Remodeling Planning Domains Using Macro Operators and Machine Learning

by

Maher Alhossaini

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2013 by Maher Alhossaini
Abstract

Remodeling Planning Domains Using Macro Operators and Machine Learning

Maher Alhossaini
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2013

The thesis of this dissertation is that automating domain remodeling in AI planning using macro operators and making remodeling more flexible and applicable can improve the planning performance and can enrich planning. In this dissertation, we present three novel ideas: (1) we present an instance-specific domain remodeling framework, (2) we recast the planning domain remodeling with macros as a parameter optimization problem, and (3) we combine two domain remodeling approaches in the instance-specific remodeling context.

In the instance-specific domain remodeling, we choose the best macro-augmented domain model for every incoming problem instance using a predictor that relies on previously solved problem instances to estimate the macros to be added the domain. Training the predictor is achieved off-line based on the observed relation between the instance features and the planner performance in the macro-augmented domain models. On-line, the predictor is used to find the best remodeling of the domain based on the problem instance features. Our empirical results over a number of standard benchmark planning domains demonstrate that our predictors can speed up the fixed-remodeling method that chooses the best set of macros by up to 2.5 times. The results also show that there is a large room for improving the performance using instance-specific over fixed remodeling approaches.

The second idea is recasting the domain remodeling with macros as a parameter optimization. We show that this remodeling approach can outperform standard macro learning tools, and that it can significantly speed up the domain evaluation preprocessing required to train the predictors in instance-specific remodeling, while maintaining similar performance.

The final idea applies macro addition and operator removal to the instance-specific domain remodeling. While maintaining an acceptable probability of solubility preservation, we build a predictor that adds macros and removes original operators based on the instance's features. The results show that this new remodeling significantly outperforms the macro-only fixed remodeling, and that it is better than the fixed domain models in a number of domains.
I would like to thank all the people who provided me with their advice and support. I want to thank my supervisor, Chris Beck, for his advice and support throughout the program. His insightful comments and his broad knowledge provided me with invaluable experience. I want to thank my committee members for their insightful comments and helpful suggestions. I want also to thank my colleagues. Thanks to Christian for his help in many occasions and for the useful discussions, and thanks to Tiago for the useful and interesting discussions.

I would like also to express my great thanks to my wife for her great support and help during my studies.

I dedicate this dissertation to my parents, who have helped and supported me before and throughout this long journey.
# Contents

1 Introduction .................................................. 1
  1.1 Motivations .................................................. 2
  1.2 Automatic and Flexible Domain Remodeling ............... 4
    1.2.1 Instance-Specific Framework ....................... 4
    1.2.2 Remodeling As Parameter Optimization ............ 5
  1.3 Contributions ............................................. 6
  1.4 Dissertation Outline .................................... 7

2 Background and Literature Review .......................... 9
  2.1 Classical Planning ......................................... 9
    2.1.1 Definitions ......................................... 10
    2.1.2 Solving The Classical Planning Problem .......... 11
    2.1.3 State-of-The-Art Planners ......................... 13
  2.2 Learning in Planning ...................................... 15
    2.2.1 Definitions ......................................... 15
    2.2.2 How To Learn ........................................ 16
    2.2.3 Learning in the Planning Literature ............... 17
  2.3 Macros in Planning ........................................ 18
    2.3.1 Definitions ......................................... 19
    2.3.2 Macro Processing .................................... 20
    2.3.3 Macro-FF ........................................... 21
    2.3.4 Wizard ............................................... 21
    2.3.5 Other Macro Work .................................... 22
  2.4 Reformulation in Planning ................................ 24
    2.4.1 Representation-Based Reformulations .............. 25
    2.4.2 Reformulation to Non-planning Problems .......... 25
    2.4.3 Reformulation By AddingAtoms ..................... 26
    2.4.4 Reformulation By Removing Atoms (Abstraction) ... 26
    2.4.5 Reformulation By Removing Operators .............. 27
    2.4.6 Reformulation By Adding Macros and Removing Operators ............ 29
  2.5 Algorithm Tuning ......................................... 29
    2.5.1 ParamILS ........................................... 29
  2.6 Summary .................................................. 31
3 Learning to Choose Instance-Specific Macro Operators

3.1 Introduction .......................................................... 32
3.2 Motivations ............................................................ 32
3.3 Approach .............................................................. 33
  3.3.1 Definitions ..................................................... 34
  3.3.2 System Details ................................................ 35
  3.3.3 Macro Sources ................................................ 39
  3.3.4 Learning and Features ..................................... 40
3.4 Claims and Research Question ................................. 43
  3.4.1 Feasibility Experiment ...................................... 44
  3.4.2 Hypothesis .................................................... 45
3.5 Experimental Design ............................................. 45
  3.5.1 Settings ........................................................ 45
  3.5.2 Tests ........................................................... 49
3.6 Experimental Results ............................................ 50
  3.6.1 Results Summary ............................................. 52
3.7 Analysis and Discussion ........................................ 52
  3.7.1 Analysis ........................................................ 52
  3.7.2 Discussion .................................................... 59
3.8 Summary ............................................................. 62

4 Macro Subset Learning As Parameter Configuration

4.1 Introduction .......................................................... 63
4.2 Motivations ............................................................ 64
4.3 ParamILS .............................................................. 65
4.4 Approach .............................................................. 65
  4.4.1 Interface with ParamILS ..................................... 66
  4.4.2 Definition .................................................... 67
  4.4.3 System Details ................................................ 67
4.5 Claims and Research Questions ................................. 69
4.6 Experimental Design ............................................. 70
  4.6.1 Settings ........................................................ 70
  4.6.2 Tests ........................................................... 72
4.7 Experimental Results ............................................ 73
  4.7.1 Fixed Macros Test ........................................... 73
  4.7.2 Instance-Specific Macros Test ............................ 74
4.8 Analysis and Discussion ........................................ 75
  4.8.1 Analysis ........................................................ 75
  4.8.2 Discussion .................................................... 77
4.9 Summary ............................................................. 79
# Remodeling With Macros Addition and Operators Deletion

## Introduction

## Motivations

## Definitions

## Approach

- **5.4.1 Approximating Coverage**
- **5.4.2 System Details**
- **5.4.3 Feature Selection**

## Claims and Research Questions

## Experimental Design

- **5.6.1 Settings**
- **5.6.2 Tests**

## Experimental Results

## Analysis and Discussion

- **5.8.1 Analysis**
- **5.8.2 Discussion**

## Summary

# Conclusions and Future Work

## Contributions

- **6.1.1 Major Contributions**
- **6.1.2 Other Contributions**

## Future Work

- **6.2.1 Individual Macro Learning As Parameter Optimization**
- **6.2.2 Feature Selection For Clustering**
- **6.2.3 Remodeling And Planner Tuning As Parameter Optimization**
- **6.2.4 Operator-Remodeling As Parameter Optimization**
- **6.2.5 A Portfolio Of Reformulated Domains**

## Conclusion

# Bibliography

# Domains Used

## Logistics

## Blocksworld (4 operators)

## Pipesworld (no tankage)

## Mystery

## Mprime

## Freecell

# Manual Macro Operators

## Logistics

## Blocksworld (4-operators)
B.3 Pipesworld (no-tankage) ................................. 118
  B.3.1 PDDL Definition ..................................... 118
B.4 Mystery ..................................................... 122
  B.4.1 PDDL Definition ..................................... 122
B.5 Mprime ..................................................... 123
  B.5.1 PDDL Definition ..................................... 124
B.6 Freecell .................................................... 125
  B.6.1 PDDL Definition ..................................... 126

C Generators .................................................. 128
  C.1 Pipesworld (no-tankage) ................................. 128
    C.1.1 Pipes ................................................. 128
    C.1.2 Batches .............................................. 129
    C.1.3 Goals ................................................. 129
    C.1.4 Code ................................................ 129
  C.2 Other Domains ......................................... 129

D Features Used ............................................. 130
  D.1 Freecell Domain Features ................................. 130
  D.2 State Statistics Features ................................. 130
  D.3 Taxonomic Syntax Features ............................... 131
    D.3.1 Macro-addition Experiments .......................... 131
    D.3.2 Operator Remodeling Experiment ..................... 132
Chapter 1

Introduction

Although choosing the right model to represent a problem is essential for solving it efficiently, problem remodeling is still far from being an integral part of the AI planning process. Modeling planning domains is not an easy task. Finding a model that is unlikely to have to be modified, either due to the discovery of a flaw in the design or a performance inefficiency of the planners, is difficult. In general, state-of-the-art planning does not utilize the full power of remodeling to speed up planning. For example, even planners equipped with remodeling components, such as Macro-FF [11], rely on finding a single, fixed remodeling of the domain that is used to solve all instances of the domain.

This work is inspired by Simon’s work on modeling the “ill structure problems” [80]. Simon suggests that most real-life problems are difficult or impossible to model correctly (ill structured), and hence require integrating problem solving and problem remodeling into one unit that alternates between solving and modeling. In this dissertation, we present novel approaches that follow a similar viewpoint: they view remodeling as part of the planning process. Our approaches incorporate domain remodeling into the planning of each problem instance, and make the remodeling more general and more accessible.

The thesis of this dissertation is that automating the remodeling of planning domains, and achieving the remodeling using a flexible framework, will enrich state-of-the-art planning and will improve the planning speed. Achieving remodeling automatically with every problem instance is one step towards the vision of knowledge-based problem solving suggested by Simon [80]. Flexibility of remodeling allows us to combine different domain modeling approaches and/or use different tools to achieve the remodeling. Flexibility in remodeling is desirable since it makes remodeling both more useful and applicable compared to fixed, single remodeling approaches.

The goals of this work are: to make domain remodeling more automatic and to make it more flexible and accessible in the planning process. To achieve these goals we suggest three general ideas: (1) we introduce a remodeling framework that integrates remodeling and the problem solving in an online, instance-specific component, (2) we combine different remodeling approaches in the new framework, and (3) we generalize remodeling by recasting it as an optimization problem. To implement each of these ideas, we present three approaches:

1. A remodeling approach that learns to add macro operators to the domain definition per instance, based on the problem instance features.

2. A remodeling approach that learns to add macros and remove original operators from the domain definition per instance and takes the solubility-preserving into account.
Figure 1.1: Goals of the thesis, and their relation to the ideas and the actual approaches we developed.

3. A reduction approach that recasts the remodeling problem using macros as a parameter optimization problem.

Figure 1.1 shows the goals, general ideas, and approaches we present in this dissertation and their relation to each other. In Section 1.2, we introduce the approaches in more detail.

We conduct a number of experiments to test the feasibility and effectiveness of our approaches. The empirical results over a number of standard benchmark planning domains demonstrate that our remodeling approaches can outperform the original domain, the best-on-average remodeling and the state-of-the-art action-based remodeling methods. The results also show that there is a significant room for improving the performance, sometimes by orders of magnitude, using instance specific remodeling over fixed-domain remodeling. The new approaches are also shown to be easily extendible to other kinds of remodeling.

In the following section we describe our motivations. Next, we describe the two major goals we are trying to achieve and the approaches we developed to achieve them. Then, we explain the contributions of this work. Finally, we describe the contents of the following chapters in this dissertation.

1.1 Motivations

Our work is motivated by the original ideas of the work of Simon [80] on designing a solver for “ill structured problems”, that is similar to the general problem solver (GPS) for well structured problems. Simon proposed a dynamic remodeling-solving process that combines the GPS with a retrieval system, and maintains a knowledge base that is used to modify the problem model continually. The retrieval system relies on recognizing the problem’s structure from previously encountered problems, and retrieves the appropriate knowledge to solve them. It achieves this by considering different modeling alternatives in its knowledge base.

Our approaches are inspired by this dynamic approach. We use machine learning to automatically recognize and retrieve the planning domain models appropriate for the problem instance based on the problem’s features, combining different kinds of domain remodeling, and using a general approach to filter them.
The motivations of this dissertation can be summarized as follows:

1. **Combining Remodeling and Solving in Planning** – The solving process includes two steps: the domain design and applying the solving technique. Simon [80] suggests combining them into one process that alternates between modifying the design and solving. The domain model design is a significant part of the planning process that has not received as much attention as the search techniques. For example, it was found that some domains can be solved significantly faster if they were put into different form, as in the blocksworld domain’s two versions [45]. Therefore, it is important to find the best domain remodeling before solving.

2. **Retrieval Processing** – Not only should remodeling accompany planning, but also it should be used to solve every incoming planning instance. Many real life problems are regarded as “ill structured”: they are defined based on what should *not* be done rather on what should be done [80]. So, there is not one final definition of a problem domain that we know should be used to solve instances of the problem. We often find the domain definition either too specific (captures too many details) where it cannot represent all the problem instances as we want, or it can be too general, which makes the search costly. This realization is not because we did not study the requirements well, but because we cannot know them well until after we solve a number of real instances. Simon suggests adding to the problem solver a retrieval system, “which continually modifies the problem space by evoking from long-term memory new constraints, new subgoals and new generators for design alternatives.” [80, p. 191]. Our first step towards Simon’s retrieval system that brings remodeling into solving is a dynamic domain remodeling preprocessor: a remodeling that changes the planning domain for every new problem instance.

3. **Contextual Remodeling** – The remodeling system should consider only relevant information. Real world domains are often more diverse than the domains presented in the International Planning Competitions (IPC): they contain many constants, actions, and facts that are not relevant to every problem instance. From a general problem solver perspective, planning needs to be more flexible to work in different environments even if they contain much irrelevant information. In addition to (2), we know that the knowledge used to remodel the domain for every instance should not be an accumulation of what was learned so far, but a selection of relevant information that is based on features: “We can also view this retrieval system as a recognition system that attends to features in the current problem space and in external memory (e.g. models and drawings), and, recognizing features as familiar, evokes relevant information from memory which it adds to the problem space (or substitutes for other information currently in the problem space).”[80, p. 192]. Automating the extraction and usage of relevant and useful domain knowledge for each instance is not an easy task: we generally cannot know which piece of knowledge is useful except by experience. Therefore, the use of machine learning to automate knowledge acquisition and usage is essential.

4. **Expanding the Limits Of Remodeling: Hybrid Approaches** – It is useful to combine orthogonal remodeling approaches. Remodeling tools for planning domains often follow a single approach. For example, abstraction, domain augmentation with macro operators, or removing redundant actions, have been used individually to achieve domain remodeling. However, to design a system as powerful as the solver suggested by Simon [80], we perhaps need to adopt orthogonal
remodeling approaches. Some approaches, such as macro addition and operator removal, are complementary: they fix each other’s problems. Therefore, such approaches should be combined and unwanted side effects should be overcome.

5. **Reduction Advantages** – Reducing one problem to another problem and solving using a solver for the latter can simplify the solving and benefit from the advantages and advances of the latter’s solvers. Instead of writing a new algorithm to solve our problem, we can recast the problem into a generalized form that is solvable by a generic solver, and use that solver. Recasting the macro-based remodeling as a domain parameter optimization problem is one example of such reduction. By this recasting, we formalize the domain model as a parameter vector that can be (1) expanded by adding more domain characteristics to it, or (2) improved by using a different parameter optimization tool.

### 1.2 Automatic and Flexible Domain Remodeling

In this dissertation, we introduce two orthogonal ideas to remodel a planning domain: the instance-specific remodeling framework, and remodeling as optimization of the domain parameters. In this section we explain the approaches we developed to implement each of these two ideas. Note that we also use a third idea that is found in the literature but has not been applied before in similar frameworks: combining remodeling approaches.

To implement the idea of instance-specific remodeling framework, we present two approaches, the first is solubility-preserving and the second is not. The non-solubility-preserving approach relies also on the idea of combining two remodeling approaches. To implement the remodeling as parameter optimization idea, we recast the problem of domain remodeling using macro operators as a parameter optimization problem and use an external parameter optimization tool.

#### 1.2.1 Instance-Specific Framework

Traditionally, macro operators have been added to the planning domain definition with the basic operators to speed up planning [34, 63, 11, 69]. The macros improve the search because they act as shortcuts...
to useful states. Figure 1.2 shows the traditional framework that can be found in the literature. Offline, the domain definition is provided to a macro learning tool. The macro tool generates and filters macro operators and adds them to the domain definition to produce a new macro-augmented domain. Online, the macro-augmented domain is used, instead of the original domain, to solve new problem instance.

In this dissertation, we introduce an instance-specific framework to remodel the planning domain with macros. In the instance-specific remodeling we learn a relation between the problem instances and the best way to remodel them, and then we use the learned relation to predict the best remodeling for future instances. Similarly it has two phases: offline and online as shown in Figure 1.3. The offline phase performs the learning, and the online phase performs the prediction and the remodeling based on the prediction. The predictor is fast, and is consulted for each instance to be solved.

We present two instance specific remodeling approaches: macro remodeling and operator remodeling. The macro remodeling only adds macros to the domain and uses the predictor to estimate the best macro set for each problem instance. Using an existing set of macro operators, we dynamically add subsets of macros to the domain definition based on the experience about the relationship between the problem instance’s features and the performance of the macro subset on the instance.

Operator remodeling is not solubility-preserving. This remodeling combines the two kinds of remodeling: macro addition and operator removal, which can significantly speed up planning compared to the macro-only remodeling. It may allow us to replace some domains with completely new ones for each instance. However, since we remove operators, the remodeling predictors may not be solubility-preserving: some solvable instances can become unsolvable. Therefore, we try to maintain a lower bound on the number of solvable instances that became unsolvable by filtering out the remodelings.

1.2.2 Remodeling As Parameter Optimization

An approach by which we make the remodeling more flexible and accessible in planning is to recast the remodeling components as parameters to the domain that can be optimized. In our approach, we represent the macros as binary parameters, and we see their presence in the domain as assignment to the parameters. Therefore, finding the best parameter assignment means finding the best remodeling. We use a general parameter tuning tool to perform the remodeling.
Figure 1.4: Domain remodeling by a parameter tuning: offline, the tuner changes the domain, the
domain is fed to the planner, and the planner’s results are fed back to the tuner to change the domain
further. The tuner then returns optimized parameters that represent the domain remodeling

Figure 1.4 shows that main idea of remodeling as domain parameters tuning. Offline, the tuning tool
remodels the domain by modifying the domain parameter assignment, and using the modified domain
to run the planner on some problem instances. Then, the planner’s runtime and output are fed back to
the tuning tool to make a decision about the domain modification and try another remodeling.

This approach is orthogonal to the instance-specific approach. We use this approach, offline, to build
a predictor that can remodel the domain online in the macro-addition remodeling.

Remodeling under this reduction is a more general and flexible technique because it is achievable by
a generic algorithm parameter tuner, and because it is expandable to include other kinds of parameters.
The advantages of such reduction include:

1. Any generic parameter tuner can be used, and hence all advances in that field can be utilized.

2. Other kinds of parameters of the domain can be included with the macros, which makes remodeling
   more general. For example, we can consider the original domain operators as parameters.

3. Any other kinds of parameters of the planner can also be included, which can make the planning
   process more general. For example, the set of domain operators and the search technique or
   heuristic can be optimized simultaneously to improve the performance.

1.3 Contributions

This work has the following main contributions:

1. We introduce a learning approach to choose macro operators for remodeling planning domains
   per problem instance and show that it can improve planning speed significantly compared to the
   fixed-macro-set remodeling approach.

2. We introduce a novel way to represent the macros in the domain: we cast them as domain parame-
ters that can be optimized. Using a generic parameter optimization tool to perform the remodeling,
we approximate the exhaustive approach we followed to evaluate the macros, and save off-line pre-
processing time. We show that the performance of resulting remodeled domain is not harmed much 
by this approximation compared to the exhaustive approach, while the new approach requires 
significantly less preprocessing time.

3. We remodel the planning domains by removing basic operators as well as adding macros in an 
instance-specific context. This new remodeling is more powerful, since it significantly reduces the 
search branching factor, which can improve the performance. However, operator removal harms 
solubility-preserving. We present a method to empirically maintain a high probability of solubility-
preserving on the problem instances while learning to remodel the domains, and we show that our 
approach can improve the planning speed significantly compared to fixed-remodeling schemes while 
maintaining a high probability of solubility-preserving.

1.4 Dissertation Outline

This dissertation contains six chapters. Here, we describe the contents of each of the following five 
chapters.

Chapter 2 presents the literature review, background, and the basic definitions needed for this disser-
tation. First, the chapter discusses the field of classical planning, its solving methods, and state-of-the-art 
planners. Next, we review the main components and applications of machine learning in planning. Then 
we discuss macro learning in planning, and review well-known tools and works in that field. Then 
we discuss different kinds of domain remodeling, and review the work in each. Finally, we review the 
automated parameter optimization tool that we use in the dissertation.

Chapter 3 presents our method of learning macro subsets in an instance-specific context. We describe 
the learning method and the algorithms used to select the features and perform the training. Then 
we describe the experiment we conducted to test the effectiveness of the method compared to the 
non-learning scheme using different planners and macro sources. We demonstrate that our resulting 
predictors can select macros that can significantly outperform the best expected fixed macros in a 
number of domains. We further analyze the results and use a domain analysis tool to explain the 
different observations we find.

In Chapter 4 we cast macro learning as parameter configuration in both the fixed-macros and instance-
specific-macros contexts. We explain the new learning method in both contexts, and we describe the 
experiments we conducted and the results. We demonstrate that the macros found in the fixed remodeling 
context are comparable to two state-of-the-art macro acquisition tools. We also demonstrate that the 
parameter configuration method can significantly reduce the preprocessing time required for the training 
in the instance-specific context, while often maintaining the good performance.

In Chapter 5 we present a new method to remodel the planning domain by adding macro operators 
and removing original operators in both fixed and instance-specific contexts. We describe our method 
and how we deal with the lack of solubility-preserving that results from removing the operators. Then, 
we describe the experiment we conducted to test the effectiveness of the method compared to the non-
learning scheme using different macro sources. We demonstrate that the operator removal can outperform 
the macro-addition-only technique in the fixed remodeling context, and we show that our new predictors 
can select domain models that are significantly better than the best expected remodeling in a number 
of domains.
Finally, Chapter 6 contains the conclusion. We present a summary of the work. Then, we present the contributions in light of the results and the analysis. Finally, we present the future work.

Appendix A contains the description of the benchmark domains used in the experiments. Appendix B contains a description of the manually constructed macro operators used in some of the experiments. Appendix C contains a description of the problem instance generators. Appendix D describes the features we used in the experiments.
Chapter 2

Background and Literature Review

This chapter reviews the literature directly related to the domain remodeling methods we propose, along with the required background, including classical planning, learning in planning, macro operators, domain remodeling, and parameter optimization.

First, we discuss classical planning, considered one of the hard problems in Artificial Intelligence [29]. While there are many generic techniques for improving planning speed, such as heuristic search and problem reformulation to other well known problems [49, 54, 23], significantly more efficient planning can be achieved by exploiting specific knowledge about the planning problem [3, 24, 58]. Our focus in this dissertation will be on one form of knowledge, macro operators.

Macros are sequences of planning operators representing useful shortcuts in the search space [11, 20, 67]. We learn to use them to remodel the planning problem as part of solving each problem instance. In the second section of this chapter, we discuss macro acquisition and usage, then we review relevant macro work.

To automate the per-instance remodeling, we use supervised machine learning. Learning has been used to automate the acquisition and use of different kinds of knowledge in planning, such as policies and macro actions [90, 67, 1, 2]. In the third section of this chapter we discuss learning in planning, and we review the related literature.

Macro-addition remodeling is only one form of domain remodeling. In this dissertation, we generalize macro remodeling by removing some original problem operators as well as adding macros. We discuss the kinds of remodeling and review related works.

In this dissertation, we also recast domain remodeling with macros as an optimization problem. Like remodeling, the optimization is achieved in an offline preprocessing step that precedes the solving. In the last section of this chapter, we discuss the tool we use to perform the domain optimization: ParamILS [50].

2.1 Classical Planning

Planning is the organization of a system’s actions to change the world’s state to achieve some pre-stated objectives [40]. One motivation for the study of planning is the need for an automated tool to model and organize the access to resources using pre-defined actions. Another is its application to computational intelligence: planning is reasoning about actions, which is required for an automated intelligent agent
A planning problem is represented by a pair \((\Sigma, \Pi)\) [40]. Generally, \(\Sigma\) is a state transition system that represents the environment and describes the possible world states and actions. The problem to be solved within this environment is represented by \(\Pi\), which defines the initial situation of the world and the objective to be achieved. To plan is to fulfill the objectives for the agent, who is the actor in the environment. There are many types of planning depending on the assumptions made about the environment \((\Sigma)\) and the problem \((\Pi)\). We focus here on classical planning.

In classical planning, a number of assumptions are made about the world [40]. For example: the number of states is finite; the world does not change except by the agent’s actions; the agent knows the actions’ effects with certainty before doing the action; the states of the world are fully observable to the agent; the actions are executed instantaneously; and finally, the goals are expressed as explicit facts, and there are no constraints on the way actions are sequenced except by action definition. In classical planning we look for a sequence of actions that achieves the goals.

There are two flavors of the classical planning problem: plan existence and plan optimality. The former is concerned with finding any solution plan that achieves the goal. The latter is concerned with finding a solution plan that achieves the goal and achieves the optimality criterion (which is often to minimize the length of the plan). Classical planning is a PSPACE-complete problem [30, 13] for both flavors of the problem, although some planning problem instances can be solved fairly quickly. Our focus in this dissertation will be on the plan existence problem.

Representing the problem is important for solving it. One well-known language in classical planning is STRIPS [33, 71]. Conventionally, a STRIPS planning problem is represented by a planning domain \((D)\) and a planning instance \((i)\) of that domain. The planning domain contains a description of the states of the environment \((\Sigma)\) and their transitions. The planning instance is a description of a particular problem: it describes the facts in the initial state and the goal [71]. The following sub-section provides a formal definition of STRIPS planning.

### 2.1.1 Definitions

In classical planning we define the following basic concepts based on the STRIPS [33] formalism:

- **A predicate**, \(p\), is a tuple \((v_1, v_2, ..., v_n)\), where \(p\) is the predicate name, and \(v_1, ..., v_n\) are variables.

- **A substitution**, \(\theta\), is a function that maps variables to constants.

- **An atom**, \(l\), is an instantiation to a predicate \(p\); i.e., it is the result of applying a substitution \(\theta\) to the variables of \(p\): \(\theta(p) = l\). The substitution can also be applied to a set of predicates \(\Psi\) to produces a set of atoms \(\Lambda\). So, \(\theta(\Psi) = \Lambda\) if \(\forall p \in \Psi : \theta(p) \in \Lambda\).

- **A state**, \(s\), is a set of atoms. It represents the conjunction of facts that describes a situation. We assume a closed world, so the objects and the atom set are finite and known. Thus, we represent the truth of all atoms in the state: the atom’s presence in the state means it is true and its absence means it is false.

- **An operator**, \(o\), is a tuple \((pre_o, add_o, del_o)\), where \(o\) is the operator name, and \(pre_o, add_o, del_o\) are sets of predicates. The same variable names may be used in different predicates of these sets.
An action, \( a \), is a tuple \((\text{pre}_a, \text{add}_a, \text{del}_a)\), where \( a \) is the action name, and \( \text{pre}_a \), \( \text{add}_a \), and \( \text{del}_a \) are sets of atoms. An action is an instantiation of an operator. So \( \exists o = (\text{pre}_a, \text{add}_a, \text{del}_a), \theta: \theta(o) = a \) iff \( \theta(\text{pre}_a) = \text{pre}_a, \theta(\text{add}_a) = \text{add}_a \), and \( \theta(\text{del}_a) = \text{del}_a \). Actions represent instantiated transitions between states. So, \( \text{pre}_a \) is the precondition atom set that must be true in a state when applying the action; \( \text{add}_a \) is the set of added atoms, called the positive effects; and \( \text{del}_a \) is the set of deleted atoms, or the negative effects.

**An Action** \( a = (\text{pre}_a, \text{add}_a, \text{del}_a) \) is Applicable in State \( s \) if \( \text{pre}_a \subseteq s \). We define a function \( a \) to represent the state transition as follows:

\[
\text{If } a \text{ is applicable in } s, \text{ then the application of } a \text{ to } s \text{ is a state } a(s) = s \setminus \text{del}_a \cup \text{add}_a.
\]

**An Operator** \( o = (\text{pre}_o, \text{add}_o, \text{del}_o) \) is Applicable in state \( s \) if one of its instantiations (actions) \( \theta(o) = (\theta(\text{pre}_o), \theta(\text{add}_o), \theta(\text{del}_o)) \) is applicable in \( s \), i.e., \( \text{pre}_{\theta(o)} \subseteq s \).

A Plan \( \pi \) is a sequence of actions \((a_1, a_2, \ldots, a_n)\).

A Domain \( D \) is composed of a set of predicates \( \Psi \) and a set of operators \( O \) over \( \Psi \): \( D = (\Psi, O) \).

A Problem Instance \( i \) from domain \( D = (\Psi, O) \) is a pair \((s_0, g)\), where \( s_0 \) is the initial state, and \( g \) is a set of atoms that represent the goal. The atoms are instantiated predicates from \( \Psi \). The constants used in the substitution that instantiated the initial state and goal are the Objects of the problem instance. There are often many problem instances from the domain. A complete planning problem statement includes both the domain and the problem instance.

A Plan \( \pi = (a_1, \ldots, a_n) \) is a Solution to Instance \( i = (s_0, g) \) if:

1. \( a_1 \) is applicable in \( s_0 \) and \( a_j \) is applicable in \( a_{j-1}(a_{j-2}(\ldots a_1(s_0))\ldots) \) for \( 2 \leq j \leq n \).
2. \( g \subseteq a_n(a_{n-1}(\ldots a_1(s_0))\ldots) \)

2.1.2 Solving The Classical Planning Problem

Solving a planning problem instance means finding a solution plan to that instance. This is generally achieved by a planner that searches in a directed graph called the search space, which represents the problem. The planner begins from an initial node in the graph, iteratively expands and tests successor nodes for the goal criteria, and builds a plan. When a node that achieves the goals is found, the problem is declared solvable and the solution is returned. If all nodes are proved not to contain the desired goals, then the problem is declared unsolvable.

There are many types of search spaces. For example, there is the state space, which is the space of states as defined above, and the plan space, which is the space of partial-order plans (see Section 2.1.3.) Also, if the planning problem was reduced to another problem (e.g. propositional satisfiability (SAT) in the BLACKBOX planner [54], or constraint satisfaction problem (CSP) in the GP-CSP planner [23]) the search space becomes that of the problem to which the planning problem was reduced.

There are many effective search algorithms: forward or backward chaining search, and a local or complete search [40]. However, they are impractical without strong guidance, since the search space is large. Generally, there are two methods that guide the search: general heuristics and techniques that depend on domain-specific knowledge [77, 40].
The general heuristics are designed to guide the search without any specific knowledge about the domain, so they are domain-independent. Domain-specific knowledge techniques are designed to use knowledge about the domain to speed up the search [40].

**Heuristics**

Heuristic search relies on building a *heuristic evaluation function*, \( h \), which is a mapping from the state space to the positive real numbers (plus 0). It is an estimate of the state’s distance to the goal. *Admissible* heuristics never overestimate the distance to the nearest goal [77], and they are guaranteed to find optimal solutions (in terms of the plan length) using A* search [77].

Usually, heuristics are built based on a relaxed version of the underlying problem: the solution of the relaxed problem is used to estimate the distance from the state to the goal. One example of an important heuristic is the relaxed planning graph (RPG) heuristic used in the FF planner [49] \( (h_{RPG}) \). The \( h_{RPG} \) heuristic predicts the distance from the state to the goal using the planning graph, a structure presented in the GRAPHPLAN planner [8]. The \( h_{RPG} \) heuristic builds a relaxed version of planning graph where the delete effects of the actions are removed.

The planning graph consists of alternating layers of fact nodes and action nodes. It represents all possible actions that can be taken in each action layer to reach the last layer, which contains the goals. GRAPHPLAN finds parallel plans by choosing non-contradicting actions in each layer. A relaxed version of the planning graph omits the delete effect of the actions, changing the problem to an easier one. \( h_{RPG} \) finds a parallel plan in the relaxed planning graph originated at the current state, and returns the total number of actions of that plan.

The \( h_{RPG} \) heuristic is not always accurate, since it can return a number of actions that is different from what is necessary to achieve the goal. The \( h_{RPG} \) heuristic can either underestimate or overestimate the real distance, making it an inadmissible heuristic. It is inadmissible because it does not account for the real effect of the complex interaction between actions, which may result in a shorter plan than the relaxed plan. However, the RPG heuristic was shown to be very effective in guiding the search [49].

**Techniques that Depend on Domain-specific Knowledge**

The techniques that rely on extracting and exploiting knowledge from the domain have been used to build problem models and algorithms to solve problems efficiently [40]. It is essential to automate the extraction of useful knowledge. Once automatic knowledge extraction tools are built, they can be used to extract domain-specific knowledge for all domains, while the alternative approach, manual knowledge extraction, requires analyzing every incoming domain. For example, tools that automatically extract macro actions have been successful in improving planning speed [67, 11].

One example of useful domain knowledge is the use of search control rules. These rules can guide the search by changing the default choices of the planner based on previous experience of the planner. Planners tailored to use manually-extracted domain-specific control rules, such as TLplan [3] and TALplanner [24], have shown a significant planning speed-up in the Third International Planning Competition (IPC) [58]. Automatically extracted search control rules presented by Borrajo and Veloso [10], for example, were built using inductive learning techniques, and were used in the PRODIGY planner [65]. They also improved the planning speed.

Another example of domain knowledge is case-based planning. Cases are traces of previously solved problems that are saved and later used in search states where the state features are matched with the
An important example of domain knowledge is macro operators. Macro operators are sequences of operators that are linked by one or more arguments, and act like one operator. Macros are useful because they can reach useful states in the search tree quickly. They are also convenient since they can be incorporated easily in the planning domains: we can remodel the domain by adding them as basic operators. Examples of recent planners that extract and use macros are Marvin [20] and Macro-FF [11]. A detailed discussion of macros is given in Section 2.3.

Although some domain-specific knowledge techniques are useful, using too much domain knowledge can slow down the search, in a form of the utility problem [64]. The utility problem pertains to the trade-off between the amount of knowledge used or added to the domain and the search speed. For example, adding too many macro operators to the domain definition increases the number of actions to examine (search branching factor), which can slow down the search significantly. Thus, a careful choice of knowledge is essential.

Next, we review state-of-the-art planning, and we discuss two planners we use in this work: the Fast Forward planner [49] and the VHPOP [91] planner.

2.1.3 State-of-The-Art Planners

Planning research has evolved rapidly in recent years. Many well-known planners were developed based on efficient search techniques. Examples of such planners include: HSP-2 [9], FF [49], SatPlan [53], LPG [39], VHPOP [91], SGPlan [14], FD [42], and LAMA[74]. Below, we discuss FF and VHPOP: two popular state-of-the-art planners that we will later use in our experiments.

**FF**

The Fast Forward (FF) planner [49] is a deterministic, state-space-based, forward-search planner. It was a winner in the AIPS 2000 planning competition [4], and has been widely used in the literature. FF runs in two main phases: first, the Enforced Hill Climbing (EHC) search phase, and second, when EHC fails to find a solution, the Best-first search (BFS) phase. EHC is a local search technique used to search quickly for a solution without backtracking, while BFS is an exhaustive search that uses a greedy best first search [77] algorithm. As discussed in Section 2.1.2, FF uses the $h_{RPG}$ heuristic in its search.

The EHC greedily explores better states (with smaller $h_{RPG}$ values) until it falls into a **plateau**, where neighboring states have the same heuristic value as the current state. At that point EHC will try to escape to a better state by applying a breadth-first search. EHC stops when it reaches a goal state, or when it is stuck in a **dead-end** state $s$, from which the goal is unreachable, and $h_{RPG}(s) = \infty$.

An important enhancement in EHC is that it only considers **helpful actions** for state transition. Helpful actions are found while calculating the $h_{RPG}$ value of the state: in addition to finding the number of actions needed to reach the goal in the RPG, the heuristic saves the actions that are the first to add the goal atoms in the RPG or the first to add preconditions of such actions, recursively, and appear in the first action layer. Only those actions are considered for transitioning from that state. Although using the helpful actions is an incomplete approach in general, since it can omit some valid actions, empirical results show that constraining the search to helpful actions improves the performance [49].

However, EHC has disadvantages. It can get stuck in a plateau for a long time, or even worse, it
can fall into a local minimum, which is a non-goal state whose heuristic evaluation is the least among its neighboring states [44]. Local minima are evidence of the heuristic inaccuracy in the domain, since they have the least heuristic value although they are not goal states.

The BFS search is a complete systematic search. In BFS, the planning is achieved by systematically testing and expanding the states in the search queue. BFS uses the RPG heuristic to sort the states in the search queue, and expands only those with the lowest heuristic value. It is a greedy best-first search [77] with a state evaluation function \( f(s) = h_{RPG}(s) \).

FF has many advantages that made it the winner of the AIPS 2000 planning competition. It is still considered one of the best for solving STRIPS domains, especially those that do not contain local minima like logistics, miconic, and ferry. The strengths of FF are that: (1) the RPG heuristic is relatively accurate and quick to compute, and (2) the helpful actions are extremely useful in guiding the search [49]. However, FF also has some known weaknesses, such as: (1) EHC can get stuck in local minima, and (2) in some domains, FF can fall into dead-end situations, needing to abandon the local search and backtrack; this is done without using an effective method to handle backtracking, as FF only uses a normal greedy best-first search [49].

**VHPOP**

Versatile Heuristic Partial Order Planner (VHPOP) [91] is a recent POCL (Partial-Order Causal-Link) planner: it searches in a space of partial-order-plan states. VHPOP uses a combination of state-space planners’ heuristics and other POCL planners’ heuristics during the search. VHPOP was shown to be competitive to POCL planners at the third International Planning Competition [58].

In POCL planning, the partial-order-plan state is defined as a tuple: \( (A, L, O, B) \), where \( A \) is the set of actions, \( L \) is the set of causal links, \( O \) is the set of ordering constraints of actions in \( A \), and \( B \) is the set of binding constraints on the action variables [91]. A causal link from action \( a_1 \) to action \( a_2 \) using a literal \( l \) is a commitment by the planner to use \( l \), as an effect of \( a_1 \), to fulfill the precondition of \( a_2 \). An ordering constraint between a pair of actions \( (a_1, a_2) \) is simply a commitment by the planner to make \( a_1 \) precede \( a_2 \) in the plan, while a binding refers to variables’ assignments or unification.

The goal of POCL planning is to find a partially ordered plan that is flaw-free. In a plan, the flaw is either an open condition, which is an unfulfilled precondition of one of the plan’s actions, or a threatened causal link, which is a causal link that can be broken, for example, by putting one of the plan’s actions that deletes the literal of the causal link between the link’s actions.

Algorithm 1 describes the general search of a POCL planner [91] used by VHPOP. The planner generates the initial state, which is a partial empty plan containing two artificial actions: one that has the goal atoms \( g \) as preconditions, and one that has the initial state atoms as effects. The planner maintains a fringe \( P \) of unexplored partial plans. It selects a plan and removes it from \( P \). If the selected plan has no flaws, the algorithm stops and returns it. Otherwise, it selects a flaw of the plan and fixes it. There can be many different fixes to a flaw. For example: the action that threatens a causal link can be put before or after the causal link actions. Therefore, all fixed versions of the plan will be added to \( P \). The algorithm stops when \( P \) becomes empty.

VHPOP depends on two components: the plan selector and the flaw selector. The plan selection efficiency depends on the number of remaining actions needed to reach the goal, and the effort needed to add each of the remaining actions and fix its flaws. VHPOP tries to estimate the number of remaining actions using a variation of the additive heuristic used in the HSP planner [9] (a state-space planner),
and tries to subtract some positive interaction between the actions. The expected effort needed to fix the flaws is also included in the plan selection heuristic. For the flaw selection, VHPOP combines a number of strategies from older POCL planners like UCPOP [72] and SNLP [62]. VHPOP also introduces new strategies such as the conflict-driven selection, which prefers flaws that cause conflicts to the plan in early stages of the search, hopefully speeding up the search.

Algorithm 1: General POCL algorithm.

begin
1 $P \leftarrow \text{InitialPartialPlan}(s_0, g)$
2 while $P \neq \emptyset$ do
3 $\pi \leftarrow \text{PlanSelector}(P)$
4 $P \leftarrow P \setminus \{\pi\}$
5 if $\text{Flaws}(\pi) = \emptyset$ then
6 return $\pi$
7 else
8 $f \leftarrow \text{FlawSelector}(\pi)$
9 $P \leftarrow P \cup \text{AllFixes}(f, \pi)$
10 end
11 end
12 return false
end

In terms of plan quality, VHPOP was one of the top planners in the IPC-3 [58]. However, VHPOP was much slower than state-of-the-art state-space planners such as FF.

2.2 Learning in Planning

Machine learning has been used to improve planning by automating the acquisition and use of domain-specific knowledge and correcting heuristic inaccuracies based on past experience [92]. Many specific forms of knowledge have been learned to improve planning: macro operators, the heuristic error in a domain, action policies, and control rules are examples of such knowledge forms [92, 36].

In the following, we present some basic definitions, describe how learning in planning is achieved, and review the relevant literature.

2.2.1 Definitions

A feature $x(i)$ is an attribute of the problem instance $i$ of a domain $D$ that can always be measured. For example, the number of blocks on-table in the initial state is a feature of the problem instances in the blocksworld domain. A more general feature that is domain-independent is the number of atoms in the initial state. We use a capital letter $X(i)$ to refer to a vector of features $(x_1(i), x_2(i), ..., x_n(i))$.

A domain of feature $x$, $\text{dom}_x$, is a set of the real numbers. A value $y \in \text{dom}_x$ assigned to $x(i)$ is a measurement of $x$ in instance $i$. Similarly for a feature vector $X$, we define a domain of $X$, $\text{dom}_X$, and a measurement $Y \in \text{dom}_X$. A label, $f$, is a special attribute of the instance whose value is known in the training instances but is typically unknown in general. The objective of learning is to predict the label value $f(i)$ of new instances $i$ using the other features $X(i)$. 
A **Training Set** $T_{X,f}$ is a set of feature-label pairs. For each item, the set describes the observed data items' features $X(i)$ and the item labels $f(i)$. So:

$$T_{X,f} = \{(X(i), f(i)) \mid i \text{ is a data item, } X(i) \text{ is the data item features, and } f(i) \text{ is the item's label} \}$$

### 2.2.2 How To Learn

To learn a particular form of knowledge in planning, we primarily need to (1) choose a learning technique, and (2) find the features of the domain, problem instance, and/or state that is directly related to the learned aspect. For example, in search control rules, the atoms of a state can be the features we use to predict the best action to be taken in the state. In the following, we discuss the learning techniques and the learning features.

#### Learning Technique

*Supervised inductive learning* [92, 36] uses a set of training examples to make a prediction about future examples. We are given a set of training examples $\{(X(i), f(i)) \mid 1 \leq i \leq n\}$. We would like to find a hypothesis function $h(X(i))$ that minimizes the prediction error. An example of the prediction error is the squared difference between the hypothesis function and the objective function [7]:

$$\sum_{i=1}^{n}(h(X(i)) - f(i))^2.$$  

The hypothesis function is represented by a prediction model (or a predictor.) An example of the prediction model is a linear function of the features. The algorithm used to find such model is the learning algorithm. Examples of learning algorithms include linear regression, which is used to find linear functions [7] and SMO [73] which is used to find Support Vector Machine (SVM) models. In planning, linear regression, for example, has been used to find a predictor that estimates the heuristic function error in the FF planner [89].

#### Features

The choice of features is critical in learning, as they are used to make the prediction. The choice of the features depends on the planning aspect learned. In this work, we are interested in instance-specific aspects of planning, therefore, we will focus on features of the problem instance.

There are many sources of features in the planning literature. In the following, we list a number of feature sources in the literature:

1. binary variables representing the existence of the atoms in the initial state and goal [84, 88].

2. statistics about the predicates and operators in the domain [75, 76].

3. statistics about the facts in the problem instance [75, 76].

4. SAS+ [5] representation of variables' values in the initial state and the goal [26]. The SAS+ representation uses numerical variables to represent states instead of atoms; STRIPS atoms are converted into SAS+ variables and values by analysing how they are changed using the actions [42].

5. the causal graph's statistical information [76]. The causal graph is a structure used by some planners to represent the dependency between the variables in SAS+ representation [25].

6. the number of objects of each type in the problem instance [1].
Chapter 2. Background and Literature Review

7. a taxonomy syntax that classifies the objects into classes based on their participation in relations appearing in the initial state and goal [88, 89, 90]. More about taxonomy syntax is explained later when we present the feature selection algorithm in Section 3.3.4.

While there are many choices for the features, not all features suit all purposes. For example, plain binary variables of facts (1) and the SAS+ features (4) cannot represent all problem instances accurately, since the number of features is limited by the maximum number of objects in the instance. SAS+ may represent an instance from the logistic domain (see Appendix A) by assigning a variable to each truck indicating its location. If we take the variable of each truck’s location as a feature, there will always be some instances that have more truck-location features than others, since we can come up with instances that have more trucks than any training instance. If we ignore the new truck-location features, we will introduce inaccuracy in the representation.

The statistics of the predicates and operators (2) are only domain-specific structure, and cannot be used for instance-specific aspects. The causal graph features (5) have been used [76] to predict the runtime and the solubility of problem instances of some planners. However, it was found to be worse than the basic features drawn from statistics of the facts in the problem instance (3) and statistics of the predicates and operators in the domain (2) [76]. The number of objects of each type in the problem instance (6), the number of the atoms in the problem instance (3), and the taxonomy syntax classes (7) are suitable for instance-specific learning, and are general features. They are the ones that we chose to use in our work.

The taxonomic syntax technique (7) can be supported by a selection algorithm, since it produces a large number of features. The feature selection algorithm finds set of features that maximizes the likelihood of the data. Our feature selection algorithm is explained in Chapter 3.

2.2.3 Learning in the Planning Literature

Inductive supervised learning has been used to acquire different aspects of knowledge in planning [92, 36]. In the following paragraphs we briefly discuss some of these aspects and present some example works.

The main focus of this dissertation is learning to remodel the domain with macro operators. Macro actions are, traditionally, learned from sequences of actions found in problem instances’ solutions or search trees. Macro actions are generalized to macro operators, that can be added to the domain as a basic operators to speed up planning [67]. More about macro learning will be discussed in Section 2.3.

The heuristic error is another aspect that has been learned in planning. Heuristics are general approaches for solving the planning problem that can be inaccurate and need to be corrected. Yoon et al. [89] learned to correct $h_{RPG}$’s error using linear regression that models the difference between the relaxed plan length and the observed distance to goal in a given state. The training examples are generated from the solution plans of some problem instances of the domain: for each state on the plan path, the difference between the observed distance to goal and the heuristic value is used for training. This difference is modeled as a linear function of taxonomic syntax features derived from the states' relaxed planning graph. The authors present a feature selection algorithm that maintains a set of features, generates new feature from the old features, and greedily selects the ones that most improve the coefficient of determination ($R^2$) [82] value. The experimental results show that their learner can enhance the planner’s performance. We use a variation of that algorithm in our work in Section 3.3.4.

In a more recent heuristic-related learning work, Domshlak et al. learn to choose from existing
heuristics based on the problem instance’s features [26]. Trying to speed up optimal planning, the authors approximate the maximum heuristic of a number of admissible heuristics using learning. The maximum of admissible heuristics is admissible and is more accurate than any of them. However, finding the heuristic values to select their maximum is time consuming and can affect the planning speed. So, the authors built a predictor that tries to predict which of two admissible heuristics should be used in a state. The predictor is used and trained online for each state using a Naive Bayes algorithm. When the confidence in the prediction is low, both heuristics are calculated, and the best is used to update the model along with the state’s features, which are SAS+ variables. The results show that the selective heuristic performs better than the maximum heuristic and each of the individual heuristics.

Another aspect to learn in planning is the search control rules. Borrajo and Veloso presented a system called HAMLET [10] that uses the PRODIGY planner [65], a flexible planner that allows adding control rules to the search. HAMLET calls PRODIGY to generate search trees that are used to extract training examples. It generates explanations of the decisions found in the search trees and forms them into control rules used to decide when to apply an action and which action to apply. Then, the rules are refined to make the too specific more general, and to make a too general (and incorrect) rule more specific. The rule learning is not time consuming and the rules improve planning speed gradually. However, the generalization of rules is not always correct, because of the high flexibility of this technique that allows learning even from one path in the tree.

Learning has also been used to select whole search techniques or planners for problem instances. For example, Vrakas et al. presented a flexible and adaptive planning system called HAP [84] that is configurable using some parameters to perform different search techniques in different settings. The authors learn which values of HAP’s parameters are suitable to a given planning instance with known features. For training, the planner is run using all possible parameter configurations on training instances. Then, a number of rules that classify the parameters quality on problem instances are generated. The results show that the learned configuration and rules outperformed the best manual configuration of HAP on average.

Roberts et al. presented a system that learns to predict the best planners for a problem instance from 28 different planners [75]. The prediction is based on the domain and problem instance features, the ability of the planner to solve the problem instance, and the runtime taken to solve it. Offline the system builds an algorithm portfolio using the runtime and solubility data of the planners on the training instances. Online, given a problem instance, the system follows three steps: (1) it finds a set of planners, called the cover, that best solves the problem; (2) it ranks the planners and allocates execution time to each one in a round robin or a serial execution strategy; (3) the execution strategy is used to run the cover’s planners on the problem instance. The authors showed that their system performs better on average than random planners in the cover set, and that it outperformed individual planners in solving the problems on average. In a subsequent work [76], the authors show that using more training instances and harder instances gives more accurate predictions. The authors also found that the domain topology class [44] of the training instances was an indicator of the predictability.

2.3 Macros in Planning

In this section, we switch focus to one particular knowledge-based technique that is a main subject of this dissertation: macro actions and macro operators. A macro operator is a collection of ordered
operators that act as a single operator. A macro action is an instantiation of a macro operator. Macro operators are often used to solve problem instances by adding the macros to the domain definition as basic operators. Because macros improve planning speed – and because they are easy to represent, construct, and use – they have been attractive planning tools [11, 34].

Macros speed up planning mainly because they act as shortcuts to deeper states in the search space. Generally, the planning problem’s complexity is related to two main causes: the branching factor and the depth of the search tree. While techniques exist for reducing the effect of the branching factor, such as using helpful actions in the FF planner [49], the depth of the search tree remains a serious issue. Acquiring knowledge from previous search provides a strategy for tackling the problem: sequences of actions that lead to useful states can be reused in future searches to reduce the search tree depth and to save the search effort. Macros are also valuable in local search techniques since they can escape large plateaux [20] or local minima [35]. In some cases, macros can also help improve the underlying search heuristic [63].

Figure 2.1 shows a macro action composed of two basic actions from the blocksworld domain. The macro action (unstack-putdown c a) is the ordered application of actions (unstack c a) and (putdown c). It can be generalized to a macro operator (unstack-putdown ?x ?y) by replacing the constants by variables.

Macros like other forms of knowledge suffer from the utility problem. Using too many macros or macro operators with a large number of instantiations can negatively affect performance, since they increase the search branching factor. Also, using long macro actions can lead to some suboptimal plans. Therefore, a careful choice of the number and length of the macros is required [12].

Next, we give formal definitions of macro actions and operators, discuss how macros are extracted and used, and review the research on planning with macros.

### 2.3.1 Definitions

A **Macro Operator** $m$ in domain $D = (\Psi, O)$ is an operator composed of a sequence of operators $(o_1, o_2, ..., o_n) \in O^n$, where $n \geq 2$. Every operator $o_i$ in $m$ has at least one variable shared with another operator $o_j$ in $m$. This constraint of the macro’s variables is necessary but not sufficient to make it useful.

A **Macro Action** $ma$ is an instantiation of a macro operator. A macro action is **sound** if it is equivalent to a valid application of its actions in any state. A macro operator is **sound** if all of its instantiations (macro actions) are sound.
2.3.2 Macro Processing

Most of the macro acquisition works in the literature follow three main stages: the generation, the filtering and the usage of the macros.

1. **Macro generation**: Macros mainly are either generated artificially by exhaustive formation of sequences of basic operators [34, 11], or by lifting action sequences from solution plans [67] or search trees [20, 35] of previously solved problems.

   The source of macros is important to performance because finding good macros is not an easy task. Uninformed search in the macro operators space [34, 22] has been shown to be less effective than recently used techniques that combine forms of pruned searches with other techniques such as lifting [67, 11].

2. **Macro filtering**: Macro filtering is arguably the most important phase of the macro acquisition process [61, 35]. Typically, a large number of macros can be generated for a given planning domain, and keeping track of them can slow down the search [64]. For this reason, a way to filter, select, or rank the macros is required. The macro filter prunes macros based on some criteria such as a maximum macro length [11], empirical performance on some training instances [67], or frequency of occurrence in solution plans [11, 63].

   While the filtering according to the performance of individual macros is useful, the net effect of combining the macros should not be ignored. This is because macros interfere with each others' effects, which can harm the performance. Wizard [67] considered filtering the macro sets as well as the individual macros. This dissertation is based on macro set filtering: we search in the space of macro subsets for a set of macros that solves problem instances efficiently.

3. **Using macros**: Macros can be classified according to the context on which they are applied in planning to three types: domain-specific, instance-specific, and state-specific macros:

   (a) Domain-specific macros are associated with one domain. They usually come in the form of macro operators and are added to the domain as basic operators. Once these macros are acquired and filtered, they are used in any future instance in the given domain. Most of the macro works are domain-specific. Macro-FF [11] and Wizard [67] are examples of this type.

   (b) Instance-specific macros are used per individual problem instance. Before solving the problem instance, we choose one set of macro operators to be added to the domain as basic operators. The set is chosen based on its performance on similar, previously solved instances. This use of macros is novel and it will be discussed in detail in Chapter 3.

   (c) State-specific macros are those that are used within a given state. During the search, only some macros are considered in a state based on the state’s features. Some work used control rules to choose the macro to use in a given state according to that state’s features [38]. Another work, the Marvin planner [20], acquires the macro actions during the search and uses them in later states of the search to escape plateaux. Marvin’s macros are state-specific because they are continually added during the search: macros used in early stages of the search are not necessarily the same as the macros in later stages. However, the macros are not used later in new instances.
Next, we review two state-of-the-art macro acquisition tools that we use in this dissertation: Macro-FF and Wizard. Then, we review some other macro-related works.

### 2.3.3 Macro-FF

Macro-FF [11] is a planner built on top of the FF planner, and it generates macro operators and uses them in the search. Based on the way it generates macros, Macro-FF has two versions: CA-ED and SOL-EP. CA-ED is a system that extracts macro operators, adds them to the domain, and uses the new domain with the FF planner to solve future instances. In SOL-EP, the search is modified to incorporate the application of instantiated macros.

The Macro-FF CA-ED system learns macro operators and then adds them to the domain to speed up planning. The authors [11] follow four steps to find the macro operators: (1) they analyze the domain to discover abstract components, described in the next paragraph, (2) they generate artificial macro operators (assembled domain operators) from the components, (3) they filter and rank the macros using the training examples, and (4) they plan using the macros.

First, a domain analysis is done to extract the abstract components. A static graph that represents the relation between the problem constants is generated. The nodes in the graph represent the constants, and the edges represent static predicates that involve the two connected constants. A predicate is static if it appears in the initial state and is not in the delete effects of any operator. The abstract components are constructed from the subgraphs of the static graph. Every abstract component represents a unique set of constants that are semantically related to each other but not to constants of other components.

Second, macro operators are generated by searching for valid sequences of operators. A valid macro operator is constructed by adding one operator to the previously constructed macro such that: the operator’s preconditions are not deleted by the macro, it does not cause the macro to reach a state already achieved by the macro, its preconditions contain at least one fact that is added by the last operator of the macro, the macro size is limited, and the new macro’s preconditions do not relate two constants from different components.

Third, the macros are filtered by counting the number of their occurrences in the solutions of some training problem instances. The more often the macro appears as an instantiated macro action, the higher the chance that the macro will remain unfiltered. Then, the macros are ranked: the macro that appears the most in the plan paths in the training problem search tree gets higher ranking.

Finally, the macro operators are used in the CA-ED system by simply adding them to the domain. In SOL-EP, since it can deal with domains of higher complexity than STRIPS domain, the whole structure of the macro operators is stored and used during the search.

The experimental results show that planning performance improves after adding the macros. However, Macro-FF does not fully utilize previous experience, since the learned macros are not taken from all optimal solutions. Also, it is not always possible to find useful abstract components to build the macros.

### 2.3.4 Wizard

In Wizard [70, 69], Newton et al. present a genetic algorithm to learn macro operators from randomly generated small problems. Instead of using the macros generated only by lifting actions from previous plans, or the macros manufactured from the available actions, the genetic algorithm uses both
approaches. The authors, first, needed a set of small problems called the seeding problems to generate the macros by lifting them from the seeding problems’ solutions. Second, the macros are iteratively modified using genetic operators that can randomly change the macro’s actions, and evaluated using a set of larger problems called the ranking problems.

Wizard works in two phases: a chunking phase followed by a bunching phase. The chunking phase goal is to find macro operators that are useful when added individually to the domain. The bunching phase goal is to find sets of macro operators that are, on average, useful when added to the domain. This two-phase approach is critical in Wizard’s success and shows its level of sophistication towards macro filters, since finding a set of good individual macros does not mean that they will work well collectively.

The chunking phase is achieved through a genetic algorithm, following these main steps:

1. An initial population of macros is generated by lifting action sequences from plans of the seeding problem. Each macro is evaluated to assign a numerical rating.

2. Repeatedly, some macros and a genetic operator are randomly selected, where the operator is applied to the macros to generate new macros.

3. The macros are evaluated with the ranking problems. The domain of the each problem is augmented with the macro to be evaluated, and the problems are solved using the new domain. A number of criteria are used to evaluate the macro, such as the time of solving the augmented domains compared to the old domains (without the macro), the number of solved ranking problems, the frequency of the macro in the resulting plan, and others. The difference in the runtime is the most important in the evaluation.

   The genetic operators used include: lifting a random sequence from a plan, extending the macro by adding an action to one of its ends, shrinking the macro by removing an action from one of its ends, and splitting the macro into two parts at a random point.

4. During this evaluation process, the macros are filtered to include only good macros. Inferior individuals are replaced by superior ones to maintain a constant number of macros in each iteration.

Steps (2), (3) and (4) are repeated until a stopping condition is met.

In the bunching phase, the resulting macros are filtered to find the best subset of macros. The macros resulting from the chunking phase are fed to another genetic algorithm. Initially, random macro subsets are generated. Then, each subset is evaluated by measuring the performance of the planner before and after adding it to the original domain, similar to the chunking phase evaluation. The genetic operators in this phase include: adding a macro, deleting a macro, and the union of two subsets. Finally, the macro subset with the highest utility is chosen.

The experimental results show that there is an improvement in the performance when Wizard’s macro operators are added to the domains, independent from the domain and the planner used.

2.3.5 Other Macro Work

In this sub-section, we review some of the important and relevant works in macro action.
STRIPS

In a paper that followed the first STRIPS planning system, Fikes et al. presented a new approach to planning that uses macro operators [34]. They generalize the solutions (plans) of previously solved planning problems by replacing constants with variables to obtain large macro operators. All of the plan-based macros and their subsequence macros are stored efficiently in a data structure called a triangle table. The macros are used in subsequent searches by adding them to the domain as regular operators. This approach can cause a huge increase in the branching factor of the search, since a large number of macros is used. It was shown to work only on very small domains and instances.

REFLECT

The REFLECT planner [22] generates macros by creating sequences of operators. It performs domain analysis to evaluate the macros. Unlike STRIPS that generates macros from solution plans, REFLECT considers all possible sequences of operator pairs as macros. There is a large number of such pairs, so the system removes (1) macros that have no real effects (such as pickup-putdown in a blocksworld domain), (2) macros that involve the construction of an incorrect state, and (3) macros that have no shared variables. Finally, the macros are used as basic operators, and are added to the domain. The results show that REFLECT’s macros can improve the planning speed.

MORRIS

Noticing the problem caused by maintaining a large number of macros, Minton presents the MORRIS planner [63], which filters the macros by keeping only the useful and frequently used sequences.

Similar to STRIPS, MORRIS converts all plans to long macro operators, and considers all the subsequences of these generalized plans. Unlike STRIPS, it keeps frequently used macros (called the s-macros), and macros that are used to solve difficult parts of the problem (called the t-macros). These macros are acquired by analyzing the search tree, and recognizing the mistakes made by the search heuristics. Then, MORRIS adds the macros to the domain as normal operators. It was found that the t-macros, even though rare, are much more useful in improving the planning speed than the s-macros.

MACLEARN

In MACLEARN [52], the macros are generated from sequences of actions that lead from one peak (local maxima in the heuristic function) to another peak during the search of some problem instances. These macros go through a static filter, which deletes redundant and lengthy macros, and uses other domain-specific criteria to delete macros. Then they go through a dynamic filter that uses some problem instances to evaluate the macros using the instances’ solution plans. The resulting macro operators are also added to the domain permanently and used as normal operators. The results show that using MACLEARN to plan with macros was faster than planning without macros in a number of domains.

Marvin

The Marvin planner [20] is a modified version of the FF planner that uses macros designed to escape plateaux. It is similar to FF in that it has the two stages: an enhanced hill-climbing search (EHC) stage, and a best-first-search stage (BFS), but it has three modifications. First, instead of using a breadth-first search to explore plateaux, Marvin uses a least-bad-first search, which avoids exploring nodes with
increasing heuristic value. Second, Marvin uses previously discovered sequences of action that escaped plateaux to form macro actions, which are then applied to escape future plateaux. Finally, instead of the BFS, Marvin uses a variant of greedy best-first search that explores better successor nodes in the search quickly before completing expansion.

After detecting a plateau and escaping it with a sequence of actions in the EHC phase, the actions that lead from the root of the plateau to a better state are used to build a macro operator that Marvin uses to escape similar plateaux in the future. Only when a plateau is encountered, the macro actions are used as normal actions in the least-bad-first search. Not all macros are used: only the macros that have in their first step a helpful action (see Section 2.1.3) for the current state are considered in the search. The results show that Marvin’s plateaux escaping macros improve the performance in a number of domains.

Rule-based Macros

García-Durán et al. learned state-specific control rules that can include in their right hand side the application of a macro operator [38]. The control rules are obtained by the learning system HAMLET (see Section 2.2.3 for more details.) This work uses macros in a state-specific context. However, the macros are included as basic operators as well, i.e., they are usable if no rule is triggered. Therefore, this method is still domain-specific.

The authors tested the approach on a number of domains. For the domains where macros were available, the learning rules did not improve the speed in most cases.

Given the previous work, it can be noticed that the macros are either used in the domain level, where the differences between the instances within the domain is not considered, or in the state level, where the overhead that results from using the macros can be an issue.

2.4 Reformulation in Planning

Earlier in this chapter, we discussed how macro operators are used to remodel the planning domain by adding them as normal operators. In this section, we discuss how a planning domain is generally remodeled in the literature and we review different methods of remodeling that are relevant to our thesis.

Reformulation is the approach by which we use a known, useful solving strategy to solve our problem: the problem is put in the canonical form that allows the particular solving strategy to be applied [59]. The reformulation approach should also be able to translate the resulting solution back to a solution of the original problem when needed. In this dissertation, the words “reformulation” and “remodeling” are used interchangeably.

The two main goals of reformulation in planning have been to speed up the search and to improve the domain theory. We discussed in the previous section how the macro-based domain reformulation is intended to achieve the first goal. The second goal involves finding the right set of operators and predicates to describe the environment of the agent or to describe the class of problems to be solved [79]. While the second goal is essential for tailoring planning domains for their intended tasks, the first goal will be our focus.

Similar to macro usage in Section 2.3.2, reformulation can be achieved in the context of the problem instance or the domain. In the instance-specific reformulation, the problem (the instance and/or the
domain) is remodeled for each new attempt to solve it. In the domain-specific reformulation, the planning domain is remodeled only once, and then it is used to solve every problem instance from the domain.

There are different methods of reformulations in planning. For example, planning problems can be converted into another class of general problems (e.g. propositional satisfiability – SAT), and then solved using a specialized solver (e.g. SAT solvers). Alternatively, the definition of a planning problem represented in one form (STRIPS, for example) can be converted to another form (SAS+, for example), making it possible to solve the problem quickly with a different planner. The domain definition can also be modified by adding and/or removing atoms or actions to speed up planning. In the following we review six different reformulations: representation-based; reformulation to non-planning problems; atom addition; abstraction; operator removal; and macro addition and operator removal. In this dissertation we use two methods of domain reformulations: reformulation by adding macro operators, and reformulation by adding macro operators and removing original domain operators. We will not discuss macro addition reformulation here since we discussed it in Section 2.3.

2.4.1 Representation-Based Reformulations

Some works used a simple conversion of the representation from the classical fact-based representation (STRIPS form) to another: for example, the multi-valued variable representation (SAS+ form.) This kind of conversion is achieved automatically in some planners like LAMA [74] and FD [42] as a pre-processing step, since they use data structures that require the SAS+ representation. This reformulation is instance-specific, since it is required for every new instance.

2.4.2 Reformulation to Non-planning Problems

In the literature, the planning problem has been remodeled as a sequence of propositional satisfiability problems (SAT) as in the BLACKBOX planner [54] and SatPlan [53]. The planning problem is represented as a number of binary variables and a formula of these variables, such that finding a satisfying variable assignment solves the planning problem. These variables and the connectives between them in the formula are obtained from the planning graph structure and its constraints (see Section 2.1.2).

The planning problem has also been remodeled similarly as a constraint satisfaction problem (CSP) in the GP-CSP planner [23], and solved using a CSP solver.

van den Briel et al. reformulated and solved the planning problem using integer programming (IP) techniques; they decomposed the problem into different components, then combined them using a branch-and-cut algorithm [83]. Using this technique, each component corresponds to a state variable and represents a network flow problem with layers of nodes. The layers represent planning steps, and the nodes of each layer correspond to the different values of the variable. The arcs represent value changes with actions. The authors solve the problem using an integer programming formulation of the components, which is fed into a branch-and-cut algorithm.

In a different work, using the STAN planner and the TIM system [59, 57], the planning problem was converted into a set of small, identifiable problems that can be solved efficiently using specialized solvers. Some of these sub-problems are identified within the planning problem by noticing their “fingerprints” or generic behavior. This kind of reformulation is also instance-specific, since it is required for every new instance.
2.4.3 Reformulation By Adding Atoms

Some works remodeled the planning problem instance directly by adding goals to speed up planning. Domshlak et al. used a goal reformulation to improve the performance of existing heuristics [27]. They show that some abstraction-based heuristic (e.g. structural patterns) are sensitive to the number of goal facts: problem instances with multi-fact goals are solved faster using some abstraction-based heuristics than their corresponding, modified, single-fact goal copies. The authors therefore suggested the use of landmarks, which are implicit sub-goals that must be achieved before finding a solution plan, to modify the goals. Landmarks can be found quickly using sound but incomplete algorithms. Using the SAS+ representation, the authors added a binary variable to the goal for each landmark, that indicates whether the landmark was achieved or not. Each of these variables is initially unachieved and must be achieve in the goal. Using two abstraction-based heuristics: merge-and-shrink and the fork-decomposition, they tested the performance of two A*-based search algorithms with and without landmark variables. The results showed an improvement in the planning performance in a number of domains. This kind of remodeling is instance-specific, since the landmarks are added to the goal as the problem instance is to be solved.

2.4.4 Reformulation By Removing Atoms (Abstraction)

Planning problems can be remodeled by modifying the domain or problem instance definition using abstraction. Abstraction in planning is a simplification of the problem instance to a smaller one that has fewer atoms and may be easier to solve. The solution plan of the abstract instance can be used directly to construct a solution plan for the original problem [55], or can be used in the search heuristics [43].

To solve the problem instance directly, abstractions are applied in a hierarchical manner, where at the top lies the simplest form of the problem, and at the bottom lies the original problem. The idea is that remodeling the problem into a simpler one, solving it, and modifying the resulting plan to be a solution plan of the original problem can be easier than direct planning. Knoblock presented a system called ALPINE which generates hierarchies of abstractions and uses them in the PRODIGY planning system [55]. Abstraction makes a mapping from the literals of the abstract domain to the literals of the source domain. Therefore, in a hierarchy of abstraction, each literal is assigned a number that represents its level in the hierarchy. At hierarchy level $i$, only literals numbered $j \geq i$ are allowed to appear in states, operators, and plans. Thus, there is a mapping from states, operators, and plans in abstract level $i$ to the states, operators, and plans, respectively, of another abstract level $j \geq i$. The goal is to convert the solution plan of each level $i$ to a solution plan of level $i-1$ by adding actions, until the original problem is solved.

The system maintains an ordered monotonicity property which minimizes the interaction between the hierarchy levels, reducing the need for major changes to the abstract plans, and hopefully improving performance. The property is maintained by limiting the plan modifications to the actions that add literals from the same hierarchy level. However, this property does not guarantee performance because not all tasks are hierarchically organized. ALPINE solves planning problem in two steps. First, a hierarchy of abstraction is chosen. Then, a hierarchical version of the PRODIGY planner is used to solve the hierarchies starting from the simplest. The results show that ALPINE can improve the planning speed in some domains.

Abstraction is useful because it maintains the theory of the domain: what holds in the original domain
holds in the abstract domain, but not the inverse. However, not all problem instances are hierarchically organized, therefore, the modifications needed for the abstract plans can be costly. It is clear that this kind of remodeling with abstraction is instance-specific, since it is needed for each instance.

While domain remodeling by abstraction can be used directly to improve planning, it can also be used indirectly in heuristic functions. Helmert et al. presented an admissible, abstraction-based heuristic that depends on the size of the optimal abstract plan [43]. The heuristic is calculated as the length of the optimal plan in the abstract state space.

The main advantage of using abstraction in heuristic functions is that they remove many unimportant states from the state space while preserving the paths. However, abstract state space can still be too large for explicit search for optimal plans. The idea in Helmert et al.’s paper is to limit the size of the abstractions while preserving the relevant states and paths. This requires shrinking the abstract state space to maintain the tractability of the heuristic. The heuristic, then, is calculated by explicitly finding the shortest path from the start state to a goal state in the shrunken state space.

The heuristic algorithm begins with a set of small abstractions built using single-variable projection. The authors assume SAS+ representation of the problem instances, and therefore instead of removing atoms, an abstraction is achieved by removing variables from the SAS+ version of the problem instance. Projection abstractions (or projections) are generated by ignoring all variables except the projection variables, fusing the states where the projection variables have the same values. The abstractions, then, are merged and shrunk iteratively. Initially, two single-variable projections are merged using a product operation. Then, the abstraction size is checked: if the resulting abstraction’s number of states exceeds a constant $N$, the abstraction is shrunk. A different single-variable projection is chosen next to be merged with the resulting abstraction, and the result might be shrunk. The merge-and-shrink process is repeated.

The choice of the single-variable projection in the merging step is based on the position of the projection variable in the problem instance’s causal graph [25]. The choice depends on how close the variable is to goal variables. Initially, the goal variables’ projections are chosen, and next, projections of variables on which the goal variables depend are chosen, and so on.

The shrinking mechanism generates a new type of abstraction that is not necessarily a projection of the original problem: the new abstraction’s states cannot be mapped back to their original states. To shrink an abstraction’s state space, the two abstract states that have the largest distance from the goal and the initial state are fused together to form one state.

The results show that this merge-and-shrink heuristic is more accurate (explores fewer nodes) than the pattern databases heuristic $h_{PDB}$ [28], which relies on projection abstractions only. The results also show that this heuristic achieves better performance using the $A^*$ algorithm than the blind heuristic, the admissible $h_{max}$ heuristic of the HSP planner [9], and the pattern databases heuristic $h_{PDB}$. The search with merge-and-shrink heuristic was also faster than a number of planners.

### 2.4.5 Reformulation By Removing Operators

Irrelevant operators can increase the number of states explored in the search in general. Even planners with the ability to detect irrelevant actions can sometimes fail to find them [66]. One way to modify the problem definition for speed gain is to remove irrelevant and redundant actions. In the following we give formal definitions of the concepts of action redundancy and irrelevance as explained in the literature, we show why redundant action removal is difficult, and we review methods used to prune operators and
actions from the planning domain.

Not all actions and operators can be removed safely, since a solvable problem instance can become unsolvable by removing some actions, and detecting these actions is hard. An action is solution irrelevant (redundant) with respect to a planning problem if it does not appear in at least one solution of the problem instance [32]. An action is completely relevant (necessary) with respect to a planning problem if it is contained in all solutions of the problem instance [66, 32]. Ferrara et al. prove that checking the redundancy of a specific action in a given problem instance is as hard as the planning problem itself [32] (PSPACE-complete [13, 29]). They also show that finding a minimal number of actions to solve a problem instance is PSPACE-complete.

Some works used the fact that most planners cannot recognize many kinds of irrelevant informations while solving the planning problems. Nebel et al., for example, solved the planning problem by exploiting the irrelevance property of some initial facts and actions by identifying the states that are relevant to the goal and performing relaxed parallel planning though regression [66]. The approach depends on trying to find the “probably relevant initial facts” of the given problem instance. These are the initial state facts that are thought to be used in at least one solution of the problem. Ignoring all other initial facts, regression-based actions, and objects that do not involve these facts will not affect the correctness of the solution. Using only the relevant facts, actions and objects will reduce the number of choices for the planner. The probably relevant initial facts can be found by building an AND-OR directed-acyclic graph called the fact-generation-tree starting from the goal facts. The graph represents all the paths (to a certain depth) through which every initial state fact can be reached from the goal facts using relaxed ground actions with no delete effects. Using this graph, the minimum set of facts necessary to achieve every possible solution trace can be found. Although this graph can be built in polynomial time, finding this set of facts from the graph is NP-hard, because the minimum necessary initial facts sets of all possible solutions must be acquired and there can be an exponential number of them. The authors used approximation by limiting the number of accepted initial fact sets. They proposed a method to combine and use the resulting sets. Although there was a significant speed-up in the results, this method is not solubility-preserving, as it can return a ‘no plan’ result for a solvable planning problem.

Other works used a similar idea that relies on the fact that there are many actions that can achieve the same goal, and some of them are subsumed by others, i.e., they can be replaced by them. Haslum and Jonsson, for example, relied on removing ground actions that can be replaced by other actions in the plan [41]. In a preprocessing step, it is proved that an action is subsumed by a sequence of actions that do not contain it, and then, the action is removed. This method is solubility-preserving but requires a lot of time for the preprocessing, and the preprocessing is required for each instance since subsumption is proven within the problem instance.

Chrpa and Barták presented a method to remove some domain actions by modifying operators so that they cannot be instantiated to the actions [18]. The authors noticed that some operators are not used in the plans unless some of their precondition (effects) appear in the initial state (goal). So they reformulated the domain by replacing the operators whose ground actions have preconditions (effects) that appear only in the initial state (goal) of training instances plans by new copies of the operators that are only applicable in the initial state (goal). This method can reduce the branching factor since it reduces the number of applicable actions. However, it is not clear whether such operators appear frequently in the planning domains.
2.4.6 Reformulation By Adding Macros and Removing Operators

Few papers have combined the use of macros with the operator/action removal. The only paper we are aware of is Chrpa’s [16], where the author combines his work on macro acquisition [17] with his work on operator removal [18]. Chrpa presented a method to extract macro operators based on the action dependencies [17]. As described earlier, in [18] the authors replace operators by less applicable versions to reduce the branching factor. In [16], the results of combining the two approaches show that it is better than either of the methods in some domains.

2.5 Algorithm Tuning

Algorithm parameter tuners are tools used to improve the performance of a generic algorithm by analyzing the algorithm’s performance when run with different configurations of its parameters. Controlling the behavior of complex solvers typically requires many parameters. Configuring these parameters in order to produce the best performance is a tedious, trial-and-error process. To automate that process, a number of automatic parameter tuning tools were invented. One such tool is F-race [6] which has shown the ability to improve the performance of local search algorithms.

In this dissertation, we recast the domain remodeling problem as a parameter optimization problem, meaning that it can be solved using a parameter tuning tool. This dissertation focuses on solving the planning problem by choosing from a large number of domain reformulations, with the important goal in mind to enhance planning by choosing the reformulation that solves the target problem instances efficiently. Making such a choice is, as a process, similar to tuning the parameters of an algorithm to find its best performance. We use a generic parameter tuner, ParamILS [50], for the purpose of choosing domain reformulations.

2.5.1 ParamILS

Hutter et al. presented ParamILS [50], an automatic algorithm parameter tuner for performance optimization. ParamILS is based on the Iterative Local Search algorithm (ILS) [50, 60], a local search algorithm that iteratively applies greedy search and random moves in the search space.

ParamILS is a general-purpose algorithm tuner designed to deal with a very large space of numerical and categorical parameters. It takes as input: the target algorithm, the algorithm’s parameters with their possible values, and training problem instances acceptable by the target algorithm. It, then, produces a parameter configuration, which is a full assignment to the algorithm’s parameters.

ParamILS has different versions: BasicILS and FocusedILS, each capable of operating with and without adaptive capping. BasicILS is a variation of the ILS algorithm in the parameter configuration space, in which a neighboring configuration is reached by changing exactly one parameter value. FocusedILS is an extension to BasicILS where the search is improved by minimizing the number of training instances required to compare a pair of parameter configurations. The adaptive capping method is applicable in both BasicILS and FocusedILS: it is used to interrupt unnecessary runs while in the process of comparing parameter configurations. Comparisons in ParamILS are based on the algorithm runtime using both configurations on the training instances. Adaptive capping is used to speed up the search by maintaining an upper bound on the configuration’s runtime. For example, if the time already spent to evaluate one parameter configuration $\theta$ is longer than what has been spent on the best parameter configuration $\theta_{best}$,
then $\theta$ has exceeded the runtime upper bound, and should be immediately rejected.

In its local search, ParamILS maintains a variable $\theta_i$, whose domain is the parameter configurations, for each iteration $i$, and begins by assigning initial parameter configuration values to $\theta_0$. It also keeps track of the best parameter configuration found $\theta_{best}$ at all times. In the end, it returns $\theta_{best}$. In the following we describe the general outline of ParamILS:

1. Let $i = 0$, and update the initial configuration $\theta_0$ as follows:

   (a) Assign the initial configuration value supplied by the user to $\theta_0$.

   (b) Generate a number $r$ of random parameter configurations. If any one of these configurations is better than $\theta_0$’s configuration, it replaces the values stored in $\theta_0$.

   (c) Perform a random walk of length $s$ in the parameter configuration space, without evaluating the parameters, starting from a random neighbor of $\theta_0$’s configuration. Assign the last parameter configuration value to $\theta_0$.

   (d) Perform a greedy search starting from $\theta_0$’s configuration: find the best neighboring configuration to $\theta_0$’s configuration, and keep looking for a better configuration among the resulting configuration’s neighbors, until no such configuration is found. Assigning the resulting configuration value to $\theta_0$.

2. Increment $i$.

3. Begin iteration $i$ by initializing the temporary variable $\theta$, letting $\theta = \theta_{i-1}$.

4. Perform a random walk in the parameter configuration space of length $s$ starting from a random neighbor of $\theta$’s configuration. Assign the last parameter configuration to $\theta$.

5. Perform a greedy search starting from $\theta$’s configuration: find the best neighbor to $\theta$’s configuration, and keep looking for a better configuration among the resulting configuration’s neighbors, until no such configuration is found. Assigning the resulting configuration value to $\theta$.

6. If the configuration contained in $\theta$ is better than the iteration variable’s configuration ($\theta_i$’s) then let $\theta_i = \theta$.

7. With probability $p$, choose a new random parameter configuration value for $\theta_i$.

8. Increment $i$ and repeat steps 3 to 7 for another iteration. Stop when ParamILS times out or when the best parameter $\theta_{best}$ converges.

The authors tested ParamILS on three target algorithms, each with a large number of possible configurations, and three sets of benchmark problem instances [50]. The experimental results show that all versions of ParamILS often find configurations that significantly improve the target algorithms’ performance, as compared to their default configurations, random search and standard local search algorithms. The theoretical results show the convergence of FocusedILS.
2.6 Summary

In this chapter, we presented the background to our work. First, we discussed classical planning, how a planning problem is solved, and we gave examples of state-of-the-art planners. Second, we reviewed learning in planning: we briefly discussed the learning algorithms and features, and we reviewed work in the literature. Third, we discussed macros and classified them, and reviewed the important research on macros. Fourth, we discussed the reformulation of the planning problem as a key topic of the dissertation. We discussed a number of reformulation methods in the literature and reviewed work corresponding to each method. Finally, we discussed algorithm tuning tools. Specifically, we reviewed ParamILS, a general parameter configuration tool that we use in Chapter 4. In the next chapter we present and explain our first approach to improve planning using macro operators and machine learning.
Chapter 3

Learning to Choose Instance-Specific Macro Operators

3.1 Introduction

The acquisition and use of macro operators has been shown to be effective in improving the speed of AI planners [21, 11]. However, current macro acquisition works focus on finding macro sets that, when added to the domain, can improve the average solving performance on all instances. In this chapter, we present an instance-specific macro learning method, which depends on building a predictor to estimate, for any planning problem instance, the best subset of previously collected macros for the instance.

The instance-specific predictor chooses a subset of the available macros to solve the problem instance based on the problem instance’s features and the automatically learned knowledge about the relation between the features and the macros. To find this relation, we use machine learning methods. Training the predictor is done off-line based on the observed correlation between training instance features and planner performance in macro-augmented domains. Online, we extract the features of the new problem instance, and the predictor is used to estimate the appropriate set of macros to add to the domain.

Our empirical results over six standard planning domains demonstrate that our predictors can significantly outperform the fixed-macro-set method that chooses the best average macro set. We also demonstrate that there is a potential to improve the performance significantly by showing that the oracle predictor is significantly faster than the fixed-macro-set in most domains.

The primary novelty of this chapter is that it achieves domain remodeling on an instance-specific basis rather than a domain basis. This instance-specific remodeling is an important step towards the full integration of remodeling and solving, as suggested by Simon [80]. The application of this general idea enriches planning since it opens a new avenue to solve the planning problems.

3.2 Motivations

Our approach to learn instance-specific macros is motivated by the following:

1. Exploiting The Differences Among Instances – Given a set of macro operators, the effects of the utility problem that accompanies the use of macros can be reduced by considering the dif-
ferences among the problem instances. As discussed in Section 2.1.2, macros suffer from the utility problem: adding too many macros can slow down the search since it increases the search branching factor [64]. Macro acquisition tools have focused on avoiding the utility problem, primarily by ranking the macros and pruning the less useful ones [11, 67]. Some tools filter sets of macros to avoid the utility problem that results from the macros’ redundant effects [67]. But the utility problem can still exist, since these tools depend on finding a single set of macros to solve all future problem instances.

One method to evaluate the utility of a macro operator is to count the frequency of its occurrence in previous solution plans [11]. It is noticeable that problem instances differ substantially: they contain different initial facts and goals, and therefore require different plans to solve them. Since instances’ solution plans differ substantially, finding an action subsequence that is common in all plans is unlikely. Finding non-contradicting instantiations of many macro operators in the plans of every instance is even less likely. Therefore, using specific macros to solve specific instances is more likely to improve the utility problem than using all macros to solve all instances.

2. Learning To Remodel – Automating the choice of macros for each problem instance is not easy: we generally cannot know which macro set is useful for the instance except by experience. Therefore, using machine learning to automate the choice of macros is important. There is no method to discover the best subset of macros for a problem instance without solving the instance using the macros. However, similar instances share similar properties, and macros that worked for one instance may work for a similar instance. Machine learning has been successfully used to improve planning speed by finding a relation between the learned knowledge and the problem instances features [90, 75]. Given the advancements of the field of machine learning [7], we think that using machine learning to identify the relation between the problem instance features and the macro-based remodelings is promising.

3.3 Approach

The goal of our work is to improve the performance of planners by building an automatic predictor that chooses, for a given problem instance, a macro set that is relevant and useful based on the instance’s features. Our approach relies on considering all the subsets of the macros provided. We evaluate each macro subset on each training instance by running the planner on the macro-augmented domain and the training instance, and registering the runtimes. Then, we use the evaluations and a machine learning tool to train our predictors to choose the best macro set for the problem instance based on the instance’s features. We use these runtimes to train our prediction models in order to predict which macro subset is best for future problem instances.

In the next subsection we give the formal definitions of the reformulated domains (or r-domains), the utility of r-domains, and other definitions related to the predictors. Then, we explain the approach in detail. We address two issues: the nature of the macros to be provided to the system and the features that will be used for the learning and prediction.
3.3.1 Definitions

In the following, we present the definitions necessary for describing our approach to learn instance-specific remodeling.

**R-Domains**

Given a planner $P$, domain $D = (\Psi, O)$, initial macro set $S$, and a distribution over the problem instances in $D$ we define the reformulated domain. The definition is general enough to allow macro addition as well as operator removal, which we will need for Chapter 5.

A Reformulated Domain (or r-Domain) $D_{M,S,L}$ is a remodeling of a planning domain $D = (\Psi, O)$. Given a set of initial macros $S$, the domain $D$ is reformulated by adding the set of macro operators $M \subseteq S$ to $D$, and removing the set of basic operators $L \subseteq O$. Thus, $D_{M,S,L} = (\Psi, O \setminus L \cup M)$. For example, $D_{\emptyset,S,\emptyset}$ is the unchanged domain $D$, $D_{M,S,\emptyset}$ is $D$ after adding the macros of $M \subseteq S$, and $D_{S,S,O}$ is $D$ after removing all of its original operators and adding all the macro operators in $S$.

An Instance-Specific Reformulation (or just Reformulation) $\Delta_{D,S}$ is a function that maps problem instances $i$ from the domain $D = (\Psi, O)$ to r-domains. $\Delta_{D,S}$ can have a different domain definition for each problem instance. Given a distribution $\text{Dist}_D$ over the problem instances $i$ from $D$: $\forall i \sim \text{Dist}_D, \exists M \subseteq S, L \subseteq O : \Delta_{D,S}(i) = D_{M,S,L}$.

**R-Domains Utility**

Utility of an r-domain $d$ on a problem instance $i$, $(U_{(i,P)}(d))$ is a measure of the performance of an r-domain $d$ based on problem instance $i$ of domain $D$ using the planner $P$. $U_{(i,P)}(d)$ maps the r-domain $d$ to real numbers and represents the quality of the r-domain for $i$ and $P$. For example, we can let: $U_{(i,P)}(d) = -\text{DecisionTime}(P, i, d)$, where $\text{DecisionTime}(P, i, d)$ is the time taken by the planner $P$ on problem $i$ and domain $d$ before it halts and gives a decision. If the planner requires a long time, we can add a cutoff to the definition as we will see in our experiments.

Utility of a reformulation $d$ on a distribution over instances $U_{(\text{Dist}_D,P)}(d)$ is a measure of the performance of the reformulation $d$ based on problem instances of domain $D$ drawn from a distribution $\text{Dist}_D$. We define it to be the expected performance of $d$ on the domain problem instances, i.e.:

$$U_{(\text{Dist}_D,P)}(d) = \mathbb{E}_{i \sim \text{Dist}_D}[U_{(i,P)}(d(i))]$$

Utility of a reformulation $d$ on a set of instances $U_{(I_D,P)}(d)$ is a measure of the performance of the reformulation $d$ based on problem instances $i \in I_D$ of domain $D$. It is used to approximate the utility on the instance distribution when we do not know the distribution. We define it to be the average performance of $d$ on the provided problem instances $I_D$:

$$U_{(I_D,P)}(d) = \hat{\mathbb{E}}_{i \in I_D}[U_{(i,P)}(d(i))] = \frac{\sum_{i \in I_D} U_{(i,P)}(d(i))}{|I_D|}$$

For simplicity, we will refer to $U_{(I_D,P)}(d)$ by $\hat{U}(d)$ when it is understood from context.
Learning

A Classifier function $\text{Classify}(i, T_{X,f})$ is a function that uses a classification-based learning algorithm to compute a prediction for the label $f(i)$ of a data item $i$ based on the trained model on the training set $T_{X,f}$. The classification-based algorithm is a procedure that tries to predict the category of a data item based on training on similar items and their categories. The predicted label is a discrete value (category) that appears in $T_{X,f}$. An example of a classification function is the output of a logistic regression algorithm.

A Regression function $\text{Reg}(i, T_{X,f})$ is a function that uses a regression-based learning algorithm trained on the training set $T_{X,f}$ to predict the label of $i$ using its features. The regression-based algorithm is a procedure that outputs a real value that represents the mapping of a data item based on a continuous function learned from sample data items and their real-value mappings. Here, the predicted label can be any real number. An example of regression functions is the output of a linear regression algorithm.

3.3.2 System Details

The system follows the instance-specific macro learning framework we introduced in Chapter 1. The system works in two phases: an off-line training phase and an on-line prediction phase. Figure 3.1 shows a diagram of the whole system. In the off-line phase, we train machine-learning based prediction models to relate measures of the problem instance features to the performance of the planner in domains augmented with macro sets. Online, the features of a new instance are measured and the prediction model (or the predictor) is used to identify a macro set to be added to the domain. The problem instance is then solved with the added macro set. The goal of the predictor is to find the macro subset that, when added to the domain and used to solve the instance, will minimize the solution time.
Figure 3.2: A schematic diagram that describes the main steps in the training phase.

Training

The goal of the training phase is to come up with an accurate predictor. We introduce two types of prediction models: a Direct model, which predicts the best macro subset directly based on the problem features, and a Time model which, first, predicts the runtime of the planner on the problem instance with each macro subsets based on the problem’s features, and then chooses the macro subset whose predicted runtime is the smallest.

Figure 3.2 shows the general structure of the training phase. The arrows represent the order of execution of the steps in the system. The dotted arrow between steps 2 and 3 means that the steps are ordered in our system but are not required to be in general, as will be explained later. Given the planning domain \( D \), the training phase is performed as follows:

1. **Macro sets and instances generation:** As depicted in Figure 3.3, the original domain is provided to a macro source (Wizard or Macro-FF for example) and a set of \( n \) macros is obtained.

   Next, the domain generator creates \( K = 2^n \) macro subsets from the original \( n \) macros and augments the original domain with each subset in turn, producing \( K \) different r-domains: \( \{D_{M,S,\emptyset} | M \subseteq S\} \). Then, a problem generator for the domain is executed to create \( N \) training instances.

2. **Collecting runtimes:** As shown in Figure 3.4, each of the training instances is solved using the same planner with each of the augmented domains generated in step 1. The runtime for each macro subset and problem instance is recorded.

3. **Gathering features:** As shown in Figure 3.5, the training instances are passed to a feature selector. The feature selector automatically tries to find a set of features that is correlated to the runtimes of the macro sets on the training instances. When the features are chosen, a set of numeric feature measures is produced for each training instance, and a description of the features is generated for later feature extraction. Although this step can be achieved in parallel with the previous step, in this system we use the runtime data from the previous step to filter the features as will be described in Section 3.3.4.

4. **Training the time predictor:** Independently, for each augmented domain, a Time prediction model is generated as shown in Figure 3.6 using a supervised machine learning tool.
Figure 3.3: The process of generating the macro-augmented domains and the training problem instances.

Figure 3.4: Runtime collection.
Chapter 3. Learning to Choose Instance-Specific Macro Operators

Figure 3.5: Feature extraction and selection.

Figure 3.6: Training the Time Model.

Figure 3.7: Training the Direct Model.
The time predictor is composed of $K$ small time predictors. The purpose of a small predictor $i$ is to predict the solving time of a problem instance on the domain augmented with the $i$th macro subset. To train a small predictor $i$, the runtime of each training instance on macro set $i$ is paired with the instance’s features and passed to a regression learner. Typically, the final Time predictor must be created by using the results of all individual predictors. When provided with a new instance, the Time predictor runs each of the individual small predictors and chooses the macro subset with the smallest predicted runtime.

5. **Training the direct predictor:** The Direct predictor training is achieved in a similar way, preceded by steps 1 through 3, as described in Figure 3.7.

First, a macro set selector accepts the runtimes resulting from executing one training instance with each of the macro subsets. Then, it selects the macro subset that has the lowest runtime on that instance, and outputs the problem instance’s features measurements and the number of the selected macro subset.

The resulting pairs of (features, macro-subset-number) are then passed to a classification-based learner to produce the Direct predictor. The resulting predictor directly predicts the appropriate macro subset number based on the given problem instance’s features.

**Prediction**

On-line, the prediction component accepts a problem instance from the same domain, extracts its features, chooses the macro subset that is estimated to perform best with the instance, based on the features, and then outputs a new domain consisting of the original domain augmented with the macro subset. The problem instance can now be solved using the new domain.

**3.3.3 Macro Sources**

We need an initial set of macros to choose the subsets from. In general, the macros can come from any source, but we experiment with three sources: the Wizard system, the Macro-FF planner, and manually constructed macros.

As described in Section 2.3.4, Wizard generates and filters the macro operators in its chunking phase using a genetic algorithm. It then chooses a subset of the macros in the bunching phase and outputs the result. We capture the macros that result from Wizard’s chunking phase and use them as our initial set of macros. In Macro-FF (see Section 2.3.3), we use the macros from the CA-ED version. For a given domain, CA-ED generates macro operators to add them to the domain definition statically in a preprocessing step. Then, it uses the FF planner to solve the problem instances using the new domain. We chose these macro tools because they are known to find useful macros and improve the planner’s performance.

The third source is the manually constructed macros. The manual macros were tested on a small set of instances for both performance and validity. The purpose of using manual macros is to generalize the scope of tested macros, since we know that useful macros can also be found manually.
3.3.4 Learning and Features

Here we describe the learning algorithms and the features we used to build the predictors. The system relies on machine learning algorithms to train the prediction components. The training also requires finding a set of features that describe the problem instance and can be used to identify the appropriate macros for that instance.

Learning Algorithms

As previously discussed in Section 3.3.2, we have two main prediction methods: a direct and a time predictors. For the time prediction method, each of its small time predictors relies on a regression learner, since each predictor tries to predict the runtime of the planner on the macro-augmented domains. The direct predictor relies on a classification learner, since it chooses one macro set from a set of macro subsets.

There are different classification and regression algorithms to choose from. For the time model, we chose the M5P algorithm [85]. For the direct model, we chose the SMO-SVM algorithm [73]. Our choice of these predictors is based on the fact that the runtimes (and the choice of the best macros set) were not found to be linearly associated with the features. The M5P algorithm and the SMO-SVM algorithms are capable of finding non-linear relation between the features and the predicted values.

Feature Selection and Extraction

Our system relies on selecting a set of features that form the basis for prediction. These features should be relevant to the macro-augmented domain performance. Many sources of features can be found in the literature, as described in Section 2.2.2, but we chose three sources:

1. Statistics about the facts in the problem instance. In Section 2.2.2, we discussed a number of feature sources where this kind of feature was number (3). We choose two simple features of this kind: the number of atoms in the initial state, and the number of atoms in the goal state.

2. A general feature selection and extraction method that is based on a taxonomic syntax language. In Section 2.2.2, this kind of feature is number (7). We use a similar method to the one discussed in [90] that depends on the taxonomic syntax and filters the features according to their correlation to the runtime of the training examples on some selected macro-augmented domains. See below for details. This method is automatic and domain-independent.

3. Manually selected domain-dependent features. In addition to the domain-independent and automatically selected features, we use some domain-dependent, manually selected features in one benchmark domain: freecell.

Refer to Appendix D for more details on the selected features.

In our experiments, we use a combination of the features of (1) and (2) in all domains, and we add the features of (3) in the freecell domain. Because the features in (1) are straightforward and those in (3) are domain-dependent and hence cannot be an integral part of the system, we rely heavily on the taxonomy syntax features.

A taxonomic syntax language [90] represents general features for arbitrary planning domains and problem instances. It allows the construction of classes of objects organized into levels in a hierarchical
Chapter 3. Learning to Choose Instance-Specific Macro Operators

Algorithm 2: Feature-Selectors

\textbf{input} : Maximum level: $L_{max}$, runtime of selected r-domains on the training instances: 
\textit{runtime}, domain: $D = (\Psi, O)$

\textbf{output}: Selected features $\Phi$

\begin{verbatim}
1 begin
2    $\Phi \leftarrow \{\}$;
3    $l \leftarrow 0$;
4    $\Phi_l \leftarrow \text{BasicFeatures}(D)$;
5    $\Phi \leftarrow \Phi \cup \Phi_l$;
6    \textbf{while} $\Phi_l \neq \emptyset$ \textbf{and} $l < L_{max}$ \textbf{do}
7           $\Phi' \leftarrow \text{ExpandFeatures}(\Phi_l, D)$;
8           $l \leftarrow l + 1$;
9           $\Phi_l \leftarrow \text{SelLevFeatures}(\Phi', \Phi, \text{runtime}, D, c_l)$;
10          $\Phi \leftarrow \Phi \cup \Phi_l$;
11      \textbf{end}
12 end
\end{verbatim}

manner. In our feature selection algorithm, the classes are recursively constructed, beginning from level 0, where basic concept classes are built. Examples of the basic concepts are: all objects in an instance, the objects of a particular type, and the parameters of a unary predicate in the initial state. Notice that the ‘All objects’ class is necessary as it guarantees considering all objects in the next levels. For example, in domains where types are not defined, objects that do not participate in unary relations will not be included in any class, and hence will not appear in the next level. Subsequent levels’ classes are recursively constructed from previous levels classes using the language connectors. The language connectors are: the intersection between two classes, the complement of a class, and the relational extension. Relational extension classes represent the objects that participate with an object of another class as arguments of a particular predicate symbol in either the initial state or the goal. The features of the problem instances correspond to the selected classes. They are, then, measured by counting the number of objects belonging to the classes.

Algorithm 3: ExpandFeatures

\textbf{input} : Previous level features: $\Phi_l$, domain: $D = (\Psi, O)$

\textbf{output}: New features: $\Phi_{\text{new}}$

\begin{verbatim}
1 begin
2    $\Phi_{\text{new}} \leftarrow \{\}$;
3    \textbf{foreach} $\phi \in \Phi_l$ \textbf{do}
4        $\Phi_{\text{new}} \leftarrow \Phi_{\text{new}} \cup \text{Complement}(\phi)$
5      \textbf{end}
6    \textbf{foreach} $\phi_1, \phi_2 \in \Phi_l$ \textbf{do}
7        $\Phi_{\text{new}} \leftarrow \Phi_{\text{new}} \cup \text{Intersection}(\phi_1, \phi_2)$
8      \textbf{end}
9    \textbf{foreach} $\phi \in \Phi_l$ \textbf{do}
10       \textbf{foreach} $p \in \Psi$ \textbf{do}
11          $\Phi_{\text{new}} \leftarrow \Phi_{\text{new}} \cup \text{RelationalExtension}(p, \phi)$
12        \textbf{end}
13    \textbf{end}
14    $\Phi_{\text{new}} \leftarrow \text{DeleteZero}(\Phi_{\text{new}})$;
15 end
\end{verbatim}
Chapter 3. Learning to Choose Instance-Specific Macro Operators

Algorithm 4: SelLevFeatures

input: New features: $\Phi_{\text{new}}$, All feature classes: $\Phi$, runtimes of selected r-domains: $\text{runtime}$, domain: $D = (\Psi, O)$, maximum number of features $c_l$

output: Selected new features: $\Phi_{\text{sel}}$

1 begin
2  $\Phi_{\text{sel}} \leftarrow \{\}$;
3  $r_{\text{max}} \leftarrow \text{FindRSQ}(\Phi, S, D, \text{runtime})$;
4  stop $\leftarrow$ false;
5  while $|\Phi_{\text{sel}}| < c_l$ and not stop do
6    stop $\leftarrow$ true;
7    foreach $\phi \in \Phi_{\text{new}}$ where $\phi \notin \Phi_{\text{sel}}$ do
8      $r \leftarrow \text{FindRSQ}(\phi \cup \Phi_{\text{sel}} \cup \Phi, S, D, \text{runtime})$;
9      if $r > r_{\text{max}}$ then
10         $\phi_{\text{max}} \leftarrow \phi$;
11         $r_{\text{max}} \leftarrow r$;
12         stop $\leftarrow$ false;
13     end
14    end
15    if not stop then
16      $\Phi_{\text{sel}} \leftarrow \Phi_{\text{sel}} \cup \{\phi_{\text{max}}\}$
17  end
18 end
19 end

Algorithm 5: FindRSQ

input: Features: $\Phi$, macros: $S$, domain: $D = (\Psi, O)$, runtime lists: $\text{runtime}$

output: Total R-squared value: $r_{\text{total}}$

1 begin
2  $r_{\text{total}} \leftarrow R^2(\text{runtime}(D, S, \emptyset), \Phi)$;
3  foreach $m \in S$ do
4    $r \leftarrow R^2(\text{runtime}(D_{\{m\}}, S, \emptyset), \Phi)$;
5    $r_{\text{total}} \leftarrow r_{\text{total}} + r$;
6  end
7 end

Our feature selector is shown in Algorithm 2, Algorithm 3, Algorithm 4, and Algorithm 5. It is a simplified and modified version of Yoon et al.‘s heuristic learning algorithm [90]. Yoon et al.‘s algorithm uses the planning graph structure to build more classes, while we use only the initial state’s and the goal’s atoms. We did not use the partial states that appear in the planning graph as in Yoon et al.‘s algorithm because we want to reduce the overhead of building the planning graph, and since the facts directly extracted from the planning graph are not clearly related to the real effect of the macros. We also use a different method to filter the features.

The algorithm selects the features that maximize the linear fit with respect to the runtime of some selected r-domains. The linear fit is measured by finding the coefficient of determination ($R^2$) value [82]. We measure the $R^2$ value of the features measurement of each instance with the corresponding runtime of a selected r-domain (the original domain for example). The set of features that has the highest $R^2$ value with the selected r-domains is the one that can best predict the r-domains’ runtime. Features that score the highest $R^2$ value on all of the selected r-domains are chosen. To make the process more
manageable, we restrict the r-domains to those where only one macro is added to the domain, and to the original domain. So, for the identification of the features, we restrict the number of these r-domains to a linear \(O(n)\).

As shown in Algorithm 2, at the initial level, \(l_0\), the basic features are selected and extracted. In the BasicFeatures subroutine we generate one class that represents all objects of the instance, and other classes for the objects that appear as the arguments of a unary predicate in the goal or the initial states. The classes may include types since they can be seen as static unary predicates in the initial state.

At a subsequent level, we expand the features generated in the previous level using the ExpandFeatures subroutine, and then filter them using SelLevFeatures. In ExpandFeatures, as shown in Algorithm 3, we generate the complement class using the subroutine Complement, the intersection class using Intersection, and the relational extension classes using RelationalExtension from the old features. Then, the subroutine DeleteZero is called to remove from the new features those that have the measurement ‘0’ (or classes that have no objects) on all training instances. Such features are removed because they are clearly not useful for prediction, and also because they cannot participate further in generating new features using the connectives described above.

After the new features are created, they are filtered as shown in Algorithm 4. In SelLevFeatures, a constant number of classes, \(c_l\), is selected from the new classes generated at the current level \(l\). A class is selected if the runtime of the r-domain on the training instances has a better linear fit with that class’s feature measurement (represented by the number of the class’s object) in the training instances. The FindRSQ subroutine measures such fit in Algorithm 5. In FindRSQ, the \(R^2\) value of the feature vector on the runtime of the planner on a selected r-domain is measured. The sum of the \(R^2\) values for all r-domains is returned as an indicator of the linear fit of the feature. The class that has the maximum \(R^2\) sum is selected with ties broken randomly. We select \(c_l\) classes at each level \(l\) as long as there are classes that improve the linear fit.

Finally, Algorithm 2 returns the selected features. In our implementation, the algorithm also extracts the values of these selected features from the instances, since the feature measurements of the training instances are used in the filtering.

### 3.4 Claims and Research Question

Two reasons lead us to believe that the approach described above can be useful in improving planning:

1. Using a fixed macro set for all problem instances in the domain does not account for the differences between instances: not all macros work well with all instances. For example, certain sequences of actions that might be frequently found in the plans are not found in every solution plan of the instances. In fact, using all the available macros may be produce worse performance for some instances.

2. Machine learning can be used to find a link between the instance-specific macro sets and problem instances’ features. In fact, machine learning has been successfully used to establish a range of domain-knowledge [92, 36]. For example, supervised learning has been successfully used to predict, based on simple features, the planner performance on problem instances [76].

Therefore, we claim that the ability to learn to choose instance-specific macros is promising, in that it is likely to improve planning speed, as compared to the fixed-macro-set scheme.
Following that claim, we conducted a feasibility experiment to evaluate the ability of instance-specific predictors, in general. We did this by looking at the performance of the ‘perfect predictor,’ or the oracle predictor which knows in advance what the best r-domain choice is for each problem instance. We also look for one r-domain that has the lowest average runtime in the training instances, and we call it the ‘best-on-average’. Then, we compared the imaginary predictor’s performance to the fixed best-on-average r-domain’s performance.

### 3.4.1 Feasibility Experiment

To begin testing the feasibility of the instance-specific predictors, we conducted an experiment to test whether the perfect predictor’s performance is indeed significantly better than the performance of the fixed best-on-average r-domain. If this is not the case, then using any instance-specific prediction model would not outperform the best fixed r-domain; we could simply fix the domain model of all problem instances to the best-on-average r-domain, yielding a performance as good as perfect instance-specific predictor.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Perfect predictor</th>
<th>Best-on-average</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics</td>
<td>1.04</td>
<td>1.08</td>
</tr>
<tr>
<td>blocksworld</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td>mprime</td>
<td>*0.29</td>
<td>7.64</td>
</tr>
</tbody>
</table>

Table 3.1: Feasibility Experiment. The number in parentheses is the number of test instances for each domain. The other numbers in the cells represent the average runtime in seconds of the reformulation, and the asterisk (*) means that there is a significant difference with $p \leq 0.05$ using a paired $t$-test [19].

Table 3.1 shows the results of our feasibility experiment. We tested the performance of the FF planner on different domains: logistics, blocksworld, and mprime. We used manual macros that are described in Table 3.3. In each domain, we ran FF on a small number of instances generated randomly on all r-domains that can result from adding the macros, and we registered the runtimes. The results show that the difference in performance between the perfect predictor and the best-on-average r-domain was not significant in logistics and blocksworld, but that it was significant in mprime. We noticed that the first two domains are considered ‘easy’ for FF: all of their instances are solved in its local search phase, EHC [44]. See Section 2.1.3 for more details on FF’s EHC phase.

These results suggest that if we use an instance-specific macro learning approach with the FF planner for domains solvable by FF’s EHC, then the best instance-specific predictor performs only slightly better than the fixed macro scheme. An explanation of this phenomenon is that FF can effectively handle the increase in the branching factor, caused by macro addition, in these domains: logistics and blocksworld. The actions in these domains are reversible [44]. The search cost is not exponentially related to the increase in branching factor in domains with reversible actions, as there is no need to backtrack to a previous action choice. Any slow down in such domains is usually caused when the planner falls into a local minima or a plateaux; in both of these cases, the heuristic fails to guide the search and a backtracking-based search is the only means to escape. Once the right set of macros, ones that can escape these local minima or plateaux, is found, the need for backtracking should vanish. Therefore, adding more macros to the domain, beside those that escape the local minima, does not harm the search significantly. This means that there is only one set of macros that makes a large difference in performance, which is the one that escapes plateaux and local minima; once that set of macros is added,
other macros can only change the performance in a marginal way. Therefore, we propose that instance-specific macros may not be as effective in such domains, and that only a fixed set of macros is useful in these domains.

3.4.2 Hypothesis

Based on the arguments and claims presented above, we make the following hypothesis.

**Hypothesis 1.1** Learned instance-specific macros can be significantly faster than fixed macro sets. However, in the special case of the FF planner and easy domains for FF, the performance of the instance-specific approach is not expected to be better than the best fixed macro sets.

**Justification** Finding the same macro in the solution plans of all instances is unlikely, because instances differ substantially. This suggests that there is an advantage to using an instance-specific macro approach. It is difficult to know in advance whether a macro is appropriate for a problem instance, so previous experience of running the macro on similar instances can be useful.

The results of the feasibility experiment show that there is a significant opportunity to improve the performance of the best fixed r-domain by using an instance-specific predictor in some domains. However, in the domains that FF solves in its local search, the branching factor and the number of explored nodes are not necessarily related exponentially. In these domains, all actions are reversible, so there is no need for backtracking, and FF can solve the instance of these domains in its local search. The only thing that can slow down this local search is in the depth of the solution, and the existence of local minima, in which FF can get stuck for a long time since it can only use a blind search. Once we find the macros that can escape these cases (see Section 2.3), adding more macros will only produce a marginal change in the performance. This is because increasing the branching factor can harm the performance only marginally in the local search phase [35]. Therefore, choosing the best macro subsets for each individual instance is likely to produce performance that is only slightly better than that gained using the best fixed subset that we can identify.

3.5 Experimental Design

The experiments are intended to test the hypothesis by evaluating the performance of the instance-specific prediction models and comparing them to the best fixed macro-augmented r-domain. We compare the average planning time of a number of common reformulations and r-domains on test instances using different benchmark domains, planners, and macro sources. In the next subsection, we describe the components of the experiment: the initial macros, the planners, the learning, the domains, the compared r-domains and reformulations, and the tests.

3.5.1 Settings

**Hardware and Software Details**

Our experiment was conducted on a Beowulf cluster, where each node consists of two Dual Core AMD 270 CPUs, 4 GB of main memory, and 80 GB of disk space. All machines run Red Hat Enterprise Linux 4. We used the Ruby scripting language to write all the code.
Planners

We experiment with two state-of-the-art planners: version 2.3 of the FF planner [49], which is a state-space planner, and version 2.2 of the VHPOP planner [91], which is a partial-order-plan-space planner. See Section 2.1.3 for more details on FF and VHPOP. We downloaded FF directly from the author’s web page.\footnote{We downloaded FF from: \url{http://www.loria.fr/~hoffmanj/ff.html}, on October 27, 2007. We later found that FF website was moved to \url{http://fai.cs.uni-saarland.de/hoffmann/ff.html}.} We downloaded VHPOP from its official page.\footnote{\url{http://code.google.com/p/vhpop/}, on June 8, 2011.}

Planning Domains

We use six benchmark domains from the International Planning Competition (IPC). We categorize the domains to easy for FF domains, which are: logistics and blocksworld, and hard for FF domains, which are: mprime, mystery, freecell, and pipesworld. These domains are chosen to cover the spectrum of STRIPS domains described by Hoffmann [44]. The easy domains are solved by the FF planner using its EHC (local search) phase only [44, 45, 46]. For more information about the domains’ operators and predicates, please refer to Appendix A.

Problem Instances

The training and test instances were generated by artificial problem generators. For the logistics, blocksworld, mystery, mprime, and freecell domains we used the generator associated with the FF package.\footnote{The generators were downloaded from the FF planner’s website, \url{http://www.loria.fr/~hoffmanj/ff-domains.html}, on October 27, 2007.} For the pipesworld (no-tankage) domain, we could not find an official generator,\footnote{We contacted the authors of the domain but we could not obtain the generator.} so we wrote one from scratch: we used the problem instances from the IPC-4 as a guide. We maintained the same location and pipe connection topology, types of pipe, products and their interactions exactly as in IPC-4 instances, and we randomized the location of the batches. For more details on the problem generators see Appendix C.

<table>
<thead>
<tr>
<th>Domain</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics</td>
<td>$a \in [1, 3], c \in [4, 6], s \in [4, 6], p \in [20, 52]$</td>
</tr>
<tr>
<td>blocksworld</td>
<td>$n \in [2, 50]$</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>$p \in [2, 4], b \in [1, 9], g \in [1, 3]$</td>
</tr>
<tr>
<td>mprime</td>
<td>$l \in [5, 5], f \in [30, 30], s \in [1, 2], v \in [1, 2], c \in [2, 8]$</td>
</tr>
<tr>
<td>mystery</td>
<td>$l \in [5, 5], f \in [30, 30], s \in [1, 2], v \in [1, 2], c \in [2, 8]$</td>
</tr>
<tr>
<td>freecell</td>
<td>$f \in [2, 4], c \in [2, 8], s \in [2, 4], l \in [3, 13]$</td>
</tr>
</tbody>
</table>

Table 3.2: The parameters used to generate the instances and the training and test set size ranges. A description of the parameters can be found with each domain generator. For the pipesworld, $p,b,g$ represent the number of pipes, number of extra batches (added to a fixed number of batches that depends on $p$), and number of goals, respectively.

The parameters used to generate the problems of each domain are shown in Table 3.2.

Macro sources

We split our experiments over three different macro sources, using one source for each experiment. We use Wizard’s chunking phase [67], Macro-FF’s CA-ED version [11], and some manually constructed
### Domain Macro operators

<table>
<thead>
<tr>
<th>Domain</th>
<th>Macro operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics</td>
<td>unload-airplane-load-truck, load-truck-drive-truck-unload-truck</td>
</tr>
<tr>
<td>blocksword</td>
<td>pickup-stack, unstack-putdown, unstack-stack, unstack-putdown-unstack-putdown</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>push-start-push-end, pop-start-pop-end, push-unitarypipe-pop-unitarypipe, push-start-push-end-2</td>
</tr>
<tr>
<td>mprime</td>
<td>load-move-unload, load-move, move-unload, donate-move</td>
</tr>
<tr>
<td>mystery</td>
<td>load-move-unload, load-move, move-unload</td>
</tr>
<tr>
<td>freecell</td>
<td>sendtofree-sendtohome, sendtofree-move-colfromfreecell, sendtofree-sendtofree</td>
</tr>
</tbody>
</table>

Table 3.3: Our manually constructed macros.

Macros. In the following we describe each source:

**Wizard:** We used the version of Wizard that was used in the IPC. We obtained it directly from the authors. To get the initial set of macros, we ran the chunking phase of Wizard only to get a set of individually useful macros. We compare Wizard’s performance to our predictors by running Wizard’s bunching phase using the resulting macros to find a useful subset of these macros. As parameters to the genetic algorithm, we used five epochs, and a population of size 9. We changed the default macro utility threshold in some domains to get macros in all domains we try. The utility threshold was set to 150, 100, 100, 100, and 15 for logistics, blocksworld, mystery, pipesworld, and freecell domains, respectively. The mprime domain that we have is not compatible with Wizard. Refer to Section 2.3.4 for more details on Wizard.

**Macro-FF** We used the CA-ED version of Macro-FF (see Section 2.3.3 for details.) We obtained the macros from the author’s website.¹

**Manual Macros** We tried to find macros that are intuitively useful and can improve the speed of the search. Table 3.3 shows the schema of our macros. For more information about the exact construction of the macros refer to Appendix B.

We consider \( n \) macros in the initial macro set \( S \). We set an upper limit to \( n \) to reduce the total number of macro combinations in each domain. In our experiment \( n \leq 5 \).

**Learning Models**

We use the machine learning tool WEKA [86] for the learning. We use the M5P regression algorithm [85] for the time model, and the SMO algorithm [73] with quadratic kernel for the direct model. The reason for using such non-linear algorithms is that the runtime of the r-domains tend to follow an exponential or high polynomial curve rather than a linear curve.

In the time model, the number of small predictors to consult for a new problem instance is exponential to the number of macros. Therefore we limit the number of r-domain runtime predictors for the time predictor to the \( r = 8 \) top ranked r-domains from the training.

Chapter 3. Learning to Choose Instance-Specific Macro Operators

Features

We used the three feature sources mentioned above: the taxonomic syntax, the simple state statistics, and the manual domain-specific features. We use the first and second feature sources for all the domains, and we use the third for the freecell domains only. The selected features of each domain in the experiments is found in Appendix D. We refer to the feature vector by \( X \).

Although the selection and extracting of the features is achieved in one step for the training and testing instances, we found the time needed for extracting and measuring the features negligible in our experiments.

Timeout Handling

In every run of a problem instance, we set the cut-off time \( c \) to one hour. The runtime of timed-out instances is registered as one hour and the test instance for which all r-domains timed out is not considered for the evaluation, since it does not differentiate among the r-domains. This particular issue will be discussed in more detail in Section 3.7.2. Instances that could not be solved due to errors (e.g., insufficient memory) are considered as timed out.

Utility

We define the utility under the instance-specific macro learning. We need the utility function to describe the reformulations that will be compared in this experiment. Given a planner \( P \), a domain \( D \), a problem instance \( i \), and a set of macro operators \( S \), the utility of an r-domain \( D_{M,S,\phi} \) on the problem instance \( i \) is defined as follows:

\[
U_{(i,P)}(D_{M,S,\phi}) = \begin{cases} 
\text{runtime}_P(i, D_{M,S,\phi}) & \text{if } P \text{ naturally halts without interruption.} \\
-c & \text{otherwise}
\end{cases}
\]

Where \( \text{runtime}_P \) is the real CPU time taken by planner \( P \) to find a plan that solves the problem instance \( i \) or to decide that there is no plan that solves it, and \( c \) is the duration after which the execution of the planner is interrupted as discussed earlier.

We found the time required for consulting the prediction models and the time needed to extract the features negligible. So, we did not include these times in the utility.

Models to Compare

Given a planner \( P \), a planning domain \( D \), a set of macros \( S \), and a set of training and testing instances \( I_{\text{train}} \) and \( I_{\text{test}} \) generated from \( D \), we define the following reformulations:

**ORIG** is the original domain \( D \) without any macros added.

\[
\forall i \in I_{\text{test}}: \text{ORIG}(i) = D_{\emptyset, S, \emptyset}.
\]

**BOA** is the best macro-augmented r-domain on average based on complete evaluation of all macro-augmented r-domains on the training instances.

\[
\forall i \in I_{\text{test}}: \text{BOA}(i) = D_{M', S, \emptyset}
\]

where \( M' = \arg \max_M \hat{U}_{(I_{\text{train}}, P)}(D_{M, S, \emptyset}) \).
**FULL** is the domain augmented with all available macro operators:
\[ \forall i \in I_{test}: \text{FULL}(i) = D_{S,S,\emptyset}. \]

**TIME** is the time predictor. It is a set of small predictors that predict the runtime of the problem instance on each macro-augmented domain:
\[ \forall i \in I_{test}: \text{TIME}(i) = D_{\arg\max M \text{Reg}(i, T_{X,U_M}, S, \emptyset), S, \emptyset} \]
where \( \text{Reg} \) is a regression function that uses the M5P regression algorithm on a training set \( T_{X,U_M} \) to predict the utility of \( D_{M,S,\emptyset} \).
\[ T_{X,U_M} = \{(X(i), U_M(i)) | U_M(i) = U(i,P)(D_{M,S,\emptyset}), i \in I_{train}\} \]
See Section 2.2.1 for more details on the training set definition.

**DIR** is the direct predictor:
\[ \forall i \in I_{test}: \text{DIR}(i) = D_{\text{Classify}(i, T_{X,BEST}, S, \emptyset)} \]
where \( \text{Classify} \) is a classifier function that uses the SVM-SMO classification algorithm with a quadratic kernel on the training set \( T_{X,BEST} \) to predict the best r-domain for instance \( i \), where:
\[ T_{X,BEST} = \{(X(i), \text{BEST}(i)) | \text{BEST}(i) = \arg\max M \text{U}(i,P)(D_{M,S,\emptyset}), i \in I_{train}\} \]

**PERF** is the perfect predictor. It is an imaginary instance-specific macro-based reformulation that achieves the highest utility on the test problem instance based on exhaustive evaluation of all macro-augmented domains.
\[ \forall i \in I_{test}: \text{PERF}(i) = D_{M'(i),S,\emptyset} \]
where \( M'(i) = \arg\max M \text{U}(i,P)(D_{M,S,\emptyset}) \).

**WIZ** is the r-domain that results from adding the macros suggested by Wizard’s bunching phase.
\[ \forall i \in I_{test}: \text{WIZ}(i) = D_{\text{Wizard}_b(S),S,\emptyset} \]
where \( \text{Wizard}_b(S) \) is the macro subset returned by the bunching phase of the Wizard tool run on the macro \( S \) that result from the chunking phase \( \text{Wizard}_c \). Note that \( \text{Wizard}_b \) is defined only when \( \text{Wizard}_c = S \).

**MFF** is the r-domain of Macro-FF’s CA-ED. In the experiments where Macro-FF macros are used, MFF is defined as follows:
\[ \forall i \in I_{test}: \text{MFF}(i) = \text{FULL}(i) \]

### 3.5.2 Tests

We would like to understand the generality of the results in terms of the planner and the macro source. Therefore, we conduct five experiments:

1. With FF and macros from Wizard’s chunking phase.
2. With FF and macros from Macro-FF’s CA-ED.
3. With FF and manually constructed macros.
4. With VHPOP and manually constructed macros.
5. With VHPOP and Macro-FF macros.

In each of these experiments, we look at the average runtime of the planner on test instances using the macro-based reformulations and r-domains described above.

These experiments are to test hypothesis 1.1. Generally in the experiments where PERF is significantly faster than BOA, we expect to find DIR and/or TIME significantly faster than BOA. In these experiments, there is a chance of improving the planning speed significantly by using macro remodeling. In experiments 1, 2, and 3, we conduct a complete instance-specific macro learning experiment using FF and easy and hard domains. However, using the results of the feasibility experiment from Section 3.4.1, we expect to find PERF significantly faster than BOA in hard domains, and we expect to find PERF not significantly faster than the BOA, or only faster by a small margin, in easy domains.

### 3.6 Experimental Results

The results in general show that our predictors (TIME and DIR) were significantly faster than BOA in a number of domains. In FF’s easy domains, the difference between PERF and BOA was small.

To conclude the significance of difference between the compared models, we used a paired \( t \)-test \[19\] with \( p \leq 0.05 \). We say that a reformulation is significantly better or worse than another only if we find a statistically significant difference using to the \( t \)-test.

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA</th>
<th>FULL</th>
<th>TIME</th>
<th>DIR</th>
<th>WIZ</th>
<th>PERF</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics (224)</td>
<td>1.20</td>
<td>0.77</td>
<td>1.30</td>
<td>+0.80</td>
<td>+0.77</td>
<td>1.21</td>
<td>+0.68</td>
</tr>
<tr>
<td>blocksworld-4ops (250)</td>
<td>1278.00</td>
<td>0.56</td>
<td>0.56</td>
<td>46.94</td>
<td>+22.14</td>
<td>100.82</td>
<td>+0.31</td>
</tr>
<tr>
<td>pipesworld-nt (213)</td>
<td>163.51</td>
<td>167.47</td>
<td>174.04</td>
<td>147.06</td>
<td>184.57</td>
<td>152.79</td>
<td>101.25</td>
</tr>
<tr>
<td>mystery-5 (281)</td>
<td>155.64</td>
<td>155.64</td>
<td>473.46</td>
<td>186.81</td>
<td>155.33</td>
<td>143.98</td>
<td>20.26</td>
</tr>
<tr>
<td>freecell-A (196)</td>
<td>533.99</td>
<td>533.35</td>
<td>436.03</td>
<td>+339.35</td>
<td>517.42</td>
<td>543.43</td>
<td>+32.17</td>
</tr>
<tr>
<td>FF-MACROFF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocksworld-4ops (250)</td>
<td>1267.80</td>
<td>3.04</td>
<td>3.04</td>
<td>2.91</td>
<td>-93.2</td>
<td>N/A</td>
<td>2.02</td>
</tr>
<tr>
<td>pipesworld-nt (122)</td>
<td>413.32</td>
<td>413.32</td>
<td>424.90</td>
<td>+219.98</td>
<td>425.60</td>
<td>N/A</td>
<td>48.91</td>
</tr>
<tr>
<td>freecell-A (247)</td>
<td>371.27</td>
<td>174.79</td>
<td>174.79</td>
<td>157.07</td>
<td>243.17</td>
<td>N/A</td>
<td>36.14</td>
</tr>
<tr>
<td>FF-MANUAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logistics (192)</td>
<td>3.80</td>
<td>3.80</td>
<td>2136.43</td>
<td>9.30</td>
<td>3.85</td>
<td>N/A</td>
<td>+3.59</td>
</tr>
<tr>
<td>blocksworld-4ops (200)</td>
<td>947.25</td>
<td>3.24</td>
<td>10.57</td>
<td>-69.19</td>
<td>50.08</td>
<td>N/A</td>
<td>*1.77</td>
</tr>
<tr>
<td>pipesworld-nt (296)</td>
<td>567.91</td>
<td>465.38</td>
<td>471.41</td>
<td>455.53</td>
<td>580.40</td>
<td>N/A</td>
<td>+17.63</td>
</tr>
<tr>
<td>mystery-5 (296)</td>
<td>239.48</td>
<td>239.48</td>
<td>201.36</td>
<td>162.77</td>
<td>146.00</td>
<td>N/A</td>
<td>+39.46</td>
</tr>
<tr>
<td>mprime-5 (320)</td>
<td>521.53</td>
<td>8.81</td>
<td>46.94</td>
<td>49.15</td>
<td>*3.33</td>
<td>N/A</td>
<td>+0.29</td>
</tr>
<tr>
<td>freecell-A (150)</td>
<td>296.43</td>
<td>325.21</td>
<td>558.36</td>
<td>+278.51</td>
<td>377.01</td>
<td>N/A</td>
<td>+58.04</td>
</tr>
<tr>
<td>VHPOP-MACROFF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocksworld-4ops (181)</td>
<td>1140.01</td>
<td>200.35</td>
<td>200.35</td>
<td>260.14</td>
<td>262.28</td>
<td>N/A</td>
<td>2.78</td>
</tr>
<tr>
<td>pipesworld-nt (101)</td>
<td>387.44</td>
<td>387.44</td>
<td>502.53</td>
<td>+278.51</td>
<td>377.01</td>
<td>N/A</td>
<td>+32.32</td>
</tr>
<tr>
<td>freecell-B (251)</td>
<td>326.07</td>
<td>326.07</td>
<td>558.36</td>
<td>+278.81</td>
<td>+199.34</td>
<td>N/A</td>
<td>+94.31</td>
</tr>
<tr>
<td>VHPOP-MANUAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logistics (247)</td>
<td>316.06</td>
<td>316.06</td>
<td>2794.21</td>
<td>+233.10</td>
<td>282.26</td>
<td>N/A</td>
<td>+163.55</td>
</tr>
<tr>
<td>blocksworld-4ops (187)</td>
<td>1217.10</td>
<td>157.14</td>
<td>332.81</td>
<td>138.12</td>
<td>176.65</td>
<td>N/A</td>
<td>+2.48</td>
</tr>
<tr>
<td>pipesworld-nt (105)</td>
<td>282.92</td>
<td>282.92</td>
<td>2883.67</td>
<td>282.92</td>
<td>282.92</td>
<td>N/A</td>
<td>+20.51</td>
</tr>
<tr>
<td>mystery-5 (210)</td>
<td>418.79</td>
<td>83.23</td>
<td>854.99</td>
<td>*177.59</td>
<td>180.61</td>
<td>N/A</td>
<td>+13.04</td>
</tr>
<tr>
<td>mprime-5 (200)</td>
<td>2222.08</td>
<td>379.01</td>
<td>3507.06</td>
<td>531.90</td>
<td>1563.37</td>
<td>N/A</td>
<td>+179.49</td>
</tr>
<tr>
<td>freecell-B (221)</td>
<td>1154.35</td>
<td>214.58</td>
<td>214.58</td>
<td>174.20</td>
<td>192.34</td>
<td>N/A</td>
<td>+47.90</td>
</tr>
</tbody>
</table>

Table 3.4: Average runtime (in seconds) of each tested reformulation constructed from different macro sources using different planners. There are five experiments: one for every pair of a planner and a macro source (except VHPOP-WIZARD.) The asterisk (*) (circle (◦)) means that the predictor was significantly faster (slower) than BOA with \( p \leq 0.05 \) and the plus (+) (minus (−)) means that the predictor was significantly faster (slower) than WIZ or MFF. The number in parentheses is the number of test instances. MFF is defined only in FF-MACROFF and VHPOP-MACROFF.

Table 3.4 shows the average runtime of the r-domains and reformulations in the five experiments. In the first experiment, FF-WIZARD, we found TIME significantly faster than BOA in the freecell...
domain, but BOA was not significantly different from TIME and DIR in the other cases. It is important to mention that the macros produced by Wizard’s chunking phase in pipesworld, mystery, and freecell had low quality according to Wizard’s utility measure. This might have affected PERF in pipesworld. On the easy domains (logistics and blocksworld), we found that PERF was significantly better than BOA but with small difference. In the hard domains (pipesworld, mystery, freecell), there was a large significant difference between PERF and BOA. Compared to WIZ, DIR was significantly faster in 2 out of 5 domains, and TIME was faster in 2 domains. WIZ was not faster than DIR or TIME in any domain. WIZ was not statistically different from BOA except in blocksworld. ORIG had close performance to the realistic r-domains and reformulations (i.e., not including PERF) except in blocksworld, where it was much worse than BOA, TIME, DIR, and WIZ.

In the second experiment, FF-MACROFF, we used the macros produced by Macro-FF for blocksworld, freecell, and pipesworld-nt. BOA was comparable to the predictors. TIME was significantly faster than BOA in pipesworld, but BOA was faster than DIR in blocksworld. In freecell, there was no significant difference between the predictors and BOA, probably due to the nature of the single macro (sendtohome-sendtohome) used in this domain. This macro was used on most instances and had either a small harmful effect or a substantial positive effect. In other words, it was not suitable for instance-specific remodeling, even though PERF was much better than BOA. There was a small difference between PERF and BOA in the easy domain (blocksworld), and a noticeably large difference in the other domains. MFF was faster than DIR in blocksworld. TIME was significantly faster than MFF in pipesworld. The predictors were as fast as MFF in the other cases. ORIG was often worse than the predictors and BOA.

In the third experiment, FF-MANUAL, we found no difference between the predictors and BOA in the easy domains, except in blocksworld where BOA was faster than TIME. In the hard domains, we found DIR significantly faster than BOA in 3 out of 4 domains: mystery, mprime and freecell. BOA was not different from DIR in pipesworld, and was not significantly faster than TIME. There was a small statistical difference between PERF and BOA on logistics and blocksworld. On the hard domains, there was a large significant differences between PERF and BOA. Specifically, the ratio between BOA’s and PERF’s runtimes in the easy domains were: 1.05, and 1.83, and in the hard domains were: 26.40, 6.07, 30.38, and 13.57. FULL and ORIG were often worse than BOA and the predictors.

In the fourth experiment, VHPOP-MACROFF, we found that DIR was significantly faster than BOA in the freecell domain, and BOA was not faster than any predictor. PERF was many times faster than BOA in all of domains. The prediction models were significantly faster than MFF/FULL in 3 out of 6 cases. In the other 3 cases, the predictors were not significantly slower than MFF. The predictors were also faster than ORIG in 5 out of the 6 cases.

In the fifth experiment, VHPOP-MANUAL, we found TIME significantly faster than BOA in logistics. However, the predictors’ performance was much worse than BOA in mystery, and DIR was worse than BOA in mprime. We found that the PERF was significantly faster than BOA in all domains. In many domains in VHPOP experiments, VHPOP was not able to solve a considerable number of problem instances due to internal errors or lack of memory. There were more such errors in the VHPOP-MANUAL experiment than the VHPOP-MACROFF experiment since there was more macros and hence larger r-domains. These errors negatively affected the results as they harmed the reliability of the predictor, since the instances that halt with error are registered as timed-out for the training and the testing.
The predictors’ macros were often better than the fixed macro subset chosen by the macro acquisition tools, Wizard and Macro-FF. While this may seem an unfair comparison since the predictors utilize a complete evaluation of all the macro subsets, it is important to note that this evaluation is achieved only once offline. Furthermore, as the number of initial macros was usually small, the combinatorics are manageable.

3.6.1 Results Summary

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Predictor</th>
<th>vs. BOA</th>
<th>vs. macro tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF-WIZARD (5)</td>
<td>DIR</td>
<td>(+0, −0)</td>
<td>(+2, −0)</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>(+1, −0)</td>
<td>(+2, −0)</td>
</tr>
<tr>
<td>FF-MACROFF (3)</td>
<td>DIR</td>
<td>(+0, −1)</td>
<td>(+0, −1)</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>(+1, −0)</td>
<td>(+1, −0)</td>
</tr>
<tr>
<td>FF-MANUAL (6)</td>
<td>DIR</td>
<td>(+3, −0)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>(+0, −1)</td>
<td></td>
</tr>
<tr>
<td>VHPOP-MACROFF (3)</td>
<td>DIR</td>
<td>(+1, −0)</td>
<td>(+1, −0)</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>(+0, −0)</td>
<td>(+2, −0)</td>
</tr>
<tr>
<td>VHPOP-MANUAL (6)</td>
<td>DIR</td>
<td>(+0, −2)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>(+1, −1)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: Comparing the predictors to BOA and the macros tools in terms of the number of domains in which there was a significant difference in each of the five experiments. The number in parentheses in front of the experiment name is the number of domains in the experiment. In each cell entry (+x, −y), x and y represent the number of domains where the predictor was significantly better and worse, respectively, than BOA or the macro tool.

Table 3.5 summarizes important results from Table 3.4. For each experiment, it shows the number of domains in which each predictor was significantly better (or worse) than BOA and the compared macro tool. Generally, DIR was the best choice against BOA in two experiments, TIME was the best choice in two, and BOA was better than the predictors in one. TIME was the best choice against MFF in both of Macro-FF experiments, and both of TIME and DIR were better than WIZ.

3.7 Analysis and Discussion

In this section, we analyze the results by discussing their relation to our hypothesis; we will also take a deeper look at the data and explain our observations in depth. Then, we discuss the work as a whole by revisiting the design decisions and experimental choices.

3.7.1 Analysis

Our experimental results demonstrate that our automatically trained, instance-specific macro predictors can significantly outperform the best average standard fixed-macro approach in 7 out of 23 domain / planner / macro-source combinations; meanwhile it is worse in 5, two of which are easy domains. In those easy domains, there was a ceiling effect that prevented the predictors from outperforming BOA, as we discuss in the next subsections.

While this level of performance was not observed on all domains and planners, we believe that the up to 2.5 times improvement in planning time we found on some domains constitutes a significant
contribution, because it signals a new step towards the automated knowledge vision described by Simon [80], and because it achieves – using the same planner, macros and resources – a significantly faster planning speed. One of the major challenges in domain-independent planning is that, to a large degree, it either ignores previous domain knowledge, or it falls into the trap of the utility problem, which accompanies all such knowledge acquisition, as discussed in Section 3.2. By contrast, our approach automates the use of domain knowledge and attempts to fix the utility problem using machine learning.

In addition to showing that the predictors outperformed the best expected fixed-macro approaches in a number of domain, the results show that the predictors outperform the state-of-the-art macro tools. Using state-of-the-art planners, an automatic feature selector derived from the literature, and macros from state-of-the-art macro tools, our Direct predictor was shown capable of significantly outperforming Wizard in 2 out of 5 cases. Using the same settings, the Time predictor outperformed Wizard in 2 out of 5 domains, and outperformed Macro-FF in 3 out of 6 cases.

In the next subsections, we investigate the reasons behind this approach’s performance, and we discuss the cases in which it did not work. Although the results show an improvement in planning speed beyond the best-on-average fixed remodeling, we think that more light should be shed on the nature of the learning. For instance, it is important to know how strong the correlation between the discovered features and the macros is, and whether the predictors are able to detect that relation. While we have argued that this approach is important and novel, there are still combinations of planners and domains for which our predictors were not significantly better than BOA. In the next subsections, we analyze specific planner/domain combinations so as to better understand the applicability of our approach.

### Learning the Feature-to-Macro Relation

The key aspect of learning instance specific macros is to find a relation between instance features and macro performance. While our results demonstrate that the learning was successful in a number of cases, we would like to further understand the nature of such relations embodied in the predictors.

<table>
<thead>
<tr>
<th>macro set</th>
<th># best</th>
<th># correct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s = 1</td>
<td>s = 2</td>
</tr>
<tr>
<td></td>
<td>s = 1</td>
<td>s = 2</td>
</tr>
<tr>
<td>{}</td>
<td>9</td>
<td>154</td>
</tr>
<tr>
<td>{load-move-unload}</td>
<td>121</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>117</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>154</td>
<td>0</td>
</tr>
<tr>
<td>other sets</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.6: Relation between the capacity of the truck $s$, the macro sets, and the correct choice of the DIR predictor. Columns 2 and 3 are the number of instances on which the macro set was best. Columns 4 and 5 are the number of instances on which the direct predictor correctly chose the corresponding macro set.

In the FF-MANUAL/mprime combination, the macro `load-move-unload` works well. Intuitively, it is a short-cut that moves a package between locations in one step rather than three. However, when the truck can carry more than one package, `load-move-unload` will tend to result in trucks traveling partially empty and then returning to load the next package. Furthermore, the delete relaxation in this case is less informative, as the truck is in both its original and new location simultaneously, meaning that the heuristic does not realize that the return trip is necessary.

Our training results showed that the number of available spaces in the truck, $s$, was strongly correlated to the runtime of the macro: with a truck capacity of one, the `load-move-unload` macro usually performed best, but with two spaces the original domain was faster. The direct predictor was able
to learn this relation as indicated by the results in Table 3.6. The second and third columns of Table 3.6 show that the feature $s$ is highly correlated to the choice of the subsets. When $s = 1$, the set \{load-move-unload\} is the best choice for 75.63% of the test instances, and when $s = 2$, the empty set {} is the best macro set choice for 96.25% of the test instances. The fourth and fifth columns show that the direct predictor can correctly detect that correlation. When $s = 2$ and {} is the best subset to choose, DIR was 100% correct in choosing {}. When $s = 1$ and \{load-move-unload\} is the best subset, DIR was correct 96.7% of the time by choosing the subset. This result shows a high precision of the predictor on the subsets {} and \{load-move-unload\}. However, DIR was not correct when the best subset was different from these two subsets.

This high accuracy of DIR can explain why DIR outperforms BOA by at least two-fold. BOA’s macro set is \{load-move-unload, load-move, donate-move\}, which is not the best macro set on any individual instance, and it has an average runtime of 8.81 seconds as seen in Table 3.4. The empty set, {} – which was the best on 50.93% of the instances as can be calculated from Table 3.6 – has an average runtime of 521.53 seconds. The subset \{load-move-unload\}, which was the best on 39.69% of the instances, has an average runtime of 31.56 seconds. So, on 90.63% of the instances, the best subset is either {} or \{load-move-unload\}, and in the remaining 9.37%, BOA’s set is not the best. DIR, on the other hand, chose the best subset for 84.7% of the instances, and has an average runtime of 3.34 seconds. This discussion shows evidence that DIR outperformed BOA because (1) the feature selector found a useful feature, and (2) DIR used that feature to make accurate predictions.

VHPOP Errors

We found that the runs in VHPOP experiments, especially those using many macros, produced many errors. In the mprime domain, for example, we found that about 26% of the runs ended with errors. Not only do these errors affect the reliability of the predictors, since the predictors are trained on incorrect data, but they also cause a bias toward specific macro subsets. For example, we found that the BOA r-domain in mprime and mystery had fewer errors than most of the other r-domains. This means that the comparison between the r-domains is significantly biased, which may affect the choice of BOA.

There was not a single cause of these errors. In some cases, the cause was a lack of memory, and in others it was an unknown cause produced by the planner. Therefore, it is difficult to eliminate these errors; for example, we cannot be sure they will be avoided if we repeat the experiments with more memory.

Remodeling Effects On Search Topology

One important measure of the quality of a domain remodeling involves the way which the planner explores the new state space. There are a number of phenomena in the search topology that affect the search speed. Specifically, high frequency of dead-end and local minimum states is evidence that the search heuristic is inaccurate. In this context, a good r-domain would be one that forces the planner to explore states that are not on dead-ends or local minima. Depending on the type of remodeling, this can be hard or easy to achieve. Using macro addition only, we cannot generally force the planner to escape the dead-end states. We may avoid some dead-ends, however, by using macros that change the heuristic in a way that makes it prefer states that do not lead to dead-ends. Other remodeling approaches may be better in avoiding dead-ends because they remove the actions that lead to the dead-end states. We claim that remodeling by macro operator addition is unlikely to improve the frequency of dead-end states
encountered in the search. For local minima, however, we claim that macros are able to improve the search topology. This is because the local minima are escapable states: a useful sequence of actions can help us avoid them altogether. The ability to escape local minima using macros has been demonstrated by some macro-based tools such as Marvin [20], MicroHillary [35] and MACLEARN [52] (see Section 2.3 for more details). In our work, we did not include the effect on search topology in the utility of the r-domains, we only used a measure of runtime to choose the best r-domains. It is interesting however to investigate if our approach can reduce local minima situations, or, even reduce the dead-end situations.

Torchlight is a tool that can be used to give an estimate of the frequency of dead-ends and local minima in the search space of a problem instance [47, 46]. Given the parameters $s$ and $d$, Torchlight generates $s$ random states by applying a random action walk starting from the initial state up to a given depth: $d \ast h_{RPG}(s_0)$, where $h_{RPG}$ is the FF planner’s heuristic function: an estimate of the number of steps FF needs to reach the goal. In this experiment, we chose the parameter values: $s = 100, d = 5$. Torchlight tests the $s$ states for the existence of local minima and dead-ends. Torchlight outputs the number of success states (states which are not local minima or dead-ends) and the number of dead-end states. For each macro-augmented r-domain, we call Torchlight on each test instance, and we find the average percentage of success states and dead-end states for that r-domain.

Using Torchlight, we can test the effect of our predictors’ macros on the search topology. We want to know whether the instance-specific approach chooses macros that make the search easier in general. Answering this question is important since it tells us whether our macro evaluation method, which relies only on runtime evaluation, can be made more informative. If the r-domains make a difference in the topology, then perhaps using a tool such as Torchlight in the learning, in addition to measuring the runtime, would lead to a more useful method for choosing the r-domains. A large number of local minima and/or dead-ends are likely to lead to a slow down. However, although the change in the search topology can be significant, the runtime may not be affected as much. For example, if the path chosen by the heuristic to the goal is local-minima-free, even if there are many local minima around it, the search will be completed quickly. Therefore, incorporating the search topology analysis in the r-domains’ evaluation may not be straightforward.

Table 3.7 shows the results of running Torchlight on three of the tested domains using the manual macros. The results show that PERF and BOA significantly increase the number of success states compared to ORIG in the blocksworld domain, but they do not increase the number of success states in other domains. PERF and BOA are not found to reduce the number of dead-end states significantly in any domain. TIME and DIR also improved the number of success states in blocksworld, but they did not affect the success states in the other domains. In mprime, PERF, TIME and DIR increased the number of dead-ends slightly. Perhaps because the direct predictor was accurate in mprime, the relatively small number of dead-ends did not affect the performance.

These results confirm our expectations: generally, the search topology was significantly improved by adding macros in the local-minima-rich domains, but was not improved in the dead-end-rich domains. In the following we discuss a phenomenon that is related to the search topology and explain it with reference to these results.

**Easy Domains and the Ceiling Effect**

We found that PERF was significantly faster than BOA in the VHPOP experiments, and in the hard domains in FF experiment, by orders of magnitude. However, in the easy domains for the FF planner, we
Table 3.7: Torchlight analysis results on three domains with the FF planner and manually extracted macros. The numbers in each cell represents the average percentage of success states / dead-end states, respectively, according to Torchlight analysis.

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA</th>
<th>TIME</th>
<th>DIR</th>
<th>PERF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocksworld</td>
<td>27.84 / 0.00</td>
<td>79.60 / 0.00</td>
<td>77.91 / 0.00</td>
<td>70.74 / 0.00</td>
<td>64.74 / 0.00</td>
</tr>
<tr>
<td>Mystery-5</td>
<td>32.01 / 43.61</td>
<td>46.99 / 33.92</td>
<td>41.04 / 38.77</td>
<td>38.85 / 38.55</td>
<td>38.04 / 38.97</td>
</tr>
<tr>
<td>Mprime-5</td>
<td>55.51 / 5.00</td>
<td>49.65 / 7.33</td>
<td>41.10 / 14.62</td>
<td>42.76 / 24.85</td>
<td>42.89 / 24.11</td>
</tr>
</tbody>
</table>

often found little room to improve the performance by using instance-specific macro predictors, because PERF was not much faster than the best fixed macro set.\(^6\)

We believe that this ceiling effect arises, first, because the easy domains do not have dead-ends, and second, because there exists a fixed macro set that allows FF to escape local minima in all instances of the domain. We focus on one easy domain (blocksworld) to test this hypothesis.

The blocksworld domain, as discussed by Hoffmann, has no dead-ends for FF because all of its operators are reversible [46]. In the four-operator version of blocksworld, which we are testing, FF suffers from local minima while in the three-operator version FF faces no local minima with an unbounded escape path. The three-operator blocksworld is equivalent to the macro set \{pickup-stack, unstack-putdown, unstack-stack\}, with the original domain operators removed. In our results, BOA chose to augment the four-operator domain with the set \(M = \{\text{pickup-stack, unstack-putdown, pickup-stack-\_pickup-stack}\}\). Observe that unstack-stack can be achieved by \(M\) in a finite number of steps because, with the right instantiation, \((\text{unstack-putdown; pickup-stack})\) achieves the same effect as \text{unstack-stack}. The macro \text{pickup-stack-pickup-stack} is simply a double application of \text{pickup-stack} in a state, and it does not introduce additional local minima, since it is composed of operators from the three-operator blocksworld. Therefore, using an \(r\)-domain with \(M\) as the lone set of operators, any blocksworld instance can be solved without encountering local minima. Our hypothesis, therefore, is that adding \(M\) to the four-operator blocksworld results in a domain with almost no local minima with an unbounded escape paths.

If this hypothesis is true, any runtime advantage possessed by the best instance-specific macros over a BOA will be marginal. There is nothing that an instance specific macro operator can do to substantially increase performance over BOA because BOA already removes the primary component of runtime for FF in domains without dead-ends. Furthermore, if the PERF macro set already escapes local minima, using the BOA macro set, instead, will not significantly decrease performance. For example, if the original domain for a given problem instance has no local minima, the empty macro is likely to be (marginally) faster that BOA. The BOA macro set is, also, unnecessary in the sense that it is not needed in this instance to escape local minima. However, according to Finkelstein and Markovitch, adding unnecessary macro operators will slow down hill-climbing search by only a constant factor [35]. Thus the improvement of the empty set over BOA will be small.

To test this hypothesis, we use Torchlight to perform local analysis of the search topology of the macro-augmented \(r\)-domains in the blocksworld domain. Because we used the four-operators version of blocksworld domain, we expected to find a large number of local minima in ORIG (as depicted in Hoffmann’s paper [46]), and because we used operators from the three-operators blocksworld domain for the remodeling, we expected a small number of local minima in BOA. We also expect to find PERF to

\(^6\)In the easy domains of FF-WIZARD, FF-MACROFF and FF-MANUAL, the mean difference between PERF and BOA is small: 0.09, 0.21, 1.02, 0.21, and 1.47 seconds.
be no better than BOA in terms of local minima states, according to the hypothesis.

Table 3.7 shows that ORIG, BOA and PERF in blocksworld had 72.16%, 20.40%, and 35.26% average local minima states, respectively. TIME and DIR had 22.09% and 29.26% local minima. This result supports our hypothesis that the added macros remove a significant number of local minima from the four-operators blocksworld original domain, and that the number of local minima removed by PERF is not an improvement on the number removed by BOA, but in fact, PERF was slightly worse than BOA. The results on TIME and DIR support that conclusion, since they, too, did not significantly reduce the number of local minima as compared to BOA.

The same cannot be said about the hard domains, since they contain dead-ends. The dead-ends cannot be escaped and are, generally, hard to avoid using macros. To avoid dead-ends in these domains, one has to design the macros specifically for that purpose: the macros should trick the heuristic into choosing a path different from that which leads to the dead-end. However, there are many dead-ends in a given problem instance, and there are many kinds of dead-ends, each of which might require different macros if they are to be avoided. Therefore, it is difficult to find one set of macros that can avoid all dead-ends in all instances. This was also supported by the findings described in Table 3.7: neither BOA nor PERF were able to significantly reduce the number of dead-ends found in the tested hard domains, although both proved faster than ORIG.

In summary, this analysis, along with the previous subsection, suggest that using a domain topology analysis similar to the Torchlight analysis in the utility of the r-domains can be useful in estimating the best macro set and in improving the quality of training. More importantly, this analysis also provides a topological-based insights into the reasons for the performance of macros.

Macro Sources Bias

The nature of the initial set of macros significantly affects the performance of the predictors. Our approach relies on an external macro source. All of the macro sources we used focus on finding either an individual macro, or a set of macros that is expected to achieve a shorter runtime for most instances. However, these macros may not work well in an instance-specific context, where it is unlikely for one macro subset to be superior to all other subsets on all instances; if it were the case that one such subset existed, then there would be nothing to learn.

Wizard and Macro-FF choose macros based on planning speed, or on the macros’ frequency in the plans. These tools are biased toward generating macros that work well, on average, on all instances, rather than macros that work well on specific clusters of instances.

An alternative approach would be to use a macro acquisition tool to independently produce macros for different clusters of problem instances. Given a set of macros $S$, we can cluster the training instances according to their instance-specific performance on $S$; for example, we could group the instances that have the same best macro set into one cluster. To reduce the number of clusters, we can analyze the macro subsets runtimes to discover the macro subsets that dominate others in terms of performance, allowing us to remove the dominated subsets. Then, for each cluster, we can use a macro acquisition tool to produce a new set of macros that works well for that cluster’s instances. We may then be able to begin with a new set of macros, one that is already biased toward instance specific performance. However, finding a good clustering of the instances according to the macros’ performance requires more thought and experiment, since we still need to find the features that achieve such clustering.
Summary

To summarize, we can categorize the planner-macro-domain combinations according to their support to the hypothesis. Table 3.8 shows the different categories, whether they support the hypothesis, and their corresponding domain combinations. The categories are: VHPOP Errors, Bad Macros, Ceiling Effect, Better, and Others.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Planner-Macro-Domain Combinations</th>
<th>% Improvement over ORIG</th>
</tr>
</thead>
</table>
| VHPOP Errors     | VHPOP caused many error in the execution which affects the predictor’s accuracy. Instance-specific approach was not tested well. | VHPOP-MANUAL-pipesworld-nt  
                   |                                                                                             | VHPOP-MANUAL-mystery-5  
                   |                                                                                             | VHPOP-MANUAL-mprime-5  
                   |                                                                                             | VHPOP-MANUAL-freecell |  
| Bad Macros       | The macros had low quality that they are not better than the original domain. Instance-specific approach was not tested well. | FF-WIZARD-pipesworld-nt  
                   |                                                                                             | FF-WIZARD-mystery-5 |  
| Ceiling Effect   | The Perfect Predictor’s performance was not much better than BOA. Instance-specific approach was not feasible. | FF-WIZARD-logistics  
                   |                                                                                             | FF-WIZARD-blocksworld-4ops |  
                   |                                                                                             | FF-MACROFF-blocksworld-4ops |  
                   |                                                                                             | FF-MANUAL-logistics |  
                   |                                                                                             | FF-MANUAL-blocksworld-4ops |  
| Better           | A predictor was better than BOA. Instance-specific approach was useful.        | FF-WIZARD-freecell-A  
                   |                                                                                             | FF-MACROFF-pipesworld-nt |  
                   |                                                                                             | FF-MANUAL-mystery-5 |  
                   |                                                                                             | FF-MANUAL-mprime-5 |  
                   |                                                                                             | FF-MANUAL-freecell-A |  
                   |                                                                                             | VHPOP-MACROFF-freecell-B |  
                   |                                                                                             | VHPOP-MANUAL-logistics |  
| Others           | Predictors were not better or worse than BOA.                                | FF-MACROFF-freecell-A  
                   |                                                                                             | VHPOP-MACROFF-blocksworld-4ops |  
                   |                                                                                             | VHPOP-MACROFF-pipesworld-nt |  
                   |                                                                                             | VHPOP-MANUAL-blocksworld-4ops |  

Table 3.8: Analysis summary of all domain combinations. The category description explains the common property of the category’s domains, and when to use the instance-specific approach. The last column shows the percentage of improvement in mean runtime of (BOA, TIME, and DIR) compared to ORIG.

In the VHPOP Errors category, the hypothesis could not be tested on a fair basis as random effect of VHPOP’s errors made it impossible to learn. Therefore, we do not use this category in the evaluation. In the Bad Macros category, none of the original macros (i.e., the set from which we select) really improve the performance in the problem instances. A macro may improve the performance marginally in some instances, but there are no subsets of the macros that improve the performance significantly over the original domains. Therefore, we also do not use this category in the evaluation. In Wizard, it is easy to detect such macros since there is a measure of quality for each macro and macro set. In the Ceiling Effect category, we have shown that it is unlikely to improve the performance in these domains as described in Section 3.4.1. Thus, we already excluded these domains in the hypothesis. What makes the difference is the remaining two categories: Better and Others. In the Better category, instance-specific predictors can improve the performance significantly compared to BOA. In Others, the predictors were neither better nor worse than BOA. In these two categories, Table 3.8 also shows the percentage of improvement in mean runtime of BOA, TIME and DIR compared to ORIG. For each domain, we calculate the fraction of improvement to ORIG’s mean runtime for the intended reformulations. It can be noticed that in the Better category, BOA did not improve the performance much compared to ORIG while the predictors did. In the Others category, the improvement percentages of BOA and the predictors were close. It should be emphasized, however, that high percentages do not necessarily imply statistical significance.

This shows that the hypothesis is supported in 7 domains and not supported in 5. In the 7 domains where it is supported, we showed an example domain where there was a link between the features and
the macros that led to that performance improvement. In the other 5 domains, no correlation between the selected features and the performance was found.

### 3.7.2 Discussion

Here, we discuss the choices we made in regard to the system design and experiments, and the next step toward improving this approach.

#### Problem Instances Source

One important experimental choice concerned the nature of the problem instances to be used for the training and testing. There are two issues arising from that choice that we would like to discuss: the source of the training and testing instances, and the level of difficulty, for the planner, that we chose for the instances to meet.

To make better predictions and to gather statistical conclusions, we needed to use a large number of instances in our experiments. The IPC does not, generally, provide enough instances to serve for both the training and testing in the domains we tested; we found that we needed about 250 instances for the training – to make good predictions – and 150 instances for the testing – to get statistical significance. It should be noted, also, that the training and test instances should both be drawn from the same distribution to ensure the results are reliable, and so we cannot use only some problem instances from IPC. With that in mind, although we used IPC domains, it was necessary to get the training and testing problem instances from independent problem generators.

In our experiments, we did not exclusively choose hard or easy problem instances, but we did deliberately choose both hard and easy. By easy (hard) instances we mean that the planner requires a short (long) time to solve them on the original domain. Note that the choice between easy and hard instances is often independent from the choice of easy and hard domains: we generate easy and hard instances for all domains. Only in logistics, where the FF planner does not time out even on very large problem instance, we had to choose relatively easy instances.

We believe it is necessary to test the performance of the predictors on both easy and hard instances. Ideally, for an easy instance, the trained predictor should detect that it is easy for ORIG, and therefore choose ORIG. At the same time, the predictor should not choose r-domains that perform worse than ORIG on that easy instance. On hard instances, the predictor should detect that an instance is hard for ORIG, and if some other r-domain solves them more quickly, the predictor should choose that r-domain. Therefore, choosing easy and hard problem instances is important to testing the predictors.

#### Timeout Handling

In addition to the choice of instances, it is important to discuss how we handle the planner runtime. In this work, we removed instances that timed-out on all macro sets. It is known that removing the instances that timed out on any macro set will change the real mean and variance of the sample, since we are ignoring some data. Keeping them, on the other hand – and registering the cut-off time as their runtime in the cases where there is a time out – also changes the real mean and variance, because in that case we are using incorrect data for the real runtime. Removing some instances, such as those which are not solved by any macro subset, is probably less harmful to the mean and less biased than removing or keeping all timed-out instances; we will discuss this next.
There are four reasonable alternatives to dealing with these timeouts. In the following we discuss each and explain the alternative we chose.

1. **Removing only fully-censored instances:** It is a common practice to remove data points that are censored (timed-out) in all tested data (fully-censored) [87]. This practice has been done in scientific fields in which data is scarce and omitting partially-censored data points is not feasible.

   In our work, we needed to consider as much data as possible since the training requires many instances, and because instances that do not time out on any r-domain are rare. We found that making statistical conclusions also requires the inclusion of as many useful instances as possible. Therefore, we think it is reasonable to include partially-censored data points for the training and testing. Fully-censored data points contain no additional information in themselves, and so it may not be harmful to ignore them.

2. **Repeating the experiment with larger cutoff time:** Probably the best method is to re-run the instances with a larger cut-off times, whenever a timeout occurs, until there is no instance that times out. But there are two problems:

   (a) Some instances take a very long time, and that makes the method impractical.

   (b) Even if the instances that take a long time are solved within the new cutoff time, bias may appear. Registering a large runtime for these few instances will increase the residual error significantly; this is based on the fact that because of their large value, only these few instances really affect the average. Therefore, we should either use a statistic other than the average to evaluate the r-domains, or not increase the cutoff time so much that bias may occur.

   We think our choice of $c$ is appropriate. We chose a 60 minutes cutoff time which is larger than the 30 minutes cutoff time used in the IPC.

3. **Using the cutoff time for all timeouts:** One approach to deal with censoring is to consider a maximally conservative approach, such as that described by Etzioni and Etzioni [31]. This approach considers all timed-out instances, registering the cutoff time as the runtime. Given that the fully-censored instances may have real runtimes that work against the conclusion that is made based on the uncensored and partially censored instances, it is more conservative to register their runtime as the cutoff time. Registering the cutoff time for the fully-censored instances will make the average difference smaller, and will make conclusions of statistical significance more difficult to reach in general.

   However, this approach is insensitive to the fact that the fully-censored data may in fact support the conclusion made based on the partially censored and uncensored data. So there is a chance of declaring the difference between r-domains insignificant, when it is in fact significant. This defeats the purpose of statistical tests, since it limits our ability to detect real cases by relying on an inaccurate assumption (that all r-domains are equal on the very hard instances – where timeout is declared).

   In general, this approach suffers from the following problems:

   (a) This method assumes that the insensitivity resulting from the maximally conservative approach can be overcome by adding more instances. The method was developed to test learning
systems in planning, and it assumes that generating and running instances is easy compared to finding real-life data which is often difficult to gather. However, we think this assumption is inaccurate, since adding more data in our approach requires running the instances on a large number of r-domains, which is costly. Notice that it is rare to find instances that do not time out on any r-domain, especially in hard domains. So, it is not guaranteed that adding more instances will reduce the inaccuracy caused by the censoring. Not adding more instances makes the approach biased towards not making conclusions, which is also problematic, especially if a clear difference can be noticed without the fully-censored data.

(b) Ties in real runtime are very unlikely, and assuming that the timed-out instances have the same runtime is unrealistic. It is even more unlikely when the runtimes are larger: in our experiments, the easiest instances sometimes have the same small runtime of 0.01 seconds, but we found almost no ties in harder instances.

4. **Predicting the runtime of censored data:** Probably a more reasonable approach than (3) is to consider employing a predictive method in the censored data points as described by Schmee and Hahn [78]. The idea is to use a linear function of the instances’ features to predict the runtime of the timed-out instances. The function is iteratively modified to reduce error in the predictions. Therefore, this approach will not register the cutoff time as the runtime to the timed-out instance, but will register a number larger than the cutoff time and based on a linear function for the r-domain. Thus, this approach will consider the instances on which all r-domains time out.

While this approach might be useful in the training, it is perhaps not useful in the testing. We do not know the real distribution of the r-domains runtimes on test instances. In the worst case, we may incorrectly decide that one r-domain is better than another in some problem instance based on wrong predictions. Such erroneous decision would have major implications on our evaluation. This is because we are predicting unbounded large numbers and comparing them, without knowledge of the exact r-domain runtime distribution across the problem instances. We could not find a simple technique that can correct that error without making assumptions about the data distribution. While making strong assumptions (such as that the data always follows an exponential distribution) is acceptable with regard to the training instances, it is not suitable for the testing.

Therefore, we think that our choice (1) to remove instances on which all r-domains time out is the most reasonable.

**The Exhaustive Evaluation**

Although the exhaustive approach with which we evaluated the macro subsets is accurate enough to train the prediction models to choose the best instance-specific macro sets, it is also costly. In order to get good instance-specific predictions, we needed to consider as many runs as possible, but evaluating all macro subsets becomes intractable as the number of initial macros $n$ increases. A scalable approach would be to approximate the performance of the best macro set on problem instances by performing a local search in the space of macro subsets.

In the next chapter, we focus on filtering the r-domains rather than the context in which the r-domains are used. We present a novel method for remodeling the planning domain using macros, one that casts the macros as parameters of the domain and uses a generic parameter optimization tool to
find the best macro subset. This approach makes remodeling more general and flexible since it views the remodeling components as domain parameters that can be tuned. In addition, it can also speed up the instance-specific macro subsets evaluation step, since it uses parameter optimization tools that are fast in choosing useful parameter configurations.

3.8 Summary

In this chapter, we introduced an instance-specific domain remodeling approach that depends on learning macro sets. We presented the approach and explained it in detail, and we presented a feature selection algorithm that we use for the learning. We claimed that this approach, which uses a predictor to estimate the macro subsets for each problem instance, can improve the planning speed significantly as compared to traditional approaches, which use a fixed macro subset to remodel the domain for all instances. We excluded from our claim cases in which the planning domain is ‘easy’ and the FF planner is used. The results show that our approach can improve the planning speed significantly as compared to the best-on-average fixed macro subset approach in a number of ‘hard’ domains. We showed that our feature selection method can find useful features, and we showed that our predictors can learn the relation between these features and the macro subsets. We, then, analyzed the macro performance in one ‘easy’ domain. We explained that the significant amount of change in its topological structure, which resulted from adding particular macros, made our approach unlikely to be effective in such domain. Finally, we discussed some issues in the approach and a number of methods to improve it.
Chapter 4

Macro Subset Learning As Parameter Configuration

4.1 Introduction

We now switch our focus from the context in which macro-based remodeling is applied to the process of remodeling – using macro subsets – itself. In this chapter, we will recast the process of remodeling using macro subsets as a general parameter optimization. In this recasting we represent the macro operators as parameters of the domain, and we use a parameter configuration tool to search for the best macro subsets in both fixed and instance-specific macro contexts. Each macro is represented as a binary variable that indicates whether the macro is added to the domain or not. We then use a general-purpose parameter configuration tool – ParamILS [50] – to perform a fast local search in the space of the binary-variable vectors, which represent the macro subsets. In the fixed macro learning scheme, we are searching for a vector (set of macros) that maximizes the planner’s average performance on all instances. In the instance-specific macro learning context, the prediction model is trained to predict which macro subset the parameter tuning tool will suggest for the instance, rather than determining the optimal subset for that instance. The learning algorithm we use to build this predictor is based on the one used in Chapter 3.

One problem that we faced in the instance-specific macro learning approach presented in Chapter 3 is that, as the size of the initial macro set increases, the time required to evaluate the macro subsets – and, consequently, to train the predictors – increases exponentially. A reasonable approach to tackle this problem is to use a local search approach in the space of the macro subsets instead of trying all the subsets on the problem instances. Parameter tuning tools – such as F-race [6] and ParamILS [50] – perform fast local searches in the space of parameter configurations. Parameter tuning tools have been shown to contribute to significant improvements in the performance of the target algorithms, without needing much time to do so [6, 50]. See Section 2.5 for more details.

By reducing the macro subset learning problem to a general parameter optimization problem, we have achieved our goal of making the remodeling process more flexible and applicable. This reduction makes remodeling more flexible because it treats the process as a tuning step; in that, this reduced process is no different from the planner’s tuning step. Our reduction approach also makes remodeling more flexible because we can now recast other domain components as parameters. This approach makes
remodeling more applicable in planning, since it uses a general tuning tool to achieve the remodeling, and therefore implementing a new remodeling approach only requires finding the appropriate mapping between the parameters accepted by the tuning tool and the components of the domain that we want to remodel. This approach can make remodeling more flexible and extendible in that it makes remodeling a part of a tuning scheme, which can involve optimizing planner parameters and domain parameters.

In the following sections of this chapter: we describe the motivations of our approach; we describe the approach in detail; we described our experiments; and we analyze and discuss the results, as well as the work in general.

4.2 Motivations

There are three primary reasons to recast planning domain remodeling with macros as a parameter configuration problem:

1. **Domain Remodeling As Planner Preprocessing** – Remodeling should not be seen as a different form of preprocessing, distinct from a typical planner’s preprocessing that attempts to improve the planner’s performance. A general preprocessing method intended to improve a solver’s performance involves the use of a parameter optimization tool that tunes the solver’s parameters to maximize performance on training problem instances. Using such a tool to choose the remodeling of a domain is no different from using it to choose a search strategy or a search heuristic. For example, in a given problem instance, there is no practical difference between choosing a set of macro operators to remodel the domain, and choosing a search heuristic: the former improves planning by adding useful shortcuts to the search space, while the latter improves planning by ranking the search nodes for exploration. In the end, both serve to optimize the planner’s behavior. Therefore, they should be considered as belonging to the same preprocessing phase, and hence, they could be achieved using the same kind of tools.

2. **Benefits of Reduction** – The existence of efficient, general purpose parameter configuration tools makes it practical to recast the macro subset learning as a parameter configuration problem. Automated parameter tuning has advanced significantly over the past few years; it is now possible to use a tuning tool to find parameters that improve performance substantially more than can be achieved by basic search approaches, experts or even by the original designer of the solver [50] (Refer to Section 2.5 for more details on parameter tuning.) Thus, using such general tools can improve domain remodeling, since they utilize advanced methods to improve the speed and the quality of the remodeling. The reduction of a hard problem to another form, making it solvable by a faster tool, is a common practice in the literature. For example, Blackbox [54] and SatPlan [53] convert the STRIPS planning problem to a propositional satisfiability (SAT) problem and pass it to a SAT solver as described in Section 2.4.2.

3. **Faster Evaluation in Instance-Specific Remodeling** – The exhaustive evaluation featured in instance-specific macro learning can be replaced by an incomplete evaluation that is both accurate and quick. As we saw in Chapter 3, one main obstacle we faced in the instance-specific macro subset learning was the exponential number of subsets that needed to be evaluated, which limited the number of macros we were able to use. This problem can be solved by finding an approximation of the best macro subset in a reasonable time. A reasonable method to make such an approximation
is to use a local search in the space of macro subsets, a task which parameter tuning tools can efficiently achieve.

4.3 ParamILS

We use ParamILS [51], an algorithm tuning tool, to learn the macro subsets. ParamILS, as discussed in Section 2.5.1, is a general-purpose algorithm tuner designed to work with large spaces of both numerical and categorical parameters using the Iterative Local Search (ILS) algorithm [50, 60] – a search technique that proved to outperform other local search techniques for parameter tuning. We discussed the different versions of ParamILS in Section 2.5.1. The version we use in this work is the FocusedILS version, which is equipped with adaptive capping.

There are a number of reasons that we chose ParamILS as our parameter tuning tool. The reasons are as follows:

1. **Applicability in Many Contexts** – ParamILS is a general-purpose algorithm tuner designed to work on arbitrary algorithms and to deal with very large parameter spaces composed of numerical and categorical parameters. Some other optimization tools are used only to configure certain kinds of algorithms; for example, F-race is used to configure local search algorithms. In addition, ParamILS allows for the use of categorical parameters, which we use to represent the macros in the domains.

2. **Fast Evaluation** – ParamILS uses a number of efficient methods to decrease the time needed to evaluate the parameter configurations:

   (a) It uses Iterative Local Search (ILS), which relies on greedy search and random moves to determine the best parameter configuration.

   (b) It uses adaptive capping, a technique used to interrupt the evaluation of parameter configurations whose runtime exceeds that of the best configurations.

   (c) The FocusedILS version of ParamILS does not use all training instances to compare the parameter values; instead, it uses only a small number of them.

   Refer to Section 2.5.1 for more details on ILS, adaptive capping and FocusedILS.

3. **Convergence** – Not using all the training instances to compare the parameter configurations can be a disadvantage, since we do not know whether the resulting parameter configuration is as good as the one evaluated on all instances. However, Hutter et al. show that FocusedILS is guaranteed to converge to the parameter configuration of BasicILS, the version that uses all instances to compare the parameters [51]. Thus, FocusedILS also maintains the same configuration quality that results from using all training instances.

4.4 Approach

In this section, we explain how we use ParamILS in our work in order to learn macro subsets. As in the exhaustive search, ParamILS can be used to choose macros to remodel the planning domain in both the fixed and the instance-specific schemes. In the fixed macros scheme, the domain is tuned by running
the planner on a number of instances and on a number of macro-augmented r-domains; this is meant to determine the best macro subset, which should be used permanently for solving future instances. In the instance-specific scheme, the domain is tuned for each cluster of instances that share the same feature measurements, while the resulting r-domains are used, with their corresponding cluster features, for the training.

In the following we explain in detail how each scheme is achieved. First we show how our system interacts with ParamILS, then we present some necessary definitions, and finally we explain the fixed macro scheme and the instance-specific macro scheme, in detail.

### 4.4.1 Interface with ParamILS

For a given algorithm with tunable parameters, ParamILS works as a black box to find the parameter configuration that optimizes the algorithm’s performance on a set of problem instances. Therefore, the inputs to ParamILS are as follows:

1. The target algorithm to be optimized.
2. The parameters of the algorithm and their possible values.
3. A set of training problem instances, to be used for the tuning.

The outputs of ParamILS include the best parameter configuration found.

The target algorithm is not called directly by ParamILS; rather, it is passed to ParamILS in a wrapper code. The wrapper code calls the target algorithm routine with the parameter values specified by ParamILS, then it registers the returned results and measures the runtime. The parameters are the key to improving the average performance of the target algorithm. With the right parameter values, the
target algorithm’s average performance is optimized. The training problem instances should be solvable by the planner, and should represent the distribution of instances in which we are interested.

Figure 4.1 shows how we use ParamILS in our work. We represent the initial set of macros as a vector of binary parameters, with one parameter for each macro. A parameter is assigned the value 1 if we choose to add the corresponding macro to the domain for solving the problem instance; otherwise it is assigned a 0 value. As shown in Figure 4.1, ParamILS calls the wrapper code by passing a training instance and a parameter assignment to it. The wrapper code, in this case, performs the following tasks:

1. It builds a new macro-augmented domain, based on the parameters passed to it, using a domain builder.
2. It runs the planner using the new domain and the problem instance that was passed by ParamILS.
3. It returns the result with the runtime. The results can be ‘SAT’ if a solution plan was found, ‘UNSAT’ if no solution was found, or ‘TIMEOUT’ if the planner took more time than specified as the cutoff time.

In its local search algorithm, ParamILS calls the wrapper a number of times using different macro sets and training instances. It eventually converges to one parameter configuration (or macro subset) as described in [50]. Finally, we interpret the returned binary parameter values back into a macro subset: a return of 1 means we include the corresponding macro, and 0 means we exclude it.

ParamILS can be used to find macro subsets in two contexts: the fixed-macros scheme, and an instance-specific macros scheme. In the fixed-macros scheme, we attempt to establish one set of macros that is suitable for all domain instances. This scheme is straightforward and is achieved by a single call to ParamILS using the initial macro set and the training instances as described above. In the instance-specific scheme, we look for a set of macros for every cluster of problem instances. We use the macros found for each cluster to train an instance-specific predictor, as described in the previous chapter, Section 3.3. In the following subsections we describe each scheme in more details.

4.4.2 Definition

Based on the definitions presented in Section 2.2.1, we define the following:

A cluster \(I^{X=Y}\) of instances is a subset of \(I\) where all instances share the same measurements \(Y\) of \(X\); i.e., \(\forall i \in I : i \in I^{X=Y}\) if and only if \(X(i) = Y\). Therefore, a feature vector \(X\) with domain \(\text{dom}_X = \{Y_1, Y_2, ..., Y_n\}\) can cluster a set of instances \(I\) into a set of disjoint clusters \(C\), where \(C = \{I^{X=Y}\} | Y \in \text{dom}_X\).

4.4.3 System Details

Given a number of macros \(S\) in a domain \(D\), a planner \(P\), and a set of training instances \(I_{\text{train}}\), we can learn macro subsets using two schemes: a fixed scheme and an instance-specific scheme.

Fixed Macros Scheme

The fixed scheme is straightforward: to search for a macro subset, we use all instances in \(I_{\text{train}}\) as the training instances for ParamILS. We use the planner \(P\) and the domain \(D\) as arguments to the wrapper
code, as described in Section 4.4.1, and we interpret all the macros in $S$ as a binary parameter vector of size $|S|$. For example, if $|S| = 5$, the parameter configuration $(0, 1, 0, 1, 1)$ means that we chose to add only the second, fourth and fifth macros to the domain. To begin the search, we initialize the parameter vector to all-ones for ParamILS. The returned parameter values are then interpreted as a macro subset that can be used to solve future instances.

**Instance-Specific Macro Scheme**

In this scheme we take the following steps:

1. We select and extract the features $X$ from the training instances $I_{train}$. The features are identical to the features described in the previous chapter, Section 3.3.4: taxonomic syntax, simple statistics, and manual features.

2. We use the extracted features $X$ to cluster $I_{train}$. By clustering $I_{train}$ we mean the gathering of all instances $i \in I_{train}$ into groups, with instances included in each group sharing the same feature measurement $X(i) = Y$. So, for each observed $Y$, there is one group. The resulting clusters are $C_{train} = \{I_{train}^{X=Y} | Y \in \text{dom} X\}$. Some instances may have feature measurements equal to those of other instances, therefore, $|C_{train}| \leq |I_{train}|$.

3. For a cluster of training instances $c_j \in C_{train}$ with a feature measurement $Y_j$, we use ParamILS to find the parameter configuration $pc_j$ with the shortest runtime of the instances $c_j$ on the planner $P$. The parameter configuration represents the macro subset used to solve the instances in $c_j$.

4. After ParamILS finds the appropriate set of macros for the cluster $c_j$, we register the features' measurement $Y_j$ with the chosen macro numbers in the file that we use to training the predictor.

5. We repeat steps (3) and (4) for all observed feature measurements $Y_j \in \text{dom} X$ in the training instances $I_{train}$.

6. We have established a training file that relates the problem instance’s features to the macro subset that ParamILS suggests is the best fit for it. We feed the training file into a supervised learning tool, which produces a direct prediction model. The new direct predictor will then try to estimate which macro subset ParamILS will suggest based on the given instance’s features. As in the case of exhaustive evaluation (see Section 3.3.4), we use the SMO classification algorithm, equipped with quadratic kernel for the learning.

This kind of learning is similar to the direct model approach discussed in the previous chapter, which is used to predict the best macro subset based on the features of the problem instances. We do not use the time model in this context because we do not have the runtime of every training instance on every macro subset. While ParamILS maintains a sizable log file in order to record all the execution details, there is information that it misses, mainly as a result of the techniques it uses to increase the speed of the evaluation. The adaptive capping technique that was discussed in Section 2.5.1, for example, prunes the evaluation of macro subsets that take more time than the best subset the moment they are recognized. FocusedILS, the version that we use, also uses an evaluation technique that minimizes the number of training instances needed to select one macro subset over another. Therefore, the runtime data needed to build the time predictor for each macro subset is not complete.
4.5 Claims and Research Questions

We recast the macro subset learning problem as a parameter optimization problem, in which we consider each macro operator as a binary parameter, and we consider the macro subset used for domain reformulation as a parameter vector. By accepting the domain modeling as part of the planning process, and thus using a general algorithm optimization tools to improve its speed, we have extended the concept of planning to formally include domain remodeling. To achieve the remodeling we use the ParamILS tool. We claim that using this extension to the planning process and using ParamILS for remodeling planning domains can improve the planning speed at least as much as can be done using the traditional macro acquisition tools, and sometimes we can improve upon that speed.

In instance-specific macro subset learning discussed in Chapter 3, one problem was that we needed to evaluate an exponential number of macro subsets. ParamILS uses local search and other useful techniques to increase the speed of the search for useful parameter configurations. Thus, we also claim that using ParamILS in the instance-specific context as described in Section 4.4.3 can improve the planning performance over that achieved by the best fixed macro subset while reducing preprocessing time significantly.

Therefore, we present the following hypotheses for testing, and we present a justification for each one:

**Hypothesis 2.1** Using our ParamILS-based approach in the fixed scheme is at least as efficient as using other state-of-the-art tools, such as Wizard, which search for the best subset of macros given a set of training examples, or Macro-FF, which tries to directly find the best macro subset.

**Justification** ParamILS is a flexible tool that uses efficient search and statistical methods to find useful parameters in a large parameter configurations space. The space of macro subsets, which are represented as parameter configurations, is an example of a large search space. As discussed in Section 4.3, ParamILS can find good parameter configurations in little time by utilizing strong evaluation techniques such as: iterative local search, adaptive capping, and FocusedILS. This remodeling approach can also be extended to tune other parameters, such as the planner’s parameters, in addition to choosing macro subsets, which gives it an advantage over these tools.

**Hypothesis 2.2** If the instance-specific macro predictor that is based on the exhaustive evaluation of macro subsets improves the performance significantly, as compared to fixed-macros schemes, then using the predictor based on ParamILS’s approximate evaluation, will also improve the performance significantly, as compared to fixed-macros schemes, while reducing the pre-processing time significantly.

**Justification** The instance-specific macro learning approach was shown to be able to improve the planning speed significantly as compared to the best-on-average fix-macro remodeling approach. However, the increase of the preprocessing time in the instance-specific approach is exponential with respect to the number of macros used. This is because the training relies on evaluating all macro subsets on all training instances in order to choose the best macro subset that will be provided to the learner for each instance. Therefore, using a time-limited, incomplete local search in the space of macro subsets can be more practical than using an exhaustive search. Macro acquisition tools [68, 67] have used incomplete search, but not in an instance-specific context.
4.6 Experimental Design

In the following we detail the models compared in the experiments and how we test the hypotheses. We conduct two different kinds of experiments, one for each hypothesis as explained in the next sections. In the following we explain the settings of the experiments, and next we explain how we conducted each one.

4.6.1 Settings

Hardware and Software Details

We used the same hardware and software as in Section 3.5.1. To build the new system, we used version 2.3 of ParamILS, downloaded directly from the authors’ website.\(^2\) The wrapper code described in Section 4.4.1 was written in the Ruby scripting language.

Planners, Domains, Instances, and Macro Sources

We used the same set of planners, benchmark domains, problem instances, and macro sources that we used in the previous chapter, Section 3.5.1.

Learning Model and Features

Just like the previous chapter, we use the machine learning tool WEKA [86] for the learning with the SMO algorithm with quadratic kernel for the ParamILS-based direct predictor described in Section 4.4.3. We also used the three feature sources mentioned in the previous chapter: the taxonomic syntax, the simple statistics of the initial state and goal, and the domain-specific manually selected features for the freecell domain only. See Section 3.5.1 for more details.

Timeout Handling

For ParamILS optimization, the cut-off time to find a macro set for a given cluster of instances was varied depending on the number of initial macros and the difficulty of solving the instances with the original domain on the used planner. The cut-off times are shown in Table 4.1. For most experiments the cut-off time was one or two hours. For easy domains the time was decreased (e.g., in the logistics-FF-WIZARD the cut-off was 0.25 hours) while for harder domains it was increased (e.g., in the mprime-VHPOP-MANUAL the cut-off was two hours).

For the test, we run all test instances on all r-domains and we set the cut-off CPU time to one hour, just as in Section 3.5.1. We record the cut-off time as the runtime of timed-out instances. As in the previous chapter, the test instances for which all macro sets timed out are not considered in the evaluation. Runs that were interrupted by errors are treated as timed-out. The time for feature extraction, running the predictor, and adding the selected macros to the domain is negligible and is not included in the total runtime.

Table 4.1: The ParamILS cutoff times (in hours) used to find a macro set per cluster in the instance-specific experiment.

<table>
<thead>
<tr>
<th>Domain-Planner-Macro-source</th>
<th>cutoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics-FF-WIZARD</td>
<td>0.25</td>
</tr>
<tr>
<td>blocksworld-4ops-FF-WIZARD</td>
<td>2.00</td>
</tr>
<tr>
<td>pipesworld-nt-FF-WIZARD</td>
<td>2.00</td>
</tr>
<tr>
<td>mystery-5-FF-WIZARD</td>
<td>1.00</td>
</tr>
<tr>
<td>freecell-A-FF-WIZARD</td>
<td>2.00</td>
</tr>
<tr>
<td>blocksworld-4ops-FF-MACROFF</td>
<td>2.00</td>
</tr>
<tr>
<td>pipesworld-nt-FF-MACROFF</td>
<td>0.50</td>
</tr>
<tr>
<td>freecell-A-FF-MACROFF</td>
<td>1.00</td>
</tr>
<tr>
<td>logistics-FF-MANUAL</td>
<td>1.00</td>
</tr>
<tr>
<td>blocksworld-4ops-FF-MANUAL</td>
<td>2.00</td>
</tr>
<tr>
<td>pipesworld-nt-FF-MANUAL</td>
<td>2.00</td>
</tr>
<tr>
<td>mystery-5-FF-MANUAL</td>
<td>1.00</td>
</tr>
<tr>
<td>mprime-5-FF-MANUAL</td>
<td>1.00</td>
</tr>
<tr>
<td>freecell-A-FF-MANUAL</td>
<td>2.00</td>
</tr>
<tr>
<td>blocksworld-4ops-VHPOP-MACROFF</td>
<td>2.00</td>
</tr>
<tr>
<td>pipesworld-nt-VHPOP-MACROFF</td>
<td>0.50</td>
</tr>
<tr>
<td>freecell-B-VHPOP-MACROFF</td>
<td>0.50</td>
</tr>
<tr>
<td>logistics-VHPOP-MANUAL</td>
<td>1.00</td>
</tr>
<tr>
<td>blocksworld-4ops-VHPOP-MANUAL</td>
<td>2.00</td>
</tr>
<tr>
<td>pipesworld-nt-VHPOP-MANUAL</td>
<td>2.00</td>
</tr>
<tr>
<td>mystery-5-VHPOP-MANUAL</td>
<td>1.00</td>
</tr>
<tr>
<td>mprime-5-VHPOP-MANUAL</td>
<td>2.00</td>
</tr>
<tr>
<td>freecell-B-VHPOP-MANUAL</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Utility

The utility function is the same as in the previous chapter. The only difference is that $U$ is not defined on all instances and r-domains, since we do not evaluate the training instances on all r-domains. For the test instances, it is as defined in Chapter 3.

Clustering

Using a set of selected features $X$, the training instances $I_{train}$ is clustered by $X$, and the resulting clusters are $C_{train}$. Therefore:

$$C_{train} = \{ I_{train}^Y : Y \in dom_X \}.$$

And we define $C_{test}$ similarly:

$$C_{test} = \{ I_{test}^Y : Y \in dom_X \}.$$

Models to Compare

Given a planner $P$, a planning domain $D$, a set of macros $S$, a set of training and testing instances $I_{train}$ and $I_{test}$ generated from $D$, and a features vector $X$ that clusters $I_{train}$ to $C_{train}$ and $I_{test}$ to $C_{test}$, we define the following r-domains and reformulations:

**ORIG, BOA, PERF, and DIR** are exactly the same as defined in the previous chapter. BOA, PERF, and DIR are obtained by exhaustive evaluation of all macro subsets on training instances.
BOA-P is the best r-domain on average based on ParamILS evaluation on the training instances. It is the result of applying ParamILS in the fixed macro scheme described in Section 4.4.3. So, formally:

$$\forall i \in I_{test}: \text{BOA-P}(i) = D_{\text{ParamILS}(P,D,S,I_{train}),S,\emptyset}$$

where the \text{ParamILS} function returns the macro subset that ParamILS suggests on planner $P$, training instances $I_{train}$ and macros $S$ using the wrapper code. It should be noticed here that we passed the whole training set $I_{train}$ to ParamILS to find the best macro subset on average.

DIR-P is the direct predictor based on ParamILS. The training of this predictor is based on the instance-specific scheme described in Section 4.4.3. It is trained based on the features associated with each cluster of the training instances and the macro subset resulting from running ParamILS on the cluster’s instances.

It has the similar definition to DIR (see Section 3.5.1) but with different training as follows:

$$\forall i \in I_{test}: \text{DIR-P}(i) = D_{\text{Classify}(i,T_{X,\text{ParamILS}}),S,\emptyset}$$

and $T_{X,\text{ParamILS}} = \{(X(i), \text{ParamILS}(P,D,S,c_j)) | c_j \in C_{train}, i \in c_j\}$.

PERF-P is the perfect predictor based on the ParamILS evaluation of the testing instances. We cluster the test instance just as we did with the training instances, and we call ParamILS for each cluster.

$$\forall c_j \in C_{test}, i \in c_j: \text{PERF-P}(i) = D_{\text{ParamILS}(P,D,S,c_j),S,\emptyset}.$$

WIZ and MFF are the r-domains found by Wizard’s bunching phase and Macro-FF CA-ED, respectively. See Section 3.5.1 for more details on these r-domains.

4.6.2 Tests

We test the two hypotheses presented in Section 4.5 with two experiments. In the first experiment, we evaluate the performance of the fixed macro set found by ParamILS. In the second experiment, we evaluate the performance of the instance-specific macro predictor that is based on ParamILS evaluation.

Experiment 2.1: Fixed Macro set Learning

In this experiment, we test Hypothesis 2.1. We compare the performance of ParamILS in finding macros to two state-of-the-art macro tools: Wizard and Macro-FF. We compare the average runtime of the macro set found by Wizard’s bunching phase (WIZ) to the best macro set found by ParamILS (BOA-P). The initial set of macros $S$ is provided by Wizard’s chunking phase as described in the previous chapter. We run Wizard’s bunching phase to evaluate subsets of $S$ and find the best fixed subset. We also compare both WIZ and BOA-P to the best-on-average fixed macro set performance (BOA).

We also compare to Macro-FF, a tool that directly tries to find the best set of macros using abstract components. We start with Macro-FF’s macros, and we compare the macro subset produced by ParamILS (BOA-P) to these macros (MFF).

Experiment 2.2: Instance-Specific Macro set Learning

In this experiment, we test Hypothesis 2.2. We compare the performance of the direct predictor that results from choosing the macro sets using ParamILS (DIR-P) to the direct predictor based on the
Chapter 4. Macro Subset Learning As Parameter Configuration

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA</th>
<th>WIZ</th>
<th>BOA-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics</td>
<td>1.20</td>
<td>0.77</td>
<td>1.21</td>
<td>+0.77</td>
</tr>
<tr>
<td>blocksworld-4ops</td>
<td>1278.00</td>
<td>0.56</td>
<td>100.82</td>
<td>+0.56</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>163.51</td>
<td>167.47</td>
<td>152.79</td>
<td>163.51</td>
</tr>
<tr>
<td>mystery-5</td>
<td>155.60</td>
<td>155.60</td>
<td>264.29</td>
<td>264.29</td>
</tr>
<tr>
<td>freecell-A</td>
<td>500.91</td>
<td>503.51</td>
<td>536.37</td>
<td>441.65</td>
</tr>
</tbody>
</table>

Table 4.2: Average runtimes (in seconds) of the r-domains constructed from the chunking phase of Wizard using the FF planners. The plus (+) means that WIZ (BOA-P) was significantly faster than BOA-P (WIZ) with $p \leq 0.05$.

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA</th>
<th>MFF</th>
<th>BOA-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocksworld-4ops</td>
<td>1267.80</td>
<td>3.04</td>
<td>3.04</td>
<td>3.04</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>413.32</td>
<td>413.32</td>
<td>424.90</td>
<td>424.90</td>
</tr>
<tr>
<td>freecell-A</td>
<td>371.27</td>
<td>174.79</td>
<td>174.79</td>
<td>174.79</td>
</tr>
<tr>
<td>blocksworld-4ops</td>
<td>1140.01</td>
<td>200.38</td>
<td>200.38</td>
<td>262.29</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>294.48</td>
<td>294.48</td>
<td>668.21</td>
<td>+294.48</td>
</tr>
<tr>
<td>freecell-B</td>
<td>326.07</td>
<td>326.07</td>
<td>558.36</td>
<td>+326.07</td>
</tr>
</tbody>
</table>

Table 4.3: Average runtimes (in seconds) of the domain models constructed by Macro-FF using the FF planner. The plus (+) compares MFF and BOA-P. See Table 4.2 for details.

exhaustive evaluation (DIR). On domains where DIR is significantly faster than BOA, we would like to know whether DIR-P is also significantly faster than BOA.

Also, in all domains, we expect the preprocessing time required by ParamILS in the instance-specific context to be significantly less than the preprocessing time required by the exhaustive evaluation.

We also record the performance of the best instance-specific predictor (PERF-P) and the best fixed r-model (BOA-P) of ParamILS evaluation for later analysis.

4.7 Experimental Results

In the following two subsections, we show the results of the two experiments described above. In all experiments, we used a paired $t$-test [19] with $p \leq 0.05$ to test statistical significance.

4.7.1 Fixed Macros Test

In the fixed macro scheme experiment, we found that the ParamILS method was effective in finding good macro sets in the tested domains as compared to the best-on-average. Table 4.2 and Table 4.3 show the relevant results. In 10 out of 11 domains, BOA-P was not significantly different from BOA. BOA-P was faster than WIZ in 2 out of 5 domains, and faster than MFF in 2 out of 6 domains. BOA-P was also better than ORIG in 5 out of 11 domains.

Table 4.2 shows that there was no significant difference in performance between BOA-P and BOA except in the mystery domain, in which BOA was faster. In two domains (logistics and blocksworld), BOA-P achieved significantly better performance than WIZ and ORIG. In the remaining three domains, there was no significant difference between BOA-P and either of WIZ or ORIG. In freecell-A and pipesworld-nt, all r-domains were almost equal. This maybe due to the quality of the macros produced
by Wizard’s chunking phase in these domains, especially given that, according to Wizard, those macros had low utility value.

Table 4.3 compares BOA-P to Macro-FF’s macro set, MFF, using the FF and VHPOP planners. In all domains, BOA-P was not different from BOA. In the pipesworld-nt-VHPOP and freecell-VHPOP domains, BOA-P was significantly faster than MFF by up to 2.27 times. In the remaining four domains, BOA-P was not statistically different from MFF. BOA-P was, also, significantly faster than ORIG in three domains: blocksworld-4ops-FF, freecell-A-FF, and blocksworld-4ops-VHPOP. It was equal to ORIG in the remaining domains.

In summary, BOA-P was significantly faster than the macro tools in 4 out of 11 domains, and was not statistically different in 7. In 6 of these 7 domains, there was a ceiling effect: BOA-P was statistically not different from BOA, the best-on-average r-domain obtained by exhaustive evaluation. In the domain where there was no ceiling effect, WIZARD-mystery, BOA-P and the macro tools were worse than BOA because BOA was in this case the original domain, and both BOA-P and WIZ choose a non-empty macro set. In this domain, the macros produced by Wizard’s chunking phase had very low quality.

### 4.7.2 Instance-Specific Macros Test

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>Exhustive</th>
<th>ParamILS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BOA</td>
<td>DIR</td>
<td>PERF</td>
</tr>
<tr>
<td><strong>FF-WIZARD</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logistics</td>
<td>1.20</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>blocksworld-4ops</td>
<td>1278.00</td>
<td>0.56</td>
<td>22.14</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>163.51</td>
<td>167.47</td>
<td>184.57</td>
</tr>
<tr>
<td>mystery-5</td>
<td>155.60</td>
<td>155.60</td>
<td>153.93</td>
</tr>
<tr>
<td>freecell-A</td>
<td>500.91</td>
<td>503.51</td>
<td>476.97</td>
</tr>
<tr>
<td><strong>FF-MACROFF</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocksworld-4ops</td>
<td>1267.8</td>
<td>3.04</td>
<td>93.2</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>413.32</td>
<td>413.32</td>
<td>425.60</td>
</tr>
<tr>
<td>freecell-A</td>
<td>371.27</td>
<td>174.79</td>
<td>243.17</td>
</tr>
<tr>
<td><strong>FF-MANUAL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logistics</td>
<td>3.80</td>
<td>3.80</td>
<td>3.85</td>
</tr>
<tr>
<td>blocksworld-4ops</td>
<td>1118.18</td>
<td>53.87</td>
<td>53.87</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>477.41</td>
<td>277.85</td>
<td>440.39</td>
</tr>
<tr>
<td>mystery-5</td>
<td>239.48</td>
<td>239.48</td>
<td>*146.00</td>
</tr>
<tr>
<td>mprime-5</td>
<td>521.53</td>
<td>8.81</td>
<td>*3.34</td>
</tr>
<tr>
<td>freecell-A</td>
<td>296.43</td>
<td>352.21</td>
<td>*194.53</td>
</tr>
<tr>
<td><strong>VHPOP-MACROFF</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocksworld-4ops</td>
<td>1140.01</td>
<td>200.38</td>
<td>262.28</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>294.48</td>
<td>294.48</td>
<td>260.47</td>
</tr>
<tr>
<td>freecell-B</td>
<td>326.07</td>
<td>326.07</td>
<td>*199.34</td>
</tr>
<tr>
<td><strong>VHPOP-MANUAL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logistics</td>
<td>316.06</td>
<td>316.06</td>
<td>282.26</td>
</tr>
<tr>
<td>blocksworld-4ops</td>
<td>1217.1</td>
<td>157.14</td>
<td>176.65</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>297.86</td>
<td>297.86</td>
<td>297.86</td>
</tr>
<tr>
<td>mystery-5</td>
<td>433.73</td>
<td>92.36</td>
<td>251.25</td>
</tr>
<tr>
<td>mprime-5</td>
<td>1048.81</td>
<td>292.49</td>
<td>597.20</td>
</tr>
<tr>
<td>freecell-B</td>
<td>1154.55</td>
<td>214.58</td>
<td>192.34</td>
</tr>
</tbody>
</table>

Table 4.4: Experiment 2: Average runtimes (in seconds) of the reformulations, each of which constructed from different macro sources using different planners. There are five experiments: one for each pair of planner and macro source (except VHPOP and Wizard.) The asterisk (*) means that the reformulation was significantly faster than BOA with \( p \leq 0.05 \). 'Pre' refers to the preprocessing time, as measured in hours.

In Table 4.4, we show the runtime of the r-domains and predictors in the five sub-experiments. As noted in hypothesis 2.2, we would like to show that in combinations of macro source / planner / domain in which DIR performs significantly better than BOA, DIR-P also outperforms BOA. In addition, we want to show that the required offline preprocessing time is significantly less when DIR-P is used than when DIR is used. In all of the four combinations in which DIR performed significantly better than
BOA, DIR-P was also significantly better than BOA. Support for hypothesis 2.2 is laid out in more detail below.

- In three sub-experiments, FF-WIZARD, FF-MACROFF and VHPOP-MANUAL, there were no domains in which DIR performed significantly better than BOA; thus, these domains are not relevant to our hypothesis. However, this result leads us to ask about the direct relation between DIR and DIR-P, which we will discuss in detail in the Analysis section.

- In two sub-experiments, FF-MANUAL and VHPOP-MACROFF, a total of four of the nine planning domains revealed significant differences between DIR and BOA, with DIR outperforming BOA in those four cases. On each of these domains, DIR-P was also significantly better than BOA, supporting our hypothesis.

- As expected, the offline learning time for DIR-P was found to be substantially less than that of DIR. DIR-P exhibited significantly less learning time in 22 of the 23 planner / domain combinations. The exception is logistics-FF-WIZARD, in which the very short planning times of the training instances led to an evaluation for the exhaustive approach that was 16 times faster than the evaluation for the ParamILS-based approach. Across all macro source/planner/domain combinations, the median speed-up for DIR-P is 5.11 times.

4.8 Analysis and Discussion

In this section, we analyze the results by discussing their relation to the hypotheses, and by taking a deeper look at the data. Then, we discuss the whole work by revisiting the system design and possible improvements to it.

4.8.1 Analysis

In the experiments, we tested the utility of recasting macro subset learning as parameter configuration. In the fixed macro subset learning scheme, we showed that ParamILS-based macro learning performs as well or better than state-of-the-art macro learning tools; our system improved upon their performance in four domains while showing no significant difference in seven. In 10 of these 11 domains, the macro set proposed by our system was as good as the best-on-average macro set, which represents the best expected performance yet found of any fixed-macro set approach. This result supports hypothesis 2.1 (see Section 4.5).

In the context of instance-specific macro learning we compared a predictor, one that maps problem instance features to macro sets generated by ParamILS, to a predictor created by solving problem instances on an exponential number of macro-augmented r-domains. In all four macro source / planner / domain combinations where the latter predictor outperformed the best-on-average fixed macro set, the ParamILS-based predictor outperformed the best-on-average set as well. In addition, the preprocessing time has been reduced through the use of the new approach. The median preprocessing speed-up was by 5.11 times. These results support hypothesis 2.2. The results of experiments 1 and 2 demonstrate that the macro subset learning quality is improved when the approach is applied, and that such recasting of remodeling as parameter optimization is useful in both contexts.
To further examine the instance-specific results, we looked at PERF-P and DIR-P. We note that PERF-P, the predictor built with ParamILS using the testing instances, performed as well as PERF on many domains (20 out of 23). If we compare DIR and DIR-P directly we find that the ParamILS-based predictor matched the performance of the exhaustive predictors in 19 of the 23 combinations. DIR is significantly better in four cases, all using VHPOP-MANUAL. In the following we discuss these observations in more detail.

**VHPOP’s Errors and DIR-P**

In the VHPOP experiment, we found that some instances that were solved without error in the exhaustive search cause errors in ParamILS runs, and vice versa. Errors occur randomly and frequently in these experiments and especially in VHPOP-MANUAL mystery and mprime. Such errors may have caused ParamILS to select macro subsets that are different from those selected through exhaustive evaluation, resulting in different levels of predictor performance. Not only was the DIR-P predictor performance not accurately trained, it was also not accurately tested, given that the test instances can contain errors. This may have caused the difference in performance between DIR and DIR-P. In the FF sub-experiments, meanwhile, DIR and DIR-P were not found to be significantly different; in fact, they were almost identical. In the VHPOP experiments, especially those in which we use large number of macros and hard domains, DIR-P became significantly worse than DIR.

**Comparing DIR-P and DIR**

We added a new layer of learning to our system by using ParamILS to find the best macros for the training instances. The original learning layer involved only the training of the predictor to choose macro subsets based on the problem instances’ features. The new layer learns to choose a macro subset for each cluster of instances. We cluster the instances based on their feature measurement, and we run ParamILS for each cluster to find the best macro subset on each cluster of instances. One advantage of the new layer is that it reduces the large pre-processing time caused by the need to evaluate all of the $2^n$ macro subsets on all problem instances for the purpose of training. The performance of the new predictor, DIR-P, which depends on both layers, was close to the original, DIR, which depends only on the original learning layer. DIR-P was also different from the perfect, PERF, in most cases.

In two domains, freecell-FF-WIZARD and logistics-VHPOP-MANUAL, we found that the DIR-P predictor significantly improves the performance over BOA while the DIR predictor does not. This result is somewhat unexpected; we used the same training instances and the same set of features to train both predictors. They were also tested on the same set of instances. Therefore, we needed to directly compare the performance of DIR and DIR-P to know whether the new layer of learning improves, harms or does not affect the prediction process.

In all tested domains, we found that the performance on DIR and DIR-P was not significantly different, except in the domains mprime, mystery, pipesworld, and freecell – all in the VHPOP-MANUAL experiment, in which DIR was faster. However, in these domains, the error affected the performance significantly as discussed in the previous subsection. Therefore, we conclude that the difference in performance between DIR and DIR-P is very small, even though we found that DIR-P outperformed DIR as compared to BOA in two domains, and even though DIR was directly faster than DIR-P in four domains from the VHPOP-MANUAL experiment (in which the planner caused errors).
More Practical Instance-Specific Learning

Instance-specific macro learning has become more practical under this new parameter tuning approach. The resulting performance was not significantly harmed by the additional layer of learning macros per cluster (in 19 out of 23, the DIR-P was better or equal to DIR; the other cases are in VHPOP-MANUAL, in which there were a lot of errors, as discussed above). Meanwhile, the preprocessing time was significantly reduced. Therefore, we can conclude that the use of ParamILS significantly improved instance-specific macro learning.

This result allows us to increase the number of initial macros and perhaps to improve the remodeling. Since we know that starting with a larger macro set does not necessitate a larger preprocessing time, we can add more useful macros that can enrich the instance-specific learning process. Adding more macros improves the chances of finding macro subsets that are suitable for instance-specific learning.

4.8.2 Discussion

Here, we discuss the broader implications of our work. Specifically, we discuss the significance of the contribution, and the work’s future applications.

Individual Macro Learning As Parameter Optimization

In our experiments, we have found that if the individual macros do not improve the performance significantly over the original domain, we may not be able to ensure an adequate performance in the instance-specific context. In a number of domains, e.g. FF-WIZARD-pipesworld-nt and FF-WIZARD-mystery-5, we found that the quality of all the r-domains constructed from the initial set of macros was not significantly better than the original domain’s. Therefore, we propose to use a parameter optimization tool such as ParamILS to generate individual macros as opposed to filtering the macro sets. However, we will not further evaluate or build on this proposal in this work. This approach can generate macros of fixed length that have a minimal sufficiency in order to be useful.

We begin with the domain $D = (\Psi, O)$, planner $P$, and a set of instances $I_{train}$. We limit our focus to macros $m = (o_1, o_2, ..., o_n)$ of a constant size $n$ (e.g., macro with two operators: $n = 2$), and represent each operator of the macro with one parameter. The domain of these operator parameters is the set of operator indices: $\{1, 2, ..., |O|\}$, where $|O|$ is the number of operators in the domain.

To make the macro meaningful, we also need to link its operators. For each operator in the macro we should select one precondition to be added by the preceding sequence of operators. So, for a given operator $o_i$ in the macro $m = (o_1, o_2, ..., o_n)$, we specify the precondition predicate that is achieved by the prefix sequence of operators $(o_1, o_2, ..., o_{i-1})$ in the macro. Hence, we need two more parameters for each operator $o_i$, $i > 1$; one to specify which precondition predicate of $o_i$ we selected to link the operator to the prefix, and one to specify which positive effect predicate of the prefix we chose to fulfill that precondition. Notice that there can be two or more positive conditions with the same predicate name. In summary, for each operator we need three parameters: one is the operator number, and the other two are made of the linking predicates numbers.

Running the tuning tool with these parameters can generate many macros from which one can then be selected. The selection of macros could be achieved by running the planner with the domain, augmented with the macro, on some training instances; the macro with the smallest runtime should have the highest utility. However, invalid macros may appear as a result of the invalid substitutions that the planner
can make. Therefore a validator should also be used to check the validity of the resulting plans on each instance, and macros that result in invalid plans should have a very low utility.

Finally, the macro with the highest utility is returned. This approach can also be easily extended to generate more than one macro at a time.

Another Way of Clustering

In the previous chapter, Section 3.7.1, we discussed how macros that are not specifically designed for instance-specific learning can harm the performance. We also suggested an approach that relies on the use of macro generation tools to independently produce macros for different clusters of problem instances.

However, there remains the problem of selecting the features. One way to cluster the instances is to use the feature selector to determine the clusters. Now that the number of macros is not an issue, we can freely search for macros suitable for instance specific contexts. This search can occur in the space of both features and macros; the utility of the macro subset should represent the speed of the planner with the macro subset, as well as the ability of the features to cluster the instances in a way that makes it faster than the best-on-average performance. The best on-average performance can be found by BOA-P as described above.

The approach in Section 3.7 suffers from two problems: the macros may not be effective, and the features may not distinguish them. Even if we can generate the macros for the instances, we do not know whether the macros selected for each cluster will be significantly better than the best macro set on average. The features should be able to cluster the instances in a balanced manner that correlates the features to the best macro set of each cluster. Therefore, to ensure that the learning performs better than the best-on-average r-domain, we need to satisfy two conditions:

1. For each cluster of instances $c_j$, the best macro set $\text{PERF}_{c_j}$ should be better than BOA. This condition is meant to minimize the ceiling effect.

2. More importantly, there should be a high correlation between the features used to cluster the instances $\Phi$ and the runtime of each cluster’s $c$ macro set $\text{PERF}_c$. This condition is meant to ensure that the selected features can be used to predict the macros.

Therefore, we suggest an approach that tests both conditions before selecting a feature to cluster the instances. Given a planner and a set of instances $I_{train}$, the training algorithm should begin with an empty set of features $\Phi$, and then greedily select one feature, $\phi$, at a time, to be added to $\Phi$. Then we cluster the training instances using the new features $\Phi_2 = \Phi \cup \{\phi\}$. We use a parameter optimization tool to generate a macro set for each cluster as described above in Section 6.2.1. Next, using the union of these macros, $S$, (or using the tuning tool on all instances $I_{train}$ as described in Chapter 4) we find the best macro subset on average (BOA) on all instances.

Now, we evaluate the feature $\phi$ by measuring how closely it satisfies the two conditions described above. To measure the point to which $\phi$ satisfies the first condition, we can calculate the sum of the differences between the runtime of BOA and the best macro set on each cluster $c_j$. To measure how much $\phi$ satisfies the second condition, we can calculate the $R^2$ value of $\Phi_2$ and the best macro set of each cluster on a set of instances $I_2$. This, of course, requires running all instances of $I_2$ on the macro set of each cluster. By quantifying the satisfaction of the two conditions, we can evaluate and select the features that best fit instance-specific macro selection.
Generalization to A Novel Remodeling-Planning Framework

If the ultimate goal is improving planning performance, then we should not only choose the r-domains, but we should also choose the planner that improves performance for the given training instances. As discussed in Section 4.2, this method is not only suitable for remodeling, but it can also signal that remodeling and planning can be achieved together to reach the maximum performance. For example, when the domain cannot be improved further, we may want to look for another planner that works better with it. We might find that a particular combination of r-domains and planners works well, either in the fixed scheme or in the instance-specific scheme. Therefore, it can be useful to tune both the domain and the planner simultaneously.

To achieve that, we could add one integer-valued parameter, representing the chosen planner, to our normal binary-valued parameters that represent the macros. Or we can add a number of parameters that are used to directly control the planner’s search behavior, if any such parameters exist. With this new parameters vector, ParamILS (or perhaps another parameter tuner) would seek the planner-r-domain combination that achieves the best performance for the training instance.

Generalization to A More Flexible Remodeling

This technique opens the door for a new category of remodeling. Since the number of r-domains to be tested is not an issue anymore, we can learn r-domains built through different remodeling methods. For example, we can remove operators in addition to adding macros, or remove operators’ pre-conditions. These remodeling examples may improve the planning speed by reducing the size of the search space. However, they will also significantly increase the number of choices we have for the remodeling; thus, using a reduction approach like the one described in this chapter may become useful.

In the next chapter, we choose a new remodeling method: we learn to choose r-domains that result from adding macros and removing original operators. We begin with an exhaustive approach, as discussed in Chapter 3, and we apply ParamILS evaluation in a preliminary experiment that we discuss in detail.

4.9 Summary

In this chapter we presented a new method to remodel planning domains, in which we recast the remodeling component (the macro operators) as parameter to the domain that are tunable using a parameter optimization tool. We used the parameter tuning tools, ParamILS [50] to performs a fast local search in the space of macro subsets to find the set that improves the performance the most. We presented the approach in two contexts: a fixed remodeling context and an instance-specific remodeling context. In the instance-specific context we learned the best macro subset for clusters of training instances using ParamILS, and used the macros to train a predictor as in Chapter 3. The results show that the new approach performs as well as the traditional macro acquisition approaches in the fixed remodeling context. In the instance-specific framework, the results show that the ParamILS-based approach does not harm the performance significantly compared to the exhaustive approach, and significantly improves the pre-processing time.

This reduction of the domain components have made domain remodeling more flexible, since: (1) other optimization tools can now be used to remodel the domains, and (2) other modifiable domain components that affect the performance can be cast as a domain parameter to be tuned for performance.
This flexibility was achieved while maintaining a state-of-the-art performance, and requiring relatively small pre-processing time.
Chapter 5

Remodeling With Macros Addition and Operators Deletion

5.1 Introduction

Our goal in remodeling the planning domain is to improve the speed of planning by transforming the definition of the problem model to another that is easier for the planner to solve, and so that the solution can be reduced to a solution of the first problem. Macro operators can achieve that goal: they transform the domain, they speed up the planners by making shortcuts in the state space, and they produce plans that are reducible to valid plans in the original domain. However, macros suffer from the utility problem: too many macros may harm the performance by increasing the search branching factor.

Work has been done to reduce the number of macro operators as described in the previous chapters and in the literature. Little attention has been paid to a different way to reduce the effect of the macro utility problem. For example, Chrpa combined his operator-replacement reformulation method, called “entanglement” [18], and his macro acquisition method [17] to reformulate the domain in order to speed up planners [16]. As discussed in the literature review, Section 2.4.5, in his entanglement method [18] he reformulated the domain by replacing those operators with ground actions that have preconditions (effects) that appear only in the initial state (goal) of training instances plans with new copies of the operators that are only applicable in the initial state (goal). The macro acquisition method he used [17] relies on the dependency between actions in the domain and the problem instance. The problem with Chrpa’s approach is that it focused only on actions whose preconditions (effects) appear in the initial state (goal). There are generally not many such actions. Another problem is that the new domains are not solubility-preserving; thus, when a planner declares that there is no solution under these domains, it is not necessarily true that the problem is unsolvable.

We propose a new approach to remodeling planning domains: we add macro operators to the domain and remove original operators. The approach is more general than Chrpa’s in that it allows removing any operator from the domain while adding macros. Although removing operators still can render the domains not solubility-preserving, we make the problem more controllable by taking an empirical approach to preserve solubility.

Our approach works in both the fixed-remodeling context and the instance-specific-remodeling contexts, while empirically maintaining a high level of solubility-preserving on the training instances. We
Chapter 5. Remodeling With Macros Addition and Operators Deletion

call the approach “operator remodeling.” Our approach relies on an exhaustive search for the best combinations of macros and original domain operators that comprise the r-domains that can solve the problem instances correctly and quickly. We use machine learning, as we did in Chapter 3 and Chapter 4, to find a relation between the problem instance features and its corresponding r-domain.

Although performance may be improved just by removing some operators, the solubility-preserving of a solver with this new reformulation is not guaranteed if the reformulation is only based on empirical observations. Deciding whether an operator is necessary to solve a problem instance, in general, is PSPACE-complete [32]. To tackle this issue, we empirically maintain a minimum probability of solubility-preserving on the training instances for each accepted r-domain; if the r-domain cannot solve a proportion of the solvable problem instances, we do not accept it. We will discuss the solubility-preserving of this method in detail later.

In the next sections, we motivate our approach. Then, we present some definitions, we present our approach, we explain our claims, we show how we test those claims, we present the results, and we analyze and discuss the work in general.

5.2 Motivations

There are five factors that lead us to consider removing original operators in addition to using macro operators:

1. Reducing The Branching Factor – One of the main problems with adding macros is the resulting increase in the branching factor. Unless the macros quickly lead to good states, the planner performance will be negatively affected by this increase. The increase in branching factor is not only caused by the number of macro operators added to the domain, but also by the large number of each macro’s instantiations. This means that minimizing the number of macros does not prevent an explosive increase in the branching factor.

2. Dealing With Unsolvable Instances – Macros do not work well in unsolvable problem instances because they are shortcuts to states in the search space. In turn, proving that a problem instance is unsolvable does not require adding shortcuts to the search space graph, but does require proof that action paths do not lead to the goal.

3. Not All Operators are Needed All the Time – In an instance-specific context, not all operators are needed to solve every instance. For example, the donate operator in the mprime domain is not necessary if, in the current problem instance, there is sufficient fuel in all locations.

4. Adding Macros Increases Action Redundancy – Some operators may become redundant or unnecessary after the addition of macros. Many action-removal works [66] rely on the fact that planners cannot recognize many kinds of irrelevant information (like redundant or unnecessary actions) while solving the planning problems [18]. In the blocksworld domain for example, the pickup operator is not necessary if we add the macro operator pickup-stack. This is because after a pickup, we should either stack the block, which imitates the pickup-stack macro effects, or we putdown the block, which returns to the same state. Adding pickup-stack also makes the fact (arm-holding x), which represents the robot arm holding a block x, unnecessary. The (arm-holding x) is not a goal and is not intended to be, but we cannot infer this piece of information from the domain theory; it can be only inferred inductively from previous instances.
5. **Learning A Combination of Operators And Macros** – This kind of remodeling allows us to learn to replace the domain operators with better ones. Some sets of operators are prone to local-minima, and some are prone to dead-ends; replacing them with a new set of operators that are less prone to such situations can be very useful. For example, the 4-operator blocksworld domain is hard for the FF planner to solve because it contains local minima. However, the 3-operator blocksworld is easier to solve for FF because it does not have the same problem [45] [46]. We can remodel the 4-operator blocksworld domain by adding the following macro operators: unstack-putdown, unstack-stack, and pickup-stack. However, when we add them, we will have a large number of operators (7), thus increasing the branching factor significantly. Since we know that the 3 new operators are sufficient to solve any problem in the 3-operator blocksworld domain, and can solve reasonable 4-operators domain instances, we can remove all of the 4 original operators: pickup, putdown, stack, and unstack; this leaves the 3 new operators, improving the solving speed significantly.

Notice, however, that our artificial 3-operator blocksworld domain is not logically equivalent to the 4-operators blocksworld, because, for example, the state in which the robot arm is holding a block (meaning it contains the predicate (arm-holding x)) is unreachable. However, this particular property is neither a goal nor an initial state in the problems we normally face in this domain. This means that an exact remodeling which can represent all possible states in the original domain model is not necessary in order to find a solution to typical domain problems. There are some states that are not goals because they actually represent a transition to another state. Therefore, the idea is to only consider important steps towards the goal in the search. However, knowing if a step is important is purely a domain specific knowledge that cannot be presented in the domain; that is, we cannot prove a statement that says: “the fact (arm-holding x) cannot be in the goal or in the initial state.” However, we can learn such statement based on instances.

### 5.3 Definitions

In the following, we present the definition of some concepts important to this chapter. Specifically, we define the solubility, solubility-preserving, and coverage of r-domains and reformulations.

There is not an agreement in the literature on a term that describes the ability of a reformulation (or reduction) to preserve the solubility of instances in planning domains. For example, Chen and Yao used the term ‘completeness-preserving’ to describe their state-space reduction in planning [15]. Nebel et al. used the term ‘solution-preserving’ to refer to the same property [66]. Others like Chrpa’s did not use a specific term for the property [16]. We choose the term solubility-preserving, as we think it is easier to understand and more accurate. We use the term to describe the property of a remodeling on single instance. To describe the property for a remodeling to be solubility-preserving on a randomly chosen instance, we use another term: coverage.

We define the following, using a planning domain \( D = (\Psi, O) \); a set of sound macro operators \( S \) constructed from \( D \); a reformulation \( r_{D,S} \) constructed with \( D \) and \( S \); an r-domain \( d_{D,S} \) constructed with \( D \) and \( S \); a problem instance \( i = (s_0, g) \) from \( D \); and a distribution \( Dist_D \) over instances of \( D \).

\( i \) is **Solvable** in \( d_{D,S} \) if: there is a plan \( \pi \) constructed from the operators of the r-domain \( d_{D,S} \) that is a solution to \( i \). We informally also say \( d_{D,S} \) solves \( i \). Notice that testing whether an r-domain
solves an instance is PSPACE-Complete [29]. More formally:

\[
solvable(i, d_{D,S}) = \begin{cases} 
1 & \text{if a solution to } i \text{ constructed from } d_{D,S}'s \text{ operators exists.} \\
0 & \text{otherwise.}
\end{cases}
\]

\(d_{D,S}\) is Solubility-Preserving for Instance \(i\) if: whenever \(i\) is solvable in \(D\), it is solvable in \(d_{D,S}\). More formally:

\[
solvable(i, D) \implies solvable(i, d_{D,S}) \quad (5.1)
\]

We define a function \(sp\) as follows:

\[
sp(P, i, d_{D,S}) = \begin{cases} 
1 & \text{if condition (5.1) holds.} \\
0 & \text{otherwise.}
\end{cases}
\]

An example of a solubility-preserving r-domain is \(D_{M,S,\phi}\), since \(S\) is a sound set of macro operators and hence preserves solubility. An example of an r-domain that is not solubility-preserving for a non-trivial instance is \(D_{\phi,S,O}\), since it does not contain any operators.

The Coverage Ratio \(\gamma_{r_{D,S}}\) of a Reformulation \(r_{D,S}\) on a Distribution \(Dist_D\) is: The probability that \(r_{D,S}(i)\) is solubility-preserving for an instance \(i\) picked at random from \(Dist_D\). More formally:

\[
\gamma_{r_{D,S}} = Pr(sp(i, r_{D,S}) = 1) = E_{i \sim Dist_D}[sp(i, r_{D,S})]
\]

Notice that the coverage ratio is the expected value of the random variable representing the function \(sp()\).

In general, we say that a Reformulation \(r_{D,S}\) is Solubility-Preserving if \(\gamma_{r_{D,S}} = 1\).

\(r_{D,S}\) has a \(\gamma\)-Coverage on a Distribution \(Dist_D\) if:

\[
\gamma_{r_{D,S}} \geq \gamma \geq 0
\]

5.4 Approach

We combine two methods of remodeling a planning domain. The first involves adding macro operators to the domain description as normal operators, and the second removes original operators from the domain to improve the speed. Previous work [11], [69], including Chapter 3, has shown that acquiring and adding macros is generally helpful in a number of domains. Works such as [18] show that operator removal can be useful. However, operator removal is not a solubility-preserving method.

In this approach, offline we exhaustively construct r-domains by adding macro operators and removing original operators. By evaluating all possible r-domain combinations on training instances, we achieve two goals: we search for the r-domain that most improves the speed of the planner on the instance, and we estimate the coverage of the r-domains, since some r-domains may not be solubility-preserving.
We first obtain a set of macro operators. We assume that the macro operators provided are sound, meaning that they have the same effect and preconditions as a valid application of a particular sequence of operators. Then, we enumerate all possible sets of macro operators and/or original operators from the domain. Some of the sets are clearly not solubility-preserving, like the empty set that contains no original operators and no macros, and some are known to be solubility-preserving like the original domain’s. With each set of operators and macros, we construct an r-domain that has the same predicates and constants as the original domain.

We then generate a number of training instances. We look for the r-domain that has the smallest runtime and an acceptable estimated coverage on the training set, as we will describe in next sections. Finally, we construct learning models designed to predict, based on the features of the problem instance, the best r-domain for that instance. As in Chapter 3, we have two prediction models: a time model and a direct model.

Notice that solubility-preserving r-domains, as described in the previous section, should eventually solve all the instances and return a valid solution plan if there is one, or return ‘no plan’ if the instance has no solution. Other r-domains may or may not solve a solvable problem instance. Therefore, we want to filter the r-domains by maintaining a lower bound on their coverage ratio, which is defined in Section 5.3. However, we cannot know whether a given r-domain is solubility-preserving for a problem instance if the planner times out while solving it. Thus, we cannot directly find an expectation of the r-domain’s γ-coverage in general. Therefore, we include the possibility of a timeout in the r-domain’s coverage ratio and the related definitions, as will be discussed next.

5.4.1 Approximating Coverage

We can approximate the coverage ratio \( \gamma_{r,D,S} \) for a reformulation \( r_{D,S} \) by sampling instances from the distribution \( \text{Dist}_D \) and finding whether \( r \) is solubility-preserving on these instances. However, this requires finding the value of the \( \text{solvable}(i, r_{D,S}(i)) \) function, which is a PSPACE-hard problem. Therefore, we propose a second approximation to \( \gamma_{r,D,S} \), that introduces new definitions to take the timeout of the planner into account. Specifically, we combine the cutoff time \( c \) with the definitions of solubility, solubility-preserving, and coverage.

For a domain \( D = (\Psi, O) \), a reformulation \( r_{D,S} \), an r-domain \( d_{D,S} \), an instance \( i \) from \( D \), a planner \( P \), and a cut-off time constant \( c \), we present the following definitions:

\( i \text{ is } c\text{-Solvable in } d_{D,S} \text{ by } P \) if: running \( P \) with \( i \) and \( d_{D,S} \) returns a valid solution plan to \( i \) within time \( c \). If \( P \) halts normally and returns no plan within time \( c \), then \( i \) is not \( c \)-solvable by \( d \) on \( P \). More formally, we define the function:

\[
c\text{-Solvable}(P, i, d_{D,S}) = \begin{cases} 
0 & \text{if } P \text{ returns ‘no plan’ within time } c. \\
1 & \text{if } P \text{ returns a solution plan to } i \text{ within } c.
\end{cases}
\]

\( c\text{-Solvable} \) is not defined if \( P \) times out after time \( c \) or returns an error.

\( d_{D,S} \text{ is } c\text{-Solubility-Preserving for } i \) if: whenever planner \( P \) returns a plan for \( i \) using some r-domain \( d'_{D,S} \), constructed from \( D \) and \( S \), within time \( c \), we find that running \( P \) with \( d_{D,S} \) and \( i \) also returns a solution plan within time \( c \). More formally:
Chapter 5. Remodeling With Macros Addition and Operators Deletion

(∃d′_{D,S} : c-Solvable(P, i, d′_{D,S})) \implies c-Solvable(P, i, d_{D,S}) \tag{5.2}

We define a function c-sp as follows:

\[
c-sp(P, i, d) = \begin{cases} 
1 & \text{if condition (5.2) holds.} \\
0 & \text{if condition (5.2) does not hold.}
\end{cases}
\]

Now that we have defined solubility and solubility-preserving in the presence of time outs, we can approximate the coverage ratio of a reformulation. Remember that the coverage ratio is the expected value of the sp function, which tells whether a r-domain is solubility preserving for an instance. To approximate \( \gamma_{r_{D,S}} \), we sample a set of instances \( I \) from \( Dist_D \) and find the number of instances on which \( r_{D,S} \) is solubility-preserving. However, it may be impossible to tell whether some instances are solubility-preserving. For example, instances on which all r-domains timed out cannot be used in this calculation, since their c-solubility preserving, which is the practical form of solubility-preserving, is undefined. Therefore, we need to filter the samples to include only those for which we surely know that \( r_{D,S} \)'s c-solubility-preserving is defined. We do that by introducing a subset we call the accepted sampling subset of \( I \).

The Accepted Sampling Subset \( I_{r_{D,S}}^* \) of \( I \) with respect to \( r_{D,S} \) is: the set of instances \( i \in I \) for which the c-sp(\( P, i, r_{D,S}(i) \)) is defined. The value of c-sp is defined only when \( r_{D,S} \) did not timeout on the instance, and either there is an r-domain that c-solves the instance or a macro-only r-domain did not timeout on the instance.

More formally:

\[
I_{r_{D,S}}^* = \{i | i \in I, c-Solvable(P, i, r_{D,S}(i)) \text{ is defined, and } (\exists d' : c-Solvable(P, i, d') = 1, \text{ or } \exists M \subseteq S : c-Solvable(P, i, D_M, S, \emptyset) \text{ is defined.}) \}
\]

The c-Coverage Ratio \( \hat{\gamma}^l_{r_{D,S}} \) of \( r_{D,S} \) Using \( I \) is: the proportion of instances from the accepted sampling subset of \( I \), \( I_{r_{D,S}}^* \), on which \( r_{D,S} \) is c-solubility-preserving to the total instances of \( I_{r_{D,S}}^* \).

More formally:

\[
\hat{\gamma}^l_{r_{D,S}} = \frac{\sum_{i \in I_{r_{D,S}}^*} c-sp(P, i, r_{D,S}(i))}{|I_{r_{D,S}}^*|}
\]

For example, when \( d = D_{M,S,\emptyset} \), we know that \( \hat{\gamma}^l_{d} = 1 \) if \( d \) can c-solve one instance of \( I \). Notice that \( \hat{\gamma}^l_{r_{D,S}} \) is not defined if \( |I_{r_{D,S}}^*| = 0 \). This can happen when all macro-addition r-domains time out and none of the other r-domains returns a solution plan within time \( c \). In this case, there is no r-domain that can be used to know whether the instance is really unsolvable, since r-domains that remove operators are not solubility-preserving in general.

For simplicity, we may omit the instances set name \( I \) from the c-coverage ratio when it is understood from the context. For example, we may use \( r_{D,S} \) has c-\( \gamma \)-Coverage on Instances \( I \) if:

\[
\hat{\gamma}^l_{r_{D,S}} \geq \gamma \geq 0
\]
For example, $D_{M,S,\emptyset}$ has $c\cdot\gamma$-Coverage on $I$ for any $\gamma$, since $\hat{\gamma}_{D_{M,S,\emptyset}}^I = 1$.

### 5.4.2 System Details

We learn to remodel planning domains by adding macros and removing basic operators while maintaining a minimum coverage. Figure 5.1 shows the modified framework under this approach. The approach is similar to Chapter 3’s approach except that we use a new component, the Operator Selector, to choose the domain operators to consider for removal. This gives us control over the number of r-domains generated.

Given a domain $D = (\Psi, O)$, a planner $P$, a set of sound initial macro operators $S$, and the minimum coverage ratio $\gamma$, the offline training phase of our instance-specific remodeling method works as follows:

1. We run the operator selector to choose the operators to be removed. If we know that the removal of an individual operator $o$ will increase the number of unsolvable instances above an acceptable level, we do not choose it. This step is analogous to the step in which the macro source filters out the less useful individual macros in Section 3.3.3. The operator selector performs the following steps:

   (a) Given a set of small problem instances $I_{sel}$, the planner is run on each instance in $I_{sel}$ with two r-domains: the r-domain that contains all initial macro operators $S$ and original operators $D_{S,S,\emptyset}$, and the r-domain $D_{S,S,\{o\}}$, which is the original domain augmented with the initial macros $S$, but with one basic operator $o$ removed.

   (b) We calculate $D_{S,S,\{o\}}$’s average coverage ratio $\hat{\gamma}_{D_{S,S,\{o\}}}^I$, which is the proportion of instances solved correctly without $o$ divided by the total number of correctly solved instances. Here, we have two r-domains to choose from: the domain with all macros and all operators ($D_{S,S,\emptyset}$) and the domain with all macros and operators except $o$ ($D_{S,S,\{o\}}$). So:

   $$\hat{\gamma}_{D_{S,S,\{o\}}}^I = \frac{\sum_{i \in I_{sel}^*} c\cdot\text{solvable}(P, i, D_{S,S,\{o\}})}{|I_{sel}^*|},$$

   where $I_{sel}^* = \{i | i \in I_{sel}, c\cdot\text{solvable}(P, i, D_{S,S,\emptyset}) \text{ and } c\cdot\text{solvable}(P, i, D_{S,S,\{o\}}) \text{ are defined.}\}$

   (c) If $\hat{\gamma}_{D_{S,S,\{o\}}}^I < \gamma$, we must not choose operator $o$ for removal, since we know that any r-domain that does not contain that operator is likely to be solubility-preserving only on a proportion...
of instances less than $\gamma$. If $\gamma_{I_{\text{sel}}}^{I_{\text{sel}}} \geq \gamma$, then we can add $o$ to the set of removable operators $Q$ that is initially empty.

(d) We repeat the steps (a),(b) and (c) for all domain operators, or until $|Q| = q$, where $q$ is a constant of the maximum number of operators we would like to remove. To break ties as a result of $q$, since more than $q$ operators may be removable, we add to $Q$ the operators $o$ whose corresponding r-domains, $D_{S,S,\{o\}}$, have the smallest average runtime on the instances of $I_{\text{sel}}$.

2. We generate all possible r-domains that result from adding macros and/or removing operators in $Q$: $\{D_{M,S,L}|M \subseteq S, L \subseteq Q\}$

3. We exhaustively run all training instances $I_{\text{train}}$ on these r-domains and register the results.

4. We accept only r-domains that solve at least a proportion $\gamma$ of the training instances correctly, and we discard the other r-domains. So:

$$\text{accept} = \{D_{M,S,L}|M \subseteq S, L \subseteq Q, \gamma_{D_{M,S,L}}^{I_{\text{train}}} \geq \gamma\}$$

We know that some r-domains should be accepted unless all instances time out. For example, the original domain model $D$ (with no macros added or operators removed) should be accepted because $\gamma_{D}^{I_{\text{train}}} = 1$, and all r-domains that only add macros should be accepted.

5. We use the same approaches of Chapter 3 to train predictors; i.e., we select features and use machine learning algorithm to train two prediction models: the Time predictor and the Direct predictor.

This approach builds on our previous approach of instance-specific macro addition. However, the feature selection is different because we need to consider the correlation between the removable operators and the features. Next we show the modification we made to the feature selection.

5.4.3 Feature Selection

Only one part of the feature selector in this section is changed. In addition to evaluating the features according to their correlation with the runtime of the domain $D$ augmented with single macros from $S$, we also evaluate them using their correlation with the domain $D$ augmented with all macros in $S$ and with one original operator $o \in Q$ removed.

The only procedure that needs to be changed is FindRSQ as shown in Algorithm 6. The function $\text{runtime}(D_{S,S,\{o\}})$ returns a list of runtimes of the planner on the training instances and the r-domain $D_{S,S,\{o\}}$. So, $r_{\text{total}}$ will include the correlation to the operators as well as the macros.

5.5 Claims and Research Questions

The use of macros discussed in Chapter 3 suffers from an obvious drawback: it increases the number of redundant actions in the problem instance. While maintaining a probability of solubility-preserving in the training data as described above, we claim that the performance of the best fixed r-domain that is equipped with operator deletion and macro addition is significantly faster than the best fixed r-domain using only macro addition.
Algorithm 6: FindRSQ

<table>
<thead>
<tr>
<th>input</th>
<th>features: ( \Phi ), macros: ( S ), domain: ( D = (\Psi, O) ), runtime lists: ( \text{runtime} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>total R-squared value: ( r_{total} )</td>
</tr>
</tbody>
</table>

1 begin
2 \( r_{total} \leftarrow R^2(\text{runtime}(D_{\emptyset}, S, \emptyset), \Phi) \);
3 \textbf{foreach} \( m \in S \) \textbf{do}
4 \( r \leftarrow R^2(\text{runtime}(D_{\{m\}}, S, \emptyset), \Phi) \);
5 \( r_{total} \leftarrow r_{total} + r \);
6 \textbf{end}
7 \textbf{foreach} \( o \in O \) \textbf{do}
8 \( r \leftarrow R^2(\text{runtime}(D_S, S, \{o\}), \Phi) \);
9 \( r_{total} \leftarrow r_{total} + r \);
10 \textbf{end}
11 end

As we consider the instance-specific approach, we no longer need to think of all operators as necessary, since some instances do not need all operators in order to reach the goal. We claim that learning to choose the best r-domains in an instance-specific context is feasible and can improve planning speed significantly, as compared to the fixed r-domains scheme.

Therefore we would like to test the following hypotheses:

**Hypothesis 3.1** While maintaining a high c-Coverage ratio \((\geq \gamma)\), the performance of the best fixed r-domain built with operator remodeling is significantly better than the performance of the best fixed r-domain built using macro addition only.

**Justification** The branching factor increase caused by macros can be reduced if we remove basic operators, a process that can speed up the search [66]. However, removing operators is not a solubility-preserving approach, i.e., some problem instances that are solvable using a planner can become unsolvable when we remove an operator. It is known that proving the solubility preservation of action removal approaches is as hard as planning [32]. So, accepting r-domains that solve a proportion \( \gamma \) of the solvable training instances makes the operator removal approach more practical.

**Hypothesis 3.2** While maintaining a high c-Coverage ratio \((\geq \gamma)\), the performance of the instance-specific prediction models that rely on operator remodeling is significantly better than using fixed r-domains that rely on the same kind of remodeling.

**Justification** In this kind of remodeling, we expect even more performance improvement, since we know that in general some operators are not needed in every problem instance. Operators might not be necessary, either because the domain is rich enough to tolerate their absence in some instances, or because the addition of macro operators renders them redundant.

### 5.6 Experimental Design

To test the two hypotheses, we conduct one experiment with different macro sources. We use the FF planner, and we use six planning domains (four hard and two easy.) The following describes the experimental settings and tests.
5.6.1 Settings

Hardware and Software Details

We use the same hardware and software as in the previous chapters. Refer to Section 3.5.1 for details on this subject.

Planners

We used the FF planner only. The VHPOP caused errors that made prediction harder in the experiments of previous chapters, so we decided not to use it here.

Domains, and Macro Sources

We used the same set of benchmark domains as we used in previous chapters. We also used the same macro sources of previous chapters: Wizard, Macro-FF and the macros we manually constructed. See Section 3.5.1 for details on this subject.

Problem Instances

In the remodeling experiment we use a new set of instances because we had noticed that the number of training instances can contribute to better learning. We generate the instances using the problem generators discussed earlier in Section 3.5.1 and Appendix C. The parameters used to generate the new problems of each domain are shown in Table 5.1.

<table>
<thead>
<tr>
<th>Domain</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics</td>
<td>( a \in [1,3], c \in [4,6], s \in [4,6], p \in [50,62] )</td>
</tr>
<tr>
<td>blocksworld</td>
<td>( n \in [2,50] )</td>
</tr>
<tr>
<td>pipesworld</td>
<td>( p \in [2,4], b \in [3,5], g \in [1,2] )</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>( p \in [2,4], b \in [3,5], g \in [1,2] )</td>
</tr>
<tr>
<td>mprime</td>
<td>( t \in [5,5], f \in [30,30], s \in [1,2], v \in [1,2], c \in [5,8] )</td>
</tr>
<tr>
<td>mystery</td>
<td>( t \in [5,5], f \in [30,30], s \in [1,2], v \in [1,2], c \in [5,8] )</td>
</tr>
<tr>
<td>freecell</td>
<td>( f \in [4,4], c \in [8,8], s \in [4,4], l \in [13,13] )</td>
</tr>
</tbody>
</table>

Table 5.1: The parameters used to generate the training and test instances. A description of these parameters can be found with each domain generator. In regard to the pipesworld, the parameters \( p, b, g \) represent the number of pipes, the number of extra batches (added to a fixed number of batches that depends on \( p \)), and the number of goals, respectively. Refer to Appendix C for more information.

Features

As shown in Algorithm 6, we modified the feature selection algorithm by evaluating the features with respect to the deleted operators as well as to the added macros. The rest of the feature selection algorithm is found in Subsection 3.3.4.

The resulting taxonomy syntax features, the freecell domain features and the simple facts features are described in Appendix D.

Timeout Handling

We used the same cut-off time, \( c \), as was used the exhaustive macro-addition experiment for both the training and the testing: one hour of CPU time.
Operator Selector

We chose \( q = 3 \) to be the maximum number of removable operators \( Q \). In two domains: mystery-FF-WIZARD and pipesworld-nt-FF-WIZARD the feature selector did not select any operator, i.e. \( Q = \{ \} \); in those cases, we used the results presented in Chapter 3, since the resulting r-domains are only macro-augmented ones. The selected operators for all domains are shown in Table 5.2. For more information about the domains’ original operators, refer to Appendix A.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Selected Operators (Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF-WIZARD</td>
<td>{load-truck, unload-truck, unload-airplane}</td>
</tr>
<tr>
<td>logistics</td>
<td>{pickup, stack, unstack}</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>{}</td>
</tr>
<tr>
<td>mystery-5</td>
<td>{}</td>
</tr>
<tr>
<td>freecell</td>
<td>{sendtofree, sendtofree-b, newcolfromfreecell}</td>
</tr>
<tr>
<td>FF-MACROFF</td>
<td>{pickup, stack, unstack}</td>
</tr>
<tr>
<td>blocksworld</td>
<td>{push-start, push-end}</td>
</tr>
<tr>
<td>freecell</td>
<td>{sendtohome}</td>
</tr>
<tr>
<td>FF-MANUAL</td>
<td>{load-truck, unload-truck, unload-airplane}</td>
</tr>
<tr>
<td>logistics</td>
<td>{pickup, putdown, unstack}</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>{push-start, pop-start, pop-end}</td>
</tr>
<tr>
<td>mprime-5</td>
<td>{move, donate}</td>
</tr>
<tr>
<td>mystery-5</td>
<td>{move, load, unload}</td>
</tr>
<tr>
<td>freecell</td>
<td>{sendtofree, sendtofree-b, colfromfreecell}</td>
</tr>
</tbody>
</table>

Table 5.2: The operators selected for potential removal in each domain.

Coverage

We set the minimum coverage ratio \( \gamma \) to 0.98. In the training instances, we approximate the coverage ratio of an r-domain \( d \), \( \gamma_d \), by using the \( c \)-Coverage definition as presented in Section 5.4. As discussed, we accept r-domains \( d \) where \( \hat{\gamma}_d^\text{train} \geq \gamma \) for the learning.

To test the coverage of the reformulations, we calculate the \( c \)-Coverage ratio of the r-domains and reformulations on \( I_{test} \), \( \hat{\gamma}_d^\text{test} \), as defined in Section 5.4. We say that a reformulation \( d \) maintained a high coverage ratio if \( \hat{\gamma}_d^\text{test} \geq \gamma \).

Utility

We define utility under the instance-specific operator learning. Given a planner \( P \), a domain \( D \), a problem instance \( i \), and a set of macro operators \( S \), the utility of an accepted r-domain \( D_{M,S,L} \) on the problem instance \( i \) is defined as follows.

\[
U_{(i,P)}(D_{M,S,L}) = \begin{cases} 
-\text{runtime}_P(i, D_{M,S,L}) & \text{if P naturally halts without interruption.} \\
-c & \text{otherwise}
\end{cases}
\]

Where \( \text{runtime} \) and \( c \) are as discussed earlier. The utility is undefined for r-domains that are not accepted.
Models to Compare

Given a planner \( P \), a domain \( D = (\Psi, O) \), an initial macro set \( S \), training and testing instances \( I_{\text{train}}, I_{\text{test}} \), we compare the performance of the following common domains models on the testing instances:

**ORIG, BOA-M** are the same as defined earlier, where BOA-M is BOA. The ‘M’ added here stands for Macros, and we use it to indicate a distinction between the new best-on-average r-domain (BOA-R) and the macro-addition only r-domain.

**BOA-R** is the best r-domain on average based on exhaustive evaluation of all accepted r-domains on the training instances.

\[
\forall i \in I_{\text{test}} \text{ BOA-R}(i) = \arg_{D_{M,S,L}} \max U(I_{\text{train}}, P)(D_{M,S,L}),
\]

where \( \hat{\gamma}^{I_{\text{train}}}_{D_{M,S,L}} \geq \gamma \)

**PERF-R** is the perfect predictor based on exhaustive evaluation of all accepted domain models on the testing instances.

\[
\forall i \in I_{\text{test}} \text{ PERF-R}(i) = \arg_{D_{M,S,L}} \max U(i, P)(D_{M,S,L}),
\]

where \( c \cdot \text{sp}(P, i, D_{M,S,L}) = 1 \).

**DIR-R** is the direct predictor based on the exhaustive evaluation of all accepted domain models on the training instances.

\[
\forall i \in I_{\text{test}} \exists M \subseteq S, L \subseteq O: \text{DIR-R}(i) = D_{M,S,L} \text{ where } (M, L) = \text{Classify}(i, T_{X,U_1}), \text{ and } T_{X,U_1} = \{(X(i), U_1(i)) | i \in I_{\text{train}}, U_1(i) = \arg_{(M', L')} \max U(i, P)(D_{M',S,L'}) \text{ s.t } D_{M',S,L'} \text{ is accepted } \}
\]

As in previous chapters, \( \text{Classify} \) is chosen to be an SVM-SMO classification algorithm with a quadratic kernel from the WEKA [86] tool. In this experiment, we set \( r \), the number of top r-domains to be considered in the training of RID-R to 16.

**TIME-R** is the time predictor based on the exhaustive evaluation of all accepted domain models on the training instances.

\[
\forall i \in I_{\text{test}} \exists M' \subseteq S, L' \subseteq O: \text{TIME-R}(i) = D_{M',S,L'}, \text{ where } (M', L') = \arg_{(M, L)} \min \text{Reg}(i, T_{X,U_2}), \text{ and } T_{X,U_2} = \{(X(i), U_2(i)) | i \in I_{\text{train}}, U_2(i) = U(i, P)(D_{M,S,L})\}
\]

Reg is as defined in subsection 4.5.

As in previous chapters, Reg is chosen to be the M5P regression algorithm from the WEKA [86] tool. In this experiment, we set \( r \), the number of top r-domains to be considered in the learning of TIME-R, to 16.

5.6.2 Tests

To test hypothesis 3.1, we compare the performance of the fixed operator remodeling to the fixed macro-only remodeling. We specifically compare the average runtimes of BOA-R to BOA-M. We also note the
difference between BOA-R to ORIG in order to know the magnitude of improvement over the domain without any remodeling.

To test hypothesis 3.2, we compare the performance of the operator remodeling under the instance-specific scheme to that under the fixed scheme. We specifically compare the average runtime of PERF-R to BOA-R to know if the learning is feasible. We expect a result similar to that described in Chapter 3: that PERF-R is significantly faster than BOA-R in the hard domains. Then, we compare the average runtime of the predictors DIR-R and TIME-R to the best average r-domain, BOA-R.

In all cases, we also find the coverage ratio of the new r-domains and predictors. We expect their coverage ratio to be higher than $\gamma$.

5.7 Experimental Results

The results show that BOA-R is faster than BOA-M in most domains. TIME-R and DIR-R are significantly faster than BOA-R in a number of domains; they are also faster than BOA-M in many domains, sometimes by orders of magnitude.

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA-M</th>
<th>BOA-R</th>
<th>TIME-R</th>
<th>DIR-R</th>
<th>PERF-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocksworld (225)</td>
<td>127.86</td>
<td>0.52</td>
<td>0.05</td>
<td>*0.07</td>
<td>*0.06</td>
<td>*0.04</td>
</tr>
<tr>
<td>Pipesworld-nt (213)</td>
<td>161.51</td>
<td>167.47</td>
<td>167.47</td>
<td>161.52</td>
<td>177.82</td>
<td>*101.26</td>
</tr>
<tr>
<td>Mystery-5 (216)</td>
<td>155.64</td>
<td>155.64</td>
<td>155.64</td>
<td>172.50</td>
<td>132.45</td>
<td>*20.26</td>
</tr>
<tr>
<td>Freecell-A (199)</td>
<td>513.21</td>
<td>359.14</td>
<td>234.75</td>
<td>239.16</td>
<td>118.09</td>
<td>*10.03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA-R</th>
<th>TIME-R</th>
<th>DIR-R</th>
<th>PERF-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocksworld (225)</td>
<td>127.86</td>
<td>0.52</td>
<td>0.05</td>
<td>*0.07</td>
<td>*0.06</td>
</tr>
<tr>
<td>Pipesworld-nt (213)</td>
<td>161.51</td>
<td>167.47</td>
<td>167.47</td>
<td>161.52</td>
<td>177.82</td>
</tr>
<tr>
<td>Mystery-5 (216)</td>
<td>155.64</td>
<td>155.64</td>
<td>155.64</td>
<td>172.50</td>
<td>132.45</td>
</tr>
<tr>
<td>Freecell-A (199)</td>
<td>513.21</td>
<td>359.14</td>
<td>234.75</td>
<td>239.16</td>
<td>118.09</td>
</tr>
</tbody>
</table>

Table 5.3: Average runtimes in seconds followed by the number of incorrectly solved test instances (between the parentheses) in the cells for the common reformulations using FF and three macro sources. The asterisk (*) (the plus (+)) means that the model was significantly faster than BOA-R (BOA-M) with $p \leq 0.05$.

Table 5.3 shows the results of the remodeling using macro addition and operator removal. Under each domain name, the number of test instances is shown between parentheses. The numbers in the remaining cells represent the average runtime and the number of instances solved incorrectly by the corresponding r-domain. In the FF-WIZARD-pipesworld-nt and FF-WIZARD-mystery-5 domains, the
operator selector did not choose any operator to remove, and therefore they are identical to the macro-only experiment.

In 5 out of 14 domains, BOA-R was significantly faster than BOA-M. In the rest, BOA-R was either marginally better or equal to BOA-M, except in FF-MACROFF-pipesworld-nt, where it was marginally worse. There was a significant improvement in the average performance and a very high coverage ratio was maintained. In the blocksworld domain for example, the performance of BOA-R was significantly faster than BOA-M by an order of magnitude.

The experiment results were similar to the macro addition experiment in terms of learning feasibility: the imaginary perfect prediction model (PERF-R) was significantly faster, often a number of times faster, than BOA-R in the domains that are not solvable by FF’s local search (mystery, mprime, freecell, pipesworld). In the easy domains it was faster, but only by a small margin.

In 3 out of 14 domains, TIME-R was significantly faster than BOA-R. In 5 out of 14, DIR-R was significantly faster than BOA-R. BOA-R was not significantly faster than the predictors in any domain. Compared to BOA-M, each of TIME-R and DIR-R was significantly faster in 7 and 9 domains, respectively, and in some cases by orders of magnitude.

5.8 Analysis and Discussion

5.8.1 Analysis

By showing that the best fixed r-domain that is based on operator remodeling (BOA-R) was significantly faster in many cases than the best macro-augmented r-domain (BOA-M), we have shown that our approach, which adds macros and removes operators, significantly improves performance when compared to the approach that only adds macros.

The results also show that the perfect predictor in the operator remodeling context (PERF-R) was significantly faster than BOA-R. In a number of domains, the predictors (TIME-R and DIR-R) have significantly improved in terms of speed compared to the fixed-remodeling approach (BOA-R). The predictors also were significantly faster than BOA-M in many domains, sometimes by orders of magnitude. All results hold while TIME-R, DIR-R, BOA-R and PERF-R maintain a high coverage ratio ($\geq \gamma$).

These results support the hypotheses 3.1 and 3.2. In general, they show that instance-specific operator remodeling can achieve a significant planning speed up in a number of domains with orders of magnitude compared to macro-only schemes and compared to the original domains.

Solubility Preservation

Although this method is not solubility-preserving, we found that the resulting reformulations (the instance-specific predictors and the fixed predictor) almost always solved all solvable test instances correctly. It was only in the FF-MANUAL-Mystery-5 domain that BOA-R and TIME-R could not solve four and two solvable instances, respectively. This represents a coverage ratio of 0.986 and 0.993 of the test instances, which is greater than the 0.98 minimum coverage margin allowed for each r-domain. In all domains, the DIR-R predictor solved all test instances correctly or timed out. This shows that the negative side effects caused by the fact that operator removal is not solubility-preserving are mostly controllable, while the resulting models are indeed significantly faster.
Torchlight Analysis and Performance

Our results indicate that the new remodeling method can replace a domain with a new domain that makes the search faster. It is interesting to observe the extent to which the performance of the new approach is related to the search topology. So, as in Chapter 3, we use the Torchlight tool to analyze the new reformulations and r-domains. We run Torchlight on three domains from the FF-MANUAL experiment: blocksworld, mprime, and mystery. The freecell and pipesworld domains caused errors in Torchlight’s preprocessing step.

In Chapter 3 we observed that the search topology was significantly improved in the blocksworld-4ops domain because of the added macro operators. In blocksworld, we expect to find even more improvement in the search topology by removing original operators. According to [45], the 3-operators version of blocksworld does not contain local minima under the h+ heuristic while the 4-operators version does contain local minima. We also know that some original operators in the 4-operators blocksworld will be made redundant by adding our macros; for example, the unstack operator became unnecessary by adding the macros unstack-putdown and unstack-stack (see Table 5.2 in Section 5.6.1). Therefore, we expect to find a significant improvement in the search topology after undertaking the operator remodeling compared to the topology in the original domain and macro-only remodeling.

However, we do not expect a significant improvement in the search topology in mprime or mystery domains. For these hard domains, mprime and mystery, the positive effect of the new remodeling on the search topology is not clear. These domains contain dead-ends, which forces the planner to backtrack, which significantly harms the progress towards the goal. So, in these domains, we should examine the effect of the two remodeling methods we follow: macro addition and operator removal.

By adding macro operators to hard domains, we add a number of shortcuts that are known to be useful to the search space. If the shortcut leads to a state on the path to a goal state, then the shortcut may improve the performance. But if the shortcut is to a strongly connected component of dead-end states, then it may harm the performance. Similarly, removing operators makes some reachable states unreachable because actions are removed. If the removed actions lead to a goal state, then the performance may be harmed. But if the removed actions originally led to dead-ends, the performance may be improved. So, in hard domains, the performance is dependent on whether the added and/or removed actions lie on goal paths or paths to dead-ends. However, it is difficult to guess what will happen to the search topology if we add and/or remove operators only based on their runtime on other instances.

In relation to each of the four domains, we look at eight reformulations: ORIG, BOA-M, BOA-R, TIME-R, DIR-R, PERF-M, PERF-R, and RAND. PERF-M is the perfect predictor using macros only, and RAND is the reformulation that randomly chooses an r-domain uniformly. As described earlier in Chapter 3, Torchlight generates $s$ random states by random application of actions to a certain depth, which is a function of the parameter $d$. We chose default parameter values: $s = 100, d = 5$ to run Torchlight. Torchlight outputs describe the number of success states, which are states not on local minimum, and the number of dead-end states, all from the set of $s$ random states. We run Torchlight on each test instance using each accepted r-domain, and we find the average percentage of success states and dead-end states for each reformulation. The results are shown in Table 5.4 and Table 5.5.

In blocksworld, the reformulation clearly improved the search topology. Recall that we do not remove all operators in blocksworld, so we cannot replace the four-operator blocksworld with the three-operator domain. In terms of local minima, BOA-M achieved a significant improvement over ORIG, as it did...
Chapter 5. Remodeling With Macros Addition and Operators Deletion

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA-M</th>
<th>BOA-R</th>
<th>TIME-R</th>
<th>DIR-R</th>
<th>PERF-M</th>
<th>PERF-R</th>
<th>RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocksworld</td>
<td>27.84</td>
<td>79.6</td>
<td>92.71</td>
<td>74.97</td>
<td>99.47</td>
<td>64.74</td>
<td>89.7</td>
<td>75.51</td>
</tr>
<tr>
<td>Mystery-5</td>
<td>32.01</td>
<td>46.99</td>
<td>29.36</td>
<td>31.21</td>
<td>31.68</td>
<td>38.04</td>
<td>32.56</td>
<td>28.59</td>
</tr>
<tr>
<td>Mprime-5</td>
<td>55.51</td>
<td>49.65</td>
<td>49.65</td>
<td>49.73</td>
<td>41.23</td>
<td>42.89</td>
<td>38.51</td>
<td>45.79</td>
</tr>
</tbody>
</table>

Table 5.4: Torchlight analysis results of some domains tested in FF-MANUAL experiment. The number in each cell represents the average percentage of success states according to torchlight analysis.

<table>
<thead>
<tr>
<th>Domain</th>
<th>ORIG</th>
<th>BOA-M</th>
<th>BOA-R</th>
<th>TIME-R</th>
<th>DIR-R</th>
<th>PERF-M</th>
<th>PERF-R</th>
<th>RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocksworld</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.31</td>
<td>0.59</td>
</tr>
<tr>
<td>Mystery-5</td>
<td>43.61</td>
<td>33.92</td>
<td>50.25</td>
<td>48.31</td>
<td>45.01</td>
<td>38.97</td>
<td>45.41</td>
<td>52.04</td>
</tr>
<tr>
<td>Mprime-5</td>
<td>5.00</td>
<td>7.33</td>
<td>7.33</td>
<td>9.04</td>
<td>12.28</td>
<td>24.11</td>
<td>15.39</td>
<td>13.41</td>
</tr>
</tbody>
</table>

Table 5.5: Torchlight analysis results of some domains tested in FF-MANUAL experiment. The number in each cell represents the average percentage of dead-end states according to torchlight analysis.

in Chapter 3. But BOA-R achieved even more improvement than BOA-M. DIR-R achieved the best improvement: it results in a search topology with close to 100% success states. PERF-R also improves PERF-M significantly. In blocksworld, all reformulations had no dead-ends except PERF-R, since it can choose r-domains that are not solubility-preserving to solve some instances quickly. In these instance, there might be a straight path from the initial state to the goal, but when Torchlight takes random actions from the initial state, it may arrive at a state from which it cannot find a path to a goal state.

In the hard domains (mprime and mystery) the picture is different. The perfect predictor PERF-R has fewer success states and more dead-ends than the BOA-R, although the former is orders of magnitude faster. Also, BOA-R generally has fewer success states and more dead-ends than BOA-M. The predictors are also worse than BOA-M. In the these domains, all reformulations are generally close to RAND.

Although evaluating the r-domains based on runtime can lead to choosing r-domains that are faster, it may also lead to a harder-to-navigate search space. For example, in the mprime-5, PERF-R has less success and more dead-end states than ORIG, and in mystery-5 PERF-R has almost the same number of success and dead-end states as ORIG. This suggests that using runtime only in the utility is not enough to improve search topology in hard domains.

5.8.2 Discussion

Issue of Solubility Preservation

Again, lack of solubility preservation is a disadvantage of this approach. However, resolving the solubility preservation is as hard as planning [32]. Therefore, we should examine the quality of our solubility-preserving approximation. While we do not guarantee a minimum coverage ratio, we do find a minimum expected coverage of the fixed reformulations by accepting only r-domains with average $c$-Coverage ratios $\geq \gamma$. The results show that the accepted r-domains (e.g. BOA-R) have $c$-Coverage ratios $\geq \gamma$ on the test instances. For the predictors, the minimum expected coverage is not guaranteed since the predictor can choose a different r-domain for each instance, which may result in smaller average $c$-Coverage ratios than $\gamma$. Empirically, however, this was not observed in the results, as the DIR-R and TIME-R have $c$-Coverage ratios that are equal to or better than BOA-R in all domains.
Other Ways To Evaluate the r-Domains

The runtime-based utility function used is not very informative. The preference for the r-domains is based on the runtime on the problem instance; this is done without taking account of the unobservable effects resulting from macro addition and operator removal on the search space or the heuristic search topology. An interesting idea would be to include some domain analysis results in the utility formulation. This is interesting because we cannot guarantee that the heuristic will not follow a different path to the goal on a similar problem instance, therefore we should also account for the heuristic function’s behavior.

However, while using the runtime does not limit us to specific planners, using a heuristic local search analysis is planner-specific: the planner’s heuristic should be known. Even if we find that an r-domain improves the search topology of a particular heuristic, we still cannot guarantee that it will improve the search topology of another planner that does not use that heuristic. For example, Torchlight analyzes the domains under the h+ heuristic topology, and FF approximates the h+ heuristic; therefore it is perhaps a good idea to include Torchlight analysis results in the utility of the r-domains in addition to the runtime. However, using Torchlight with VHPOP is not necessarily useful, since VHPOP is less reliant on the h+ heuristic. Even though VHPOP uses a variation of the HSP heuristic – also an approximation of h+ – in its operator-selection, it uses a number of other heuristics for the fault-selection.

The Exponential Number Of r-Domains

We are still faced with main problem from Chapter 3: the number of r-domains to evaluate is exponential to the number of macros and domain operators. In this chapter, we choose a maximum number $q = 3$ of operators to remove. This may help to control the exponential growth of the number of subsets but will limit the applicability of our approach. In Chapter 4, we suggested a parameter-configuration-based approach to this problem in the macro-addition-only case. We suggested using ParamILS for each cluster of instances to resolve the exponential increase in the number of r-domains.

It is beyond the scope of this chapter to cover the integration of the instance-specific operator remodeling with the ParamILS evaluation method. In this subsection, however, we introduce some preliminary experiments in which we attempt to integrate the non-solvability-preserving operator remodeling with the ParamILS evaluation method.

The r-domain’s utility we used in Section 5.6.1 requires exhaustive r-domains evaluation on all instances before the coverage ratio can be found. If we want to apply ParamILS evaluation to the r-domains of the operator remodeling, we need to define a new utility; we would like to assign a high value to r-domains that solve the problem instance fast and correctly. Therefore, one simple utility for the task is as follows:

$$U_{(i,P)}(d) = \begin{cases} \neg \text{runtime}_{P(i,d)} & \text{if } P \text{ halts normally and returns a solution plan.} \\ -c & \text{otherwise} \end{cases}$$

Notice that this utility is different from the one we used in Chapter 4: it considers only instances that can be solved (i.e., a solution plan is returned). We consider unsolved instances as timed-out, which will make ParamILS not prefer the r-domain that could not solve those instances.

Using this utility, we applied the same approach described in Chapter 4 to the operator remodeling r-domains in three domains: blocksworld, mprime, and mystery. The setting to the experiment are similar to those described in Chapter 4. The instances and the macros are the same as in this chapter.
Table 5.6: Average runtimes (in seconds) of the r-domains constructed from the manually-constructed macros using the FF planners. The asterisk (*) means that the model was significantly faster than BOA-R with $p \leq 0.05$. Pre is the preprocessing time measured in hours.

The new aspect here is the DIR-R-P predictor, which is the direct predictor that is based on ParamILS evaluation using the utility function described above. The results are shown in Table 5.6.

Table 5.6 shows the average runtime in seconds and the number of incorrectly solved instances for each of the following r-domains and predictors: ORIG, BOA-M, BOA-R, DIR-R, and DIR-R-P. It also shows the preprocessing time in hours.

The table shows that the average runtime as well as the coverage of the new predictor is harmed. In mystery-5, the predictor is not significantly faster than DIR-R, BOA-M or BOA-R. In mprime-5, the predictor is significantly worse than DIR-R, BOA-M and BOA-R. However, in the blocksworld domain the result is better, DIR-R-P was significantly faster than BOA-M and BOA-R. The ratio of correctness is still acceptable in all tested domains ($\geq 0.98$). The preprocessing time is significantly less using the ParamILS method, as compared to the exhaustive method, in all domains.

Perhaps the reason for the difference in performance between DIR-R and DIR-R-P is the increase in the number of r-domains to choose from. In DIR-R, we choose from the accepted r-domains only ($\gamma^I_{d,train} \geq 0.98$). In DIR-R-P, we choose from all possible r-domains.

A possible solution to this utility problem would involve including a measure of the r-domain coverage in the utility. This measure can represent the coverage of the r-domain on a number of small random instances.

5.9 Summary

In this chapter we combined two remodeling approaches and applied the new remodeling in an instance-specific framework. We have shown that the operator-based remodeling can significantly improve the planning speed in a number of domains over the macro-only remodeling. We have also shown that the instance-specific predictors of this remodeling can significantly outperform the fixed r-domain. We show that the average empirical coverage of the new reformulations is acceptable.

We used the Torchlight tool to analyze the topology of the search using operator-based remodeling. We found that in the one easy domain, blocksworld, the performance is significantly related to the change in the search topology. In the other domains, we found that the change in the search topology was not directly related to the performance of the r-domains.

Finally, we suggested using a parameter optimization approach to evaluate the r-domains for the predictors. We found that this approach requires more thought, since the coverage of the r-domains was not taken into account.
Chapter 6

Conclusions and Future Work

The two main goals of this dissertation were to (1) automate domain remodeling and (2) make it more flexible and applicable. We achieved these goals via the introduction of three ideas. The first goal was achieved by introducing the instance-specific remodeling framework that relies on machine learning. Under that framework, remodeling was automated by making it a part of solving the problem instance, bringing the process closer to the system suggested by Simon [80].

The second goal was achieved by reducing remodeling to a parameter optimization problem and by combining orthogonal remodeling approaches. The reduction introduced flexibility to the domain remodeling; for a remodeling type, we cast its components as parameters that can be combined with other domain parameters. This reduction also achieved increased applicability, since it allows us to use any generic parameter optimization tool to do the remodeling. Combining orthogonal remodeling approaches is expected to solve more instances, and hence it improves remodeling flexibility.

In this dissertation, we implemented these general ideas, illustrated in Figure 1.1 of Chapter 1, by introducing three novel approaches. We first presented a planner-independent, instance-specific domain remodeling approach that uses macro operators and machine learning. Second, we introduced an approach to recast the problem of domain remodeling as parameter optimization. Third, we introduced a new domain remodeling approach that depends on macro addition and operator removal in instance-specific and fixed contexts, and that takes solubility-preserving into account. Our empirical results over a number of standard benchmark planning domains showed that our approaches can outperform the state-of-the-art tools as well as traditional remodeling methods that rely on fixed domain-specific remodeling. In analyzing these results, we showed that the remodeling process is also more flexible: remodeling is done by the recasting of the domain component, and we can, for example, achieve a new remodeling by combining existing remodelings. We can also extend this remodeling-as-optimization approach by simultaneously tuning the planner configurations and the domain components.

In this chapter, we restate our contributions in light of the analysis and discussion of previous chapters, we present our future work, and we finally conclude.

6.1 Contributions

In this section we present the major contributions of this work in light of the results and analysis; we also mention other contributions that are not directly related to the thesis of this dissertation.
6.1.1 Major Contributions

1. **Introducing An Instance-Specific Framework** – We introduced a novel learning approach for choosing macro operators for remodeling planning domains for each problem instance. We have shown that it can increase planning speed significantly compared to the fixed macros scheme. By analyzing the results on the prediction model, we found a strong correlation between the extracted features and the macros, and we found that the predictor used that correlation to make accurate predictions. We also analyzed the search topology on the remodeled domain, and found that even though our approach can improve the search, it did not improve the search topology, and we suggested that including topology analysis in the utility may yield better results. Based on the search topology analysis, we also explained the observation that the approach is less effective in the FF planner and a class of ‘easy’ domains.

2. **Reducing Domain Remodeling To Parameter Optimization** – We introduced a novel way to learn macros by casting them as a parameters to the planner and choosing them using a parameter tuning tool. In addition to enabling us to use advanced parameter tuning tools for remodeling, this approach approximates our exhaustive approach of evaluating the instance-specific macros. We show that the approximate approach is still as effective as the exhaustive approach in finding instances-specific macros while requiring significantly less preprocessing time. We also show that it is as effective as the state-of-the-art macro-based remodeling tools. We explained that this approach opens the door to combining different kinds of remodeling that can be achieved by casting the components of each kind of remodeling as different parameters.

3. **Combining Remodeling Approaches** – We similarly introduced a new form of remodeling to the planning domains by removing basic operators as well as adding macros in an instance-specific context, and we have shown that this can improve the planning speed significantly as compared to a fixed macros scheme. Applying topology analysis, we also found that the search topology worsens with the remodeling, which, again, suggests that it should be considered in the utility. We also showed that the probability of preserving solubility on the problem instances using this remodeling was controllable, and that, empirically, this probability is high in the direct predictors.

6.1.2 Other Contributions

1. **Feature Selector for Remodeling** – To select features for the purpose of instance-specific remodeling, we built a feature selector based on the heuristic-learning feature selector presented in [90]. The modified feature selector presented in Section 3.3.4 filters the features based on their relation to the runtime of the planner on the reformulated domains and the training instances.

2. **Pipesworld Domain Generator** – In the process of experimenting, we wrote a generator of the pipesworld domain. This generator is described in detail in Appendix C. This generator can be used to generate random pipesworld domain instances.

6.2 Future Work

Throughout the discussions of this dissertation, we presented a number of potentially useful directions in which to pursue research. We present a summary here.
6.2.1 Individual Macro Learning As Parameter Optimization

In Section 4.8, we proposed to use parameter optimization to generate individual macros directly, as opposed to filtering them. This approach generates fixed-length, minimally-useful macros.

We limit the number of operators in a macro to a fixed number. Therefore, we are effectively choosing one macro at a time from a large but finite set of macros. To ensure that the macro is useful, we force each operator to be linked to the preceding sequence of operators in the macro by making one of its preconditions to be satisfied by one the positive effects added by the preceding sequence. In doing so, we prevent the construction of macros that contain unlinked operators. In the parameter optimization tool, the operators’ numbers and the linked atoms’ numbers are used as the parameters that define the macro.

To generate a useful macro, different macros with different parameter values should be tried. We could generate a macro according to the set of parameters, and then run the planner with the macro on a set of training instances. The macro with the shortest runtime on the instances should have highest utility. However, invalid macros can appear as a result of invalid substitutions made by the planner. Therefore, a validator should also be used on each instance to check the validity of the resulting plans. With that in place, invalid plans should also lower the macro’s utility measure. Finally, the macros with the highest utility are returned.

6.2.2 Feature Selection For Clustering

In Section 3.7, we discussed how macros that are not specifically generated for an instance-specific use can be harmful to the performance. In Section 4.8, we suggested an approach that uses ParamILS to generate the macros independently. Using that approach, we ran ParamILS for each cluster of instances to find instance-specific macros.

One problem with this approach is the difficulty in choosing the right set of features with which to cluster the instances. Even though we generate macros for each cluster using a separate call to a macro tool, we still cannot guarantee that the predictor will be able to use the cluster’s features to successfully choose the macros, and we do not know whether or not the macros created will be significantly better than the best-on-average macro set. Therefore, we need the features to divide the instances according to instance-specific macro performance, and we need the best macros of the clusters to be significantly faster than the best-on-average macro set for all instances. However, satisfying both ease of classification and superiority of performance is not easy.

In Section 4.8.2, we suggested an approach that iteratively chooses features and generates macros. The algorithm should begin with an empty set of features, greedily selecting one feature at a time to cluster the training instances. For each cluster, we use a macro tool to generate macros that perform best on that cluster. Using a set of small test instances, we evaluate the new feature-macros pair according to (1) their instance-specific performance, and (2) ease of classification. We then repeat the process using new features.

By quantifying the satisfaction of the two conditions, we try to enforce the choice of useful features-macros combinations. However, such an approach is time consuming and requires a good macro generation tool.
6.2.3 Remodeling And Planner Tuning As Parameter Optimization

The domain parameters discussed in Chapter 4 can be incorporated into a larger scheme of parameter optimization. Formalizing the planner and the domain model as one unit allows for highly flexible system and incorporates the domain design into the process of solving. For example, we can think of choosing the search algorithm and/or the heuristic as parameters of the planner that can be changed. Tuning these planner parameters simultaneously with the domain model parameters may improve the planning speed significantly.

A quick and simple planner parameter can specify which planner to use from a set of planners. Systems such as PbP [37] have incorporated macro learning with choosing planners in a domain-specific framework. In each domain, PbP selects a portfolio of planners and macro operators and uses them to solve future instances of the domain. However, PbP is only a domain-specific preprocessor; it is not instance-specific.

In the instance-specific context, not only can we choose the reformulated domain that suits each problem instance, but we can also choose the combination of a planner and domain that works best for that specific instance. This is motivated by the fact that while some planners are better than other planners and some reformulated domains are better than other reformulated domains in some cases, it is not clear whether instances within the same domain can be solved with different planners-domain combinations. In the work of Roberts et al., for example, it was shown that the domain features are strongly related to the planner choice [76].

6.2.4 Operator-Remodeling As Parameter Optimization

Replacing the exhaustive evaluation in the operator remodeling approach with parameter optimization, while taking solubility-preserving into account, may lead to a better remodeling. In this case, the parameters of the domain model are binary variables that represent both the macros to be added and the domain’s original operators. The motivation here is to reduce the preprocessing time, and to extend the flexibility of macro-based remodeling by removing operators. We have shown in Chapter 4 that using a parameter optimization tool for the remodeling significantly reduces the preprocessing time of instance-specific macro-based remodeling, while not significantly harming the predictors’ performance.

We have shown some preliminary results from a version of this approach in Section 5.8. We treated the reformulated domains that return “no plan” for the problem instances as if they had timed out; this is unlike the case in Chapter 3, where we just registered the runtime of such domains. We used ParamILS to find instance-specific reformulated domains. The results of this preliminary experiment show that the average runtime of the new direct predictor was generally worse than that resulting from the exhaustive evaluation. The preprocessing time was significantly faster, however. Given there is no filtering of the reformulated domains before learning, one reason for this poor performance is probably the large number of reformulated domains that the direct predictor had to choose from. In the exhaustive approach, as described in Chapter 5, we filter the reformulated domains based on their coverage ratio before performing the learning, which results in a relatively smaller set of domains from which to choose. In ParamILS evaluation, it is not feasible to achieve such a filtering step since it is based on finding the solubility of all instances on all reformulated domains, which is what we are trying to avoid.

To solve this training problem, we suggest using a new utility function that includes a measure of coverage. Given a set of small instances, the coverage can be measured by the ratio of instances
solved correctly to the total number of solved instances. The rest of this approach is as described in Section 5.8.2: we cast the operators and macros as parameters, we cluster the instances, and we use a parameter optimization tool on each cluster with the new utility function to evaluate the domains.

If we can maintain the accuracy of the predictors in this approach, we may improve the planning process significantly. Given the performance of the operator remodeling described in Chapter 5, this approach may enrich planning significantly while requiring relatively little preprocessing.

6.2.5 A Portfolio Of Reformulated Domains

Instead of using one predictor, we can, for example, use the time and direct predictors simultaneously in a portfolio approach [56] to solve the problem instances. In a portfolio, each of the chosen reformulated domains is given a slice of time to execute on the problem instance in a round-robin fashion.

Combining solving approaches using portfolios can be useful [56]. A portfolios of predictors may be especially useful in some hard domains in which the problem instances either are solved in a fraction of a second, using the r-domain, or time out. However, this approach requires that the solving techniques are orthogonal, meaning they should each work well on different kinds of instances. If the techniques are orthogonal, we may benefit from combining them into a portfolio that uses both of them. But if they both solve the same or similar sets of instances, we will not benefit from combining them.

6.3 Conclusion

The thesis of this dissertation is that automating domain remodeling in planning by incorporating the remodeling into solving, and that making remodeling more flexible and applicable, both enriches planning and improves the planning speed. We developed three approaches to achieve that goal.

In Chapter 3, we built a instance-specific planning framework in which dynamic domain remodeling accompanies the solving of each problem. The approach used only one type of remodeling: macro addition. This approach opens the door for a new kind of planning that views remodeling as part of the solving process and keeps remodeling independent from the planner used. We found that this framework – just by using a simple remodeling approach and general and simple features – can improve planning speed significantly, as compared to traditional macro-addition remodeling, in a number of domains and using state-of-the-art planners.

In Chapter 4, we built a system that achieves remodeling by recasting the problem of evaluating the domain model components (the macros) as a parameter optimization problem, and then by using a state-of-the-art parameter tuning tool, ParamILS, to perform the optimization. We show that this reduction casts the remodeling as a general optimization problem, which can be extended to include other kinds of optimization, e.g. planner optimization, or other kinds of remodeling. One of the benefits we found when we applied this flexible remodeling approach in an instance-specific framework is that it reduces the preprocessing time significantly while not harming performance, as compared to an exhaustive macro evaluation approach.

In Chapter 5 we presented a more general instance-specific remodeling approach that combines macro addition and removing original operators while controlling the lack of solubility-preserving that accompanies operator removal. This approach is another step toward the combination of solving and remodeling. It enriches planning by combining two remodeling approaches and overcoming the lack of
solubility-preserving side effect. We show that we can also combine these approaches when using the re-
duction method presented in Chapter 4. This approach significantly improves the planning performance
compared to the best-on-average remodeling.

Thus, the results and the analysis show that planning was made more flexible by these approaches,
and that the performance of planning was improved significantly compared to traditional and state-of-
the-art tools.
Bibliography


Appendix A

Domains Used

Hoffmann [44] has categorized some of the benchmark planning domains into classes based on their
search topology using the $h+$ heuristic. In our experiments, we specifically chose the domains to cover
a spectrum of these classes. Specifically, we chose from four domain types:

1. without local minima, without dead-ends
2. with local minima, without dead-ends
3. with dead-ends recognizable by $h+$
4. with dead-ends unrecognizable by $h+$

In the following we describe each domain we used. The full PDDL definition of each domain, can be
found in the following:


A.1 Logistics

The logistics domain describes an environment where trucks and airplanes transfer packages to different
locations within cities. Its operators are: load-truck, load-airplane, unload-truck, unload-airplane,
drive-truck, and fly-airplane. The load-truck operator loads a truck with an object that is
at the same location. The load-airplane operator loads an airplane with and object at an air-
port. unload-truck and unload-airplane unload the loaded truck and airplane respectively. The
drive-truck moves a truck from one location to another within the same city, while fly-airplane
moves an airplane from one airport location in a city to another airport in another city.

The logistics domain belongs to the first domain type: it has no dead-ends for the FF planner, and
it has no local minima with respect to the FF planner’s heuristic. Moreover, the logistics domain search
space is directed. A directed search space is the space in which every action can be immediately reversed.

A.2 Blocksworld (4 operators)

The blocksworld domain represents an environment where there are a number of blocks $n$, a table, and a
robot arm. The block can be on another block or on the table, and the arm can hold one block at a time.
Appendix A. Domains Used

When there is nothing on the block, the block is considered clear. In the initial state, the $n$ blocks are put arbitrarily on each other and on the table. Each block can have at most one block on it and can be on at most one block. The goal is represented similarly: we need the blocks to be put on each other and on the table according to another arbitrary configuration. The allowed operators are: pickup, putdown, unstack, and stack. The pickup operator makes the empty arm pick a clear block that is on the table, the putdown operator makes the arm put the held block on the table, the unstack operator makes the empty arm pick a clear block that is on another block, and the stack operator makes the arm put the held block on another block.

The Blocksworld domain is of the second domain type: it contains local minima. The cause of the local minima is that the heuristic ignores the delete effects of the actions, leading to a state with a heuristic value lower than its neighbors. Although blocksworld’s search space is not easy for FF to navigate, it is a directed search space.

A.3 Pipesworld (no tankage)

The pipesworld is, relatively, a newer domain compared to the previous ones. It first appeared in fourth international planning competition (IPC-4). What is required in this domain is to control the flow of oil derivatives through a pipeline network, obeying various constraints, such as product ordering compatibility, and pipes first-in-first-out restrictions. There are different types of products to be transported, and some are not compatible with the others. The interesting aspect of this domain, according to the planning competition organizers, is that, if one inserts something into one end of a pipeline segment, something completely different might come out at the other end, which makes planning harder\(^1\). The network’s structure and pipes connections sizes are fixed for the planning competition problem instances, except that in smaller instances only a subgraph of the original pipes connection graph is used.

There are locations where the batches of oil derivative are placed. The locations are connected by pipes, that can be of two types: normal and unitary. The unitary pipes contain one batch at a time. A normal pipe contains more batches, and every normal pipe has a size. There are five types of batches, represented as constants, some of which are not allowed to be next to each other in a pipe. The operators are: push-start, push-end, pop-start, pop-end, push-unitarypipe, and pop-unitarypipe. The push-start pushes one batch of material from the source location of a normal pipe into the pipe if the first batch in the pipe was compatible with the new batch. The push-end must follow push-start because it takes the last batch of the pipe on which push-start was applied and puts it in the destination location. pop-start and pop-end work similarly but with exchanging the destination and source locations. The operator push-unitarypipe is similar to push-start except that it works only on unitary pipes, and it moves the old batch to the destination immediately, i.e., it does not need an end action. The operator pop-unitarypipe is similar to push-unitarypipe except that it exchanges the source and destination.

The pipesworld domain also contains recognized dead-ends for $h^+$, so it is from the third type of domains. This does not mean, however, that it does not cause dead-ends in FF’s search. In fact, FF could not solve a number of pipesworld instances using its EHC phase due to dead-ends, and switched to the BFS phase.

\(^1\)See \url{http://www.tzi.de/~edelkamp/ipc-4/domains.html}.\)
Appendix A. Domains Used

A.4 Mystery

The mystery domain is a harder version of the logistics domain, where fuel is required to move trucks, and limited fuel and truck sizes are available. The airports and airplanes are removed from the definition, and the locations are connected in a ring shape. The allowed operators in mystery are: move, load, and unload. The operator move makes a truck move from one location 'a' to another connected location 'b' if location 'a' has fuel. Notice that the fuel is associated with the location not the truck. The operator load transfers a cargo from a location into the truck that is at the same location if the truck has some space left. The operator unload transfers a cargo from a truck to the location of the truck.

The mystery domain contains unrecognized dead-ends for the h+ heuristic and for the FF planner, so it is from the fourth type of domains. This is mainly caused by the addition of the limited, consumable fuel resource.

A.5 Mprime

The mprime domain is a modified version of the mystery domain, where fuel is also required to move trucks, and limited fuel and truck sizes are added, but the fuel can be transferred between unconnected locations. The allowed operators in mprime are: move, load, unload, and donate. The operators move, load, and unload are the same operators in the mystery domain. The operator donate, however, immediately transfers one fuel unit from one location ‘a’, that contains more than one fuel unit, to another location ‘b’, reducing the fuel level of ‘a’ by one and increasing the fuel level of ‘b’ by one. The two location do not need to be connected, and the truck is needed for the transfer.

The mprime domain contains unrecognized dead-ends for h+ and the FF planner, so it is also from the fourth type of domains.

A.6 Freecell

The freecell domain is a more general version of the Freecell card game, represented in the STRIPS language. It allows for a different number of cards, suits, free cells, and columns. There are a number of suits and a number of cards, each card belongs to a suit and it has a unique order in that suit. The cards are distributed in columns, where they are stacked. Only stackable cards can be stacked on each other in columns. There are free cells where each can hold one card, and there are home cells, where each set of cards from the same suit home cell can be held in order. The goal, in general, is to move all cards from the columns to the home cells.

The domain has 10 operators: move, move-b, sendtofree, sendtofree-b, sendtonewcol, sendtohome, sendtohome-b, homefromfreecell, colfromfreecell, newcolfromfreecell. The operators move and move-b move card between columns. The move-b version generates a new stack as a result of the move. The operators sendtofree and sendtofree-b send a card from the top of a column to a free cell. Similarly, the two versions depend on whether or not a new stack is generated. The operator sendtonewcol sends a card from the top of a column to a new (empty) column. The operators sendtohome and sendtohome-b send a card to home cell similarly. The operator homefromfreecell sends a card to a home cell from a free cell, and each of colfromfreecell and newcolfromfreecell sends a card to a column from a free cell similarly, where newcolfromfreecell generates a new column.
The freecell domain also contains unrecognized dead-ends for h+ and the FF planner, so it is from the fourth type of domains. The dead-ends are mainly caused by the ordering restrictions in the columns and the limited number of free cells.
Appendix B

Manual Macro Operators

We manually wrote some sound macros to be added to the domains. Sound macro operators are sequences of operators whose preconditions and effects represent the real preconditions and effects of the composite application of the operators. The macros of each domain are listed in the following sections.

B.1 Logistics

We used two macros in the logistics domains:

unload-airplane-load-truck: It is the operators unload-airplane and load-truck acting on the same object.

load-truck-drive-truck-unload-truck: It is the sequence of the three operators load-truck, drive-truck and unload-truck acting on the same object.

B.1.1 PDDL Definition

(:action unload-airplane-load-truck
  :parameters (?p ?a ?l1 ?t)
  :precondition
  (and
   (obj ?p)
   (airplane ?a)
   (location ?l1)
   (in ?p ?a)
   (at ?a ?l1)
   (at ?t ?l1)
   (truck ?t)
  )
  :effect
  (and
   (not (in ?p ?a))
   (in ?p ?t)
  )


B.2 Blocksworld (4-operators)

**pickup-stack** : The two operators pickup and stack acting on the same block.

**unstack-putdown** : The two operators unstack and putdown acting on the same block.

**unstack-stack** : The two operators unstack and stack acting on the same block.

**unstack-putdown-unstack-putdown** : The macro operator unstack-putdown applied twice to the two top blocks of a stack putting them on the table.

### B.2.1 PDDL Definition

( :action pickup_stack
   :parameters (?ob ?underob)
   :precondition (and (clear ?ob) (on-table ?ob) (arm-empty) (clear ?underob))
   :effect (and (not (clear ?underob)) (not (on-table ?ob)) (on ?ob ?underob)
             (clear ?ob)))
Appendix B. Manual Macro Operators

118

(:action unstack_putdown
  :parameters (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
  :effect (and (on-table ?ob) (clear ?underob) (not (on ?ob ?underob))))

(:action unstack_stack
  :parameters (?ob1 ?ob2 ?ob3)
  :precondition (and (on ?ob1 ?ob2) (clear ?ob1) (arm-empty) (clear ?ob3))
  :effect (and (clear ?ob2) (not (on ?ob1 ?ob2)) (not (clear ?ob3)) (on ?ob1 ?ob3) (clear ?ob1))

(:action unstack_putdown_unstack_putdown
  :parameters (?ob1 ?ob2 ?ob3)

B.3 Pipesworld (no-tankage)

push-start-push-end : The two operators push-start and push-end acting on the same normal pipe.

pop-start-pop-end : The two operators pop-start and pop-end acting on the same pipe.

push-unitarypipe-pop-unitarypipe : The operator push-unitarypipe followed by pop-unitarypipe on the same unitary pipe.

push-start-push-end-2 : The two operators push-start and push-end acting on the same two-segments pipe.

B.3.1 PDDL Definition

(:action push-start-push-end
  :parameters(
    ?pipe - pipe
    ?batch-atom-in - batch-atom
    ?from-area - area
    ?to-area - area
    ?first-batch-atom - batch-atom
    ?product-batch-atom-in - product
    ?product-first-batch - product
    ?last-batch-atom - batch-atom
    ?next-last-batch-atom - batch-atom
  )
  :precondition
(and
    (normal ?pipe)
    (first ?first-batch-atom ?pipe)
    (connect ?from-area ?to-area ?pipe)
    (on ?batch-atom-in ?from-area)
    (not-unitary ?pipe)
    (is-product ?first-batch-atom ?product-first-batch)
    (last ?last-batch-atom ?pipe)
    (follow ?last-batch-atom ?next-last-batch-atom)
)
:effect
(and
    (first ?batch-atom-in ?pipe)
    (not (first ?first-batch-atom ?pipe))
    (follow ?first-batch-atom ?batch-atom-in)
    (not (on ?batch-atom-in ?from-area))
    (not (push-updating ?pipe))
    (not (follow ?last-batch-atom ?next-last-batch-atom))
    (last ?next-last-batch-atom ?pipe)
    (not (last ?last-batch-atom ?pipe))
    (on ?last-batch-atom ?to-area)
)
)

(:action pop-start-pop-end
 :parameters(
   ?pipe - pipe
   ?batch-atom-in - batch-atom
   ?from-area - area
   ?to-area - area
   ?last-batch-atom - batch-atom
   ?product-batch-atom-in - product
   ?product-last-batch - product
   ?first-batch-atom - batch-atom
   ?next-first-batch-atom - batch-atom
)
 :precondition
  (and
   (normal ?pipe)
   (last ?last-batch-atom ?pipe)
   (connect ?from-area ?to-area ?pipe)


(on ?batch-atom-in ?to-area)
(not-unitary ?pipe)
(is-product ?last-batch-atom ?product-last-batch)
(first ?first-batch-atom ?pipe)
(follow ?next-first-batch-atom ?first-batch-atom)
)
:effect
(and
(last ?batch-atom-in ?pipe)
(not (last ?last-batch-atom ?pipe))
(follow ?batch-atom-in ?last-batch-atom)
(not (on ?batch-atom-in ?to-area))
(not (pop-updating ?pipe))
(not (follow ?next-first-batch-atom ?first-batch-atom))
(first ?next-first-batch-atom ?pipe)
(not (first ?first-batch-atom ?pipe))
(on ?first-batch-atom ?from-area)
)
)

(:action push-unitarypipe-pop-unitarypipe
:parameters(
  ?pipe - pipe
  ?batch-atom-in-1 - batch-atom
  ?from-area - area
  ?to-area - area
  ?first-batch-atom - batch-atom
  ?product-batch-atom-in-1 - product
  ?product-first-batch - product
  ?batch-atom-in-2 - batch-atom
)
:precondition
(and
  (first ?first-batch-atom ?pipe)
  (connect ?from-area ?to-area ?pipe)
  (on ?batch-atom-in-1 ?from-area)
  (unitary ?pipe)
  (is-product ?first-batch-atom ?product-first-batch)
)
(on ?batch-atom-in-2 ?to-area)
)
:effect
(and
(not (first ?first-batch-atom ?pipe))
(not (last ?first-batch-atom ?pipe))
(on ?first-batch-atom ?to-area)
(last ?batch-atom-in-2 ?pipe)
(not (last ?batch-atom-in-1 ?pipe))
(not (first ?batch-atom-in-1 ?pipe))
(not (on ?batch-atom-in-2 ?to-area))
(on ?batch-atom-in-1 ?from-area)
)
)

(:action push-start-push-end-2
 :parameters(
   ?pipe - pipe
   ?batch-atom-in - batch-atom
   ?from-area - area
   ?to-area - area
   ?first-batch-atom - batch-atom
   ?product-batch-atom-in - product
   ?product-first-batch - product
   ?last-batch-atom - batch-atom
 )
 :precondition
 (and
 (normal ?pipe)
 (first ?first-batch-atom ?pipe)
 (connect ?from-area ?to-area ?pipe)
 (on ?batch-atom-in ?from-area)
 (not-unitary ?pipe)
 (is-product ?first-batch-atom ?product-first-batch)
 (last ?last-batch-atom ?pipe)
 (follow ?last-batch-atom ?first-batch-atom)
 )
 :effect
Appendix B. Manual Macro Operators

B.4 Mystery

load-move-unload: The three operators load, move, and unload applied on the same package by the same truck moving between two connected locations.

load-move: The two operators load and move applied on the package located in the same location from which the truck moves.

move-unload: The two operators move and unload applied on the package loaded in the moving truck.

B.4.1 PDDL Definition

(:action load_move_unload
   :parameters (?c - cargo ?v - vehicle ?l1 ?l2 - location ?s1
               ?s2 - space ?f1 ?f2 - fuel)
   :precondition (and (at ?c ?l1)
                 (at ?v ?l1)
                 (has-space ?v ?s1)
                 (space-neighbor ?s2 ?s1)
                 (conn ?l1 ?l2)
                 (has-fuel ?l1 ?f1)
                 (fuel-neighbor ?f2 ?f1))
   :effect (and (not (at ?c ?l1))
                (at ?c ?l2)
                (not (at ?v ?l1))
                (at ?v ?l2)
                (not (has-fuel ?l1 ?f1))
                (has-fuel ?l1 ?f2))
)

(:action load_move
   :parameters (?c - cargo ?v - vehicle ?l1 ?l2 - location ?s1
               ?s2 - space ?f1 ?f2 - fuel)
   :precondition (and (at ?c ?l1)
                 (at ?v ?l1)
                 (has-space ?v ?s1)
                 (space-neighbor ?s2 ?s1)
                 (conn ?l1 ?l2)
                 (has-fuel ?l1 ?f1)
                 (fuel-neighbor ?f2 ?f1))
   :effect (and (not (at ?c ?l1))
                (at ?c ?l2)
                (not (at ?v ?l1))
                (at ?v ?l2)
                (not (has-fuel ?l1 ?f1))
                (has-fuel ?l1 ?f2))
)
Appendix B. Manual Macro Operators

\( ?s2 - \text{space} \) \(?f1 \) \(?f2 - \text{fuel} \)

:precondition (and (at ?c ?l1)
    (at ?v ?l1)
    (has-space ?v ?s1)
    (space-neighbor ?s2 ?s1)
    (conn ?l1 ?l2)
    (has-fuel ?l1 ?f1)
    (fuel-neighbor ?f2 ?f1))

:effect (and (not (at ?c ?l1))
    (in ?c ?v)
    (not (has-space ?v ?s1))
    (has-space ?v ?s2)
    (not (at ?v ?l1))
    (at ?v ?l2)
    (not (has-fuel ?l1 ?f1))
    (has-fuel ?l1 ?f2))

):action move_unload

:parameters (?v - \text{vehicle} \) ?l1 ?l2 - \text{location} \) ?f1 \) ?f2 - \text{fuel} \) ?c - \text{cargo} \) ?s1 ?s2 - \text{space} \)

:precondition (and (at ?v ?l1)
    (conn ?l1 ?l2)
    (has-fuel ?l1 ?f1)
    (fuel-neighbor ?f2 ?f1)
    (in ?c ?v)
    (has-space ?v ?s1)
    (space-neighbor ?s1 ?s2))

:effect (and (not (at ?v ?l1))
    (at ?v ?l2)
    (not (has-fuel ?l1 ?f1))
    (has-fuel ?l1 ?f2)
    (not (in ?c ?v))
    (at ?c ?l2)
    (not (has-space ?v ?s1))
    (has-space ?v ?s2))

)

B.5 Mprime

\text{load-move-unload :} The three operators load, move, and unload applied on the same package by the same truck moving between two connected locations.

\text{load-move :} The two operators load and move applied on the package located in the same location
from which the truck moves.

**move-unload**: The two operators *move* and *unload* applied on the package loaded in the moving truck.

**donate-move**: The two operators *donate* and *move* applied on the same location from which the truck moves.

### B.5.1 PDDL Definition

```
(:action load_move_unload
  :precondition (and (at ?c ?l1)
    (at ?v ?l1)
    (has-space ?v ?s1)
    (space-neighbor ?s2 ?s1)
    (conn ?l1 ?l2)
    (has-fuel ?l1 ?f1)
    (fuel-neighbor ?f2 ?f1))
  :effect (and (not (at ?c ?l1))
    (at ?c ?l2)
    (not (at ?v ?l1))
    (at ?v ?l2)
    (not (has-fuel ?l1 ?f1))
    (has-fuel ?l1 ?f2)))
)

(:action load_move
  :precondition (and (at ?c ?l1)
    (at ?v ?l1)
    (has-space ?v ?s1)
    (space-neighbor ?s2 ?s1)
    (conn ?l1 ?l2)
    (has-fuel ?l1 ?f1)
    (fuel-neighbor ?f2 ?f1))
  :effect (and (not (at ?c ?l1))
    (in ?c ?v)
    (not (has-space ?v ?s1))
    (has-space ?v ?s2)
    (not (at ?v ?l1))
    (at ?v ?l2)
    (not (has-fuel ?l1 ?f1))
    (has-fuel ?l1 ?f2)))
)
Appendix B. Manual Macro Operators

(:action move_unload
  :parameters (?v - vehicle ?l1 ?l2 - location ?f1 ?f2 - fuel
  ?c - cargo ?s1 ?s2 - space)
  :precondition (and (at ?v ?l1)
    (conn ?l1 ?l2)
    (has-fuel ?l1 ?f1)
    (fuel-neighbor ?f2 ?f1)
    (in ?c ?v)
    (has-space ?v ?s1)
    (space-neighbor ?s1 ?s2))
  :effect (and (not (at ?v ?l1))
    (at ?v ?l2)
    (not (has-fuel ?l1 ?f1))
    (has-fuel ?l1 ?f2)
    (not (in ?c ?v))
    (at ?c ?l2)
    (not (has-space ?v ?s1))
    (has-space ?v ?s2))
)

(:action donate_move
  ?v - vehicle )
  :precondition (and (not-equal ?l1 ?l2)
    (has-fuel ?l1 ?f11)
    (fuel-neighbor ?f12 ?f11)
    (fuel-neighbor ?f13 ?f12)
    (has-fuel ?l2 ?f21)
    (at ?v ?l2)
    (conn ?l2 ?l3))
  :effect (and (not (has-fuel ?l1 ?f11))
    (has-fuel ?l1 ?f12)
    (not (at ?v ?l2))
    (at ?v ?l3))
)

B.6 Freecell

sendtofree-sendtohome: The two operators sendtofree and sendtohome acting on the same card.

sendtofree-move-colfromfreecell: The three operators sendtofree, move, and colfromfreecell acting on the a stack.
sendtofree-sendtofree: The operator sendtofree applied twice on the cards of one stack.

### B.6.1 PDDL Definition

(:action sendtofree_sendtohome
   :parameters (?card1 ?card2 ?card3 - card ?cells ?ncells -
   cellnum ?suit - suit ?vcard - num ?homecard - card
   ?vhomecard - num)
   :precondition (and
      (clear ?card1)
      (on ?card1 ?card2)
      (cellspace ?cells)
      (cellsuccessor ?cells ?ncells)
      (on ?card2 ?card3)
      (home ?homecard)
      (suit ?card2 ?suit)
      (suit ?homecard ?suit)
      (value ?card2 ?vcard)
      (value ?homecard ?vhomecard)
      (successor ?vcard ?vhomecard)
   )
   :effect (and
      (incell ?card1)
      (cellspace ?ncells)
      (not (on ?card1 ?card2))
      (not (clear ?card1))
      (not (cellspace ?cells))
      (home ?card2)
      (clear ?card3)
      (not (on ?card2 ?card3))
      (not (home ?homecard))
   )
)

(:action sendtofree_move_colfromfreecell
   :parameters (?card1 ?card2 ?card3 ?card4 - card ?cells1
   ?cells2 - cellnum)
   :precondition (and
      (clear ?card1)
      (on ?card1 ?card2)
      (cellspace ?cells1)
      (cellsuccessor ?cells1 ?cells2)
      (clear ?card4)
      (canstack ?card2 ?card4)
   )
)
(on ?card2 ?card3)
(canstack ?card1 ?card2)
):

effect (and
(on ?card2 ?card4)
(clear ?card3)
(not (on ?card2 ?card3))
(not (clear ?card4))
(not (incell ?card1))
(not (cellspace ?cells2))
(not (clear ?card2))
)
)

(:action sendtofree_sendtofree
 :parameters (?card1 ?card2 ?card3 - card ?cells1 ?cells2
 ?cells3 - cellnum)
 :precondition (and
 (clear ?card1)
 (on ?card1 ?card2)
 (cellspace ?cells1)
 (cellsuccessor ?cells1 ?cells2)
 (on ?card2 ?card3)
 (cellsuccessor ?cells2 ?cells3)
 )
 :effect (and
 (incell ?card1)
 (not (on ?card1 ?card2))
 (not (clear ?card1))
 (not (cellspace ?cells1))
 (incell ?card2)
 (clear ?card3)
 (cellspace ?cells3)
 (not (on ?card2 ?card3))
 (not (clear ?card2))
 (not (cellspace ?cells2))
 )
)
Appendix C

Generators

We used the problem generators that come with the FF planner \(^1\). However, for the pipesworld (no tankage) domain we wrote our own generator.

C.1 Pipesworld (no-tankage)

We contacted the author of the pipesworld domain, but we were not able to get the problem instance generator for the pipesworld (no tankage) domain. We could not find any other domain generator for this domain. So, we wrote our own generator based on the problem instances provided in the 4th International Planning Competition (IPC-4) \(^2\). The problem instances in the domain competition have a fixed network structures with fixed pipes sizes, and the problem instance is a subgraph of the fixed network graph.

The domain generator’s parameters are:

- **# pipes** is the number of pipes (the number of edges in the graph). It also determines the number of locations (nodes in the graph) since the graph’s connectivity is fixed to follow that of IPC-4 problems.

- **# batches** is the number of batches in the problem. It must be greater than or equal to the number of segments in all pipes plus the maximum pipe size. The number of segments of the pipes is also fixed in the IPC-4 problems. Adding the maximum pipe size is needed to make the problem solvable, since moving a batch from one location to another requires pushing a number of segments that is equal to the size of the pipe. All of the IPC-4 problems satisfy this condition.

- **# goals** is the number of goals: where a batch should be in the goal state. It must be less than or equal to the number of batches not in pipes, since the batches in pipes cannot be taken out except with other batches.

C.1.1 Pipes

The pipes are distributed according to a fixed graph structure based on the IPC instances. The IPC instances have five locations and five pipes. An instance’s pipes network is represent as a prefix of the

\(^1\)http://www.loria.fr/~hoffmann/ff-domains.html on October 26, 2007
following pipes:

\[(1, 2, 2), (1, 3, 2), (3, 4, 1), (2, 3, 3), (1, 5, 4), (2, 6, 3), (4, 6, 1)\]

where \((i, j, k)\) is a pipe/edge in graph originating from location \(i\), ending at location \(j\), and has a size \(k\).

C.1.2 Batches

The batches are distributed randomly to fill out the pipes first, taking under consideration the ordering constraints of the batches. The ordering constraints are based on the batches type, which are constants as defined in Appendix A. The remaining batches are distributed randomly at the location.

C.1.3 Goals

The goal batches are randomly assigned using uniform distribution.

C.1.4 Code

We use two Ruby executables, one calls the other. Both are needed to generate an instance. See:

http://www.cs.toronto.edu/~maher/thesis_files/pipesworld_nt/pipesworld_nt.rb

and


C.2 Other Domains

For all other domains, we used the problem generator from the domain collection page under the FF planner's web page. We downloaded them directly from:

http://www.loria.fr/~hoffmann/ff-domains.html

on 26/Oct/2007. The description of the generators’ parameters is also found in the link. We later found that the FF website, with the domains description and generators, was moved to:

http://fai.cs.uni-saarland.de/hoffmann/ff.html
Appendix D

Features Used

In the following we describe the features selected for the experiments. There are two kinds of features: domain-specific and domain-independent. The domain-specific are only in the freecell domain. The domain-independent are the taxonomic syntax and the simple facts statistics.

D.1 Freecell Domain Features

In the freecell domain we use a domain-specific feature extractor to find features specific to freecell. These features are not general, i.e., they are not extractable from any domain instance, since they are dependent on the number of objects in the problem instance. Therefore, we had to choose maximum numbers of objects in the instance. We chose the maximum number of suits to be 4, the maximum number of cards in each suit to be 13, and the maximum number of columns to be 8. If the feature value does not exist (e.g. in smaller instances) we register a filler value (-1). For each card, there are two features:

**Column feature:** a feature that represents the column number of that card in the initial state. The domain of this feature is $[1..8]^1$.

**Position feature:** a feature that represents the position of that card within its column in the initial state. The domain of this feature is $[1..52]^2$.

So, since the maximum number of cards is 52, there are 104 domain-specific features in the freecell domains.

D.2 State Statistics Features

We use the number of atoms in the initial state and the number of atoms in the goal as features. They are two domain-independent features that are easy to extract.

---

$^1$The maximum is 8 because that is the maximum number of columns.

$^2$The maximum is 52 because that is the maximum number of cards.
D.3 Taxonomic Syntax Features

The taxonomic syntax classes are automatically acquired features as described in Chapter 3. In this section, we describe the features selected by the feature selection algorithms in each of Chapter 3 and Chapter 5. As described in Section 3.3.4, the algorithm selects a number of classes in each level, starting from level 0. The features are the number of objects in each class: there is exactly one feature for each selected class. These classes have different types and descriptions:

**everything:** It is a unique class that represents all objects in the problem instance. It is a level-0 feature.

**type:** The types as described in the PDDL language. For example: the objects that are of type `vehicle` in the mprime domain are put into the class (type, VEHICLE). This is a level-0 class.

**i1pre:** The objects that are arguments to a unary predicate in the initial state. For example: (i1pre, CLEAR) is the set of objects that are arguments to the unary predicate `CLEAR` in the initial state. It is a level-0 class.

**g1pre:** The objects of a unary predicate in the goal state. It is also a level-0 class.

**not:** The complement of a class from the previous level. For example: (not, 1) is the set of all objects that are not in “class 1” of the previous level.

**int:** The intersection of the sets of objects of two classes from the previous level. For example: (int, 0, 5) represents the objects that are in both “class 0” and “class 5” of the previous level.

**iNpre:** The relational extension of a non-unary predicate in the initial state with a class from the previous level. The class description begins with the predicate name, then the previous level’s class number, followed by the number of the argument of that class in the predicate as it appears in the initial state, and finally the number of the predicate argument that contains the objects of the new class. For example: (iNpre, CANSTACK, 0, 2, 1) is the set of objects that are in the first arguments (1) of the non-unary predicate `CANSTACK` in the initial state, where the second argument (2) is an object from “class 0” (0) of the previous level. Notice that the class numbering begins from 0, but the argument numbering begins from 1. This class can appear in all levels $l > 0$. Refer to Algorithm 3 for details on how relational extension is used.

**gNpre:** The relational extension of a non-unary predicate in the goal state with a class from the previous level.

D.3.1 Macro-addition Experiments

We used the same set of features to conduct the experiments of Chapter 3 and Chapter 4. We first selected and extracted the features in Chapter 3 experiment, and used the same features in Chapter 4. However, in these experiments, we did not save the seed required to generate the features, and therefore we could not generate the same features using our feature selector tool. A detailed description of the features in taxonomic syntax, also, could not be re-generated, because that requires re-running the feature selector code with the seed and selecting the option that prints a feature description file. However, the feature files themselves are available, and were used to train the predictors as described in...
Table D.1: The selected taxonomic syntax features for the FF-WIZARD remodeling experiment.

<table>
<thead>
<tr>
<th>domain</th>
<th>level</th>
<th>feature classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>logistics</td>
<td>0</td>
<td>{everything} {i1pre, OBJ} {i1pre, TRUCK} {i1pre, LOCATION} {i1pre, AIRPLANE} {i1pre, CITY} {i1pre, AIRPORT}</td>
</tr>
<tr>
<td>blocksworld</td>
<td>0</td>
<td>{everything} {i1pre, CLEAR} {i1pre, ON-TABLE}</td>
</tr>
<tr>
<td>mystery-5</td>
<td>same</td>
<td>as macro-addition experiment. Refer to the experimental details in Section 5.6.1.</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>same</td>
<td>as macro-addition experiment. Refer to the experimental details in Section 5.6.1.</td>
</tr>
<tr>
<td>freecell</td>
<td>0</td>
<td>{everything} {type, CARD} {type, COLNUM} {type, CELLNUM} {type, NUM} {type, SUIT} {i1pre, CLEAR} {i1pre, CELLSPACE} {i1pre, COLSPACE} {i1pre, HOME} {i1pre, BOTTOMCOL} {g1pre, HOME}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>(int, 10, 11) {iNpre, CANSTACK, 6, 1, 2} {iNpre, SUIT, 6, 1, 2} {iNpre, CANSTACK, 10, 1, 2} {iNpre, CANSTACK, 10, 2, 1} {iNpre, VALUE, 10, 1, 2} {iNpre, ON, 11, 2, 1}</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>(iNpre, ON, 0, 2, 1) {iNpre, CANSTACK, 0, 2, 1} {iNpre, VALUE, 1, 1, 2} {iNpre, VALUE, 3, 1, 2} {iNpre, CANSTACK, 4, 2, 1} {iNpre, CANSTACK, 6, 1, 2} {iNpre, VALUE, 6, 1, 2}</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>(int, 0, 4) (int, 0, 5) {iNpre, ON, 4, 2, 1} {iNpre, SUIT, 4, 1, 2} {iNpre, ON, 5, 2, 1} {iNpre, SUCCESSOR, 6, 2, 1}</td>
</tr>
</tbody>
</table>

Table D.2: The selected taxonomic syntax features for the FF-MACROFF remodeling experiment.

<table>
<thead>
<tr>
<th>domain</th>
<th>level</th>
<th>feature classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocksworld</td>
<td>0</td>
<td>{everything} {i1pre, CLEAR} {i1pre, ON-TABLE}</td>
</tr>
<tr>
<td>pipesworld-nt</td>
<td>0</td>
<td>{everything} {type, PIPE} {type, AREA} {type, PRODUCT} {type, BATCH-ATOM} {i1pre, UNITARY} {i1pre, NOT-UNITARY} {i1pre, NORMAL}</td>
</tr>
<tr>
<td>freecell</td>
<td>0</td>
<td>{everything} {type, CARD} {type, COLNUM} {type, CELLNUM} {type, NUM} {type, SUIT} {i1pre, CLEAR} {i1pre, CELLSPACE} {i1pre, COLSPACE} {i1pre, HOME} {i1pre, BOTTOMCOL} {g1pre, HOME}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>(iNpre, ON, 0, 1, 2) {iNpre, CANSTACK, 6, 1, 2} {iNpre, CANSTACK, 6, 2, 1} {iNpre, VALUE, 10, 1, 2} {iNpre, ON, 11, 2, 1}</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>(int, 1, 6) (int, 2, 4) {iNpre, ON, 1, 1, 2} {iNpre, SUIT, 1, 1, 2} {iNpre, VALUE, 2, 1, 2} {iNpre, SUCCESSOR, 3, 2, 1} {iNpre, CANSTACK, 6, 1, 2}</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>(int, 4, 5) {iNpre, CANSTACK, 6, 1, 2} {iNpre, CANSTACK, 0, 2, 1} {iNpre, SUIT, 1, 1, 2} {iNpre, SUIT, 2, 1, 2} {iNpre, ON, 6, 2, 1}</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>(int, 2, 5) (int, 3, 4) {iNpre, ON, 1, 2, 1} {iNpre, CANSTACK, 1, 1, 2} {iNpre, ON, 5, 2, 1} {iNpre, VALUE, 5, 1, 2}</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>(int, 0, 2) {iNpre, SUIT, 2, 1, 2} {iNpre, ON, 3, 2, 1} {iNpre, ON, 4, 2, 1} {iNpre, VALUE, 4, 1, 2} {iNpre, SUCCESSOR, 5, 2, 1}</td>
</tr>
</tbody>
</table>

Table D.3: The selected taxonomic syntax features for the FF-MANUAL remodeling experiment.

the experiments. The training and testing files of the instance-specific experiments, each accompanied with a feature file, can be found in the following page:


D.3.2 Operator Remodeling Experiment

In the operator-removal macro-addition experiment we selected new features using the new feature selection algorithm described in Subsection 5.4.3.

Table D.1, Table D.2, and Table D.3 show the selected feature classes in each of the planner-macros combination of the remodeling experiment.