we obtain state $s'_n \supseteq s_n$ which is also the goal state. □

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

Corollary 4.8: Let $\pi = \langle a_1, \ldots, a_i, \ldots, a_{i+k}, \ldots, a_n \rangle$ be a plan solving planning problem P and $a_{i+l} \nleftrightarrow a_{i+m}$ for every $0 \leq l < m \leq k$. Let λ be a permutation over sequence $\langle 0, \ldots, k \rangle$. Then plan $\pi' = \langle a_1, \ldots, a_{i+\lambda(0)}, \ldots, a_{i+\lambda(k)}, \ldots, a_n \rangle$ also solves planning problem P.

Proof: The proof can be done in the following way. Action $a_{i+\lambda(0)}$ is shifted to position i by repeated application of lemma 4.7. The relation $a_{i+l} \nleftrightarrow a_{i+m}$ remains valid during these shifts so we can then shift $a_{i+\lambda(1)}$ to position $i+1$ etc. □

4.2 Auxiliary algorithms

This section discusses how the relations of action dependencies or independencies can be computed from given training plans. The idea of algorithms for the computation of these relations is quite straightforward. First, we have to compute the relation of straight dependency (definition 4.1) from a given training planning problem and a plan solving it. In addition, we compute also the set of predicates E (definition 4.3). The algorithm (figure 4.2) has a quite straightforward behavior. Each predicate p is annotated by $d(p)$ which refers to the last action that created it. We simulate the execution of the plan and each time action a_i is executed, we find the dependent actions by exploring $d(p)$ for all predicates p from the precondition of a_i.

Procedure COMPUTE-STRAIGHT-DEPENDENCY(`IN` planning problem $P = (\Sigma, s_0, g)$, `IN` plan $\pi = \langle a_1, \ldots, a_n \rangle$, `OUT` straight dependency relation SD, `OUT` set of predicates E)

1: $SD := \emptyset$;
2: **ForEach** $p \in s_0$ **do** $d(p) := 0$;
3: **For** $i := 1$ **to** $|\pi|$ **do**
4: **ForEach** $j \in \{d(p) \mid p \in p(a_i)\}$ **do**
5: $SD := SD \cup (a_j, a_i)$;
6: $E(a_j, a_i) := \{p \mid d(p) = j \wedge p \in p(a_i)\}$;
7: **EndForeach**
8: **ForEach** $p \in e^+(a_i)$ **do** $d(p) := i$;
9: **EndFor**
10: **ForEach** $j \in \{d(p) \mid p \in g\}$ **do**
11: $SD := SD \cup (a_j, a_{n+1})$;
12: $E(a_j, a_{n+1}) := \{p \mid d(p) = j \wedge p \in g\}$;
13: **EndForeach**

Figure 4.2: Algorithm for computation of the relation of straight dependency

Theorem 4.9: Algorithm COMPUTE-STRAIGHT-DEPENDENCY is sound.

Proof: We must show that for every planning problem $P = (\Sigma, s_0, g)$ and plan $\pi = \langle a_1, \ldots, a_n \rangle$ solving P the algorithm COMPUTE-STRAIGHT-DEPENDENCY computes the straight dependency relation SD ($(a_j, a_i) \in SD$ iff $a_j \rightarrow a_i$) and the set of predicates E ($E(a_j, a_i) = (e^+(a_j) \cap p(a_i)) \setminus \bigcup_{t=j+1}^{i-1} e^+(a_t)$ iff $a_j \rightarrow a_i$ (Def. 4.3)). From lines 2 and 8 it is clear that $d(p)$ always refers to an action which created p (actions in the cycle (lines 3-9) are processed incrementally, i.e., from a_1 to a_n). i-th step of the cycle (lines 4-7) identifies all actions a_j satisfying $a_j \rightarrow a_i$ (line 4) and computes $E(a_j, a_i)$ (line 6). At first, let us show that all conditions from Definition 4.1 ($j < i$, $(e^+(a_j) \cap p(a_i)) \neq \emptyset$ and $(e^+(a_j) \cap p(a_i)) \not\subseteq \bigcup_{t=j+1}^{i-1} e^+(a_t)$) are met. It is clear that $j < i$ (actions are processed incrementally). We already showed that if $d(p) = j$, then $p \in e^+(a_j)$ and there is no action a_t, $t \in \langle j+1, i-1 \rangle$ such that $p \in e^+(a_t)$. Together with $j \in \{d(p) \mid p \in p(a_i)\}$ (line 4) we can see that the remaining conditions are also met. Similarly,

$E(a_j, a_i) := \{p \mid d(p) = j \wedge p \in p(a_i)\}$; (line 6) corresponds with the conditions in Definition 4.3. The soundness of the cycle (lines 10-13) considering special action a_{n+1} can be proved analogically. □

Theorem 4.10: Algorithm COMPUTE-STRAIGHT-DEPENDENCY runs at most in $O(n^2)$ steps, where $n = |\pi|$.

Proof: It is clear that the cycle (lines 3-9) is performed just n times. The cycle (lines 4-7 or 10-13) is performed at most n times, because every action can be straightly dependent only on actions performed before it. Computation of the set $\{d(p) \mid p \in p(a_i)\}$ (line 4 or 10) can be done in $O(|p(a_i)|)$ (similarly on the 6-th and 12-th line), where the set of all grounded predicates referring to the given planning problem is extensionally defined (it costs the constant time to access the particular predicate). If we assume that the actions' preconditions or positive effects contain only the small number of predicates (it holds for larger n), then the algorithm COMPUTE-STRAIGHT-DEPENDENCY runs in $O(n^2)$ steps. □

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 (acyclic) graph, for example using the algorithm presented in [57]. The algorithm runs in the worst case in $O(n^3)$ steps.

Independency relation can be computed by testing all pairs of actions, whether they satisfy the conditions from Definition 4.5. It can be easily seen that such an approach can be done in $O(n^2)$ steps.

4.3 Decomposition of planning problems

Decomposition of planning problems into smaller or easier subproblems seems to be a very promising technique. Some sort of decomposition has been used in STRIPS planning method [29], where goals were satisfied successively. From state-of-the art planners, for instance SGPLAN [44] uses some decomposition techniques. In this section, we discuss possibilities of decomposition of planning problems from theoretical point of view with respect to action dependencies in plans [18].

The following definition formally follows an intuitive idea what a subplan should stand for.

Definition 4.11: Let π be a plan solving planning problem $P = (\Sigma, s_0, g)$. *Subplan* π_i is a subsequence of π (not necessarily continuous) which solves some planning problem $P_i = (\Sigma_i, s_{0_i}, g_i)$, where $\Sigma_i \subseteq \Sigma$.

The following lemma describes how plans can be decomposed into subplans and what relationship is between subplans and planning subproblems.

Lemma 4.12: Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan solving planning problem $P = (\Sigma, s_0, g)$, $a_0 = (\emptyset, \emptyset, s_0)$ and $a_{n+1} = (g, \emptyset, \emptyset)$. Let $\pi_i = \langle a_i, \ldots, a_{i+m} \rangle$ be a subsequence of π. π_i is a subplan which solves a planning subproblem $P_i = \Big(\Sigma, \bigcup_{r \in \{0,\ldots,i-1\}, u \in \{0,\ldots,m\}} E(a_r, a_{i+u}),$
$\bigcup_{s \in \{m+1,\ldots,n+1\}, v \in \{0,\ldots,m\}} E(a_{i+v}, a_s)\Big)$.

Proof: It is clear that both π and π_i are plans whose actions belong to the planning domain Σ. The predicates needed for the execution of the subplan π_i can be provided only by the actions performed before the actions from π_i (including the special action a_0). These predicates represent the initial state of subproblem P_i. Analogically we know that predicates obtained by execution of subplan π_i are needed for the other actions from π (including the special action a_{n+1}) that are straightly dependent on them. These predicates represent the goal of subproblem P_i. □

Decomposition of a planning problem into subproblems described in lemma 4.12 is made from plans. It means that we have to solve an original problem before it can be decomposed. It may be accommodated with some learning techniques that can detect some relationships between the original problems and its subproblems. Even though we allow picking only subsequences from the original plans, according to lemma 4.7 the actions in the original plans can be moved. One example of such a decomposition are macro-operators. This technique is studied in Chapter 5. Another example of a 'special' kind of planning subproblems is such a problem whose solution contains actions that are independent on themselves (due to Corollary 4.8 the actions can be performed in any order). This idea has been studied in [18] and more deeply in [17], formally:

Definition 4.13: Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan solving planning prob-

lem $P = (\Sigma, s_0, g)$, $a_0 = (\emptyset, \emptyset, s_0)$ and $a_{n+1} = (g, \emptyset, \emptyset)$. Let $level : A \rightarrow N_0$ be a function such that:

1. $level(a_0) = 0$
2. For each $j < i$, where $a_j \leftrightarrow a_i$ $level(a_j) < level(a_i)$ holds.
3. For each action a $level(a)$ is as small as possible.

Definition 4.14: Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan solving planning problem $P = (\Sigma, s_0, g)$, $a_0 = (\emptyset, \emptyset, s_0)$ and $a_{n+1} = (g, \emptyset, \emptyset)$. Let a sequence of actions with the same value of *level* function be a *layer*. Let sl_k be an *after-layer state* defined as $sl_k = \bigcup E(a_i, a_j)$, for every i, j where $level(a_i) \leq k$ and $level(a_j) > k$.

Layering, introduced in the definitions above, decomposes plans into layers, where every layer contains actions that are all independent on themselves (following the point 2 in Def. 4.13), i.e., they can be performed in any order. After-layer states are such states that are gained after performance of actions in first k layers. Even though we present here only the definition of layering, it may be useful in connection with planning techniques based on the Planning Graph. Even though we did not prove it, we believe that by layering, we will be able to transform 'classical' sequential plans (i.e., makespan of the 'classical' sequential plans is equal to the number of actions in these plans) to plans considering parallel actions, usually provided by Planning Graph based planners, for example, SATPLAN.

4.4 Plans optimization

Investigation of action dependencies can be also useful in plans optimization [15]. State-of-the-art planners, however, usually do not provide optimal plans and often the optimality is very low. Complexity results [37, 40] showed that finding optimal plans can be much more harder than any plans. Nevertheless, we can provide possible approaches that remove unnecessary actions from plans making them shorter but of course not optimal at least. However, provided approaches is currently only in theoretical shape, but we believe that the approaches will be useful, especially with the other methods (see Chapter 5 and 6) presented in this thesis.

One kind of unnecessary actions are actions that are not required to acquire the goal. In terms of actions dependencies, we mean such actions on which the special goal action is not dependent.

Proposition 4.15: Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan solving planning problem $P = (\Sigma, s_0, g)$, $a_0 = (\emptyset, \emptyset, s_0)$ and $a_{n+1} = (g, \emptyset, \emptyset)$. Let $A = \{a_i \mid 0 < i \leq n, a_i \nrightarrow^* a_{n+1}\}$ be a set of actions. If all actions from A are removed from π then the new plan is valid and solves problem P.

Proof: Let $A^0 = \{a_i \mid 0 < i \leq n, \forall j : a_j \in \pi \vee a_j = a_{n+1},\ a_i \nrightarrow a_j\}$ be a set of actions, where no other action is straightly dependent on them. According to Definition 4.1, we know that there is no action from π which requires predicates provided by actions from A^0. It means that actions from A^0 can be removed from π without losing its validity. Let $A^1 = \{a_i \mid 0 < i \leq n, \forall j : a_j \in \pi \setminus A^0 \vee a_j = a_{n+1}, a_i \nrightarrow a_j\}$ be a set of actions, where no other action (except actions from A^0) is straightly dependent on them. We showed before that actions from A^0 are unnecessary and can be removed without losing plan validity. We also know that there is no action from π (except those from A^0) which requires predicates provided by actions from A^1. It means that actions from A^1 can be also removed from π without losing its validity. We can construct sets $A^2, \ldots A^n$ analogically (it is clear that $A^k = \emptyset$ for $k > n$) and show that actions in these sets are unnecessary and can be removed from π without losing its validity. It can be easily observed from the acyclicity of the relation of straight dependency that $A = A^0 \cup A^1 \cup \ldots \cup A^n$. □

The successive application of *inverse actions* on some state results in the same state as before the application. Planning domains usually contains the inverse actions (for example, actions PICKUP(A) and PUTDOWN(A)). By observation of straight dependencies between inverse actions, we are able to detect useless couples of inverse actions and remove them from plans without losing their validity.

Proposition 4.16: Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan solving planning problem $P = (\Sigma, s_0, g)$, $a_0 = (\emptyset, \emptyset, s_0)$ and $a_{n+1} = (g, \emptyset, \emptyset)$. Let $\{i_1, \ldots, i_k\}$ be a set of indices such that $0 < i_1 < \ldots < i_k \leq n$. Let $s = \gamma(s_0, \langle a_1, \ldots a_{i_1 - 1} \rangle)$. If $\gamma(s, \langle a_{i_1}, a_{i_2}, \ldots, a_{i_k} \rangle) = s$ and $\forall j, l : 0 \leq j < k,\ l \neq i_{j+1} : a_{i_j} \rightarrow a_{i_{j+1}} \wedge a_{i_j} \nrightarrow a_l$, then actions $a_{i_1}, \ldots, a_{i_k}$ can be removed from π without losing its validity.

Proof: It is clear from the assumption that the successive application of actions $a_{i_1}, \ldots, a_{i_k}$ leads to the same state as before the application. Indices $i_1, \ldots i_k$ are not necessarily successive, but together with the condition that action a_{i_2} is straightly dependent only on a_{i_1}, action a_{i_3} is straightly dependent only on a_{i_2} etc. we can see that predicates provided by actions $a_{i_1}, \ldots, a_{i_{k-1}}$ are not required by any other action from π (except $a_{i_1}, \ldots, a_{i_k}$). Predicates provided by action a_{i_k} may be required by other actions from π, but from the assumption we know that all these predicates are presented in the state before the application of action a_{i_1}. □

Approach described in Proposition 4.16 can be applied repeatedly until the assumptions hold. In this case, we are able to remove only the unnecessary actions (satisfying the assumptions from Propositions 4.15 and 4.16). As mentioned before this approach does not provide optimal plans in all cases, because it does not take into the account certain situations, where some longer subsequence of actions can be replaced by a shorter one. For example, sequence $DRIVE(t1, l1, l2), DRIVE(t1, l2, l3)$ means that truck $t1$ goes from $l1$ to $l3$ via $l2$. If we do not need $t1$ in $l2$ and there is a direct road from $l1$ to $l3$ we may replace the sequence by a single action $DRIVE(t1, l1, l3)$. However, for larger sequences it becomes a hard problem.

Chapter 5

Learning Macro-operators by plans investigation

This chapter is devoted to a method for learning macro-operators. The method is based upon investigation of action dependencies and independencies in the training plans [16]. Knowledge learned by the method is domain dependent, but the method itself is domain independent. The method differs from the existing ones (for example MacroFF [11]) by the fact, that the method is able to detect even pairs of actions for assemblage that are not adjacent in the training plans and, in addition, the method detects unnecessary primitive operators that can be replaced by generated macro-operators.

5.1 Identifying actions for assemblage

As discussed in Section 3.3., a macro-action can be obtained by assembling a sequence of two or more actions. Macro-actions can be encoded like the 'normal' actions which may be useful for the planning systems. Similarly, macro-operators can be obtained by assembling a sequence of two or more planning operators.

Now, the task is how the sequences of actions suitable for possible assemblage to macro-actions can be identified. We have a planning domain, several (simpler) planning problems and the plans (sequences of actions) solving them. We can analyze these (training) plans, where we focus on the actions that are or can be successive. If we analyze the training plans only by looking for the successive actions, then we can miss many pairs of non-

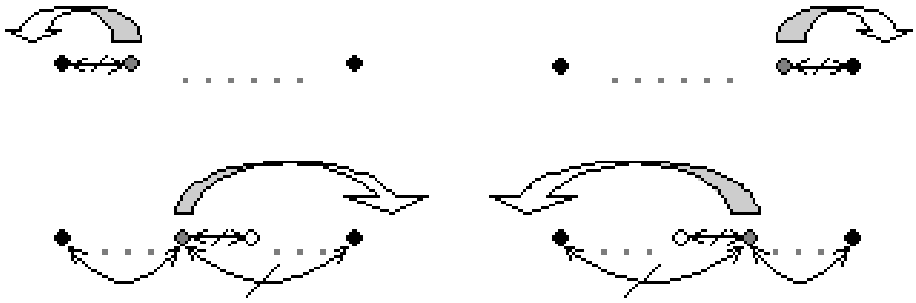


Figure 5.1: Four different situations for moving the intermediate actions (grey-filled) before or behind one of the boundary actions (black-filled).

successive actions that may be successive in 'special' circumstances. By the 'special' circumstances we mean that the intermediate actions can be moved without the loss of plans' validity before or behind the actions to make them successive. If we recall the main property of the independency relation (definition 4.5), then we can found out how the intermediate actions can be moved. Figure 5.1 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 it (according to lemma 1.4). If the intermediate action is not independent on one of the boundary actions, then we can 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).

Algorithm (fig. 5.2) is based on the 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 (became 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 obtain also the lists of action indices representing the (intermediate) actions that must be moved before (respectively behind) actions a_i and a_j. The usage of these lists will be explained in the following section.

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

1: $D := \{k | i < k < j\}$
2: $L := R := \emptyset$
3: **Repeat**
4: $chg := false$
5: $k := min(D)$ **or** 0 **if** $D = \emptyset$
6: **If** $k > 0$ and $(i, k) \in S$ **then**
7: $D := D \setminus \{k\}$
8: $chg := true$
9: $L := L \cup \{k\}$
10: **EndIf**
11: $k := max(D)$ **or** 0 **if** $D = \emptyset$
12: **If** $k > 0$ **and** $(k, j) \in S$ **then**
13: $D := D \setminus \{k\}$
14: $chg := true$
15: $R := R \cup \{k\}$
16: **EndIf**
17: $Z := \{x | x \in D \wedge (i, x) \notin S\}$
18: $k := max(Z)$
19: **If** $k > 0$,$(k, j) \in S$ **and ForEach** $l \in D \wedge l > k$ $(k, l) \in S$ **holds then**
20: $D := D \setminus \{k\}$
21: $chg := true$
22: $R := R \cup \{k\}$
23: **EndIf**
24: $Z := \{x | x \in D \wedge (x, j) \notin S\}$
25: $k := min(Z)$
26: **If** $k > 0$,$(i, k) \in S$ **and ForEach** $l \in D \wedge l < k$ $(l, k) \in S$ **holds then**
27: $D := D \setminus \{k\}$
28: $chg := true$
29: $L := L \cup \{k\}$
30: **EndIf**
31: **Until** not chg
32: **If** $D = \emptyset$ **then Return** true **else Return** false

Figure 5.2: Algorithm for detecting pairs of actions that can be assembled.

5.2 Generation of macro-operators

If we consider the classical representation, we know that the planning domains include planning operators rather than 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 detecting such operators, which can be assembled into macro-operators, is based on the investigation of training plans, where we explore pairs of actions (instances of operators) that can be assembled on more places.

Definition 5.1 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. 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*.

Informally said, the matrix of candidates describes how many pairs of particular instances of operators are presented in training plans. In addition, the matrix contains the minimal sets of arguments that the particular pairs of operators are sharing. To get more familiar, see the example in figure 5.3.

The algorithm (fig. 5.4) constructs the matrix of candidates from the given set of training plans solving the planning problems in the same domain (i.e, this approach is domain-independent, but generates domain-dependent knowledge). It is quite straightforward how the numbers N in the matrix of candidates are computed. The computation of the sets of variables that operators share should be clarified. For instance, if we recall our example of BlocksWorld domain, we know that there are operators UNSTACK(box,surface) and STACK(box,surface). If we decide to make a macro-operator UNSTACK-STACK (consisting of UNSTACK and STACK operators in this order), then we can also see that box is always the same (we are unstacking and stacking the same box in time), only the surface may

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

Figure 5.3: Matrix of candidates built from two training plans (depotprob1818 and depotprob7512) generated by SGPLAN

differ. Generally, we observe which parameters (objects) are shared by actions and we select such parameters that are shared by all pairs of actions (instances of the given operators) that can be assembled.

Now, we have to explain the purpose of lists L and R that are generated in function DETECT-IF-CAN-ASSEMBLE (fig. 5.2). If we have to update the plans by replacing the selected actions by the macro-actions (instances of the generated macro-operators), then we must also reorder other actions to keep the (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 $a_{i,j}$ (an example can be seen in fig. 5.5):

- actions $a_1, \ldots a_{i-1}$ remain in their positions
- actions listed in L are moved (in order) to positions $i, \ldots, i + |L| - 1$
- 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 $j, \ldots, n-1$)

Procedure CREATE-MATRIX(`IN` set of plans P, `OUT` matrix M)

1: **Set** M as empty square matrix
2: **ForEach** π **in** P **do**
3: **Compute** D as a relation of straight dependency on actions from π
4: **Compute** S as a relation of independency on actions from π
5: **ForEach** $(i, j) \in D$ **do**
6: **If** DETECT-IF-CAN-ASSEMBLE(i, j, S, L, R) **then**
7: **Set** k as the id of the operator whose a_i is an instance
8: **Set** l as the id of the operator whose a_j is an instance
9: **Compute** V as a set of arguments that a_i and a_j share
10: **If** $M_{k,l}$ is empty **then**
11: $M_{k,l} := \langle 1, V \rangle$
12: **Else**
13: $\langle N, OV \rangle := M_{l,k}$
14: **If** a_i resp. a_j were not yet selected as a candidate with l-th operator resp. k-th operator **then** $N1 := N + 1$ **else** $N1 := N$
15: $M_{l,k} := \langle N1, OV \cap V \rangle$
16: **EndIf**
17: **EndIf**
18: **EndForeach**
19: **EndForeach**

Figure 5.4: Algorithm for creating the matrix of candidates.

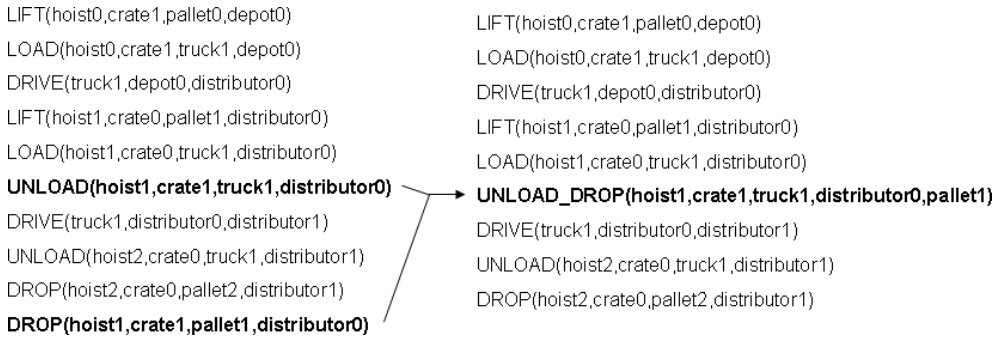


Figure 5.5: Example showing how the training plan is updated when the given pair of action is assembled into the macro-action

Procedure GENERATE-MACRO(`IN` set of plans P, `OUT` set of macro-operators O)

1: $O := \emptyset$
2: **Repeat**
3: $picked := false$
4: CREATE-MATRIX(P,M)
5: **If** SELECT-CANDIDATE(M,C) **then**
6: $picked := true$
7: ASSIGN-INEQUALITY-CONSTRAINTS(C)
8: $O := O \cup \{C\}$
9: UPDATE-PLANS(P,C)
10: **EndIf**
11: **Until** not *picked*

Figure 5.6: Algorithm for generating macro-operators.

```
(: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 5.7: Example of PICKUP-STACK macro-operator.

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

Figure 5.8: Example of PICKUP-STACK macro-operator, where the arguments were set as equal.

To generate the macro-operators from the training plans (in the given domain) we can use the following approach (formally in fig. 5.6). The macro-operators are generated repeatedly until no other macro-operator can be generated. 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 the macro-operators (SELECT-CANDIDATE) which means that such a candidate must satisfy certain conditions (it will be explained later). To ensure the soundness of the generated macro-operators we have to assign inequality constraints for the macro-operators arguments. It prevents a possible instantiation of invalid macro-actions if these arguments are set as equal. In figure 5.7 we can see an example of PICKUP-STACK macro-operator. If the arguments are set as equal (fig. 5.8) we can simply see that such an instance is applicable, but invalid (when unfolded). Inequality constraints can be easily detected in the following way:

- Instantiate the given macro-operator in such a way that just two arguments have the same value
- Define a planning problem, where the preconditions of the instantiated macro-operator represent the initial state and the positive effects of the instantiated macro-operator represent the goal.
- Unfold the instantiated macro-operator into primitive actions
- Apply the primitive actions on the initial state of the problem
- If we successfully reach the goal, then these arguments may be equal, otherwise the inequality constraint for these arguments must be defined
- This approach must be applied for every pair of macro-operator's arguments (considering the types, we have to test only such pairs that have the same type)

After the macro-operator is generated 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 selection of the proper candidates for becoming macro-operators (SELECT-CANDIDATE). We suggested to select 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 number N in field $M_{i,j}$ of the matrix of candidates and $V_{i,j}$ represent the set of variables shared by the i-th and j-th operators):

$$max\left(\frac{N_{i,j}}{f(o_i)}, \frac{N_{i,j}}{f(o_j)}\right) \geq b \tag{5.1}$$

$$\frac{N_{i,j}}{\sum_k f(o_k)} \geq c \tag{5.2}$$

$$a(o_i) + a(o_j) - \mid V_{i,j} \mid \leq d \tag{5.3}$$

Condition 5.1 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 which prevents selecting such operators whose instances do not appear successively so often. It is clear that if we set the bound 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 the 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 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 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 5.2 allowing only 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 predefined constant c.

Another problem, we are faced with, 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, the macro-operators

usually have more parameters than the 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 the maximum number of parameters for each macro-operator by a predefined constant d (condition 5.3). If there are more candidates satisfying all the conditions then we prefer the candidate with the maximum value of the expression listed in condition 5.1.

We must also decide which macro-operators will be added to the domain and which primitive operators will 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 other (non-training) problems. Fortunately, in IPC benchmarks, we used for experiments, it usually does not happen (we did not experience any such a 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.

5.3 Soundness and Complexity

Our method assumes that all inputs (especially the training plans) are valid. Training plans are generated by planners participating on IPCs, so we do not assume that the planners are producing invalid plans. Soundness and complexity of the algorithms for constructing the straight dependency and independency relations have been discussed in Section 4.2. The algorithms listed in this Chapter are discussed in the following paragraphs.

Proposition 5.2: Algorithm DETECT-IF-CAN-ASSEMBLE (fig. 5.2) 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 the intermediate actions before or behind the 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 (fig. 5.1), where two of them are 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 fig. 5.1), because the soundness and complexity of the other ones can be proved analogically. First (lines 5-10), if $a_i \nleftrightarrow a_{i+1}$, then by applying of lemma 4.7, we can move a_{i+1} before a_i without loss of plan's validity and it takes a constant time (i.e., $O(1)$). Second (lines 17-23), assume that $a_i \leftrightarrow a_k$, $k < j$ and k is the largest possible value. If $\forall l : k < l < j \; a_k \nleftrightarrow a_l$, then by repetitive applying of lemma 4.7, we can move a_k behind a_j also without loss of plan's validity. It can take at most $O(l)$ steps. Analogically for the situations (lines 11-16 and 24-30). The algorithm always terminates because in each run of the loop (lines 3-31) 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. So, in the worst case the algorithm requires $O(l^2)$ steps to perform. □

Remark 5.3: In the last proposition (5.2), we proved the soundness of DETECT-IF-CAN-ASSEMBLE function (i.e., if this function returns true, then the actions can be become neighbors and hence assembled). We did not manage to prove whether this function is able to detect all the possible pairs of action for assemblage.

Proposition 5.4: Algorithm CREATE-MATRIX (fig. 5.4) 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: At first the algorithm computes the relations of straight dependency and independency (fig. 4.2), which takes at most $O(n^3)$ steps (see Section 4.2). For each training plan the algorithm explores each pair of actions being in the relation of straight dependency (lines 5-18) by algorithm DETECT-IF-CAN-ASSEMBLE (fig. 5.2), which is sound (proposition 5.2). It can be simply seen that 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 algorithm DETECT-IF-CAN-ASSEMBLE in the worst case can be performed in $O(n^2)$ steps (proposition 5.2 - considering that l is close to n). Summarized, it gives us the time complexity $O(n^4)$ in the worst case. □

Theorem 5.5: Algorithm GENERATE-MACRO (fig. 5.6) is sound and

can be computed in the worst case in $O(n^5)$ steps, where n the total length of all the training plans.

Proof: From the soundness of algorithms DETECT-IF-CAN-ASSEMBLE (proposition 5.2) and CREATE-MATRIX (proposition 5.4) we know that each candidate for assemblage represents a pair of actions that can be assembled without loss of plans' 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 (lines 2-11) the total 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 worst case time complexity $O(n^4)$ of algorithm CREATE-MATRIX (proposition 5.4), it gives us the time complexity $O(n^5)$ in the worst case. Consider that procedures ASSIGN-INEQUALITY-CONSTRAINTS (line 7) and UPDATE-PLANS (line 9) have much less time complexity than the procedure CREATE-MATRIX. □

Remark 5.6: The time complexity estimation of the algorithm GENERATE-MACRO can be lowered to $O(n^4)$. Considering how many times the cycle (lines 2-11) is usually performed, we can easily find out that it is much lower than the total length of the training plans.

Remark 5.7: It is well known that if we add a generated macro-operator into the domain, then the domain remains valid. It means that the completeness of the planning process is not violated. If we decide to remove certain primitive operators from the domain (i.e., such primitive operators that are replaced by the generated macro-operators), then we may lose the completeness of the planning process. On the other hand it never occurred during our experiments. From our observation we found out that the completeness is not usually lost, while the training problems and testing problems have similar styles of the initial states and goals (differs only by the number of objects).

5.4 Experimental results

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 the planning problems, 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 the 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 the 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 [48] and SGPLAN 5.22 [44] both for the generation of the training plans (for the learning phase) and for the comparison of the running times and plans quality. We also used LAMA [65] for the comparison of running times and plans quality (not for the learning phase). The choice of the planners was motivated by superior results that the planners achieved on the (several last) IPCs. Because SATPLAN cannot handle negative preconditions (which are necessary for representation of inequality constraints), we used a tool called ADL2STRIPS[1] which can produce grounded STRIPS domain from ADL domain.

5.4.1 Tested domains

We chose several planning domains for our experiments to ensure that the proposed approach is generally applicable (rather than specific for a particular planning domain). In particular, we used domains well known from the IPC.

Blocks domain is a planning domain from the 2nd IPC. The domain consists of a table, a gripper and cubical blocks. The 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.

Depots is a planning domain from the 3rd IPC. This domain accommodates both blocks and logistics environments. 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

[1] Available on IPC4 website

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.

Zeno Travel is a planning domain from the 3rd IPC. This domain involves transporting people around in planes, using different modes of movement: fast and slow.

Satellite is a planning domain from the 4th IPC. This domain is one of the domains inspired by space-applications. It involves planning and scheduling a collection of observation tasks between multiple satellites, each equipped in slightly different ways.

Rovers is a planning domain from the 3rd IPC. Inspired by planetary rovers problems, this domain requires that a collection of rovers navigate a planet surface, finding samples and communicating them back to a lander.

Gripper is a planning domain from the 1st IPC. In this domain, there is a robot with two grippers. It can carry a ball in each. The goal is to take N balls from one room to another.

Gold Miner is a planning domain from the 6th IPC learning track. A robot is in a mine and has the goal of reaching a location that contains gold. The mine is organized as a grid with each cell either being hard or soft rock. There is a special location where the robot can either pickup an endless supply of bombs or pickup a laser cannon. The laser cannon can shoot through both hard and soft rock, whereas the bomb can only penetrate soft rock. However, the laser cannon will also destroy the gold if used to uncover the gold location. The bomb will not destroy the gold.

5.4.2 Learning phase

As mentioned in the previous text, the generation of macro-operators depends on pre-defined bounds b, c and d (conditions 5.1, 5.2 and 5.3). 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 generate a training plan was (mostly) within tenths of second.[2] Despite the high (worst-case) time complexity $O(n^5)$ (theorem 5.5), the average time taken by one run of our method (GENERATE-MACRO procedure) was within the tenths of second.[3]

[2]performed on XEON 2.4GHz, 1GB RAM, Ubuntu Linux
[3]performed on Core2Duo 2.66GHz, 4GB RAM, Win XP SP2

Problem	SGPLAN			SATPLAN		
	b	c	d	b	c	d
Blocks	0.8	0.05	3	0.8	0.05	3
Depots	0.8	0.1	5	0.8	0.1	5
Zeno travel	0.8	0.1	7	-	-	-
Rovers	0.8	0.1	7	-	-	-
Satellite	-	-	-	0.8	0.1	3
Gripper	0.8	0.03	4	0.6	0.03	6
Gold miner	0.8	0.08	3	0.8	0.08	3

Table 5.1: Settings of the bounds for SGPLAN's and SATPLAN's training plans

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 (see table 5.1). First, bound d was set to $N+1$ (except Satellite domain and Gripper domain for SATPLAN's training plans), where N represents the largest 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 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 Gripper domain it was set to 0.03. The reason of keeping the bound c low $(0.03 - 0.05)$ rested in the fact that in Blocks and Gripped domains, all the primitive operators were replaced by the generated macro-operators. The choice of the planner for the generation of the training plans brought several differences - only in Blocks domain, it resulted in the same result. In 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 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 Satellite domain, when SGPLAN's training plans were used. In 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 Gold Miner domain, the planners preferred different operators which resulted in slightly different results of learning.

Domain	Added macro-operators	Removed primitive operators
Blocks	PICKUP-STACK, UNSTACK-STACK, UNSTACK-PUTDOWN	PICKUP, PUT-DOWN, STACK, UNSTACK
Depots	LIFT-LOAD, UNLOAD-DROP	LIFT, LOAD, UN-LOAD, 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, CAL-IBRATE, SWITCH-OFF
Gold Miner	MOVE-PICKUP-LASER, MOVE-DETONATE-BOMB-MOVE-PICK-GOLD	PICKUP-LASER, PICK-GOLD, DETONATE-BOMB

Table 5.2: Suggestion of our method - the best results for the particular domains

The results of learning (best for the particular domains we achieved) are showed in table 6.1. We stated only such alternatives that provided the best results in the running times and plans quality comparison for the particular domains.

5.4.3 Running times and plans quality comparison

In this evaluation, we used SGPLAN 5.22, an absolute winner of the IPC 5, SATPLAN 2006, a co-winner of the optimal track in the IPC 5, and LAMA, a winner of the IPC 6 suboptimal track. The benchmarks ran on XEON 2.4GHz, 1GB RAM, Ubuntu Linux. The results are presented in tables 5.3, 5.4 and 5.5. We chose such problems (in the 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. The less complex problems were solved in the updated domains almost as fast as or a bit slower than in the original ones (except Rovers domain in SATPLAN's

Problem	Time (in seconds)			Plan length		
	orig	upd-SG	upd-SAT	orig	upd-SG	upd-SAT
Blocks14-0	>600.00	0.03	0.03	N/A	48	48
Blocks14-1	>600.00	0.03	0.03	N/A	44	44
Blocks15-0	>600.00	0.32	0.32	N/A	88	88
Blocks15-1	179.84	0.05	0.05	114	54	54
depots1817	24.56	15.52	20.71	100	104	94
depots4534	>600.00	0.53	54.71	N/A	112	110
depots5656	410.94	0.32	7.70	133	132	82
depots7615	8.48	1.88	2.14	98	102	91
zeno-5-20a	0.88	0.75	-	98	101	-
zeno-5-20b	1.07	0.77	-	92	97	-
zeno-5-25a	1.74	1.05	-	124	122	-
zeno-5-25b	0.57	0.58	-	117	125	-
rovers4621	2.31	0.03	-	48	44	-
rovers5624	0.10	0.02	-	52	52	-
rovers7182	4.32	0.12	-	90	91	-
rovers8327	3.53	0.06	-	78	71	-
gripper16	0.05	0.05	1.11	135	135	101
gripper17	0.06	0.06	1.31	143	143	107
gripper18	0.06	0.07	1.56	151	151	113
gripper19	0.06	0.07	1.83	159	159	119
gripper20	0.07	0.08	2.13	167	167	125
satellite26	3.73	-	29.99	138	-	138
satellite27	4.73	-	13.20	138	-	139
satellite28	12.87	-	260.26	193	-	193
satellite29	18.69	-	70.36	195	-	195
satellite30	31.57	-	117.52	231	-	231
satellite31	56.65	-	201.36	272	-	272
gminer7x7-06	err	0.01	0.01	N/A	33	30
gminer7x7-07	err	0.02	0.01	N/A	34	65
gminer7x7-08	err	0.01	0.01	N/A	25	26
gminer7x7-09	err	0.01	0.01	N/A	29	32
gminer7x7-10	err	0.01	0.02	N/A	33	43

Table 5.3: Comparison of running times and plans lengths (we assume that macro-actions are unfolded into primitive actions) for SGPLAN.

Problem	Time (in seconds)			Plan length		
	orig	upd-SAT	upd-SG	orig	upd-SAT	upd-SG
Blocks14-0	23.58	3.14	3.14	38	56	56
Blocks14-1	38.06	3.84	3.84	36	88	88
Blocks15-0	46.90	7.24	7.24	40	60	60
Blocks15-1	45.68	7.63	7.63	52	142	142
depots4321	5.24	4.40	2.07	43	41	38
depots5656	222.42	>600.00	143.33	70	N/A	59
depots6178	6.82	43.14	26.11	51	50	42
depots7654	10.04	25.96	16.45	41	56	39
depots8715	35.96	46.95	err	50	38	err
zeno-3-10	3.77	-	4.17	31	-	35
zeno-5-10	34.07	-	48.19	42	-	38
zeno-5-15a	92.13	-	30.93	50	-	51
zeno-5-15b	err	-	err	N/A	-	N/A
rovers4621	182.20	-	>600.00	47	-	N/A
rovers5624	4.30	-	>600.00	62	-	N/A
rovers8327	1.17	-	>600.00	45	-	N/A
gripper8	>600.00	8.14	0.03	N/A	53	71
gripper9	>600.00	12.86	0.06	N/A	59	79
gripper10	>600.00	19.78	0.04	N/A	65	87
gripper11	>600.00	err	0.07	N/A	err	95
gripper12	>600.00	err	0.06	N/A	err	103
satellite15	82.79	88.25	-	68	70	-
satellite16	>600.00	115.07	-	N/A	69	-
satellite17	129.39	127.62	-	74	73	-
satellite18	25.05	24.40	-	44	43	-
satellite19	>600.00	574.46	-	N/A	66	-
gminer7x7-06	6.00	5.07	6.34	33	35	34
gminer7x7-07	6.08	4.91	5.82	38	38	37
gminer7x7-08	3.06	2.08	2.81	25	25	25
gminer7x7-09	4.24	3.47	4.26	33	30	29
gminer7x7-10	5.96	4.83	6.05	35	35	35

Table 5.4: Comparison of running times and plans lengths (we assume that macro-actions are unfolded into primitive actions) for SATPLAN.

Problem	Time (in seconds)			Plan length		
	orig	upd-SG	upd-SAT	orig	upd-SG	upd-SAT
Blocks14-0	0.12	0.08	0.08	84	84	84
Blocks14-1	0.13	0.06	0.06	52	44	44
Blocks15-0	0.44	0.10	0.10	144	52	52
Blocks15-1	0.27	0.14	0.14	112	62	62
depots1817	>600.00	93.68	>600.00	N/A	122	N/A
depots4534	243.61	1.39	9.81	122	67	107
depots5656	>600.00	0.53	7.70	N/A	70	98
depots7615	>600.00	5.71	61.61	N/A	77	78
zeno-5-20a	1.22	0.87	-	91	91	-
zeno-5-20b	1.55	0.75	-	83	91	-
zeno-5-25a	2.85	0.98	-	95	105	-
zeno-5-25b	7.18	1.42	-	100	115	-
rovers4621	0.06	0.06	-	47	47	-
rovers5624	0.08	0.04	-	50	50	-
rovers7182	0.23	0.18	-	90	90	-
rovers8327	0.15	0.10	-	71	77	-
gripper16	0.05	0.06	3.76	101	135	101
gripper17	0.05	0.07	4.49	107	143	107
gripper18	0.06	0.08	5.26	113	151	113
gripper19	0.07	0.08	6.13	122	159	119
gripper20	0.07	0.10	7.02	128	167	125
satellite26	3.85	-	7.37	139	-	139
satellite27	2.80	-	3.63	135	-	139
satellite28	>600.00	-	11.76	N/A	-	194
satellite29	16.82	-	19.88	190	-	191
satellite30	72.18	-	32.63	229	-	229
satellite31	40.46	-	67.47	269	-	272
gminer7x7-06	0.22	0.04	0.03	170	31	31
gminer7x7-07	0.04	0.04	0.03	65	34	65
gminer7x7-08	>600.00	0.03	0.03	N/A	25	26
gminer7x7-09	0.14	0.04	0.03	130	29	32
gminer7x7-10	0.30	0.04	0.03	176	31	43

Table 5.5: Comparison of running times and plans lengths (we assume that macro-actions are unfolded into primitive actions) for LAMA.

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 Gripper domain where the running times were slightly worse and Satellite domain where the results were significantly worse. The quality of plans[4] generated in the updated domains was not much worse, however, sometimes the quality was slightly better and, surprisingly, in one Blocks problem more than twice better. The best results SGPLAN reached in 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 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 Blocks domain, SATPLAN was able to generate the plans faster, but at the price of significantly worse plans quality. On the other hand, 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 second (for the domain updated on the basis of SGPLAN's training plans). The reason of that rests in the fact that SATPLAN uses 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 Satellite and Gold Miner domains. Errors thrown by SATPLAN were caused by an insufficient memory or a large domain file (produced by ADL2STRIPS tool).

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

[4]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

5.4.4 Additional discussion

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 6 training plans for each domain (depending on the length of the training plans), we usually gathered enough knowledge for 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 the planning problems from the IPCs usually differ by the number of objects and not by the different types of the initial states or goals.

The generated macro-operators used in the comparison were in almost all cases combined only from two primitive operators, except in Gripper and Gold-Miner domains. Despite the construction of more complex macro-operators may reduce a depth of 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. At first, the training plans should be optimal (shortest) or nearly optimal, because non-optimal plans may contain flaws (useless actions) that may prohibit detection of useful macro-operators or useless primitive operators. At 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 which 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 the FF-based heuristics and in addition the Causal Graph heuristics. The FF-based planning techniques usually experience difficulties with plateaux. So, if there are such macro-operators that help the FF-based planner to escape the plateaux then the performance of the planner should significantly increase. It has been already studied in [22]. SATPLAN is a planner that translates Planning Graph into SAT and then use a SAT solver to solve the problem. The potential success, in this case, mainly rests in the reduction of makespan (i.e., the numbers of the 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 the 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 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 the most of problems). State-of-the-art systems Marvin or Macro-FF (SOL-EP version) are build on 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 the macro-operators genetically from the training plans which is quite a different policy than our method does. The usability of the macro-operators is evaluated by monitoring of running behavior of planners on updated training problems (by the macro-operators). WIZARD, in comparison to our method, reported better results, for example, in Satellite domain, but on the other hand, WIZARD spends many hours for learning phase, where our method spends seconds. Macro-FF (CA-ED version) generates the macro-operators from the analysis of static predicates, then adds them into the domain and then generate the training plans (with the macro-operators). Unlike that, our method generates macro-operators from the 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 the losing completeness). For example, in Depots domain, our method and Macro-FF (CA-ED version) found the same macro-operators. Our method, in addition, removed 4 (resp. 2) primitive operators by using SGPLAN (resp. SATPLAN) for generation the training plans. Removing the primitive operators brought much more benefit to the

planners' performance and often also to a plans quality.

Chapter 6

Eliminating unpromising actions

This chapter is devoted to eliminating actions that are unnecessary for the planning process [19]. Opposite to the other works [39, 42] (discussed in Section 3.2), which eliminate only unreachable actions, we are trying to eliminate actions that are normally reachable, but unnecessary for the planning process. Concretely, we are trying to learn connections between the actions and the initial or goal predicates.

6.1 Theoretical background

The planning systems process the planning domains with the corresponding planning problems to produce their solutions, plans. Informally said, the planning domains serve like the abstract templates for given environments, and the planning problems, on the other hand, describe those environments concretely and define certain planning tasks. The main challenge for planning is exploration of a huge search space defined by the number of applicable actions. We focus on the problem of the excessive number of actions that may mislead the planners when looking for the solutions. If we consider an operator with s arguments and a planning problem with n (untyped) objects then the number of all possible instances of this operator is n^s. It means, simply, that too many actions have to be considered during the planning process. In fact, many of these actions may be useless and not all these actions can be pruned by checking their reachability by the methods discussed

in Section 3.2. Our goal is to reduce the number of such useless actions (especially reachable ones) by analyzing simpler (training) plans.

We found out that in some cases there exists a connection between the operators' predicates in the preconditions and the initial predicates or the operators' predicates in the positive effects and the goal predicates. Now, we formally define several notions describing these kinds of connections.

Definition 6.1: Let $P = \langle \Sigma, s_0, g \rangle$ be a planning problem, o be a planning operator from Σ and p be a predicate. Operator o is *entangled by init (resp. goal)* with predicate p in planning problem P if and only if $p \in p(o)$ (resp. $p \in e^+(o)$) and there exists a plan π that solves P and for every action $a \in \pi$ which is an instance of o and for every grounded instance p_{ground} of the predicate p holds: $p_{ground} \in p(a) \Rightarrow p_{ground} \in s_0$ (resp. $p_{ground} \in e^+(a) \Rightarrow p_{ground} \in g$).

Definition 6.2: Let Σ be a planning domain, o be a planning operator from Σ and p be a predicate. Operator o is *fully entangled by init (resp. goal)* with predicate p if and only if there does not exist any planning problem P over Σ where o in not entangled by init (resp. goal) with p in P. In addition we define a set $plans(P, o, p, init$ (resp. $goal)) = \{\pi \mid \pi$ is a solution of P, if π contains the instance(s) of o, then the conditions of entanglement between o and p regarding Def. 6.1 must hold$\}$.

The entanglement by init (resp. goal) says that in a particular planning problem we can use only the actions sharing a predicate (or predicates) in the preconditions (resp. the positive effects) with the initial (resp. goal) predicates. The full entanglement extends this for every solvable planning problem in a particular domain.

If we recall our example of the BlocksWorld problem (from Chapter 2), then we can see on figure 6.1 a simple example of entanglements. We can see that the operator UNSTACK is fully entangled by init with predicate *on*. Analogically, the operator STACK is fully entangled by goal with predicate *on*. It means that we allow only to unstack boxes from their initial position and stack boxes to their goal positions. Picking up and putting down is unlimited (i.e., every box can be putted down to the table and picked up from the table without any restrictions).

In the following lines, we shall show that all the static predicates (the predicates that do not appear in the effects of any operator) with respect to

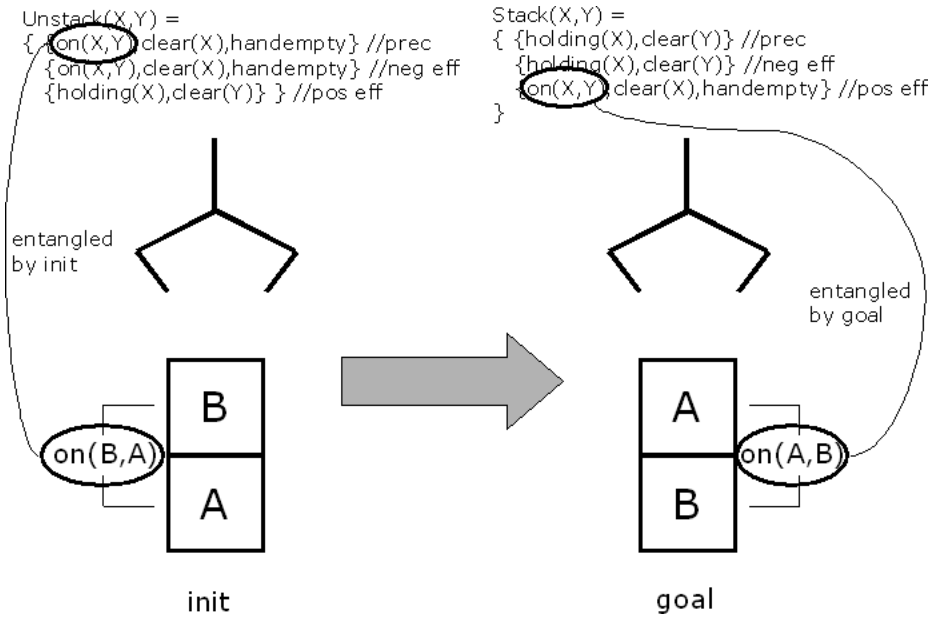


Figure 6.1: An example of entanglements in a simple BlocksWorld problem

the operators having these predicates in preconditions satisfy the conditions to be fully entangled by init.

Definition 6.3: Let Σ be a planning domain and p be a predicate such that there does not exist any substitution Θ and any operator o belonging to Σ such that $p\Theta \in e^-(o)$ or $p\Theta \in e^+(o)$ (in the other words $p\Theta$ represents a variant of p). Then p is a *static predicate* with respect to Σ.

Proposition 6.4: Let Σ be a planning domain and p be a static predicate (with respect to Σ). Then for every operator o belonging to Σ where there exists a substitution Ψ such that $p\Psi \in p(o)$, it holds that o is fully entangled by init with p and for every planning problem P $plans(P, o, p, init)$ contains all plans that solve P.

Proof: Let $P = \langle \Sigma, s_0, g \rangle$ be an arbitrary solvable planning problem and p be a static predicate with respect to Σ. Let s be an arbitrary state reachable from s_0 which means that there exists a valid sequence of actions (instances of the operators from Σ) transforming state s_0 to s. We shall show that

all instances of p from s_0 belong also to s and no other instance of p can belong to s. From the assumption we know that no variant of p appears in positive or negative effects of any operator from Σ. It means that we cannot add or remove any instance of p from s_0 which implies that in any reachable state from s_0 we have the same instances of p. Consequently for any action a in the above valid sequence of actions starting in s_0, it holds $p_{ground} \in p(a) \Rightarrow p_{ground} \in s_0$ (otherwise the action is not applicable to a reachable state). Hence, operator o from Σ giving these actions as its instances is entangled by init with predicate p in problem P. As P can be arbitrary planning problem in domain Σ, operator o is fully entangled by init with p. It is also clear that $plans(P, o, p, init)$ contains all plans that solves P. □

Definition 6.2 ensures the existence of plans for every solvable planning problem when pruning the actions violating the full entanglement conditions. For the static predicates as we proved before, the sets of plans solving particular (solvable) planning problems remain the same. In the general case, each full entanglement somehow restricts the sets of plans allowing only such actions that do not violate the particular full entanglement. However, the restrictions of the sets of plans differ regarding to the particular full entanglements. We have to restrict using the full entanglements together in such a way that every solvable planning problem remains solvable even if actions violating at least one full entanglement are pruned.

Definition 6.5: Let Σ be a planning domain, SFE a set of triples $SFE = \{(o_1, p_1, t_1), \ldots, (o_n, p_n, t_n)\}$, where o_i is an operator from Σ, p_i is a predicate from Σ and $t_i \in \{init, goal\}$ (i.e., o_i is fully entangled by t_i on p_i). If for every solvable planning problem P over Σ such that $\bigcap_{i=1}^{n} plans(P, o_i, p_i, t_i) \neq \emptyset$ holds, then SFE is *a set of compatible full entanglements.*

The following proposition formally describes how the full entanglement affects the possible operators' instances (actions).

Proposition 6.6: Let Σ be a planning domain, $P = \langle \Sigma, s_0, g \rangle$ be an arbitrary planning problem that is solvable and $SFE = \{(o_1, p_1, t_1), \ldots, (o_n, p_n, t_n)\}$ be a set of compatible full entanglements. Let A be the set of all actions a meeting the following conditions:

1. a is an instance of planning operator o from Σ
2. if $(o, p, init) \in SFE$ then $\forall\Theta : \ p\Theta \in \ p(a) \ \Rightarrow p\Theta \in \ s_0$
3. if $(o, p, goal) \in SFE$ then $\forall\Theta : \ p\Theta \in \ e^+(a) \ \Rightarrow p\Theta \in \ g$

Let Σ' be a grounded planning domain containing the set of actions A and the corresponding set of grounded predicates. Then, planning problem $P' = \langle \Sigma', s_0, g \rangle$ is solvable and the plan for P' is a plan for P too.

Proof: We have to prove that the reformulated planning problem P' is solvable and a plan for P' is also a plan for P. We can simply see that every plan for P' contains only the actions from A. As we can see from the assumption, the actions from A meet three conditions. For the first condition, it is clear that any solution of P contains only such actions that are instances of the operators defined in Σ. For the second condition, we know that if any operator o is fully entangled by init with any predicate p, then we allow only such actions (instances of o) whose instances of p in the preconditions correspond with instances of p belonging to the initial state. From definition 6.1 (entanglement) we know that there exists a plan π containing such actions satisfying the following conditions. If action $a \in \pi$ is an instance of operator o which is entangled by init with any predicate p in P then $\forall\Theta : p\Theta \in p(a) \Rightarrow p\Theta \in s_0$ (Θ is a substitution from variables to constants, so $p\Theta$ is a grounded predicate). Analogically it holds for the third condition. Considering SFE is a set of compatible full entanglements, we know that there exists at least one plan which satisfy the entanglement conditions (def. 6.1) for all operators and predicates that are fully entangled. Now it is clear that P' is also solvable. □

Informally said, if some operator is fully entangled on some predicate (or predicates) then we can omit such instances of this operator where the predicate (or predicates) in the preconditions or positive effects are not corresponding with the particular predicates belonging to the initial or goal states. We assume that by detecting (compatible) full entanglements we can omit a lot of unnecessary actions. Full entanglements can be understood as a piece of knowledge that we can pass to existing planners via the definition of the planning problem without updating the source code of the planner. There are (at least) two ways how the information about full entanglements can be encoded in the planning domain/problem. The first option rests in

```
(:action unstack
 :parameters  (?x ?y - block)
 :precondition (and (on ?x ?y) (clear ?x) (emptyhand) (stai_on ?x ?y))
 :effect (and (holding ?x) (clear ?y)
              (not (on ?x ?y)) (not (clear ?x)) (not (emptyhand))))
```

Figure 6.2: Example of reformulated unstack operator which is entangled by init with predicate *on* - stai_on represents an added static predicate

the production of grounded domains (and problems) as we did it in Proposition 6.6. The main disadvantage of this approach rests in the fact that the grounded domains may be very large which may cause a significant slowdown of the planner in the pre-processing phase (loading the problem). It may also disallow some other techniques such as lifting. The second option rests in the extension of the planning domains by special static predicates. We shall present this approach in the following paragraphs.

Definition 6.7: Let Σ be a planning domain and $P = \langle \Sigma, s_0, g \rangle$ be an arbitrary planning problem. If operator o from Σ is fully entangled by init (resp. goal) with predicate p and p is not a static predicate then we define *reformulated domain* $Ref(\Sigma, o, p)$ and *reformulated problem* $Ref(P, o, p)$ in the following way:

1. create a new predicate p' which is not present in Σ and has the same arity as p
2. create an operator $o' = (p'(o), e^-(o), e^+(o))$, where $p'(o) = p(o) \cup p'$ (the arguments of p' correspond with the arguments of p)
3. create a reformulated domain $Ref(\Sigma, o, p)$ from Σ by adding p' and replacing o by o'
4. create a reformulated problem $Ref(P, o, p) = \langle Ref(\Sigma, o, p), s'_0, g \rangle$, where $s'_0 = s_0 \cup \{p'\Theta \mid p\Theta \in s_0 \text{ (resp. } p\Theta \in g)\}$

An example of reformulated operator unstack which is entangled by init with predicate *on* is showed in fig. 6.2. It is clear that we have to also add

stai_on predicates into the initial state (s_0) in such a way that if $on(X, Y) \in s_0$, then $stai\_on(X, Y)$ is added to s_0.

Theorem 6.8: Let $P = \langle \Sigma, s_0, g \rangle$ be a planning problem and $SFE = \{(o_1, p_1, t_1), \ldots, (o_n, p_n, t_n)\}$ be a set of compatible full entanglements. Let P' be a problem obtained from P by a successive application of $Ref(P, o, p)$ for every $(o, p, t) \in SFE$, where p is not a static predicate. Then, if P is solvable then P' is solvable and the plan for P' is a plan for P too.

Proof: Recall Definition 6.7, where $Ref(P, o, p)$ was introduced. We added the predicate p' into the reformulated domain and problem. It is clear that p' is a static predicate, because it does not appear in the effects of any operator. From the proof of Proposition 6.4 we know that static predicates hold through the whole planning process without being changed. It can be easily observed that no action with the instance of p' in the precondition that do not correspond with the initial instances of p' can be applied to any state. From the 2nd and 4th point of Def. 6.7 we can see that the arguments of p' correspond with the arguments of p. From Proposition 6.6 we know that if we remove such operators' instances from the domain that 'break' full entanglements from SFE (recall conditions 2 and 3 from Proposition 6.6), then it does not affect the solvability of the reformulated problem.□

6.2 Heuristic detection of full entanglements

In the previous section, we showed that if we find out the full entanglement relations then we can restrict the set of grounded actions (Definition 6.7 and Theorem 6.8) to be assumed during planning without affecting solvability of the planning problem. The remaining question is how the (compatible) full entanglements can be detected. Theoretically, we are facing the problem of validating the entanglements for all solvable planning problems over the particular domains. For the static predicates the detection is easy and can be realized for every domain (Proposition 6.4). For the other predicates we have two options. First, we can prove the (compatible) full entanglements theoretically (it is not practically possible to explore all solvable planning problems) or second, we can use the heuristics to guess that a given operator can be fully entangled with a given predicate by exploring only a fraction

of planning problems (training planning problems). In the first case, we can fully exploit the theoretical results from the previous section and we are sure that we cannot break the solvability of planning problems by the pruning of actions mentioned above. In the second case we cannot assure that the solvability is preserved because the heuristic method does not guarantee full entanglement (theoretically).

In our opinion, the first option may require domain-dependent approaches and hence a domain expert who can handle it. Therefore, in the rest of the chapter we will focus on the second option with the goal to have a fully automated domain-independent approach. It motivates us to develop a method for detection of full entanglement based on heuristics.

The main idea of our approach is following. Instead of exploring all planning problems and finding a plan validating the condition from Definition 6.1, we assume only a subset of problems, so called training problems, together with the existing plans for these problems. The entanglement condition as specified in Definition 6.1 is checked only for these training problems and plans and if validated we declare the full entanglement as the following heuristics specifies.

Heuristics 1: Let Σ be a planning domain. If for each training planning problem P over Σ together with a plan solving the problem it holds that operator o from Σ is entangled by init (resp. goal) on predicate p then operator o is considered as fully entangled by init (resp. goal) with predicate p. We also consider that all detected full entanglements are compatible.

Heuristics 1 gives us an opportunity to develop the algorithm (fig. 6.3) for detection of full entanglement (by init or goal) at the cost of losing its soundness. It means that the algorithm may declare full entanglement even if it does not hold. On the other hand we believe that it happens only occasionally because the planning problems (over the same planning domain) usually differ only in the number of objects.

The algorithm starts with an assumption that every operator from given domain is fully entangled by init (resp. goal) with every predicate listed in the corresponding preconditions (resp. positive effects). The behavior of the algorithm is quite straightforward. The algorithm is verifying if the conditions of entanglement are satisfied in each training plan for every operator and the corresponding predicate. If the condition of entanglement is broken (once is enough) then we set the particular pair (operator and predicate) as

Procedure DETECT-FULL-ENTANGLEMENTS(`IN` planning domain Σ, `IN` set of planning problems and their plans, `OUT` a set of compatible full entanglements)

1: **Set** the full entanglements by init (resp. goal) between all operators and predicates from the corresponding preconditions (resp. positive effects) from Σ
2: **ForEach** planning problem P **do**
3: **ForEach** operator o from Σ **do**
4: **ForEach** $p \in p(o)$ **do**
5: **ForEach** $a \in \pi$ where a is an instance of o and π is a plan solving P **do**
6: **If** $\neg\exists\Theta : p\Theta \in s_0 \wedge p\Theta \in p(a)$, where s_0 is an initial state of P **then Unset** the full entanglement by init between o and p
7: **EndForeach**
8: **EndForeach**
9: **ForEach** $p \in e^+(o)$ **do**
10: **ForEach** $a \in \pi$ where a is an instance of o and π is a plan solving P **do**
11: **If** $\neg\exists\Theta : p\Theta \in g \wedge p\Theta \in e^+(a)$, where g represents a set of goal predicates of P **then Unset** the full entanglement by goal between o and p
12: **EndForeach**
13: **EndForeach**
14: **EndForeach**
15: **EndForeach**

Figure 6.3: Algorithm for heuristic detection of the full entanglements by init or by goal.

not fully entangled.

Theorem 6.9: Let n be the number of training planning problems $P_i = \langle \Sigma, s_0^i, g^i \rangle$ and π_i be a plan solving P_i. Let O be the set of operators from Σ and $\wp$ be the set of predicates from Σ. Then the worst case time complexity of the algorithm is $O\left(|O||\wp| \sum_{i=1}^{n} (|\pi_i|) \left(\sum_{i=1}^{n} (|s_0^i|) + \sum_{i=1}^{n} (|g^i|)\right)\right)$.

Proof: At, first we show that the cycle (lines 5-7) runs in at most $|\pi_i||s_0^i|$ steps. We are going through the given plan π_i and for each action a and the given predicate p, we are trying to find a predicate from s_0^i that matches the instance of p being presented in $p(a)$. From this, we get the time complexity at most $|\pi_i||s_0^i|$. Similarly for the cycle (lines 10-12), where we get the time complexity at most $|\pi_i||g^i|$. Hence the cycle (lines 4-8) has the time complexity at most $|\wp||\pi_i||s_0^i|$, because we can perform the cycle (5-7) for every predicate presented in the domain Σ. Similarly, the cycle (lines 9-13) has the time complexity at most $|\wp||\pi_i||g^i|$. The cycle (3-14) has the time complexity at most $|O||\wp||\pi_i|(|s_0^i| + |g^i|)$, because we perform the cycles (lines 4-8 and 9-13) for every operator and the given plan π_i. The main cycle (lines 2-15) is performed for every training planning problem P_i and its plan π_i. That gives us the time complexity in the worst case $O\left(|O||\wp| \sum_{i=1}^{n} (|\pi_i|) \left(\sum_{i=1}^{n} (|s_0^i|) + \sum_{i=1}^{n} (|g^i|)\right)\right)$. The time complexity of the line 1 is clearly $|O||\wp|$ which do not affect the complexity of the main cycle. □

Remark 6.10: In the most common planning domains and their corresponding planning problems it holds that the number of the operators and the number of the predicates are quite small. Similarly, the number of the goal predicates is usually small. Hence we can lower the time complexity estimation to $O\left(\sum_{i=1}^{n} (|\pi_i|) \sum_{i=1}^{n} (|s_0^i|)\right)$.

The low time complexity of the algorithm means that we can run the algorithm even for more training problems. However, we need to consider that every training problem must be solved before the algorithm can start computation (we need a training plan). Even though there are good planners around, solving of many training problems can still be very time consuming. It means that we should use 'reasonable' training plans - usually toy problems - only.

Moreover, the existing planners frequently do not produce the shortest plans even though some of them successfully participated in optimal tracks

of IPC. This may cause problems to our heuristic algorithm (fig. 6.3) because if the training plan contains actions that are not necessary to solve the problem, these actions may break the condition of full entanglement. Hence we suggest to weaken the heuristics further to allow ‘a few‘ violations of the entanglement condition.

Heuristics 2: Let Σ be a planning domain. If for each training planning problem P over Σ holds that operator o from Σ is NOT entangled by init (resp. goal) on predicate p in less or equal than $n\%$ times (‘flaws‘ ratio), then operator o is considered as fully entangled by init (resp. goal) with predicate p. We also consider that all detected full entanglements are compatible.

The ‘flaws‘ ratio describes the ratio between the number of violations of full entanglements and the total number of instances of a particular operator in all training plans. To follow heuristics 2 the detection algorithm can be updated in such a way that instead of unsetting of full entanglements we increase the number of violations of these full entanglements (lines 6 and 11) and decide about the full entanglement at the end based on the ‘flaws‘ ratio. Clearly, the risk of ‘false positive‘ detection of full entanglements is getting higher as the ‘flaws‘ ratio raises.

‘Flaws‘ ratio introduces a parameter to the algorithm which raises the question how to set this parameter. In the lines bellow we shall present an approach which can help to determine the ‘flaws‘ ratio.

1. set $flaws$ to n, where $n \in (0; 1)$; according to our experiments we suggest starting with $n = 0.1$
2. generate the entanglements by the modified algorithm using ‘flaws‘ ratio $flaws$
3. compare the generated entanglements to the entanglements obtained by the original algorithm (without ‘flaws‘). If same then quit (we are not able to gather additional entanglements).
4. generate a reformulated domain and reformulated training problems considering the generated entanglements (according to Definition 6.6)
5. run the planner on all the reformulated training problems. If succeed then quit (we found proper full entanglements with respect to training plans).

6. otherwise set $flaws := flaws - \epsilon$ ($\epsilon > 0$, for example $\epsilon = 0.01$) and go to the second step.

Remark 6.11: The time complexity of the Heuristics 2 do not differ from the time complexity of the Heuristics 1. Considering the algorithm determining the 'flaws' ratio, we have to run the planner again to test if the reformulated domain works with the training problems (step 5 in the previously mentioned approach). According to the complexity of classical planning we may reach at worst EXPSPACE-complete problem. On the other hand, we consider only toy problems that are solved fast.

6.3 Experimental results

We evaluated our approach experimentally in the following way. At first, we looked for what our methods can learn (how many new predicates are added to modified domains by applying Definition 6.6). At second, we compared running times and plans quality[1] of well known planners SATPLAN 2006 [48] (winner of the 5th IPC optimal track), SGPLAN 5.22 [44] (winner of the 5th IPC sub-optimal track) and LAMA [65] (winner on the 6th IPC sub-optimal track) on a couple of planning domains well known from the IPC. In summary, we proceeded the evaluation in the following steps:

- generate several simpler training plans (by SATPLAN),
- run our methods (with and without 'flaws' ratio) for detection of entanglements,
- generate reformulated domains and problems considering the detected entanglements,
- run the planners both on the original problems and on the reformulated problems and compare results.

6.3.1 Tested domains

We chose several planning domains for our experiments to ensure that the proposed approach is generally applicable (rather than specific for a partic-

[1] In this case, the plans quality is greater if the plan lengths is smaller.

ular planning domain). In particular, we used domains well known from the IPC. Some domains are the same as in Chapter 5.

Depots is a planning domain from the 3rd IPC. This domain accommodate both the blocks and logistics environments. 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. The trucks can behave like 'tables', since the pallets on which crates are stacked are limited.

Driver Log is a planning domain from the 3rd IPC. This domain involves driving trucks around delivering packages between locations. The complication is that the trucks require drivers who must walk between the trucks in order to drive them. The paths for walking and the roads for driving form different maps on the locations.

Zeno Travel is a planning domain from the 3rd IPC. This domain involves transporting people around in planes, using different modes of movement: fast and slow.

Storage is a planning domain from the 5th IPC. This domain is about moving a certain number of crates from some containers to some depots by hoists. Inside a depot, each hoist can move according to a specified spatial map connecting different areas of the depot. The test problems for this domain involve different numbers of depots, hoists, crates, containers, and depot areas.

Gold Miner is a planning domain from the 6th IPC learning track. A robot is in a mine and has the goal of reaching a location that contains gold. The mine is organized as a grid with each cell either being hard or soft rock. There is a special location where the robot can either pickup an endless supply of bombs or pickup a laser cannon. The laser cannon can shoot through both hard and soft rock, whereas the bomb can only penetrate soft rock. However, the laser cannon will also destroy the gold if used to uncover the gold location. The bomb will not destroy the gold.

Matching Blocksworld is a planning domain from the 6th IPC learning track. This is a simple variant of the blocks world where each block is either positive or negative and there are two hands, one positive and one negative. The twist is that if a block is picked up by a hand of opposite polarity then it is damaged such that no other block can be placed on it. The interaction between the hands and blocks of the same polarity is just as in the standard blocks world.

Problem	no flaws ratio		with flaws ratio	
	Unary	Binary	Unary	Binary
Depots	1(2)	2(2)	1(2)	3(3)
Driverlog	1(1)	2(2)	1(1)	2(3)
Storage	2(2)	1(1)	2(2)	1(1)
Zenotravel	0(0)	2(2)	0(0)	2(2)
GoldMiner	3(3)	0(0)	3(3)	0(0)
MatchingBW	5(9)	2(4)	5(9)	3(5)
Parking	3(4)	2(2)	3(4)	2(2)
Thoughtful	15(47)	3(6)	15(47)	3(6)

Table 6.1: The number of added unary and binary predicates into the domains and how many times they were added to operators' preconditions (in brackets).

Parking is a planning domain from the 6th IPC learning track. This domain involves parking cars on a street with N curb locations where the cars can be double parked but not triple parked. The goal is to move from one configuration of parked cars to another configuration by driving the cars from one curb location to another.

Thoughtful is a planning domain from the 6th IPC learning track. This domain models a simplified version of Thoughtful Solitaire, which is a solitaire variant that is played with all of the cards visible. The rules follow those described in [8] but simplified so that one can turn each card from the talon rather than 3 cards at a time.

6.3.2 Learning phase

For the learning phase we used 3 to 6 training planning problems depending on the particular domain. For the generation of the training plans we used SATPLAN. The selected training planning problems were not too complex which results that the training plans were generated mostly within tenths of seconds. The detection of the full entanglement with generation of reformulated domain and problems took at most half of second[2].

In table 6.1 it is shown how many unary and binary predicates were added into the domains and how many times they were added to operators' pre-

[2]performed on Core2Duo 2.66GHz, 4GB RAM, Win XP

conditions (in brackets). We focused only on unary and binary predicates, because nullary predicates did not bring any useful information and ternary predicates (or more) were not presented in tested domains (or never detected as fully entangled). We also compared both methods - without and with ‘flaws‘ ratio. We found out that using the ‘flaws‘ ratio contributed in Depots, Driver Log and Matching BlocksWorld domains. The other domains remained the same as reformulated by the method without ‘flaws‘ ratio.

6.3.3 Running times and plans quality comparison

The results of the evaluation are showed in tables 6.2 (running times comparison) and 6.3 (plan lengths comparison)[3]. Symbol ’-’ in the cells of the table was used only in columns representing reformulated problems by the method with ‘flaws‘ ratio where it indicates the fact that this method did not reveal additional knowledge than the method without ‘flaws‘ ratio (see the previous subsection). Term ’err’ in the table cells means that the planner terminated with an unexpected error. All reformulated problems remained solvable except thoughtful-s7-t5b and thoughtful-s7-t5c. This was caused by the fact that the (toy) training problems for Thoughtful domain took in account full entanglements that were applicable for the problems of type thoughtful-s5-t4, but too restrictive for the problems of type thoughtful-s7-t5.

The best results were acquired by SATPLAN. Running times of reformulated problems (especially those that were generated by the method with ‘flaws‘ ratio) were (mostly significantly) better in all tested cases. The plan quality (the less plan length the better) were also in most cases better in the reformulated problems. Even though SATPLAN produces optimal plans in makespan, it does not ensure the optimality with respect to the number of actions. SATPLAN simply finds the first plan with the lowest makespan (actions can run in parallel). The reformulated problems operate with the smaller number of actions than the original ones. It may result in the better plan quality for the reformulated problems.

SGPLAN’s running times of the reformulated problems were mostly better than the original ones (especially in the reformulated Matching BlocksWorld domain and Parking domain). However, in some cases the running times of the reformulated problems were significantly worse. The reason of this may lie in SGPLAN’s heuristics that is based on heuristics used in Metric-FF [41].

[3]performed on XEON 2.4GHz, 1GB RAM, Ubuntu Linux

problem	SATPLAN			SGPLAN			LAMA		
	orig	ref no fr	ref with fr	orig	ref no fr	ref w fr	orig	ref no fr	ref with fr
depotprob1817	>1000	>1000	>1000	24.69	391.46	0.15	331.03	95.94	3.64
depotprob1916	137.57	49.27	5.32	0.41	>1000	80.95	1.70	0.58	0.14
depotprob4321	5.32	2.24	0.48	0.02	0.10	0.00	4.96	3.69	0.03
depotprob4398	1.08	0.72	0.27	0.04	0.03	0.00	0.23	0.10	0.03
depotprob5646	0.39	0.26	0.08	0.01	0.01	0.00	0.17	0.07	0.02
depotprob5656	222.84	76.73	5.92	411.33	337.18	0.24	>1000	>1000	3.42
depotprob6178	6.87	4.54	1.50	0.11	0.06	0.02	11.66	6.06	0.06
depotprob6587	3.35	2.18	0.55	0.06	0.06	0.00	0.43	0.19	0.05
depotprob7654	10.08	6.13	1.39	0.09	0.04	0.01	1.48	46.58	12.41
depotprob8715	35.68	28.31	8.70	0.26	0.13	0.05	1.66	0.54	0.17
dlog-2-3-6a	24.67	3.52	3.35	0.94	0.05	0.02	2.20	1.68	1.68
dlog-2-3-6b	3.82	1.31	1.25	0.01	0.01	0.01	1.61	1.03	1.01
dlog-3-3-6	2.86	1.00	0.90	0.56	0.14	0.07	0.15	1.63	1.63
dlog-4-4-8	28.94	13.08	8.72	0.04	0.01	0.01	0.13	0.06	0.06
dlog-5-5-10	>1000	167.16	169.36	>1000	25.20	17.82	18.24	7.55	7.43
dlog-5-5-15	507.89	110.19	121.43	6.93	3.81	1.63	14.33	6.20	6.04
dlog-5-5-20	>1000	>1000	406.60	23.15	>1000	>1000	17.75	9.24	8.88
storage-11	85.47	9.25	-	0.00	err	-	0.56	0.08	-
storage-12	38.01	9.65	-	0.00	0.00	-	0.70	0.06	-
storage-13	153.45	64.26	-	0.00	0.00	-	1.25	0.02	-
storage-14	24.70	23.58	-	0.00	0.00	-	5.15	0.33	-
storage-15	8.28	0.60	-	0.01	0.00	-	0.88	0.56	-
ztravel-3-7	1.90	1.59	-	0.01	0.00	-	0.04	0.04	-
ztravel-3-8a	1.71	1.31	-	0.01	0.00	-	0.05	0.04	-
ztravel-3-8b	0.86	0.65	-	0.01	0.00	-	0.04	0.03	-
ztravel-3-10	3.79	3.78	-	0.02	0.01	-	0.05	0.05	-
ztravel-5-10	34.49	22.53	-	0.22	0.19	-	0.24	0.20	-
ztravel-5-15a	92.04	40.70	-	0.12	0.09	-	0.48	0.37	-
ztravel-5-15b	err	err	-	0.58	0.47	-	0.62	0.48	-
gold-miner-7x7-01	5.99	4.97	-	err	0.00	-	0.30	0.04	-
gold-miner-7x7-02	4.36	3.58	-	err	0.00	-	0.16	0.04	-
gold-miner-7x7-03	4.12	3.46	-	err	0.01	-	0.29	0.04	-
gold-miner-7x7-04	9.15	7.62	-	err	0.01	-	0.21	0.02	-
gold-miner-7x7-05	9.78	8.16	-	err	0.00	-	0.38	0.03	-
gold-miner-7x7-06	6.00	4.96	-	err	0.00	-	0.18	0.02	-
gold-miner-7x7-07	6.08	4.85	-	err	0.01	-	0.04	0.02	-
gold-miner-7x7-08	3.06	2.60	-	err	0.00	-	>1000	0.02	-
gold-miner-7x7-09	4.24	3.61	-	err	0.01	-	0.14	0.03	-
gold-miner-7x7-10	5.96	4.92	-	err	0.00	-	0.29	0.05	-
thoughtful-s5-t4d	3.20	0.47	-	0.04	0.02	-	err	err	-
thoughtful-s5-t4e	44.41	2.56	-	0.05	0.02	-	err	err	-
thoughtful-s5-t4f	5.46	0.94	-	0.02	0.02	-	err	err	-
thoughtful-s5-t4g	3.92	1.11	-	0.03	0.02	-	err	err	-
thoughtful-s5-t4h	3.26	0.58	-	0.04	0.01	-	err	err	-
thoughtful-s7-t5a	288.57	99.73	-	0.10	205.71	-	err	err	-
thoughtful-s7-t5b	136.44	unsolvable	-	1.39	unsolvable	-	err	err	-
thoughtful-s7-t5c	232.32	unsolvable	-	0.13	unsolvable	-	err	err	-
matching-bw-n15a	27.81	19.93	1.95	>1000	>1000	0.25	0.22	0.22	0.25
matching-bw-n15b	34.25	10.18	1.37	>1000	>1000	170.39	0.36	0.31	0.38
matching-bw-n15c	26.80	9.39	1.20	>1000	20.89	284.65	0.71	1.74	0.14
matching-bw-n15d	40.74	14.41	1.45	>1000	>1000	>1000	0.51	0.29	0.18
matching-bw-n15e	59.00	14.62	1.73	>1000	>1000	47.17	0.17	0.80	0.07
matching-bw-n20a	>1000	189.75	15.00	>1000	>1000	>1000	0.46	0.37	0.25
matching-bw-n20b	245.12	54.35	4.39	>1000	>1000	0.35	3.63	3.44	0.93
matching-bw-n20c	363.36	60.94	5.62	>1000	533.89	237.64	>1000	55.72	1.36
matching-bw-n20d	195.87	43.04	4.31	>1000	10.40	>1000	0.77	1.43	0.48
parking-a	>1000	399.11	-	0.67	0.02	-	0.16	0.15	-
parking-b	>1000	98.13	-	0.73	0.11	-	0.22	0.14	-
parking-c	304.47	17.68	-	0.53	0.08	-	0.16	0.12	-
parking-d	>1000	889.98	-	0.02	0.01	-	0.34	0.13	-
parking-e	>1000	167.20	-	0.76	0.05	-	0.31	0.14	-
parking-f	>1000	>1000	-	0.47	0.27	-	0.20	0.13	-
parking-g	>1000	>1000	-	17.89	0.83	-	14.16	0.58	-
parking-h	>1000	>1000	-	19.39	0.71	-	0.55	1.01	-

Table 6.2: Comparison of the running times (in seconds) of original problems and problems reformulated by our methods (without and with ‘flaws‘ ratio).

problem	SATPLAN			SGPLAN			LAMA		
	orig	ref no fr	ref with fr	orig	ref no fr	ref w fr	orig	ref no fr	ref with fr
depotprob1817	N/A	N/A	N/A	100	99	95	118	111	93
depotprob1916	79	74	70	83	N/A	57	62	59	60
depotprob4321	43	46	39	41	35	34	39	38	37
depotprob4398	32	36	35	28	28	28	30	27	26
depotprob5646	31	28	28	26	26	28	26	26	26
depotprob5656	70	71	69	133	70	62	N/A	N/A	63
depotprob6178	51	51	46	48	48	37	41	41	39
depotprob6587	30	29	26	28	26	24	25	25	23
depotprob7654	41	39	40	35	33	33	33	35	33
depotprob8715	50	46	44	34	34	36	34	33	33
dlog-2-3-6a	46	43	38	41	42	42	44	49	49
dlog-2-3-6b	27	27	28	32	32	32	39	30	30
dlog-3-3-6	37	38	35	46	41	41	51	46	46
dlog-4-4-8	54	52	53	47	47	47	44	44	44
dlog-5-5-10	N/A	98	95	N/A	103	105	132	114	114
dlog-5-5-15	92	84	89	110	106	106	112	95	95
dlog-5-5-20	N/A	N/A	92	105	N/A	N/A	127	114	114
storage-11	22	20	-	17	N/A	-	32	20	-
storage-12	24	24	-	17	20	-	32	20	-
storage-13	18	18	-	18	20	-	38	20	-
storage-14	22	22	-	19	26	-	32	24	-
storage-15	25	26	-	21	20	-	22	20	-
ztravel-3-7	22	19	-	18	18	-	15	18	-
ztravel-3-8a	27	25	-	27	27	-	24	23	-
ztravel-3-8b	27	29	-	29	29	-	28	28	-
ztravel-3-10	31	38	-	36	36	-	31	30	-
ztravel-5-10	42	38	-	40	40	-	41	39	-
ztravel-5-15a	50	53	-	60	60	-	47	47	-
ztravel-5-15b	N/A	N/A	-	55	55	-	57	57	-
gold-miner-7x7-01	35	35	-	N/A	33	-	176	31	-
gold-miner-7x7-02	32	32	-	N/A	30	-	161	28	-
gold-miner-7x7-03	32	32	-	N/A	34	-	257	32	-
gold-miner-7x7-04	43	42	-	N/A	41	-	130	41	-
gold-miner-7x7-05	39	41	-	N/A	39	-	157	39	-
gold-miner-7x7-06	33	34	-	N/A	33	-	182	33	-
gold-miner-7x7-07	38	38	-	N/A	36	-	65	34	-
gold-miner-7x7-08	25	25	-	N/A	27	-	N/A	25	-
gold-miner-7x7-09	33	29	-	N/A	31	-	130	29	-
gold-miner-7x7-10	35	35	-	N/A	33	-	176	31	-
thoughtful-s5-t4d	37	37	-	28	30	-	N/A	N/A	-
thoughtful-s5-t4e	41	36	-	33	31	-	N/A	N/A	-
thoughtful-s5-t4f	36	32	-	31	28	-	N/A	N/A	-
thoughtful-s5-t4g	38	38	-	32	32	-	N/A	N/A	-
thoughtful-s5-t4h	41	36	-	31	34	-	N/A	N/A	-
thoughtful-s7-t5a	58	65	-	50	49	-	N/A	N/A	-
thoughtful-s7-t5b	65	N/A	-	84	N/A	-	N/A	N/A	-
thoughtful-s7-t5c	71	N/A	-	51	N/A	-	N/A	N/A	-
matching-bw-n15a	42	42	42	N/A	N/A	46	68	62	50
matching-bw-n15b	56	52	52	N/A	N/A	54	66	78	60
matching-bw-n15c	38	42	38	N/A	74	34	66	72	46
matching-bw-n15d	40	42	42	N/A	N/A	N/A	58	60	56
matching-bw-n15e	34	36	36	N/A	N/A	42	40	44	34
matching-bw-n20a	N/A	52	50	N/A	N/A	N/A	62	66	66
matching-bw-n20b	48	50	48	N/A	N/A	60	78	68	50
matching-bw-n20c	54	54	54	N/A	90	36	N/A	70	72
matching-bw-n20d	46	46	46	N/A	74		86	92	64
parking-a	N/A	16	-	20	31		19	31	-
parking-b	N/A	15	-	22	36		20	31	-
parking-c	12	12	-	25	29		15	23	-
parking-d	N/A	14	-	18	18		31	18	-
parking-e	N/A	13	-	29	56		27	20	-
parking-f	N/A	N/A	-	25	39		21	19	-
parking-g	N/A	N/A	-	34	52		30	20	-
parking-h	N/A	N/A	-	37	58		19	38	-

Table 6.3: Comparison of the plan lengths of original problems and problems reformulated by our methods (without and with 'flaws' ratio).

FF based heuristics is vulnerable to problems that may contain dead-ends (i.e. we can reach such a state from which the goal is unreachable). Pruning of actions done by our methods may cause that some problems become dead-ended. The quality of plans was better in most of the reformulated Depots problems, slightly worse in the reformulated Storage problems and significantly worse in the reformulated Parking domain.

LAMA's running times of the reformulated problems were mostly better than the original ones. However, similarly to SGPLAN, the running times of several reformulated problems were worse. The reason of this also may rest in LAMA's heuristics (based on FF and Causal Graph) that may be vulnerable to problems with dead-ends. The most interesting and also a bit surprising result was achieved in Gold Miner domain, where the quality of solutions of reformulated problems were significantly better than in original ones.

6.3.4 Discussion

The presented results showed that our approach is reasonable and can help planners and increase their performance. SATPLAN, which gained the best results, is based on transforming of the planning graph [9] into SAT formulae. We suppose that SATPLAN benefits from our methods because our methods result in a significant reduction of the size of the planning graph. Planners like SGPLAN or LAMA using FF-based heuristics may occasionally experience difficulties when using our approach. It was discussed in the previous subsection that the main problem of this (we suppose) rests in the fact that problems reformulated by our methods may contain dead-ends. On the other hand in most of problems our methods are still helpful. For instance, Matching BlocksWorld domain is a good example of problems with dead-ends. As we anticipated, SGPLAN experienced difficulties when solving the original problems. Our methods helped SGPLAN to solve these problems, often in a very good time. It shows that our methods can be successfully used in a connection with planners based on FF-based heuristics (like SGPLAN) mostly for problems with dead-ends. In addition, many planning problems, especially real world ones, have dead-ends.

Chapter 7

Conclusion and future work

This thesis brings the contributions to the area of learning for classical planning. Besides the theoretical background that we proposed, we focused on learning additional (domain-dependent) knowledge from the training plans and encoding it back to the domains or problems. Hence, we do not need to modify the existing planners' source code.

First of the proposed methods is a method for generating macro-operators and removing useless primitive operators from planning domain. The method explores pairs of actions (not necessarily adjacent), which 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 can be used with arbitrary planners. The presented evaluation (Section 5.4) showed that using the 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 analyzing of plans structures much easier. In real world applications, it may be more difficult to use our method properly (for example, 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.

The second proposed method performs domain and problem transformations that can prune many unnecessary actions that may mislead planners when solving the planning problem. The proposed methods are based on the detection of connectivity (here defined as full entanglements) between the

operators' instances in the training plans and the initial or goal predicates in the corresponding planning problems. The main advantage of the proposed approach rests in the possible reduction of the search space. The presented experimental evaluation (Section 6.3) confirmed that in most cases our approach reduced the time needed to find a solution. Like in the first case, the reformulated domains and problems can be used with common planners without changing their source code.

Despite the progress given by this thesis, there still remain many possibilities for future research. Even though the presented methods (macro-operators, entanglements) were developed and evaluated separately, we see an opportunity in putting them together. It may produce more sophisticated knowledge and increase the performance of planners. Because our methods, especially the method for generating the macro-operators, also depends on the pre-defined parameters (a different set of parameters may result in different outputs) it is quite necessary to evaluate somehow these outputs, because we have to produce the best (or nearly the best) ones. This task still requires research, but there exists a system [66] that is able to predict planners' behavior on particular domains and problems. The possible connection of our methods with this system may be beneficial.

References

[1] M. Ai-Chang, J. L. Bresina, L. Charest, A. Chase, J. C. jung Hsu, A. K. Jónsson, B. Kanefsky, P. H. Morris, K. Rajan, J. Yglesias, B. G. Chafin, W. C. Dias, and P. F. Maldague. Mapgen: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8–12, 2004.

[2] J. F. Allen. Planning as temporal reasoning. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA*, pages 3–14, 1991.

[3] J. F. Allen and J. A. G. M. Koomen. Planning using a temporal world model. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG*, pages 741–747, 1983.

[4] G. Armano, G. Cherchi, and E. Vargiu. A parametric hierarchical planner for experimenting abstraction techniques. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico*, pages 936–941, 2003.

[5] G. Armano, G. Cherchi, and E. Vargiu. Dhg: A system for generating macro-operators from static domain analysis. In *IASTED International Conference on Artificial Intelligence and Applications, part of the 23rd Multi-Conference on Applied Informatics, Innsbruck, Austria*, pages 18–23, 2005.

[6] F. Bacchus and M. Ady. Planning with resources and concurrency: A forward chaining approach. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA*, pages 417–424, 2001.

[7] D. E. Bernard, E. B. Gamble, N. F. Rouquette, B. Smith, Y. W. Tung, N. Muscettola, G. A. Dorias, B. Kanefsky, J. Kurien, W. Millar, P. Nayal, K. Rajan, and W. Taylor. Remote agent experiment ds1 technology validation report. Technical report, Ames Research Center and JPL, 2000.

[8] R. Bjarnason, P. Tadepalli, and A. Fern. Searching solitaire in real time. *International Computer Games Association Journal*, 30(3):131–142, 2007.

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

[10] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK*, volume 1809 of *LNCS*, pages 360–372, 1999.

[11] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research (JAIR)*, 24:581–621, 2005.

[12] T. Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69:165–204, 1994.

[13] A. Cesta, G. Cortellessa, M. Denis, A. Donati, S. Fratini, A. Oddi, N. Policella, E. Rabenau, and J. Schulster. Mexar2: Ai solves mission planner problems. *IEEE Intelligent Systems*, 22(4):12–19, 2007.

[14] A. Cesta, S. Fratini, and F. Pecora. Planning with multiple-components in omps. In *New Frontiers in Applied Artificial Intelligence, 21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2008, Wroclaw, Poland*, volume 5027 of *LNCS*, pages 435–445, 2008.

[15] L. Chrpa. Using of a graph of action dependencies for plans optimization. In *Proceedings of the International Multiconference on Computer Science and Information Technology, Wisa, Poland*, volume 2, pages 213–223, 2007.

[16] L. Chrpa. Generation of macro-operators via investigation of action dependencies in plans. *Knowledge Engineering Review*, 2009. to appear.

[17] L. Chrpa and R. Bartak. Looking for planning problems solvable in polynomial time via investigation of structures of action dependencies. In *Tenth Scandinavian Conference on Artificial Intelligence, SCAI 2008, Stockholm*, pages 175–180, 2008.

[18] L. Chrpa and R. Bartak. Towards getting domain knowledge: Plans analysis through investigation of actions dependencies. In *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, Coconut Grove, Florida, USA*, pages 531–536, 2008.

[19] L. Chrpa and R. Bartak. Reformulating planning problems by eliminating unpromising actions. In *SARA 2009: The Eighth Symposium on Abstraction, Reformulation and Approximation, Lake Arrowhead, CA, USA*, 2009. to appear.

[20] L. Chrpa, P. Surynek, and J. Vyskocil. Encoding of planning problems and their optimizations in linear logic. In *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Wurzburg, Germany, Revised Selected Papers*, volume 5437 of *LNCS*, pages 54–68, 2009.

[21] A. Coles, M. Fox, and A. Smith. Online identification of useful macro-actions for planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA*, pages 97–104, 2007.

[22] A. Coles and K. A. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research (JAIR)*, 28:119–156, 2007.

[23] G. Collins and L. Pryor. Planning under uncertainty: Some key issues. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montral, Qubec, Canada*, pages 1567–1575, 1995.

[24] R. Dave, S. Ganapathy, M. Fendley, and S. Narayanan. Dynamic path planning of ground robots and uninhabited aerial vehicles in human search and rescue missions. In *Proceedings of the 1st Indian International Conference on Artificial Intelligence, IICAI 2003, Hyderabad, India*, pages 315–322, 2003.

[25] C. Dawson and L. Siklóssy. The role of preprocessing in problem solving systems. In *IJCAI*, pages 465–471, 1977.

[26] C. Domshlak and J. Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research (JAIR)*, 30:565–620, 2007.

[27] K. Erol, J. A. Hendler, and D. S. Nau. Htn planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence. Seattle, WA, USA*, volume 1, pages 1123–1128, 1994.

[28] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76:75–88, 1995.

[29] R. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[30] B. R. Fox and K. G. Kempf. Planning, scheduling and uncertainty in the sequence of future events. In *UAI*, pages 395–402, 1986.

[31] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2:193–210, 1998.

[32] A. Gerevini and I. Serina. Lpg: A planner based on local search for planning graphs with action costs. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, Toulouse, France*, pages 13–22, 2002.

[33] M. Ghallab, E. Nationale, C. Aeronautiques, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld. Pddl the planning domain definition language. Technical report, 1998.

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

[35] O. Gimenez and A. Jonsson. On the hardeness of planning problems with simple causal graphs. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA*, pages 152–159, 2007.

[36] V. P. Gupta. Resources constrained modernization planning (recomp). In *IEEE International Conference on Communications, ICC '84, Links for the Future: Science, Systems & Services for Communications, Amsterdam, The Netherlands*, volume 1, pages 189–193, 1984.

[37] M. Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[38] M. Helmert. A planning heuristic based on causal graph analysis. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), Whistler, British Columbia, Canada*, pages 161–170, 2004.

[39] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006.

[40] M. Helmert. New complexity results for classical planning benchmarks. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK*, pages 52–62, 2006.

[41] J. Hoffmann. The metric-ff planning system: Translating ignoring delete lists to numeric state variables. *Journal of Artificial Intelligence Research (JAIR)*, 20:291–341, 2003.

[42] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001.

[43] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research (JAIR)*, 22:215–278, 2004.

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

[45] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989.

[46] P. Jonsson and C. Backstrom. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100:125–176, 1998.

[47] H. Kautz and B. Selman. Planning as satisfiability. In *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria*, pages 359–363, 1992.

[48] H. Kautz, B. Selman, and J. Hoffmann. Satplan: Planning as satisfiability. In *Proceedings of the fifth IPC*, 2006.

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

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

[51] J. Kvarnström and P. Doherty. Talplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):119–169, 2000.

[52] J. Kvarnström, F. Heintz, and P. Doherty. A temporal logic-based planning and execution monitoring system. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia*, pages 198–205, 2008.

[53] R. Mak and J. D. Walton. The collaborative information portal and nasa's mars rover mission. *IEEE Internet Computing*, 9(1):20–26, 2005.

[54] N. McCain and H. Turner. Causal theories of action and change. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, Providence, Rhode Island.*, pages 460–465, 1997.

[55] T. L. McCluskey. Combining weak learning heuristics in general problem solvers. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy*, pages 331–333, 1987.

[56] D. Mcdermott. The 1998 ai planning systems competition. *AI Magazine*, 21:35–55, 2000.

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

[58] S. Minton. Selectively generalizing plans for problem-solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA*, pages 596–599, 1985.

[59] S. Minton and J. G. Carbonell. Strategies for learning search control rules: An explanation-based approach. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy*, pages 228–235, 1987.

[60] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103:5–47, 1998.

[61] D. S. Nau, S. K. Gupta, and W. C. Regli. Ai planning versus manufacturing-operation planning: A case study. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montral, Qubec, Canada*, pages 1670–1676, 1995.

[62] D. S. Nau, S. J. J. Smith, and K. Erol. Control strategies in htn planning: Theory versus practice. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, Madison, Wisconsin, USA*, pages 1127–1133, 1998.

[63] M. A. H. Newton, J. Levine, M. Fox, and D. Long. Learning macro-actions for arbitrary planners and domains. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA*, pages 256–263, 2007.

[64] E. P. D. Pednault. Adl: Exploring the middle ground between strips and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89). Toronto, Canada*, pages 324–332, 1989.

[65] S. Richter and M. Westphal. The lama planner using landmark counting in heuristic search. In *Proceedings of the sixth IPC*, 2008.

[66] M. Roberts, A. E. Howe, B. Wilson, and M. desJardins. What makes planners predictable? In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia*, pages 288–295, 2008.

[67] F. Teichteil-Königsbuch and P. Fabiani. Autonomous search and rescue rotorcraft mission stochastic planning with generic dbns. In *Artificial Intelligence in Theory and Practice, IFIP 19th World Computer Congress, TC 12: IFIP AI 2006 Stream, Santiago, Chile*, pages 483–492, 2006.

[68] Z. Xing, Y. Chen, and W. Zhang. Maxplan: Optimal planning by decomposed satisfiability and backward reduction. In *Proceedings of the fifth IPC*, 2006.

[69] H. L. S. Younes and M. L. Littman. Ppddl1.0: An extension to pddl for expressing planning domains with probabilistic effects. Technical report, Carnegie Mellon University, 2004.

[70] N. L. Zhang and W. Lin. A model approximation scheme for planning in partially observable stochastic domains. *Journal of Artificial Intelligence Research (JAIR)*, 7:199–230, 1997.

[71] B. Ziebart, A. K. Dey, and J. A. Bagnell. Fast planning for dynamic preferences. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia*, pages 412–419, 2008.

Appendix A

Contents of CD

This thesis include an attached CD with an additional material in electronic form. Concretely, the contents of the CD is following:

Domains — Contains the original and reformulated planning domains and problems. The original planning domains and problems are taken from IPC website (http://ipc.icaps-conference.org)

Results — Contains the results in XLS files.

Sources — Contains the source codes for the generator of the macro-operators and the entanglement detector which are implemented in SWI-Prolog version 5.6.8. There are also domains and problems encoded in prolog which are required by the generator of the macro-operators and the entanglement detector.

Thesis — Contains the text of this thesis in PDF format.

The planners we used for the experiments are not listed on the CD, because it may cause the possible violation of their licenses.