THE UNIVERSITY OF CALGARY

DoLittle: A Learning Multi–Strategy Planner

by

Hansjörg Baltes

A DISSERTATION

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JUNE, 1996

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a dissertation entitled "DoLittle: A Learning Multi–Strategy Planner" submitted by Hansjörg Baltes in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Supervisor, Dr. Bruce MacDonald Department of Computer Science	Dr. Steven Minton External Examiner
Dr. Lisa Higham Department of Computer Science	Dr. Laurence Turner Department of Electrical Engineering
Dr. Brian Gaines Department of Computer Science	

Date

Abstract

Multi-strategy planning focuses on the selection and combination of different problem solving methods. Since planning is intractable in complex domains, researchers have developed different methods to restrict, restructure, or reorder the search space and to search the new space. These reformulations of the search space are based on assumptions about the domain or other features of the task such as the problem order, plan structure, or subgoal hierarchy. These planners, then, work well in domains where the underlying assumptions are met, and fail otherwise. Furthermore, in complex domains it is possible that only parts of a task can be efficiently solved with a given planning method. But for other parts of the tasks, a different planning strategy may be appropriate. The goal of multi-strategy planning is to alleviate this problem by selecting and combining different problem solving methods on a single problem.

First, planning is seen as search through the space of partial plans. Different planning strategies can be described by the language of partial plans, the set of transformations on partial plans, and the search method.

Secondly, the thesis develops a theory of multi-strategy planning and shows that a multi-strategy planner can exponentially improve performance over a single strategy planner and derives sufficient conditions for this improvement.

Thirdly, the thesis proposes *general operators* (STRIPS operators with added refinements) as a representation for different planning strategies and

shows how general operators can represent different planning methods.

Fourthly, the thesis develops a search control method that, given a planning method expressed as a general operator reduces the associated search space similarly to the original problem solving strategy.

Since the generation of general operators may be cumbersome by hand, and since the system is intended as a part of a learning apprentice system, DoLittle learns new general operators from examples. The planning bias learners are highly specific methods that have knowledge of DoLittle's operator set and search method and create new general operators to exploit a given planning bias.

Through an empirical evaluation, this research shows (a) that multi-strategy planning improves the performance over single strategy planning in some toy domains, (b) that multi-strategy planning can solve problems in at least one complex domain (the kitchen domain), and (c) and that an unordered subproblem coordinated multi-strategy planner performs better in the kitchen domain than a problem coordinated one.

Acknowledgements

This work was supported by an NSERC grant to my supervisor Dr. Bruce MacDonald. I received a scholarship from the Alberta Microelectronics Corporation ('91 – '94) and the MacMahon Stadium Society ('93). The computer science department of the University of Calgary supported me with teaching and research assistantships.

I owe a great deal to my supervisor, Dr. Bruce MacDonald, who helped me in my years at the University of Calgary in many ways. He provided me with guidance when needed, but also allowed me enough freedom to pursue my own ideas. I learned a great deal from him. I also would like to thank his wife Sue for tolerating many late night sessions. A special thanks also to Dr. Lisa Higham for her comments on earlier drafts of the analysis chapter.

Graham Birtwistle always provided me with good advice on life, the universe, and everything over our tea times.

Cameron Patterson is not only a good friend, but also introduced me to VHDL design. I enjoyed collaborating with him on the design of the VLSI design course in Calgary.

My thanks to the other people in the research community with whom I had many enjoyable moments discussing research: Robert Holte, John Anderson, Qiang Yang, and Diana Gordon.

My greatest thanks go out to my mother Relinde, my father Johannes, and my sister Beate for their love and continuing encouragement to strive for perfection.

A special thanks belongs to the Moore family: Barry, Marguerite, Nick, Tony, Lisa, Allison, Caroline and her family. They were my home away from home and I will always remember the great times we spent together. My only regret is that I haven't been out to Montreal yet.

There are also the many friends that I made: Graham Lai and his chess lessons, Sami Cokar, Stella and William Lee, Beverly and Richard Haller, Peter Graumann, and Randy Menzer. Many thanks also to the friends in speed skating, the ice hockey team, my friends from the Malaysian student club, and the aikido club for making me forget my work for a while.

Last but not least, this thesis would not have been possible without Ching Ching Cheah's love and support. Although, we were thousands of miles apart, we were never closer and I've got the phone bill to prove it.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

> The stumbling way in which even the ablest of the scientists in every generation have had to fight through thickets of erroneous observations, misleading generalizations, inadequate formulations, and unconscious prejudice is rarely appreciated by those who obtain their scientific knowledge from textbooks.
>
> **James Bryant Conant**, *Science and Common Sense*, 1951.

Planning is a popular research area in artificial intelligence. Unfortunately, theoretical results show that planning is indeed a difficult problem and that it can not be solved in general. However, by making assumptions about the domain, planning methods have been developed that work well in domains that satisfy the underlying assumptions. Unfortunately, these planning methods do not perform well in other domains. The main focus of this work is multi-strategy planning, that is a system that can combine different planning strategies to improve its performance. There are three types of improvement. Firstly, a multi-strategy planner can increase the coverage (i.e., the set of problems that can be solved efficiently) over single strategy planners, if for example the coverage consists of the union of the individual strategies. Secondly, there may be problems that can not be solved efficiently by a single strategy, because no strategy can solve all resulting subproblems efficiently.

These problems may be solvable by a multi-strategy planner if it contains strategies for all subproblems. Thirdly, different planning strategies can be combined such that the disadvantages of one strategy can be alleviated by some other strategy. For example, a strategy may lead to often re-occuring subproblems which decreases the performance of the planner. By combining this strategy with another (i.e., caching), this decrease in performance can be alleviated. The improvement of a practical multi-strategy planner is in general a combination of these three methods. Furthermore, a multi-strategy planning system allows the analysis of different strategies within a uniform framework, which allows for better comparisons between strategies.

1.1 Planning

The ability to solve problems is one of the most interesting aspects of human or artificial intelligence. In the broadest sense, problem solving can be interpreted as any goal directed behavior of an agent. However, this definition of problem solving is vague and fits almost all human behavior.

Therefore, this thesis focuses on a particular type of problem solving, the strategic planning problem. Informally, strategic planning is the type of planning humans do if they decide on a course of action before acting. Since this type of reasoning is an integral part of the design of any autonomous system, a domain independent planner is desirable so that the planner can be reused.

There are many different formalizations of strategic planning. At the heart of a planning formalization is the representation of change. The simplest representation is based on situations, snapshots of the world [**?**]. Other more expressive possibilities include modal logics [**?**], procedural representations [**?**], and event-based representations [**?**].

This thesis focuses on the classical planning paradigm, based on a calculus of situations. Situation calculus is based on these assumptions: (a) only one

event occurs per discrete time step, (b) time step duration does not matter, and (c) that the state changes are important only at the start and end of a time step, not during a time step.

1.2 Problem solving and planning

Planning has been an early and popular research area in artificial intelligence [?, ?, ?, ?, ?, ?, ?, ?].

The classical planning problem is: Given (a) a set of operators, that describe all possible actions of an agent, (b) a description of the world, called the initial state, and (c) a description of a set of desired states, called the goal, find a sequence of operators that transforms the initial state into a goal state.

The set of possible actions is complete, and correct. The agent knows all actions and all effects of the actions are known. Also, all relevant facts about the world are known by the agent. An agent knows exactly whether an action can be executed and what the result of this action will be.

Although this representation of planning is limited in its expressive power, for example it can not deal with more than one agent acting at a time, it is nevertheless of academic and practical importance. Even in this limited representation planning is intractable. The difficulty of the planning problem is its computational complexity: the time to find a plan grows exponentially with the length of the solution. This means that planning is infeasible for large problems with many primitive solution steps. Therefore, to solve a problem, the search space has to be reduced. A search space can be reduced by restricting, restructuring, or reordering the original space. The different methods of reducing the search space, their applicability conditions, and their effect on the search space are of interest in planning research.

Planning is also of practical importance because it is fundamental in the design of autonomous agents, robots as well as softbots [?]. These agents are

ultimately designed to alleviate the load on the user by performing tedious tasks, such as household chores or cleaning up a file system. Examples of other domains of great practical importance are machine shop scheduling, transportation problems, or VLSI layout. In these domains, the vast number of possible situations and goals makes preprogrammed solutions infeasible.

1.3 Planning biases and strategies

Based on the notion of an inductive bias in machine learning [**?**], this thesis introduces *planning bias* to describe assumptions about the domain that allow a designer to reduce the search space. Examples of planning biases include assumptions about the structure of the search space, the domain description, the plan structure, the problem set, and the order of the problems. A *planning strategy* is any method of restructuring, restricting, or reordering the search space of a planner to exploit some planning bias. Planning systems based on particular planning strategies work well if the underlying assumptions are met, and fail if they are not. Unfortunately, no single bias has been found to be superior or even sufficient in all domains. This led researchers to conclude that planners must use different planning biases in different domains [**?**].

Comparison of strategies is made more difficult, because the underlying assumptions (planning biases) are often implicit and applicability conditions for different planning biases are not well understood. Furthermore, a planning strategy may only be appropriate for parts of a problem rather than the complete problem.

1.4 Multi–strategy planning

The main contribution of this thesis is to develop a unified method for using multiple planning strategies on a single problem. Rather than developing

another planning bias, this thesis focuses on the selection and combination of different planning strategies on a single problem. This type of planning is referred to as multi-strategy planning. There are a number of reasons why a multi-strategy planner is desirable. This work focuses on improving performance. Another advantage of a uniform framework is that it allows a comparison of different planning strategies. Since the planning biases of most planning strategies are implicit, it is difficult in general to predict how a given planning strategy will perform on a given problem. A uniform framework allows better comparison of different planning strategies and thus allows a mapping of planning problems to planning strategies. From this, the planning biases of different planning strategies can be determined and methods for predicting the performance of a planning strategy can be developed.

1.4.1 The need for multi-strategy planning

This subsection motivates the need for a multi–strategy planning system with a short example. As mentioned previously, there are many different planning strategies, three of the most prominent ones are case–based, abstraction-based, and macro-based planning. The example will show that none of them alone is sufficient, and that a combination of approaches is required to solve the example problem efficiently.

Assume that I want to fly with a small airplane from Calgary to Fairmont, a small airport about 100 miles west of Calgary. Before the flight, I create a flight plan, detailing altitudes, speeds, and headings for the trip. This scenario fits well in the classical planning paradigm. Calgary is a big international airport with standard procedures and my home base. Since I am familiar with the airport, I can retrieve a case (Calgary, west bound departure) for the first part of the flight. When departing Calgary to the West, there is a fixed sequence of legs that will lead me out of the Calgary airspace. Once outside of Calgary, there are two "ways" to get to the destination Fairmont: either high and direct over the mountains, or low following

the valleys. However, the choice of exact route depends on many different factors, including forecasted weather (cloud levels and winds), purpose of the flight (sight seeing vs. speed), and type of aircraft. With too many variables case–based planning is unwieldy. Similar to abstraction-based planning, I choose an abstract plan (direct or through the valleys) and refine it further. Arriving at the destination, there is a standard procedure (fly a rectangular pattern around the runway) for landing at a small airport. In contrast, to a case, however, key aspects of this procedure are parameterized. For example, the exact headings depend on the orientation of the runway. To create this part of the flight plan, a macro describing a variabilized version of the pattern is most suited.

1.4.2 Requirements for multi-strategy planning

To combine different planning strategies, a multi–strategy planner must provide the following capabilities:

1. a usable definition of planning strategies
2. a representation covering different planning strategies
3. a decision procedure that determines when a planning strategy is appropriate
4. a search control method that given a planning strategy in the above representation results in similarly (preferably identical) structured search spaces.

To achieve the first goal, this thesis interprets planning as search through the space of partial plans. In this framework, a planning strategy is defined by its language for describing partial plans, its transformations on evolving plans, and its search method.

In this thesis, *general operators* are proposed as a solution to the second problem. Syntactically, general operators are similar to STRIPS operators.

Associated with a general operator are not only the STRIPS pre-conditions and effects, but also a set of *refinements*. A *refinement* is a sequence of operators that guarantees that the effects of the parent operator are achieved, but may have additional pre-conditions and/or effects. However, the semantics of general operators are different from primitive operators, since they encode search control knowledge. A general operator contains applicability conditions, that is a set of conditions that describe under what conditions the planning strategies described by this general operator should be applied. As will be shown in section **??**, general operators are powerful enough to represent many different planning strategies, such as case-based planning, macro-operators, abstraction-based planning, subgoals, reactive rules, forward chaining, etc.

Since general operators include applicability conditions, they imply a decision procedure as solution to the third requirement. The problem is that for some planning strategies it is difficult to determine whether they will be able to reduce the search cost or not. The decision procedure described in this thesis is equivalent to matching the applicability conditions of a general operator.

The representation of a planning strategy is not sufficient as can be seen in the following macro-operator example. Most macro-operator planners add macros to the operator set and chain macros together to reduce the solution length, whereas some systems generate a macro-operator set that completely replaces the original operator set, for example Korf's MPS system [**?**]. Another example is the difference between cases and macros. The representation of macros and cases are similar, but the resulting search spaces are very different. Cases are retrieved and adapted to a new situation by possibly substituting variables, inserting operators, and/or replacing operators. The retrieval is based on a similarity metric, to find the most suitable candidate. Macros are compiled decision steps and are used to create direct links between previously only indirectly connected nodes in the search space. Macros

are not adapted to a new situation, they are only chained together to form a new plan.

Therefore, a multi-strategy planner must have a search control method that takes advantage of different planning strategies given their representation as general operators. This thesis proposes such a static unified search control method as a solution to the fourth requirement. The planner is adapted through the generation of new general operators that are added to the operator set. Another possibility would be to have one search control method for each strategy. Then the combination of planning strategies is more difficult, since now the control of the planner has to be changed on a program level instead of through a general operator. The main intuition behind the search control method is that different planning strategies can be compared and ordered based on the size of the resulting search space.

We use a strongest-first heuristic: the planning strategy resulting in the smallest search space is tested first. Checking a small search space first has two benefits: (a) if a solution exists in that space, it can be found more quickly, and (b) if no solution exists the failure can be recognized more quickly. First, the most similar general operator to the current problem is retrieved and tested. If the general operator represents a macro that exactly matches the problem, a solution is found. If the objects do not match, the general operator is adapted to the current situation by substituting variables. If the plan still fails, the search method adapts the operator sequence further by inserting, removing, and replacing operators. If the operator sequence does not lead to a solution, a new subproblem space is created. There are three types of subproblems spaces: an abstract subgoal, a serial subgoal, and a general subgoal. The subgoals differ by the constraints on the search algorithm when trying to find a solution to the subproblem.

1.5 Learning general operators

Assuming an efficient search control method that can use many different planning biases, the question remains how the system acquires suitable general operators. Because general operators are a weak representation with few constraints on the parent operator and its refinements, the creation of general operators must be carefully controlled to avoid increasing the branching factor too much.

General operators can be pre-programmed, created automatically through an analysis of the domain description, or acquired from supervised or unsupervised learning. DoLittle, the implementation of multi-strategy planning described in this thesis, is part of an instructable system and is able to learn new general operators from previous problem solving episodes. After a problem is solved or aborted the derivation is passed on to a set of planning bias learners that analyze the problem solving trace and create general operators to speed up the planning process on similar problems in the future. The plan derivation may be annotated by the user to control the learning.

Creation of new general operators is controlled by providing a set of planning bias learners, e.g., a case-based planning learner, that have knowledge about the specific search control method that the planner uses. Therefore, a planning bias learner knows *sufficient* conditions for improvement, i.e., the learner knows that if a certain set of conditions are met, the addition of a specific general operator will improve performance. However, the learning modules do not know the *necessary* conditions for improvement, since those are too difficult to analyze. This means that the learning modules are over-conservative; in certain cases a learning module will reject a general operator even though it would improve performance. The learning modules are conservative, to balance the increase in the branching factor and the search reduction.

DoLittle uses a dynamic filter to check the performance of the learned operators in subsequent problem solving episodes and to check whether the

sufficient conditions of the learner are true or not.

1.6 Outline of the thesis

Chapter ?? motivates multi-strategy planning through an extended example. The domain is a simulated kitchen robot that is able to make beverages such as tea, coffee, or milk. The example shows that the minimum necessary search length for a multi-strategy planner is significantly less than that of four single strategy planners: means-ends analysis, macro-based planning, case-based planning, and abstraction-based planning.

Chapter ?? reviews a collection of important planning systems. These planners represent common planning strategies or methods for improving performance. In section ??, the plan space search paradigm is introduced and the different planning methods are compared within this paradigm. This comparison results in a practical definition of planning strategies. A planning strategy is defined by a plan language, a set of plan transformations, and a search method.

Chapter ?? develops a theoretical framework for multi-strategy planning. This framework is based on a model of abstraction developed in section ??. The framework identifies a number of dimensions for the comparison of multi-strategy planning systems (problem coordinated vs. subproblem coordinated, ordered vs. unordered, exhaustive vs. non exhaustive, and the decision procedure). Section ?? shows that the planning strategies described in chapter ?? match the model of abstraction. Section ?? shows sufficient conditions under which a multi-strategy planner exponentially reduces the cost of finding a solution compared to means-ends analysis, case-based, abstraction-based, and macro-based planning.

Chapter ?? describes the design of the unordered subproblem coordinated multi-strategy planner DoLittle. In section ??, the representation of different planning strategies is shown. The representation combines meta

knowledge (applicability conditions) with a description of a set of planning strategies. Section ?? describes DoLittle's search control method, which emulates different planning strategies given their representation as general operators.

Chapter ?? contains the description of DoLittle's planning bias learners. Currently, DoLittle contains three different learners: cases, macros, and abstractions. The different learners illustrate the different planning biases of the associated strategies.

Chapter ?? describes the empirical evaluation of DoLittle on three domains; the blocksworld, the towers of Hanoi, and the kitchen domain. Multi-strategy planning improves the performance over single strategy planning in all domains. There best single strategy planner varied for each domain. The case-based planner is the best single strategy planner in the blocksworld and the kitchen domain, the abstraction-based planner is the best one in the towers of Hanoi domain. The experiments in the kitchen domain show that (a) multi-strategy planning as implemented in DoLittle is able to solve complex problems in the kitchen domain, and (b) that a subproblem coordinated multi-strategy planner performs better than a problem coordinated one.

Chapter ?? compares DoLittle to other multi-strategy learning and planning systems.

Chapter ?? summarizes the main contributions of this dissertation and provides some comments about possible extensions to DoLittle and other future work.

Chapter 2

Scenario

> Few things are harder to put up with than the annoyance of a good example.
>
> **Mark Twain**, *Pudd'nhead Wilson*, (Chap. 1, Pudd'nhead Wilson's Calendar).

In this chapter, the idea of multi-strategy planning is expanded through a more detailed example: a simulated kitchen with a one-armed mobile robot (described in section **??**). The chapter introduces four common planning strategies: means-ends analysis (subsection **??**), case-based planning (subsection **??**), planning using abstractions (subsection **??**), and macro-operators (subsection **??**). Certain parts of the kitchen domain either support or violate the different underlying assumptions of the different planning strategies.

This chapter simplifies the description of planning in the example by making the following assumption: The cost of planning is mostly determined by the number of decision steps necessary to find a solution, i.e., the decision length. A decision step is a choice point in the search, for example choosing an operator from a set of operators or an instantiation for a variable. In most planning systems, the decision length is identical to the solution length, that is the number of primitive operators in the solution. As will be shown in chapter **??**, the decision length is the dominant factor in planning cost and

is mostly insensitive to the branching factor.

In the kitchen domain, none of the described planning strategies is sufficient by itself. Although the planning strategies solve parts efficiently, the violation of some underlying assumptions means that other parts are solved inefficiently. This means that the planner has to search for a significant part of the solution and the cost of planning using the individual planning methods is exponential in the length of the solution.

The example shows that multi-strategy planning (subsection **??**) significantly reduces the decision length and that the cost of planning is constant in the decision length.

At the conclusion of this chapter (section **??**), the results derived from the kitchen domain are generalized to other domains.

2.1 The kitchen domain

Assume that we are designing a planning system for an intelligent domestic robot; a robot that does common household chores, such as cleaning, cooking, and painting the house. Although such a robot must be able to deal with the complexities of the real world (uncertainty of actions, restrictions on perception, or interactions among multiple agents), a strategic planning component is nevertheless important. For example, strategic planning can prevent the robot from painting itself into a corner. The robot is a simple one, it can move around the house and it has one arm that it can use to affect the environment.

As part of its job, the robot is required to prepare simple beverages such as tea, coffee with cream, or milk with honey. The robot is used in varied tasks around the house, so its strategic planner is designed as a domain-independent planning system that reuses the planning and learning components of the robot.

Figure **??** shows the simulated kitchen domain. Although I would be

extremely interested in a household robot to diversify my pizza and beer diet, a simulation instead of a physical robot was used. Since the focus of this research is on the strategical planning capabilities of such a robot a simulation is an appropriate simplification.

There are four separate locations in the kitchen: at the sink, the table, the stove, and the fridge. The kitchen contains the following appliances: a sink, a garbage-can, a microwave, a coffee-maker, a stove, and a fridge with an ice dispenser. A cupboard with cups and glasses and a drawer for silverware is available. There is a coffee maker on the table, and a kettle on the stove. On a shelf which can be reached from the sink, there is a tea box with tea bags, a jar of instant coffee, and a coffee jar, as well as jars with honey and sugar.

In this thesis, the robot's task are restricted to preparing beverages, such as tea, coffee with cream and sugar, or milk with honey.

A complete list of primitive operators is given in table **??**. The complete domain description of the kitchen domain in the PRODIGY domain description language is shown in appendix **??**. Although the domain is similar to Hammond's CHEF system [**?**], the primitive operators are at a lower level of abstraction than the ones used by Hammond, for example PICKUP-FROM-CUPBOARD versus CHEF's STIR-FRY-VEGETABLES.

This level of abstraction was chosen so that it is at least plausible to implement the primitive operators as specific behaviors of the robot, similar to Brooks's artificial life systems [**?**, **?**]. For example, there are two different operators, SCOOP-INSTANT-COFFEE and SCOOP-HONEY, although both have similar functions, i.e. using a spoon to add an ingredient. However, the actions of the robot are different when handling honey as opposed to instant coffee powder, since in the first case, the spoon has to be rotated to stop the honey from running off, whereas the spoon has to be kept still to avoid dropping the instant coffee powder. Combining the operators SCOOP-INSTANT-COFFEE and SCOOP-HONEY into a single operator would make the representation of the domain easier, but would make implementing the oper-

ators as behaviors more difficult. In the following paragraphs, the different operator classes are described.

The operator MOVE-ROBOT moves the robot from a location to a neighboring location, e.g., (MOVE-ROBOT AT-SINK AT-TABLE). To move the robot from the sink to the stove, two primitive moves are necessary: (MOVE-ROBOT AT-SINK AT-TABLE) followed by (MOVE-ROBOT AT-TABLE AT-STOVE).

The second class of operators allows the agent to pick up and put down objects in various locations. There is a separate version of the PICKUP and PUTDOWN operator for the table, the drawer, the sink, and so on. There is also a special operator that enables the robot to throw things into the garbage can.

There are two primitive operators to open/close things. One handles containers (e.g., the tea box, the honey jar, or the coffee jar) and one handles appliances (e.g., the microwave, the fridge, or the drawer).

The remaining operators use different appliances and objects in the domain. For example, there are operators to use the microwave (HEAT-WATER-IN-MICROWAVE), the silverware (CUT, STIR), and the water tab (FILL-WITH-WATER $OBJECT, TURN-WATER-OFF).

There are also operators to handle ingredients, such as tea, milk, or sugar. The robot can scoop ingredients (e.g., honey, sugar, or instant coffee) with a spoon into cups or glasses, stir beverages, and fill the coffee maker. It can also pick up tea bags from the tea box.

The kitchen domain is a complex domain: plans often consist of hundreds of primitive steps, the number of objects and operators in the domain is larger than in other toy domains. Therefore, the kitchen domain is clearly beyond the capability of brute force planners, such as simple depth-first search. Table **??** shows the primitive steps to create a cup of tea. This plan contains 30 elementary steps. PRODIGY is unable to find a solution to this problem

Figure 2.1: The kitchen domain

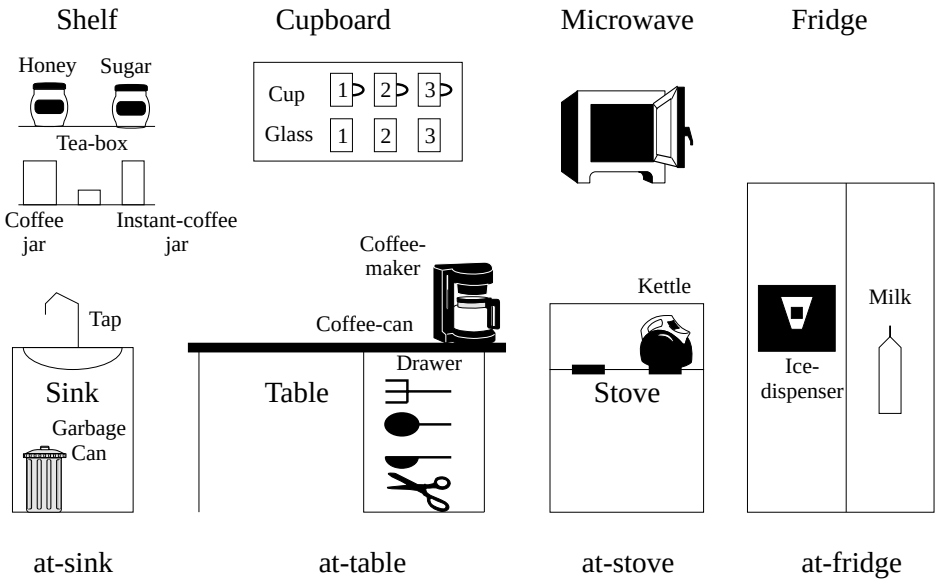


Table 2.1: Primitive operators in the kitchen domain

O1 Move-Robot $FromLoc $ToLoc
O2 Pick-Up-From -Drawer $Object,
-Table, -Sink, -Fridge,
-Microwave, -Cupboard, -Stove,
-Shelf
O3 Put-In -Drawer $Object
-Table, -Sink, -Fridge,
-Microwave, -Cupboard, -Stove,
-Shelf
O4 Put-In-Garbage-Can $Object
O5 Open-Door $Appliance, Close-Door $Appliance,
O6 Open-Container $Container, Close-Container $Container
O7 Heat-Water-In-Microwave, Heat-Milk-In-Microwave
O8 Cut $Object, Stir $Cup
O9 Fill-With-Water $Cont, Turn-Water-Off
O10 Get-Tea-Bag, Make-Tea $Cont,
O11 Scoop-Instant-Coffee, Make-Instant-Coffee $Cont,
O12 Scoop-Honey, Add-Honey-to-Milk $Cont,
Add-Honey-To-Tea $Cont
O13 Add-Sugar-to -Milk $Cont,
-Tea, -Instant-Coffee
Get-Sugar
O14 Pour-Milk $Cont, Add-Milk $Cont

in the resource limits allocated during the tests[1].

2.2 Comparison of different planning methods

Let us consider the methods that four different planning strategies use in the kitchen domain: means-ends analysis (subsection ??), case-based planning (subsection ??), abstraction-based planning (subsection ??), and macro operators (subsection ??). The performance of these planning strategies is compared to that of a subproblem coordinated multi-strategy planner, that is a planner that can combine them on a single problem (subsection ??).

The problem in this example is to create a cup of instant coffee with sugar. The initial state for all problems is shown in figure ??.

To simplify the problem, it is assumed that the planner has previously made a cup of tea and remembered this plan. A solution to making tea is shown in table ??.

A solution to the problem of making instant coffee with sugar is similar to the plan for making tea. However, after heating a cup of water in the microwave, the robot must fetch the instant coffee jar and a spoon. After making instant coffee, the robot must also fetch the sugar and add it to the coffee. The latter part of the solution to this problem is shown in table ??. The total plan for making instant coffee with sugar contains 42 steps.

To estimate the cost of solving the example problem using a given planning system, this thesis computes the minimum decision length of a problem. The decision length is the number of choices the planner has to make to find a solution. The only choice a forward chaining planner makes is to select an operator and variable binding for an operator that is applicable to the current state. Therefore, a forward chaining planner's decision length is equal to the solution length, the number of operators in the solution. Chapter ?? gives

[1]The maximum of 15000 search nodes was exceeded.

Table 2.2: Making tea in the kitchen domain

```
1  OPEN-DOOR CUPBOARD                 ; get a cup and fill it
2  PICK-UP-FROM-CUPBOARD CUP1         ; with water
3  MOVE-ROBOT AT-TABLE AT-SINK
4  PUT-IN-SINK CUP1
5  FILL-WITH-WATER CUP1
6  TURN-WATER-OFF
7  PICK-UP-FROM-SINK CUP1             ; heat the cup with water in
8  MOVE-ROBOT AT-SINK AT-TABLE        ; the microwave and put it
9  PUT-ON-TABLE CUP1                  ; on the table
10 MOVE-ROBOT AT-TABLE AT-STOVE
11 OPEN-DOOR MICROWAVE
12 MOVE-ROBOT AT-STOVE AT-TABLE
13 PICK-UP-FROM-TABLE CUP1
14 MOVE-ROBOT AT-TABLE AT-STOVE
15 PUT-IN-MICROWAVE CUP1
16 CLOSE-DOOR MICROWAVE
17 HEAT-WATER-IN-MICROWAVE CUP1
18 OPEN-DOOR MICROWAVE
19 PICK-UP-FROM-MICROWAVE CUP1
20 MOVE-ROBOT AT-STOVE AT-TABLE
21 PUT-ON-TABLE CUP1
22 MOVE-ROBOT AT-TABLE AT-SINK
23 PICK-UP-FROM-SHELF TEA-BOX         ; get a tea-bag and put it in
24 MOVE-ROBOT AT-SINK AT-TABLE
25 PUT-ON-TABLE TEA-BOX               ; the cup, dispose of it
26 OPEN-CONTAINER TEA-BOX             ; afterwards
27 GET-TEA-BAG
28 MAKE-TEA CUP1
29 MOVE-ROBOT AT-TABLE AT-SINK
30 PUT-IN-GARBAGE-CAN OLD-TEA-BAG
```

Table 2.3: Making instant coffee with sugar in the kitchen domain

```
    ...
22  MOVE-ROBOT AT-TABLE AT-SINK              ; identical to the
                                             ; plan for making tea
23  PICK-UP-FROM-SHELF INSTANT-COFFEE-JAR    ; replace tea-box
                                             ; with
24  MOVE-ROBOT AT-SINK AT-TABLE              ; instant-coffee-jar
25  PUT-ON-TABLE INSTANT-COFFEE-JAR
26  OPEN-CONTAINER INSTANT-COFFEE-JAR
27  OPEN-DOOR DRAWER                         ; add steps and use
28  PICK-UP-FROM-DRAWER SPOON                ; a spoon
29  SCOOP-INSTANT-COFFEE
30  POUR-INSTANT-COFFEE CUP1
31  STIR CUP1                                ; stir instant coffee
32  PUT-DOWN-ON-TABLE SPOON
33  MOVE-ROBOT AT-TABLE AT-SINK
34  PICK-UP-FROM-SHELF SUGAR-BOX
35  MOVE-ROBOT AT-SINK AT-TABLE
36  PUT-ON-TABLE SUGAR-BOX
37  OPEN-CONTAINER SUGAR-BOX
38  PICK-UP-FROM-TABLE SPOON
39  SCOOP-SUGAR
40  ADD-SUGAR CUP1
41  STIR CUP1
42  PUT-ON-TABLE SPOON                       ; done
```

more details of those concepts. Since search is exponential in the decision length, it is the dominating factor in the solution cost. Section ?? analyses the dependency of search cost on decision length and branching factor.

2.2.1 Means-ends Analysis

Means-ends analysis looks at the differences between the current state and the desired goal state and selects a difference. All operators are classified by the difference that they reduce[2]. Means-ends analysis then chooses a relevant operator, that is an operator that reduces the selected difference, and a variable binding. An operator may either be applied to the current state (if all its preconditions are satisfied) or it may be added to the sequence of pending operators. The unsatisfied preconditions of the operator yield new subgoals.

Table ?? gives a brief summary of how means-ends analysis tries to generate a plan to make instant coffee with sugar. First the differences between the initial state and the goal state are found. Here there are three differences: (a) the cup does not contain instant coffee, (b) the cup is not on the table, and (c) there is no sugar in the cup. This example assumes that the planner selects the first difference and tries to achieve it. In the kitchen domain, there is only one operator to make instant coffee MAKE-INSTANT-COFFEE and only one variable binding will yield the desired effect ($CUP = CUP1). Comparing the preconditions of the operator MAKE-INSTANT-COFFEE shows, that there are some un-satisfied pre-conditions. Selecting an unsatisfied pre-condition creates a new subgoal and the planner is called recursively. In our example, the planner selects (CONTAINS CUP1 HOT-WATER) as a new subgoal.

Means-ends analysis works well in domains with only a few relevant operators and with linear subgoals. Subgoals are linear, if they can be achieved in any order, and can be achieved without violating previously achieved subgoals [?]. These conditions are problematic in the kitchen domain; for ex-

[2]This classification is computed automatically by analyzing the effects of an operator.

Table 2.4: Means-ends analysis example

Goal	(CONTAINS CUP1 INSTANT-COFFEE) (IS-ON CUP1 TABLE) (CONTAINS CUP1 SUGAR)
Initial State	shown in figure **??**
Differences	(CONTAINS CUP1 INSTANT-COFFEE) (IS-ON CUP1 TABLE) (CONTAINS CUP1 SUGAR)
Select Diff	(CONTAINS CUP1 INSTANT-COFFEE)
Select Op. and Binding	MAKE-INSTANT-COFFEE CUP1
Compare Pre-cond	(CONTAINS CUP1 HOT-WATER) (HOLDING SPOON) (CONTAINS SPOON INSTANT-COFFEE-POWDER) (IS-ON CUP1 TABLE) (IS-AT ROBOT AT-TABLE)
Select Pre-cond and and create new subgoal	Subgoal (CONTAINS CUP1 HOT-WATER)

ample, there are eight operators that can achieve (HOLDING $X) and plans are often not linear. In the example above, the goal (CONTAINS CUP1 HOT-TEA) must be achieved before achieving (IS-ON CUP1 TABLE), since heating up the cup requires that it is moved from the table, to the microwave. Another example of a non-linear subgoal is in the example, where (CONTAINS CUP1 HOT-WATER) must be achieved before the other alternatives, such as (HOLDING TEA-BAG).

PRODIGY, a means-ends analysis planner can not find a solution within the resource limits (15000 nodes), since the search is not guided sufficiently. Also, since means-ends analysis does not make use of previous planning episodes, it will have to create a plan to make tea with sugar from scratch. In our example, the solution length for making instant coffee with sugar is 42 steps for means-ends analysis and the average branching factor in the kitchen domain is around 3.5, which results in a worst case search space of around

7×10^{22} nodes.

2.2.2 Case-based planning

This section gives a brief example of case-based planning. A more detailed description can be found in section **??**. Case-based planning attempts to reduce the cost of planning by modifying a previously generated plan to fit the new problem instead of creating plans from scratch. The changes include substituting variables, replacing some operators, or removing operators from the original plan.

One feature of case-based planning is its use of powerful indexing methods to retrieve similar cases, since one expects the planner to contain many plans in memory. The indexing is usually based on goal similarity.

The planning bias of case-based planning is that: (a) similar plans can be found efficiently through indexing, (b) plans can be quickly modified to fit the new situation, and (c) the individual adaptations are mostly independent.

In this example, a case-based planner has previously solved the problem of making tea, and retrieves this plan when solving the problem of making instant coffee with sugar.

Next, the resulting plan is analyzed and problems are fixed. For example, at step 23, the tea box has to be replaced with the instant coffee jar, because tea does not occur in the goal specification. Ignoring the cost of removing unnecessary operations, a successful solution requires the creation of a suffix plan that gets the instant coffee jar, a spoon, and the sugar. Then the robot must scoop the instant coffee and sugar into the cup. This suffix plan are steps $23 - 42$ of the solution in table **??**.

In absence of other knowledge, there is no more guidance for the creation of the suffix plan, and a case-based planner uses a weak planner, such as means-ends analysis, to create this plan.

Therefore, in this example, solving the instant coffee problem requires a search of depth 20. This is much better than the original decision length of 42

steps, but still expensive. This estimate is an underestimate, since it ignores the cost of indexing, and of removing unneeded plan steps. However, in the worst case the search space is still too large to allow case based planning to solve this problem (around 76 billion nodes if the branching factor is 3.5). The underlying requirement of case-based planning that is violated is that the cost of the individual adaptation is small. In the example, the cost of creating the suffix plan is too big.

2.2.3 Abstraction-based planning

Abstraction based planning ignores low level details in the beginning to create an abstract plan. The abstract plan is then refined to include the omitted low level details. For example, an abstract planner may ignore the existence of hot water or the current location of the robot, and create an abstract plan consisting of one abstract operator: either (a) brew fresh coffee, or (b) use instant coffee. The abstract plan for making instant coffee might look as follows:

ABS-PLAN: MAKE-INSTANT-COFFEE-WITH-SUGAR
1) Fill cup with water
2) Heat water
3) Add instant coffee
4) Add sugar

This abstract plan is then refined to include more details. For example, step two (HEAT-WATER) is refined, by choosing a method to heat the water (the stove or the microwave). If the planner uses the microwave, the refinement of step two is similar to the macro GET-HOT-WATER shown in the next section. The differences arise from lower level details such as the current position of the robot or the status of the doors.

One disadvantage of abstraction-based planners is their inability to provide guidance in the search for an refinement of abstract operators, which can be a non-trivial search problem. For example, heating a cup of water in the microwave is a common operation in the kitchen domain, but an abstract

planner has to search for a refinement of this operator every time it is needed. Getting a cup of water requires at least 6 steps in itself.

The crucial factor in abstraction-based planning is that all individual search problems are small enough to be manageable. This requires that the individual search problems are roughly of the same size. Subproblems should have roughly the same size, since each subproblem's contribution to the final solution is linear in the length of the subproblem, whereas the cost of finding a solution to the subproblem grows exponentially with its length. If one subproblem requires many more solution steps than the other subproblems, its cost will overwhelm any improvement on the other subproblems.

ALPINE, a popular abstraction-based planner, creates a two-level abstraction hierarchy for the example. As will be shown in the analysis of two-level abstraction hierarchies in subsection **??**, the optimal partitioning of a problem into two abstraction levels is given if the abstract plan contains $\sqrt{l}$ abstract operators that each correspond to a ground level problem of length $\sqrt{l}$. In our example, this will result in decision lengths of $\sqrt{42} \approx 7$. The total cost of abstraction-based planning in this example is thus the cost of solving 1 abstract problem and 7 subproblems in the ground space each with a decision length of 7 steps.

2.2.4 Macro-based planning

Macros are fixed sequences of primitive operators with parameterized arguments. The idea is that by creating macros for often used subtasks, problems that include these subtasks can be solved more efficiently, since fewer reasoning steps are necessary. Objects in the subtasks are commonly generalized so that learned macros are more widely applicable. Traditionally, a macro learner extracts sequences of operators based on some operationality criteria, generalizes them, and adds the resulting macro to the operator set. The disadvantage of adding a macro-operator to the search space is that it increases the branching factor of the resulting space by adding extra connections be-

tween previously only indirectly linked states. Etzioni showed that even a small increase in the branching factor must be accompanied by a large reduction in the search size to be useful [**?**]. This analysis is substantiated through experimental results that show a slowdown for indiscriminate macro-learners [**?**, **?**, **?**, **?**].

Minton identifies another problem of macro-based planning, the hidden cost of matching macros. Since matching an operator to the current situation is an expensive operation (exponential in the number of variables), Minton shows that even if macros would not increase the branching factor, they may still outweigh their benefits by increasing the cost of node expansion, since at each node, the planner has to test whether a macro is applicable or not.

The goal of a macro-learner is to find common subsequences in plans and to generalize them. In the scenario shown in this chapter, the planner has to create two plans ({make-tea} and {make-instant-coffee-with-sugar}).

A common subsequence of these and many other plans in the kitchen domain is captured by the macro Macro-Fill-with-water $Cup. The operator Macro-Fill-with-water is useful, since it allows the planner to get a cup of water using one operator as opposed to reasoning about 4 primitive operators necessary to achieve the goal.

MACRO: Macro-Fill-with-Water $Cup
1) Put-in-Sink $Cup
2) Fill-With-Water $Cup
3) Turn-Water-Off
4) Pick-Up-from-Sink $Cup

However, what if the robot is not at the sink, but at a different location? In this case the macro can not be used. Furthermore, the Fill-with-water $Cup only works if the water is currently turned off. Of course, a planner might add macros for situations in which the robot is at the table, the stove, and so on. But since the above conditions (location of the robot and state of the water tab) are independent, a macro learner would have to create a

macro for each pair of conditions (location, water on/off). Other conditions are whether the water is needed at the end of the macro or not, or whether the sink is empty or not.

This means that the number of macros to cover all situations may grow exponentially in the length of the macros. This will result in a large increase in the branching factor and the cost of expanding a node. In fact, macro learners that augment the original operator set are based on the implicit constraint that one or a few macros are sufficient to cover most situations.

Therefore, macros in general are short, widely applicable operator sequences. Long macros have the problem that since they can not be adapted to the new situation, they are too specific. However, a macro must be useful in a large set of instances to overcome the utility problem. Therefore, assume that the longest macro contains 4 primitive operators. Four primitive operators seems to be the longest of reusable operator sequences. For example, the macro described in section **??** created the macro MACRO-FILL-WITH-WATER shown above. Now even under the optimistic assumption that a planner has macros for all needed operator sequences of size up to 4, the decision length of making instant coffee with sugar is still $\lceil l/4 \rceil = \lceil 42/4 \rceil = 11$.

Ignoring the cost of an increase in branching factor and the additional match cost, macro-based planning is too expensive to be practical (roughly $1,000,000$ nodes with a branching factor of 3.5 in the example).

2.2.5 Multi-strategy planning

The intuition behind a multi-strategy planner is that (as shown in the previous examples) planning systems are able to solve parts of a problem efficiently, but have difficulties with other parts. The problem is seen as a set of subproblems. There is an efficient solution method for each subproblem, but not a single one that works well for the whole problem.

The goal of multi-strategy planning is to design a planning system that can:

1. break down the original problem into subproblems,

2. select an efficient solution method for each subproblem and solve the subproblems using this method

3. combine the solutions to the subproblems into a complete solution.

Assume that there is a planning system that combines four different problem solving methods: means-ends analysis, cases, abstractions, and macros. In the example, the planner finds the shortest solutions, for example by using an admissible search function, such as iterative deepening or breadth-first.

First, it will retrieve the plan to make tea and adapt it. This leads to a new problem, the creation of a suffix plan that contains 20 primitive operators. However, instead of using means-ends analysis to create the plan, a multi-strategy planner can make use of an abstraction hierarchy. If the planner creates a two-level abstraction hierarchy as in subsection **??**, the individual subproblems will be of size $\sqrt{20} \approx 4$. There is one abstract problem and 4 ground space problems. Instead of solving the ground space problems using means ends analysis, the planner has learned macros to solve them. Therefore, the solution length of solving them is equal to $4/4 = 1$. If the planner also contains abstract macros, all subproblems can be solved by application of a single operator. Then the decision length of multi-strategy planning is a single step.

The important feature of a multi-strategy planner is its ability to solve subproblems efficiently and combine the resulting solutions. As can be seen in the make-instant-coffee-with-sugar example, the separation into subproblems is dynamic, for example a macro was used to solve an abstract problem which in turn generated a set of ground subproblems.

2.3 Conclusions

The previous example motivates multi-strategy through an example in the kitchen domain. A multi-strategy planner was able to do significantly better (decision length of 1) than planners based on means-ends analysis (42), cases (20), abstraction (7), or macros (11).

However, some details were omitted in the example for readability. For example, the additional cost due to an increase in the branching factor, the specific selection method for choosing different biases, etc. These issues are discussed in chapter ??.

Although, an efficient planner in the kitchen domain is desirable, the goal of this research is to provide a domain independent framework. Another example of multi-strategy planning is shown in subsection ??. The kitchen domain, however, contains a few features that make it a good example domain. Solutions contain many primitive operators. Also, there exist useful abstraction hierarchies and macros, but the resulting subproblem spaces are non-trivial. This distinguishes the kitchen domain from other commonly used domains. For example, in the blocksworld domain with three blocks, no nonlooping problem solution can contain more than six steps.

Chapter 3

Literature Review

> Progress is the injustice each generation commits with regard to its predecessors.
>
> **E. M. Cioran** (b. 1911) Rumanian born French philosopher, *The Trouble with Being Born*, ch. 8, 1973.

This chapter formalizes the concept of a planning problem and reviews previous work in this area. Section **??** lists some important previous approaches. Section **??** describes a unified framework for comparing planning strategies. In this framework, planning is viewed as search through the plan space. Different planning strategies are shown to be equivalent to different plan languages, sets of transformations on evolving plans, and search methods. This framework is used to develop a usable definition of a planning strategy. Section **??** compares different planning strategies by their transformation set and their plan representation language.

3.1 The planning problem

A domain in planning is the subset of the world that the robot is interacting with, for example, the kitchen domain described in chapter **??**. The task of

a domain independent planner is to find a description of a course of action that transforms the domain from a given state into a desired state. The domain description includes a set of *operators*. The operator set describes all possible actions of an agent in the domain. The first problem is to find a suitable representation for actions; one that allows reasoning about states before and after application of an action.

Even given a suitable representation, the planning problem is difficult. Erol et al. prove that for a popular representation, it is EXPSPACE complete [?]. The reason is that for some domains, the length of the optimal solution (i.e., the number of primitive operators in the shortest plan to solve the problem) grows exponentially with the input size. For example, adding one disk to the tower of Hanoi puzzle doubles the length of the solution. This result holds for function free description languages with delete lists (datalog language), for example the STRIPS representation, a minimally adequate representation language. Therefore, a planner must somehow reorder, reduce, or restructure the search space based on some assumption about the domain.

This section reviews early planning work and the structure of their search spaces. It first discusses totally ordered planners: the GPS (subsection ??), the STRIPS system (subsection ??), and the PRODIGY system (subsection ??). Secondly partial order planners are investigated (Subsection ??). Subsections ?? and ?? discuss relaxed and reduced abstraction hierarchies respectively. Lastly, it introduces CHEF as an example of a case-based planner (subsection ??).

3.1.1 The GPS system

Early work in planning includes Newell's "General Problem Solver" (GPS) [?, ?], a planner intended to both automatically solve interesting problems and to test the plausibility of a theory of human problem solving. Newell et al. analyzed human problem solving behavior by taking a detailed transcript of subjects in a logic domain. This transcript included utterances and meta

planning remarks of the user, for example, what particular goal a subject was trying to achieve.

The main contribution of GPS was its distinction of task specific knowledge and domain independent planning knowledge. Furthermore, through analysis of transcripts of human problem solving episodes, Newell developed the means-ends analysis search strategy. The basic idea behind means-ends analysis is to reduce differences between the current and the goal state. Operators are classified by the difference they may reduce. The differences are ordered according to their difficulty and more difficult ones are reduced first. An operator relevant to a selected difference is chosen. If the operator is applicable, it is applied and the search continues in a depth-first manner. If the operator is not applicable, rather than rejecting the operator, the unsatisfied preconditions of the operator yield new subgoals. A more detailed example of means-ends analysis is shown in subsection **??**.

GPS was limited in its expressive power because of its representation of objects. Ernst extended GPS by generalizing objects. The GPS system was able to solve simple logic reformulation problems automatically.

3.1.2 The Strips system

This subsection discusses STRIPS, a planning system designed at SRI international by Fikes et al. STRIPS' main contribution to planning is the (strict) STRIPS assumption, which states that only predicates mentioned in the effects of an operator are affected by the execution of an operator, all other predicates remain unchanged. This assumptions provides a practical solution to the frame problem.

Using the STRIPS assumption, it is easy to compute (a) the resulting state for a sequence of operators, (b) the correctness of a sequence (preconditions of all operators are satisfied, and (c) whether a plan achieves a goal (resulting state satisfies the goal).

The search control method used by STRIPS is a generalization of GPS's

means-ends analysis. In contrast to GPS, however, it does not assign difficulty levels to classes of differences.

The Frame, Ramification, and Qualification Problems

At the heart of any reasoning about actions is the problem of specifying the state of a complex world before and after application of an operator. Although other formulations are sometimes used [**?**], the formulation of the *frame* problem used in this thesis is the one first described by McCarthy and Hayes [**?**]. It is the problem of representing what predicates are not changed by an action. For example, the action PUT-CUP-IN-CUPBOARD in the kitchen domain does not change the location of the microwave. Solving these problems requires the addition of frame axioms that specify which predicates remain unchanged after execution of an action. For example, a frame axiom that specifies that PUT-CUP-ON-TABLE does not change the location of the cupboard. However, this approach may lead to a combinatorial explosion in either the number of frame axioms needed or the computational cost of applying those axioms.

The *ramification* problem is the complementary problem of specifying which predicates are changed by an action [**?**]. For example, PUT-CUP-IN-CUPBOARD does usually not change the location of the cupboard. What about the case in which the cup is heavy enough to tear the cupboard of the wall? The problem is that some effects of an operator may be context dependent and that an encoding of all possible contexts for an operator is intractable.

The *qualification* problem is the problem that operators have usually unspecified preconditions [**?**]. For example, the operator OPEN-DOOR has the following preconditions: (a) the arm is empty, (b) the door is closed, and (c) the door can be reached from the current location of the robot. If those conditions are satisfied, the planner assumes that the operator will succeed and the drawer will be open after applying it. However, what if the drawer

is nailed shut, the handle is missing, and so on? Again, it is infeasible to specify all conditions under which an action may fail.

The Strips representation

Most modern planning systems still use a variant of the STRIPS representation, e.g., PRODIGY, ALPINE, TWEAK, and DOLITTLE. This subsection describes a popular version of the STRIPS representation. The original STRIPS program allowed more complex preconditions, which in turn may require arbitrarily complex theorem proving to check the correctness of a plan. The STRIPS representation of operators consists of three lists, a list of preconditions, an add-list and a delete-list. Preconditions describe sets of states in which the operator is applicable. Add- and delete-lists specify which predicates are added or deleted respectively from the state description after application of this operator.

STRIPS, however, does not address the ramification or qualification problem. The effects of an operator only depend on the operator itself, instead of the state in which the operator is applied. This means that the STRIPS representation does not allow the representation of conditional effects. For example, STRIPS can not describe an electric toggle switch (If the light is on, then turning the switch results in the light being off, and vice versa). In practice this means that there may be more operators necessary to describe a domain that one would expect. In the example above, the toggle switch operator must be represented by two operators, one with the added precondition that the light is one, one with the light being off. In the blocksworld domain, the absence of conditional effects requires four operators (UNSTACK, PICK-UP) and (STACK, PUT-DOWN), instead of the more natural (PICKUP, PUTDOWN) operators.

Other approaches to the frame, ramification, and qualification problems include unrestricted logics, such as general frame axioms [**?**] and circumscription [**?**]. These approaches are computationally too expensive for practical

planning.

3.1.3 The Prodigy system

Prodigy is an ongoing research project in planning, machine learning, and knowledge acquisition at Carnegie Mellon University. The first version, PRODIGY2, was developed by Minton in 1988 [?]. PRODIGY2 is a means-ends planner similar to STRIPS. Whereas the original PRODIGY system was unable to interleave plans to achieve different subgoals, and was thus unable to find solutions to some problems (e.g., the famous Sussman anomaly), the latest version of PRODIGY4 supports non-linear planning by allowing interleaving of different subgoals. This is the main difference between the two versions. The interleaving of subproblems greatly increases the search space. Therefore, PRODIGY4 uses some search reduction techniques to improve performance: dependency-directed backtracking and look-ahead. Dependency-directed backtracking removes binding nodes as backtrack candidates for unachievable subgoals. Lookahead removes nodes that, for example, necessarily lead to a state or goal loop without computing a complete set of bindings.

PRODIGY4's [?, ?] domain description language (PDL) is based on the STRIPS assumption, but a number of extensions allow more expressiveness and simplify the specification of more complex domains. PDL represents preconditions of actions in a typed first order predicate logic. It allows conjunction, disjunction, negation, and quantified (existentially and universally) variables that may have a specific type. Furthermore, the effects of an operator may contain conditional effects.

3.1.4 Partial order planning: Noah, Nonlin, Tweak

This subsection discusses partial order planning. Partial order planning is sometimes referred to as non-linear planning in the literature. In this thesis, however, non-linear planning is the ability to interleave different subgoals,

whereas partial order planning is the generation of partially ordered as opposed to totally ordered plans. The intuition is that one partial order plan can be completed into a set of totally ordered plans (total extension), and that there may be an exponential number ($n!$ for an partial order plan with n unordered operators) of totally ordered plans that correspond to one partial order plan.

Noah

Sacerdoti's NOAH system was designed to instruct a human apprentice in assembly tasks [**?**]. NOAH extended the classical planning paradigm in three important aspects:

- partial order planning
- plan critics and plan debugging
- hierarchical planning

NOAH's representation language was based on the STRIPS assumption and similar to the original STRIPS representation with some notable enhancements. NOAH's operator representation encodes procedural as well as declarative knowledge. In contrast to previously mentioned systems with totally ordered plans, plans in NOAH are partially ordered and hierarchical. Operators consist of preconditions and effects as well as procedural nets. Associated with an operator are a set of refinements that expand into more detailed subgoals. The original problem is replanned into a sequence of more detailed subgoals, that will achieve the toplevel goal. At the lowest level, there are goals that can be directly achieved by a primitive operator.

NOAH's search strategy is a top down strategy. NOAH tries to avoid backtracking by delaying decisions as long as possible, which is why it is sometimes referred to as *least-commitment planning*. The motivation is to delay ordering and binding choices for operators until enough information is

available to make the correct choice, instead of guessing and then possibly having to backtrack. The expansion of an operator is controlled by "semantics of user problems" functions. All domain knowledge is encoded in these expansion functions.

The expansion of subgoals may lead to interacting subgoal problems. NOAH tries to overcome these problems by (a) not ordering subgoals unless necessary, and by (b) debugging a developing plan.

To detect potential interaction problems, NOAH creates a table of multiple effects. Each entry in the table is a predicate that is either added or deleted by more than one operator in the plan. NOAH uses plan critics to resolve conflicts in a plan. A plan critic is a domain specific or domain independent repair method.

NonLin

One main disadvantage is the lack of backtracking in NOAH. NONLIN is an extension of NOAH that adds backtracking and more powerful plan critics. The plan representation in NOAH was extended in NONLIN by a justification structure, so that the planner can keep track of why certain operators or constraints are in the plan. The justification structure allowed NONLIN to do a better job at debugging a flawed plan. For example, it was able to deduce for how long a given effect had to be maintained.

Tweak

Chapman's TWEAK system formalized non-linear planning and provided a complete set of plan transformations and a heuristically adequate truth criterion. TWEAK uses a representation of actions that is similar to the STRIPS representation. During the generation of a plan, not all parameters of an operator need to be instantiated. For example, TWEAK may contain the operator template in table **??**. The constraints show the possible constraints on variables that TWEAK supports.

Table 3.1: TWEAK operator template example

Operator OP1($X, $Y, $Z)
Preconds: ...
Effects: ...
Constraints: ($X = OBJECT1)
$Y =(≠) $Z

TWEAK supports the following variable constraints: (a) codesignation (two variables must be equal), (b) non-codesignation (two variables must not be equal), or (c) instantiation (a variable must be equal to a constant) as is shown in the example above. TWEAK's plan representation consists of totally or partially ordered operator templates, called *steps*.

TWEAK starts out with an empty plan and adds ordering constraints, binding constraints, and operator templates until all preconditions of all operators are established. Chapman proved that TWEAK's algorithm is complete, that is adding orderings, binding constraints, and operator templates is sufficient to guarantee that TWEAK can find a partial order plan that subsumes a successful plan if one exists.

One problem of partial order planning is the cost of evaluating the truth criterion. If all variables range over infinite sets, a greedy algorithm can be used to compute variable instantiations. If variables only range over finite sets, the complexity of finding a binding of variables is NP-hard. More powerful representations (such as including conditional effects, and quantified variables) make the truth criterion undecidable.

3.1.5 The Sipe system

The SIPE system [**?**] is a practical partial order planning systems. It provides a powerful formalism (*epistemological adequacy*) for describing domains while maintaining reasonable performance (*heuristic adequacy*). SIPE supports

hierarchical descriptions of a domain through abstract operators and abstract goals. It has a special resource reasoning component and is able to post and reason about variable constraints. SIPE includes continuous variables and a limited form of temporal reasoning. SIPE supports the creation of conditional plans, that is plans, that contain more than one possible execution path. The particular path taken depends on some condition that is tested during execution time.

The operator representation language in SIPE is an enhanced version of the STRIPS representation. SIPE supports typing of and posting of constraints on free variables. The set of possible constraints includes `(not)-is-value`, `(not)-same-as-variable`, `(not)-with-attribute-equal`. Variable constraints also support SIPE's continuous variable reasoning system. The preconditions of an operator in SIPE are slightly different from previous planning systems. SIPE's preconditions are used to encode meta-knowledge (when a certain operator *should* be applied), instead of when it *can* be applied. SIPE does not subgoal on unsatisfied preconditions, but applies operators only if the preconditions are true. The *purpose* of an operator determines what goal the operator is supposed to achieve. An operator may have an optional *start time*, which SIPE uses to support primitive temporal planning. A operator may have a *plot* (refinement) associated with it. The plot contains step by step instructions for the action represented by the operator. A plot is a procedural network that consists of the following node types: process, choice process, goal node. Process nodes apply a given more concrete operator. Choice process nodes select one operator from a set and apply it. Goal nodes require the achievement of a given predicate. This is SIPE's mechanism for subgoaling on preconditions as in means-ends analysis. SIPE supports conditional effects through a deductive causal theory. The deductive theory greatly simplifies the specification of complex domains.

3.1.6 Relaxed abstraction hierarchies: AbStrips, AbTweak

This section discusses a specific type of abstraction-based planning, relaxed abstraction hierarchies. Another type of abstraction, reduced abstraction hierarchies are discussed in the next subsection **??**.

The main motivation of any abstraction-based planner is to overcome the tyranny of detail that a planner that works only in ground plans is faced with. For example, when planning a trip from New York to Los Angeles, it seems reasonable to first create an abstract plan (go to the airport, take plane) and then refine this plan to more detailed levels (go to telephone, call taxi, and so on).

AbStrips

Sacerdoti extended the original STRIPS system to support abstractions in ABSTRIPS [**?**]. ABSTRIPS is similar to STRIPS, but creates abstract operators by dropping preconditions. Each predicate that occurs in the precondition of an operator is assigned a criticality value. For example, in the kitchen domain, the PICK-UP-FROM-DRAWER operator has the precondition (OPEN DRAWER). Dropping this literal results in an abstract operator that picks up an object from the drawer independently of whether the drawer is open or not. This abstract operator will be refined by either adding a step to open the drawer (if the drawer is closed) or by applying the primitive operator PICK-UP-FROM-DRAWER directly.

ABSTRIPS plans in a hierarchy of problem spaces. At abstraction level i, all preconditions of operators with criticality less than i are ignored. Removal of preconditions results in additional links in the original state space, since the abstract operator is applicable in a larger set of states. This type of abstraction is called *relaxed* abstraction.

The criticality levels in ABSTRIPS are first assigned by the user and

further refined by the system. The user guesses at the importance of different precondition literals and creates a partial order of criticality value for these literals. AbStrips then analyzes the precondition literals and changes them according to the following rules:

- a literal that can not be affected by any operator, is the assigned the maximum criticality value. These literals are called static literal.
- if AbStrips can find a short plan to achieve a precondition given that the other preconditions are true, this precondition is assumed to be a detail and is assigned a criticality value equal to the one assigned by the user.
- if AbStrips can not find a short plan, the precondition is assigned a criticality value higher than any value in the partial order.

The resulting abstraction hierarchy is based on the difficulty in achieving a predicate and consists of three classes, static predicates at the highest level, predicates for which no short plan can be found are next, and lastly details (predicates for which a small plan can be found).

The original work [?] did not include an analysis and some parts of the algorithm were unclear. Knoblock further analyzed AbStrips to find conditions for improvement [?]. The work shows that since all preconditions are tested in isolation, AbStrips is based on the implicit assumption that the details are independent. Furthermore, since predicates are only removed from the preconditions, and not from the goals, it is possible that AbStrips will create an incorrect plan. When adding AbStrips to the Prodigy planning system, performance is much worse than that of Prodigy alone (roughly by a factor of six with respect to solution time). The combined Prodigy and AbStrips system did however reduce the length of the generated plans.

AbTweak

Yang added hierarchical planning to the partial-order planner TWEAK. As with the original TWEAK system, ABTWEAK's main goal was to provide a formalization of hierarchical partial-order planning and to prove completeness of the resulting planning system.

3.1.7 Reduced abstraction hierarchies: The Alpine System

Knoblock developed ALPINE, an extension to the PRODIGY system that automatically creates abstraction hierarchies based on the *ordered monotonicity* property [**?**]. Knoblock shows that ordered monotonicity can reduce the complexity of planning to be linear in the length of the solution under certain assumptions. These assumptions are true for example in the towers of Hanoi domain. The critical assumptions in the analysis are: that the abstraction hierarchy divides the problem into roughly equal sized subproblems, and that the solution found by ALPINE is the optimal solution.

Each abstract problem space is created by dropping predicates from the problem description (preconditions and effects of operators), so called *reduced abstraction*. This is the major difference between ALPINE and ABSTRIPS. ALPINE's reduced abstraction removes predicates from preconditions, effects, states, and goals, whereas ABSTRIPS removes predicates from preconditions only. Another difference is that ALPINE's abstraction hierarchies are problem dependent (there is a different abstraction hierarchy for each problem), in contrast to ABSTRIPS's domain dependent hierarchies (all problems in the domain have the same abstraction hierarchy).

The ordered monotonicity property guarantees that a literal introduced at a level will not be changed by refining the abstract plan at a lower level. For example, if an abstract plan adds the literal (IS-ON DISK3 PEG1), the large disk can not be moved during refinement of the abstract plan to lower

abstraction levels.

A comparison between ALPINE and ABSTRIPS in the STRIPS robot domain shows that ALPINE improves the performance of PRODIGY roughly by a factor of three [?].

3.1.8 Case-based planning: Chef

This subsection discusses Hammond's CHEF system [?]. Section ?? is a brief example of case-based planning in the kitchen domain.

CHEF is a case-based reasoning system that generates new recipes in the szechuan kitchen domain. Instead of generating a new plan from scratch, CHEF retrieves a previous plan and adapts it to the new task. CHEF first retrieves the most similar plan from its plan memory using a domain-dependent similarity metric. Although a case-based planner will usually contain many different plans in its memory, the retrieval phase is assumed to be cheap. The cost of case retrieval can be reduced by hash tables or discrimination nets.

The retrieved plan is then superficially adapted to the new situation by substituting variables. An execution module simulates the plan and detects possible failures. CHEF analyzes the failures to determine why a given operator failed and what goal it was trying to achieve. Each failure is associated with a *thematic organizations packet*. These TOPs associate a plan failure with a domain-independent repair strategy. CHEF uses TOPs to anticipate failure and when learning from failure. The TOPs are also used to provide a selection of repair strategies.

The set of repair strategies consists of 17 different methods:

1. Replace an operator by one that achieves the same goal, but does not have the same side-effect (ALTER-PLAN:SIDE-EFFECT).

2. Replace an operator by one that is missing a given pre-condition, but achieves the same goals (ALTER-PLAN:PRECONDITION).

3. Add an operator to re-establish a deleted side-effect before it is needed (RECOVER).

4. Reorder the running of two steps (REORDER).

5. Increase the balance of two parts, e.g. water and flour (ADJUST-BALANCE:UP).

6. Decrease the balance of two parts. (ADJUST-BALANCE:DOWN)

7. Add a concurrent plan step that removes an unwanted side-effect while it is created (ADJUNCT-PLAN:REMOVE).

8. Add a concurrent plan step to provide a missing pre-condition (ADJUNCT-PLAN:PROTECT).

9. Split one operator into two operators that achieve the same goal (SPLIT-AND-REFORM).

10. Increase the duration of a plan step (ALTER-TIME:UP).

11. Decrease the duration of a plan step (ALTER-TIME:DOWN).

12. Replace an object by one that has all desired but none of the undesired features (ALTER-ITEM).

13. Replace a tool by one that has no undesired side-effect (ALTER-TOOL).

14. Move an operator before some operator (ALTER-PLACEMENT:BEFORE).

15. Move an operator after some operator (ALTER-PLACEMENT:AFTER).

16. Add a step that changes an undesired attribute to a desired one (ALTER-FEATURE).

17. Add a step to remove an undesired attribute (REMOVE-FEATURE).

The order of repair strategies depends on domain-dependent and domain-independent control rules. For example, one rule states that adding a preparation step is easier than adding a cooking step. Another one states that it is generally easier to add a single operator instead of a sequence of operators.

CHEF's plan representation language is more expressive than the standard STRIPS representation. It supports primitive temporal planning; operators in a plan have a duration, and may be executed concurrently. Furthermore, it supports a more powerful object representation: objects have attributes (so-called features) and have specific types. Furthermore, CHEF distinguishes between ingredients and tools to reduce the bindings and plan adaptations.

3.1.9 Macro-operators

Almost as old as planning itself is the idea to speed it up by learning and exploiting domain features. For example, the CHEF system described in subsection ?? learns from previous planning episodes.

Another popular method is to extract subsequences from successful plans, generalize them, and add them to the operator set. The resulting new operator is called a macro operator or macro.

There are many different methods, however, for generating new macro-operators. Approaches range from domain compilation to event driven learning. Different macro-learning methods are discussed in section ??. This section focuses on the planning process, that is how are newly generated macros used when solving a new problem.

The most popular method is to add the new macro to the operator set [?, ?, ?, ?, ?, ?, ?, ?, ?]. When solving the next problem, the planner uses the macro identically to a primitive operator.

Other planners ([?, ?, ?]) create an abstract search space that consists of only macros. The planner first finds a prefix plan to reach a state in the

macro-space from the initial state. Then it finds a solution in the macro-space. Lastly, it finds a suffix plan from the final state in the macro-space to the original goal. This method is similar to abstraction-based planning discussed in subsection **??**, with the main difference that the solution in the macro-space implies a solution in the ground space and that there is no need to search for the ground solution.

3.2 Planning as plan space search

Korf has previously analyzed the planning problem as a state space search problem [**?**]. Starting from the initial state, operators are applied to create new states. In this framework, the planning problem is identical to a graph search problem: to find a path from the initial state to a state that satisfies the goal predicate. The nodes of the graph correspond to possible world states and the arcs to application of operators.

This framework does not lend itself to the analysis of more powerful planning algorithms (such as partial-order, abstraction-based, or case-based planning), because these strategies reason about sets of states and transitions between them, rather than individual states. These planning strategies, however, are essential to simplify planning in richer domains, for example, the kitchen domain described in section **??**.

Therefore, this section describes the planning process as search through the space of possible plans. This paradigm was developed from the plan space search paradigm generally used to describe partial order planners. This thesis, however, adds extensions to include other planning systems, such as abstraction-based, case-based, and macro-based planning.

The planner maintains a set P of possible candidates for successful plans. This plan set initially contains only the "null" plan. The planner repeatedly executes the following loop until either a solution is found or no more plan candidates are available:

1. First a possible candidate p is retrieved from P and tested. If p is a solution to the problem, the planner returns p. If there are no more candidates, the planner signals failure. The plan set initially contains the "null" plan.

2. If p is not a successful plan, a plan transformation t from a set of possible transformations T is applied yielding a new candidate c. More than one plan transformation may be applied or a plan transformation may generate more than one new candidate. The resulting candidates are collected in the new candidate set C. The new candidate set C may be empty, if there are no more applicable transformations to the current plan p in the set of transformations T.

3. The new candidate set C is added to the plan set, yielding a new P. Goto step **??**.

In this framework, a domain independent planner is defined as follows:

Definition 1 (Planner) *A **planner** $\mathcal{P}$ is a tuple* $(\mathcal{L}_\mathcal{S}, \mathcal{L}_\mathcal{G}, \mathcal{L}_\mathcal{O}, \mathcal{L}_\mathcal{P}, \mathcal{T}, \mathcal{M})$.

- $\mathcal{L}_\mathcal{S}$ *is the state language, describing possible situations. A situation is a snapshot of the world. A common state language* $\mathcal{L}_\mathcal{S}$ *consists of sets of predicates, describing the true assertions in the situation.*

- $\mathcal{L}_\mathcal{G}$ *is a goal description language. An expression in* $\mathcal{L}_\mathcal{G}$ *assigns a truth value to expressions in the state language* $\mathcal{L}_\mathcal{S}$*. If a state expression satisfies* $\mathcal{L}_\mathcal{G}$ *it is called a goal. Any plan that reaches a goal from the initial state is called a solution.*

- $\mathcal{L}_\mathcal{O}$ *is a description language for operators, i.e., actions for the agent to affect the world. An operator* O *is a function that takes a state expression* $s \in \mathcal{L}_\mathcal{S}$ *as its input and results in a new state* $s' \in \mathcal{L}_\mathcal{S}$*. Application of an operator* O *results in the special state* (NIL)*, if the operator is not applicable in state* s*.*

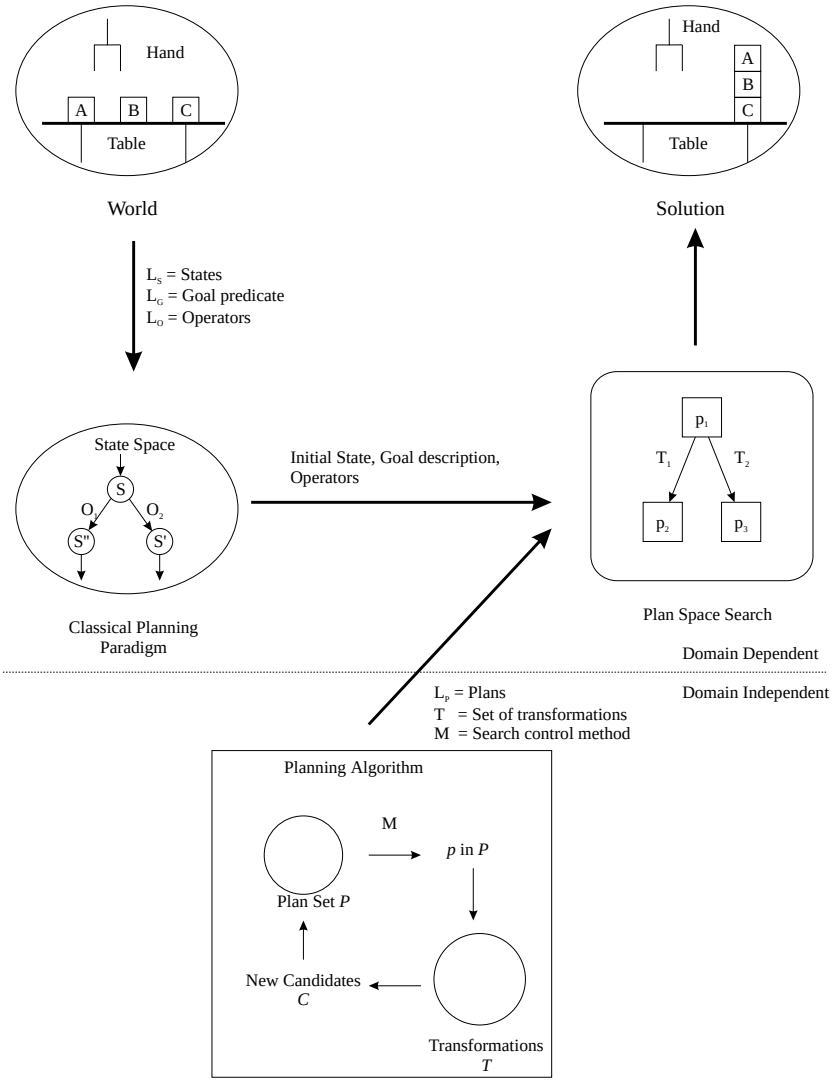
- *$\mathcal{L}_\mathcal{P}$ is the plan language, the language describing evolving plans. The plan language must be able to express plans, that is a set of operators, an ordering on the operators in the set, and a set of constraints on variable instantiations. Early planning systems supported only totally ordered, fully instantiated plans, that is the ordering imposed on the operators is a total order and the variable constraints only supported the instantiation of variables. More recently, partial-order planners support partially ordered plans with co- and non-codesignation constraints for variable instantiations.*

- *T is a set of transformations. A transformation t is a function that takes a plan expression from $\mathcal{L}_\mathcal{P}$, and returns a new candidate plan c expressed in $\mathcal{L}_\mathcal{P}$.*

- *M is the plan selection method. Given a set of possible plans P expressed in $\mathcal{L}_\mathcal{P}$, M selects the plan p to be tested next.*

Figure ?? is a graphical representation of the plan space paradigm. First, the world is represented in the representation language of the planner ($\mathcal{L}_\mathcal{S}$, $\mathcal{L}_\mathcal{G}$, $\mathcal{L}_\mathcal{O}$). The initial state is expressed in the state language $\mathcal{L}_\mathcal{S}$, the goal states are represented through a goal expression from $\mathcal{L}_\mathcal{G}$, and the actions are expressed as operators in $\mathcal{L}_\mathcal{O}$. This domain dependent reformulation converts the problem into a state space search problem. The initial state, goal description and operators are combined with the planning algorithm to reformulate the state space search into a plan space search. The planning algorithm attempts to find a sequence of transformations on expressions from $\mathcal{L}_\mathcal{P}$ to find a plan that solves the problem. It defines the plan space search by defining the nodes (expressions in $\mathcal{L}_\mathcal{P}$) and transitions between nodes (transformations from T). Once a solution is found, it is translated into a world solution by executing the plan. Before execution, it may be necessary to compute a total ordering of operators from the partial order and an instantiation of variables that satisfies the constraints.

Figure 3.1: Planning as plan space search



To further simplify the classification of different planning systems, the definition of the planner is broken up into two parts: the representational classification (consisting of $\mathcal{L}_{\mathcal{S}}$, $\mathcal{L}_{\mathcal{G}}$, $\mathcal{L}_{\mathcal{O}}$) and the operational classification (consisting of $\mathcal{L}_{\mathcal{P}}, \mathcal{T}, \mathcal{M}$). The representational classification determines the translation of a real world domain into a state space search, and the operational classification determines the search space.

3.3 Representational Classification

The state $\mathcal{L}_{\mathcal{S}}$, goal $\mathcal{L}_{\mathcal{G}}$, and operator $\mathcal{L}_{\mathcal{O}}$ languages determine the description of a domain. They determine the presentation of the input to the planner. The following subsections describe the STRIPS representation language and the PRODIGY description language (PDL), a more powerful domain description language, which is used by PRODIGY and DOLITTLE. As most modern planning systems, PRODIGY and DOLITTLE still use a variation of the STRIPS representation language, but support conditional effects and quantified variables.

3.3.1 The Strips domain language

STRIPS's state description language describes states as sets of non-negated predicates with instantiated arguments. There are no variables and no negated predicates allowed in a state description. The predicates form an implicit conjunction. For example, the following is a description of a possible state in the blocksworld with three blocks:

Example: STRIPS representation of a state in the blocksworld
State $S = \{$(ON A B),
(ON B C),
(ON C TABLE),
(CLEAR A),
(HAND-EMPTY) $\}$

It describes a state in which the three blocks A, B, and C are stacked to form a tower. To reduce the length of a state description, STRIPS makes the closed world assumption: Predicates that are not mentioned in the state description are assumed to be false. Therefore, there are no negated predicates in a state description. Given the state expression above, it follows that all other predicates, for example (ON B TABLE), are false.

The goal language is a set of predicates, that may contain free variables and negated predicates. The free variables are implicitly existentially quantified. A goal predicate G is true for a state if there is an assignment to each free variable, such that the state contains all positive predicates and none of the negated predicates in G. The following expression is a goal description that is satisfied by state S.

Goal G = { (ON $V1 B), (NOT (ON B TABLE)) }

In STRIPS operators are represented by three lists: the precondition-, add- and delete-list. For example, the following is a STRIPS operator in the blocksworld domain that picks up one block from another.

PICKUP-FROM-BLOCK($B1, $B2)
Preconds. { (HAND-EMPTY), (CLEAR $B1)
(ON $B1 $B2) }
Add-list { (CLEAR $B2) (HOLDING $B1) }
Del-list { (HAND-EMPTY) (ON $B1 $B2) }

The preconditions of an operator are a set of possibly negated predicates that may contain variables. The operator is applicable in a given state, (i.e., does not result in the (NIL) state), if and only if the preconditions match the state. Free variables in the preconditions are implicitly existentially quantified. Therefore, the operator UNSTACK in the previous example is applicable in the state S, if ($B1 = A) and ($B2 = B). The process of assigning atoms to variables is called *binding*. A binding must assign constants to all free variables in the preconditions.

The add and delete lists of an operator are also called its effects. All free variables that only occur in the effects of an operator are assumed to be universally quantified. These variables are referred to as *wildcards*, since they may match any object in the domain. An example of the use of wildcards is shown in the description of the PRODIGY domain language in the next subsection (subsection **??**).

The effects of an operator consist of lists of predicates that have either atoms or variables as arguments. Existentially quantified variables that occur in the effects of an operator must be mentioned in its pre-conditions. The predicates in the add-list are bound using the binding suggested by the preconditions and added to the current state description. Then predicates in the delete list are removed from the state description respectively.

For example, applying operator UNSTACK to the state S results in the new state S':

State $S' = \{$(CLEAR B), (HOLDING A), (ON B C), (ON C TABLE)$\}$

This state language and operator language is adequate for small domains and serves as a useful example. Many people have extended the standard STRIPS language to include conditional effects, universal and existential quantification of variables, and other extensions to simplify description of more complex domains. In section **??**, an example of a richer description language is provided. However, those extensions are still based on the STRIPS assumption and do not increase the representational power of the description language. In other words, it does not increase the set of domains that can be described, but it simplifies their description.

3.3.2 The Prodigy domain language

The Prodigy domain language includes a number of extensions to the STRIPS operator representation language to simplify description of more complex domains and to enhance the performance of the planner. The extensions include conditional effects, universal and existential quantification of variables, and a

type hierarchy for variables. Furthermore, it does not always make the closed world assumption. Using control rules, a user may encode domain specific problem solving knowledge to speed up planning.

The representation of states (the state description language $\mathcal{L}_S$) in PDL is similar to the STRIPS state description language. A state is a list of fully instantiated literals that form an implicit conjunction. See the previous subsection for an example. An important difference is that PDL supports two types of predicates, closed-world and open-world. An *open world* literal has three instead of two possible values: *true*, *false*, or *unknown*. An open world predicate is *unknown* if it is absent from a state description. Therefore, a state description may include positive or negative open world literals.

PDL's goal language $\mathcal{L}_G$ extends the STRIPS representation by allowing variables to be existentially or universally quantified. The following expression defines a goal in the blocksworld:

Goal = (EXISTS $B1 (FOR-ALL$B2 (NOT (ON $B2 $B1))))

The goal is satisfied in all states, with a block that has no blocks on it.

PRODIGY allows two different types of actions: operators and inference rules. Operators are similar to STRIPS operators with preconditions and effects (add-/delete-lists). PDL also supports universally quantified variables and conditional effects in operators.

Inference rules are syntactically identical to operators, but have slightly different semantics. Whereas the literals in the effect of an operator are assumed to be closed world literals, the literals in the effects of an inference rule are assumed to be open world literals, that is the absence of such a literal in a state description does not imply that its negation is true.

PRODIGY4 also supports typed variables. A designer may specify a static type hierarchy for objects in the domain. In general, variable types are finite, but PRODIGY4 supports a special *INFTYPE* variable type that may be used to describe variables that range over an infinite set, e.g., integers. *INFTYPE* variables must be used with a generator function, that is a function that for

an unbound variable returns a "reasonable" set of bindings.

The PRODIGY4 domain description language also allows a user to specify domain specific knowledge as control rules. Control rules affect the search of the planner and thus are not part of the representational classification itself. PRODIGY's means-ends analysis algorithm contains five different choice points: selection of (a) a node, (b) a goal, (c) a relevant operator, (d) a set of bindings for the operator, and (e) whether to apply the operator or subgoal on its preconditions. A control rule may select, reject, or prefer one choice over another for any of the four possible choice types. The applicability conditions for a control rule are restricted formulae of meta-predicates, such as (CURRENT-GOAL) or (TRUE-IN-STATE). The user can specify meta-predicates using arbitrary lisp code.

The following table (table **??**) is a kitchen domain example of complex operator descriptions in the prodigy domain language.

The operator PICKUP illustrates the most important aspects of PRODIGY's domain description language. First, it shows the use of typed variables and the syntax for variable descriptions. PICKUP defines two variables $OBJECT and $LOCATION respectively. $OBJECT must be a movable item and $LOCATION must be of type location. The free variable $OBJ is in the effects, and is thus implicitly a universally quantified variable. This behavior is an extension of STRIPS' wildcard feature, since variables may make use of the type-hierarchy. It also shows the use of conditional effects. If any object, movable or stationary, is next to the object to be picked up, it is not next to $OBJECT after application of PICKUP.

The inference rule INFER-ARM-FULL is an example of a rule that infers an open world predicate and adds it to the state description. The absence of (ARM-FULL) does not automatically imply that it is false in the current state. An inference rule has an associated mode, EAGER or LAZY, which corresponds to eager or lazy evaluation of the inference rule. Eager inference rules are evaluated for every state, whereas lazy inference rules are only

Table 3.2: Example: Operator in the prodigy domain language

```
Operator: PickUp
        Params $object $location)
        Preconds ($object movable-items) ; typed vars.
                 ($location location)
                 (and (arm-empty)
                      (is-at robby $location)
                      (is-reachable $object $location)
                      (clear $object)
        Effects ($obj2 (or movable-items stationary-items)
                (del (arm-empty))
                (del (is-reachable $object $location))
                (add (holding $object))
                (if (next-to $obj2 $object)
                   (del (next-to $obj2 $object)))

Inference-Rule: Infer-Arm-Full
        Mode EAGER/LAZY
        Params:
        Preconds (Exists $obj (holding $obj)) )
        Effects (add (arm-full))

Control-Rule: Make-Instant-Coffee
        IF (and (candidate-goal (have-coffee))
                (not (coffee-can-available))
        THEN (select operator Make-Instant-Coffee)
```

evaluated if they contain necessary effects.

The control rule MAKE-INSTANT-COFFEE forces PRODIGY to make instant coffee if no coffee-can is available, presumably, because a plan using the coffee-maker will eventually fail.

3.4 Operational classification

The operational classification is determined by three components: the plan description language $\mathcal{L}_{\mathcal{P}}$, the method M to select a plan from the candidate set, and the set of plan transformations T. The representational classification influences the operational classification, because the plan language has to support the semantics of the representational classification. However, the operational classification strongly influences the search process. Therefore, in the following subsections, this thesis focuses on the comparison of different planning systems with respect to their operational classification. Because of the enormous number of planning systems, the comparison was based on an archetypical example of a given problem solving strategy, for example TWEAK for partial-order planners. A list of the compared systems is shown in table ??, summarizing the plan languages $\mathcal{L}_{\mathcal{P}}$ and the transformation sets T. There are at least four popular search methods M: depth-first, breadth-first, iterative deepening, or best-first. Although there are subtle influences that the search method has on the plan language, the comparison in the table ignores M. Also included in the table is the multi-strategy planner DOLITTLE. Details about DOLITTLE's design are in chapter ??. Partial-order planning is discussed in this section for completeness, but DOLITTLE currently does not support it. The reasons for leaving out partial-order planning are discussed in section ??.

Table 3.3: Operational classification of different planning systems

Planner	Plan lang. $\mathcal{L}_\mathcal{P}$	Transform. set T
Forward Chaining	Total order Instantiated variables Plan head	Append to plan head Advance current op.
Means-ends	Total order Instantiated variables Plan head and plan tail	Append to plan head Prepend to plan tail Advance current op.
Case-based CHEF	Total order Instantiated variables Plan skeleton Concurrent plans	Insert operator Remove operator Reorder operator Replace operator Change var. bindings Move current op.
Auto. subgoals STEPPING STONE Relaxed Abstraction	Total order Instantiated variables Uniform trees Plan head and plan tail	Append to plan head Prepend to plan tail Advance current op. Create probl. space
Abstraction ALPINE	Total order Instantiated variables Uniform trees Plan head and plan tail	Append op. at level i Prepend op. at level i Advance curr. op. (i) Create probl. space $(i+1)$
Macros MACLEARN	Total order Instantiated variables Plan head	Append op. sequence Advance current op.
Multi-strat. DOLITTLE	Total order Instantiated variables Plan skeleton Non-uniform trees	Move current op. Insert op. sequence Remove, -order, -place ops. Change var. binding Create problem space
Partial-order TWEAK	Partial order Constrained variables	Add operator Add variable constraint Add operator ordering

3.4.1 Forward chaining planning

This subsection investigates a simple planning system FC-PLANNER that uses forward chaining. $\mathcal{L}_{\mathcal{S}}$, $\mathcal{L}_{\mathcal{G}}$, and $\mathcal{L}_{\mathcal{O}}$ are as described in subsection ?? above.

The planner repeatedly picks an applicable operator from the operator set and applies it to the current state. Application of this operator results in a new current state.

The plan language $\mathcal{L}_{\mathcal{P}}$ represents totally ordered sequences of instantiated operators. There can be no free variables in an evolving plan. Also, for efficiency reasons, the plan language does not allow plans that contain state loops, that is plans that visit the same state twice. If a plan that contains a state loop is a solution, then in the classical planning paradigm, there is an equivalent plan with the state loop removed that is also a solution.

The plan transformation set T contains one method: appending a fully instantiated operator to an evolving plan. Although there is only one plan transformation method, there is possibly more than one successor plan, since there may be more than one possible operator that is applicable, or more than one variable binding for an operator.

M picks a plan from the plan set using a depth first traversal, that is the children of a candidate plan are picked from left to right. A planner using breadth-first search or best-first search can be created by changing M only.

3.4.2 Means-ends analysis

STRIPS uses means-ends analysis, a search control method developed by GPS. In contrast to forward chaining planning, means-ends analysis is a backpropagation method. The differences between the current state and the goal are computed. Then a *relevant* operator, that is an operator which removes the difference is instantiated and added to the plan. There are two possibilities for the addition of the operator: (a) add and apply the operator

as the last operator of the plan head, or (b) add the operator as the first operator of the plan tail. Adding and applying the operator in (a) leads to a new current state, whereas in (b) the current state remains unchanged. Additionally, applying method (a) can only be used if all preconditions of the operator are satisfied.

Partial plans (expressed in $\mathcal{L}_{\mathcal{P}}$) of a means-ends analysis planner consist of a plan head and a plan tail. Both plan sequences contain fully instantiated, totally ordered operators. Plans including state loops and goal loops are discarded. A goal loop exists, when an operator used to achieve a goal literal requires achievement of the same goal literal to satisfy its preconditions.

There are two plan transformation methods in the transformation set T: (a) as in forward chaining planning, an instantiated operator can be appended to the plan head, or (b) a new relevant operator can be prepended to the plan tail. The main differences between PRODIGY2 and PRODIGY4 are the details of the implementation of method (b) in the transformation set T. In PRODIGY2 the transformation set T only supports addition of an operator that is relevant to an unsatisfied precondition of the first operator in the plan tail, whereas PRODIGY4's transformation method supports addition of an operator relevant to any unsatisfied precondition in the plan tail.

3.4.3 Partial-order planning

A partial-order planner, for example TWEAK, supports a more powerful plan representation language. The plan representation language $\mathcal{L}_{\mathcal{P}}$ supports a partial order of an operator set and usually constraints on variable bindings such as co-designation and instantiation.

The plan transformations in T allow addition of an operator to a plan, adding an ordering constraint for an operator, and adding a binding constraint.

3.4.4 Case-based planning

A case-based planner, such as CHEF, retrieves a similar plan and adapts it to the new problem. The plan language $\mathcal{L}_\mathcal{P}$ supports totally ordered, fully instantiated sequences of operators.

The set of plan transformations T contains many transformation methods. In contrast to other planners, the transformations are highly specialized. For example, an operator in CHEF may only be replaced by another, if it either removes an unsatisfied precondition or an unwanted side-effect. The plan is not divided into a plan head and plan tail, a case-based planner may make changes to the plan anywhere in the operator sequence.

3.4.5 Automatic subgoaling

This subsection describes automatic subgoaling as exemplified by STEPPING STONE in the plan space paradigm [?]. Automatic subgoaling creates a series of goal expressions (descriptions of states, subgoals) that when traversed lead an agent to the goal. Each subgoal generates a subproblem. The initial state of the subproblem is the goal state of the previous subgoal, and the goal is to reach a state satisfying the current subgoal.

The plan language $\mathcal{L}_\mathcal{P}$ describes a plan at two distinct levels. Since preconditions and goal expressions both identify sets of states, the top level can be interpreted as abstract operators describing a series of subgoals, and the bottom level contains sequences of primitive operators that solve the associated subgoals.

STEPPING STONE uses a means-ends analysis planner to solve the underlying subproblems. Therefore, T contains the means-ends plan transformations as well as one additional transformation to create a new subproblem and solve it. The subproblems are generally solved in left to right order.

Some automatic subgoaling systems are based on the serial subgoal assumption. This means that the resulting subproblems are constrained to

disallow all plans that undo previously achieved subgoals.

Automatic subgoaling is similar to abstraction-based planning, in that both create a series of subproblems. The main difference is that abstraction-based planning searches for a solution to the "abstract" problem, whereas automatic subgoaling does not need to search for a solution to the abstract problem. A relaxed abstraction is represented by an operator that has some preconditions removed. The sequence of abstract operators generates a sequence of subgoals. Therefore, the plan language and set of plan transformations of a relaxed-abstraction based planner is similar to that of an automatic subgoaling planning system. The only difference is that a relaxed abstraction-based planner may contain trees with any number of levels, as opposed to automatic subgoaling which contains two levels.

3.4.6 Abstractions

This section discusses reduced abstraction-based planning (ALPINE) in the plan space paradigm. A reduced abstraction-based planner simply creates a series of constrained problem spaces. It requires a separate planner to solve the individual problems. ALPINE uses PRODIGY as its underlying problem solver.

A partial plan expression in $\mathcal{L}_{\mathcal{P}}$ in ALPINE represents a tree of abstract or primitive operator sequences. There is one level in the tree for every abstraction level. The tree is uniform, i.e., all subtrees at one level have the same number of levels. The representation of operator sequences is based on the representation of the underlying planner (PRODIGY). Additionally, an abstract operator at a level corresponds to a constrained subproblem at the next lower abstraction level.

T is a superset of the transformation set of the underlying planner, in this case PRODIGY. Additionally to PRODIGY's transformation set discussed in subsection **??**, ALPINE's transformation set includes a transformation to create an operator's associated subproblem. In ALPINE, this transformation

is limited to a left to right traversal. Other planning systems, for example SIPE, allow the generation of subproblems in any order.

3.4.7 Macro operators

Since macros are compiled sequences of primitive operators, in general the plan language $\mathcal{L}_\mathcal{P}$ and the set of transformations T are dependent on the underlying planner. In Iba's MACLEARN system, the underlying planner is a simple depth-first planner, whereas most EBL-based macro-learners are based on a means-ends analysis planner. However, in both cases, the plan language $\mathcal{L}_\mathcal{P}$ and the transformation set T are generalized to handle sequences of operators, instead of single operators only.

3.5 Discussion

This section summarizes the results of the comparison of different planning systems. The chapter describes a collection of important and popular planning systems (GPS, STRIPS, PRODIGY, NOAH, SIPE, ABSTRIPS, ALPINE, CHEF, and MACLEARN). The idea is that these planning strategies have been used with success previously and therefore should be included in a multi-strategy planner.

The plan space paradigm is introduced because the popular state space paradigm is unable to represent many of the planning systems. The plan space search paradigm is commonly used to describe partial-order planners but is extended here to cover also macro-based, abstraction-based, and case-based planning. The plan space search paradigm distinguishes between the representational and operational classification of a planner.

The operational classification of a planner allows us to create a workable definition of a planning strategy: It is a plan language, a set of transformations on the expressions in the plan language, and a search method. This

allows the comparison of different planning strategies. The results of this comparison are summarized in table **??**.

From this comparison, a plan language $\mathcal{L}_{\mathcal{P}}$ and a set of plan transformations T powerful enough to be able to use the described planning strategies (except for partial-order planning) is developed in chapter (chapter **??**).

Chapter 4

Multi-strategy planning

> The world can doubtless never be well known by theory: practice is absolutely necessary; but surely it is of great use to a young man, before he sets out for that country, full of mazes, windings, and turnings, to have at least a general map of it, made by some experienced traveler.
>
> **Lord Chesterfield** (1694-1773), English statesman, man of letters. *Letter, 30 Aug. 1749* (first published 1774; repr. in *The Letters of the Earl of Chesterfield to His Son*, vol. 1, no. 190, ed. by Charles Strachey, 1901).

This chapter introduces multi-strategy planning and develops a framework for the comparison of different multi-strategy planning systems. It then analyses some of the planning systems described in the previous chapter with respect to this framework. This leads to an analysis of an example (based on the one shown in the scenario in chapter **??**), which shows why multi-strategy planning can outperform other planners.

The analysis of the example shows that under certain conditions the worst case complexity of a multi-strategy planner based on case-based, macro-based, and abstraction-based planning is better than that of the respective single strategy planners. This means that this set of conditions describes

sufficient conditions under which a multi-strategy planner has a lower worst case time complexity than the compared single strategy planners.

Section ?? summarizes previous complexity results for planning. Section ?? introduces the concept of a planning bias and planning strategy. An analytical model for a planning strategy is proposed in section ??. A framework for multi-strategy planning is developed in section ??. Section ?? analyzes different planning strategies introduced in chapter ?? with respect to the multi-strategy framework. Section ?? derives conditions under which a multi-strategy planner can produce an exponential speed up over three single-strategy planners: abstraction-based, macro-based, and case-based planning.

4.1 Analysis of practical planning

Theoretical results show that the general planning problem[1] is very hard. Theoretical results about the complexity of planning can be grouped into two different classes: (a) results about the complexity of some particular domain, and (b) results about the complexity of particular representation languages (i.e., classes of domains). In the most general case the planning problem is undecidable [?, ?]. The proof is based on the equivalence of a planning problem to the halting problem of a turing machine or a logic program.

Some researchers have investigated the complexity of specific domains. The results show that even for some toy domains, the problem of finding an optimal solution is NP-hard. Examples are the blocksworld ([?, ?]), or the tile sliding puzzle [?].

More recently, researchers focused on the effect of the representation language on the complexity of planning. This allows us to focus on the complexity of classes of domains. Bylander investigates the complexity of planning

[1] Theoreticians use the term *problem* to refer to the general problem under investigation. In planning research, a problem is a specific operator set, initial state, and goal state. This would be referred to as a *problem instance* by theoreticians. This thesis uses planning problem to refer to the general problem and problem to mean a problem instance.

using the STRIPS representation and a STRIPS representation with a deductive theory [**?**, **?**]. Bylander also shows that planning is PSPACE-complete, even if each operator in the operator set is limited to a maximum of two preconditions and two effects. If the operators only contain one positive precondition and two effects, planning is NP-complete.

Erol et al. show that planning is EXPSPACE-complete for operators with delete lists given a function free description language with a fixed number of constant symbols (datalog language, e.g., the STRIPS representation language) [**?**]. If there is an infinite number of ground terms (e.g., functions are allowed), planning is undecidable. A problem belongs in the complexity class EXPSPACE if the space required to solve the problem grows at most exponentially in the length of the input. The complexity class 2EXPTIME is the set of general problems with a time complexity that grows at most exponentially with an exponential function of the input size (doubly exponential).
Since the following inequality holds,

$$\text{EXPTIME} \subseteq \text{EXPSPACE} \subseteq \text{2EXPTIME}$$

it follows that the time complexity of planning is between exponential and doubly exponential. Therefore, in the remainder of this chapter, the analysis is concerned with the algorithmic complexity of planning algorithms, in particular abstraction.

The worst case complexity of a planning algorithm A on a problem p using the plan space search paradigm described in subsection **??** depends on the branching factor b_p and the *decision length* l_p of the problem. The branching factor b_p is the maximum number of transformations applicable to a plan expression. The decision length l_p is the number of choices that a planner has to make to derive a solution from the null plan. In the plan space search paradigm introduced in chapter **??**, the only decision a planner makes is the selection of a plan transformation. Therefore, in the plan space search paradigm, the decision length is the number of plan transformations

necessary to derive a plan from the null plan. For example, for a depth-first and means-ends analysis planner, the decision length is equal to the number of primitive operators in the plan, i.e., the *solution length*.

The thesis defines the abbreviation Problems(b, l) to mean the set of problems that are in the domain and have a branching factor less than b and a decision length less than l. Using this abbreviation, the worst case time complexity of planning with parameters b and l is the maximum cost of solving any problem in the set of problems Problems(b, l).

$$\text{Problems}(b, l) = \bigcup_{p \in \text{Problems}} p \text{ such that } \begin{array}{l} b_p \leq b \\ l_p \leq l \end{array}$$

$$\text{Cost}(A, b, l) = \max\{\text{Cost}(A, p)\ p \in \text{Problems}(b, l)\} \tag{4.1}$$

The cost Cost(A, p) is the actual cost of solving problem p using planning strategy A. From the definition, it trivially follows that

$$\text{Cost}(A, p) \leq \text{Cost}(A, b, l) \forall p \in \text{Problems}(b, l)$$

The worst case cost of means ends analysis is bounded by:

$$\text{Cost}(\text{MEA}, b, l) \in O(b^l) \tag{4.2}$$

This is the size of a tree with depth l and a branching factor of b. A theoretical result shows that EXPTIME is a proper superset of EXPSPACE. This shows that an EXPSPACE-complete problem has a time complexity that is at least exponential in the input length and therefore establishes a lower bound on the time complexity of planning for any algorithm. Applying this result to means-ends analysis planning leads to the following equation:

$$\text{Cost}(\text{MEA}, b, l) \in \Theta(b^l) \tag{4.3}$$

Table ?? summarizes the results of an experiment conducted to find the average branching factor for various PRODIGY domains. PRODIGY was run on a selection of its standard domains as well as the kitchen domain. The calculation for the average branching factors included only successful problems with at least two steps in the solution, to rule out small problems with large branching factors. Without this omission, the reported branching factors would have been even higher. For each problem, the average branching factor was calculated, and the minimum and maximum branching factors for a domain were recorded. The average branching factor for a domain was calculated as the average of the branching factors in the corresponding problem set. Table ?? contains the names of the domains, the number of problems that were tested, the minimum, average, and maximum of the branching factor for the domain as well as minimum, average, and maximum of the solution lengths. The averages for the branching factors ranged from 1.24 (path-planning) to 3.55 (kitchen). The average solution length of the domains varied from 3.47 to 62.40. Other researchers report similar results for other problems with branching factors of at least 3 [?].

What makes planning impractical is that the complexity can be exponential in the decision length. There are two possibilities to ameliorate this complexity: *concretization* and *abstraction*. Concretization attempts to decrease the branching factor b. Examples of concretization are means-ends analysis, partial-order planning, and Prieditis' concretization [?], which creates an abstract problem space by removing some operator choices. Abstraction aims at reducing the decision length, for example reduced or relaxed abstraction-based planning, automatic subgoaling, and macro-operators[2].

In the practical planning paradigm, typically (a) a solution includes hundreds of primitive operators, and (b) there are resource limits on the amount of computation that can be done in the planning stage. The practical plan-

[2]The term abstraction is overloaded in AI. This thesis uses abstraction as defined here and uses (relaxed/reduced) abstraction-based planning to refer to specific algorithms such as ABSTRIPS and ALPINE.

Table 4.1: Branching factor for PRODIGY domains

Domain	Problems	b (min/avg/max)	length l (min/avg/max)
blocksworld	15	(1.41/2.98/6.00)	(2.00/3.47/10.00)
eightpuzzle	12	(1.03/1.30/2.00)	(3.00/58.70/231.00)
eightpuzzle-II	6	(1.19/1.87/2.93)	(4.00/12.00/32.00)
extended-bw	12	(1.55/2.48/3.46)	(2.00/3.50/6.00)
extended-strips	5	(1.57/2.27/2.72)	(2.00/4.00/8.00)
gridworld	10	(1.07/1.34/2.00)	(3.00/21.30/75.00)
kitchen	5	(2.00/3.55/4.00)	(6.00/23.20/49.00)
multirobot	10	(1.37/1.67/2.45)	(2.00/5.80/10.00)
path-planning	5	(1.03/1.24/1.64)	(5.00/62.40/183.00)
schedworld	8	(1.33/1.75/2.24)	(4.00/6.50/13.00)
schedworld-II	11	(1.04/1.53/2.24)	(4.00/10.80/26.00)

ning paradigm describes the kind of problems an intelligent agent is faced with when operating in a complex domain, such as the kitchen domain or an information system. A first observation is that a better planning method must be used, since even a branching factor of 2 would lead to a complexity of 2^{100} for a plan with 100 primitive operators in it[3].

There are few problems in the PRODIGY test suite that fall into the practical planning paradigm. Only the eightpuzzle and path-planning domains include problems with more than 100 steps in the solution. These domains, however, have unusually small branching factors (branching factor less than 1.3).

Common resource limits include a time bound or a bound on the number of nodes that can be generated. Users would not be willing to wait indefinitely for a cup of tea. Given a resource limit, the lower limit of equation **??** can be reformulated to show the maximum decision length as a function of the

[3] 2^{75} is the estimated number of particles in the universe.

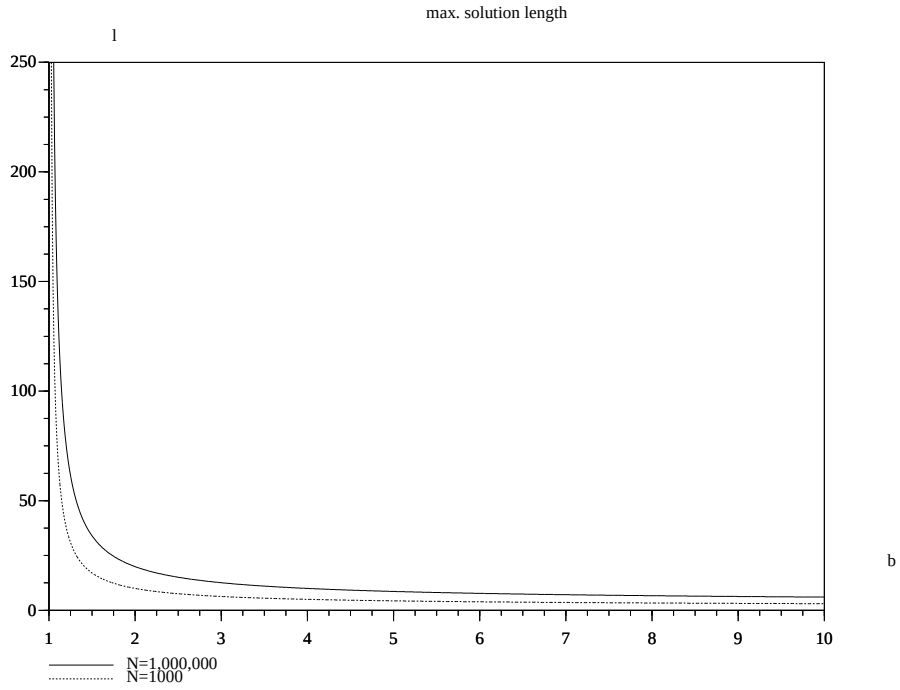
branching factor b and the resource limit N (ignoring a term $\log k$ for some constant k).

$$\text{max. dec. length } l \approx \frac{\log N}{\log b} \tag{4.4}$$

Figure ?? graphs this equation for different values of N. Note that for small values of b (less than 1.70, the maximum decision length drops rapidly and for values of b greater than 1.70, the maximum decision length is less than 30, even if the resource limit is $1,000,000$ nodes. The graph shows that the maximum decision length is largely insensitive to the specific resource limit and the specific branching factor. There is a difference of less than 10 in the maximum decision length between a resource limit of $1,000,000$ and $1,000$ nodes for any branching factor greater than 1.99. For example, a PC 486DX2-66 can solve around $100,000$ nodes in less than 1 hour. Furthermore, the difference in the maximum decision length between a branching factor of 2 and 10 is only 13 steps for a resource limit of $1,000,000$ nodes. Note that the maximum decision length for a branching factor of 2 and 10 with a resource limit of $1,000,000$ nodes is a factor of about 3.3. What is important to notice is that for reasonable practical values, this difference amounts to only 12 steps, since the maximum decision length decreases so quickly. Therefore, the decision length is the dominant factor. For practical planning ($l \geq 100$) within the resource limit $N = 1,000,000$, the branching factor must be less than about 1.15. In this case, there are few candidate transformations to a plan and the algorithm rarely has to make a choice.

Because of the dominant role of decision length l, most planning research has focused on abstraction and not concretization methods and this thesis will also focus on methods to reduce the decision length l. This does not mean that a reduction in the branching factor can not lead to substantial improvements, but that reducing the decision length by a constant factor leads to much larger improvement than reducing the branching factor by the same factor.

Figure 4.1: Max. decision length as a function of the average branching factor b and the node limit N



4.2 Planning bias

This section explains the notion of a planning bias. Similarly to the definition of inductive bias in machine learning [?], a *planning bias* is defined as follows:

Definition 2 (Planning Bias) *A **planning bias** is a set of assumptions with the aim of simplifying the current planning problem.*

As mentioned previously, the reduction of the search space is based on assumptions about the problem, such as the shape of the search space (heuristic function, strong linearity), the structure of new and previous solutions (case-based planning), the sequence of problems (easier to more difficult ones), etc.

A planning bias makes certain assumptions, that simplify planning. To improve planning performance, these assumptions must be exploited efficiently.

Definition 3 (Planning Strategy) *A **planning strategy** is a search control method that exploits a given planning bias by restructuring the search space and searches the associated space.*

One method of restructuring a search space is to cut out certain parts of the original search space, this is called restricting the search space. Another method (reordering) changes the order in which the space is searched by for example preferring plans with the least number of open goals. Restructuring, however, also includes methods that change the original search space into a different one. This is generally referred to as problem reformulation [?].

Note that a planning bias may lead to several different planning strategies. For example, in a domain where most failures are generated because of the wrong order of the operators (planning bias), partial order planning (planning strategy) or plan reordering (planning strategy) can be used to reduce the search. It is also possible that a planning strategy may arise

from different planning biases. For example, a system may generate macro-operators (planning strategy) to traverse difficult parts of the search space (planning bias) or to compile commonly used subtasks (planning bias).

For most planning systems, the underlying planning biases are implicit and hard to determine. For example, only recently have the planning biases of partial-order and total-order planning been investigated. Therefore, researchers focus on the planning strategies as opposed to the planning biases.

Different biases can be classified with respect to their effect on the search space. A *strong* bias is one that greatly reduces the search space; it makes strong assumptions. A *weak* bias makes few assumptions and hence does little to limit the search space. Of course, by cutting out large portions of the search space, there is a danger of cutting out the part containing the solution. A *correct* bias is one that does not prevent a system from finding the solution, otherwise a bias is *incorrect*. Since the search space for strong biases is small, testing a strong bias is cheaper than testing a weak bias. If a bias is incorrect, the system will have to search the complete associated search space, before being able to conclude that a solution can not be found.

The definition of a planning bias allows the distinction between underlying assumptions and their implementation in planning systems. Unfortunately, most often people refer to their implementation instead of the underlying assumptions. For example, cases and macros have similar representations, but are based on different assumptions about the domains. Even commonly used representations, such as macros, can be based on completely different assumptions. For example, the STRIPS macro learner generates new macros from successfully solved problems so that they may be reused as subtasks [**?**]. The MACLEARN system generates new macros using a peak to peak heuristic to avoid valleys in the heuristic evaluation function [**?**]. Korf designed a macro learner based on serial decomposability [**?**]. Although based on different assumptions, all systems are referred to as macro learners. This thesis uses the term macro to refer to an operator sequence, and macro-based planning

to refer to extraction of common subsequences from a successful plan. For example, Korf's MPS system generates macros, but is not a macro-based planner.

4.3 A model of abstraction

This section introduces a model of abstraction and analyzes the effect of abstraction on the complexity of planning. Korf used a similar analysis to discuss the effect of macro-operators and mentions the relationship to reduced abstraction-based planning [?]. Knoblock uses a similar analysis to show that the complexity of reduced abstraction-based planning, under certain assumptions, is linear in the solution length [?]. Other researchers showed a reduction in search cost by using a similar analysis if a problem can be divided into smaller subproblems [?]. The analysis in this section proposes a model of abstraction to include combinations of different planning systems based on the plan space search paradigm described in chapter ??.

Table ?? summarizes the symbols used in the analysis and gives a brief explanation of their meaning.

The analysis determines the worst case complexity of abstraction under a set of (best case) assumptions. Clearly, abstraction can not reduce the worst case complexity of the general planning problem. The previously mentioned theoretical results show that the worst case complexity of a planner is exponential in the length of the solution. If the assumptions are not met, the complexity of the planner is still exponential. The analysis is, however, important because it identifies a set of assumptions that, if valid, are guaranteed to reduce the complexity substantially over means-ends analysis. The analysis shows the reduced worst case complexity of a set of problems (the ones that satisfy the assumptions).

Abstraction transforms the original search problem into m_A smaller problems, with branching factors $b_1, b_2, \ldots, b_{m_A}$ and solution lengths of

Table 4.2: Table of Symbols

$A_1, A_2, \ldots$	problem solving strategies
b	average branching factor of the ground space
$b_1, b_2, \ldots$	avg. branching factors of the abstract subproblems
b_A	max. branching factor for abstraction A
D	decision procedure with cost $\text{Cost}(D, b, l)$
E, E_A	max. node expansion cost (for strategy A)
k	number of levels in abstraction hierarchy
l	decision length in the ground space
$l_1, l_2, \ldots, l_m$	decision length of the abstract subproblems
l_A	max. dec. length for subproblems generated by A
m	minimum of the reduction factors
m_A	reduction factor of planning strategy A
μ_A	applicability probability of planning strategy A
N	resource limit (e.g., time or node limit)
n	number of problem solving strategies
$r_A(l)$	blow-up function for an abstraction A
$\text{Problems}(b, l)$	problems in the domain with $b_p \leq b$ and $l_p \leq l$
$\text{Cost}(A, p)$	actual cost of planning strategy A on problem p
$\text{Cost}(A, b, l)$	worst case cost of A for $\text{Problems}(b, l)$
$\text{ECost}(A, b, l)$	expected cost of A for $\text{Problems}(b, l)$
$\text{Good}(A, b, l, \tau)$	problems in $\text{Problems}(b, l)$ that A reduces
$\text{Bad}(A, b, l, \tau)$	problems in $\text{Problems}(b, l)$ that A can not reduce
$\text{Certain}(A, b, l, D, \tau)$	problems that D classifies in $\text{Good}(A, b, l, \tau)$
MEA	means ends analysis
SUB	automatic subgoaling, reduction factor, m_S
A2 (AM)	abstraction-based planning, m_{A2} (m_{AM})
CBP	case-based planning, m_C, b_C, l_C
MAC	macro-based planning, $m_M, b + b_M, l_M$
PC-MSP-O	problem coordinated msp with oracle
PC-MSP	problem coordinated msp without oracle
2-PC-MSP	two planning strategy PC-MSP
O-SP-MSP	ordered subproblem coordinated msp
SP-MSP	unordered subproblem coordinated msp

$l_1, l_2, \ldots, l_{m_A}$. The planner then solves the individual subproblems. The solutions to the individual subproblems are combined to form a solution to the original problem. There is also a setup cost function c_i, that is the cost of setting up subproblem i. This setup cost is assumed to be constant for all problems. Therefore, the actual cost of abstraction for a problem p denoted by $\text{Cost}(A, p)$ is given in equation **??**, where E_A is the constant node expansion cost and $b_{i,p}$ ($l_{i,p}$) is the branching factor (decision length) of the i^{th} subproblem.

$$\text{Cost}(A, p) = \sum_{i=1}^{m_A} (c_i + E_A \, b_{i,p}^{l_{i,p}}) \tag{4.5}$$

The abstract algorithm may find a suboptimal solution. Therefore, we define a "blow up function" $r_A(l)$, which is the maximum ratio of decision length l' to optimal decision length l of abstraction A on any problem p in the domain. Therefore, $r_A(l)$ makes it possible to bound the length of the solution found by the abstraction. The following equation describes this relationship:

$$\sum_{i=1}^{m_A} l_i = l' \leq l \, r_A(l) \tag{4.6}$$

The following definitions of b_A, l_A, and c_A allow us to bound the cost of abstraction for all problems ($\text{Problems}(b, l)$) with a branching factor less than b and a decision length less than l:

$$\begin{aligned} b_A &= \max\{b_i \; \forall i \in 1 \ldots m_A, \forall p \in \text{Problems}(b, l)\} \\ l_A &= \max\{l_i \; \forall i \in 1 \ldots m_A, \forall p \in \text{Problems}(b, l)\} \\ c_A &= \max\{c_i \; \forall i \in 1 \ldots m_A, \forall p \in \text{Problems}(b, l)\} \end{aligned}$$

Then:

$$\text{Cost}(A, b, l) \leq \sum_{i=1}^{m_A} (c_A + E_A \, b_A^{l_A}) = m_A (c_A + E_A \, b_A^{l_A}) \tag{4.7}$$

For a given l', the upper bound on the cost will be minimized for a given m_A, if

all subproblems have roughly the same cost. Each subproblem's contribution to the complete solution only grows linearly in the length of its subproblem, but the work required to find it grows exponentially with the subproblem length. Therefore, if one subproblem is much larger than the remaining ones, its cost will overshadow any savings in the other subproblems.

Since the actual cost $\mathrm{Cost}(A, p)$ is strongly influenced by the length of the subproblem solutions l_A, the analysis makes the assumption that the subproblems have identical solution lengths. For example, any problem in the towers of Hanoi domain can be broken up, such that the sizes of the individual subproblems are identical [**?**]. Solving m_A subproblems of length l_A must result in a plan of at least length l. Using this assumption, it follows:

$$\begin{aligned} l_A &= l_i \; \forall i \in 1 \ldots m_A \\ m_A l_A &= l' \leq l \, r_A(l) \\ l_A &\leq \tfrac{l \, r_A(l)}{m_A} \end{aligned}$$

Replacing l_A in equation **??** yields the following equation.

$$\mathrm{Cost}(A, b, l) \leq m_A \, c_A + m_A \, E_A \, b_A^{\frac{l \, r(l)}{m_A}} \tag{4.8}$$

In the following subsection, the different planning strategies are analyzed with respect to their effect on the abstract branching factor b_A and the decision length l_A. It is assumed that the cost of setting up a subproblem c_A is minor compared to the search cost and that the abstraction finds the optimal solution ($r(l) = 1$). The analysis only concentrates on the search cost. The expensive part of planning is the search cost, not for example the cost of loading a problem or an operator set.

4.4 Multi-strategy planning framework

This section first defines multi-strategy planning and then describes a framework for the comparison of different multi-strategy planning systems.

Definition 4 (Multi-strategy planning) *A **multi-strategy planner** is defined as a planning system that coordinates a set of two or more planning strategies on a set of problems or on a single problem.*

Multi-strategy planning extends the set of possible planning strategies by allowing total and partial planning strategies. A *total* planning strategy returns a plan, i.e., a ground solution; a *partial* strategy returns a partial plan, e.g., a sequence of subproblems. Partial planning strategies can be converted to total planning strategies by combining them with a weak planning method such as forward chaining. Reduced abstraction hierarchies or automatic subgoals are examples of partial planning strategies, since they require another planning strategy to refine the partial plan further. This categorization is related to the completeness of a planning strategy, since a complete planner must be necessarily a total planner, and a partial planner is necessarily incomplete. However, a total planner may or may not be complete.

One problem in machine learning is the distinction between a dynamic bias and a fixed bias system. A similar problem occurs in the definition of multi-strategy planning, because of the distinction between a set of planning strategies and a single planning strategy. In both cases, one could argue that even a dynamic biasing/multi-strategy planner executes a fixed algorithm and that it therefore only uses an (albeit more complex) bias/planning strategy. The thesis avoids this problem by being pragmatic and demanding that a single planning strategy must be exploited by a single planner. If a system uses more than one previously defined (i.e., referred to in the literature or implemented as a unique planning system) problem solving strategy, it is referred to as a multi-strategy planner. For example, case-based planning (e.g., CHEF) is considered a planning strategy, since it was implemented in

the CHEF system. Other examples of single strategy planners are means-ends analysis [?], relaxed abstraction based planning [?], and macro-based planning [?].

The definition of multi-strategy planning emphasizes two of the main features of multi-strategy planners: (a) the combination of different planning strategies, and (b) the emphasis on coordination of different strategies.

The remainder of this section discusses different approaches to multi-strategy planning. The description progresses from simple to more complex systems. The dimensions of comparison are problem vs. subproblem coordination and ordered vs. unordered coordination. The multi-strategy planning framework is based on an extension to the analysis of abstraction as described in section ??.

4.4.1 Problem coordination

From the discussion of abstraction in the previous section, a planning strategy A can be seen as a method to transform a planning problem into m_A subproblems. To be useful, the total cost of all subproblems must be less than that of the original problem. Otherwise, the abstraction does not lead to a performance advantage. In general, a planning strategy A can not reduce the complexity of all problems, but only improves performance on problems that satisfy the underlying assumptions of the planning bias. Therefore, it partitions the original problem set *Problems* into two sets Good and Bad. The cost of solving problems in Good is reduced by the abstraction, the cost of the other problems (Bad) is unchanged. A planning bias assigns a complexity from more than two possibilities to each problem in the set of problems. Some problems may lead to significant reduction, some to smaller reductions, and some to no reduction. In this case, a threshold τ can be used. The Good and Bad sets can be defined by a complexity limit τ, which determines the maximum resource limit of a planner. All problems that are solved within a resource limit τ are considered "good" problems. An abstraction

may always generate problems with the same or even a higher complexity than the original problem, in which case its *Good* set is the empty set, i.e., it does not improve performance on any problem.

$$\text{Good}(A, b, l, \tau) = \bigcup_{p \in \text{Problems}(b,l)} p \text{ such that } \text{Cost}(A, p) \leq \tau$$

$$\text{Bad}(A, b, l, \tau) = \bigcup_{p \in \text{Problems}(b,l)} p \text{ such that} p \notin \text{Good}(A, b, l, \tau)$$

If the abstraction finds an optimal solution and id all subproblems have the same solution length, the following equation describes a planning strategy (see equation **??**) and defines the worst case complexity of an abstraction A with a maximum branching factor b and a maximum decision length l. The node expansion cost E_A is approximated by the maximum node expansion cost. Recall that b_A is the maximum branching factor of any subproblem and m_A is the number of subproblems generated by planning strategy A.

$$\text{Cost}(A, p) \leq \left(\begin{array}{ll} m_A \, E_A \, b_A^{l/m_A} & \forall p \in \text{Good}(A, b, l, \tau) \\ E_A \, b^l & \forall p \in \text{Problems}(b, l) - \text{Good}(A, b, l, \tau) \end{array} \right) \tag{4.9}$$

It is assumed that the decision length is the dominant factor, i.e., abstraction is beneficial if applicable, and that there is no extra cost otherwise. This means that b_A is not too big to negate the reduction in decision length.

$$m_A \, b_A^{l/m_A} \ll b^l$$

The expected cost $\text{ECost}(A, b, l)$ of a planning strategy A over a set of problems $\text{Problems}(b, l)$ is defined as the cost of solving a problem weighted by the probability that the problem occurs. The probability is determined by a probability distribution over the set of problems. The Good (Bad) sets are the sets of problems that require cost less than (greater or equal to) the threshold.

$$\text{ECost}(A,b,l) \leq \sum_{p \in \text{Good}(A,b,l,\tau)} m_A\, E_A\, b_A^{l/m_A} \times \text{prob}(p) + \sum_{p \in \text{Bad}(A,b,l,\tau)} E_A\, b^l \times \text{prob}(p) \tag{4.10}$$

One factor in the expected cost of a planning strategy is the probability of having to solve a problem in the Good set. Therefore, the analysis assigns an applicability probability μ_A of getting a problem in the good set using strategy A. This probability is dependent on the values of the maximum branching factor and maximum decision length for the problems. The implicit parameters b and l are omitted for readability. Therefore, μ_A should be read as $\mu_{A,b,l}$ in the remainder of this chapter.

$$\mu_A = \sum_{p \in \text{Good}(A,b,l,\tau)} \text{prob}(p)\ \forall b, l$$

Then equation **??** can be replaced by the following equation

$$\text{ECost}(A,b,l) \leq \mu_A(m_A\, E_A\, b_A^{l/m_A}) + (1-\mu_A) E_A\, b^l \tag{4.11}$$

Problem coordinated multi-strategy planner with oracle

Assume that we are designing a multi-strategy planning system based on the abstract planning strategies $A_1, A_2, \ldots, A_n$. The simplest form of multi-strategy planner selects a problem, picks the best planner A_i for the problem, and solves it using planning strategy A_i.

Let us for the moment consider that there is an oracle that, given any problem p, returns the planning strategy with the smallest cost. This type of multi-strategy planning is refered to as problem coordinated multi-strategy planning with an oracle (PC-MSP-O). In this case, the expected cost of a multi-strategy planner is the minimum of the costs of the individual strategies A_i for problem p weighted by the probability that this problem occurs.

$$\text{Cost}(\text{PC-MSP-O}, b, l) = \sum_{p \in \text{Problems}(b,l)} \text{Min}(\text{Cost}(A_i, p)\ \forall i \in 1 \ldots n) \times \text{prob}(p) \tag{4.12}$$

The analysis assumes that all planning strategies have the same branching factor b and node expansion cost E. Otherwise b can be approximated by the maximum branching factor and E by the maximum node expansion cost, and equation **??** is an upper bound on the cost of the simple multi-strategy planner. The decision length (i.e., the number of reasoning steps necessary to find a solution) is reduced by a factor m_i, such that m_i is the maximum of the reduction factors of all planning strategies A_i such that $p \in \text{Good}(A_i, b, l, \tau)$. The decision length is reduced by the maximum factor, because the oracle provides us with the cheapest planning strategy to solve the problem. If none of the planning strategies are able to reduce the decision length for some problem, I assume that the last strategy A_n has a reduction factor of one ($m_n = 1$) and is applicable to all problems $\text{Good}(A_n, b, l, \tau) = \text{Problems}(b, l)$. If such a planning strategy does not exist, the planning strategy set can be extended to include strategy A_n. If the good sets are not disjoint, the sets can be changed by only keeping a problem in the good set with the maximum reduction factor. The oracle returns only one planning strategy (the one with the maximum reduction factor), which is used to solve the problem. Therefore, whether the problem is an element of some other planning strategy with a smaller reduction factor or not does not affect the complexity of the planner. Therefore, in the remainder of the analysis, the good sets are disjoint. That is, each problem p belongs to exactly one good set $\text{Good}(A_i, b, l, \tau)$ for some planning strategy A_i. Then, the expected cost of problem coordinated multi-strategy planning is given by equation **??**, where μ_1 is the probability of getting a problem in the Good set of A_1, μ_2 is the probability of getting a problem in the Good set of A_2 and so on.

$$\text{ECost(PC-MSP-O}, b, l) \leq \mu_1 m_1 E b^{l/m_1} + \mu_2 m_2 E b^{l/m_2} \ldots \tag{4.13}$$

Problem coordinated multi-strategy planning has received considerable attention in parallel systems, since it is easy to implement on a parallel system. The advantage is that there is little communication overhead, since the different planning strategies are independent.

Problem coordinated multi-strategy planner without oracle

Unfortunately, designing an oracle that determines a priori which planner will result in good performance is a hard problem. In the worst case, a planner may have to test all available planning strategies. For most planning strategies, important factors influencing performance are either unknown or expensive to test [**?**]. Therefore, it is unlikely that such an oracle exists. In the worst case, a multi-strategy planner may coordinate at a problem level and interleave, one step at a time, the execution of each planning strategy. Then the planner will also find the optimal solution, but the cost is increased by a factor of n, where n is the number of planning strategies used. The worst case cost of a problem coordination planner without an oracle (PC-MSP) is given by equation **??**.

$$\text{ECost(PC-MSP, b, l)} \leq n[\mu_1 m_1\, E b^{l/m_1} + \mu_2 m_2\, E b^{l/m_2} + \mu_3 m_3\, E b^{l/m_3} \ldots] \tag{4.14}$$

The difficulty is to determine for a planning strategy whether a problem is a member of the good set. This equation shows that there is at most a factor n difference between a problem coordinated planner with and without an oracle. This cost is too expensive to be practical, so cheaper methods for predicting the performance of a planner on a problem are necessary for this type of multi-strategy planning. For example, ALPINE's cost of creating an

abstraction hierarchy is linear in the length of the problem description [**?**]. Some other planning biases are harder to test. For example Bylander shows that determining operator decomposability is PSPACE-complete [**?**].

In general, there is a trade-off between the prediction accuracy and the prediction cost. For example, a simple (cheap) prediction method may only classify a few problems correctly as belonging to the good set. Therefore the set $\text{Good}(A, b, l, \tau)$ of a planning strategy A is replaced by a set $\text{Certain}(A, b, l, D, \tau)$, which is the subset of problems in $\text{Good}(A, b, l, \tau)$ that are *known* to be in the good set based on some decision procedure D. The closeness of the match between $\text{Certain}(A, b, l, D, \tau)$ and $\text{Good}(A, b, l, \tau)$ depends on the quality of the decision procedure D.

$$\text{Cost}(A, p) \leq \begin{pmatrix} m_A \, E_A \, b_{A,p}^{l/m_A} & \forall p \in \text{Certain}(A, b, l, D, \tau) \\ E_A \, b^l & \text{other problems in Problems}(b, l) \end{pmatrix} \quad (4.15)$$

For example, the multi-strategy planner designed in the next chapter (chapter **??**) uses induction to find a description of the *Good* sets of different planning strategies. The applicability conditions of a planning strategy are encoded as context, preconditions, and effects of a general operator. The decision cost $\text{Cost}(D, b, l)$ (cost of determining whether a problem is in the good set) is the match cost of matching the applicability conditions of the planning strategy, which is equal to the match cost of an operator.

Replacing the good sets $\text{Good}(A, b, l, \tau)$ by the known good sets $\text{Certain}(A, b, l, D, \tau)$ in the computation of the applicability probability μ yields equation **??** which gives the cost of problem coordinated multi-strategy planning with a decision procedure D. Note that this also changes the probability of getting a problem in the good set. Therefore, we define the D revised applicability probability of a planning strategy as follows:

$$\mu_{D,A} = \sum_{p \in \text{Certain}_{(A,b,l,D,\tau)}} \text{prob}(p) \; \forall b, l$$

In the remainder of this chapter, the decision procedure D is treated as an implicit paramter of μ and thus omitted.

$$\text{ECost}(\text{PC-MSP}, b, l) \leq n \, \text{Cost}(D, b, l) + \mu_1 m_1 \, E b^{l/m_1} + \mu_2 m_2 \, E b^{l/m_2} + \ldots \tag{4.16}$$

4.4.2 Subproblem coordination

This subsection investigates multi-strategy planning systems that coordinate different strategies at a subproblem level instead of a problem level. In contrast to problem coordinated multi-strategy planning, where a planning strategy is selected to solve the complete problem, a subproblem coordinated multi-strategy planner can change the planning strategy during solution of a problem and select a strategy for each subproblem.

The classes of planning strategies that may be used in a subproblem coordinated multi-strategy planner are larger than that of a problem coordinated one. The former allows the use of partial, whereas the latter requires total planning strategies.

All planning strategies used in subproblem coordinated multi-strategy planning must have an associated decision procedure to determine the applicability of a planning strategy. The cost of this decision procedure must be less than the cost of solving the original problem. Once all strategies have been applied, the subproblems are solved using some weak method, i.e. each planning strategy is applied exactly once. This type of coordination is useful, if a subproblem created by planning strategy A_i or resulting from applying some other planning strategy A_j to it, can not be reduced further by A_i.

Ordered subproblem coordinated multi-strategy planner

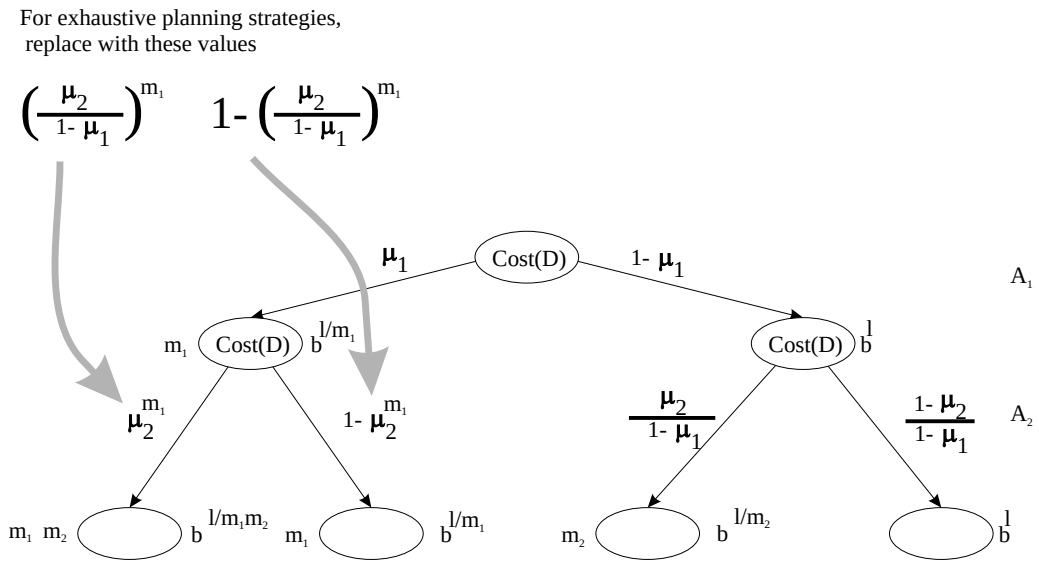
The first subproblem coordinated multi-strategy planner uses a static ordering on the set of planning strategies. The first planning strategy is applied to the problem p. If the problem p is in the known good set of the first strategy, it will be transformed into m_1 subproblems of size l/m_1. If p is not in the known good set, p is unchanged. Then the next strategy is applied to the resulting set of subproblems or the original problem.

Examples of ordered subproblem multi-strategy planning systems are Minton and Knoblock's Alpine/EBL system ([**?**]) and McCluskey's FM system [**?**]. Since ALPINE and automatic subgoaling are partial planning strategies, they lend themselves to an ordered subproblem approach, since they preprocess the domain to generate new problem specifications.

Figure **??** is an example of an ordered subproblem coordinated multi-strategy planner with two planning strategies (2-O-SC-MSP) with μ_1 and μ_2 being the probabilities that a problem p is in their respective disjoint known good sets, and m_1 and m_2 being their reduction factors. There is also no backtracking across planning strategies, that is, if a problem reduction is found, it can be refined to a complete solution (downward solution property). It is assumed that the subproblems are generated from the same probability distribution as the original problem, that is the applicability probabilities μ_1 and μ_2 are the same for generated subproblems and user requested problems. In practice, one would expect that the generated subproblems are somehow similar and therefore, the probability that all of them are in the good set is greater than that of a randomly chosen set of problems. This is based on a strong independence assumption for the generated subproblems.

An *exhaustive* planning strategy A guarantees that all subproblems generated by planning strategy A are not in the good set of A. Such a planning strategy reduces all problems as much as possible. For example, ALPINE can not further reduce an abstraction hierarchy that it created. The same is true for FM's automatic subgoaling strategy. An exhaustive planning strategy

Figure 4.2: Ordered subproblem coordinated multi-strategy planning



results in different conditional probabilities over the set of problems. The new conditional probability is 0 for all problems in the good set of A, and the conditional probability for all other problems is the original probability normalized by $1-\mu_A$. Figure **??** shows how the costs must be updated in the left part of the tree, if planning bias A_1 is an exhaustive planning strategy.

The analysis derives an upper bound on the cost of ordered subproblem coordinated multi-strategy planning. If the reduction factors $m_1, m_2, \ldots, m_n$ and the decision length l are large enough, then the cost is mostly determined by the subproblem with the largest decision length. Therefore, if all subproblems have the same length, the cost can only be significantly reduced if all subproblems can be further reduced. Although the reduction of some subproblems will improve the performance somewhat in practice, this performance improvement is ignored in the analysis. An upper bound on the cost of ordered subproblem coordinated multi-strategy planning with two

exhaustive planning strategies is given by:

$$\begin{aligned}&\text{ECost}(\text{2E-O-SC-MSP}, b, l) \leq \\ &\mu_1 \left(\tfrac{\mu_2}{1-\mu_1}\right)^{m_1} m_1 m_2 E b^{l/m_1 m_2} + \mu_1 \left(1 - \left(\tfrac{\mu_2}{1-\mu_1}\right)^{m_1}\right) m_1 E b^{l/m_1} \\ &+ \mu_2 m_2 E b^{l/m_2} + (1 - \mu_1 - \mu_2) E b^l 2\,\text{Cost}(D, b, l)\end{aligned} \tag{4.17}$$

The cost of ordered subproblem coordinated multi-strategy planning consists of three possibilities. First if all problems can be reduced, $m_1 m_2$ problems of size $l/(m_1 m_2)$ must be solved. Secondly, if only one strategy is applicable, the reduction results in m_1 problems of size l/m_1 or m_2 problems of size l/m_2. Thirdly, if none of the strategies can reduce the problem, the problem complexity remains unchanged.

If the planning strategies are not exhaustive, the probabilities in the left branch of figure **??** are not normalized and the expected cost of ordered subproblem coordinated planning is higher if the planning strategies are not exhaustive. Equation **??** shows the cost of ordered subproblem coordinated multi-strategy planning with two non-exhaustive planning strategies.

$$\begin{aligned}&\text{ECost}(\text{2-O-SC-MSP}, b, l) \leq \\ &\mu_1 \mu_2^{m_1} m_1 m_2 E b^{l/m_1 m_2} + \mu_1 (1 - \mu_2^{m_1}) m_1 E b^{l/m_1} + \mu_2 m_2 E b^{l/m_2} \\ &+ (1 - \mu_1 - \mu_2) E b^l + 2\,\text{Cost}(D, b, l)\end{aligned} \tag{4.18}$$

From equation **??**, two more conclusions can be drawn. First, it shows that if all strategies have similar reduction factors, the cost of ordered subproblem coordinated planning is minimized if the strategies are checked in increasing order of applicability probability. Given two strategies A_1 and A_2 with $\mu_1 > \mu_2$ and $m_1 = m_2$, A_2 should be checked first. This result can be derived by comparing the expected costs in both cases. The difference in expected cost for the two planners (MSP12 uses A_1 and then A_2, MSP21 uses A_2 and then A_1) is given by

$$\begin{aligned}&\text{ECost}(\text{MSP21}, b, l) - \text{ECost}(\text{MSP12}, b, l) = \\ &\mu_1\mu_2(\mu_2^{m-1} - \mu_1^{m-1})(mEb^{l/m} - m^2Eb^{l/m^2})\end{aligned}$$

This difference is negative, which implies that the planning strategy with the lower probability should be tested first. Secondly, if the reduction factors are not equal ($m_1 > m_2$), but the two planning strategies have similar applicability probabilities μ, then testing the weakest (minimum reduction factor) strategy MSP21 first minimizes the cost.

$$\begin{aligned}&\text{ECost}(\text{MSP21}, b, l) - \text{ECost}(\text{MSP12}, b, l) = \\ &-\mu^{m_2+1}m_2Eb^{l/m_2} + \mu^{m_1+1}m_1Eb^{l/m_1} + (\mu^{m_2+1} - \mu^{m_1+1})m_1m_2Eb^{l/m_1m_2}\end{aligned}$$

Assuming that the term involving m_2 is the dominant term ($m_1 > m_2 \Rightarrow b^{l/m_2} \gg b^{l/m_1}$), it follows that the difference is negative and that the expected cost of MSP21 is less than that of MSP12.

This results also hold true, if we assume that the planning strategies generate similar subproblems, that is either all or no subproblem can be refined further. This counter intuitive result holds true for an *ordered* subproblem coordinated planner. The situation changes for an *unordered* subproblem coordinated planner, since the question of which bias to apply first is meaningless. Remember that the good sets are disjoint, so there is at most one applicable planning strategy at each level.

Equation **??** shows the difference in costs between problem coordinated planning with an oracle and ordered subproblem coordinated planning with two non-exhaustive planning strategies. The difference for exhaustive strategies would be greater by approximately a factor of $1/(1-\mu_1)^{m_1}$.

$$\begin{aligned}&\text{ECost}(\text{2-PC-MSP-O}, b, l) - \text{ECost}(\text{2-O-SC-MSP}, b, l) = \\ &\mu_1\mu_2^{m_1}E(m_1\, b^{l/m_1} - m_1m_2b^{l/m_1m_2}) - 2\text{Cost}(D, b, l)\end{aligned} \tag{4.19}$$

If the decision cost Cost(D, b, l) is small enough compared to the search costs, it shows that the expected cost of ordered subproblem coordinated multi-strategy planning (O-SP-MSP) is only slightly lower than that of problem coordinated multi-strategy planning with an oracle (PC-MSP-O). This happens because the probability of reducing *all* subproblems generated by A_1 is small $\mu_1\mu_2^{m_1}$. In practice, reducing a subset of all subproblems will reduce the cost, but this effect is ignored in the analysis. It is assumed that if only some of the generated subproblems can be reduced further, the complexity of the remaining ones will dominate the expected cost ($m_1\, b^{l/m_1} \gg m_1 m_2\, b^{l/m_1 m_2}$).

The analysis shows that compared to a problem coordinated multi-strategy planner with an oracle, subproblem coordinated planning has a lower expected cost than problem coordinated one (with and without an oracle). The expected savings decrease greatly with an increase in the number of generated subproblems if all subproblems must be reduced to improve performance significantly.

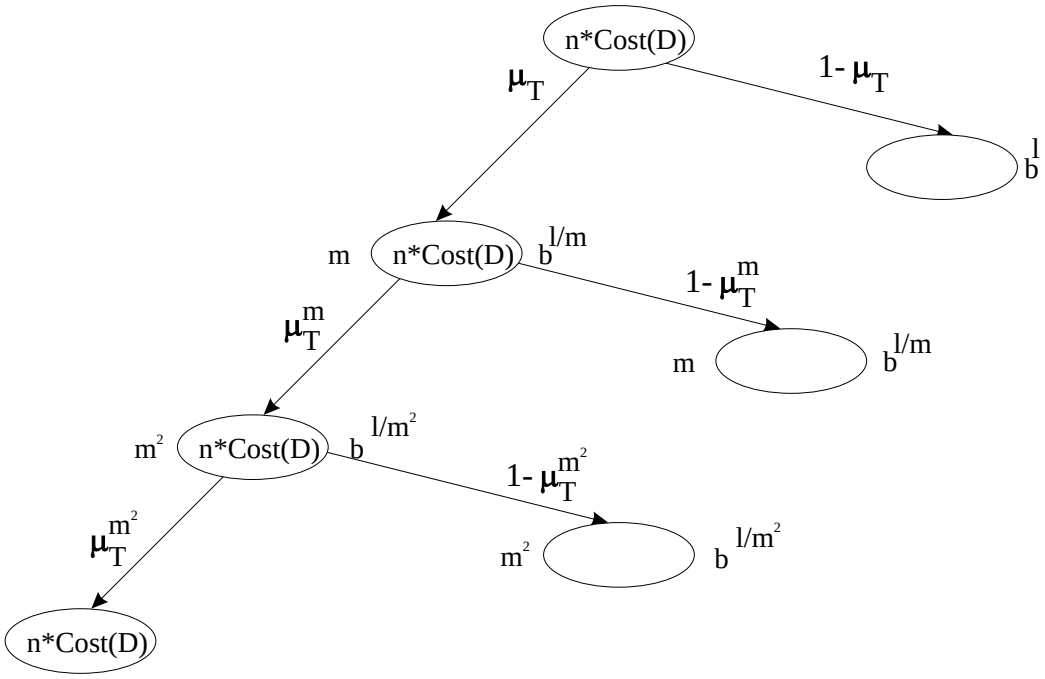
Unordered subproblem coordinated multi-strategy planner

The last multi-strategy planning system described in this framework is one with subproblem coordination, but no ordering on the set of planning strategies. This is denoted as unordered subproblem coordinated multi-strategy planning (SP-MSP). The known good sets of the multi-strategy planners are disjoint, i.e., a problem is exactly in at most one known good set. If all planning strategies have the same reduction factor m, the different planning strategies can be approximated by a single strategy with an applicability probability of $\mu_T = \sum_{i=1}^{n} \mu_i$ and a common reduction factor of m.

Figure **??** is a graphical representation of unordered subproblem coordinated multi-strategy planning with three reduction levels. The maximum number of reductions is not limited by the number of planning strategies, as

in the ordered case. At the top level, the problem can not be refined with probability $1-\mu_T$, in which case the problem complexity is b^l. Otherwise, the problem generates m subproblems. Given that the solution length is large enough, we assume that a search reduction can only improve performance if all m subproblems can be refined further. The probability that all subproblems can be refined is μ_T^m. The planning strategies are non-exhaustive strategies. If some of the planning strategies are exhaustive, the following cost is an overestimate, since the probabilities at the next level would have to be normalized by the probability of the previously selected planning strategy.

Figure 4.3: Unordered subproblem coordinated multi-strategy planning



Then the cost of unordered subproblem coordinated multi-strategy planning is given in equation **??**, where k is the number of iterations that the set of all problems can be reduced successfully. In the figure, k is equal to the

number of level in the tree (3).

$$\text{ECost}(\text{SP-MSP}, b, l) \leq nk\text{Cost}(D, b, l) + \sum_{i=0}^{k} \mu_T^{\frac{m^i-1}{m-1}}(1-\mu_T^{m^i})m^i E b^{l/m^i} \quad (4.20)$$

For two planning strategies, the reduction probability is ($\mu_T = \mu_1 + \mu_2$).

The expected cost of ordered versus unordered subproblem multi-strategy planning can be computed. The probability of solving a problem of size b^l is identical in both cases ($1-\mu_T$). The first difference occurs in the probability of solving a problem of size $b^{l/m}$. Because of the dominance assumption, the remaining terms ($b^{l/m^2}, \ldots$) are ignored.

$$\begin{aligned}\text{ECost}(\text{2-O-SC-MSP}, b, l) - \text{ECost}(\text{2-SP-MSP}, b, l) \approx \\ ((\mu_1+\mu_2)^{m+1} - \mu_1\mu_2^m)mEb^{l/m}\end{aligned} \quad (4.21)$$

Comparing the expected costs of both planners shows that unordered subproblem coordinated multi-strategy planning has a lower expected cost than ordered one.

Table ?? summarizes the results in this section for two planning strategies. It shows the expected cost of different multi-strategy planning systems and their relationship. The expected cost decreases from the top to the bottom.

As can be seen, unordered subproblem coordinated planning may improve performance over ordered one, because the probability that a problem can be further reduced is higher than that of any single strategy. The exact improvement depends on the number of levels.

Table 4.3: Expected cost of different multi-strategy planners

Problem coordinated, without oracle

Total planning strategies

$$\text{ECost}(\text{2-PC-MSP}, b, l) = 2\text{Cost}(D, b, l) + \mu_1 m_1 \, Eb^{l/m_1} + \mu_2 m_2 \, Eb^{l/m_2} + (1 - \mu_1 - \mu_2) Eb^l$$

Problem coordinated, with oracle

Total planning strategies

$$\text{ECost}(\text{2-PC-MSP-O}, b, l) = \mu_1 m_1 \, Eb^{l/m_1} + \mu_2 m_2 \, Eb^{l/m_2} + (1 - \mu_1 - \mu_2) Eb^l$$

Subproblem coordinated, ordered

Total or partial exhaustive planning strategies

$$\text{ECost}(\text{2E-O-SC-MSP}, b, l) = + 2\,\text{Cost}(D, b, l) + \mu_1 \left(\tfrac{\mu_2}{1-\mu_1}\right)^{m_1} m_1 m_2 Eb^{l/m_1 m_2} + \mu_1 \left(1 - \left(\tfrac{\mu_2}{1-\mu_1}\right)^{m_1}\right) m_1 Eb^{l/m_1} + \mu_2 m_2 Eb^{l/m_2} + (1 - \mu_1 - \mu_2) Eb^l$$

Subproblem coordinated, ordered

Total or partial planning strategies

$$\text{ECost}(\text{2-O-SC-MSP}, b, l) = 2\,\text{Cost}(D, b, l) + \mu_1 \mu_2^{m_1} m_1 m_2 Eb^{l/m_1 m_2} + \mu_1 (1 - \mu_2^{m_1}) m_1 Eb^{l/m_1} + \mu_2 m_2 Eb^{l/m_2} + (1 - \mu_1 - \mu_2) Eb^l$$

Subproblem coordinated, unordered

Total or partial planning strategies

$\mu_T = \mu_1 + \mu_2$, k is the number of levels

$$\text{ECost}(\text{2-SP-MSP}, b, l) = 2k\text{Cost}(D, b, l) + \sum_{i=0}^{k} \mu_T^{\frac{m^i - 1}{m-1}} (1 - \mu_T)^{m^i} m^i Eb^{l/m^i}$$

4.5 Analysis of different planning strategies

This section analyses different planning strategies. The results of this analysis is used to (a) show that these strategies fit into the framework described in the previous section and (b) to motivate an unordered subproblem multi-strategy planner based on case-based, abstraction-based, and macro-based planning.

4.5.1 Analysis of automatic subgoaling

This subsection investigates the model of abstraction with respect to automatic subgoaling. The abstract branching factors (i.e., the branching factor of the search spaces for the subproblems) are equal to the branching factor in the original search space, since the search space for solving subgoals and the original problem are identical.

$$b_S = b$$

Replacing b_A and c_A in equation **??**, yields equation **??** to describe the cost of automatic subgoaling.

$$\text{Cost}(\text{SUB}, b, l) = m_S\, c_S + m_S E b^{\frac{l r(l)}{m_S}} \tag{4.22}$$

Automatic subgoaling improves performance if $\text{Cost}(\text{SUB}, b, l)$ is less than the cost of standard planning, that is:

$$\text{Cost}(\text{SUB}, b, l) \leq E b^l \Rightarrow m_S c_S + m_S E b^{\frac{l r(l)}{m_S}} \leq E b^l$$

Ignoring the cost of setting up a problem, and a term of $\log m_S$, automatic subgoaling is useful if

$$m_S \geq r(l)$$

Automatic subgoaling is worthwhile, if the number of subgoals grows faster than the "blow up function." In the special case that automatic subgoaling finds the optimal (i.e., shortest) solution, then $r(l) = 1$ and automatic subgoaling is beneficial for any $m_S > 1$.

Under those assumptions automatic subgoaling is identical to the model of an abstraction $\text{Cost}(A, b, l)$ (Equation **??**) with $m_S = m_A$ and $b_A = b_S$ as described in section **??**.

4.5.2 Analysis of two-level abstraction based planning

This subsection analyses abstraction based planning with two levels of abstraction. As described previously, abstraction based planning solves a problem first in an abstract space with branching factor b_{A2} and l_{A2} and refines the solution to a solution in the ground space. Each operator in the abstract plan yields a subproblem in the ground space. Assume that an abstract planner yields m_{A2} subproblems, then there is a total of $m_{A2} + 1$ subproblems, m_{A2} ground problems and one abstract problem. The ground problems have a branching factor of b and solution length l_{A2}.

$$\text{Cost}(\text{A2}, b, l) = Eb_{A2}^{m_{A2}} + \sum_{i=1}^{m_{A2}} Eb^{l_{A2}} \tag{4.23}$$

Since the ground problems are solved in the original search space, their branching factor is equal to the original branching factor b. Furthermore, the abstract problem space consists of sets of ground states, and therefore, the branching factor in the abstract space is limited by the branching factor of the ground space $b_{A2} \leq b$.

If the solution found by the abstraction based planner is identical to an optimal solution, it follows that $l_{A2} = l/m_{A2}$. The abstract solution length is equal to m_{A2} to yield m_{A2} subproblems. Replacing b_A, l_A yields the following cost:

$$\text{Cost}(\text{A2}, b, l) = Eb^{m_{A2}} + m_{A2} Eb^{l/m_{A2}} \tag{4.24}$$

Comparing equation **??** to this equation shows that automatic subgoaling is cheaper than two-level abstraction based planning if both have the same reduction factor. The difference is a constant $b^{m_{A2}}$, since abstraction-based planning searches for a sequence of suitable subgoals, whereas this sequence is already known in automatic subgoaling.

The cost in equation **??** is minimized, if the two terms are the same size $m_{A2} \approx l/m_{A2}$ For a two-level abstraction-hierarchy, the cost is minimized for $m_{A2} = \sqrt{l}$. Knoblock [**?**] derives an identical result, based on the ratio of search space sizes.

The cost of the model of abstraction and the cost of a two-level abstraction hierarchy are almost identical (ignoring the cost of solving the abstract problem) $\text{Cost}(\text{A2}, b, l) \approx \text{Cost}(\text{A}, b, l)$.

4.5.3 Analysis of multi-level abstraction based planning

The analysis of two-level abstraction-based planning can be generalized to abstraction hierarchies with multiple levels. A multi-level abstraction hierarchy results in a tree of subproblems. Assume (1) that each abstract plan yields m_{AM} subproblems, (2) that again the abstraction based planner finds an optimal solution, and (3) that the branching factors in the abstract search spaces are bounded by b, then it can be shown that an abstraction based planner with multiple levels of abstraction can reduce the complexity to be linear in l rather than exponential:

The leaf nodes of the subproblem tree consist of ground level problems. If k is the number of levels in the abstraction hierarchy, then there are m_{AM}^k ground problems, and each ground problem has a size of l/m_{AM}^k. The number of abstract problems is the size of the tree not including the ground level,

that is the number of nodes in a tree of depth $k-1$. Since each abstract problem generates m_{AM} problems at the next level, there are $\frac{m_{AM}^k-1}{m_{AM}-1}$ abstract problems, each of length m_{AM}.

$$\sum_{i=0}^{k-1} m_{AM}^i = \frac{m_{AM}^k - 1}{m_{AM} - 1}$$

The total cost of multi-level abstraction based planning is thus given by,

$$\text{Cost}(\text{AM}, b, l) = \frac{m_{AM}^k - 1}{m_{AM} - 1} E b^{m_{AM}} + m_{AM}^k E b^{l/m_{AM}^k} \tag{4.25}$$

The total number of problems is $\frac{m_{AM}^k-1}{m_{AM}-1} + m_{AM}^k$. For $k = \log_{m_{AM}} l$, the terms in equation ?? yield roughly equally size subproblems, which minimizes the cost. Substituting this value for k, the cost of multi-level abstraction based planning can be approximated as

$$\text{Cost}(\text{AM}, b, l) \leq \frac{l-1}{m_{AM} - 1} E b^{m_{AM}} \tag{4.26}$$

Since b and m_{AM} are constants, the cost in equation ?? is linear in the original solution length.

The framework for abstraction introduced in section ?? describes a two level abstraction hierarchy. Replacing $b_A = b$, $m_A = m_{A2}$, and ignoring a constant term of $b^{m_{A2}}$ transforms equation ?? into equation ??. Abstraction hierarchies with k levels can be modeled by substituting $m_A = m_{AM}^k$ and $b_A = b$ ignoring the cost of solving the abstract problems.

4.5.4 Analysis of case-based planning

Case-based planning does not generate new plans from scratch, but adapts previous plans to the new situation. Therefore, the determinant factor is the similarity between the current problem and the retrieved plan. In general, the cost of case-based planning is the sum of the costs for finding and retrieving

a similar case and for adapting this plan to the new situation.

Case-based planning is memory intensive, many plans have to be stored to cover enough problems. Plan retrieval, therefore, has to retrieve the most suitable plan from a large set of cached plans. However, this retrieval can be done cheaply, by using an indexing method (constant time) or a discrimination net (logarithmic in the number of cached plans).

Therefore, this subsection focuses on the case adaptation, which is a search problem through the space of possible plan adaptations and is thus the most expensive part of a case-based planner. Adaptation may be a non-trivial task in itself and partitioning the adaptation process into smaller subproblems may be necessary.

This analysis is based on the assumption that each adaptation forms an independent subproblem and the number of adaptations is m_C. For example, a case-based planner may have to remove a prefix plan from the case and add a suffix plan to the case to solve the new problem. One assumption of case-based planning is that these adaptations can be done from the head to the tail of the plan without interference. If two plan adaptations are not independent, they can be viewed as one macro plan adaptation with a decision length that is equal to the sum of the individual adaptations. The average branching factor for the adaptation space is b_C and l_C is the search length for an adaptation.

Then the cost of case-based planning $\text{Cost}(\text{CBP}, b, l)$ is given as:

$$\text{Cost}(\text{CBP}, b, l) = m_C b_C^{l_C} \tag{4.27}$$

The important difference between case-based planning and other planning strategies is that the branching factor b_C and the decision length l_C are independent of the solution length and branching factor of the ground space. The cost of case-based planning is determined by the syntactical similarity of the retrieved case and the new problem.

It seems, nevertheless, reasonable to assume that the length of the adap-

tation is related to the length of the problem. If the length of the case adaptation is a fraction of the original solution length, and that the number of adaptations is dependent on this factor, case-based planning can also be described by the model of abstraction proposed in equation **??**.

A particular method of case-based planning is Hank's "Systematic Plan Adaptor" (SPA) system [**?**]. SPA interprets plan generation as a special case of plan adaptation. An efficient method to unwind the derivation of a plan is used and the adaptation methods contain the original operators plus one extra operator to undo a previous planning decision. The branching factor is increased by one, since there is only one branch for undoing a previous planning step, since SPA guarantees systematicity, that is SPA guarantees that each candidate plan is only created once. Also, rather than as individual adaptations of smaller decision length, SPA views case-based planning as the creation of a single larger adaptation. It follows that $b_C = b + 1$, and equation **??** can be rewritten as:

$$\text{Cost}(\text{SPA}, b, l) = E(b+1)^{l_C} \tag{4.28}$$

There is a philosophical difference between SPA and CHEF. The former one interprets case-based planning as planning with an additional operator to undo previous planning decisions. Hammond's CHEF system sees case-based planning as an opportunity to partition the original problem into smaller subproblems and to use the current plan failures to guide this partitioning. CHEF also shows that a small set of plan adaptations is sufficient to debug many failures in at least one domain.

4.5.5 Analysis of macro-based planning

This subsection discusses the cost of macro-based planning. The planner adds macros to the operator set to reduce the decision length. The branching factor of the search space can be broken up into two parts: (a) the branching

factor b of the original search space, and (b) the additional branching factor b_M due to the new macro-operators. Let l_M be the decision length of the macro planner, then the cost of macro-based planning is given by:

$$\text{Cost}(\text{MAC}, b, l) = E(b + b_M)^{l_M} \tag{4.29}$$

The new decision length l_M depends on the exact macro learner. This analysis assumes that a macro planner reduces the decision length by a constant *factor* m_M. Reducing the decision length by a constant $l - C$ will result in no significant improvement in search complexity, since $O(l - C) = O(l)$.

The cost of a macro-based planner is given by:

$$\text{Cost}(\text{MAC}, b, l) = E(b + b_M)^{l/m_M} \tag{4.30}$$

From this equation, a set of sufficient conditions for speed up can be derived by comparing the costs of a planner with and without macros.

$$\text{Cost}(\text{MAC}, b, l) \leq \text{Cost}(\text{MEA}, b, l) \Rightarrow E(b + b_M)^{l/m_M} \leq Eb^l \tag{4.31}$$

Solving this equation for m_M leads to the following inequality:

$$m_M \geq \frac{\log(b + b_M)}{\log b} \tag{4.32}$$

This equation allows us to develop sufficient conditions for a speed up. For example, if a single macro is applicable in all states ($b_M = 1$), and that the original branching factor b is three, then the decision length has to be reduced by a factor of at least 1.26 to increase performance. In other words, the decision length must be reduced by at least 20 percent.

This analysis ignores the matching cost of macros. Minton showed that sometimes the cost of testing whether a macro is applicable or not (the matching cost) may outweigh any possible gain of the macro [**?**]. The matching cost

of an operator grows exponentially with the number of free variables. This additional cost increases the node expansion cost E and is incurred even if the operator is not applicable and thus does not add to the branching factor, but does increase the cost of expanding a node. However, this additional cost contributes only a constant factor to the search cost and is thus ignored in the theoretical analysis. For any constant increase in expansion cost, there is a minimum decision length, such that the difference in complexity outweighs any difference in expansion cost. The matching cost is of practical importance. That is why chapter ?? describes DoLittle's learning methods which do consider the matching cost.

Comparing equation ?? and equation ?? and substituting $b_A = b + b_M$ and $m_A = m_M$ shows that the framework in section ?? is an overestimate by a factor of m_M for macro-based planning. Since the cost is an over estimate of the true cost of macro-based planning, the framework is a reasonable approximation of macro-based planning.

4.6 Analysis of multi-strategy planning example

This section analyzes multi-strategy planning and derives conditions under which it can achieve an exponential speed up over three single-strategy planners (macro-based, case-based, and abstraction-based). The analysis is based on a set of assumptions. Given those assumptions, the worst case complexity of a multi-strategy planner is shown to be constant in the length of the solution as opposed to the exponential complexity of an abstraction-based, macro-based, and case-based planner. The costs derived for the individual planning strategies are underestimates, which means that the difference in costs between multi-strategy and single strategy planning may be even greater.

This analysis shows that multi-strategy planning is at least in one case

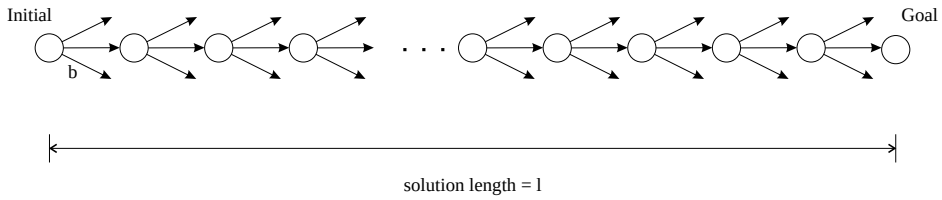
beneficial, and the set of assumption used in the analysis determines applicability conditions for multi-strategy planning. The analysis shows furthermore, that the coordination must occur at a subproblem level, instead of at the problem level. In other words, an ordered subproblem coordinated multi-strategy planner performs better than picking the best planner for a given problem (problem coordinated multi-strategy planner with an oracle).

4.6.1 MSP example: Means-ends analysis

Assume that we are given a problem whose shortest solution contains l primitive operators. The branching factor of this search space is given as b. Therefore, the worst case complexity of a means-ends analysis planner is given by:

$$\text{Cost}(\text{MEA}, b, l) = Eb^l$$

Figure 4.4: Example: analysis of means-ends planning



4.6.2 MSP example: Case-based planning

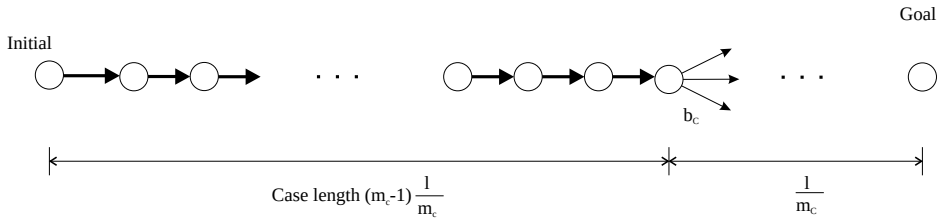
A simple case-based planner exists that can retrieve previous cases and adapt them. It uses a means-ends analysis planner to create adaptations such as prefix or postfix plans. On average all but a fraction $1/m_C$ of a case is reusable. Therefore, the decision length of a case-based planner l_C is

reduced from l to l/m_C. Furthermore, since the case-based planner uses means-ends analysis to search for adaptations, $b_C = b$. Note that this is an optimistic assumption, generally case-based planners have a higher branching factor than means-ends analysis planner (for example SPA and equation **??**). Furthermore, this example assumes that only one adaptation is necessary.

Figure **??** is an example, where the case-based planner has to create a suffix plan of length l/m_C. Replacing the variables in equation **??** leads to the following cost of case-based planning:

$$\text{Cost}(\text{CBP}, b, l) = Eb^{l/m_C} \tag{4.33}$$

Figure 4.5: Example: analysis of case-based planning



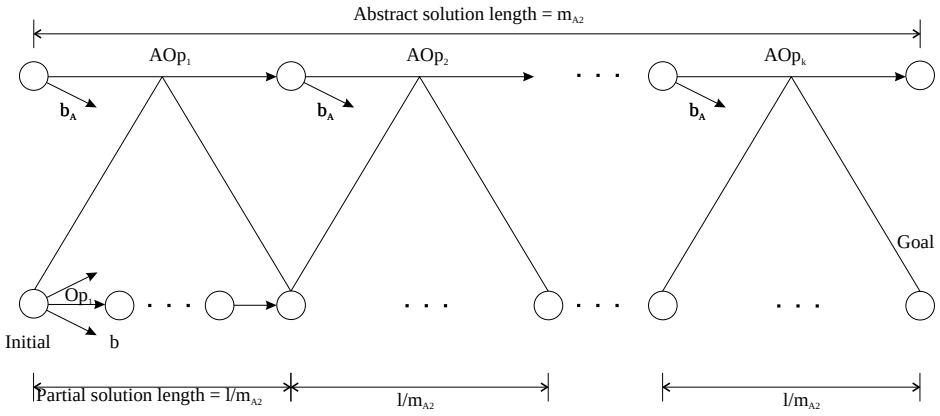
4.6.3 MSP example: Abstraction-based planning

Assume that for any problem in the domain, a two-level abstraction hierarchy exists that can partition the original problem into m_{A2} equal-sized subproblems. Also, there is no backtracking across levels. From the analysis of two-level abstraction-based planning in subsection **??**, it follows that the cost of abstraction-based planning (equation **??**) in this case is given as:

$$\text{Cost}(\text{A2}, b, l) = Eb^{m_{A2}} + m_{A2}Eb^{l/m_{A2}} \tag{4.34}$$

Figure ?? shows an example of abstraction-based planning. At the top is an abstract plan of length m_{A2}. Each abstract operator is refined by a primitive operator sequence of length l/m_{A2}.

Figure 4.6: Example: analysis of abstraction-based planning



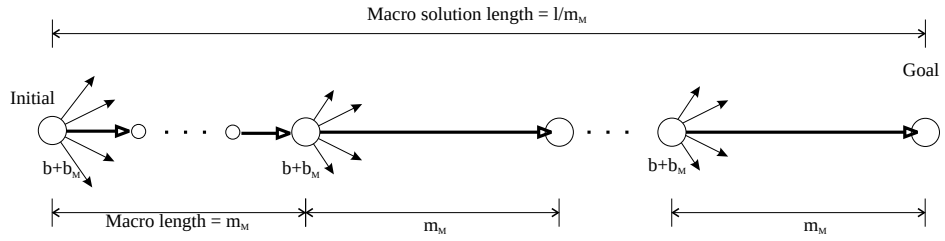
4.6.4 MSP example: macro-based planning

Each macro increases the branching factor of the search space as well as the matching cost per node. Furthermore, the number of possible macros increases exponentially with the length of the macro; there are b^{l_M} possible macros of length l_M. Therefore, it is increasingly unlikely that a macro-planner contains all necessary macros of a given length as this length increases. The analysis assumes that the macro planner contains all necessary macros of length less than or equal to some constant m_M. The intuition is that only a small subset of them are necessary to reduce the decision length sufficiently. In this analysis, the additional operators add a small constant term b_M to the branching factor.

From equation ??, the cost of macro-based planning in this example is given as:

$$\text{Cost}(\text{MAC}, b, l) = E(b + b_M)^{l/m_M} \tag{4.35}$$

Figure 4.7: Example: analysis of macro-based planning



4.6.5 MSP example: multi-strategy planning

This subsection analyses the cost of multi-strategy planning for the example. The multi-strategy planner is combining the case-based (subsection ??), abstraction-based (subsection ??), and macro-based planning (subsection ??).

The multi-strategy planner uses a simple search control method, it has access to all cases, abstractions, and macros. It uses an admissible search control such as breadth-first or iterative deepening. An admissible search method is one that guarantees that the first solution returned is the optimal solution. This simple search control method is sufficient, since there is only one necessary plan adaptation, the creation of a suffix plan. Furthermore, abstract operators are refined in left to right order, after the top-level plan has been created.

As mentioned previously, the motivation for cases is to generate operator sequences that are long enough to support reuse. Macros, on the other hand, are short, general operator sequences. Therefore, the following assumption is used to relate the sizes of the respecting search spaces:

$$l/(m_C m_{A2}) \leq m_M \leq l/m_C, l/m_{A2}$$

First, the planner generates an abstract plan that solves the problem (a) by retrieving a case, and (b) by adding an abstract suffix plan. As in the previous analysis, the retrieval of the case is cheap and its cost ignored. The abstract plan contains m_{A2} abstract operators, which each generate a subproblem of size $l/(m_C m_{A2})$. Since the resulting search spaces are smaller than m_M, a macro exists to solve them.

The cost of multi-strategy planning is therefore, the cost of (a) retrieving and adapting a suitable case, (b) generating an abstract plan of length m_{A2}, and (c) solving subproblems of size less than m_M. Since m_M is the maximum macro length, each subproblem can be solved by one macro operator. Therefore, the total cost of multi-strategy planning is given by:

$$\begin{aligned} \text{Cost}(\text{MSP}, b, l) &= Eb^1_{MSP} + Eb^{m_{A2}}_{MSP} + m_{A2} Eb^1_{MSP} \\ &= (m_{A2}+1)Eb_{MSP} + Eb^{m_{A2}}_{MSP} \end{aligned} \tag{4.36}$$

Since a multi-strategy planner allows the use of any operator, the branching factor of the associated search spaces is given as the sum of the branching factors of the individual search spaces.

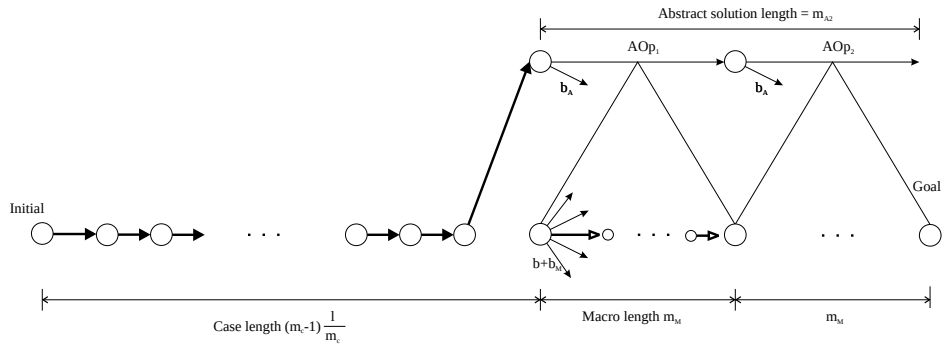
$$b_{MSP} = b + b_C + b_{A2} + b_M \tag{4.37}$$

Since b_C, b_A, b_M, b, and k_A are constants, this function is constant in the solution length l. Comparing this to the cost of means-ends analysis b^l, case-based b^{l/m_C}, abstraction-based $b^{l/m_{A2}}$, and macro-based planning b^{l/m_M} shows that multi-strategy planning may lead to an exponential reduction in search cost.

Figure **??** shows the performance of multi-strategy planning. Using a previous case, the problem is reduced to one of finding a suffix plan of length

l/m_C. This problem is further subdivided by an abstraction hierarchy, which results in ground subproblems of length m_M. The ground subproblems are solved using macros.

Figure 4.8: Example: analysis of multi-strategy planning



4.7 Discussion

This analysis, clearly, does not imply that a multi-strategy planner reduces a NP-hard problem to a constant one. It does show that under certain conditions, it may produce an exponential speed up over a single strategy planner. In general, this depends on the ability of a multi-strategy planner to divide a problem into smaller subproblems than a single strategy planner.

The analysis was based on a number of assumptions. The most crucial assumption was that the subproblems created by the planning strategy are roughly of equal size. On the other hand, if one subproblem is as difficult or even more difficult than the original problem, the planning strategy will not lead to a reduction in search cost. Another assumption is that the planner finds an optimal solution or one bounded by a blow-up function to bind the otherwise possibly infinite search spaces.

Nevertheless, the analysis is useful since it provides a insight into sufficient conditions for improvement using a multi-strategy planning system. The critical factors are the reduction of the search spaces associated with each planning strategy, the applicability probability of the planning strategy, and the cost of determining whether a problem belongs to the good set of a planning strategy.

The analysis focused on a case-based, abstraction-based, and macro-based planner as an example, but indeed can be extended to include other planning strategies. Macros, cases, and abstractions were selected because of their popularity in the literature.

However, this chapter only used a primitive method of selecting and coordinating different planning strategies. The analysis motivates the design of an unordered subproblem coordinated multi-strategy planning system DoLittle described in the next chapter (chapter **??**).

Chapter 5

DoLittle: a multi-strategy planner

> It is to be noted that when any part of this paper appears dull, there is a design in it.
>
> **Sir Richard Steele**, 1672–1729, *The Tatler*.

Based on the analysis in chapter **??**, this chapter describes the design of an unordered subproblem coordinated multi-strategy planning system that is able to combine most of the planning systems described in chapter **??**. The set of problem solving strategies includes forward chaining, means-ends analysis, case-based, automatic subgoaling, abstraction-based, reactive rules, and macro-based planning. The design and implementation shows the practical feasibility and allows the empirical evaluation of multi-strategy planning. The unified framework also allows comparison of different planning strategies. The design emphasizes three popular planning strategies: case-based, abstraction-based, and macro-based planning. These planning strategies are of interest because they require different search control strategies, and are based on different planning biases. This makes them distinct enough, so that combining them is non-trivial, and also of practical interest.

Section ?? argues for a particular design decision, the omission of partial order planning. Section ?? specifies the requirements of a multi-strategy planning system. DoLittle's representation of planning strategies and its decision procedure are described in sections ?? and ??. Section ?? describes general operators and provides examples of the representation of popular planning strategies in DoLittle. Section ?? is a description of DoLittle's domain language, a version of Prodigy4's description language with enhancements to support the representation of different planning strategies as general operator. Since a multi-strategy planner must be able to emulate different planning strategies, it requires a search control method that can emulate different problem solvers (section ??). Section ?? discusses the design of DoLittle with respect to its requirements.

5.1 Why does DoLittle not include partial-order planning?

Although the superiority of partial-order over total-order planning was long assumed by researchers, more recent research has focused on systematic comparisons. This work concluded that indeed, as with any other planning strategy, partial-order planning performs well in some instances [?], but that there are also domains in which a total-order planner is superior [?, ?].

Furthermore, a large part of partial-order planning's appeal is its ability to solve non-linear problems, i.e., problems where subgoals must be interleaved. However, the issue of linear vs. non-linear planning is different from total vs. partial-order planning, i.e., a partial-order planner may or may not be a linear planner. For example, DoLittle and Prodigy4 are non-linear, total-order planners.

The largest drawback to partial-order planning is the cost of calculating the truth criterion, i.e. whether a given predicate is true or not after a partially ordered plan. This is especially true for complex domain descrip-

tion and plan languages. However, some progress has been made to include them. For example, the UCPOP planner supports conditional effects [?]. The truth criterion is NP-hard for STRIPS like representations [?], and undecidable for more powerful ones [?]. Informally, this happens because in a partially ordered plan with n operators, there are $n!$ different traces. In the worst case, we have to test all $n!$ traces to see whether a predicate is true after execution of the operator sequence. Partial-order planners generally restrict either the representation language or the computation of the truth criterion to overcome this problem [?].

It is also harder to learn new operators from partial order plans, since it is harder to estimate the savings of a new operator. Since DOLITTLE is intended as part of an intelligent apprentice system, one constraint on the design was that DOLITTLE should be able to learn to improve problem solving inductively. Most learning methods are designed for totally-ordered planners.

Lastly, as other researchers do, I agree that humans use plan debugging instead of partial-order planning during problem solving [?]. Therefore, debugging of plans is a very important aspect of planning. However, debugging partial-order plans is complex, and there has been little research in this area. So, the design emphasizes plan debugging at the expense of partial-order planning. One direction of future work will be to incorporate partial-order planning into DOLITTLE. The main obstacle is to convert the plan debugging methods to allow for partial order plans.

However, note that the omission of partial-order planning was a design decision. There is nothing in the multi-strategy paradigm per se that disallows partial-order planning. A multi-strategy planner can combine total order and partial order planning, which in turn can reduce the expensive truth criterion to a small subset of the plan, instead of the whole plan.

5.2 Requirements

A multi-strategy planner must be able to break a problem up into subproblems, pick an appropriate problem solving strategy for each subproblem, solve the individual subproblems with it, and then combine the partial solutions to create a solution to the original problem.

Therefore, the following requirements must be met for DoLittle to be a practical multi-strategy planning system:

1. a representation language for different planning strategies
2. a decision procedure D that determines whether a planning strategy is appropriate
3. given a representation of different planning strategies, a search control method that emulates the effect that different planning strategies have on the problem space
4. a domain description language that is powerful enough to describe complex domains

The solution to problems ?? and ?? are discussed in section ?? and ??. Section ?? discusses DoLittle's domain description language. The search control method developed to solve problem ?? is shown in section ??.

The fulfillment of those requirements is tested by an empirical evaluation of a multi-strategy planning system in a set of domains. Chapter ?? shows that DoLittle is able to solve difficult problems in the kitchen domain.

5.3 DoLittle's representation of planning strategies

Clearly, to be able to use different planning strategies on a single problem, a planner must be able to represent them. This section describes *general*

operators, a generalization of traditional STRIPS operators that allows the representation of a large class of planning strategies. Subsections ?? to ?? show examples of how this representation is used.

Definition ?? of a planning strategy in section ?? is too general to be practical and to support efficient planning. However, in chapter ??, the plan space paradigm was introduced as a unified framework for planning. In this framework, different planning strategies were categorized according to their plan representation language and their set of plan transformations (see table ??). The general operator representation is based on the assumption that a planning strategy can be described by its applicability conditions and its set of plan transformations.

Ignoring partial-order planning and concurrent plans in case-based planning, the plan representation language $\mathcal{L}_{\mathcal{P}}$ has to be able to represent the following elements:

- Total order of operators
- Instantiated variables
- Plan skeleton
- Trees of problem spaces.

A plan skeleton is equal to a plan head and a plan tail where the current operator may move backwards or forwards instead of only forwards. Although, the problem solving strategies described in table ?? only require uniform trees, it seems reasonable to expand this criteria to include also non-uniform trees.

The following set of plan transformations T can be extracted from the comparison of different planning strategies shown in chapter ??:

- Move current operator
- Insert operator sequence

- Remove operator sequence
- Reorder operator sequence
- Replace operator sequence
- Change variable binding
- Create subgoal problem space
- Create serial subgoal problem space
- Create abstract subgoal problem space

The requirement to support macro-based planning requires that a multi-strategy planner can use operator sequences instead of a single operators. Insertion of an operator sequence is a generalization of appending and prepending operators. Furthermore, the current operator may be moved anywhere in the current plan, not just forward.

Practical planners have greatly extended the representation of operators by adding condition types to predicates in the preconditions of an operator, for example O-Plan2, Sipe, and Act. However, this additional information is hard to extract from a problem trace alone without additional domain knowledge.

5.4 DoLittle's decision procedure

First, the combination of different planning strategies on a single problem makes it necessary to be able to determine when a given planning strategy should be applied. As is shown in the analysis in chapter **??**, the cost of the decision procedure D is multiplied by the number of planning strategies for each reduction. The design of the decision procedure depends on the allowable cost and the required fit between the Good and Certain sets of a planning strategy.

DoLittle bases the decision procedure on applicability conditions. The applicability conditions refer to the current state of the planning process, e.g., the current state, the goal that the planner is trying to achieve, the current operator and binding, background knowledge about the domain. In the remainder of this subsection, this thesis refers to this as the planner state. Although, as will be described in the following paragraphs, the representation of the planner state uses the same methods as that of primitive operators, it is important to distinguish between the two, since one refers to the planner state (a meta state), the other one describes the state of the world. There are many different features of a planning process that may be useful in determining what planning strategy to apply, including the world state, the goal the planner is trying to achieve, the current problem space, the operator and its binding, the subgoal hierarchy, the results of the indexer, the current plan, the set of rejected plans.

Many of these features depend on the specific planning algorithm used. For example, there is no corresponding concept of subgoal hierarchy (means-ends analysis) in case based planning and vice versa for results of the indexer. Using those features would make it impossible to apply the general operator representation to other planning strategies. Therefore, DoLittle's operator representation is based on a common subset of planner state features: the current state, the set of goals the planner is trying to achieve (open goals), and the operators DoLittle is currently refining.

5.5 Description of General Operators

This section describes *general operators*, the representation of planning strategies used in this thesis.

Definition 5 (General Operator) *A **general operator** O describes the applicability conditions for a set of planning strategies. The applicability conditions of a general operator are based on the current problem solving*

context, the current state, and the set of open goals. It contains the following elements:

1. *an operator name, that must be unique.*
2. *a list of free variables.*
3. *a possibly empty set of context operators, that limit the problem spaces in which an operator can be applied.*
4. *a set of preconditions, that identify the set of current states under which the associated planning biases are applicable.*
5. *a set of open goals. The set of open goals form an implicit conjunction. The general operator can only be applied to planner states that contains all of its open goals.*
6. *a set of effects, that specify how the world is changed through application of this operator.*
7. *a set of one or more refinements, which represent planning strategies.*

General operators assume that the important characteristics of the planner state are: (a) the current problem space, (b) the current world state, and (c) the set of goals the planner is trying to achieve. As discussed in section **??** those features are applicable in all described strategies.

A general operator may have a context associated with it. For example, if general operator GOP2 has a context entry of (CONTEXT GOP1), GOP2 can only be used as part of finding a refinement for GOP1. This allows DOLITTLE to restrict the problem spaces in which an operator may be applied. Since the context can be checked quickly (simply traverse the plan structure), it also allows DOLITTLE to limit the match cost of an operator. The default is the empty context list, in which case the general operator is applicable in all problem spaces.

The applicability conditions identify sets of world states and sets of goals, in which the described planning strategy is appropriate. The STRIPS representation already presents a method for describing sets of world states through preconditions. Furthermore, general operators contain a conjunction of open goals. Although not necessary, DOLITTLE's learner restrict the set of open goals to be a subset of the effects of a general operator, since it provides better evidence that the operator is useful than only indirect achievement of a goal. If the set of open goals is a subset of the effects, it is guaranteed that the general operator will achieve at least one open goal. The special open goal (INVALID) is used to disallow retrieval of an operator by the plan transformations. Then the operator can only be used as part of a refinement. It can not be added or inserted on its own, but only as part of a different refinement's operator sequence. See subsection **??** for an example.

Since the same operator sequence may occur more than once as refinement of a general operator, applicability conditions support unlimited disjunctions. The representation of the applicability conditions is distributed over a number of general operators. However, trivial disjunctions are problematic since they do not allow efficient generalization and comparison of different applicability conditions. Therefore, preconditions and open goals form an implicit generalization hierarchy. Preconditions may be generalized by (a) replacing objects with variables, (b) changing the type of a variable, (c) replacing a literal in the preconditions with a disjunction, and/or (d) dropping terms from the conjunction of literals. Similarly, open goals are generalized by dropping goals.

A set of preconditions and open goals, however, is incomplete, because it is unable to describe negations of goals, such as working on goal G_1, but not on goal G_2. This ability is important because sometimes, a correct description is easier to represent as a counter factual, that is, it is easier to describe what states a planning bias should not be applied to.

DOLITTLE represents this type of applicability conditions through a modified version space algorithm developed by me. A *version space* is a concept learning technique developed by Tom Mitchell [?]. A version space orders the set of possible concepts according to generality. It maintains a most specific (S-set) and most general boundary (G-set) in this version space. Concepts that are below the most specific boundary are not general enough, concepts that are more general than the most general boundary are too general. The two boundaries converge, until only a single concept is left. If the version space collapses, that is there are no more concepts left, there is no concept.

One problem of version spaces is that the most general boundary G-sets may grow exponentially. The *symmetric version space* algorithm overcomes the problem of possibly infinite G-sets and has successfully been used in the string learner SHELL-CLERK. This subsection describes briefly the implementation of the symmetric version space algorithm in DOLITTLE, a more detailed description of the algorithm and the string learner SHELL-CLERK can be found in the [?]. The main idea is that the symmetric version space algorithm maintains a most specific description of all negative examples, instead of a most general description of all positive examples. If the two descriptions share an element, the closeness of the fit between the element and the description is computed and the classification is based on which description yields the closer match.

Therefore, in DOLITTLE, the search control method only retrieves the closest fit for a planning strategy. That is, if general operators OP1 and OP2 both match the current planner state, and OP1 is a generalization of OP2, then only OP2 is selected. Thus the condition "goal G_1 and not goal G_2" is represented by a pair of applicability conditions, one that only has G_1 in its set of open goals, and one that has G_1 and G_2.

A more detailed example of DOLITTLE's applicability conditions and its similarity metric is shown in subsection ??.

Since only the closest match is retrieved, the representation of applicability conditions is complete with respect to boolean formulae; in theory, all applicability conditions based on the context, the current state, and the set of open goals of the planner can be represented through general operators. The representation allows conjunction (implicit), disjunction (distributed general operators), and negation (closest match) of current state and goal predicates.

The reason the representation does not use boolean formulae is that the representation is biased towards independent subgoals and assumes that there is no harmful subgoal interaction. This allows a planner to break the original goal conjunct into smaller subproblems and therefore this assumption is made by most planning systems. The representation predicts that after achieving a goal G_1, the remaining one G_2 can be solved without interfering with the plan for the previous subgoal G_1. However, if two subgoals G_1 and G_2 interfere, a general operator with both open goals G_1 and G_2 must be created. DoLittle guarantees that in this case, only the operator corresponding to G_1 and G_2 is retrieved. In other words, DoLittle assumes that there is no subgoal interaction unless it has evidence to the contrary.

5.5.1 Representation of planning strategies

The previous subsection describes the applicability conditions of a planning strategy, that is *when* to apply a given strategy. The planning strategy itself, that is *what* to apply, is represented by a set of *refinements*.

The transformation set discussed in section **??** contains the following plan modification operators: moving the current focus anywhere in the plan, insertion, removal, reordering, and replacement of operator sequences and creation of simple, serial, and abstract problem spaces. To simplify comparison of different planning systems, DoLittle allows the user to assign a refinement type to limit the set of transformation operators. For example, if an operator sequence is typed as a MACRO, only appending and prepending of the whole sequence are allowed, not adaptation of the sequence. DoLittle's default

refinement type is GENERIC, which allows use of all plan transformations.

The design uses *refinements* to capture the set of plan transformations that should be applied:

Definition 6 (Refinement) *A **refinement** R of a parent operator O is a set of preconditions and effects and a possibly empty sequence of primitive or general operators. It contains the following elements:*

1. *a type, to specify the type of the refinement: e.g, macro, case, search space, serial search space, abstract search space, or generic. The special refinement type failure forces* DoLittle *to backtrack.*
2. *a list of free variables, which is a superset of the parent's variables.*
3. *a set of preconditions, which is a specialization of the preconditions of the parent operator O. The preconditions must guarantee that the refinement is applicable, but may be overconstrained.*
4. *a set of effects, which is a specialization of the parent operator's effects.*
5. *a possibly empty sequence of primitive/general operators.*

A case is represented as a sequence of instantiated primitive operators of type case. A refinement represents macros as a sequences of primitive operators with parameterized arguments of type macro. An abstract operator is represented by two or more refinements with specialized preconditions and effects. Since the two refinements belong to the same parent operator, they share the same preconditions and effects as the parent operator, but may have different additional preconditions and effects.

An important point is that the preconditions of a refinement may be overconstrained. Since, the applicability conditions of the general operator must be more general than those of any refinement, the preconditions of the refinement specify a most specific boundary for the applicability conditions. Adding an additional literal or forcing an instantiation of a variable

may produce correct applicability conditions, that would otherwise not be possible.

For example, assume that the refinement is the following operator sequence: MOVE-ROBOT AT-TABLE AT-SINK. PUT-IN-MICROWAVE CUP. This sequence is applicable, independently of whether the Cup contains water or not. However, if this refinement is part of a general operator to heat water, then it should only be used if the cup contains water. However, this can not be represented by the applicability conditions, because the preconditions of the general operator would be more specific than those of the refinement, which violates the definition of a refinement. This problem is solved by allowing additional preconditions for refinements, so called constraints. Then (CONTAINS CUP WATER) can be added to the preconditions of the refinement and the general operator. Although the operator sequence is applicable with and without water, DOLITTLE will only use it if the cup contains water.

A more detailed description of the different types of refinements and how DOLITTLE's search algorithm deals with them is given in subsections ?? to ??.

The following general operator is an example from the kitchen domain and illustrates the key features:

General operator example

GEN-PICK-UP-FROM-CUPBOARD
Variables $OBJECT
Context
Preconds (ARM-EMPTY)
(IS-AT ROBBY AT-TABLE)
(IS-IN $OBJECT CUPBOARD)
Open goals (HOLDING $OBJECT)
Effects (HOLDING $OBJECT)
(NOT (IS-IN $OBJECT CUPBOARD))
(NOT (ARM-EMPTY))
Refine. 1 PICK-UP-FROM-CUPBOARD($OBJECT)

Refine. 2 OPEN-DOOR(CUPBOARD)
PICK-UP-FROM-CUPBOARD($OBJECT)

The general operator GEN-PICK-UP-FROM-CUPBOARD can be used to pick up an object from the cupboard independent of whether the cupboard is open or not. First, the context of the operator is empty, which means that it can be used in any problem space. If for example, the context contains the two operators GEN-MAKE-TEA and GEN-MAKE-COFFEE, the general operator can only be used if refining either GEN-MAKE-TEA *or* GEN-MAKE-COFFEE. In the remainder of the thesis, empty context fields are omitted for readability. Secondly, as described in subsection **??**, the preconditions and open goals refer to the planner state instead of the world state. The preconditions of the operator establish that the planning strategies described in the refinements are applicable, if the current world state matches them, i.e., the arm is empty, the robot is at the table, and $OBJECT is in the cupboard. Furthermore, one goal that the planner is trying to achieve is (HOLDING $OBJECT). Adding another literal to the set of open goals results in a conjunction. To represent a disjunction, a new general operator must be created:

General operator example

GEN-PICK-UP-FROM-CUPBOARD-2
Variables $OBJECT
Context
Preconds (ARM-EMPTY)
(IS-AT ROBBY AT-TABLE)
(IS-IN $OBJECT CUPBOARD)
Open goals (NOT (IS-IN $OBJECT CUPBOARD))
Effects (HOLDING $OBJECT)
(NOT (IS-IN $OBJECT CUPBOARD))
(NOT (ARM-EMPTY))
Refine. 1 PICK-UP-FROM-CUPBOARD($OBJECT)

Refine. 2 OPEN-DOOR(CUPBOARD)
PICK-UP-FROM-CUPBOARD($OBJECT)

The two general operators together represent that the planning strategies described in their common set of refinements are applicable if the planner is trying *either* to achieve (HOLDING $OBJECT) or to negate (IS-IN $OBJECT CUPBOARD).

General operator example (continued)

REFINE. 1: MACRO
Variables $OBJECT
Preconds same as parent plus
(IS-OPEN CUPBOARD)
Effects (HOLDING $OBJECT)
(IS-IN $OBJECT CUPBOARD)
(ARM-EMPTY)
Sequence PICK-UP-FROM-CUPBOARD($OBJECT)

The first refinement consists of the single primitive operator PICK-UP-FROM-CUPBOARD. This refinement has the additional precondition that the cupboard must be open when picking up the object. The deletion of (IS-IN $OBJECT CUPBOARD) and (ARM-EMPTY) is a side-effect of PICK-UP-FROM-CUPBOARD.

General operator example (continued)

REFINE. 2: MACRO
Variables $OBJECT
Preconds same as parent plus
(NOT (IS-OPEN CUPBOARD))
Effects (IS-OPEN CUPBOARD)
(HOLDING $OBJECT)
(NOT (IS-OPEN CUPBOARD))

(NOT (IS-IN $OBJECT CUPBOARD))
(NOT (ARM-EMPTY))
Sequence OPEN-DOOR CUPBOARD
PICK-UP-FROM-CUPBOARD $OBJECT

The second refinement consists of the primitive operator sequence (OPEN-DOOR(CUPBOARD), PICK-UP-FROM-CUPBOARD $OBJECT). Since to be able to open a door, the door must be closed previously, the second refinement has the additional precondition of the cupboard door being closed. There is an additional effect describing that the cupboard door is open after application of this operator sequence.

The following subsections discuss some important planning strategies and their representation as general operators in DoLittle. To make the operators more readable, the parameter lists are omitted, since they can easily be derived from the preconditions and effects. Also in some instances, the preconditions and effects of the parent operator are not repeated in the refinements. Additional effects and preconditions in refinements are underlined for readability.

5.5.2 Forward chaining

This subsection describes representation of a forward chaining planner in DoLittle. A forward chaining planner only supports appending an operator to the plan head. Also, a operator need not be relevant to an open goal.

Table ?? is an example of an operator that forces DoLittle to use forward chaining rather than means-ends analysis. The primitive operator Pick-Up-From-Table is wrapped into a general operator. The preconditions of the general operator are identical to the preconditions of the primitive operator, but the set of open goals of the general operator is empty. This means that the operator is only applicable if the preconditions match the current state, but its applicability is independent of the set of open goals.

Table 5.1: Forward chaining

General Operator: PICK-UP-FROM-TABLE-FC
Preconditions: (ARM-EMPTY)
(IS-AT ROBBY AT-TABLE)
(IS-ON $OBJECT TABLE)
Open goals: *null*
Effects: (HOLDING $OBJECT)
(NOT (ARM-EMPTY))
(NOT (IS-ON $OBJECT TABLE))
Refine. 1: MACRO
Preconditions: (ARM-EMPTY)
(IS-AT ROBBY AT-TABLE)
(IS-ON $OBJECT TABLE)
Effects: (HOLDING $OBJECT)
(NOT (ARM-EMPTY))
(NOT (IS-ON $OBJECT TABLE))
Sequence: PICK-UP-FROM-TABLE $OBJECT

In this case, DOLITTLE behaves identical to a forward chaining planner. Depending on the plan retrieval method M, it will result in a depth-first or breadth-first planner.

Operator PICK-UP-FROM-TABLE-FC and the primitive operator PICK-UP-FROM-TABLE differ in one important characteristic: PICK-UP-FROM-TABLE-FC can be added to the plan only when its preconditions are satisfied. DOLITTLE may not subgoal on its preconditions. PICK-UP-FROM-TABLE can be added if at least one effect matches an open goal. DOLITTLE can subgoal on the preconditions.

5.5.3 Macros

In macro-based planning, a macro compiles a sequence of primitive operators into a new operator which is added to the operator set. In general, the

operator may be applied in lieu of a primitive operator, and the operator sequence must not be adapted to the new situation. Using DoLittle's representation, a macro is defined as a set of general operators with identical preconditions, but differing sets of open goals. The general operators share the same MACRO refinement, that has identical preconditions and effects to the top level operator.

The following example from the kitchen domain illustrates DoLittle's representation of a macro to fill a cup of water. The macro compiles the following operator sequence into a single operator: MOVE-ROBOT AT-TABLE AT-SINK, PUT-IN-SINK $Cup, FILL-WITH-WATER $Cup, TURN-WATER-OFF, PICK-UP-FROM-SINK $Cup, MOVE-ROBOT AT-SINK AT-TABLE. Instead of adding this macro to all problem spaces, the macro is restricted to be only used in the context of either making tea or making instant coffee. The other applicability conditions of a macro are equivalent to those of a primitive operator (means-ends analysis), that is DoLittle may subgoal on any of the operator sequence's precondition, and only one of the operator sequence's effects needs to be on the open goal list. Since the operator sequence has two effects ((CONTAINS $Cup WATER) and (NOT (CONTAINS $Cup NOTHING))), DoLittle represents the macro as two general operators, one for each effect (table ??). DoLittle is prevented from adapting the underlying operator sequence by its refinement type MACRO.

This example also illustrates that since DoLittle separates the applicability conditions from the planning strategy, it allows control over when to apply a macro operator. For example, it may be more efficient to instantiate partially the top level preconditions to at least include the (NEXT-TO) constraints, since they can not be changed by any domain operator, and it is futile to subgoal on them. Furthermore, if the macro operator should only be used to achieve (CONTAINS $Cup WATER), the second general operator FILL-CUP-WITH-WATER-M2 may be omitted.

Table 5.2: Macro operator: Fill cup with water

General Operator: FILL-CUP-WITH-WATER-M1
Context: MAKE-TEA, MAKE-INSTANT-COFFEE
Preconditions: *null*
Open goals: (CONTAINS $CUP WATER)
Effects: (CONTAINS $CUP WATER)
(NOT (CONTAINS $CUP NOTHING))
Refine. 1: MACRO
Preconditions: (IS-AT ROBBY AT-TABLE)
(NEXT-TO AT-TABLE AT-SINK)
(HOLDING $CUP)
(CONTAINS $CUP NOTHING)
(SINK-EMPTY)
(WATER-OFF)
Effects: (CONTAINS $CUP WATER)
(NOT (CONTAINS $CUP NOTHING))
Sequence: MOVE-ROBOT AT-TABLE AT-SINK
PUT-IN-SINK $CUP
FILL-WITH-WATER $CUP
TURN-WATER-OFF
PICK-UP-FROM-SINK $CUP
MOVE-ROBOT AT-SINK AT-TABLE

General Operator: FILL-CUP-WITH-WATER-M2
Context: MAKE-TEA, MAKE-INSTANT-COFFEE
Preconditions: *null*
Open goals: (NOT (CONTAINS $CUP NOTHING))
Effects: (CONTAINS $CUP WATER)
(NOT (CONTAINS $CUP NOTHING))
Refine. 1: MACRO
< identical to Refine. 1 of FILL-CUP-WITH-WATER-M1 >

5.5.4 Cases

A case is similar to a macro operator, with the following important distinctions: (a) its applicability conditions are not generalized, and (b) the operator sequence may be adapted to the new situation. Using DoLittle's representation a case consists of a general operator with preconditions and effects that determine the indexing of the case. The top level preconditions and open goals specify under what conditions this case is retrieved. DoLittle's similarity metric is based on the match between those preconditions and open goals, and the current planner state. Subsection ?? describes DoLittle's case retrieval and indexing method. The general operator has one CASE refinement with preconditions and effects derived from the operator sequence.

Table ?? is a case derived from the making tea plan shown in table ??. The preconditions of the top level operator and the refinement refer to the preconditions of the complete operator sequence, which are roughly equivalent to the initial state. The plan solves the problem of (CONTAINS $CUP TEA), which is therefore the only literal in the open goal list. The other effects of the operator sequence are considered to be side effects. The only refinement is of type CASE, which allows DoLittle to adapt it by changing variable bindings, and/or inserting, removing, replacing, or reordering operators.

5.5.5 Abstract operators

This subsection describes how DoLittle can create abstract operators or problem spaces by dropping preconditions and effects from an operator. An abstract operator consists of a general operator with abstract preconditions and effects. Associated with the general operator is a set of refinements with more specific preconditions and effects. The type of refinement may be MACRO, CASE, SUBGOAL, SERIAL SUBGOAL, or ABSTRACT SUBGOAL. Of particular interest is the ABSTRACT SUBGOAL refinement,

Table 5.3: Case operator: Make a cup of tea

General Operator: MAKE-TEA
Preconditions: < *initial state* >
Open goals: (CONTAINS $CUP TEA)
Effects: (CONTAINS $CUP TEA)
(NOT (CONTAINS $CUP NOTHING))
(OPEN MICROWAVE)
(IS-ON TEA-BOX TABLE)
...
Refine. 1: CASE
Preconditions: < *initial state* >
Effects: identical to parent effects
Sequence: OPEN-DOOR CUPBOARD
...
PUT-IN-GARBAGE-CAN OLD-TEA-BAG

which supports the creation of ordered monotonicity abstraction hierarchies. An ABSTRACT SUBGOAL problem space rejects any plan that affects the literal types mentioned in the preconditions and effects of the top level operator with the exception of establishing an effect.

Table **??** is an example of an abstract operator in the kitchen domain. It drops (IS-AT ROBBY AT-STOVE) and (IS-OPEN MICROWAVE) from the preconditions of PUT-IN-MICROWAVE. This general operator enables DOLITTLE to put a cup into the microwave independently of whether it is closed or whether the robot is at the stove. The first refinement simply applies operator PUT-IN-MICROWAVE and has the additional preconditions (IS-OPEN MICROWAVE) and (IS-AT ROBBY AT-STOVE). The second refinement consists of an adaptable operator sequence. The robot first puts the cup on the table, opens the microwave door, and picks up the cup before putting it in the microwave. The preconditions of this refinement have the additional literals (NEXT-TO AT-STOVE AT-TABLE), (IS-AT ROBBY AT-STOVE), and

(NOT (IS-OPEN MICROWAVE)). To allow DOLITTLE further adaptation of this operator sequence, it's typed as a CASE refinement. The third refinement creates an abstract problem space if the robot is not at the stove. Since the problem space is typed as an ABSTRACT SUBGOAL, none of the literal types in the top level preconditions or effects may be affected. This means that the search space is constrained to rule out any plans that for example include putting something on the table. The third refinement has the additional precondition (NOT (IS-AT ROBBY AT-STOVE)) and also an additional effect (IS-AT ROBBY AT-STOVE).

5.5.6 Automatic subgoaling

Automatic subgoaling assumes that a problem can be broken down into smaller subproblems by identifying a series of goal predicates that must be achieved to solve the problem. The planner solves the resulting subproblems from left to right. An automatic subgoaling planning strategy in DOLITTLE is represented by a general operator with one refinement that specifies the order of the subgoals that have to be achieved. Furthermore, there is one general operator for each subgoal. Subgoal operator i has subgoals $0, \ldots, i-1$ as its preconditions and subgoal i as its effect. Each subgoal operator contains either a SUBGOAL or SERIAL SUBGOAL refinement, depending on whether previous subgoals are protected or not.

For example, in the kitchen domain, any problem that contains the goal literal (CONTAINS $CUP COFFEE) (as opposed to instant coffee) can be broken down into a series of subgoals: (a) (CONTAINS COFFEE-MAKER WATER) and (b) (CONTAINS COFFEE-MAKER COFFEE-GRAIN), and (c) (CONTAINS $CUP COFFEE).

Table **??** is an example of the general operator to make coffee with the coffee-maker. Since the automatic subgoaling is applicable in all states, the preconditions of the top level general operator MAKE-COFFEE are empty. There is only one goal, the cup contains coffee. The only refinement is a

Table 5.4: Abstract operator: Make a cup of tea

General Operator: ABSTRACT-PUT-IN-MICROWAVE
Preconditions: (HOLDING $OBJECT)
(MICROWAVE-EMPTY)
Open goals: (IS-IN $OBJECT MICROWAVE)
Effects: (IS-IN $OBJECT MICROWAVE)
(ARM-EMPTY)
(NOT (MICROWAVE-EMPTY))
(NOT (HOLDING $OBJECT))
Refine. 1: MACRO
Preconditions: same as parent plus
(IS-OPEN MICROWAVE)
(IS-AT ROBBY AT-STOVE)
Effects: same as parent
Sequence: PUT-IN-MICROWAVE $OBJECT
Refine. 2: CASE
Preconditions: same as parent plus
(IS-AT ROBBY AT-STOVE)
(NEXT-TO AT-STOVE AT-TABLE)
(NOT (IS-OPEN MICROWAVE))
Effects: same as parent plus
(IS-OPEN MICROWAVE)
Sequence: MOVE-ROBOT AT-STOVE AT-TABLE
...
PUT-IN-MICROWAVE $OBJECT
Refine. 3: ABSTRACT SUBGOAL
Preconditions: same as parent plus
(IS-OPEN MICROWAVE)
(NOT (IS-AT ROBBY AT-STOVE))
Effects: same as parent plus
(IS-AT ROBBY AT-STOVE)

sequence of two general operators, corresponding to the sequence of subgoal predicates. Each of those operators corresponds to a subgoal refinement. The first subgoal operator MAKE-COFFEE-SG1 contains the open goal (INVALID). This means that it can only be used by reference in an refinement. For example, if the current goal is (CONTAINS COFFEE-MAKER WATER), DOLITTLE will not retrieve MAKE-COFFEE-SG1, whereas MAKE-COFFEE-SG2 is a candidate for putting coffee-grain into the coffee-maker.

The preconditions of operator MAKE-COFFEE-SG2 consist of the preconditions of all previous subgoals and the top level operator. Its effect is the single goal predicate (CONTAINS $CUP COFFEE). Depending on the type of problem space, the refinements of the subgoal general operators MAKE-COFFEE-SG[1...3] may be of type SUBGOAL or SERIAL-SUBGOAL. If the refinement is of type SERIAL-SUBGOAL, the preconditions of the refinements must not be affected. For example, when DOLITTLE refines MAKE-COFFEE-SG2, the literal (CONTAINS COFFEE-MAKER WATER) must not be changed.

5.5.7 Reactive rules

Briefly, a reactive rule is a planning strategy based on the planning bias that for every initial state and every goal predicate, there is one operator that leads the planner towards the goal. This type of planning is very efficient since there is no search needed [**?**, **?**]. The planner looks up the correct operator to execute. This lookup in DOLITTLE is represented by a general operator with a MACRO refinement. The refinement consists of two operators, first the primitive operator, and secondly a general operator that corresponds to the remaining subproblem. The preconditions of the general operator are identical to those of the refinement. The preconditions of the remaining subproblem operator are the postconditions of the primitive operator.

In the kitchen domain, a reactive rule may specify that any time the goal is to have a cup of hot water, and the robot is in front of the closed

Table 5.5: Automatic subgoaling: Making coffee in the kitchen domain

General Operator: MAKE-COFFEE
Preconditions: *null*
Open goals: (CONTAINS $CUP COFFEE)
Effects: (CONTAINS $CUP COFFEE)
Refine. 1: MACRO
Preconditions: *null*
Effects: (CONTAINS $CUP COFFEE)
Sequence: MAKE-COFFEE-SG1 $CUP
MAKE-COFFEE-SG2 $CUP

General Operator: MAKE-COFFEE-SG1
Preconditions: *null*
Open goals: (INVALID)
Effects: (CONTAINS COFFEE-MAKER WATER)
(CONTAINS COFFEE-MAKER COFFEE-GRAIN)
Refine. 1: SUBGOAL
Preconditions: *null*
Effects: (CONTAINS COFFEE-MAKER COFFEE)
Sequence:

General Operator: MAKE-COFFEE-SG2
Preconditions: (CONTAINS COFFEE-MAKER WATER)
(CONTAINS COFFEE-MAKER COFFEE-GRAIN)
Open goals: (CONTAINS $CUP COFFEE)
Effects: (CONTAINS $CUP COFFEE)
Refine. 1: SERIAL-SUBGOAL
Preconditions: (CONTAINS COFFEE-MAKER WATER)
(CONTAINS COFFEE-MAKER COFFEE-GRAIN)
Effects: (CONTAINS $CUP COFFEE)
Sequence:

microwave and its arm is empty, it should open the microwave, presumably since we have to put the cup into the microwave sometime during the plan. Table ?? is an example of this reactive rule in DoLittle's kitchen domain. The refinement of the top level general operator GET-HOT-WATER consists of the primitive operator OPEN-DOOR and the general operator GET-HOT-WATER-SG corresponding to the remainder of the problem. It is important to note that the preconditions of the top level operator are identical to the preconditions of the operator OPEN-MICROWAVE. DoLittle will behave opportunisticly, it will only open the microwave door if the preconditions are already true in the current state. Another possibility would be to have an empty set of preconditions; then DoLittle will subgoal on the unsatisfied preconditions of the OPEN-MICROWAVE operator. Since reactive rules are a form of forward chaining reasoning, DoLittle is prevented from subgoaling on preconditions by promoting the preconditions of the refinement to the applicability conditions of the top level operator.

5.5.8 Backward chaining

Backward chaining is similar to a reactive rule, but instead of extending the plan from the initial state, the planner propagates the goal conditions through a primitive operator. The new goal conditions are the conditions that guarantee that the suffix operator is applicable and will achieve the goal state. A backward chaining planning strategy is represented similarly to a reactive rule. The only difference is that the primitive operator and the remaining subgoal operator are reversed and the preconditions and effects are changed accordingly.

Assume in the following example (table ??) that the goal is to fill object $OBJECT with water. The general operator GEN-FILL-WITH-WATER instructs DoLittle to first satisfy the preconditions of operator FILL-WITH-WATER using the general operator PUT-IN-SINK-SG and then apply operator FILL-WITH-WATER.

Table 5.6: Reactive rule: Get hot water

General Operator: GET-HOT-WATER
Preconditions: (IS-REACHABLE MICROWAVE AT-FRIDGE)
(IS-AT ROBBY AT-FRIDGE)
(ARM-EMPTY)
(NOT (IS-OPEN MICROWAVE))
Open goals: (CONTAINS $CUP HOT-WATER)
Effects: (CONTAINS $CUP HOT-WATER)
(NOT (CONTAINS $CUP NOTHING))
Refine. 1: MACRO
Preconditions: (IS-REACHABLE MICROWAVE AT-FRIDGE)
(IS-AT ROBBY AT-FRIDGE)
(ARM-EMPTY)
(NOT (IS-OPEN MICROWAVE))
Effects: (CONTAINS $CUP HOT-WATER)
Sequence: OPEN-DOOR MICROWAVE
GET-HOT-WATER-SG $CUP

General Operator: GET-HOT-WATER-SG
Preconditions: (IS-REACHABLE MICROWAVE AT-FRIDGE)
(IS-AT ROBBY AT-FRIDGE)
(ARM-EMPTY)
(IS-OPEN MICROWAVE)
Open goals: (INVALID)
Effects: (CONTAINS $CUP HOT-WATER)
(NOT (CONTAINS $CUP NOTHING))
Refine. 1: SUBGOAL
Preconditions: (IS-REACHABLE MICROWAVE AT-FRIDGE)
(IS-AT ROBBY AT-FRIDGE)
(ARM-EMPTY)
(IS-OPEN MICROWAVE)
Effects: (CONTAINS $CUP HOT-WATER)
Sequence:

Table 5.7: Backward chaining: Fill with Water

General Operator: GEN-FILL-WATER
Preconditions: *null*
Effects: (CONTAINS $OBJECT WATER)
Refine. 1: MACRO
    Preconditions: *null*
    Open goals: (CONTAINS $OBJECT WATER)
    Effects: (CONTAINS $OBJECT WATER)
    Sequence: PUT-IN-SINK-SG $OBJECT
              FILL-WITH-WATER $OBJECT

General Operator: PUT-IN-SINK-SG
Preconditions: *null*
Open goals: (INVALID)
Effects: (NOT (WATER-ON))
         (IS-AT ROBBY AT-SINK)
         (ARM-EMPTY)
         (IS-IN $OBJECT SINK)
         (CONTAINS $OBJECT NOTHING)
Refine. 1: SUBGOAL
    Preconditions: *null*
    Effects: (NOT (WATER-ON))
             (IS-AT ROBBY AT-SINK)
             (ARM-EMPTY)
             (IS-IN $OBJECT SINK)
             (CONTAINS $OBJECT NOTHING)
    Sequence:

5.5.9 Problem Specification

An interesting ability of case-based planners is to enhance the problem specification and add additional goals to avoid failure and increase the efficiency of the case retriever. DoLittle's general operators provide an easy and efficient method for predictiing failure: a general operator whose refinement consists of a problem specification operator. The preconditions of the refinement and the problem specification operator are identical to the top level preconditions, but there are additional effects in the refinement and problem specification operator that correspond to the additional goals.

Assume that DoLittle was asked to create a plan to make a cup of hot milk. During the execution of the plan, DoLittle put the honey into the fridge as opposed to the shelf, and therefore, the honey is too hard at the end of the plan. It is important to note that in this case, the failure is not because of an incomplete domain description or uncertainty in the domain actions, but because of an incomplete problem specification. Table ?? is an example of a general operator that adds an additional literal (NOT (IS-IN HONEY-JAR FRIDGE)) to the set of goals.

5.5.10 Avoiding failure

DoLittle provides a special refinement type FAILURE to avoid unsuccessful search branches. A FAILURE refinement simply forces DoLittle to abandon the search branch and backtrack. Its representation consists of a general operator with a FAILURE refinement. This operator will force DoLittle to backtrack any time it is in a planner state specified in its applicability conditions.

For example, assume that DoLittle is trying to make coffee at the sink instead of the table. Although all ingredients could be moved, there is no operator in the kitchen domain that allows the robot to move the coffee maker, and thus this plan is doomed to failure. However, detecting this type

Table 5.8: Problem specification: Do not put honey in the fridge

General Operator: GEN-MAKE-HOT-MILK
Preconditions: *null*
Open goals: (CONTAINS $OBJECT HOT-MILK)
Effects: (CONTAINS $OBJECT HOT-MILK)
Refine. 1: MACRO
    Preconditions: *null*
    Effects: (CONTAINS $OBJECT HOT-MILK)
        (NOT (IS-IN HONEY-JAR FRIDGE))
    Sequence: AVOID-HARD-HONEY

General Operator: AVOID-HARD-HONEY
Preconditions: *null*
Open goals: (INVALID)
Effects: (CONTAINS $OBJECT HOT-MILK)
    (NOT (IS-IN HONEY-JAR FRIDGE))
Refine. 1: SUBGOAL
    Preconditions: *null*
    Effects: (CONTAINS $OBJECT HOT-MILK)
        (NOT (IS-IN HONEY-JAR FRIDGE))
    Sequence:

Table 5.9: Avoiding failure: No way to move the coffee maker

General Operator: MOVE-COFFEE-MAKER-FAILURE
Preconditions: *null*
Open goals: (IS-IN COFFEE-MAKER SINK)
Effects: (IS-IN COFFEE-MAKER SINK)
Refine. 1: FAILURE
    Preconditions: *null*
    Effects: (IS-IN COFFEE-MAKER SINK)
    Sequence: FAILURE

of failure may be expensive, since the search tree will have to be searched exhaustively. Furthermore, an analysis of static predicates (i.e., predicates whose truth value is not changed by any operator) in the search space will not detect it since some of the objects are movable. In table **??**, MOVE-COFFEE-MAKER-FAILURE forces DOLITTLE to backtrack whenever DOLITTLE is trying to achieve the goal (IS-IN COFFEE-MAKER SINK).

5.5.11 Discussion

This subsection compares DoLittle's general operator representation to other operator representation languages. DOLITTLE trades off expressibility and simplicity of learning. For example, PRODIGY allows the user to create meta-predicates and use them in the design of control rules.

DOLITTLE augments the operators with meta-knowledge about when to apply a given operator. The applicability conditions consist of the current context, current state, and the open goals. A general operator contains the applicability conditions for all its refinements.

SIPE is a practical planning system, which also uses a powerful operator representation [**?**]. It's representation is similar to other practical planning systems, e.g. ACT. For details on SIPE's operator representation, refer back to subsection **??**. Comparing DOLITTLE's representation to SIPE's represen-

tation shows the following differences:

- SIPE's plans are partial-order plans. DOLITTLE's plans are totally ordered plans.
- SIPE encodes applicability conditions as preconditions and the purpose of an operator. DOLITTLE extends the applicability conditions to include also the current problem space.
- SIPE does not allow the negation or conjunction of open goals. DOLITTLE supports negation and conjunction in the set of open goals.
- SIPE supports types and constraints on variables. DOLITTLE only allows typing of variables.
- SIPE supports temporal reasoning and reasoning about resources. DOLITTLE does neither support temporal reasoning nor reasoning about resources. This may be added in the future.
- SIPE's plot may contain process, choice process, and goal nodes. DOLITTLE supports process nodes similarly to SIPE. There are no explicit choice nodes in DOLITTLE's representation, although they can be represented by a set of refinements. DOLITTLE may generate any one of three different types of goal nodes: serial subgoals, abstract subgoals, and subgoals.
- SIPE supports conditional effects through a deductive causal theory. DOLITTLE supports conditional effects as part of the operator description.
- SIPE can not constrain the plan transformation methods. DOLITTLE can constrain the plan transformations by, for example, typing a refinement as a macro or a case.

The comparison shows that DoLittle's and Sipe's operator representations have different focuses. Sipe focuses on adding special purpose reasoning, in particular temporal and resource reasoning. Sipe is based on the assumptions that (a) the operator set contains only the necessary operators, and that (b) the partial-order planning strategy is sufficient. DoLittle focuses on a powerful representation language for applicability conditions, since there will be many general operators. Furthermore, DoLittle also focuses on the use of multiple planning strategies. Therefore, a general operator can constrain the set of plan transformations in its search space.

5.6 DoLittle's domain description language

This section describes the DoLittle domain description language and highlights differences to Prodigy4's domain description language. DoLittle's domain description language is based on Prodigy4's domain description language as described in section **??**. The Prodigy4 domain language has been used to describe many domains, such as the simple blocksworld or the more complex machine shop scheduling domain. The Prodigy domain description language provides a good trade off between making it easy to describe a complex domain and yet still making reasoning using this language tractable. Therefore, DoLittle's domain description language is similar to Prodigy4's representation language, but supports general operators and operator refinements. There are some features in Prodigy4, that are not supported by DoLittle because of the different implementation languages. Prodigy4 may contain arbitrary common lisp code, whereas DoLittle is written in C. DoLittle does its best to parse this code, print a warning, and ignore it. Therefore, the user should be able to read in Prodigy4 domain files, but the planners will behave differently because DoLittle ignores the control rules.

5.7 DoLittle's search control method

Section ?? shows that general operators are able to represent many different planning strategies. However, the representation of planning strategies alone is not enough. A multi-strategy planner must also implement different planning strategies; it must *emulate* a given strategy when provided with its general operator representation. For example, the representations of macros and cases in planning systems are similar, whereas they represent different strategies and affect the search space in different ways. Macros are concatenated only, but cases are adapted.

The search control method uses a static ordering of planning strategies which are sorted according to strength. Stronger strategies are tried before weaker ones. The reason for this ordering is that if the chosen strategy is correct, the solution can be found faster. If the strategy is incorrect, the failure will be detected more easily.

5.7.1 DoLittle's plan structures

Before describing DoLittle's search control method, this section describes the data structures used in DoLittle. The most important data structure in DoLittle as well as other planners based on the plan space search paradigm is the *plan* structure. DoLittle represents a plan as a *tree of fully instantiated operators*. The operators form a tree since refinement of an operator yields a new problem space. The solution to this problem space again is an operator sequence.

The leaves of the operator tree form the current *plan sequence*. The plan sequence is divided into the plan head and the plan tail (a sequence of *pending operators*). Associated with a plan is a *current state*, which is the state resulting from applying the plan head to the initial state.

The sequence of pending operators consists of operators that are not applied yet. The *active operator* is the head of the pending operator sequence.

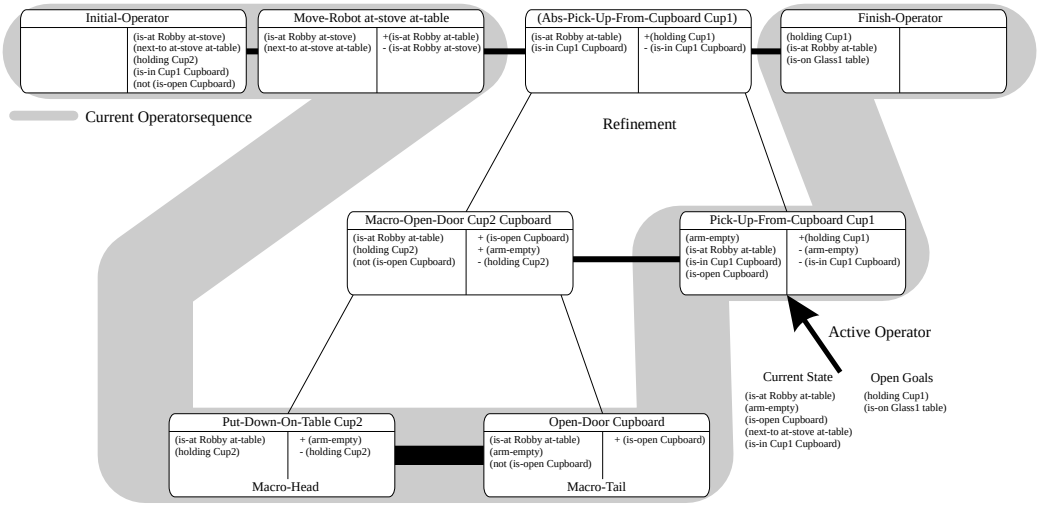
Also associated with the plan is a set of *open goals*. The set of open goals contains the unsatisfied preconditions of the operators in the plan tail. A predicate is an open goal, if and only if it is an element of the preconditions of some pending operator, and the predicate is not established by either the current state or some pending operator that occurs before it. An operator *establishes* a predicate for a target operator if (a) its add-list contains the predicate and (b) there is no operator between it and the target operator that contains the predicate in either its delete or add list. In other words, the operator is the last operator before the target operator that adds the literal and there is no other operator between it and the target that deletes the literal. An effect is a *necessary effect* if it establishes a literal for a following operator. These definitions are commonly used definitions in partial-order planning [**?**].

Each problem space is initialized with a plan that contains dummy first and last operators. The first operator has no preconditions and has effects that establish the initial state. The last operator has the goal expression as its precondition. The top level problem space initializes the operators to establish the initial state and the goal expression. The initial state for a refinement problem space is the state prior to the general operator. The goals of the refinement problem space are the effects of the general operator.

Figure **??** is an example of a partial plan. The plan contains one general operator with a two operator refinement. One operator in the refinement has again a two operator refinement. The current state and the set of open goals are shown beside the active operator.

The figure contains an example of an operator sequence. An operator may mark the start or the end of an operator sequence. If an operator is a MACRO-HEAD operator, then DoLittle will treat the whole operator sequence up to and including the matching MACRO-TAIL as a single operator. For example, in the figure, only the whole subsequence Put-Down-On-Table Cup2, Open-Door Cupboard can be replaced or reordered.

Figure 5.1: The plan data structure



5.7.2 DoLittle's algorithm

Initially, DoLittle creates a subgoal problem space with the original initial state and original goal description and the node list is reset.

Table ?? is a pseudo code description of DoLittle's main loop. First (line ??), retrieves an unexpanded node from the node list. Node retrieval depends on the retrieval method M. Currently, DoLittle supports depth-first, breadth-first, hill climbing, and best-first (with a user specified heuristic function) retrieval methods.

DoLittle signals failure if there are no more open nodes. DoLittle may be solving the original problem space or one that was created to find the refinement of a general operator. The function **Update-Problem-Space** will either signal global failure when working in the top level problem space, or will force the next higher problem space to backtrack.

Then, the current plan is tested to see whether it is complete (line ??). A plan is complete if (a) there are no more open goals and (b) no more pending

Table 5.10: DoLittle's search control algorithm

```
1  proc DoLittle(Nodes)
2     Plan, State, Goals := Select-Node(M, Nodes);
3     if no-more-nodes then
4        Update-Problem-Space(Failure, nil);
5     elsif Complete-Plan(Plan, State, Goals) then
6           if Refine-Next(Plan, State, Goals) then
7              Apply-transformation(REFINE, Plan, State, Goals);
8        else
9              Update-Problem-Space(Success, Plan);
10          fi
11    else
12          foreach t ∈ {APPLY, DEBUG, ADD}
13             Apply-transformation(t, Plan, State, Goals);
14          od
15    fi
16    DoLittle(Nodes);
```

operators. If the plan is complete, then either the next non-primitive operator is refined (line ??), or if the plan is fully refined, the plan is returned as a solution (line ??).

If the plan is not complete, a plan transformation t is applied to yield a new planner state (line ??) and new candidates.

There are three classes of plan transformations: (a) apply the next operator (subsection ??), (b) debug the plan so that the next operator can be applied (subsection ??), or (c) add a new operator and binding to satisfy an open goal (subsection ??).

The following subsections discuss the *REFINE*, *APPLY*, *DEBUG*, and *ADD* plan transformations.

5.7.3 Refinement selection

At the heart of DoLittle is its powerful refinement method. The plan transformation *REFINE* refines the next non-primitive operator in the plan. First, the possible refinements are ordered with respect to how well, they match the applicability conditions. The similarity metric is the same as for adding operators and is described in subsection **??**. Ties are broken using a static ordering: macros, cases, abstract subgoals, serial subgoals, and generic subgoals. As shown in section **??**, a refinement may have a type associated with it, which means that DoLittle will only use the type of the refinement. However, if the type of a refinement is `GENERIC`, it will try all biases in the order specified above. This may involve the removal of a suggested operator sequence to move from a case to an abstract subgoal.

Simon and Kadane show that if the cost of a search and the probability of a search is known, the optimal strategy is to search the spaces in increasing order of probability/cost ratios [**?**]. Assuming that the probabilities are equal, this results in a cheapest first strategy. However, as shown in the analysis in chapter **??**, the situation is different for a subproblem coordinated multi-strategy planner, since there may be more than one level of reduction and the probability of refining a problem further is of critical importance.

As will be shown in chapter **??**, DoLittle maintains the estimates of the success rate (application frequency) and the average refinement cost of a general operator, since they are important features that determine the utility of a general operator. However, it does not maintain the success rate and cost of each refinement and refinement type. Providing sufficiently accurate estimates for those features is too costly.

Therefore, DoLittle does not take the probability of further refinement into consideration. However, the static bias selection method approximates the optimal selection method. The different refinement methods have a large difference in their cost. For example, refining an operator by a macro does require little work. Cases are somewhat more costly, but the work required

is linear in the length of the case. The three type of search spaces are ordered with respect to their sizes. If the difference in costs is large enough, the difference in success rates and probability of further refinement can be ignored. This approximation seems to work well in practice. The reason is that the cost of a refinement in the static order is many times that of the sum of the costs of the previous biases. Therefore, even if instead of estimating the success of a refinement type, the system is given the correct refinement type for a general operator, the sum of the costs of the previous biases is small compared to the cost of the correct bias.

The first two refinement types (macros and cases) yield a sequence of operators that is added as the children of the operator to be refined. Given a macro refinement, the resulting operator sequence is added to the new plan and further adaptation is disallowed by adding a MACRO-HEAD and MACRO-TAIL marker to the head and tail of the sequence respectively. Cases allow substitution of variables and the operators are prepended to the pending operator sequence. This means that DoLittle's method for completing (insertion of an operator in the pending operator set) and debugging (removing and replacement of an operator in the pending operator set) plans may be applied. The latter three refinement types create new subproblem spaces. The initial state is the current state previous to the operator to be refined and the set of goals is the conjunction of the parent operator's effects. The three subgoal refinement types differ in the restrictions that are placed on the search space. An abstract problem space is one, in which none of the literal types of the top level operator (preconditions and effects) may be changed. In a serial subgoal problem space, the instantiated literals in the preconditions of the general operator may not be changed. A subgoal refinement has no constraints on the literals that are affected in the subproblem space.

Table 5.11: DoLittle's *REFINE* plan transformations

```
1  proc Apply-transformation(REFINE, Plan, State, Goals)
2    Op := First-Non-Refined-Operator(Plan);
3    Sorted-Ref := Sort(Refinements(Op), Sim-Metric)
4    foreach Ref := Sorted-Ref
5      Binding := Select-Binding(Ref, State, Goals);
6      if Type(Ref) = Macro ∨ Type(Ref) = Generic then
7        Add-Operator-Subtree(Op, Ref, Binding);
8        Adv-Current-Operator(Op+1);
9      fi
10     if Type(Ref) = Case ∨ Type(Ref) = Generic then
11       Add-Operator-Subtree(Op, Ref, Binding);
12       Adv-Current-Operator(Op-1);
13     fi
14     if Type(Ref) = Abstract-Search-Space
15       ∨ Type(Ref) = Generic then
16       Create-Abstract-Space(State, Goals);
17     fi
18     if Type(Ref)=Serial-Subgoal-Search-Space
19       ∨ Type(Ref) = Generic then
20       Create-Serial-Subgoal-Space(State, Goals);
21     fi
22     if Type(Ref) = Subgoal-Search-Space
23       ∨ Type(Ref) = Generic then
24       Create-Subgoal-Space(State, Goals);
25     fi
```

Table 5.12: DoLittle's *APPLY* plan transformations

```
1  proc Apply-transformation(APPLY, Plan, State, Goals)
2     Active-Seq := Head(Plan-Tail(Plan));
3     Preconds := CalcPreconditions(Op-Sequence(Plan, Active-Seq));
4     if Match(Preconds, Current-State(Plan)) then
5        Plan', State', Goals' := Apply-Operators(Plan, Active-Seq);
6        if State-Loop(Plan') then
7           return(nil);
8     else
9           return(Node(Plan', State', Goals'));
10       fi
11 else
12       return(nil);
13    fi
```

5.7.4 *APPLY* plan transformations

There are two different transformations, that advance the active operator: apply a single operator, or apply a sequence of operators. Application of a single operator is equivalent to the standard means-ends analysis transformation. The head of the plan tail, the active operator, is applied yielding a new current state. For this transformation to be successful, the preconditions of the operator must be satisfied in the current state. DoLittle also supports the application of sequences of operators. A sequence of operators may be delimited by a MACRO-HEAD, MACRO-TAIL pair, in which case DoLittle will not break it up, and only handle it as a single operator. The whole sequence may be applied or replaced. An operator sequence is classified as a MACRO, if it was added to the plan as a refinement of type MACRO.

Table ?? describes the plan transformations that apply operators. In line ??, the active operator sequence is retrieved. If the active operator is

the start of a macro, the tail of the operator sequence is found. Macros may be constructed recursively, so that MACRO-HEAD, MACRO-TAIL combinations may be nested. If the operator is not the start of a macro, the trivial operator sequence consisting of only this operator is created. The preconditions of the operator sequence are computed (line **??**) and tested against the current state (line **??**). If the preconditions match, the operator sequence is applied (line **??**), otherwise an error is signaled.

If the new plan results in a stateloop, it is rejected (line **??**), otherwise it is returned as a new candidate plan. This means that when applying an operator sequence, the plan may indeed pass through a state a second time within the operator sequence. The plan is only invalid if the final state of the operator sequence leads to a state loop.

5.7.5 *DEBUG* plan transformations

DoLittle's plan debugging is based on Chef's plan debug methods (see subsection **??**). Chef uses a more knowledge intensive plan representation language, (e.g., tools, ingredients, continuous variables, object features), so that not all of Chef's debugging methods are applicable in the Strips or DoLittle representations (e.g., ADJUST-BALANCE:UP, ADJUNCT-PLAN:PROTECT). From Chef's 17 debugging methods, 9 have equivalents in DoLittle's representation. The following table compares DoLittle's to Chef's repair strategies. Note that in DoLittle the debugging methods are not limited to a single operator, but include operator sequences. For example, DoLittle can replace a complete macro sequence, instead of just one operator.

Method	CHEF	DOLITTLE
ALTER-PLAN: SIDE-EFFECT	Replace an op. by one that has no unwanted side-effects	Includes op. sequences
ALTER-PLAN: PRECOND	Replace an op. by one that has no missing preconditions	Includes op. sequences
RECOVER	Add an operator to re-establish a deleted side-effect	ADD-OPERATOR in means-ends analysis
REORDER	Reorder running of two steps	Includes op. sequences
ADJ-BAL: UP/DOWN	Increase/Decrease the balance of two parts	no continuous variables
ADJUNCT-PLAN: REMOVE/ PROTECT	Add a concurrent plan step that removes an unwanted side-effect/provides a missing precondition	no concurrent plans
SPLIT-AND-REFORM	Split one operator into two operators	Replace an operator by non-primitive operator
ADJ-TIME: UP/DOWN	Increase/Decrease the duration of a plan step	no temporal reasoning
ALTER-ITEM/TOOL	Replace an object/tool by one that has all desired but none of the undesired features/side-effects	change var. binding (no distinction between tools (ALTER-TOOL) and items (ALTER-ITEM))
PLACE: BEFORE/ AFTER	Move an operator before/after some operator	Includes op. sequences
ALTER-/ REMOVE-FEATURE	Add a step that changes an undesired attribute to a desired one/removes an undesired attribute	Similar to ADD-OPERATOR in means-ends analysis, but no constraint on features
REMOVE	Not implemented in CHEF	Remove an op. sequence with no necessary effects

DoLittle adds a repair strategy, REMOVE, which simply removes an operator with no necessary effects from the plan. Fink and Yang comment on the benefits of justifying a plan in plan adaptation [**?**]. By applying this repair strategy repeatedly, DoLittle will only generate *well-justified* plans. A *well-justified* plan is one where the removal of a single operator does not affect the validity of the plan. A plan is *perfect-justified* if there is no set of operators (possibly disjoint) that can be removed without making the plan invalid. The problem of finding perfect-justified plans is NP-complete [**?**], and thus DoLittle only tests for well-justified plans. Fink and Yiang suggest a greedy, approximate algorithm to compute perfect justified plans. However, well-justified plan seem to be a good enough approximation in DoLittle's domains.

The repair strategies can be easily converted into an algorithm to implement the corresponding *DEBUG* plan transformations. It is important to note that this algorithm ensures that the applicability conditions of all operators are satisfied, even after the transformation. The applicability conditions for general operators require that a general operator's context matches the current context, that the operator's preconditions must match, and that *all* of its open goals are used in the remainder of the plan. Primitive operators, as mentioned previously, have an implicit means-ends bias associated with them. This means that the applicability conditions of a primitive operator are satisfied, if *one* of its effects is necessary.

However, there is a subtle difference between the applicability conditions of the general operator when adding and debugging a plan. In the later case, the applicability conditions are only checked to see whether the operator could be added during some derivation, which is not necessarily the one that was used in this planning episode. This slight difference does not seem to cause any problems in practice, since the plan debugging methods are highly specific and in general more constrained than the applicability conditions themselves.

Table 5.13: DoLittle's *DEBUG* plan transformations

```
1  proc Apply-transformation(DEBUG, Plan, State, Goals)
2    Active-Seq := Head(Plan-Tail(Plan));
3    if Unnecessary-Operator(Active-Seq, Plan) then
4      Remove-Operator(Active-Seq, Plan);
5    fi
6    foreach Binding := Replace-Vars(Active-Seq, Plan)
7      Replace-Binding(Plan, Active-Seq, Binding);
8    od
9    foreach Clobber1 := First-Clobberer(Active-Seq, Plan)
10     Place-After(Plan, Active-Seq, Clobber1);
11     Reorder(Plan, Active-Seq, Clobber1);
12     Place-Before(Plan, Active-Seq, Clobber1);
13   od
14   foreach Op' := Missing-Precondition(Active-Seq, Plan)
15     Replace-Operator(Plan, Active-Seq, Op');
16   od
17   foreach Op', Clobber := Unwanted-Side-Effect(Active-Seq, Plan)
18     Replace-Operator(Plan, Clobber, Op');
19   od
```

The order of the repair strategies is based on a *minimal change heuristic*. DoLittle tries to make the fewest changes necessary to fix the plan: (a) remove an unnecessary operator, (b) change an operators bindings, (c) change the position of two operators in the sequence, and finally (d) replace an operator with a different one.

DoLittle finds first the active operator sequence. This is either the single operator at the head of the plan tail or an operator sequence that starts at the head of the plan tail with MACRO-HEAD and extends to the matching MACRO-TAIL operator.

Then, DoLittle checks whether the active operator sequence does have necessary effects (line ??), i.e., at least one effect of the sequence must be used in the remainder of the plan. If the active operator has no necessary effects, it is removed from the plan. This will generate exactly one new node. Since the active operator sequence is the head of the plan tail, the current state *State* remains unchanged.

The second plan adaptation attempts to find new variable bindings that satisfy all preconditions of the active sequence. If such a binding exists, the variable binding is replaced (line ??). The current state remains unchanged for this plan transformation. However, changing a variable binding might lead to the violation of an operators applicability conditions. For example, assume that an open goal of the planner at this state is (HOLDING GLASS). Now, if DoLittle changes the variable binding from ($V1 = GLASS) to ($V1 = CUP), and (HOLDING CUP) is not an open goal, the applicability conditions of the operator are violated and the new plan is rejected. Otherwise, the algorithm creates the new nodes and adds them to the search space.

In line ??, DoLittle tries to find an operator that (a) is the first operator that clobbers a missing precondition, and (b) has no necessary effects up to the active operator sequence. If such an operator can be found, DoLittle tries three reorder transformations: *Place-After* puts the clobberer after the active sequence (line ??). The active sequence is updated to the next operator. *Reorder* swaps the active sequence and the clobberer (line ??). If the resulting new plan is not valid, it is discarded, otherwise the active operator remains unchanged. *Place-Before* puts the active operator sequence before the clobberer (line ??). DoLittle automatically checks the applicability conditions in the new position.

Next, DoLittle tries to find an operator that has the same effects as the active sequence but is missing some of the un-satisfied preconditions (line ??). This step generates a list of new nodes, since (a) more than one

precondition may not be satisfied, (b) more than one operator may miss this precondition, and (c) more than one binding of the new operator may match the current state.

Lastly, DoLittle checks the plan head to find an operator Clobber, that is the last operator to undo a missing precondition of the active operators (line **??**). If the missing precondition is not used in the remainder of the plan, it is a side-effect and DoLittle tries to replace operator Clobber with one that has the same necessary effects as the clobberer, but does not affect the missing precondition. This step also generates a list of children for similar reasons than the previous step. The new active operator is the operator after Clobber.

5.7.6 *ADD* plan transformations

This subsection describes how DoLittle chooses an operator or operator sequence to add as new head of the plan tail. Means-ends analysis planners, such as Prodigy, generally choose a relevant operator and order operators with respect to how well their preconditions match the current state. That is, an operator that is directly applicable in the current state is preferred over one that has unsatisfied preconditions. This approach works well with small sets of operators. However, with larger sets of operators, it does become impractical.

DoLittle uses an indexing method to retrieve the most similar operator to the current planner state. DoLittle extends Chef's indexing mechanism by considering both initial state and goals in the indexing. DoLittle retrieves all *most specific general operators* that match the current planner state (context, current state, and set of open goals), and collects the corresponding refinements.

A general operator MSO is a *most specific general operator* if and only if (a) it matches the current planner state, and (b) there is no other general operator O2 in the operator memory that matches the current planner state

and MSO is more general than O2. An operator O1 is more general than operator O2 if the planner states identified by the applicability conditions of O2 are a proper subset of those identified by the applicability conditions of O1.

Most case-based planning systems emphasize the indexing mechanism, and have developed sophisticated indexing methods. Unfortunately, these methods require additional domain knowledge, which is not available in a domain independent planner. For example, CHEF's indexing method is based on the knowledge that peas are different from water chestnuts. DOLITTLE's indexing method is based solely on the applicability conditions.

The key issue is to find quickly a plan that requires few adaptations to the new situation. Most case-based planning systems including DOLITTLE use the goal structure to guide the indexing. In a case-based planner such as CHEF features of the goal may also be used to predict interactions that should be avoided.

Table ?? is the algorithm that DOLITTLE uses when adding a new operator. The algorithm handles primitive and general operators differently. DOLITTLE assumes that relevancy is the default applicability condition for primitive operators. An operator is relevant if at least one effect matches an open goal (line ??). The operators binding is then completed and the new plan is created if the added operator does not result in a goal loop.

An operator leads to a goal loop if it either directly or indirectly supports a literal that is one of its unsatisfied preconditions. For example, assume that DOLITTLE is working on the goal (IS-AT ROBBY AT-STOVE). To achieve this goal, the operator MOVE-ROBOT AT-TABLE AT-STOVE is added to the plan. If the robot is not at the table already, this results in a new open goal (IS-AT ROBBY AT-TABLE). One way of achieving this goal is to add the operator MOVE-ROBOT AT-STOVE AT-TABLE. However, this results in a goal loop, since the unsatisfied precondition (IS-AT ROBBY AT-STOVE) (otherwise, it wouldn't have been an open goal in the first place) is part of the operator

sequence to achieve (IS-AT ROBBY AT-STOVE).

For general operators, DOLITTLE first checks the applicability conditions (preconditions, open goals in line ??, context in line ??) and then tests whether there is a more specific general operator that matches the current planner state. The generalization hierarchy is explained in section ?? in detail. DOLITTLE then completes the variable bindings and adds the operator as the new active operator (line ??). Again, the new plan is rejected if it leads to a goal loop.

Lastly, the new nodes are sorted with respect to the similarity metric (line ??). The similarity metric is based on the goodness of the fit between the applicability conditions and the planner state (context, current state, and open goals). The similarity of a general operator to the current planner state is the sum of the ratios of matched to total number of preconditions and effects. The count is reversed for negated preconditions and del-effects, for example, a delete effect is counted if it does *not* occur in the open goals, and vice versa. Ties between instantiated operators are broken arbitrarily.

Table ?? is an example of DOLITTLE's indexing mechanism. There are seven operators in the operator set, three primitive and four non-primitive ones. In this example, all operators are different instantiations or generalizations of the operator MOVE-ROBOT. The current state and the set of open goals is shown at the top. The applicability of primitive operators depends on an operator's effects, whereas for general operators, it depends on the set of open goals. Therefore, the table lists the effects for primitive operators and the set of open goals for general operators.

PRIMITIVE-OP-1 and PRIMITIVE-OP-2 are examples of relevant operators. PRIMITIVE-OP-3 is rejected since none of its effects matches an open goal.

DOLITTLE-OP-1 is rejected because its preconditions do not match the current state. Compare this to PRIMITIVE-OP-2, which is applicable, although its preconditions do not match. DOLITTLE-OP-2 is not applicable

Table 5.14: DoLittle's *ADD* plan transformations

```
1  proc Apply-transformation(ADD, Plan, State, Goals)
2    Nodes := nil
3    foreach Op ∈ Primitive-Operators
4      if Relevant(Op, Goals) then
5        foreach Binding := Possible-Bindings(Op, State, Goals)
6          Plan', , Goals' := Add-Operator(Op, Binding, Plan);
7          if (not(Goal-Loop(Plan'))) then
8            Nodes := Nodes + Node(Plan', State, Goals');
9          fi
10       od
11     fi
12   od
14   foreach Op ∈ General-Operators
15     if   Matches(Preconds(Op), State) ∧ Matches(Open-Goals(Op), Goals)
16       ∧ Matches(Context(Op), CurrentContext)
17       ∧ Most-specific(Op, State, Goals, Operator-Set) then
18       foreach Binding := Possible-Bindings(Op, State, Goals)
19         Plan', , Goals' := Add-Operator(Op, Binding, Plan);
20         if (not(Goal-Loop(Plan'))) then
21           Nodes := Nodes + Node(Plan', State, Goals');
22         fi
23       od
24     fi
25   od
26   Nodes := Sort-Nodes(Nodes, Sim-Measure);
27   return(Nodes);
```

because the literal (NOT (IS-AT ROBBY AT-STOVE)) is not an open goal. This means that this fact is not used in the remainder of the plan and is a side-effect. The main effect is that the robot is now at the table. In contrast to DOLITTLE-OP-2, operator PRIMITIVE-OP-1 is applicable. Operator DOLITTLE-OP-3 is more specific than DOLITTLE-OP-4, since the preconditions and open goals of DOLITTLE-OP-4 are a subset of those of DOLITTLE-OP-3. Therefore, operator DOLITTLE-OP-4 is rejected since it is not the most specific matching operator. Operator DOLITTLE-OP-5 has more general open goals than operator DOLITTLE-OP-3, but its preconditions are different. Thus it is applicable.

Therefore, only operators PRIMITIVE-OP-1, PRIMITIVE-OP-2, DOLITTLE-OP-3, and DOLITTLE-OP-5 are applicable. The similarity measure is the sum of the ratios of matching preconditions and effects. The operators are therefore selected in this order: DOLITTLE-OP-3, DOLITTLE-OP-5, PRIMITIVE-OP-1, and PRIMITIVE-OP-2.

5.8 Discussion

This section discusses the design of DOLITTLE, a multi-strategy planner. Since DOLITTLE attempts to emulate different planning system, it compares DOLITTLE's implementation of a given problem solving strategy to other implementations: means-ends analysis, a macro-based planner, a case-based planner, and an abstraction-based planner. This comparison shows that DOLITTLE is able to mimic the essence of those planning strategies and that it can make use of the same planning biases.

5.8.1 DoLittle as a means-ends analysis planner

This subsection compares DOLITTLE as a means-ends analysis planner to other means-ends analysis planners, in particular PRODIGY4. Means-ends analysis is the underlying planning strategy of DOLITTLE. This comparison

Table 5.15: Example of DoLittle's *ADD* plan transformations

Current State	Open Goals	
(is-at Robby at-stove)	(is-at Robby at-table)	
(holding Cup1)	(arm-empty)	
(is-open cupboard)		
Primitive-Op-1		$1/3 + 1/2$
Preconditions	Effects/Open goals	
(is-at Robby at-stove)	(is-at Robby at-table)	
	(not (is-at Robby at-stove))	
Primitive-Op-2		$0/3 + 1/2$
(is-at Robby at-sink)	(is-at Robby at-table)	
	(not (is-at Robby at-sink))	
Primitive-Op-3		not relevant
(is-at Robby at-stove)	(is-at Robby at-sink)	
	(not (is-at Robby at-stove))	
DoLittle-Op-1		no match for preconditions
(is-at Robby at-sink)	(is-at Robby at-table)	
DoLittle-Op-2		no match for effects
(is-at Robby at-stove)	(is-at Robby at-table)	
	(not (is-at Robby at-stove))	
DoLittle-Op-3		$2/3 + 2/2$
(is-at Robby at-stove)	(is-at Robby at-table)	
(holding Cup1)	(arm-empty)	
DoLittle-Op-4		too general (DoLittle-Op-3)
(is-at Robby at-stove)	(is-at Robby at-table)	
	(arm-empty)	
DoLittle-Op-5		$2/3 + 1/2$
(is-at Robby at-stove)	(is-at Robby at-table)	
(is-open cupboard)		

ignores DoLittle's enhancements so that there are only the primitive operators in the operator set and that none of the plan debugging methods are used. There are four main differences between Prodigy and DoLittle: (a) control rules, (b) matcher, (c) backtracking, and (d) operator selection.

The first three differences simplified the implementation of DoLittle and could be added to DoLittle. Only the different operator selection scheme is due to the fact that DoLittle is a multi-strategy planner.

First, Prodigy allows the user to define meta-predicates and control rules for a domain in common lisp. The implementation of control rules in DoLittle would be more difficult because it was implemented in C. Control rules are not part of the standard means-ends analysis planning paradigm, but are Prodigy's method for encoding search control knowledge.

Secondly, Prodigy4 also contains a more complex matcher [**?**], based on a RETE network. The matching cost depends on the number of free variables in the operator. DoLittle's matcher was implemented as a simple match tree. The worst case complexity of both algorithms is exponential in the number of free variables. However, the cost per match attempt is smaller for a RETE network as opposed to a simple match tree. Therefore, Prodigy's matcher will outperform DoLittle's matcher if there is a large set of possible instantiation for a variable. The domains in this thesis use a type hierarchy to restrict the possible instantiations of variables, and thus there is generally only a small set of possible instantiations. I profiled DoLittle's code to determine the cost of the matching algorithm. This experiment showed that the matcher accounted for less than five percent of the running time. Therefore, the current implementation of the matcher seems to be reasonable effective.

Thirdly, Prodigy4 extends means-ends analysis with two search reduction techniques: dependency directed backtracking and look-ahead. Dependency directed backtracking removes binding nodes that lead to an unachievable subgoal. Look-ahead removes nodes that necessarily lead to a goal loop,

without computing a complete set of bindings. It is possible to use the same techniques in DoLittle, because they affect the backtracking mechanism only. Dependency-directed backtracking backtracks to an operator selection instead of a binding selection, if no set of bindings can possibly achieve the goal. Look-ahead saves work by predicting that a node can not lead to a solution without computing a complete set of bindings. However, the amount of search reduction is limited if combined with abstraction hierarchies [**?**]. Therefore, the benefit of a smarter backtracking algorithm in a multi-strategy planner is unclear, especially since DoLittle includes plan debugging methods to reduce the amount of backtracking. Plan debugging is similar to dynamic backtracking, a backtracking method that is trying to maintain as much of the previous work as possible when undoing an incorrect choice by moving a shallow node in the search tree to a lower level [**?**]. DoLittle, therefore, does not use look-ahead and only uses a simple backtracking method (chronological backtracking).

Lastly, the main difference between DoLittle and Prodigy4 is the operator selection mechanism. Prodigy4 selects an operator by picking a goal from the open goal list, finding an operator that is relevant to the goal, selecting a binding for the operator and adding the operator to the plan. DoLittle's method is case based, operators are retrieved based on a domain independent similarity metric **Sim-Metric**. The similarity metric is based on the number of matching effects and preconditions.

However, DoLittle still guarantees that a selected operator is relevant to at least one goal in the open goal list. Therefore, if no primitive operator is a generalization of another primitive operator, the different operator selection criteria lead to a different order of expansion, but the candidates are identical. Using breadth-first search, the searches are synchronized after each level.

To estimate the effect of the differences between DoLittle and Prodigy, the following experiments compare DoLittle running in Prodigy mode (no adaptations, no general operators) to Prodigy with

and without dependency directed backtracking.

The three planners (PRODIGY-DL, PRODIGY-Dep, and PRODIGY) were run on a randomly selected set of 250 problems in the blocksworld. The algorithm for generating random problems is described in section **??**. Figure **??** shows the number of expanded nodes for all three planners. All planners use depth-first search, since the differences in performance are greater for depth-first search than for breadth-first. In breadth first search, the different operator selection methods do not play such an important role, since the search will only differ in the last level of the search tree. There was a node limit of $15,000$ nodes and a time limit of 600 CPU seconds.

Figure **??** shows that PRODIGY-DL's performance is similar to PRODIGY's with chronological backtracking (PRODIGY) and with dependency directed backtracking (PRODIGY-Dep) when measured by the number of expanded nodes. Using a paired t-test, it can be shown that indeed the difference between the number of nodes expanded by DOLITTLE and PRODIGY-Dep is not statistically significant with an α level of 0.05.

Figure **??** compares the running time of the three planners. It shows that DOLITTLE runs roughly twice as fast as PRODIGY. The speed up is caused by DOLITTLE's implementation in C instead of Common Lisp. The figure also shows that in the blocksworld, the extra cost of dependency directed backtracking outweighs the reduction in expanded nodes.

These tests also show that PRODIGY-DL is indeed an efficient planning system, comparable to another state of the art planner PRODIGY. This fact is important in the evaluation of DOLITTLE, since it shows an improvement of multi-strategy planning over single strategy planners and PRODIGY-DL. The baseline in this comparison is important because a rule of thumb is that the more inefficient a system is to begin with, the easier it is to improve its performance.

I also tried to compare the performance of PRODIGY-DL and PRODIGY in the kitchen domain. Unfortunately, PRODIGY-DL and PRODIGY are not

Figure 5.2: Comparison of Prodigy and Prodigy-DL
Cumulative nodes in the blocksworld

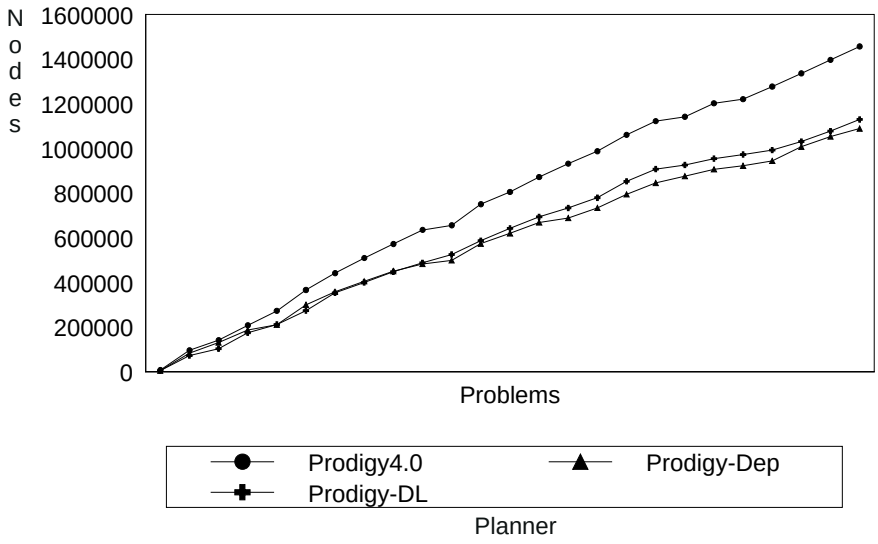
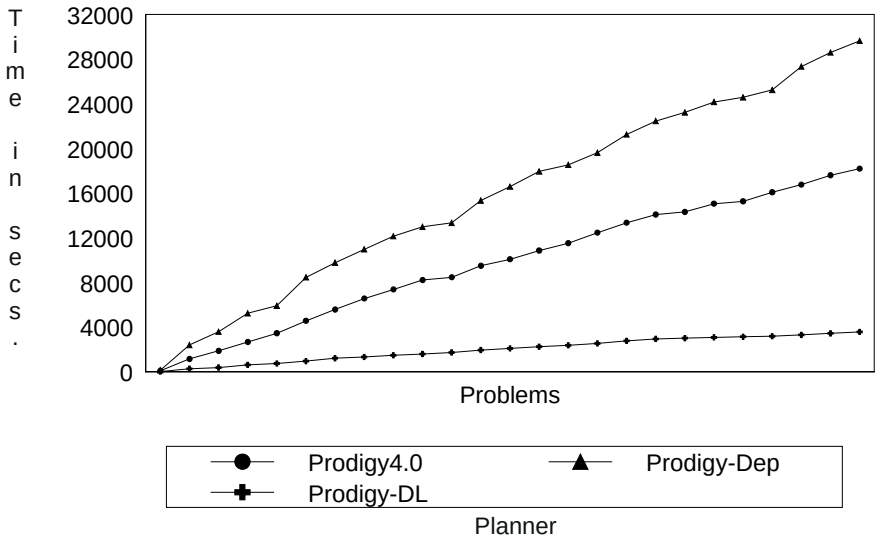


Figure 5.3: Comparison of Prodigy and Prodigy-DL
Cumulative times in the blocksworld



able to solve anything but the simplest problems in the kitchen domain.

5.8.2 DoLittle as a case-based planner

This subsection compares DoLittle and Chef, a case-based planner. Plan retrieval in Chef and DoLittle differ, because Chef's indexing mechanism is domain-dependent, whereas DoLittle uses a domain-independent scheme. Chef retrieves a plan only at the top-level of the search tree, whereas DoLittle allows retrieval of a case any time an operator is added to the plan.

The plan debugging methods of Chef and DoLittle are similar. The main difference is that Chef has a more expressive representation language, that for example includes object features and concurrent plans.

Chef associates a *thematic organizations packet* TOP with a plan failure. DoLittle associates repair strategies directly with the failure. The main function of TOPs in Chef is to predict failure. DoLittle supports the prediction of failure using a general operator (see subsection **??**). In contrast to Chef this prediction is instantiated (e.g., (not (is-soggy broccoli))) as opposed to meta-level comments (e.g., (not (not-prevent-goal soggy))).

The order of plan repairs in DoLittle is based on the least-change heuristic: REMOVE-OPERATOR, CHANGE-BINDING, PLACE-BEFORE, PLACE-AFTER, REORDER, REPLACE-MISSING-PRECONDITION, REPLACE-UNWANTED-SIDE-EFFECT. DoLittle prefers plan debugging methods that make fewer adaptations to the plan. Chef's ordering of plan adaptations is based on domain-dependent knowledge, such as "Prefer adding a preparation step to adding a cooking step." Some of those ordering heuristics are domain-independent and similar to DoLittle's least change strategy, such as "It is better to add one step than to add many steps."

5.8.3 DoLittle as an abstraction-based planner

This subsection shows that DoLittle can represent relaxed abstractions (e.g., AbStrips). The following table is a description of the towers of Hanoi domain, that emulates AbStrips's abstraction hierarchy.

Table ?? is the operator set that DoLittle uses in the towers of Hanoi domain with three disks. In addition to the three primitive operators Move-Small, Move-Medium, Move-Large, it contains two sets of operators that represent the abstraction hierarchy. The first set enables DoLittle to move the medium disk independent on where the small disk is. The second set enables DoLittle to move the large disk and ignoring the two smaller disks.

Table 5.16: Abstraction in DoLittle: Towers of Hanoi

Move-Small $PegX $PegY	
Preconditions	Effects
(is-on Small $PegX)	(is-on Small $PegY)
	(not (is-on Small $PegX))
Move-Medium $PegX $PegY	
Preconditions	Effects
(is-on Medium $PegX)	(is-on Medium $PegY)
(not (is-on Small $PegX))	(not (is-on Medium $PegX))
(not (is-on Small $PegY))	
Move-Large $PegX $PegY	
Preconditions	Effects
(is-on Large $PegX)	(is-on Large $PegY)
(not (is-on Small $PegX))	(not (is-on Large $PegX))
(not (is-on Small $PegY))	
(not (is-on Medium $PegY))	
(not (is-on Medium $PegY))	
Abs-Move-Large-1 $PegX $PegY	
Preconditions	Open Goals
(is-on Large $PegX)	(is-on Large $PegY)
Abs-Move-Large-2 $PegX $PegY	
Preconditions	Open Goals
(is-on Large $PegX)	(not (is-on Large $PegX))
Abs-Move-Medium-1 $PegX $PegY	
Preconditions	Open Goals
(is-on Medium $PegX)	(is-on Medium $PegY)
Abs-Move-Medium-2 $PegX $PegY	
Preconditions	Open Goals
(is-on Medium $PegX)	(not (is-on Medium $PegX))

Chapter 6

Learning Planning Knowledge

> We learn through experience and experiencing, and no one teaches anyone anything. This is as true for the infant moving from kicking to crawling to walking as it is for the scientist with his equations. If the environment permits it, anyone can learn whatever he chooses to learn; and if the individual permits it, the environment will teach him everything it has to teach.
>
> **Viola Spolin** (b. 1911), U.S. theatrical director, producer. *Improvisation for the Theater*, ch. 1 (1963).

The focus of this thesis is the design of a multi-strategy planner. This, however, leaves the question: "How can suitable general operators be found?" This chapter describes some example learners for acquiring general operators inductively. These learners are simple versions of common speed up learning methods, included here to facilitate an empirical evaluation. By no means are these the only ways to generate new operators. DoLittle's design does not include the planning bias learners, but treats learning methods as "plug ins," and other methods (e.g., explanation based learning, discourse analysis, or user supplied functions) are also possible. The inductive acquisition of planning knowledge is a requirement that does not stem from the multi–strategy planning paradigm per se, but from the instructable system paradigm, which

is the intended application for DoLittle.

The required inputs and outputs of the learning modules depend on DoLittle's requirements. DoLittle's planning activity focuses on general operators for macros, cases, and abstractions. DoLittle also focuses on one particular type of problem solving event, a successful (partial) solution, to learn new operators. There are many other problem solving events that allow opportunities for learning, for example learning from failure [?], subgoal interaction [?], or impasses [?]. DoLittle's planning bias learners focus on learning solely from success to emphasize a main point of this thesis, the separation of a planning biases' implementation (planning strategy) and its underlying assumptions. That is, there may be more than one way of exploiting assumptions about the domain, and there may be more than one set of assumptions that may lead to the same planning strategy. For example, two different learning systems may generate a new macro, but may be based on completely different motivations. Planning system A creates a macro by extracting commonly used sequences from a successful plan. Planner B creates macros from complete previous solutions to generalize subtasks. Planner B's planning bias is that the problems are presented in increasing order of difficulty and that solutions to earlier problems are important subtasks of later solutions.

Section ?? describes Gratch and DeJong's framework for adaptive planning. This framework interprets adaptive planning as search through the space of planner transformations. This framework identifies two main problems in adaptive planning: (a) the size of the transformation space, and (b) the cost of evaluating the utility of a transformation. As can be seen in section ??, there is a vast set of possible planner transformations in DoLittle. Section ?? describes methods for restricting the search through possible planner transformations. Methods for reducing the cost of estimating the utility of a planner transformation in DoLittle's representation are described in section ??. Section ?? introduces the sample DoLittle case-based learner.

The macro-operator learner is shown in section ??. Section ?? describes DoLittle's abstraction-based learner. Section ?? summarizes this chapter.

6.1 Learning of general operators

The complexity of planning has led to significant interest in speed up learning [?, ?, ?, ?]. The idea is to create an efficient domain-dependent planner by training a domain-independent one. Ideally, we would like to create a domain independent planner that, given only a domain description, starts solving problems in the domain. The system then gathers domain specific knowledge during the planning process and exploits this domain specific knowledge improving its planning performance. Thus, after running the planner on enough problems, it is transformed into an efficient domain-dependent planner.

Although this idea is appealing, learning to improve planning performance is not a simple problem. Therefore, many learning planners make (justifiably or unjustifiably) simplifying assumptions to reduce the cost of learning.

This section describes a framework for adaptive planning suggested by Gratch and DeJong [?]. This framework identifies three problems that an adaptive planning system must overcome. Sections ?? and ?? describe how DoLittle's planning bias learners solve the two main problems.

Similarly to the plan space search paradigm described in section ??, this framework views *learning to plan* as a search through a set of possible *planner transformations*. These transformations include, among others, addition of a macro-operator, creation of a chunk, or generation of a control rule.

For example, through generation of a macro, a macro-learner transforms the original planner into a new system which contains the previous operators and the new macro. The task then is to find a sequence of transformations that adapts the original planner into a more efficient one. The efficiency of a planner may be defined in terms of the average solution time, the number of solved problems, or the solution quality. In the instructable system paradigm,

the user will have to wait for the planner to create a plan to do some task. Therefore, in this thesis, the efficiency of the planner is determined by the expected cost of finding a solution over a set of problems (see section **??**). The utility of a planner is the negative of the expected cost and may be formalized as follows:

$$\text{UTILITY}(\text{planner}) = -\sum_{p \in \text{Problems}} \text{Cost}(p, \text{planner}) \times \text{prob}(p) \qquad (6.1)$$

The utility or efficiency of a planner is a function of the cost of a problem weighted by the probability of this problem appearing. This formalization of efficiency is too expensive to compute in practice. There are three problems that make exhaustively searching the space of planner transformations infeasible:

1. the space of transformations is too large. For example, the set of possible macro-operators that may be added to a planner is extremely large.

2. estimating the efficiency of a transformation (or even the incremental utility) of a transformation is too expensive.

3. the necessary observations are too expensive to be extracted from the environment.

To alleviate these problems, learning planning systems generally make simplifying assumptions (learning bias) when evaluating the efficiency of sequences of transformations. For example, many planners use *operationality conditions* to approximate efficiency [**?**, **?**, **?**].

Section **??** shows the approach of DoLittle's learners to the first problem. Section **??** describes the solution to the second problem. The third problem is not that critical in DoLittle's representation, since the nec-

essary information for the learners can easily be extracted from the search tree.

6.2 DoLittle's planner transformations

This section discusses the space of possible planner transformations in DoLittle. In the framework described in section ??, a planner can be characterized by its plan language $\mathcal{L}_{\mathcal{P}}$, its set of plan transformations T, and its search method M. Theoretically, a planner can be adapted by changing any of those components. However, adapting the plan language or the search method is difficult. Therefore, most adaptive planning systems, including DoLittle, control the planner through the operator set. In DoLittle, the set of plan transformations is controlled by the set of general operators.

DoLittle provides a much larger set of planner transformations than other macro learners, for example MacLearn. DoLittle allows creation/deletion of operator sequences, and allows organization of those sequences using a generalization hierarchy and a set of refinements. Note that the space of planner transformations is not searched exhaustively because it is too big. As will be shown in section ??, DoLittle uses an event-driven approach to learning. The learners identify possibilities for improvement during the search for a problem solution.

DoLittle may change any part of a general operator as long as the resulting general operator still satisfies the requirements in definition ??. The following list describes the set of possible planner transformations in DoLittle's representation:

1. Create a new general operator
2. Delete a general operator
3. Generalize applicability conditions (context, preconditions, open goals)

4. Specialize applicability conditions
5. Add a refinement to a general operator
6. Remove a refinement from a general operator
7. Change the type of a refinement
8. Add a constraint to a refinement
9. Remove a constraint from a refinement
10. Merge two general operators
11. Split a general operator into two general operators
12. Replace an operator reference by a set of refinements
13. Replace a set of refinements by an operator reference

General operators include meta knowledge, the applicability conditions (context, preconditions, open goals) of a general operator. DoLittle may adapt the search by generalizing or specializing these applicability conditions. For example, if a general operator was successful in finding a solution to the problem of making coffee with sugar, but not coffee, a learner may change the applicability conditions to only use this operator if a plan requires coffee and sugar. One constraint on the specialization of applicability conditions is that they have to be at least as general as the common preconditions and effects of the refinements.

DoLittle can also add or remove a refinement of a general operator. The purpose of adding a refinement is to reduce the cost of finding a refinement for a general operator. For example, assume that there is a general operator that has two refinements, one with the additional precondition (is-at Robby at-stove), and one with the robot being at the table (is-at Robby at-table). If DoLittle is refining this general operator and the robot is at

the fridge, DoLittle retrieves one refinement (e.g., the first refinement) and creates a new subgoal (is-at Robby at-stove). This search may be prevented if a third refinement with the additional precondition (is-at Robby at-fridge) is added.

The preconditions of a refinement may have additional constraints that can be added or removed from a refinement. However, the preconditions of a refinement must always guarantee the applicability of the associated operator sequence. In other words, the preconditions of the refinement must be at least as strong as the preconditions of the associated operator sequence.

Merging two general operators creates a new general operator (a) whose context, preconditions, open goals, and effects are the intersection of the two original ones, and (b) whose set of refinements is the union of the refinements of the original operators. If there are two general operators that have similar applicability conditions and effects, but differ in one precondition literal, DoLittle can create an abstraction of those two operators. The inverse planner transformation of splitting a general operator is also possible. For example, this may be useful, if a proposed abstraction turns out to be not useful, because it is too general.

Replacing an operator reference by a set of refinements creates a new set of refinements for a general operator, that has all references to a general operator replaced by a subset of its refinements. Since general operators may be interpreted as AND/OR trees, this is equivalent to promoting a disjunction. For example, a general operator Op1 has one refinement that contains a reference to another general operator Op2 with three refinements. However, in the context of the first general operator Op1, only two of the refinements (R1 and R2) of Op2 are useful. In this case, a learner may replace the reference to the second general operator Op2 with the two useful refinements. The original refinement of Op1 is replaced by two new refinements that are created by replacing the reference to Op2 with R1 and R2 respectively. Inversely, a learner can also replace an operator sequence of a refinement by

an operator reference.

DoLittle's special open goal literal (INVALID) allows a learner to replace all references to a general operator by its set of refinements.

6.3 Simplifications of the transformation space

The first problem in adaptive planning is the size of the transformation space. For a set of n transformations, there are 2^n possible sets of transformations. Therefore, a common simplification made by learning planners is to assume that the utility of a transformation is independent of the utility of other transformations. Then a space of n planner transformations can be searched in n steps, rather than 2^n. Even if the set of transformations is not independent, the transformations may be grouped into equivalence classes. Rather than all transformations in a class only one candidate from each class must be tested.

Generation pruning reduces the space of possible planner transformations by limiting the set of transformations that are being considered. Even if transformations are independent, the space of possible transformations may be very large. For example, the space of macro-operators consists of all legal operator sequences. Some systems use an event driven approach to macro-operator creation, that is only macro-operators that occurred during problem solving are being considered. This problem is worse in DoLittle's representation than in the macro learner representations since, as shown in section **??**, its set of planner transformations contains many more planner transformations than simple macros. Thus DoLittle's transformation space is extremely large and must be restricted.

Composition pruning reduces the space of transformations by only checking one sequence of n transformations instead of $n!$ possible sequences. The assumption is that the order in which the planner transformations are made

is not important. For example, PRODIGY/EBL's set of planner transformations consists of the set of control rules that may be added to the planner. But, PRODIGY/EBL uses the learned control rules in the order in which they were acquired, so that a different sequence of examples may lead to a different planning system.

To reduce the planning cost, DOLITTLE's planning bias learners use event driven learning and assume that the learned operators are independent and insensitive to the order in which they were acquired. This greatly reduces the size of the transformation space, which is why this approach is used by many other planning systems [**?**, **?**]. The independence assumption is justifiable in DOLITTLE's representation, since it provides a more powerful language for applicability conditions than simple macro learning systems. This means that the possibility of interference between different general operators is reduced. The insensitivity assumption is justified because the selection mechanism for selecting general operators and refinements is based on a domain-independent similarity metric and is thus mostly insensitive to the order in which the operators/refinements were acquired. The order of planner transformations only affects the selection of operators with identical similarity measures.

6.4 Simplifications in evaluation utility

The second problem in learning to plan is to determine accurately the utility of a suggested planner transformation. That is, after application of a planner transformation, the new performance of the planner must be estimated, to verify that the applied planner transformation indeed increases the utility of the planner. The efficiency depends on the (possibly unknown) distribution of problems in the domain. The brute force approach of testing the planner with and without transformation on all problems in the domain is clearly too expensive.

Some systems (ALPINE,STATIC) overcome this problem by focusing

on syntactic features of the domain and do not learn from examples. PRODIGY/EBL takes a sample of some of important features that determine the utility and averages the utility over the sample. However, there is no concept of confidence and so those systems can not determine how many examples are necessary to have an accurate enough approximation of the utility.

PALO [?] and COMPOSER [?] use a formal approach to operator learning. In particular, they guarantee that the final planner will indeed be a locally optimal planner with respect to the set of planner transformations. However, these systems are not applicable in our learning paradigm, since they require many training problems, to compute the necessary statistics for estimating the utility of a transformation. An instructable system is trained on a small set of examples.

DOLITTLE's sample learners use an estimate of the utility similar to the estimate used by PRODIGY/EBL. The motivation is to find a set of sufficient conditions, that is a set of assumptions that allow the prediction of a positive utility of a planner transformation. If the assumptions are met, it is guaranteed that the planner's performance will increase after application of a planner transformation. This does not mean that DOLITTLE can not perform poorly, if the assumptions are *not* met. There is a separate set of sufficient conditions for each planning bias learner.

Using the definition of utility in equation **??**, *incremental utility* (δ_U) is defined as the difference in efficiency before and after application of a transformation. If the probability distribution is constant for the two planners, the incremental utility of a planner P with respect to a transformed planner P' is given by

$$\delta_U = \sum_{p \in \text{Problems}} (\text{Cost}(p, \text{P}) - \text{Cost}(p, \text{P'})) \times \text{prob}(p)$$

The incremental utility is the sum of the differences in cost between the

new and the old planning system on a problem weighted by the probability that this problem occurs. It is assumed that the expensive part of the planner is the search. So, the analysis ignores effects such as increases in loading times of a domain because of the additional operators.

The following subsections discusses how DoLittle's planning bias learners derive an estimate for the incremental utility of different planner transformations. The learners estimate the incremental utility of a planner transformation by comparing the actual cost due to the planner transformation to an estimate of its savings.

6.4.1 Utility estimate of adding operators

Consider the case, where a learner has created a new general operator. At each node in the search space, DoLittle has to check whether the applicability conditions (context, preconditions, open goals) of the new operator are satisfied. To verify that the context field is satisfied can be done quickly, because DoLittle simply checks whether the planner is currently refining at least one operator mentioned in the context field. Testing the preconditions and open goals requires the computation of a binding for free variables and checking to see whether the instantiated operator is the most specific operator to match the current planner state (current state, open goals).

Computing a binding is equivalent to the match cost of the new general operator, which is exponential in the number of free variables. To check whether an operator is the most specific operator to match the current planner state, DoLittle must compare the applicability conditions of all operators that match the current planner state. A matching operator is rejected if its preconditions and open goals are a subset of some other matching operator's preconditions and open goals. The worst case complexity of this comparison is the maximum number of preconditions and open goals of an operator multiplied by the number of matching operators.

If the general operator does not match the current planner state, it has

no further effect on the search. On the other hand, if the operator matches, DoLittle will have to refine it. To refine a general operator, DoLittle must compute a binding for all refinements of the operator, and sort them with respect to the similarity metric. After selection of a refinement, the general operator is replaced by the refinement in the partial plan. There is a refinement cost associated with each type of refinement. For example, replacing a general operator with a macro has a negligible cost, whereas replacing an operator with a subproblem search space means that DoLittle must search for a solution of a possibly non-trivial subgoal.

In this model, the incremental utility of a new general operator is given by equation **??**. The utility of a general operator is determined by the difference between its benefit and its cost. The benefit is determined by the search cost that is saved. The cost is determined by the cost of finding a refinement and the cost of matching the general operator. This equation is similar to Minton's utility evaluation, but includes a term to describe the cost of refining and one for the match frequency. The refinement cost corresponds to the cost of applying a given planning strategy. Since Prodigy/EBL only adds control rules, there is no notion of refinement cost. The match frequency is the ratio of number of times the operator was matched (preconditions and open goals) to the total number of nodes. DoLittle may reject an operator quickly, if it has a highly specialized context. Prodigy/EBL computes a binding for all control rules, so the match frequency for all search control rules in Prodigy/EBL is one. Also DoLittle is more conservative than Prodigy/EBL, since it uses the maximum rather than the average match cost. In other words, DoLittle is more likely than Prodigy/EBL to reject a planner transformation. The intuition is that DoLittle can more easily find a good planner transformation because it has a larger set of planning strategies than Prodigy/EBL.

$$\begin{aligned} \delta_U &= \text{AvrSearchCost} \times \text{ApplicFreq} \\ &\quad - \text{AvrRefinementCost} \times \text{ApplicFreq} \\ &\quad - \text{MaxMatchCost} \times \text{MatchFreq} \end{aligned} \tag{6.2}$$

DoLittle's planning bias learners use the following approximations in estimating the incremental utility of adding a general operator:

- **AvrSearchCost** is the average cost of finding a solution to the planner state identified by the applicability conditions of the new general operator without the new operator.
- **AvrRefinementCost** is the average cost of finding a refinement for the general operator.
- **ApplicFreq** is the ratio of successful attempts to match the applicability conditions of the new general operator to the total number of nodes.
- **MaxMatchCost** is the worst case cost of searching for a binding of the free variables.
- **MatchFreq** is the ratio of the number of times the operator was matched to the total number of nodes

Prodigy/EBL measures the run time of the planner to determine the costs/savings of the different factors. DoLittle's learners use the number of nodes, since as a parallel system, it may execute on processors with different speeds. On a single processor an experiment in the blocksworld showed a strong correlation between run time and number of expanded nodes (90 percent), which means that the number of nodes can be used as an approximation of run time.

The most difficult factor in determining the incremental utility of a general operator is to determine its average search cost. To estimate accurately

the average search cost, a planner would have to solve a problem with and without the general operator and compute the difference in running times or number of expanded nodes. This is a clearly an expensive proposition. However, for the example that was used to generate the new operator the search cost is known, since it was solved without the suggested operator and the search cost of the new general operator can be computed from the example. To illustrate, when learning a case operator, the search cost is the total cost of solving the problem. Therefore, DoLittle's learners use the search cost of the first example as the average search cost estimate of the rule.

The average refinement cost, application frequency, and match frequency in equation ?? can be estimated during successive problem solving episodes.

The estimate of the refinement cost of a single refinement is the cost of finding a binding for any free variables and the cost of refining the operator. This cost depends on the type of refinement. Macro operators are refined with a cost of one node. Case refinements cost one node for each operator in the sequence. The estimates of costs for serial subgoals, abstract subgoals, and subgoal refinements are the costs of the search of the resulting subproblem space. The estimated cost of a generic refinement is the sum of the refinement costs for refinements of a stronger type (macro, case, abstract subgoal, serial subgoal, subgoal) than the intended one. The estimate of the refinement cost of a set of refinements is the cost of selecting a refinement plus the maximum of the costs of the individual refinements.

Equation ?? also provides two quick methods for ruling out expensive general operators. First, any new general operator whose refinement cost is greater than its average search cost will not lead to an improvement and is immediately discarded. Secondly, since the application frequency is always smaller than one, comparing the difference of average search cost and refinement cost to the maximum match cost allows DoLittle's planning bias learners to rule out new general operators, that will never lead to an improvement (even if the match frequency were one). This second check is

also used by PRODIGY/EBL. However, in contrast to DOLITTLE's representation, each rule in PRODIGY/EBL has the same fixed refinement cost and its match frequency is one.

6.4.2 Utility estimate of generalizing applicability conditions

Generalizing or specializing the applicability conditions of a general operator will affect the application frequency, the match cost, and the match frequency of a general operator. For example, replacing an object by a variable in the applicability conditions can greatly increase the match cost.

Since DOLITTLE's learners compute the new match cost and maintain statistics of the application and match frequency of a general operator, the impact of those changes can be checked.

However, generalization and specialization may also have an effect on the average search cost of an operator and the average refinement cost. This is especially true if the applicability conditions are too general and the general operator does not lead to a solution. In this case, the search is led astray and the cost of trying to find a refinement for the general operator is in vain.

Since the average refinement cost, i.e., the average cost of refining a general operator is checked, DOLITTLE's learners will detect the problem of over-generalization by its high refinement cost and therefore poor utility. In this case, the general operator can either be deleted or its applicability conditions specialized.

6.4.3 Utility estimate of changing a refinement

Adding, removing, changing the type, adding a constraint, or removing a constraint of a refinement only changes the refinement cost and possibly the average search cost of a general operator. It does not affect the match cost and frequency of the general operator or its application frequency. Any

change in the refinement cost is monitored in the following problem solving episodes and DoLittle's learners can thus verify that a proposed change indeed did increase performance.

6.4.4 Utility estimate of merging general operators

Merging of two general operators, requires computation of a new estimate for the average search cost of the new operator. DoLittle's planning bias learners use the maximum of the individual search costs as the new search cost estimate for the merged operator.

6.4.5 Utility estimate of replacing operator references

Replacing an operator by an operator reference is equivalent to removing a general operator but maintaining its refinements. This means that the match cost of the general operator is reduced to zero, since the operator is removed from consideration. DoLittle will never try to match the operator. However, the general operator may still occur as part of an operator sequence in a refinement. The only cost associated with this operator is then the refinement cost. The application frequency for an operator that can only be used by reference is one, since the operator is implicitly always applicable as part of the operator sequence.

6.5 Case learner

The following section describes some important characteristics of a case learner and DoLittle's case learner in particular. The motivation behind a case learner is to be able to solve the same or structurally similar problems more efficiently in the future. This requires that the planner (a) has a cache of previous solutions, and (b) has an indexing mechanism that returns a suitable plan from the cache to solve a new problem.

DoLittle's indexing mechanism is described in subsection ??. The task of the case learner is to create new general operators that will speed up performance in the future.

DoLittle's case learner creates general operators with case refinements from a successful plan or partial plan. A successful plan is a solution to a problem, a successful partial plan is a plan that was generated to refine a general operator. For example, a general operator that has only a serial subgoal refinement generates a partial plan during refinement of the general operator.

First, the preconditions and effects of the successful operator sequence are computed. As in Chef, the operator sequence is not generalized (e.g., replace objects with variables) a priori, instead, DoLittle's repair methods generalize the operator sequence as required when creating a new plan. This is similar to Chef's delayed generalization policy.

The problem is to determine the applicability conditions of this new general operator, that is to find a characterization of the planner states in which this operator should be applied. So far, there was only one planner state in which the case was successful, the problem space of the problem or partial problem. Therefore, DoLittle's case learner extracts preconditions and effects for the general operator from the initial state and the goals of the problem space.

The following example shows the acquisition of a case to make tea in the kitchen domain. First, DoLittle is given the problem of making tea in the kitchen domain. The initial state is the one depicted in figure ??, and the goal is to have a cup of tea. Additionally, the goal contains some constraints on the plan, so that the kitchen is not a mess afterwards.

Case learner example

Problem:	Make tea in the kitchen domain
Initial State:	see figure ??
Goals:	(CONTAINS $CUP TEA)
	(WATER-OFF)
	(IS-IN OLD-TEA-BAG GARBAGE-CAN)

DOLITTLE solves this problem and creates a plan containing 30 primitive operators. The successful plan is the one shown previously in table ??. DOLITTLE's case learner first generates a case refinement by (a) computing instantiated preconditions and effects of the operator sequence, (b) computing the parameter list, and (c) generating the list of operator references. This information creates the following refinement.

Case learner example (continued)

Refinement CASE-REF1
Mode GENERIC
Preconditions (IS-REACHABLE CUPBOARD AT-TABLE)
(IS-REACHABLE TABLE AT-TABLE)
(IS-REACHABLE MICROWAVE AT-STOVE)
(IS-REACHABLE SHELF AT-SINK)
(IS-REACHABLE GARBAGE-CAN AT-SINK)
(NEXT-TO AT-TABLE AT-SINK)
(NEXT-TO AT-TABLE AT-STOVE)
(NEXT-TO AT-TABLE AT-STOVE)
(NOT (IS-OPEN CUPBOARD))
(IS-IN CUP1 CUPBOARD)
(CONTAINS CUP1 NOTHING)
(NOT (IS-OPEN MICROWAVE))
(MICROWAVE-EMPTY)
(NOT (WATER-ON))
(SINK-EMPTY)
(IS-ON TEA-BOX SHELF)
(NOT (IS-OPEN TEA-BOX))
(IS-AT ROBBY AT-TABLE)
(ARM-EMPTY)
Effects (IS-OPEN CUPBOARD)

(NOT (IS-IN CUP1 CUPBOARD))
(IS-ON CUP1 TABLE)
(NOT (CONTAINS CUP1 NOTHING))
(CONTAINS CUP1 TEA)
(IS-HOT CUP1)
(IS-OPEN MICROWAVE)
(NOT (IS-ON TEA-BOX SHELF))
(IS-ON TEA-BOX TABLE)
(IS-OPEN TEA-BOX)
(IS-AT ROBBY AT-SINK)
(NOT (IS-AT ROBBY AT-TABLE))
(IS-IN OLD-TEA-BAG GARBAGE-CAN)
Sequence OPEN-DOOR CUPBOARD
...
PUT-IN-GARBAGE-CAN OLD-TEA-BAG

Next, DOLITTLE's case learner computes the applicability conditions for the new case. The refinement is associated with a general operator and the preconditions and effects of the operator are computed as follows. First, the objects in the preconditions and effects are parameterized. The open goals of the general operator are the subset of the effects of the refinement that match the goals of the original problem space. In the example, the original goal literal (ARM-EMPTY) is missing from the set of open goals, since it is already established by the preconditions. The preconditions are the parameterized preconditions of the refinement. Since the preconditions are the weakest conditions that guarantee achievability of the goals, the applicability conditions are therefore not based on the entire initial state, but only on the ones relevant to the current goal. This is similar to case-based planning in Prodigy, which is based on foot-print literals [**?**].

Case learner example (continued)

Operator CASE-MAKE-TEA
Params $LOC1,$LOC2,$LOC3,$CUP

Preconditions (IS-REACHABLE CUPBOARD $LOC1)
(IS-REACHABLE TABLE $LOC1)
(IS-REACHABLE MICROWAVE $LOC2)
(IS-REACHABLE SHELF $LOC3)
(IS-REACHABLE GARBAGE-CAN $LOC3)
(NEXT-TO $LOC1 $LOC2)
(NEXT-TO $LOC1 $LOC3)
(NOT (IS-OPEN CUPBOARD))
(IS-IN $CUP CUPBOARD)
(CONTAINS $CUP NOTHING)
(NOT (IS-OPEN MICROWAVE))
(MICROWAVE-EMPTY)
(NOT (WATER-ON))
(SINK-EMPTY)
(IS-ON TEA-BOX SHELF)
(NOT (IS-OPEN TEA-BOX))
(IS-AT ROBBY $LOC1)
(ARM-EMPTY)
Open goals (CONTAINS $CUP TEA)
(IS-IN OLD-TEA-BAG GARBAGE-CAN)
Effects (IS-OPEN CUPBOARD)
(NOT (IS-IN $CUP CUPBOARD))
(IS-ON $CUP TABLE)
(NOT (CONTAINS $CUP NOTHING))
(CONTAINS $CUP TEA)
(IS-HOT $CUP)
(IS-OPEN MICROWAVE)
(NOT (IS-ON TEA-BOX SHELF))
(IS-ON TEA-BOX TABLE)
(IS-OPEN TEA-BOX)
(IS-AT ROBBY $LOC3)
(NOT (IS-AT ROBBY $LOC1))
(IS-IN OLD-TEA-BAG GARBAGE-CAN)
Refinements CASE-REF1 (see above)

DOLITTLE's case learner proposes one new general operator for each successful (partial) plan. The table above shows the general operator that was

generated after solving the problem of making tea.

The new general operator is then tested to see whether it satisfies the sufficient conditions of the case learner. The sufficient conditions for a case learner are derived from an analysis of a simplified model of DoLittle, as described in subsection ??.

The average search cost is estimated as the total search cost of the current problem. Since the new case is a generic refinement, the new case is first tried as a macro, and only if that fails as a case. The refinement cost for a macro is simply the cost of refining a general operator with one operator sequence. The refinement cost of a case is the refinement cost of a macro plus the cost of refining one general operator with an operator sequence and applying each operator in the operator sequence. DoLittle's case learner uses this estimate for the refinement cost. In the worst case, the match cost is the cost of searching the space of possible instances for all free variables. The match cost can thus be estimated as the product of the size of the possible instance sets for all free variables.

For example, solving the problem of making tea required 4586 nodes. The average search cost of the case is estimated as 4586 nodes. Since there are no additional refinements and no additional free variables, the refinement cost is 31, the cost of applying a macro (1) and then an operator sequence with 30 steps in it (30). There are three location variables with four possible instantiation, and one container variable with three possible instantiations. The total match cost is thus 192 nodes. Using these estimates, the new general operator passes both plausibility checks, the one to limit the refinement cost (AvrSearchCost $>$ AvrRefinementCost), and the one to limit the match cost (AvrSearchCost $-$ AvrRefinementCost $>$ MatchCost). Therefore, DoLittle's case learner adds a plan to make tea to its general operator set.

Putting these values in equation ??, and solving for the application frequency shows that this operator must be applicable with a frequency of at least four percent if its match frequency is one.

6.6 Macro-operator learner

The earliest approaches that fell into this category were EBL based macro-operator learners [?, ?]. More recently, there have also been inductive methods

6.6.1 The optimal tunneling heuristic

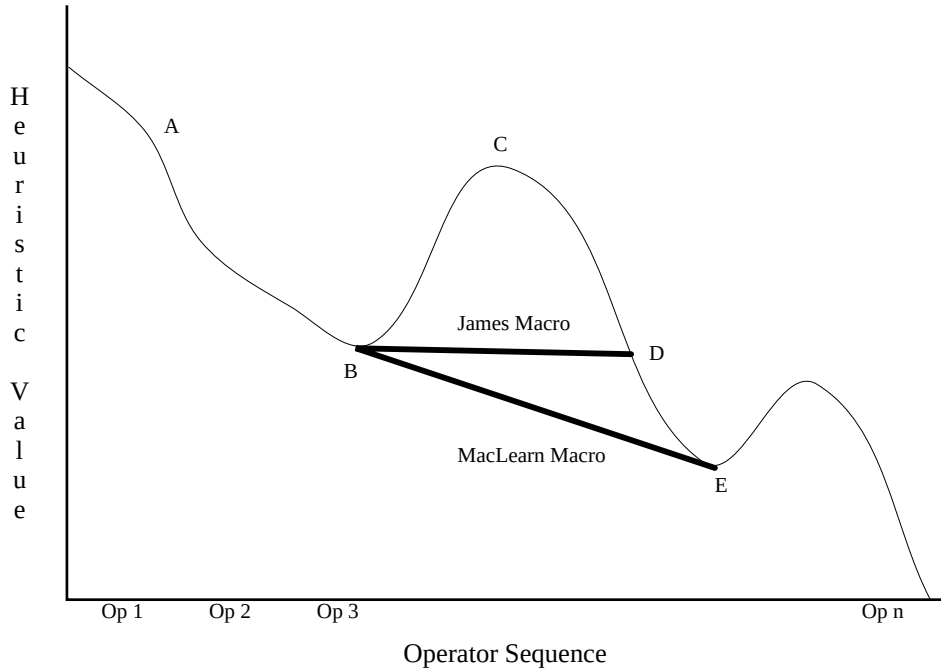
This section first describes Iba's peak to peak heuristic ([?]) and then James' optimal tunneling heuristic [?]. Iba's motivation for the peak to peak heuristic is that a macro learner must consider the relative difficulty of a problem. Since each macro increases the branching factor and the cost of node expansion, a macro should be general, so that it can be applied in a variety of instances and thus speed up the search.

Many macro-learners use a heuristic function to learn useful subsequences. DoLittle's macro-learner is based on Iba's [?] MacLearn system and James's optimal tunneling system [?]. The aim is to generate macros that allow the planner to traverse difficult parts of the search space.

Figure ?? is a graphical representation. It shows the value of the heuristic function along the operator sequence of the plan. The heuristic function increases with distance to the goal. From the start state, the search progresses smoothly up to point A. At this point, the search follows a local minimum in the heuristic evaluation function to point B. The planner has to search the surface of the valley to escape the local minimum, since each point on the surface has a better heuristic value than point C. Point E is the next local minimum in the search. Iba's system creates macros from one local minimum in the search space to the next. Therefore, MacLearn creates a macro from point B to point E. James' system creates a macro from the local minimum to the first point in the search that has a lower heuristic value than the local minimum. For example, James' optimal tunneling heuristic will create a macro from point B to D. The intuition is that after point D,

the search follows the heuristic function and thus there is no need for search control knowledge past point D. Therefore, James' system is able to create shorter, and hence more general macros. This need not always be the case. For example, if the next local minimum is at a higher heuristic value than the previous local minimum, James' system may create longer macros than MacLearn. James' design assumes best-first rather than depth-first search.

Figure 6.1: Comparison of Iba's and James' macro learners



6.6.2 DoLittle's Macro-bias learner

Applying James' optimal tunneling heuristic to DoLittle's representation is not straight-forward, since DoLittle does not use a heuristic function. Therefore, DoLittle's macro learner extends Minton's work on the MOR-

RIS system ([?]) by inferring heuristic values for nodes on the solution path as shown in the remainder of this section. The problem is how to determine that we are in a valley of the heuristic evaluation function.

As MacLearn does, DoLittle's macro learner separates the generation of a plan into difficult and easy sections. The goal is to create a macro that allows the planner to skip a difficult section. However, unlike the forward chaining planners described in the previous section, the application of this principle to a means-ends analysis planner is made more difficult since a means-ends planner inserts operators at the end of the prefix plan and at the beginning of the suffix plan. Furthermore, DoLittle with its even more powerful plan transformations may also reorder or replace operators. However, since applying an operator, adapting a plan, and selecting a refinement are more constrained than adding an operator, DoLittle's macro learner focuses on the cost of adding an operator. When adding an operator, there is much less information available than during the other plan transformations, and thus a higher chance that the planner makes the wrong choice and has to backtrack later. This means that DoLittle's macro learner may create operator sequences that do not occur in the solution, but were added sequentially to the plan. In the final solution, however, the order of the operators may be changed or operators may even be replaced by other operators or removed entirely.

Following is an example of DoLittle's macro-bias learner in the kitchen domain. The making tea problem contains the following subsequence: Pick-Up-From-Cupboard Cup1, Move-Robot at-table at-sink. Adding this subsequence requires 287 nodes and it is thus proposed as a new macro. The preconditions and effects of this subsequence are computed and objects are parameterized, to generate a refinement for the macro operator.

Macro learner example

Refinement MACRO-REF1

Mode GENERIC
Variables $Curr-Loc,$New-Loc,$Cup
Preconditions (is-reachable cupboard $Curr-Loc)
(next-to $Curr-Loc $New-Loc)
(is-open cupboard)
(is-in $Cup cupboard)
(contains $Cup nothing)
(is-at Robby $Curr-Loc)
Effects (holding $Cup)
(not (arm-empty))
(not (is-in Cup1 cupboard))
(is-at Robby $New-Loc)
(not (is-at Robby $Curr-Loc))
Sequence Pick-Up-From-Cupboard $Cup
Move-Robot $Curr-Loc $New-Loc

To complete the macro-operator, DoLittle's macro learner creates a set of associated general operators. One for each effect that is used as a subgoal. There are two effects of the operator sequence that DoLittle subgoaled on: (holding $cup) and (is-at Robby $new-Loc). The first generated general operator is shown below. This general operator is only applicable if the planner is trying to achieve (holding $cup).

Macro learner example (continued)

Operator MACRO-OP1
Variables $Curr-Loc,$New-Loc,$Cup
Preconditions (is-reachable cupboard $Curr-Loc)
(next-to $Curr-Loc $New-Loc)
(is-open cupboard)
(is-in $Cup cupboard)
(contains $Cup nothing)
(is-at Robby $Curr-Loc)
Open goals (holding $cup)
Effects (holding $Cup)
(not (arm-empty))

(NOT (IS-IN CUP1 CUPBOARD))
(IS-AT ROBBY $NEW-LOC)
(NOT (IS-AT ROBBY $CURR-LOC))
Refinements MACRO-REF1 (see above)

The second general operator that is applicable when trying to achieve (IS-AT ROBBY $NEW-LOC) is shown below.

Macro learner example (continued)

Operator MACRO-OP2
Variables $CURR-LOC,$NEW-LOC,$CUP
Preconditions (IS-REACHABLE CUPBOARD $CURR-LOC)
(NEXT-TO $CURR-LOC $NEW-LOC)
(IS-OPEN CUPBOARD)
(IS-IN $CUP CUPBOARD)
(CONTAINS $CUP NOTHING)
(IS-AT ROBBY $CURR-LOC)
Open goals (IS-AT ROBBY $NEW-LOC)
Effects (HOLDING $CUP)
(NOT (ARM-EMPTY))
(NOT (IS-IN CUP1 CUPBOARD))
(IS-AT ROBBY $NEW-LOC)
(NOT (IS-AT ROBBY $CURR-LOC))
Refinements MACRO-REF1 (see above)

The estimate of the utility of the general operator is based on a similar analysis as that of the case learner. The average search cost is estimated as the number of nodes expanded during the search for the subsequence. In this example, DOLITTLE expanded 287 nodes to generate the subsequence. The refinement costs of a macro refinement is the additional match cost of the operator sequence. Since there are no additional variables in the refinement, and since there is only one refinement, the refinement cost is estimated to be 1. The match cost is computed identically to that of the case learner, here 48. The macro passes both plausibility checks and the two general operators

are added to the operator set. The application frequency of both general operators must be at least 16 percent.

DoLittle has the additional advantage that refinement costs, application frequencies, and match cost are collected separately for each general operator. If for example, the general operator Macro-Op2 yields a low success rate (since it seems silly to pick up a cup just to move the robot), it may be removed from the operator set, independently of general operator Macro-Op1.

6.7 Abstraction learner

DoLittle's abstraction learner is based on Knoblock's ordered monotonicity abstraction generator Alpine [?]. The Alpine system is a partial planning strategy and thus not well suited to be compared to the case and macro learner. It is also too conservative in the creation of abstraction hierarchies. Alpine is unable to create an abstraction hierarchy for the blocksworld and can only create a two level abstraction hierarchy for the kitchen domain. This is because Alpine's abstraction hierarchies attempt to reduce the cost by preventing goal interferences in all problems.

The design of DoLittle's abstraction learner is one that is similar to the case and macro learner described in the previous sections, but it is based on different assumptions. DoLittle's abstraction learner generates a new abstract operator from a subsequence of a successful plan.

There is a strong relationship between justified plans and ordered abstraction hierarchies. For example, Knoblock's definition of ordered monotonicity is based on backwards justified plans [?]. Ordered monotonicity is a sufficient condition to guarantee that all generated plans are backwards justified.

DoLittle's abstraction learner uses backwards justified plans in a slightly different manner. Instead of guaranteeing that *all* generated plans are backwards justified, the abstraction learner focuses on backwards justified

operator sequences to guarantee that *some* plans are backwards justified. Although a sequence of backwards justified operators is not guaranteed to lead to ordered monotonic abstraction hierarchies in all cases, it leads to abstract operators that in at least one instance resulted in backwards justified plans. This means that DoLittle's abstraction learner is less constrained than Alpine's abstraction creator, that is it is able to generate abstractions for which Alpine is unable to generate any. On the other hand, DoLittle's abstraction learner may generate abstract operators that do not improve performance; the empirical evaluation system will remove those operators because of their poor utility. As will be shown in section **??**, the abstraction hierarchies generated by DoLittle's abstraction learner and Alpine are similar in the towers of Hanoi domain.

An operator O1 establishes a literal l for another operator O2 if and only if (a) O1 comes before O2 in the plan, (b) if the literal l is a precondition of operator O2, and (c) there is no other operator between O1 and O2 that either also establishes or negates literal l. Using the definition of establishment, a backwards justified operator is one that establishes a precondition for either a goal or another backward justified operator.

An operator is immediately backwards justified if operator O1 establishes a literal for operator O2, and O2 is the successor of O1 in the plan.

DoLittle's abstraction learner extracts sequences of immediately backwards justified operators from a successful plan to create abstract operators. The immediately backwards justified operator sequence is grouped together with the operator whose preconditions are established (the last operator in the sequence). The immediately backwards justified sequence and the last operator of the sequence form the set of refinements of a new abstract general operator. The intuition behind this method is that immediately backwards justified operator sequences are in the plan simply to set up the preconditions of the last operator.

For example, given the plan to make tea shown in table **??**, there are

11 immediately justified operator sequences. The sequences are shown in table ??. Steps 6 and 7 yield no sequences. Six sequences (3,4,5,8,9,11) lead to abstract operators that drop the current position of the robot from the preconditions of the following operator. Similarly, two sequences (1,7) abstract the state (open/closed) of an appliance (cupboard/microwave). The remaining sequences drop more than one literal.

Next, DoLittle's abstraction learner extracts a sequence, for example sequence 10, and generates one refinement for this sequence with parameterized arguments. The preconditions and effects of this sequence are as follows:

Abstraction learner example (Sequence 10)

Refinement ABSTRACT-REF1
  Mode GENERIC
  Variables $FROM-LOC,$TO-LOC,$CUP
  Preconditions (NEXT-TO $FROM-LOC $TO-LOC)
    (IS-AT ROBBY $FROM-LOC)
    (IS-REACHABLE TABLE $TO-LOC)
    (HOLDING TEA-BOX)
    (NOT (IS-OPEN TEA-BOX))
    (CONTAINS $CUP HOT-WATER)
  Effects (IS-AT ROBBY $TO-LOC)
    (NOT (IS-AT ROBBY $FROM-LOC))
    (IS-ON TEA-BOX TABLE)
    (NOT (HOLDING TEA-BOX))
    (IS-OPEN TEA-BOX)
    (NOT (CONTAINS $CUP HOT-WATER))
    (CONTAINS $CUP TEA)
    (HOLDING OLD-TEA-BAG)
  Sequence MOVE-ROBOT $FROM-LOC $TO-LOC
    PUT-ON-TABLE TEA-BOX
    OPEN-CONTAINER TEA-BOX
    GET-TEA-BAG
    MAKE-TEA $CUP

Table 6.1: Immediately justified operator sequences in the plan to make tea

1	OPEN-DOOR CUPBOARD	Sequence 1
2	PICK-UP-FROM-CUPBOARD CUP1	
3	MOVE-ROBOT AT-TABLE AT-SINK	Sequence 2
4	PUT-IN-SINK CUP1	
5	FILL-WITH-WATER CUP1	
6	TURN-WATER-OFF	
7	PICK-UP-FROM-SINK CUP1	
8	MOVE-ROBOT AT-SINK AT-TABLE	Sequence 3
9	PUT-ON-TABLE CUP1	
10	MOVE-ROBOT AT-TABLE AT-STOVE	Sequence 4
11	OPEN-DOOR MICROWAVE	
12	MOVE-ROBOT AT-STOVE AT-TABLE	Sequence 5
13	PICK-UP-FROM-TABLE CUP1	
14	MOVE-ROBOT AT-TABLE AT-STOVE	Sequence 6
15	PUT-IN-MICROWAVE CUP1	
16	CLOSE-DOOR MICROWAVE	
17	HEAT-WATER-IN-MICROWAVE CUP1	
18	OPEN-DOOR MICROWAVE	Sequence 7
19	PICK-UP-FROM-MICROWAVE CUP1	
20	MOVE-ROBOT AT-STOVE AT-TABLE	Sequence 8
21	PUT-ON-TABLE CUP1	
22	MOVE-ROBOT AT-TABLE AT-SINK	Sequence 9
23	PICK-UP-FROM-SHELF TEA-BOX	
24	MOVE-ROBOT AT-SINK AT-TABLE	Sequence 10
25	PUT-ON-TABLE TEA-BOX	
26	OPEN-CONTAINER TEA-BOX	
27	GET-TEA-BAG	
28	MAKE-TEA CUP1	
29	MOVE-ROBOT AT-TABLE AT-SINK	Sequence 11
30	PUT-IN-GARBAGE-CAN OLD-TEA-BAG	

Then, a second refinement is generated that contains only the last operator MAKE-TEA of the immediately backwards justified operator sequence.

Abstraction learner example (continued)

Refinement ABSTRACT-REF2
    Mode GENERIC
    Variables $TO-LOC,$CUP
    Preconditions (IS-REACHABLE TABLE $TO-LOC)
                  (IS-AT ROBBY $TO-LOC)
                  (IS-ON $CUP TABLE)
                  (HOLDING TEA-BAG)
                  (CONTAINS $CUP HOT-WATER)
    Effects (CONTAINS $CUP TEA)
            (NOT (CONTAINS $CUP HOT-WATER))
            (HOLDING OLD-TEA-BAG)
            (NOT (HOLDING TEA-BAG))
    Sequence MAKE-TEA $CUP

In the next step, DoLittle's abstraction learner creates a general operator to group the two refinements. The abstract operator's preconditions/effects are the intersection of preconditions/effects of the two refinements. The open goals of the abstract operators are the necessary effects (i.e., the effects that are used in the remainder of the plan) of the abstract operator.

Abstraction learner example (continued)

Operator ABSTRACT-MAKE-TEA
    Variables $TO-LOC,$CUP
    Preconditions (IS-REACHABLE TABLE $TO-LOC)
                  (CONTAINS $CUP HOT-WATER)
    Open goals (CONTAINS $CUP TEA)
    Effects (CONTAINS $CUP TEA)
            (NOT (CONTAINS $CUP HOT-WATER))
            (HOLDING OLD-TEA-BAG)

Refinements ABSTRACT-REF1 (see above)
ABSTRACT-REF2 (see above)

The match cost of the abstract general operator ABSTRACT-MAKE-TEA is 12 nodes, since there is one variable of type location (four instances) and one of type cup (three instances). The cost of selecting a refinement is the number of refinements 2. Refinement ABSTRACT-REF1 has an additional free variable, which means it has an additional match cost of 4. ABSTRACT-REF1 has no additional variables, and thus a match cost of 1. The refinement costs of a set of refinements is the maximum of the individual refinement costs. Thus, the refinement cost of the abstract operator is given as

$$\text{AvrRefinementCost} = 2 + 4 + 1 + \text{Max}(2, 6) = 13$$

The search cost of solving the associated search space was 217 nodes. Therefore, the abstract operator passes both plausibility tests and is added as a new general operator. It must be applicable with a probability of at least five percent.

6.8 Discussion

This chapter describes three sample planning bias learners for DOLITTLE: a case, macro, and abstraction learner. These learners are simple examples of planning bias learners, for illustrative purposes. Since DOLITTLE separates the design of the learners from that of the planner, many more different planning bias learners and more sophisticated methods are possible. The only requirements of a planning bias learner in DOLITTLE is that it can take the plan derivation as input and changes the operator set as a result of its analysis. This means that a planning bias learner can use other knowledge sources for learning such as examples, partial specifications, or comments

from the user.

An important aspect of DoLittle's learning mechanism is the use of a two level method for estimating the incremental utility of a planner transformation. The estimate is based on an approximation of incremental utility. These simplifications are necessary, since otherwise the complexity of learning is prohibitive, especially in the instructable learning paradigm which is based on the assumption that a planner must learn from few examples. The most significant assumption is that of independence, that is the learned general operators do not interfere. This assumption is made by many learning systems [?, ?, ?]. However DoLittle with its powerful applicability conditions is better able to reduce interference between different operators.

The incremental utility of a general operator is based on its average search cost, its average refinement cost, its application frequency, its maximum match cost, and its match frequency. The average search cost is based on the example from which the general operator was generated. The remaining factors are estimated and two plausibility tests eliminate poor general operators. The average refinement cost, the application frequency, and the match frequency are sampled in the following problem solving episodes.

The utility measurement in DoLittle's planning bias learners is based on Minton's work, with the main difference being that DoLittle's estimates are more conservative. It uses the worst case match cost instead of the average match cost in the estimate. In other words, DoLittle's learners are less likely than Prodigy/EBL to generate a rule. DoLittle's learners offset this by having a more powerful language for planner transformations, and thus a larger set of possible transformations.

Chapter 7

Evaluation

> The brightest flashes in the world of thought are incomplete until they have been proved to have their counterparts in the world of fact.
>
> **John Tyndall**, *Fragments of Science*, vol. II, Scientific Materialism.

The goal of this chapter is to establish two claims made in the thesis. First, the superiority of multi-strategy planning over three single strategy planners in two toy domains (blocksworld and towers of Hanoi) is shown. Secondly, the results in this chapter show that multi-strategy planning is able to solve problems in a more complex domain, the kitchen domain. In this domain, unordered subproblem coordinated multi-strategy planning performs better than a problem coordinated planner with an oracle.

7.1 Experimental methodology

In chapter **??**, we have seen examples of how multi-strategy planning can lead to an exponential speed up on some problems. Although this is important, the real test for a planning system is its performance on extended use. Testing

the performance of a planner on many problems instead of a single problem is important for intelligent assistants because in complex applications, it is impossible to foresee all problems, which is exactly the reason for providing the agent with a planning component.

Ideally, there would be a set of test domains, that a learning planning system can be compared against, similar to the machine learning database. This idea has the main advantage that it simplifies the comparison of many different planning systems. However, its two disadvantages are: (a) that since the test domains are known in advance, a program may be specifically optimized to do well on them, and (b) that the test domains may not match the problems that the system encounters in practice. In spite of those problems, this approach is nevertheless the most useful way of testing planning systems in practice.

Unfortunately, a set of test domains for planners is not as well established as the machine learning database. This is because the performance of a planner depends critically on the particular specification of a domain. For example, Knoblock's ALPINE system creates an abstraction hierarchy that reduces the complexity exponentially for one specification of the towers of Hanoi domain. However, for another equally intuitive and valid specification of this domain, ALPINE is unable to create an abstraction hierarchy, and is therefore unable to reduce the complexity [**?**]. The problem is that a domain specification is not unique and allows the user to add control knowledge. Obviously, adding this knowledge will greatly affect a planner's performance.

However, there are some popular domains that have been used in the evaluation of other planners. These include the blocksworld, the towers of Hanoi, the STRIPS world, and the machine shop scheduling domain. These domains are part of the PRODIGY4 release as well as other planning systems. The domains used in the empirical evaluation of DOLITTLE are taken from the set of PRODIGY4 domains. To show that DOLITTLE performance scales up to more complex domains, it was also tested on the kitchen domain,

described in chapter **??**.

All tests were run using the following methodology. First, to establish the performance of the single strategy planners, the experiments were run with DoLittle restricted to a single learning/planning method (case, macro, and abstraction). For example, in one experiment, the planner was trained and tested with only the macro planning bias learner enabled. The learners were also restricted to generate a specific type of refinement instead of a generic refinement, which is generated by default. The macro learner generated general operators with macro refinements. The case learner generated case refinements. The abstraction learner generated abstract general operators with two macro refinements. Next, to evaluate the multi-strategy planning approach, DoLittle was run with all planning bias learners and refinement types enabled. The planning bias learners generated only generic refinements.

After this training phase DoLittle was run on a set of newly randomly generated problems (250) with no learning enabled and with the respective learned general operator sets for macros, abstractions, cases, and multi-strategy planning.

The comparison included the following set of planners:

- **Prodigy-DL** is DoLittle running in Prodigy emulation mode, i.e., no adaptations, no general operators.
- **Case** is DoLittle running in DoLittle mode (with adaptations and general operators) with only the case learner turned on and all other planning bias learners turned off.
- **Macro** is DoLittle running in DoLittle mode with only the macro learner turned on.
- **Abstraction** is DoLittle running in DoLittle mode with only the abstraction learner turned on.

- **DoLittle** is DoLittle running in DoLittle mode with the macro, case, and abstraction learner enabled.

- **PC-MSP-O** is a hypothetical problem coordinated planner with an oracle. The time and number of nodes entry for this planner are generated by taking the planner that generated the minimum running time for the problem from the macro-, case-, and abstraction-based planners.

All planners used depth-first search during the tests. The reason for this choice is that admissible search strategies are too expensive in practice, especially in complex domains. To find an optimal (shortest) solution, the planner generally has to search the whole space. Depth first search only finds the first solution and can thus in general find a solution faster. Of course, the solution is not guaranteed to be optimal. To some degree, depth first search can trade off solution quality for solution length. DoLittle generates many new general operators which will greatly increase the search space of a breadth-first planner, whereas depth-first search uses DoLittle's indexing mechanism.

To compare the performance of the planners, the cumulative numbers for the expanded nodes and the running time are compared. These features are intuitive ways to compare the performance of different planning systems. However, the difficulty is that at some point the experimenter has to decide that a problem is unsolvable and terminate the planner. In general, this is done by choosing a resource limit for the number of expanded nodes or the running time. However, Segre et al. show that a comparison based on those statistics alone may be biased by the chosen resource limit [**?**]. Therefore, the experimental methodology also shows the total running time as a function of the time limit. This comparison shows the absolute performance of the planners, the percentage of solved problems, and allows the prediction of the planners behavior should the time limit be extended further.

7.2 Multi-strategy planning in the blocksworld

This section compares the performance of the described planning systems in the blocksworld. It shows that multi-strategy planning in the blocksworld improves performance by a factor of three over the best single strategy method (Cases) and by a factor of eight over a non-learning system (PRODIGY-DL). DOLITTLE also does slightly better than a hypothetical problem coordinated multi-strategy planner with a perfect oracle in the blocksworld.

First, the performance of multi-strategy planning in the blocksworld is compared against that of using a single strategy. For the learning methods (macros, cases, abstractions, and multi-strategy planning), each system was trained on a set of 100 training problems. The training problems were randomly generated, using exactly the same procedure as the one used to generate random test problems (see appendix ??). The case learner generated 8 new general operators. The macro learner created 14 new general operators. The abstraction learner created 4 new general operators.

The test phase consisted of running DOLITTLE on 250 newly randomly created problems. The results of this run are summarized in figure ?? and figure ??. The figures list the cumulative number of nodes and the cumulative running times over the problem set. A detailed listing of the blocksworld domain description, the procedure to generate random problems, and the results is given in appendix ??.

Figure ?? graphs the total running time of the system with increasing cut off values (time limit). The time limit was varied from 0 to 100 seconds. The absolute value of the curves shows the total time that the system takes to solve all problems as the time limit is increased. The slope of the curve shows how many more problems can be solved by an increase in the resource limit. If the value of the function does not increase, all problems have been solved. The shape of the curves suggests the future performance of a system

Figure 7.1: Cumulative nodes in the Blocksworld

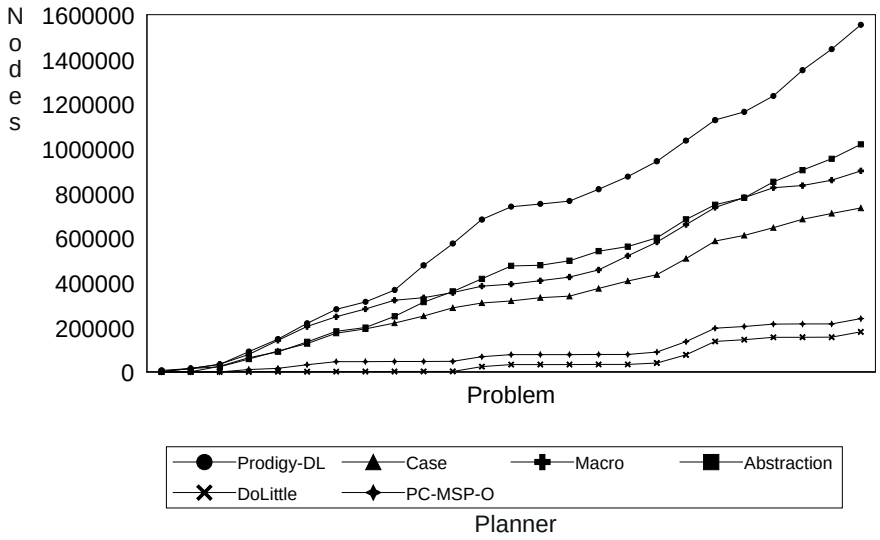
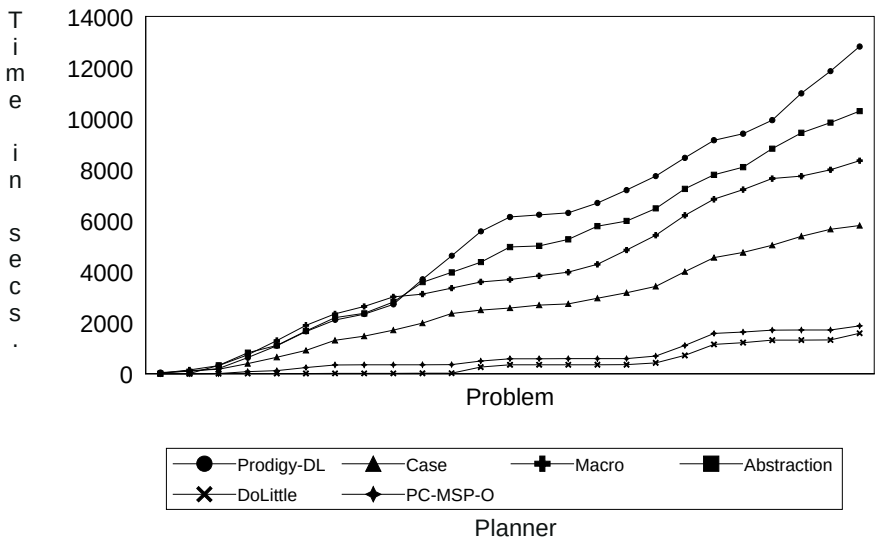


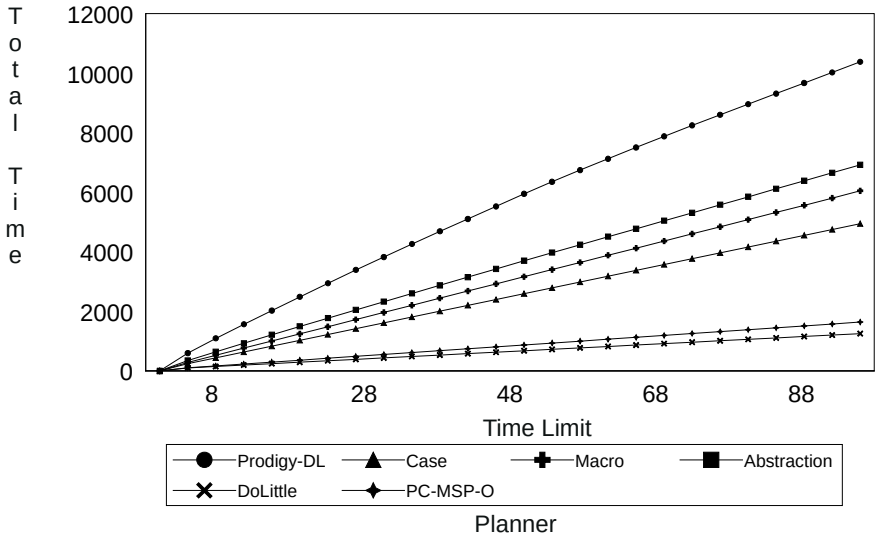
Figure 7.2: Cumulative running time in the Blocksworld



as the resource limit is increased further.

The graph shows that the multi-strategy planners solved most problems. PRODIGY-DL requires a higher time limit (50) to solve most of its problems, whereas the curves for the multi-strategy planner indicate that they were able to solve most problems in less time too. Increasing the time limit even further would not lead to a large increase in the running time of the multi-strategy planners.

Figure 7.3: Running time versus time limit in the blocksworld



The performance of the planners as seen in the values of the curves shows the same absolute performance as figure ??. DOLITTLE does slightly better than PC-MSP-O, followed by the three learning planners (cases, macros, abstractions). PRODIGY-DL has the worst performance. The abstraction-

based planner has the highest ratio of unsolved problems.

The results in the figures show that single strategy planners did improve the performance. The case learner provided the best speed up for the single-strategy planning systems. It reduced the number of expanded nodes by a factor of 2.1 and the total running time by a factor of 2.20. The reduction factors for the macro learner are 1.72 and 1.53 respectively. This is similar to the speed up reported by Minton for the PRODIGY/EBL system of around 2. The abstraction learner did improve performance slightly , but did provide the least speed up of the learning planning systems. The number of nodes was reduced by a factor of 1.52 and the running time by a factor of 1.24. This is consistent with the observation that ALPINE is unable to create any abstraction hierarchy for this domain. One would expect, therefore, that even a slightly less conservative abstraction learner will not be able to generate useful abstractions.

The reduction factors for the running time are smaller than those of the number of nodes, because of the match cost of the additional operators. The node generation rates of the different planners are: PRODIGY-DL (120.47 Nodes/sec.), case (126.26 Nodes/sec.), macro (107.80 Nodes/sec.), and abstraction (99.28 Nodes/sec.). The node generation rate of the case learner is better than that of DOLITTLE since the cases are more specialized versions of some of the primitive operators and variable bindings must only be computed once, but are used for all primitive operators in the case. The learned macros are more general and DOLITTLE generated more macros, so the macro learner has a high match cost. This leads to a decrease in the node generation rate. The abstraction learner has the lowest node generation rate, since the match cost of an abstract general operator includes the cost of matching any of the refinements if the abstract operator is applicable. Since each of the four operators has two refinements, the node generation rate is decreased because DOLITTLE may have to compute a matching for the general operator as well as its two refinements, and must compare the

two refinements.

Multi-strategy planning provided an even greater improvement. DoLittle was able to do better than the problem coordinated multi-strategy planner with an oracle, since it is able to combine planning strategies at a subproblem level, whereas PC-MSP-O can only combine strategies at a problem level. DoLittle learned fewer operators than the sum of the learned operators for the single strategy planners, since learning a specific operator can reduce the cost of solving successive problems. This in turn may lead to a smaller utility for newly suggested operators, which then may be rejected.

DoLittle reduced the number of nodes by a factor of 8.6 and the running time by a factor of 8.0. Its node generation rate was 113.04 Nodes/sec. The hypothetical planner PC-MSP-O led to a reduction factor of 6.5 for the nodes and the running time. DoLittle achieves this impressive speed up somewhat at the expense of the solution quality. The average length of the solutions generated by DoLittle are larger than that of the PC-MSP-O planner.

There are two reasons for the good performance of DoLittle in the blocksworld. Firstly, DoLittle learns cases that take a stack of blocks and unstacks all the blocks on the table. Cases are suitable in this situation since there are many different variations in initial states. Secondly, it uses macros to build towers of different sizes by picking up blocks from the table and stacking them.

Figure ?? is an example of DoLittle's improvement over single strategy planners. In this example, a macro and case are used to solve a problem efficiently. Figure (a) shows a case that DoLittle learned in previous planning episodes. The case reverses a tower of height three. Figure (b) is a macro that builds a tower of height three. The new problem is shown in figure (c). It is similar to the problem in figure (a), but after putting block (A) on the table (shown in figure (d)), DoLittle interrupts the execution of the case and adapts it to the new situation by adding the macro to the plan. Then

the remainder of the plan is executed to complete the solution. In this case, very little search is necessary. If the case were not available, DoLittle has to search for a solution to the problem of reversing the tower. Similarly, if the macro were not available, DoLittle has to build a tower of height three.

On the other hand, DoLittle does not solve all problems more efficiently than the single strategy planners. There are some problems that the case or macro-learner solve more efficiently than DoLittle. The reason for this is that the applicability conditions of a general operator are not alwayys correct, and thus may lead DoLittle away from a solution. Figure ?? is an example of such a situation. In this example a combination of a case and macro leads DoLittle astray.

First, figures (a) and (b) show a case and macro that DoLittle acquired during the training phase. The macro is identical to the one used in the good example, because all block references are parameterized in a macro. The names of the blocks were changed to show the instantiation of these variables. The new problem is shown in figure (c). Although superficially similar to the problem in figure (a), the solution is very different. In fact, the solution only requires three block movements (each movement consists of a Pick-Up,Stack pair).

Given the additional case and macro, however, DoLittle starts applying the case, until all the blocks are unstacked. At this point, DoLittle applies the macro and continues with the case, which leads to the current state shown in figure (e). Solving the problem from this state is much more difficult (12 block moves), and DoLittle fails eventually because it runs out of resources. On the other hand, given only the macro or the case, DoLittle still is able to solve the problem.

In this case, the cost of refinement for the case and macro are high, so that their utility estimates are decreased. Should the expected utility become negative, DoLittle will remove the offending operator. DoLittle may, however, maintain these operators, if they lead to improved performance in

Figure 7.4: A good problem for DoLittle in the blocksworld domain

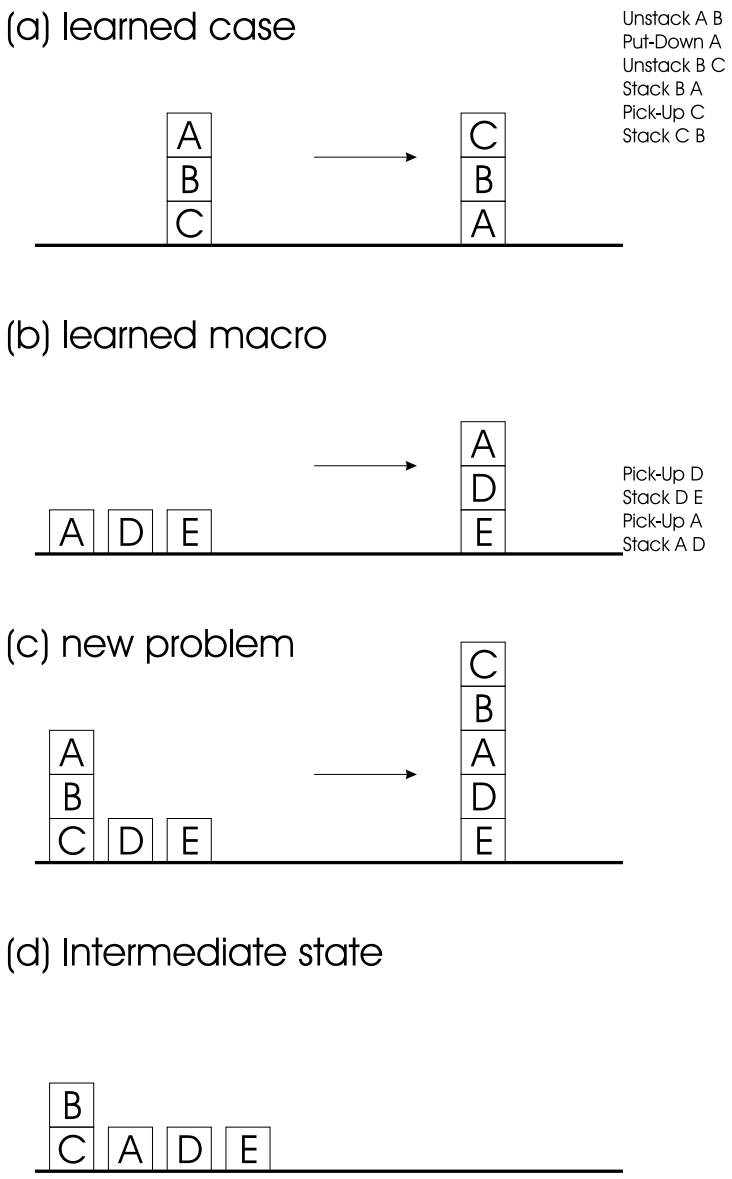
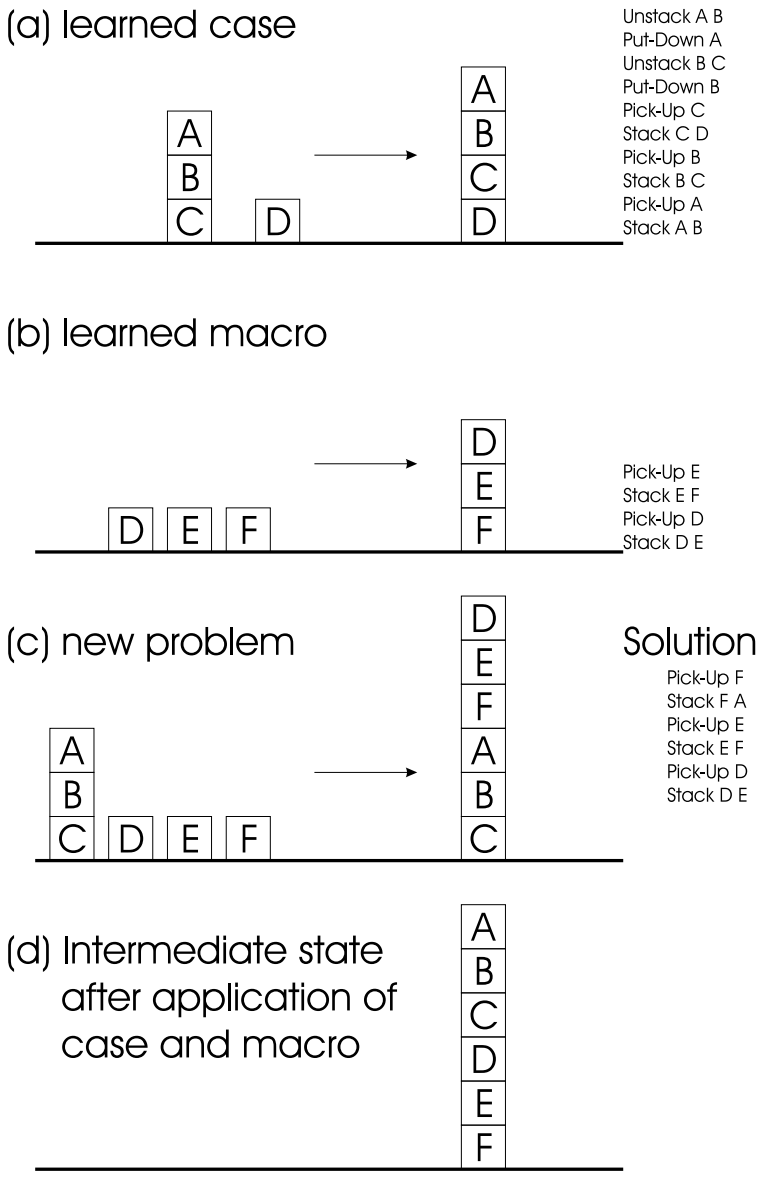


Figure 7.5: A bad problem for DoLittle in the blocksworld domain



other problems. For example, the macro was used successfully in the good example.

To establish the statistical significance of those results, a paired t-test is used. With 95 percent confidence, the difference between PRODIGY-DL and the abstraction-based planner is statistically significant (p-value 0.04) as is the difference between the case-based planner and the PC-MSP-O planner (p-value $2.23 * 10^{-8}$). The difference between PC-MSP-O and DOLITTLE is statistically not significant (p-value 0.15). The p-value shows that the observed difference in the mean of the running times (7.78 for PC-MSP-O and 6.59 for DOLITTLE) occurs statistically 15 percent of the time, even if both systems have identical mean running times.

Table ?? summarizes the results of the experiments in the blocksworld. The first row of each entry contains the total number of nodes and the total running time for solving all 250 problems. The second row contains the node generation rate and the number of successfully solved problems. The third row contains the average length of the solutions and the number of primitive and learned operators in the domain. The number of operators of the hypothetical planner PC-MSP-O is not given since the operator set depended on the most efficient planner for a given problem.

7.3 Multi-strategy planning in the towers of Hanoi

This section describes the results of the experiments run in the towers of Hanoi domain. The towers of Hanoi domain is a very popular domain. There are four disks of different size (small, medium, large, huge) and three pegs. The rules state that a disk can be moved from one peg to another if only if it does not end up on top of a smaller disk.

The experiments in the towers of Hanoi domain used the same methodology as the ones in the blocksworld. First, the learners are trained on 100

Table 7.1: Results of the blocksworld

PRODIGY-DL	1,588,614 Cum. Nodes	13,186.96 Cum. Secs.
	120.47 Nodes/sec.	161 Solutions
	3.54 avg. sol. length	4 + 0 operators
Abstraction	1,061,028 Cum. Nodes	10,686.94 Cum. Secs.
	99.28 Nodes/sec.	183 Solutions
	4.02 avg. sol. length	4 + 4 operators
Macro	907,051 Cum. Nodes	8,414.15 Cum. Secs.
	107.80 Nodes/sec.	190 Solutions
	4.41 avg. sol. length	4 + 14 operators
Case	741,144 Cum. Nodes	5,870.16 Cum. Secs.
	126.26 Nodes/sec.	201 Solutions
	4.42 avg. sol. length	4 + 8 operators
PC-MSP-O	245,533 Cum. Nodes	1,945.70 Cum. Secs.
	126.19 Nodes/sec.	234 Solutions
	3.19 avg. sol. length	
DOLITTLE	186,320 Cum. Nodes	1,648.22 Cum. Secs.
	113.04 Nodes/sec.	238 Solutions
	4.74 avg. sol. length	4 + 16 operators

randomly generated training problems. Then, the systems are tested on 250 randomly generated test problems. A detailed listing of the domain description, the procedure used to generate problems, and the results is given in appendix ??.

The abstraction learner generated one new operator. This new operator moved the large disk ignoring the position of the small and medium disk. A more detailed analysis showed that the learner also suggested abstract operators for the other disks. However, the abstract operators for the small and medium disk were rejected since they did not lead to sufficient savings. The abstract operator to move the huge disk was removed from the operator set, since its refinement cost was too high.

The macro learner generated four new operators, the case learner generated 14 new operators, and DoLittle created five new operators. DoLittle created the same abstract operator to move the large disk as the abstract operator. The other four operators created by DoLittle were created by the case learner.

Figures ?? and ?? summarize the results of the test runs. The graphs display the cumulative number of expanded nodes and running time respectively.

In the tower of Hanoi domain, the abstraction learner proved to be the best single strategy planner. The abstraction learner reduced the total number of nodes by a factor of 2.2 and the total running time by a factor of 2.5. This is not surprising, since the towers of Hanoi domain lends itself very well to abstraction-based planning [?, ?]. The performance of the abstraction learner is comparable to that of Alpine. Knoblock shows that although Alpine leads to an exponential reduction for admissible search procedures, it leads to only a moderate improvement for depth first search. In fact, the performance of the abstraction learner is better than the reduction factor (1.25) reported by Knoblock.

Somewhat surprising was the poor performance of the case learner. Fur-

Figure 7.6: Cumulative nodes in the towers of Hanoi

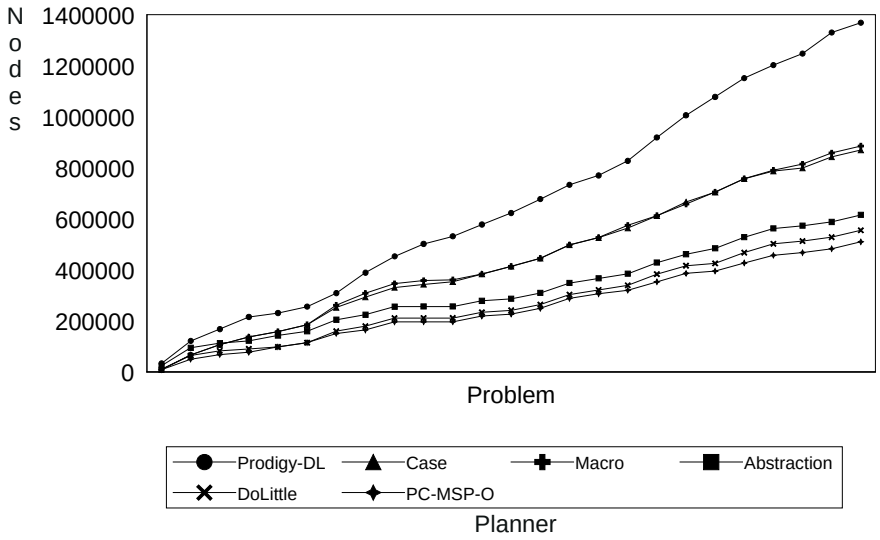
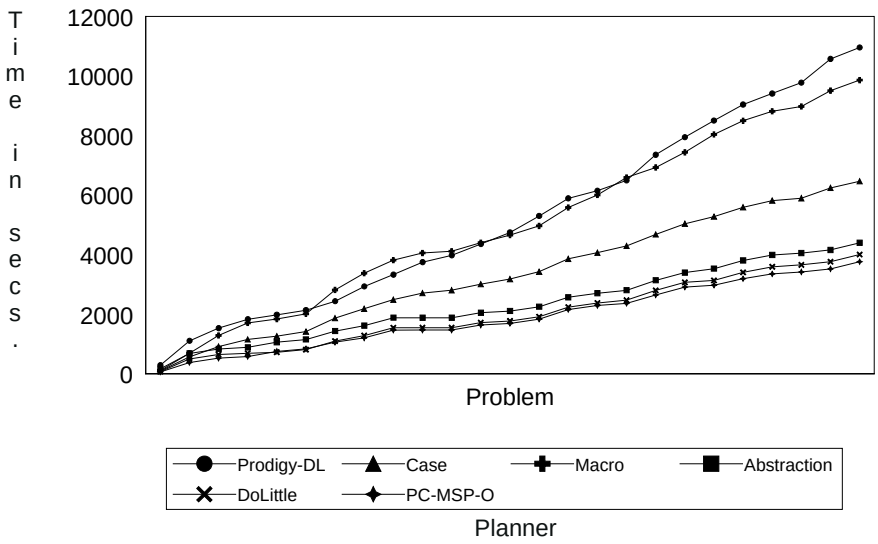


Figure 7.7: Cumulative running time in the towers of Hanoi



ther investigation showed that the problem was that plan adaptation in the towers of Hanoi domain is more difficult than in the blocksworld or kitchen domain, since the towers of Hanoi domain is very restricted. For example, assume that DoLittle retrieves a plan to move the largest disk, but has to replace Peg1 with Peg2 during adaptation of the plan. In the towers of Hanoi domain, this requires a change of all other peg assignments in the remainder of the plan. In the blocksworld, it requires only two changes (for the supporting block and the block on top).

The worst performance was that of the macro learner which reduced the total number of nodes by a factor of 1.5 and the running time by a factor of 1.1.

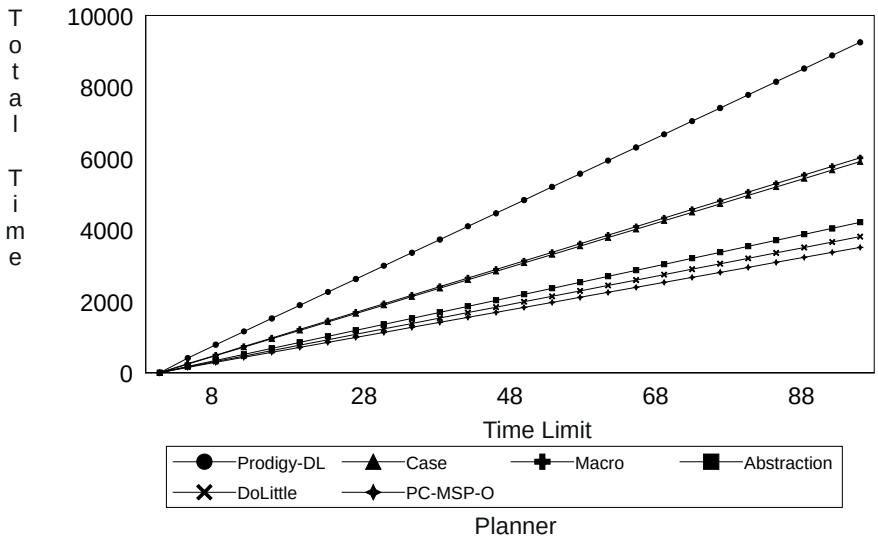
Comparing the performance of the multi-strategy planners shows that DoLittle does better than the abstraction learner, but slightly worse than the hypothetical PC-MSP-O planner. DoLittle reduces the total number of nodes by a factor of 2.4 and the total running time by a factor of 2.7.

The reduction factors are smaller in this domain as compared to the blocksworld, since Prodigy-DL is already a relatively efficient planner in the towers of Hanoi domain. This is especially true when using depth first search. The number of goal interactions is reduced since the Prodigy-DL will simply move a disk again, if it was moved to the wrong spot.

Figure **??** shows the effect of the time limit on the total running time of the system. Apart from the cumulative running time, which is the absolute value of the curve, the slope of the curve shows how many problems remain unsolved. Most problems are solved quickly by the planners. The multi-strategy planners had solved significantly more problems than the single strategy planners. The graph shows that an increase in the time limit will not change the results substantially. The predicted performance of the planners is similar to the one described.

A paired t-test was used to test the statistical significance (95 percent confidence) of the results in the towers of Hanoi domain. Comparing the

Figure 7.8: Running time versus time limit in the towers of Hanoi



performance of PRODIGY-DL and the abstraction learner shows that this difference is statistically significant (p-value $8.2 * 10^{-13}$). The difference between the abstraction learner and DOLITTLE is not statistically significant (p-value 0.07) and neither is the difference between DOLITTLE and the PC-MSP-O planner (p-value 0.32). These results indicate that the observed difference occurs with a probability of 7 and 32 percent, even if both planners have the same mean running time. There is a high correlation between DOLITTLE and the other two planners (0.94 for the abstraction learner, 0.92 for PC-MSP-O).

Table ?? summarizes the results of the experiments in the towers of Hanoi domain. The fields are identical to the ones in the summary table of the blocksworld (table ??).

7.4 Multi-strategy planning in the kitchen domain

This section shows that DOLITTLE is able to handle problems in at least one complex domain, the kitchen domain. The kitchen domain is more complex than the blocksworld or the towers of Hanoi, both in terms of the number of objects and of the number of operators in the domain. For example, there are four operators in the blocksworld and 51 operators in the kitchen domain. Also, the blocksworld contains at most twelve blocks whereas the kitchen domain contains between 45 and 51 objects, dependent on how many cups and glasses are included.

The problem in the blocksworld is the deep level of subgoaling in the domain, whereas the subgoaling in the kitchen domain is shallow. However, solutions in the kitchen domain are much longer than in the blocksworld.

The kitchen domain is too complex to solve anything but simple problems. Since DOLITTLE's case-, abstraction-, and macro-based learners currently only learn from success, it follows that if the original training set is too

Table 7.2: Results of the towers of Hanoi

PRODIGY-DL	1,398,419 Cum. Nodes	11,237.25 Cum. Secs.
	124.44 Nodes/sec.	158 Solutions
	3.91 avg. sol. length	4 + 0 operators
Abstraction	633,282 Cum. Nodes	4494.52 Cum. Secs.
	140.90 Nodes/sec.	208 Solutions
	3.28 avg. sol. length	4 + 1 operators
Macro	903,242 Cum. Nodes	9,960.81 Cum. Secs.
	90.68 Nodes/sec.	190 Solutions
	3.34 avg. sol. length	4 + 4 operators
Case	887,357 Cum. Nodes	5,553.56 Cum. Secs.
	135.40 Nodes/sec.	191 Solutions
	3.01 avg. sol. length	4 + 14 operators
PC-MSP-O	528,130 Cum. Nodes	3,859.19 Cum. Secs.
	136.85 Nodes/sec.	215 Solutions
	2.46 avg. sol. length	
DOLITTLE	573,333 Cum. Nodes	4,100.14 Cum. Secs.
	139.83 Nodes/sec.	212 Solutions
	3.47 avg. sol. length	4 + 5 operators

difficult, and DoLittle can only solve few problems, it may not be able to generate any useful new general operators. Therefore, the training set was (a) increased to 150 problems and (b) the difficulty of the problems in the training set and the test set were gradually increased. The idea is that by solving small simple problems in the beginning, DoLittle can learn the necessary general operators to solve more complex problems in the future. This method is similar to Minton's strategy for training selection [**?**] and is also applicable in the instructable systems paradigm, which assumes that the user teaches the system how to do complex tasks by solving simple ones first. DoLittle used five groups of 30 problems in the training phase and five groups of 50 problems in the test phase.

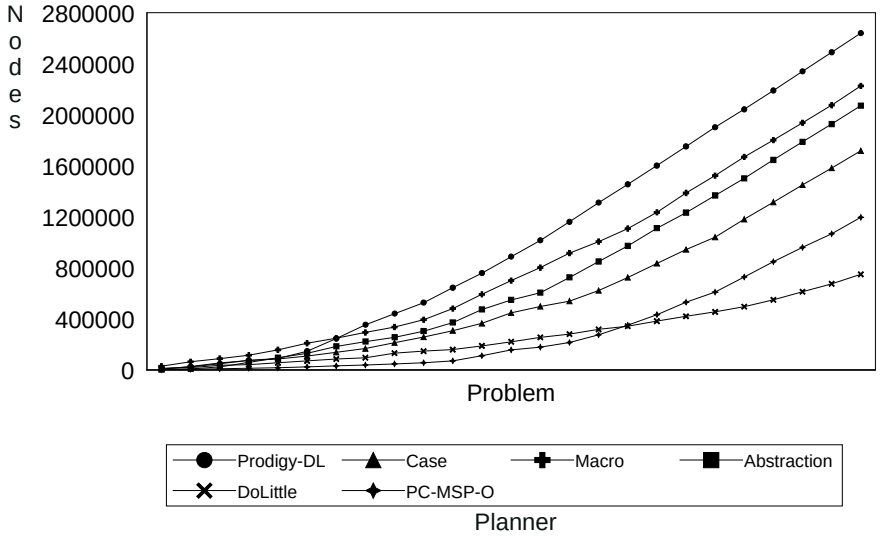
Note that the difficulty of the created problems does not increase strictly monotonic, since the difficulty does only limit the maximum difficulty of a problem, but does not guarantee that a problem with maximal difficulty is generated. In the kitchen domain, the difficulty is determined by the number and tye of drinks, that the robot has to create. The number of beverages in the goal is uniformly distributed from 0 to the maximum number of beverages (3).

The procedure for randomly generating the training and test problems is shown in appendix **??**. The case learner created 10 new general operators. The macro learner created 23 new macros. The abstraction learner added 16 new general operators to its operator set.

The test phase consisted of running DoLittle on 250 newly randomly created problems. The results of this run are summarized in figure **??** and figure **??**. The figures list the cumulative number of nodes and the cumulative running times over the problem set. The graphs show that DoLittle does not perform as well as PC-MSP-O on small problems, but is better on the tougher problems in the domain. In the left part of the graphs, PC-MSP-O performs better than DoLittle. A detailed listing of the kitchen domain description, the procedure to generate random problems, and the results is

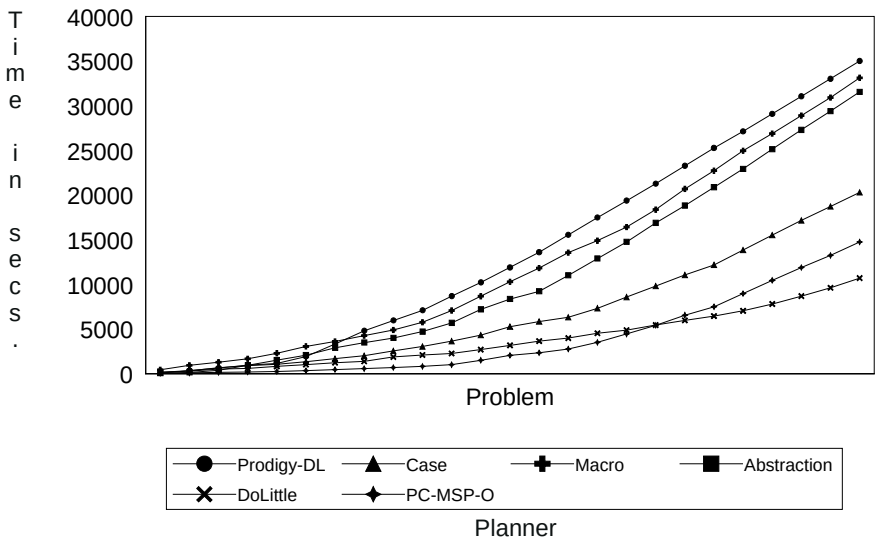
given in appendix ??.

Figure 7.9: Cumulative nodes in the kitchen domain



Similarly to the blocksworld, the single strategy planners did improve performance. Case-based planning showed itself to be the best single strategy method. This is not surprising, since the gradual increase in difficulty of the problems lend itself to a case-based approach. In fact, the earlier problems are subtasks of solving the more complex later problems. Since the graphs show the cumulative number of nodes and running time, the gradual increase in problem complexity can be seen through the change in slope of the curves. The second derivative of the curves is positive. This is not strictly true, since the experimental methodology limited the maximum complexity of a problem, but did not force the problem generator to create a maximally

Figure 7.10: Cumulative running time in the kitchen domain



difficult problem. So, although the problem generator can possibly generate a complex problem, it does not necessarily create one. In fact, also the length of the optimal solution is known of a problem, it is hard to predict the exact performance of a planner on such a problem, since it depends on the accuracy of the applicability conditions of the different operators.

The abstraction learner did better in this domain than the macro learner. This can be attributed to the fact that plans in the kitchen domain contain long sequences of immediately backwards justified operators and that the macro learner with the short macros was unable to solve many of the more complex problems.

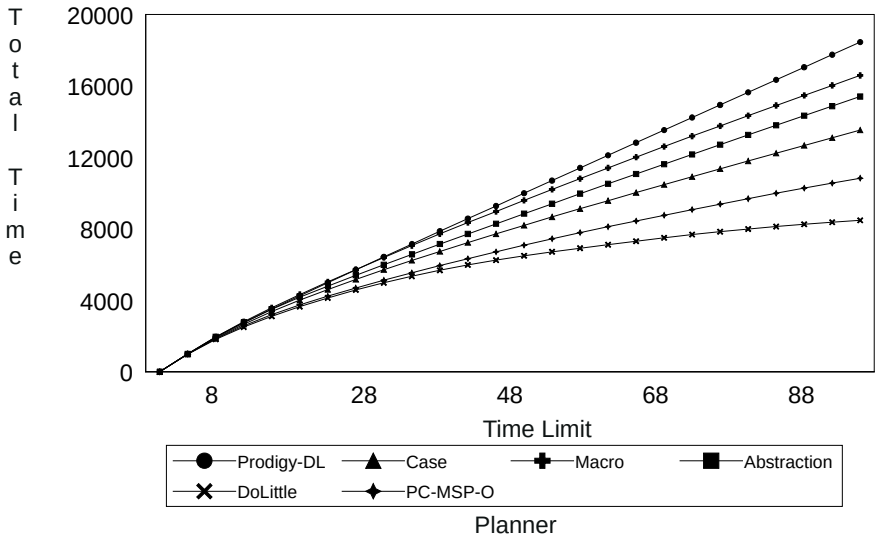
The case learner reduced the number of nodes and the running time by a factor of 1.5 and 1.7 respectively. The reduction factors of the abstraction learner are 1.2 for the number of nodes and 1.1 for the running time. The macro learner reduced the number of nodes by a factor of 1.2 and the running time by a factor of 1.05. The reduction factors of the single strategy planners is smaller than that in the blocksworld, because the learners are simple strategies. Their greatest disadvantage is that they only learn from success. In complex domains, such as the kitchen world, more powerful learning methods are necessary. The restriction to the simple planning bias learners, however, focuses the investigation on the effect of multi-strategy planning, which is the goal of this thesis.

Multi-strategy planning provided a much greater speed up than any of the single strategy planners or even the PC-MSP-O planner. The reduction factors of multi-strategy planning are 3.5 for the number of nodes and 3.2 for the running time.

Figure **??** shows the relationship between the time limit and the total running time of the system. The graph shows that DoLittle solved much more problems than the other planners. The slope of the curves for the other planners is much larger, indicating that there is a significant number of unsolved problems. Further increasing the time limit is unlikely to change

the results of the comparison.

Figure 7.11: Running time versus time limit in the kitchen domain



The statistical analysis of the results shows that the difference in performance between the case-based planner and PRODIGY-DL are statistically significant. The difference between DOLITTLE and PC-MSP-O in the kitchen domain is statistically significant with 95 percent confidence and a p-value of 0.0002.

Table ?? summarizes the results in the kitchen domain. Each entry contains the total number of nodes expanded, the total running time, the node generation rate, the number of solved problems, the average length of the generated solutions, and the number of primitive and learned operators.

Table 7.3: Results of the kitchen domain

PRODIGY-DL	2,707,539 Cum. Nodes	35,899.92 Cum. Secs.
	75.42 Nodes/sec.	74 Solutions
	26.92 avg. sol. length	51 + 0 operators
Abstraction	2,130,899 Cum. Nodes	32,419.71 Cum. Secs.
	65.73 Nodes/sec.	121 Solutions
	40.19 avg. sol. length	51 + 16 operators
Macro	2,294,079 Cum. Nodes	34,116.80 Cum. Secs.
	67.24 Nodes/sec.	110 Solutions
	40.45 avg. sol. length	51 + 23 operators
Case	1,781,524 Cum. Nodes	21,042.01 Cum. Secs.
	84.67 Nodes/sec.	152 Solutions
	53.54 avg. sol. length	51 + 10 operators
PC-MSP-O	1,252,990 Cum. Nodes	15,432.88 Cum. Secs.
	81.19 Nodes/sec.	192 Solutions
	24.44 avg. sol. length	
DOLITTLE	777,477 Cum. Nodes	11,080.69 Cum. Secs.
	70.17 Nodes/sec.	228 Solutions
	66.06 avg. sol. length	51 + 23 operators

7.5 Discussion

The empirical evaluation shows that multi-strategy planning can improve performance over single strategy planning. In all three domains (blocksworld, towers of Hanoi, and the kitchen domain) does DoLittle perform better than the single strategy planners.

More importantly, the kitchen domain shows that an unordered subproblem coordinated multi-strategy planner can perform better than a problem coordinated multi-strategy planner with an oracle. The reason for this improvement in the kitchen domain is that the subproblems resulting from the application of the first planning strategy are non trivial. In the blocksworld and the towers of Hanoi domain, the performance of DoLittle and PC-MSP-O was similar.

None of the single strategy planners is consistently better than the other strategies. The case learner performs well in the blocksworld and the kitchen domain, because the similarity metric returns appropriate plans. In the towers of Hanoi domain, the case learner is not as effective, because adapting a plan to a new situation is more difficult.

The abstraction learner performs well in the towers of Hanoi domain, since this domain can be partitioned into four abstraction levels corresponding to the four disks. It is interesting to note that DoLittle only maintains an abstract operator for the third disk and rejects abstract operators for the two smaller disks. Plans involving the two smaller disks do not lead to sufficient savings.

Chapter 8

Related Work

This chapter compares DoLittle to other related work in the area. Section ?? reviews previous work in multi-strategy planning. Section ?? compares DoLittle's macro learner to other macro-based planning systems. Section ?? compares DoLittle's abstraction learner to other abstraction-based planners. Section ?? discusses the close similarities between DoLittle's and Prodigy/EBL's utility estimates. Section ?? compares multi-strategy planning to dynamic biasing systems in machine learning.

8.1 Multi-strategy planning systems

This section discusses other multi-strategy planning systems. This thesis focuses and develops a framework of multi-strategy planning. Obviously, these previous systems are not described with respect to the framework developed in this thesis. Often, the systems are not referred to as multi-strategy planning systems, but the underlying motivation of those systems is the desire to combine different problem solving methods.

8.1.1 McCluskey's FM system

McCluskey and Porteous developed the FM system, which is aimed at combining goal reformulation, abstraction hierarchies, and control heuristic acquisition and refinement. In the multi-strategy planning framework designed in this thesis (chapter **??**), FM is an ordered subproblem coordinated multi-strategy planner.

The goal regression module attempts to create an ordering of subgoals that avoids subgoal interactions. This ordering is computed by analyzing the operators in a domain [**?**]. The output of the goal reformulator is an ordered partitioning of goal predicates, such that all goal predicates in a partition must be achieved before any of the goals in a later partition. CHAR creates heuristics for selecting operators among the set of applicable operators at a node. ABGEN generates reduced abstraction hierarchies with monotonic refinements. The algorithm is similar to ALPINE. Lower level problems are solved without affecting any of the literals at a higher abstraction level.

In contrast, DOLITTLE's combination of problem solving strategies is unordered. The ordering of planning biases is static in FM. In FM, the goal reformulator is called first, which generates subproblems for the abstraction generator. The abstraction module creates problems that are handed to the heuristics learner and the base planner (means-ends analysis). DOLITTLE is able to change dynamically the problem solving strategy during problem solving, from using cases to abstract operators for example.

Section **??** shows that DOLITTLE's representation is powerful enough to represent sequences of subgoals, applicability conditions for operators, and abstraction hierarchies. Therefore, it is possible to add the methods used by FM to DOLITTLE and combine them with other planning strategies.

8.1.2 The APS system

Another multi-strategy planning system is Gould's and Levinson's adaptive predictive search (APS) system [?]. APS is a reinforcement learner. Reinforcement learning attempts to apply the best operator (i.e., the operator that eventually will lead to the highest reward) to the current state. Therefore, APS uses the state space paradigm of problem solving. Chapter ?? shows that the state space paradigm is unable to describe more powerful planning strategies such as abstract operators or automatic subgoaling.

APS uses a pattern weight representation (pws) of control knowledge. Patterns represent sets of states and weights correspond to their significance with respect to expected rewards. A pattern is a boolean feature of a state in the search space. In general, a pattern matches a set of states and not just a single state. A weight represents an expected value of reinforcement. Search in APS progresses by finding all most specific patterns that match the current state. From this set, the pattern with the largest weight is selected and the associated operator is applied.

There are four methods for pws generation: search context rules, specialization and generalization, reverse engineering, and genetic operators. Search context rules are the only method of adding new patterns to an empty pattern set. Given a solution, a pattern that prefers making the right choice at a given state is added. Generalization is combining different patterns with similar weights. A pattern is specialized if it requires large changes in its weights. Reverse engineering generates a sequence of pws that represent a macro. The weights along the macro increase, that is the second pws in the macro has a greater weight than the first one and so on. The last pws in the macro has the greatest weight. Genetic operators are domain dependent methods for generating new patterns from the current pattern set.

APS uses only macro-operators as problem solving strategies, and focuses on multi-strategy learning to create new macros, rather than multiple planning strategies. Its retrieval of matching patterns is similar to DoLittle,

since only the most specific pattern matching a state is retrieved. DoLittle adds a context to the applicability conditions of an operator. The context identifies the problem spaces in which the general operator can be applied. APS does not contain the concept of a problem space.

8.1.3 The Alpine/EBL system

Knoblock and Minton describe experiments of combining two of Prodigy's planning strategies, abstraction-based planning (Alpine) and explanation based learning (Prodigy/EBL) [?]. In the framework described in this thesis, Alpine/EBL is an ordered subproblem coordinated multi-strategy planner.

The combination of the two strategies showed improved performance over the individual planning methods, especially in the towers of Hanoi domain. The speed up of abstraction-based planning is easily explained. Alpine creates a set of problem spaces, but has to search the individual problem spaces using means-ends analysis. Prodigy/EBL can provide search knowledge to speed up the search in the ground and abstract problem spaces. On the other hand, Alpine can improve the performance of Prodigy/EBL since the individual searches are smaller. This may lead to non-recursive explanations, whereas Prodigy/EBL would require recursive explanations. As Etzioni showed, the utility of recursive explanations is often limited and since these rules are often expensive, their cost outweighs their utility [?]. Also, abstraction can decrease the match cost of Prodigy/EBL rules, because rules are only matched in nodes belonging to the most abstract relevant problem space rather than all nodes. Since the search in the individual abstract spaces is smaller than that of the original search space, abstraction can reduce the savings of a rule.

8.1.4 Segre's adaptive inference system

Segre describes an ordered subproblem coordinated multi-strategy inference system (PROLOG interpreter) that combines EBL and subgoal caching [**?**]. The aim is to include other speed up techniques, such as antecedent ordering, dynamic abstraction, and domain theory revision. Success and failure caching is a strategy that caches previous (successful or failed) subgoals. A subgoal that is tested in the remainder of the proof can be retrieved from the cache instead of the system having to prove it again. Because of the search strategy used by the system (iterative deepening), it is guaranteed that subgoals are repeated (at least in the first iterations). There are two reasons for a failure (unable to prove the subgoal or not enough resources to prove it) and the adaptive inference system distinguishes between the two. In his experiments, Segre showed that combining subgoal caching and EBL reduced the search space over the individual strategies. This happened although the EBL component run into the utility problem and did not improve performance on its own. The reason for the improvement of the combined system is that EBL's poor performance was due to the generation of redundant subgoals and that subgoal caching prunes redundant searches. Segre refers to this phenomenon (one strategy alleviating another strategies's problem) as *speed up synergy.*

8.1.5 FLECS

Veloso and Stone describe the FLECS system, an unordered subproblem coordinated multi-strategy planner that combines eager and delayed operator ordering commitment strategies. Eager commitment results in totally ordered plans and is useful in domains with difficult operator selections. Delayed commitment produces partially ordered plans and is useful in domains with difficult goal interactions. FLECS combines the two strategies by maintaining a totally ordered plan head and a partially ordered plan tail.

FLECS uses a toggle to switch between subgoaling which leads to partially ordered plans and applying operators which leads to totally ordered plans. For a specific artificial domain, Veloso and Stone show that to solve problems without unnecessary search, a strategy that is able to switch between eager and delayed commitment is necessary. FLECS delayed commitment strategy only delays the orderings of operators, but not the instantiation of variables, and is thus not a traditional partial-order planner such as TWEAK with its variable constraints.

Veloso and Stone use heuristics to select which planning strategy to use. For example, the subgoal always before applying (SABA) heuristic results in delayed commitment. In DOLITTLE, a general operator would be associated with a new refinement type, that enables DOLITTLE to prefer partially ordered or totally ordered plans. A refinement of this type would make DOLITTLE change the value of the toggle. FLECS, however, is based on the state space search paradigm and is unable to represent and include other planning strategies such as case-based or abstraction-based planning.

8.1.6 The SOAR system

The goal of the SOAR project is the design of a plausible general cognitive architecture [**?**]. It uses chunking as its sole learning mechanism and much effort has been put into the demonstration of the different types of learning that SOAR is capable of. SOAR is a multi-strategy learner. DOLITTLE is intended as a problem solver component of a instructable system and is therefore only interested in speeding up planning and in extracting the necessary information from planning episodes. More recently, there has been work on multi-strategy planning in SOAR. Lee and Rosenbloom have investigated *monotonic multi-method planners*, that is planners that use a set of planning strategies that are organized in a linear order based on their coverage. The basic strategy is a strongest first method, similarly to the one used in this thesis. SOAR's notion of a planning strategy is limited to restrictions.

DoLittle's notion of a planning strategy is more general since it includes restructuring of the search space. DoLittle also does not require that the coverage of the planning strategies increases monotonically. The different planning strategies may cover different parts of the search space. The focus of the work in SOAR is to develop methods for switching from one specification to another. One approach is to learn conditions under which a planning strategy is guaranteed to fail and to skip those strategies. This saves SOAR the effort of trying them and finding out that they failed.

8.2 Macros

Korf describes the macro problem solver (MPS), a system that generates a macro problem space [?]. This approach is similar to abstraction, since in both cases is the original problem mapped into an abstract space. The main difference is that once a solution is found in the macro space, it is trivially refined into a ground solution. Korf identifies the *serial operator decomposability* as a sufficient condition for the effectiveness of this method. A domain is serially decomposable, if there is an ordering of the operators such that the effects of each operator only depend on the literals that precede it in the ordering. The disadvantage of this approach is the cost of finding a correct set of macro operators. Determining whether a domain is serially decomposable is PSPACE-complete [?].

Korf's MPS system is one that replaces the original operator set. In general, many systems attempt to speed up planning through the addition of macro operators to the general operator set. In all cases, the idea is to extract commonly used subsequences and to add them to the set of primitive operators. Two macro learners are described in chapter ??. DoLittle's macro learner is based on James' optimal tunneling heuristic.

8.3 Abstraction

DoLittle uses abstract operators to provide a planner with the ability to ignore low level details. Although abstraction based planning is viewed simply as another type of planning strategy in the multi-strategy planning framework, it has proven itself to be of critical importance in practical planning systems. All use some form of abstraction to overcome the tyranny of detail. Abstract operators have been used mainly in partial-order planners such as Noah [?], NonLin [?], and Molgen [?]. These systems do not learn abstract operators, but they must be provided by the user. DoLittle's abstraction learner generates new abstract operators from solutions based on the immediately backwards justified heuristic.

Wilkins' Sipe system uses different encodings of a domain at different levels of abstractions. The different encodings are hard to generate inductively. Wilkins also identified the problem of *hierarchical inaccuracy*, that is a planner may expand a part of the plan to a too detailed level before other parts of the plan are expanded at all. DoLittle (like many other planning systems, e.g. Alpine,Noah,AbStrips) overcomes this problem by expanding an abstract plan from left to right.

Knoblock defines *ordered monotonicity* as a sufficient condition for speed up using reduced abstraction based planning. DoLittle uses the same motivation, but applies it to the generation of operator-based abstraction hierarchies. Yiang shows that *ordered monotonicity* is equivalent to generating backwards justified plans only. DoLittle's abstraction learner extracts immediately backwards justified operator sequences to create abstractions. Whereas Alpine uses an analysis of all possible interactions in a domain to generate abstraction hierarchies, DoLittle's method is less conservative and only uses the observed interactions to generate abstraction hierarchies. Therefore, DoLittle will generate abstract operators in the blocksworld, whereas Alpine is unable to find an abstraction hierarchy. On the other hand, DoLittle may generate general operators that may decrease per-

formance. Since the same problem occurs in macro learning, DoLittle assumes that this case can be detected through the utility analysis and the offending abstract general operator can be removed from the operator set.

Anderson [?] and Baltes [?] describe similar methods to learn abstract operators from successful solutions. In both cases, sequences of operators are extracted from a solution and compared to other operators in the planner's operator set. If a general operator and the operator sequence share some effects, the differing preconditions and effects are abstracted and a new general operator is created that contains the operator sequence and the other general operator as refinements. The difference between the two systems is that Anderson's PLANNEREUS creates abstract operators for all pairs of operator sequences and operators that share some effects, whereas Baltes' abstraction learner is more constrained and only adds a new abstract operator if the post conditions of the pair are identical. For example, in the towers of Hanoi domain, the operator sequence Move-Small Peg1 Peg3, Move-Medium Peg1 Peg2 and the operator Move-Medium Peg1 Peg2 have identical postconditions. PLANNEREUS also creates abstract object hierarchies from observing possible variable instantiations of operators. DoLittle's abstraction learner is closely related to those systems, but uses a different selection heuristic (immediately backwards justified operator sequences).

8.4 Prodigy/EBL

Much of the design of DoLittle itself and its learning components is based on Minton's work on explanation based learning [?]. In particular, the determination of a learned operators' utility is based on an analysis similar to that of Minton. The difference is that in DoLittle general operators have different refinement costs, match costs, and match frequencies. The estimate of incremental utility was extended to include these factors. DoLittle uses general operators and Prodigy/EBL uses search control rules to represent

planning knowledge. Search control rules are more flexible than general operators in representing plan knowledge, because they may contain arbitrary lisp code. However, the representation of planning biases as search control rules would require complex code to be generated for each node, since a general operator determines a planning strategy instead of a decision at a single node. The refinements of a general operator are meta level comments that help DoLittle select a set of plan transformations. Therefore, a general operator affects the performance of the planner in the subproblems space as a whole. For example, if a refinement is of type SERIAL-SUBGOAL, DoLittle will not change literals that are contained in the preconditions of the general operator when searching for a solution. This constraint is checked for all nodes in the subproblem space. Compare this to a search control rule in Prodigy/EBL, that specifies to select a certain operator when trying to achieve some goal. As is shown by the reactive rule operator, DoLittle's general operators are powerful enough to represent selection of an operator.

One difference is that DoLittle uses induction instead of explanation based learning. This design decision was made because of the instructable system paradigm, which is also based on induction. Since DoLittle separates the planning bias learners from the planning system, it is possible to use other learning methods such as explanation based learning, supervised or unsupervised learning, or discourse analysis.

8.5 Dynamic biasing

As mentioned previously, dynamic biasing and multi-strategy learning are productive research areas in machine learning [?, ?, ?, ?]. Although these systems do not combine different problem solving strategies, they are relevant, since machine learning as well as planning can be seen as a search problem.

Provost introduces the *search the bias space* (SBS) framework for dynamic

biasing methods. Different dynamic biasing systems are compared to their search of the bias space. For example, a system may pick a bias at random. This method works well in cases where there are many strong correct biases. Exhaustively searching the space of possible biases is only possible if the sum of the costs is not too big. Therefore, this approach is only useful if there are few high cost biases.

If there are many expensive biases, the bias space must be structured to avoid high cost biases or the expensive biases must be removed from the bias space. Given the cost of a bias, the bias space can be structured so that biases are considered in increasing order of cost (iterative weakening). Provost shows that this bias selection method is optimal. The situation for subproblem coordinated multi-strategy planning is different since it has to take the probability of being able to reduce the problem further into consideration. This shows in the analysis in chapter **??**, which shows that the weakest strategy should be used first in subproblem coordinated multi-strategy planning.

In machine learning, the simple bias selection method works well. Provost attributes this to the fact that many of the problems in the machine learning database are simple [**?**].

The space of planning biases is different from machine learning biases, since there are few strong and correct biases. Provost suggest two methods to deal with the problem of few/no strong and correct biases. The first method is based on the idea of using the results of the previous search in selecting a new bias. Applied to the planning problem, this technique would use results from previous searches in creation/selection of a new bias. For example, DoLittle uses the results of previous searches by generating general operators and maintaining a set of important statistics about the different general operators for future problem solving. The second method suggested by Provost uses partial solutions to the learning problem. Planning supports this method, since DoLittle can combine solutions that were generated by

different planning strategies.

Barley uses the notion of a planning bias to describe the trade-off between the coverage and the efficiency of a planner [?]. Since planning is intractable in many domains, the achievement of a sound and complete domain independent planner is unachievable. The coverage and the speed of the planner must be balanced. A planning bias as used in Barley's work is a restriction on the planners algorithm. DoLittle's notion of a planning bias is more general since it includes restriction as well as restructuring and reordering of a search space.

Bhatnagar designed a single strategy planner FAILSAFE-2, that is based on the idea of iterative weakening [?]. The systems learns specific rules (so called heuristic censors) that cut out large parts of the search space from failure using EBL. The learned heuristic censors are overly general, which means they restrict the search space too much. If the heuristic censors prevent FAILSAFE-2 from finding a solution, the heuristic censor responsible for preventing FAILSAFE-2 from finding a solution is identified and its applicability conditions specialized so that in the future FAILSAFE-2 will be able to find a solution. DoLittle uses multiple strategies as opposed to a single strategy in FAILSAFE-2. DoLittle focuses on methods of combining different methods as opposed to finding applicability conditions.

Chapter 9

Conclusion

This chapter summarizes the contribution of the thesis and gives directions for future research.

9.1 Contributions

This section describes the contributions made by this research. The general intuition is (similarly to dynamic biasing in machine learning) to develop a system that can select and combine different planning strategies on a single problem.

The first contribution is the development of a theory of multi-strategy planning, which is discussed in chapter **??**. This framework is based on a model of abstraction which is an extension of previous models ([**?**, **?**] that distinguishes between problems that lead to a sufficient reduction and those that do not. The framework identifies important characteristics of a multi-strategy planner: problem versus subproblem coordinated, ordered versus unordered, exhaustive versus non exhaustive, and the decision procedure. The analysis shows that the expected cost is least for an unordered subproblem coordinated multi-strategy planner. Also based on this framework, it is shown that multi-strategy planning can under certain conditions exponen-

tially reduce the cost over that of a single strategy planner. The conditions under which this speed up occurs are sufficient conditions for improvement using a multi-strategy planner. The important conditions are the reduction probability and the reduction factors.

Secondly, this thesis compares different popular planning methods within the plan space search paradigm (chapter ??). It shows that different planning strategies can be described by their plan language and their set of plan transformations. This provides a usable definition of a planning strategy.

Thirdly, a set of requirements for an unordered subproblem coordinated multi-strategy planner are developed based on the multi-strategy planning framework and the comparison of different planning methods. The research identifies four features that are necessary for a practical multi-strategy planner:

- a representation language for different planning strategies
- a decision procedure to determine whether a planning strategy is appropriate
- a search control method that can emulate different planning strategies
- a domain description language sufficiently powerful to describe interesting domains

General operators are proposed as a solution to the first two problems. General operators describe applicability conditions as well as planning strategies. The applicability conditions imply a decision procedure. The search control method combines different planning strategies described as general operators and emulates their behavior. The domain description language is taken from PRODIGY and extended to support general operators.

Fourthly, three learning methods (cases, macros, and abstraction) are designed and implemented that learn popular planning strategies. The macro and case learners built on previous work, but the abstraction learner is a new

approach based on immediately backwards justified operators. The learners estimate the incremental utility of a new general operator based on a formula, which is an extension of Minton's work. The main difference is that DoLittle's general operators have different refinement costs, match costs, and match frequencies associated with them.

Lastly, the performance of a multi-strategy planner (DoLittle) against four single strategy planners (means-ends analysis, case-based, macro-based, and abstraction-based planning) is empirically evaluated. The evaluation shows that DoLittle is superior to the four single strategy planners in the blocksworld and the towers of Hanoi and is able to solve problems in at least one complex domain, the kitchen domain. The kitchen domain also shows that an unordered subproblem coordinated multi-strategy planner performs better than a problem coordinated one with an oracle.

9.2 Future work

This section discusses directions for future research. So far, the work has focused on presenting a model of multi-strategy planning and to provide an implementation that can be used as a testbed. The next goal is to better understand the interaction of different strategies and to provide methods for selecting a set of strategies. One advantage of a multi-strategy planner is that it can improve its coverage of a domain (the set of problems that can be solved efficiently) by combining planning strategies that are based on different planning biases. In this case, a set of planning strategies with different good sets is desirable. Another advantage of a multi-strategy planner is that it can use one strategy to alleviate problems of another strategy (Segre refers to this as speedup synergy).

As mentioned previously in section ??, DoLittle traded off the ability to use partial-order planning for the ability to use plan debugging efficiently. In the multi-strategy planning framework, partial-order planning is seen as

another type of planning strategy. This view is shared by Veloso and Stone, who propose a planning strategy that breaks a plan up into a totally ordered plan head and a partially ordered plan tail [**?**]. One part of future research is to add this planning strategy to DoLittle. Although FLECS does not support all features of *least commitment* planners such as Tweak, it seems to be a good compromise between the decrease due to a reduction of the search space and the increase due to a NP-hard truth criterion.

The uniform representation in DoLittle makes it an ideal testbed for the comparison of different planning strategies. This research may lead to a better understanding of the underlying planning biases for different planning strategies. This allows us to better predict the performance of different planners on new problems and to develop better methods to determine whether a planning strategy is suitable for a given domain.

So far, the main focus of DoLittle has been the methods of combining different sets of planning biases on a single problem. There are many more aspects of DoLittle that warrant further investigation. The two main ones are the acquiring of applicability conditions for planning biases, and the use of different learning strategies.

DoLittle provides a powerful language to describe the applicability conditions of planning methods. The language includes conjunction, disjunction, and negation of predicates and is based on the current state, the set of open goals, and the refinement structure. Future research will investigate methods for comparing the performance of a planning method to its expected performance and to change the applicability conditions to improve performance.

So far, DoLittle is a single-strategy learning system, since the planning bias learners described in chapter **??** only generate new general operators from successful solutions. There are many more problem solving events that may prove helpful in improving performance, such as failure, subgoal interaction, or expensive refinement. Furthermore, different learning paradigms can be used to extract the necessary information from the problem solving

event such as supervised learning, examples provided by the user, or learning by analogy.

9.3 Epilogue

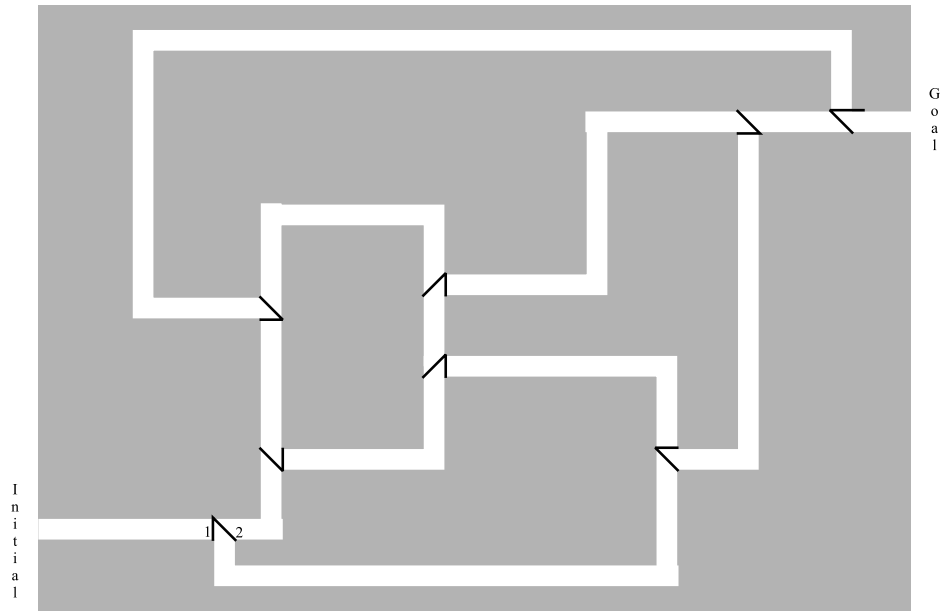
This section generalizes the results of the thesis to problem solving in general. The question is the apparent discrepancy between the complexity of planning and the ease with which humans can solve many of the problems that are hard for a computer. For example, no one would consider problems in the kitchen domain especially challenging.

The problem is that a planner simple forges ahead and attempts to solve the problem using its built-in strategy. Recently, there has been a lot of focus on the influence of the representation on the performance of the problem solver. This has lead to significant interest in problem reformulation [**?**]. The motivation behind problem reformulation is the realization that some problems require a radically different representation from the original one, so that they can be solved efficiently. Simple restrictions on the search space are often not sufficient.

This thesis adds another requirement to intelligent problem solvers: the ability to select and use different problem solving methods. For example, consider the maze problem in figure **??**. The problem is to find a path through the maze from the initial state to the goal. At some points in the maze, there are doors, which have two possible states (1 and 2 in the figure). A solution is a path and the setting for all doors on the solution path. The first maze problem can be solved quite easily. For example, the first door on the path must be in position 2, since otherwise the path is blocked. Continuing in this fashion, the user can easily find the path to the goal, since the status of all doors is determined. No search is necessary.

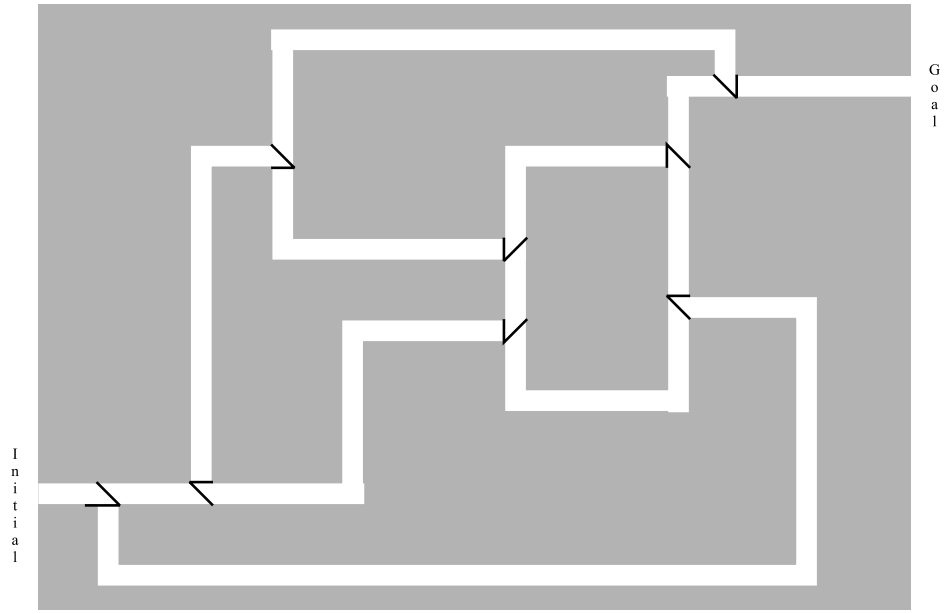
No consider the second maze problem in figure **??**. Here, the previous strategy does not lead to success. For example, the first door in the path

Figure 9.1: The first maze problem



can be in either position and thus provides no guidance for the rest of the search. However, if the strategy is changed to start from the goal and work towards the initial state, the problem can be solved without any problems. In fact, the second maze is simply the first maze rotated by 180 degrees.

Figure 9.2: The second maze problem



Clearly, a single planning strategy is not sufficient to solve all maze problems efficiently. A problem solver must have a variety of problem solving strategies available and be able to select an appropriate one for the current problem.

Other examples are logic and mathematical proofs. Over the centuries people have developed methods for proving statements in either logic or mathematics. Some well known ones are Aristotle's syllogism, enumeration, reductio ad absurdum, or (mathematical) induction. Also in computer science, there are a few well known programming methods, such as divide and

conquer, enumeration, recursion, and generalization. Often, the hard part is to find the correct problem solving strategy, finding the actual solution is then simple.

Appendix A

Implementation

This appendix describes DoLittle's implementation. It is intended mostly for people that are interested in running DoLittle themselves. It also contains details of the parallel implementation.

The implementation described in this thesis is the third version of a multi-strategy planner. Previous prototype systems were used to design and test particular parts of the system. First, an object oriented parallel system was implemented on a set of transputers. Secondly, DoLittle's search control mechanism was implemented and tested in common lisp on top of Prodigy.

The current implementation is written in NOWEB [**?**]. NOWEB is a literate programming system based on Knuth's WEB system [**?**]. It is easily ported to different operating systems, extensible, and powerful. Its main advantages is that it is independent of the underlying source language (C, Common Lisp, etc.). Using a literate programming system such as NOWEB makes it easy to generate documentation in a variety of formats from the source (e.g., LaTeX,HTML).

Since we are interested in a practical planning system, the system should be able to take advantage of current state of the art hardware. Therefore, the design of a practical planner should allow for efficient parallel or distributed implementation.

The most expensive part of planning is the search itself. Furthermore, learning new operators usually requires analysis of the plan derivation, the expanded part of the search space. Therefore, this thesis focuses on parallelization of the search and the operator learning.

The problem in distributing a planning search over a number of processors is that the search spaces associated with planning problems are generally highly irregular. Therefore, an a priori distribution is difficult.

This work uses a dynamic load balancing system to distribute the search. If a processor is finished with its part of the search space, it requests new work from neighboring processors.

General optimal dynamic load balancing itself is difficult to achieve. However, an efficient distribution is possible because of two additional assumptions. First, since the planner is solving similar problems, we assume that cost estimates for the refinement of an operator are accurate enough to support future load balancing. Note that these cost estimates are also used by the dynamic filter to check the usefulness of different planner transformations. Secondly, since most planning biases aim to reduce the solution length by possibly increasing the branching factor, the distribution algorithm distributes nodes in a breadth-first manner. For example, given a set of candidates for refinement, the load balancer distributes the candidates instead of trying to parallelize the search for a single candidate. This has the added advantage, that, in general, different refinements are mostly independent, so that communication between processors may be reduced.

Appendix B

The blocksworld domain

This version of the blocksworld domain is taken from the PRODIGY distribution.

All experiments were run on an IBM PC compatible computer with 16 MB of RAM and a 486-DX66 processor. The tables below compare DOLITTLE running in PRODIGY emulation mode (abstractions off, no general operators, relevant operator selection method) to two different multi-strategy planners (PC-MSP-O and DOLITTLE) and three single strategy planners (Cases, macros, abstractions), based on the learning methods described in chapter **??**.

B.1 Randomly generating problems in the blocksworld

The problems were created using an algorithm based on Minton's method [**?**]. The initial state is generated as follows:

- Between three and twelve blocks are generated at random.
- Each block is placed on the table with probability 1/3 or put on a randomly selected pile with probability 2/3.

- The robot is holding the last block with probability 1/3, otherwise the last block is handled exactly like the other blocks.

After creation of the initial state, a goal state is created by the same procedure as for the initial state, with the exception that the goal state contains the same blocks as the initial state.

All predicates that differ from the initial state are marked as a potential goal predicate. Predicates that are identical in the initial and the goal state are marked as potential goal predicates with probability 1/3.

The goal is created by randomly selecting between three and ten potential goal predicates and forming a conjunction of those predicates.

The main difference between this algorithm and Minton's method are that (a) Minton slowly increased the difficulty of the problem by increasing the number of nodes and the number of goals, and (b) Minton's algorithm contained an additional check to see whether the problem is indeed solvable. To show that DoLittle can break a problem up into useful subproblems, it was trained on a randomly generated set of constant difficulty instead of a slowly increasing one. The second difference is because Minton's system was a linear planner, and thus the problems must be checked to make sure that they can be solved in some linear order. Since Prodigy4 and DoLittle are non-linear planners, this check is not necessary.

B.2 Domain specification of the blocksworld

The representation of the blocksworld was taken from the Prodigy problem set. It contains four operators to avoid the need for conditional effects.

The original domain file that came with Prodigy4 did come with some inline control rules. Although, DoLittle can parse these rules, it can not evaluate the associated lisp code and these rules are thus ignored. Therefore, the rules were taken out of the domain specification. The variable constraint on operator Stack is also not used by DoLittle. Therefore, DoLittle

will generate STACK A A as an operator candidate. This however is ruled out by the goal loop detection algorithm is the next step.

```
;; This is the blocksworld according to the proposed 4.0 syntax

(create-problem-space 'blocksworld :current t)

(ptype-of OBJECT :top-type)

;(pinstance-of blocka object)
;(pinstance-of blockb object)
;(pinstance-of blockc object)

(OPERATOR
  PICK-UP
  (params <ob1>)
  (preconds
   ((<ob1> OBJECT))
   (and (clear <ob1>)
        (on-table <ob1>)
        (arm-empty)))
  (effects
      () ; no vars needed
      ((del (on-table <ob1>))
       (del (clear <ob1>))
       (del (arm-empty))
       (add (holding <ob1>)))))

(OPERATOR
  PUT-DOWN
  (params <ob>)
  (preconds
   ((<ob> OBJECT))
    (holding <ob>))
  (effects
        ()
        ((del (holding <ob>))
         (add (clear <ob>))
         (add (arm-empty))
```

```
         (add (on-table <ob>)))))

(OPERATOR
 STACK
 (params <ob> <underob>)
 (preconds
  ((<ob> Object)
   (<underob> (and OBJECT (diff <ob> <underob>))))
  (and (clear <underob>)
       (holding <ob>)))
 (effects
  ()
  ((del (holding <ob>))
   (del (clear <underob>))
   (add (arm-empty))
   (add (clear <ob> ))
   (add (on <ob> <underob>)))))

(OPERATOR
 UNSTACK
 (params <ob> <underob>)
 (preconds
  ((<ob> Object)
   (<underob> Object))
  (and (on <ob> <underob>)
       (clear <ob>)
       (arm-empty)))
 (effects
        ()
        ((del (on <ob> <underob>))
         (del (clear <ob>))
         (del (arm-empty))
         (add (holding <ob>))
         (add (clear <underob>)))))
```

The following table is an example problem for the blocksworld domain. It is the famous Sussman anomaly problem.

```
;; Sussman anomaly
```

```
(setf (current-problem)
      (create-problem
       (name sussman)
       (objects
        (blockA  blockb blockC object))
       (state
        (and (on-table blockA)
             (on-table blockB)
             (on blockC blockA)
             (clear blockB)
             (clear blockC)
             (arm-empty)))
       (goal
        (and (on blockA blockB)
             (on blockB blockC)))))
```

B.3 Empirical results in the blocksworld

This section summarizes the results of the experiments in the blocksworld. The table contains the following columns:

- **Prob** is the problem number.
- **Nodes** is the total number of nodes expanded during the search. A maximum limit of 15,000 nodes was used in all tests. Because the planner synchronizes only after certain node types, there is a chance that slightly more nodes are expanded.
- **Time** is the total CPU time used. There was a time limit of 600 CPU seconds imposed. However, in these experiments the node limit was the determining factor, since it was exceeded first.
- **Len** is the number of primitive operators in the solution. It is 0 if no solution exist or no solution was found within the resource limit.

Prob	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p1	18	0.08	1	6	0.03	1	6	0.01	1
p2	18	0.06	1	6	0.03	1	6	0.01	1
p3	15	0.06	3	12	0.04	3	13	0.04	3
p4	639	2.52	3	12	0.06	3	13	0.04	3
p5	413	1.96	2	22	0.04	2	22	0.04	2
p6	12462	76.19	2	10	0.04	2	10	0.03	2
p7	15	0.08	4	17	0.04	4	18	0.06	4
p8	9	0.03	2	9	0.01	2	10	0.02	2
p9	12	0.06	2	9	0.02	2	10	0.03	2
p10	117	0.59	4	17	0.05	4	18	0.07	4
p11	12	0.11	3	20	0.05	0	14	0.04	3
p12	720	3.67	3	13	0.05	3	13	0.05	3
p13	720	3.47	3	13	0.04	3	13	0.05	3
p14	17	0.06	2	9	0.01	2	9	0.03	2
p15	14	0.06	2	9	0.02	2	9	0.02	2
p16	11	0.03	2	9	0.02	2	9	0.02	2
p17	11	0.03	2	9	0.02	2	9	0.02	2
p18	12	0.06	1	6	0.00	1	6	0.02	1
p19	12	0.06	1	6	0.01	1	6	0.02	1
p20	11	0.03	2	9	0.02	2	9	0.01	2
p21	637	2.49	3	12	0.06	3	13	0.04	3
p22	414	2.10	2	22	0.03	2	22	0.04	2
p23	14	0.06	2	9	0.02	2	10	0.03	2
p24	7128	39.03	6	17	0.05	0	34	0.16	4
p25	3483	21.78	6	27	0.10	0	31	0.21	6
p26	156	0.67	0	16	0.06	6	16	0.05	6
p27	12	0.08	3	20	0.05	0	14	0.05	3
p28	15	0.08	3	21	0.11	3	14	0.05	3
p29	15001	101.78	0	13	0.07	3	13	0.06	3
p30	15001	110.91	0	21	0.11	5	42	0.24	9
p31	15001	88.96	0	14	0.07	0	14	0.06	6
p32	12	0.06	3	13	0.02	3	14	0.03	3
p33	15	0.11	4	17	0.07	4	18	0.07	4
p34	1239	6.97	7	23	0.13	7	31	0.15	7
p35	10299	51.83	5	15001	112.42	0	23	0.12	5
p36	12	0.08	3	22	0.05	0	14	0.05	3
p37	12	0.08	3	20	0.07	0	14	0.04	3
p38	15001	199.78	0	89	0.95	0	89	0.96	13
p39	15001	158.34	0	19	0.11	5	28	0.19	5
p40	333	1.29	0	48	0.17	2	55	0.32	9
p41	61	0.34	3	15	0.10	3	20	0.10	3
p42	15001	99.74	0	17	0.06	0	17	0.07	4
p43	40	0.17	3	13	0.08	3	13	0.07	3
p44	15	0.11	4	42	0.09	0	18	0.08	4
p45	15001	103.35	0	13	0.04	3	13	0.05	3
p46	17	0.08	4	17	0.06	4	41	0.19	6
p47	17	0.11	4	17	0.05	0	34	0.15	4
p48	11971	74.62	6	33	0.14	0	16	0.08	6
p49	7348	56.90	6	14	0.06	0	14	0.06	6
p50	29	0.11	4	10	0.03	2	30	0.10	4

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p51	1021	5.18	6	14	0.07	0	14	0.08	6
p52	716	3.81	4	10	0.04	2	13	0.08	3
p53	8467	59.22	4	17	0.12	0	17	0.13	4
p54	15001	182.87	0	28	0.19	5	55	0.61	6
p55	15001	117.85	0	33	0.27	7	41	0.48	9
p56	10299	52.56	5	15001	113.11	0	23	0.11	5
p57	10240	51.77	3	15001	112.22	0	23	0.14	5
p58	15001	151.40	0	16	0.21	0	41	0.33	6
p59	15001	97.61	0	27	0.14	0	33	0.13	6
p60	466	2.24	0	24	0.20	0	141	1.52	16
p61	1021	5.91	6	14	0.07	0	14	0.07	6
p62	5112	27.47	6	33	0.14	0	16	0.09	6
p63	1009	3.67	2	10	0.05	2	10	0.04	2
p64	756	3.70	5	6	0.05	1	134	0.48	5
p65	4931	26.99	4	17	0.07	4	17	0.06	4
p66	74	0.36	6	14	0.07	6	14	0.07	6
p67	15001	104.86	0	97	0.80	0	97	0.83	11
p68	15001	112.78	0	21	0.08	0	40	0.19	6
p69	4978	34.89	6	16	0.06	4	20	0.08	6
p70	87	0.31	3	16	0.07	3	20	0.07	6
p71	17	0.06	1	6	0.02	1	9	0.02	2
p72	69	0.31	4	17	0.05	0	17	0.05	4
p73	60	0.20	1	6	0.02	1	9	0.03	2
p74	93	0.48	7	37	0.24	0	15	0.08	7
p75	232	1.18	5	13	0.07	5	13	0.08	5
p76	15001	103.60	0	10	0.02	0	10	0.03	2
p77	14	0.08	2	9	0.01	2	10	0.03	2
p78	22	0.11	4	17	0.06	4	32	0.13	4
p79	60	0.22	1	6	0.02	1	6	0.02	1
p80	52	0.45	7	13	0.10	0	13	0.08	5
p81	15001	108.47	0	14	0.07	0	14	0.07	6
p82	912	5.18	6	52	0.24	0	52	0.23	6
p83	8982	58.44	4	20	0.07	0	20	0.09	4
p84	2038	10.05	0	21	0.18	0	22	0.16	2
p85	201	0.76	1	6	0.03	1	9	0.06	2
p86	17	0.06	1	6	0.01	1	9	0.02	2
p87	15001	135.07	0	31	0.16	0	32	0.18	4
p88	15001	111.36	0	39	0.32	0	39	0.40	8
p89	15001	98.76	0	65	0.42	0	73	0.44	11
p90	15001	106.82	0	33	0.27	0	38	0.35	7
p91	15001	163.97	0	53	1.46	0	55	1.98	11
p92	11	0.03	2	9	0.01	2	9	0.01	2
p93	15001	137.14	0	179	1.04	0	27	0.18	6
p94	15001	137.06	0	20	0.32	5	20	0.34	5
p95	31	0.25	5	19	0.10	5	24	0.11	5
p96	15001	124.80	0	179	1.27	0	62	0.92	10
p97	15001	103.18	0	17	0.07	4	17	0.07	4
p98	15001	198.49	0	24	0.47	6	25	0.16	6
p99	15001	177.16	0	13	0.05	5	37	0.16	5
p100	15001	132.30	0	34	0.61	0	34	0.60	8

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p101	15001	141.93	0	20	0.22	0	21	0.31	8
p102	12	0.08	3	20	0.07	0	14	0.05	3
p103	115	0.62	4	15	0.08	4	18	0.06	4
p104	15001	143.78	0	6	0.03	1	9	0.06	2
p105	12	0.67	3	12	0.27	3	13	0.27	3
p106	15001	120.46	0	26	0.32	0	27	0.40	6
p107	15001	134.20	0	31	0.64	0	31	0.71	3
p108	2782	13.89	3	12	0.05	3	12	0.04	3
p109	12	0.06	3	13	0.02	3	14	0.03	3
p110	15001	151.51	0	47	1.54	0	51	3.86	11
p111	15001	125.69	0	17	0.14	6	92	0.76	6
p112	12810	68.29	5	47	0.31	5	88	0.39	5
p113	15001	91.62	0	15001	66.45	0	15010	181.50	0
p114	15001	131.15	0	16	0.15	6	60	0.66	6
p115	15001	162.57	0	15	0.42	4	18	0.11	4
p116	856	5.35	6	16	0.07	6	39	0.16	6
p117	15001	247.04	0	15001	160.69	0	15001	138.70	0
p118	17	0.08	2	9	0.02	2	10	0.02	2
p119	15001	138.77	0	33	0.39	0	52	0.81	12
p120	15001	145.43	0	24	0.63	0	26	0.25	6
p121	247	1.74	8	57	0.60	0	19	0.20	8
p122	18	0.08	1	6	0.03	1	6	0.01	1
p123	18	0.06	1	6	0.03	1	6	0.01	1
p124	15001	139.02	0	42	0.26	0	28	0.21	2
p125	15	0.06	3	12	0.04	3	13	0.04	3
p126	639	2.52	3	12	0.06	3	13	0.04	3
p127	413	1.96	2	22	0.04	2	22	0.04	2
p128	12462	76.19	2	10	0.04	2	10	0.03	2
p129	15	0.08	4	17	0.04	4	18	0.06	4
p130	12	0.06	2	9	0.02	2	10	0.03	2
p131	117	0.59	4	17	0.05	4	18	0.07	4
p132	12	0.11	3	20	0.05	0	14	0.04	3
p133	720	3.67	3	13	0.05	3	13	0.05	3
p134	720	3.47	3	13	0.04	3	13	0.05	3
p135	17	0.06	2	9	0.01	2	9	0.03	2
p136	14	0.06	2	9	0.02	2	9	0.02	2
p137	11	0.03	2	9	0.02	2	9	0.02	2
p138	11	0.03	2	9	0.02	2	9	0.02	2
p139	12	0.06	1	6	0.00	1	6	0.02	1
p140	12	0.06	1	6	0.01	1	6	0.02	1
p141	11	0.03	2	9	0.02	2	9	0.01	2
p142	637	2.49	3	12	0.06	3	13	0.04	3
p143	414	2.10	2	22	0.03	2	22	0.04	2
p144	14	0.06	2	9	0.02	2	10	0.03	2
p145	7128	39.03	6	17	0.05	0	34	0.16	4
p146	3483	21.78	6	27	0.10	0	31	0.21	6
p147	156	0.67	0	16	0.06	6	16	0.05	6
p148	12	0.08	3	20	0.05	0	14	0.05	3
p149	15	0.08	3	21	0.11	3	14	0.05	3
p150	15001	101.78	0	13	0.07	3	13	0.06	3

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p151	15001	110.91	0	21	0.11	5	42	0.24	9
p152	15001	88.96	0	14	0.07	0	14	0.06	6
p153	12	0.06	3	13	0.02	3	14	0.03	3
p154	112	0.45	5	29	0.11	0	61	0.35	7
p155	15	0.11	4	17	0.07	4	18	0.07	4
p156	1239	6.97	7	23	0.13	7	31	0.15	7
p157	12	0.08	3	22	0.05	0	14	0.05	3
p158	12	0.08	3	20	0.07	0	14	0.04	3
p159	15001	199.78	0	89	0.95	0	89	0.96	13
p160	15001	158.34	0	19	0.11	5	28	0.19	5
p161	333	1.29	0	48	0.17	2	55	0.32	9
p162	61	0.34	3	15	0.10	3	20	0.10	3
p163	15001	99.74	0	17	0.06	0	17	0.07	4
p164	40	0.17	3	13	0.08	3	13	0.07	3
p165	15	0.11	4	42	0.09	0	18	0.08	4
p166	15001	103.35	0	13	0.04	3	13	0.05	3
p167	15001	112.28	0	6	0.03	0	94	0.32	5
p168	17	0.08	4	17	0.06	4	41	0.19	6
p169	17	0.11	4	17	0.05	0	34	0.15	4
p170	11971	74.62	6	33	0.14	0	16	0.08	6
p171	7348	56.90	6	14	0.06	0	14	0.06	6
p172	29	0.11	4	10	0.03	2	30	0.10	4
p173	1021	5.18	6	14	0.07	0	14	0.08	6
p174	716	3.81	4	10	0.04	2	13	0.08	3
p175	8467	59.22	4	17	0.12	0	17	0.13	4
p176	15001	182.87	0	28	0.19	5	55	0.61	6
p177	15001	117.85	0	33	0.27	7	41	0.48	9
p178	15001	134.62	0	15004	166.65	0	15001	176.96	0
p179	10240	51.77	3	15001	112.22	0	23	0.14	5
p180	15001	151.40	0	16	0.21	0	41	0.33	6
p181	15001	97.61	0	27	0.14	0	33	0.13	6
p182	466	2.24	0	24	0.20	2	141	1.52	16
p183	1021	5.91	6	14	0.07	0	14	0.07	6
p184	5112	27.47	6	33	0.14	0	16	0.09	6
p185	15001	114.10	0	15001	166.56	0	15001	114.14	0
p186	1009	3.67	2	10	0.05	2	10	0.04	2
p187	15001	103.68	0	15001	90.26	0	15001	94.46	0
p188	15001	120.37	0	15001	106.87	0	15001	99.28	0
p189	4931	26.99	4	17	0.07	4	17	0.06	4
p190	15001	106.04	0	15001	151.98	0	15001	81.65	0
p191	74	0.36	6	14	0.07	6	14	0.07	6
p192	15001	142.30	0	15001	97.14	0	15001	102.18	0
p193	15001	104.86	0	97	0.80	0	97	0.83	11
p194	15001	105.50	0	15002	116.95	0	15001	151.76	0
p195	15001	112.78	0	21	0.08	0	40	0.19	6
p196	4978	34.89	6	16	0.06	4	20	0.08	6
p197	87	0.31	3	16	0.07	3	20	0.07	6
p198	17	0.06	1	6	0.02	1	9	0.02	2
p199	69	0.31	4	17	0.05	0	17	0.05	4
p200	60	0.20	1	6	0.02	1	9	0.03	2

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p201	93	0.48	7	37	0.24	0	15	0.08	7
p202	232	1.18	5	13	0.07	5	13	0.08	5
p203	15001	103.60	0	10	0.02	0	10	0.03	2
p204	14	0.08	2	9	0.01	2	10	0.03	2
p205	22	0.11	4	17	0.06	4	32	0.13	4
p206	60	0.22	1	6	0.02	1	6	0.02	1
p207	52	0.45	7	13	0.10	0	13	0.08	5
p208	15001	108.47	0	14	0.07	0	14	0.07	6
p209	15001	102.37	0	15001	103.25	0	15001	133.19	0
p210	912	5.18	6	52	0.24	0	52	0.23	6
p211	8982	58.44	4	20	0.07	0	20	0.09	4
p212	2038	10.05	0	21	0.18	0	22	0.16	2
p213	201	0.76	1	6	0.03	1	9	0.06	2
p214	17	0.06	1	6	0.01	1	9	0.02	2
p215	15001	135.07	0	31	0.16	0	32	0.18	4
p216	15001	111.36	0	39	0.32	0	39	0.40	8
p217	15001	98.76	0	65	0.42	0	73	0.44	11
p218	15001	106.82	0	33	0.27	0	38	0.35	7
p219	15001	163.97	0	53	1.46	0	55	1.98	11
p220	11	0.03	2	9	0.01	2	9	0.01	2
p221	15001	137.14	0	179	1.04	0	27	0.18	6
p222	15001	137.06	0	20	0.32	5	20	0.34	5
p223	31	0.25	5	19	0.10	5	24	0.11	5
p224	15001	124.80	0	179	1.27	0	62	0.92	10
p225	15001	103.18	0	17	0.07	4	17	0.07	4
p226	15001	198.49	0	24	0.47	6	25	0.16	6
p227	15001	177.16	0	13	0.05	5	37	0.16	5
p228	15001	132.30	0	34	0.61	0	34	0.60	8
p229	15001	141.93	0	20	0.22	0	21	0.31	8
p230	12	0.08	3	20	0.07	0	14	0.05	3
p231	115	0.62	4	15	0.08	4	18	0.06	4
p232	15001	143.78	0	6	0.03	1	9	0.06	2
p233	12	0.67	3	12	0.27	3	13	0.27	3
p234	15001	120.46	0	26	0.32	0	27	0.40	6
p235	15001	134.20	0	31	0.64	0	31	0.71	3
p236	2782	13.89	3	12	0.05	3	12	0.04	3
p237	12	0.06	3	13	0.02	3	14	0.03	3
p238	15001	151.51	0	47	1.54	0	51	3.86	11
p239	15001	125.69	0	17	0.14	6	92	0.76	6
p240	12810	68.29	5	47	0.31	5	88	0.39	5
p241	15001	119.20	0	103	1.30	0	103	1.41	11
p242	15001	91.62	0	15001	66.45	0	15010	181.50	0
p243	15001	131.15	0	16	0.15	6	60	0.66	6
p244	15001	162.57	0	15	0.42	4	18	0.11	4
p245	856	5.35	6	16	0.07	6	39	0.16	6
p246	15001	247.04	0	15001	160.69	0	15001	138.70	0
p247	17	0.08	2	9	0.02	2	10	0.02	2
p248	15001	138.77	0	33	0.39	0	52	0.81	12
p249	15001	145.43	0	24	0.63	0	26	0.25	6
p250	247	1.74	8	57	0.60	0	19	0.20	8

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p1	6	0.02	1	6	0.03	1	16	0.02	1
p2	6	0.01	1	6	0.03	1	16	0.02	1
p3	12	0.04	3	13	0.03	3	22	0.05	3
p4	12	0.06	3	13	0.03	3	31	0.05	3
p5	34	0.07	2	22	0.04	2	35	0.07	2
p6	10	0.04	2	10	0.04	2	24	0.05	2
p7	17	0.06	4	17	0.04	4	142	0.65	4
p8	9	0.03	2	9	0.01	2	12	0.05	2
p9	9	0.03	2	9	0.02	2	12	0.04	2
p10	17	0.05	4	17	0.05	4	292	1.55	4
p11	15001	114.30	0	15001	146.32	0	20	0.05	3
p12	19	0.08	3	13	0.05	3	35	0.07	3
p13	19	0.07	3	13	0.04	3	35	0.07	3
p14	9	0.03	2	9	0.01	2	12	0.04	2
p15	9	0.03	2	9	0.02	2	11	0.04	2
p16	9	0.02	2	9	0.02	2	12	0.04	2
p17	9	0.03	2	9	0.02	2	13	0.05	2
p18	6	0.03	1	6	0.00	1	16	0.04	1
p19	6	0.02	1	6	0.01	1	16	0.02	1
p20	9	0.02	2	9	0.02	2	12	0.02	2
p21	12	0.06	3	13	0.04	3	31	0.05	3
p22	34	0.06	2	22	0.03	2	35	0.07	2
p23	9	0.03	2	9	0.02	2	12	0.04	2
p24	17	0.05	4	15001	114.36	0	15140	193.46	0
p25	31	0.16	6	27	0.10	6	15127	199.19	0
p26	16	0.06	6	48	0.18	6	77	0.23	6
p27	15001	115.77	0	15001	151.22	0	20	0.05	3
p28	21	0.11	5	21	0.11	5	22	0.09	3
p29	19	0.08	3	13	0.07	3	46	0.13	3
p30	35	0.20	5	21	0.11	5	144	0.38	9
p31	14	0.07	6	58	0.20	6	15113	223.13	0
p32	13	0.02	3	13	0.02	3	22	0.04	3
p33	17	0.06	4	17	0.07	4	442	2.74	4
p34	23	0.13	7	42	0.23	7	90	0.27	7
p35	15001	112.42	0	15002	134.53	0	15121	187.34	0
p36	15001	75.09	0	15013	150.04	0	22	0.05	3
p37	15001	81.17	0	15001	177.96	0	20	0.07	3
p38	89	0.95	13	15010	126.03	0	15159	162.02	0
p39	19	0.11	5	21	0.09	5	90	0.34	5
p40	62	0.31	2	48	0.17	5	1735	6.71	9
p41	15	0.10	5	20	0.10	3	59	0.13	3
p42	17	0.06	4	15001	119.55	0	43	0.09	4
p43	13	0.07	3	13	0.08	3	42	0.07	3
p44	15013	115.96	0	15005	128.78	0	42	0.09	4
p45	69	0.22	5	13	0.04	3	23	0.05	3
p46	17	0.06	4	42	0.15	4	25	0.09	6
p47	17	0.05	4	15001	114.59	0	25	0.09	4
p48	15001	128.67	0	33	0.14	8	15113	154.01	0
p49	14	0.06	6	58	0.23	6	15113	225.43	0
p50	10	0.03	2	40	0.10	4	16	0.04	4

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p51	14	0.07	6	15001	154.24	0	70	0.18	6
p52	10	0.04	2	13	0.05	3	23	0.04	3
p53	17	0.12	4	15003	150.95	0	56	0.13	4
p54	28	0.19	9	54	0.27	5	92	0.25	6
p55	41	0.47	9	33	0.27	7	133	0.45	9
p56	15001	113.11	0	15002	134.28	0	15121	194.63	0
p57	15001	112.22	0	15002	136.61	0	15119	195.30	0
p58	16	0.21	4	15001	186.32	0	15199	224.30	0
p59	15001	134.75	0	27	0.14	6	64	0.18	6
p60	86	0.52	3	24	0.20	2	10015	52.22	0
p61	14	0.07	6	15001	155.03	0	70	0.22	6
p62	15001	121.53	0	33	0.14	8	15121	165.26	0
p63	10	0.04	2	10	0.05	2	946	4.01	2
p64	6	0.05	1	59	0.29	5	10	0.04	5
p65	17	0.12	4	17	0.07	4	49	0.11	4
p66	14	0.07	6	128	0.43	6	59	0.18	6
p67	15001	150.51	0	97	0.80	11	203	0.90	11
p68	21	0.08	4	15001	146.90	0	42	0.11	6
p69	16	0.06	6	36	0.12	4	60	0.16	6
p70	16	0.07	6	28	0.07	3	46	0.09	6
p71	6	0.02	1	9	0.02	2	9	0.04	2
p72	17	0.05	4	15007	118.58	0	25	0.09	4
p73	6	0.02	1	9	0.02	2	11	0.02	2
p74	15001	110.16	0	37	0.24	7	15126	179.95	0
p75	13	0.07	5	36	0.28	5	1810	7.65	5
p76	10	0.02	2	15008	108.83	0	16	0.04	2
p77	9	0.02	2	9	0.01	2	12	0.04	2
p78	17	0.06	4	39	0.14	4	26	0.09	4
p79	6	0.01	1	6	0.02	1	11	0.02	1
p80	13	0.10	5	62	0.47	9	15046	72.45	0
p81	14	0.07	6	15001	164.21	0	75	0.23	6
p82	15003	163.58	0	52	0.24	6	72	0.16	6
p83	20	0.07	4	15013	136.95	0	27	0.09	4
p84	23	0.16	3	21	0.18	2	15029	75.35	0
p85	6	0.03	1	9	0.05	2	17	0.05	2
p86	6	0.01	1	9	0.03	2	9	0.00	2
p87	15001	98.43	0	31	0.16	4	15155	197.17	0
p88	39	0.32	8	15015	148.63	0	15202	201.55	0
p89	65	0.42	11	421	3.80	9	15170	203.33	0
p90	33	0.27	9	38	0.29	7	15119	125.75	0
p91	55	1.97	11	53	1.46	11	15543	259.51	0
p92	9	0.02	2	9	0.01	2	11	0.02	2
p93	15001	125.85	0	179	1.04	10	284	1.28	6
p94	20	0.32	5	22	0.28	5	128	0.40	5
p95	19	0.10	5	21	0.10	5	52	0.11	5
p96	15002	150.75	0	179	1.27	13	3791	20.12	10
p97	19	0.11	4	17	0.07	4	58	0.14	4
p98	24	0.47	6	26	0.18	6	145	0.58	6
p99	13	0.05	5	34	0.14	5	81	0.20	5
p100	34	0.61	8	191	1.24	12	15208	168.89	0

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p101	15001	202.92	0	20	0.22	3	15048	114.88	0
p102	15001	117.51	0	15001	150.33	0	20	0.07	3
p103	15	0.08	4	17	0.04	4	36	0.13	4
p104	6	0.03	1	9	0.06	2	19	0.07	2
p105	12	0.26	3	12	0.27	3	131	0.43	3
p106	27	0.92	8	26	0.32	6	15048	114.88	0
p107	31	0.64	3	15001	160.69	0	15112	94.73	0
p108	883	3.39	8	12	0.05	3	34	0.09	3
p109	13	0.03	3	13	0.02	3	17	0.05	3
p110	51	3.76	11	47	1.54	5	15048	114.88	0
p111	17	0.14	6	34	0.43	6	370	1.89	6
p112	47	0.31	7	88	0.40	5	205	0.61	5
p113	15001	66.45	0	15025	88.54	0	15023	81.81	0
p114	16	0.15	6	57	0.60	6	229	0.94	6
p115	15	0.42	4	17	0.10	4	111	0.36	4
p116	16	0.07	6	36	0.14	6	68	0.20	6
p117	15001	137.17	0	15001	160.69	0	15048	114.88	0
p118	9	0.04	2	9	0.02	2	14	0.07	2
p119	52	0.85	12	33	0.39	4	15130	135.00	0
p120	24	0.63	6	26	0.19	6	15028	84.94	0
p121	57	0.60	8	63	0.49	8	15229	216.61	0
p122	6	0.02	1	6	0.03	1	16	0.02	1
p123	6	0.01	1	6	0.03	1	16	0.02	1
p124	45	0.45	10	42	0.26	10	15185	215.62	0
p125	12	0.04	3	13	0.03	3	22	0.05	3
p126	12	0.06	3	13	0.03	3	31	0.05	3
p127	34	0.07	2	22	0.04	2	35	0.07	2
p128	10	0.04	2	10	0.04	2	24	0.05	2
p129	17	0.06	4	17	0.04	4	142	0.65	4
p130	9	0.03	2	9	0.02	2	12	0.04	2
p131	17	0.05	4	17	0.05	4	292	1.55	4
p132	15001	114.30	0	15001	146.32	0	20	0.05	3
p133	19	0.08	3	13	0.05	3	35	0.07	3
p134	19	0.07	3	13	0.04	3	35	0.07	3
p135	9	0.03	2	9	0.01	2	12	0.04	2
p136	9	0.03	2	9	0.02	2	11	0.04	2
p137	9	0.02	2	9	0.02	2	12	0.04	2
p138	9	0.03	2	9	0.02	2	13	0.05	2
p139	6	0.03	1	6	0.00	1	16	0.04	1
p140	6	0.02	1	6	0.01	1	16	0.02	1
p141	9	0.02	2	9	0.02	2	12	0.02	2
p142	12	0.06	3	13	0.04	3	31	0.05	3
p143	34	0.06	2	22	0.03	2	35	0.07	2
p144	9	0.03	2	9	0.02	2	12	0.04	2
p145	17	0.05	4	15001	114.36	0	15140	193.46	0
p146	31	0.16	6	27	0.10	6	15127	199.19	0
p147	16	0.06	6	48	0.18	6	77	0.23	6
p148	15001	115.77	0	15001	151.22	0	20	0.05	3
p149	21	0.11	5	21	0.11	5	22	0.09	3
p150	19	0.08	3	13	0.07	3	46	0.13	3

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p151	35	0.20	5	21	0.11	5	144	0.38	9
p152	14	0.07	6	58	0.20	6	15113	223.13	0
p153	13	0.02	3	13	0.02	3	22	0.04	3
p154	15001	100.82	0	29	0.11	7	15028	130.73	0
p155	17	0.06	4	17	0.07	4	442	2.74	4
p156	23	0.13	7	42	0.23	7	90	0.27	7
p157	15001	75.09	0	15013	150.04	0	22	0.05	3
p158	15001	81.17	0	15001	177.96	0	20	0.07	3
p159	89	0.95	13	15010	126.03	0	15159	162.02	0
p160	19	0.11	5	21	0.09	5	90	0.34	5
p161	62	0.31	2	48	0.17	5	1735	6.71	9
p162	15	0.10	5	20	0.10	3	59	0.13	3
p163	17	0.06	4	15001	119.55	0	43	0.09	4
p164	13	0.07	3	13	0.08	3	42	0.07	3
p165	15013	115.96	0	15005	128.78	0	42	0.09	4
p166	69	0.22	5	13	0.04	3	23	0.05	3
p167	6	0.03	1	15010	116.76	0	9	0.04	5
p168	17	0.06	4	42	0.15	4	25	0.09	6
p169	17	0.05	4	15001	114.59	0	25	0.09	4
p170	15001	128.67	0	33	0.14	8	15113	154.01	0
p171	14	0.06	6	58	0.23	6	15113	225.43	0
p172	10	0.03	2	40	0.10	4	16	0.04	4
p173	14	0.07	6	15001	154.24	0	70	0.18	6
p174	10	0.04	2	13	0.05	3	23	0.04	3
p175	17	0.12	4	15003	150.95	0	56	0.13	4
p176	28	0.19	9	54	0.27	5	92	0.25	6
p177	41	0.47	9	33	0.27	7	133	0.45	9
p178	15004	166.65	0	15005	166.80	0	15277	150.28	0
p179	15001	112.22	0	15002	136.61	0	15119	195.30	0
p180	16	0.21	4	15001	186.32	0	15199	224.30	0
p181	15001	134.75	0	27	0.14	6	64	0.18	6
p182	86	0.52	3	24	0.20	2	10015	52.22	16
p183	14	0.07	6	15001	155.03	0	70	0.22	6
p184	15001	121.53	0	33	0.14	8	15121	165.26	0
p185	15001	112.58	0	15001	166.56	0	15086	83.02	0
p186	10	0.04	2	10	0.05	2	946	4.01	2
p187	15001	89.21	0	15001	90.26	0	15065	100.87	0
p188	15001	96.87	0	15001	106.87	0	15063	100.12	0
p189	17	0.12	4	17	0.07	4	49	0.11	4
p190	15001	80.93	0	15001	151.98	0	15086	106.58	0
p191	14	0.07	6	128	0.43	6	59	0.18	6
p192	15001	91.45	0	15001	97.14	0	15084	111.69	0
p193	15001	150.51	0	97	0.80	11	203	0.90	11
p194	15038	108.15	0	15002	116.95	0	15224	205.51	0
p195	21	0.08	4	15001	146.90	0	42	0.11	6
p196	16	0.06	6	36	0.12	4	60	0.16	6
p197	16	0.07	6	28	0.07	3	46	0.09	6
p198	6	0.02	1	9	0.02	2	9	0.04	2
p199	17	0.05	4	15007	118.58	0	25	0.09	4
p200	6	0.02	1	9	0.02	2	11	0.02	2

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p201	15001	110.16	0	37	0.24	7	15126	179.95	0
p202	13	0.07	5	36	0.28	5	1810	7.65	5
p203	10	0.02	2	15008	108.83	0	16	0.04	2
p204	9	0.02	2	9	0.01	2	12	0.04	2
p205	17	0.06	4	39	0.14	4	26	0.09	4
p206	6	0.01	1	6	0.02	1	11	0.02	1
p207	13	0.10	5	62	0.47	9	15046	72.45	0
p208	14	0.07	6	15001	164.21	0	75	0.23	6
p209	15001	103.25	0	15003	151.54	0	15101	177.59	0
p210	15003	163.58	0	52	0.24	6	72	0.16	6
p211	20	0.07	4	15013	136.95	0	27	0.09	4
p212	23	0.16	3	21	0.18	2	15029	75.35	0
p213	6	0.03	1	9	0.05	2	17	0.05	2
p214	6	0.01	1	9	0.03	2	9	0.00	2
p215	15001	98.43	0	31	0.16	4	15155	197.17	0
p216	39	0.32	8	15015	148.63	0	15202	201.55	0
p217	65	0.42	11	421	3.80	9	15170	203.33	0
p218	33	0.27	9	38	0.29	7	15119	125.75	0
p219	55	1.97	11	53	1.46	11	15543	259.51	0
p220	9	0.02	2	9	0.01	2	11	0.02	2
p221	15001	125.85	0	179	1.04	10	284	1.28	6
p222	20	0.32	5	22	0.28	5	128	0.40	5
p223	19	0.10	5	21	0.10	5	52	0.11	5
p224	15002	150.75	0	179	1.27	13	3791	20.12	10
p225	19	0.11	4	17	0.07	4	58	0.14	4
p226	24	0.47	6	26	0.18	6	145	0.58	6
p227	13	0.05	5	34	0.14	5	81	0.20	5
p228	34	0.61	8	191	1.24	12	15208	168.89	0
p229	15001	202.92	0	20	0.22	3	15048	114.88	0
p230	15001	117.51	0	15001	150.33	0	20	0.07	3
p231	15	0.08	4	17	0.04	4	36	0.13	4
p232	6	0.03	1	9	0.06	2	19	0.07	2
p233	12	0.26	3	12	0.27	3	131	0.43	3
p234	27	0.92	8	26	0.32	6	15048	114.88	0
p235	31	0.64	3	15001	160.69	0	15112	94.73	0
p236	883	3.39	8	12	0.05	3	34	0.09	3
p237	13	0.03	3	13	0.02	3	17	0.05	3
p238	51	3.76	11	47	1.54	5	15048	114.88	0
p239	17	0.14	6	34	0.43	6	370	1.89	6
p240	47	0.31	7	88	0.40	5	205	0.61	5
p241	103	1.30	11	15001	142.29	0	15110	104.89	0
p242	15001	66.45	0	15025	88.54	0	15023	81.81	0
p243	16	0.15	6	57	0.60	6	229	0.94	6
p244	15	0.42	4	17	0.10	4	111	0.36	4
p245	16	0.07	6	36	0.14	6	68	0.20	6
p246	15001	137.17	0	15001	160.69	0	15048	114.88	0
p247	9	0.04	2	9	0.02	2	14	0.07	2
p248	52	0.85	12	33	0.39	4	15130	135.00	0
p249	24	0.63	6	26	0.19	6	15028	84.94	0
p250	57	0.60	8	63	0.49	8	15229	216.61	0

Appendix C

The towers of Hanoi domain

This version of the blocksworld is taken from the PRODIGY distribution, but was extended to four disks instead of three disks.

All experiments were run on an IBM PC compatible computer with 16 MB of RAM and a 486-DX66 processor. The tables below compare DOLITTLE running in PRODIGY emulation mode (abstractions off, no general operators, relevant operator selection method) to two different multi-strategy planners (PC-MSP-O and DOLITTLE) and three single strategy planners (Cases, macros, abstractions), based on the learning methods described in chapter ??.

C.1 Randomly generating problems in the towers of Hanoi

This section describes the method used to create a random problem in the towers of Hanoi domain. First, an initial state is created by placing disks on the three pegs at random. Since a state in the towers of Hanoi domain is only legal, if no bigger disk is on top of a smaller disk, this fixes the order of the disks on the pegs. Then, a goal state is created using the same algorithm

and converted into a goal conjunction. Then literals that differ between the inital state and the goal state are collected and used to generate the goal statement.

C.2 Domain specification of the towers of Hanoi domain

The representation of the towers of Hanoi domain is taken from the PRODIGY distribution and has been used by many other researchers.

```
;; The towers of Hanoi domain

(create-problem-space 'dl-abs-hanoi :current t)

(ptype-of Peg  :top-type)
(ptype-of Disk :top-type)
(pinstance-of disk1 Disk)
(pinstance-of disk2 Disk)
(pinstance-of disk3 Disk)
(pinstance-of disk4 Disk)

(OPERATOR
 Move-Small-Disk
 (params <from> <to>)
 (preconds
  ((<from> Peg)
   (<to> Peg))
  (and (on disk1 <from>)
       (~ (on disk1 <to>))))
 (effects
  ()
  ((del (on disk1 <from>))
   (add (on disk1 <to>)))))

(OPERATOR
 Move-Medium-Disk
```

```
 (params <from> <to>)
 (preconds
  ((<from> Peg)
   (<to> Peg))
  (and (on disk2 <from>)
       (~ (on disk1 <from>))
       (~ (on disk1 <to>))))
 (effects
  ()
  ((del (on disk2 <from>))
   (add (on disk2 <to>)))))

(OPERATOR
 Move-Large-Disk
 (params <from> <to>)
 (preconds
  ((<from> Peg)
   (<to> Peg))
  (and (on disk3 <from>)
       (~ (on disk1 <from>))
       (~ (on disk2 <from>))
       (~ (on disk1 <to>))
       (~ (on disk2 <to>))))
 (effects
  ()
  ((del (on disk3 <from>))
   (add (on disk3 <to>)))))

(OPERATOR
 Move-Huge-Disk
 (params <from> <to>)
 (preconds
  ((<from> Peg)
   (<to> Peg))
  (and (on disk4 <from>)
       (~ (on disk1 <from>))
       (~ (on disk2 <from>))
       (~ (on disk3 <from>))
       (~ (on disk1 <to>))
       (~ (on disk2 <to>))
```

```
       (~ (on disk3 <to>))))
 (effects
  ()
  ((del (on disk4 <from>))
   (add (on disk4 <to>)))))
```

The following is an example problem specification. The problem is to move the three smaller disks from PEG2 to PEG1 and the huge disk from PEG3 to PEG2.

```
;;; Problem automatically generated by DoLittle
(setf (current-problem)
      (create-problem
       (name p20)
       (objects
        (objects-are peg1 peg2 peg3 Peg))
       (state
        (and (on disk1 peg2)(on disk2 peg2)
             (on disk3 peg2)(on disk4 peg3)))
       (goal (and (on disk1 peg1)(on disk2 peg1)
                  (on disk3 peg1)(on disk4 peg2)))))
```

C.3 Empirical results in the towers of Hanoi

This section summarizes the results of the experiments in the towers of Hanoi domain in a table. The table contains the following columns:

- **Prob** is the problem number.
- **Nodes** is the total number of nodes expanded during the search. A maximum limit of 15,000 nodes was used in all tests. Because the planner synchronizes only after certain node types, there is a chance that slightly more nodes are expanded.

- **Time** is the total CPU time used. There was a time limit of 600 CPU seconds imposed. However, in these experiments the node limit was the determining factor, since it was exceeded first.
- **Len** is the number of primitive operators in the solution. It is 0 if no solution exist or no solution was found within the resource limit.

Prob	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p1	12	0.01	2	6	0.01	1	6	0.01	1
p2	3	0.01	0	3	0.00	0	3	0	0
p3	15001	125.7	0	6	0.01	1	9	0.02	1
p4	34	0.06	7	17	0.05	4	17	0.05	4
p5	15001	134.2	0	22	0.15	0	22	0.15	14
p6	3	0	0	3	0.00	0	3	0	0
p7	15001	160.99	0	15001	96.38	0	15001	96.49	0
p8	15001	75.38	0	18	0.08	0	18	0.08	10
p9	944	3.25	16	15001	77.78	0	15001	78.64	0
p10	15001	130.66	0	137	1.20	0	15001	119.12	0
p11	6	0.01	1	6	0.01	1	6	0.02	1
p12	15001	213.26	0	15001	151.7	0	15001	152.3	0
p13	15001	130.73	0	20	0.09	0	20	0.09	4
p14	11	0.01	2	6	0.01	1	6	0.01	1
p15	15001	126.02	0	27	0.10	5	27	0.08	5
p16	7	0.01	1	6	0.01	1	6	0.01	1
p17	15001	78.99	0	15001	150.2	0	15001	154.82	0
p18	15001	220.25	0	45	0.28	0	45	0.28	11
p19	3	0	0	3	0.00	0	3	0.01	0
p20	34	0.05	7	17	0.05	4	17	0.06	4
p21	10	0.01	2	10	0.03	2	10	0.03	2
p22	24	0.04	5	24	0.06	4	24	0.07	4
p23	17	0.02	4	19	0.06	4	19	0.06	4
p24	197	0.43	7	10	0.03	2	10	0.02	2
p25	3	0.01	0	3	0.00	0	3	0	0
p26	15001	96.32	0	15001	81.71	0	15001	81.51	0
p27	15001	141.93	0	30	0.17	0	30	0.18	6
p28	25	0.04	5	19	0.07	4	19	0.05	4
p29	15001	117.36	0	25	0.12	0	25	0.1	5
p30	1968	5.11	7	6	0.01	1	6	0	1
p31	15001	48.75	0	13	0.03	3	13	0.03	3
p32	3	0	0	3	0.00	0	3	0	0
p33	10	0.02	2	10	0.03	2	10	0.03	2
p34	24	0.03	5	19	0.05	4	19	0.06	4
p35	24	0.04	5	21	0.06	4	21	0.05	4
p36	6	0.01	1	6	0.01	1	6	0.01	1
p37	7	0	1	6	0.01	1	6	0.01	1
p38	1968	5.51	7	6	0.01	1	6	0.01	1
p39	11	0.02	2	6	0.01	1	6	0.01	1
p40	15001	160.37	0	15001	153.5	0	44	0.28	14
p41	6	0.01	1	6	0.01	1	6	0.01	1
p42	36	0.07	8	10	0.02	2	10	0.03	2
p43	3	0	0	3	0.01	0	3	0	0
p44	10	0.02	2	6	0.01	1	6	0.01	1
p45	17	0.02	4	19	0.05	4	19	0.07	4
p46	982	3.55	17	15001	69.27	0	15001	71.18	0
p47	3	0	0	3	0.01	0	3	0	0
p48	15	0.02	3	10	0.01	2	10	0.03	2
p49	35	0.04	3	10	0.03	2	32	0.13	3
p50	11	0.02	2	6	0.00	1	6	0.02	1

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p51	15001	110.11	0	10	0.04	2	32	0.13	3
p52	3	0.01	0	3	0.01	0	3	0	0
p53	10	0.01	2	6	0.01	1	6	0.01	1
p54	95	0.24	11	37	0.17	0	37	0.17	7
p55	7	0.02	1	6	0.01	1	6	0.01	1
p56	7	0.01	1	6	0.01	1	6	0.02	1
p57	6	0.01	1	6	0.01	1	6	0.01	1
p58	15001	59.39	0	15001	82.89	0	15001	82.48	0
p59	15001	78.42	0	15001	84.58	0	15001	86.01	0
p60	15001	93.24	0	15001	104.8	0	15001	104.74	0
p61	6	0	1	6	0.01	1	6	0	1
p62	13	0.02	3	10	0.03	2	10	0.03	2
p63	116	0.24	9	22	0.06	0	22	0.09	6
p64	943	3.18	16	53	0.26	0	53	0.28	9
p65	15001	121.36	0	22	0.13	0	15001	106.02	0
p66	35	0.05	3	10	0.03	2	10	0.03	2
p67	11	0.01	2	6	0.01	1	6	0.01	1
p68	15001	88.37	0	33	0.15	0	33	0.14	6
p69	15001	41.39	0	19	0.04	3	19	0.05	3
p70	10	0.02	2	6	0.02	1	6	0.01	1
p71	15001	76.78	0	19	0.10	7	38	0.19	7
p72	15001	44.14	0	20	0.05	3	20	0.05	3
p73	15001	166.76	0	15001	163.0	0	15001	161.81	0
p74	6	0.01	1	6	0.01	1	6	0.01	1
p75	15001	149.55	0	43	0.24	0	43	0.2	8
p76	834	3.9	20	19	0.10	11	43	0.33	11
p77	12	0.01	2	6	0.00	1	6	0.02	1
p78	17	0.03	4	19	0.03	4	19	0.04	4
p79	14	0.02	3	10	0.03	2	10	0.03	2
p80	15001	90.85	0	15001	128.6	0	15001	128.52	0
p81	15001	76.73	0	15001	114.8	0	15001	115.27	0
p82	7	0.01	1	6	0.02	1	6	0.01	1
p83	6	0	1	6	0.01	1	6	0.01	1
p84	15001	78.71	0	14	0.05	6	14	0.05	6
p85	26	0.03	6	29	0.07	5	29	0.07	5
p86	39	0.09	9	14	0.06	3	18	0.07	3
p87	15001	98.94	0	20	0.05	4	20	0.07	4
p88	15001	86.39	0	10	0.04	2	24	0.08	2
p89	15001	109.53	0	10	0.04	2	32	0.12	3
p90	3	0	0	3	0.01	0	3	0.01	0
p91	3	0	0	3	0.00	0	3	0	0
p92	6	0.01	1	6	0.02	1	6	0.01	1
p93	15001	232.6	0	105	0.88	0	105	0.87	19
p94	3	0.01	0	3	0.01	0	3	0.01	0
p95	1968	5.17	7	6	0.01	1	6	0.01	1
p96	16	0.02	3	10	0.03	2	10	0.03	2
p97	11	0.02	2	6	0.00	1	6	0.02	1
p98	12	0.01	2	6	0.01	1	6	0.01	1
p99	3	0	0	3	0.01	0	3	0	0
p100	3	0.01	0	3	0.01	0	3	0	0

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p101	3	0.01	0	3	0.00	0	3	0	0
p102	7	0.01	1	6	0.02	1	6	0.01	1
p103	15001	141.22	0	9	0.02	2	13	0.03	2
p104	12	0.01	2	6	0.01	1	6	0.02	1
p105	3	0	0	3	0.00	0	3	0	0
p106	15001	98.64	0	25	0.12	0	25	0.12	5
p107	15001	45.15	0	13	0.03	3	13	0.03	3
p108	6	0.01	1	6	0.00	1	6	0.01	1
p109	12	0.01	2	6	0.02	1	6	0.01	1
p110	7	0	1	6	0.01	1	6	0.01	1
p111	10	0.02	2	10	0.03	2	10	0.02	2
p112	15001	140.51	0	15001	112.8	0	15001	113.09	0
p113	12	0.01	2	6	0.00	1	6	0.01	1
p114	12	0	2	6	0.02	1	6	0.02	1
p115	15001	190.43	0	15001	109.5	0	15001	111.33	0
p116	7	0.01	1	6	0.01	1	6	0.02	1
p117	36	0.07	8	10	0.03	2	10	0.04	2
p118	15001	116.32	0	17	0.09	9	17	0.08	9
p119	10	0.01	2	6	0.01	1	6	0.01	1
p120	25	0.04	5	23	0.07	4	23	0.06	4
p121	3	0.01	0	3	0.00	0	3	0	0
p122	6	0.01	1	6	0.01	1	6	0.01	1
p123	15001	134.29	0	33	0.13	0	33	0.16	6
p124	15001	100.18	0	39	0.22	0	39	0.23	8
p125	6	0.01	1	6	0.01	1	6	0.01	1
p126	195	0.36	7	10	0.02	2	10	0.03	2
p127	3	0.01	0	3	0.01	0	3	0	0
p128	15001	126.72	0	70	0.28	9	73	0.29	9
p129	12	0.02	2	6	0.01	1	6	0.01	1
p130	86	0.16	8	10	0.03	2	10	0.03	2
p131	15001	221.14	0	15001	99.62	0	15001	99.93	0
p132	11	0.01	2	6	0.01	1	6	0.01	1
p133	15001	101.41	0	19	0.10	9	35	0.17	9
p134	6	0	1	6	0.01	1	6	0	1
p135	3	0	0	3	0.00	0	3	0.01	0
p136	15001	187.35	0	15001	85.06	0	15001	83.11	0
p137	6	0.01	1	6	0.01	1	6	0.01	1
p138	6	0.01	1	6	0.01	1	6	0.01	1
p139	195	0.4	7	10	0.01	2	10	0.03	2
p140	15001	75.64	0	31	0.10	0	31	0.14	8
p141	3	0.01	0	3	0.00	0	3	0.01	0
p142	15001	230.84	0	15001	154.6	0	15001	155.13	0
p143	36	0.04	3	10	0.04	2	32	0.11	3
p144	15001	207.29	0	15001	110.6	0	15001	111.08	0
p145	86	0.15	8	10	0.04	2	10	0.03	2
p146	15001	110.24	0	15001	124.4	0	15001	127.1	0
p147	3	0.01	0	3	0.00	0	3	0.01	0
p148	49	0.05	3	10	0.01	2	10	0.04	2
p149	24	0.02	5	24	0.07	4	24	0.07	4
p150	7	0	1	6	0.01	1	6	0	1

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p151	12	0.01	2	6	0.01	1	6	0.01	1
p152	15001	86.32	0	18	0.09	9	39	0.21	9
p153	20	0.03	5	21	0.05	4	21	0.05	4
p154	6	0	1	6	0.01	1	6	0.01	1
p155	943	3.14	16	53	0.30	0	53	0.29	9
p156	12	0.02	2	6	0.01	1	6	0.02	1
p157	3368	7.95	5	10	0.03	2	10	0.03	2
p158	15001	71.98	0	15001	82.69	0	15001	82.93	0
p159	15001	53.29	0	10	0.03	2	10	0.02	2
p160	812	3.01	17	31	0.18	0	31	0.17	8
p161	15001	97.11	0	19	0.10	0	44	0.22	10
p162	7	0	1	6	0.01	1	6	0.01	1
p163	7	0.01	1	6	0.01	1	6	0.01	1
p164	15001	108.84	0	19	0.10	10	43	0.23	10
p165	6	0	1	6	0.01	1	6	0.01	1
p166	7	0.01	1	6	0.00	1	6	0.02	1
p167	10	0	2	6	0.01	1	6	0.01	1
p168	55	0.15	13	64	0.34	0	15001	65.76	0
p169	15001	264.75	0	15001	81.86	0	15001	84.81	0
p170	15001	133.88	0	18	0.05	4	40	0.15	6
p171	15001	45.09	0	19	0.04	3	19	0.04	3
p172	15001	168.53	0	15001	157.7	0	15001	157.82	0
p173	15001	140.86	0	42	0.21	8	42	0.23	8
p174	15	0.01	3	10	0.02	2	10	0.03	2
p175	46	0.08	5	10	0.02	2	10	0.03	2
p176	15001	218.55	0	15001	144.7	0	15001	147.87	0
p177	15001	47.04	0	13	0.04	3	13	0.04	3
p178	15001	107.31	0	37	0.25	0	37	0.26	10
p179	15001	125.8	0	28	0.10	5	28	0.13	5
p180	14	0.02	3	10	0.01	2	10	0.03	2
p181	15001	62.87	0	15001	99.29	0	15001	100.61	0
p182	25	0.03	5	22	0.06	4	22	0.05	4
p183	98	0.17	8	14	0.06	6	37	0.24	7
p184	15001	94.53	0	38	0.26	0	38	0.24	10
p185	15001	97.09	0	15001	127.6	0	15001	128.95	0
p186	15001	43.67	0	10	0.04	2	10	0.02	2
p187	10	0.02	2	6	0.01	1	6	0.02	1
p188	3	0.01	0	3	0.00	0	3	0	0
p189	7	0	1	6	0.01	1	6	0.01	1
p190	15001	225.22	0	31	0.18	0	42	0.31	11
p191	11	0.01	2	6	0.01	1	6	0.02	1
p192	15001	78.59	0	18	0.08	0	18	0.08	10
p193	15001	101.79	0	39	0.23	0	39	0.23	8
p194	15001	95.15	0	25	0.12	0	25	0.12	5
p195	7	0.01	1	6	0.01	1	6	0.02	1
p196	12	0.02	2	6	0.00	1	6	0.01	1
p197	25	0.04	5	19	0.05	4	19	0.06	4
p198	6	0	1	6	0.01	1	6	0.01	1
p199	15001	198.53	0	15001	66.92	0	15001	66.92	0
p200	15001	95.16	0	50	0.39	0	50	0.36	13

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p201	15001	118.67	0	15001	140.9	0	15001	140.93	0
p202	30	0.06	7	10	0.03	2	10	0.03	2
p203	10	0.01	2	10	0.03	2	10	0.03	2
p204	15001	73.01	0	14	0.05	0	15001	76.04	0
p205	15001	41.68	0	19	0.04	3	19	0.04	3
p206	16	0.01	3	10	0.03	2	10	0.02	2
p207	12	0	2	6	0.01	1	6	0.01	1
p208	15001	138.73	0	15001	74.06	0	15001	74.66	0
p209	6	0.01	1	6	0.01	1	6	0.01	1
p210	3	0	0	3	0.01	0	3	0	0
p211	15001	73.1	0	15001	82.00	0	15001	83.79	0
p212	12	0.02	2	6	0.01	1	6	0	1
p213	6	0.01	1	6	0.01	1	6	0.02	1
p214	10	0.01	2	6	0.02	1	6	0.01	1
p215	15001	127.04	0	19	0.09	11	62	0.42	11
p216	6	0.02	1	6	0.00	1	6	0.01	1
p217	10	0.02	2	6	0.01	1	6	0.01	1
p218	15001	264.62	0	15001	91.62	0	15001	93.07	0
p219	33	0.05	7	10	0.02	0	10	0.02	2
p220	15001	44.2	0	20	0.06	3	20	0.04	3
p221	6	0.01	1	6	0.01	1	6	0	1
p222	3	0.01	0	3	0.01	0	3	0.01	0
p223	36	0.06	8	10	0.03	2	10	0.03	2
p224	15001	41.61	0	19	0.05	3	19	0.05	3
p225	15	0.03	3	10	0.02	2	13	0.03	2
p226	11	0.02	2	6	0.01	1	6	0.01	1
p227	85	0.14	8	17	0.04	4	17	0.05	4
p228	10	0.01	2	6	0.01	1	6	0.01	1
p229	15001	128.66	0	27	0.11	5	27	0.1	5
p230	15001	342.09	0	38	0.22	0	38	0.23	7
p231	15001	67.77	0	33	0.15	0	33	0.14	6
p232	15001	44.93	0	20	0.05	3	20	0.05	3
p233	15001	210.53	0	15001	81.03	0	15001	80.62	0
p234	6	0.01	1	6	0.01	1	6	0.01	1
p235	15001	119.96	0	19	0.10	11	62	0.4	11
p236	17	0.02	4	19	0.04	4	19	0.06	4
p237	10	0.02	2	6	0.01	1	6	0.01	1
p238	14	0.02	3	15	0.03	3	15	0.03	3
p239	15001	133.12	0	15001	193.6	0	15001	195.2	0
p240	36	0.07	8	10	0.01	2	10	0.04	2
p241	12	0.01	2	6	0.01	1	6	0.01	1
p242	10	0.02	2	6	0.02	1	6	0.01	1
p243	694	1.53	7	10	0.04	2	10	0.03	2
p244	3	0	0	3	0.01	0	3	0	0
p245	3	0	0	3	0.00	0	3	0	0
p246	15001	255.29	0	15001	94.80	0	15001	94.87	0
p247	3	0.01	0	3	0.00	0	3	0	0
p248	15001	116.58	0	15001	71.49	0	15001	72.08	0
p249	15001	92.05	0	10	0.03	2	13	0.04	2
p250	24	0.04	5	19	0.05	4	19	0.05	4

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p1	6	0.01	1	6	0.02	1	6	0.01	1
p2	3	0.00	0	3	0.01	0	3	0	0
p3	6	0.01	1	13	0.04	3	9	0.02	1
p4	17	0.06	4	17	0.05	4	17	0.05	4
p5	22	0.15	14	90	0.6	14	15001	120.82	0
p6	3	0.00	0	3	0.01	0	3	0	0
p7	15001	189.0	0	15001	206.64	0	15001	96.38	0
p8	18	0.08	10	73	0.34	10	15001	71.44	0
p9	15001	77.89	0	15001	78.1	0	15001	77.78	0
p10	137	1.20	21	162	1.7	25	15001	116.72	0
p11	6	0.01	1	6	0.01	1	6	0.01	1
p12	15001	137.2	0	15001	134.16	0	15001	151.74	0
p13	15001	103.9	0	15001	155.69	0	20	0.09	4
p14	6	0.01	1	9	0.01	2	6	0.01	1
p15	27	0.08	5	27	0.11	5	27	0.1	5
p16	6	0.02	1	6	0.02	1	6	0.01	1
p17	15001	149.6	0	15001	231.1	0	15001	150.26	0
p18	15001	86.84	0	15001	192.21	0	45	0.28	11
p19	3	0.00	0	3	0	0	3	0	0
p20	17	0.06	4	17	0.06	4	17	0.05	4
p21	10	0.03	2	10	0.03	2	10	0.03	2
p22	24	0.07	4	24	0.06	4	24	0.06	4
p23	19	0.05	4	19	0.06	4	19	0.06	4
p24	10	0.03	2	10	0.03	2	10	0.03	2
p25	3	0.00	0	3	0.01	0	3	0	0
p26	15001	211.4	0	15001	406.68	0	15001	81.71	0
p27	15001	104.4	0	15001	103.84	0	30	0.17	6
p28	19	0.05	4	19	0.06	4	19	0.07	4
p29	15001	80.48	0	15001	183.98	0	25	0.12	5
p30	6	0.02	1	6	0	1	6	0.01	1
p31	13	0.03	3	13	0.03	3	13	0.03	3
p32	3	0.00	0	3	0.01	0	3	0	0
p33	10	0.02	2	10	0.02	2	10	0.03	2
p34	19	0.04	4	31	0.13	6	19	0.05	4
p35	21	0.06	4	21	0.04	4	21	0.06	4
p36	6	0.01	1	6	0.02	1	6	0.01	1
p37	6	0.01	1	6	0.01	1	6	0.01	1
p38	6	0.02	1	6	0.01	1	6	0.01	1
p39	6	0.01	1	6	0.01	1	6	0.01	1
p40	15001	85.18	0	15001	105.45	0	15001	153.55	0
p41	6	0.01	1	6	0.02	1	6	0.01	1
p42	10	0.03	2	10	0.03	2	10	0.02	2
p43	3	0.01	0	3	0	0	3	0.01	0
p44	6	0.01	1	6	0.02	1	6	0.01	1
p45	19	0.05	4	19	0.05	4	19	0.05	4
p46	15001	65.40	0	15001	65.34	0	15001	69.27	0
p47	3	0.00	0	3	0	0	3	0.01	0
p48	10	0.02	2	10	0.04	2	10	0.01	2
p49	10	0.03	2	17	0.04	4	32	0.1	3
p50	6	0.01	1	6	0.01	1	6	0	1

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p51	10	0.04	2	17	0.05	4	32	0.12	3
p52	3	0.01	0	3	0	0	3	0.01	0
p53	6	0.01	1	6	0.01	1	6	0.01	1
p54	15001	75.84	0	15001	75.77	0	37	0.17	7
p55	6	0.01	1	6	0.01	1	6	0.01	1
p56	6	0.01	1	6	0	1	6	0.01	1
p57	6	0.01	1	9	0.02	2	6	0.01	1
p58	15001	82.17	0	15001	82.71	0	15001	82.89	0
p59	15001	158.5	0	15001	230.48	0	15001	84.58	0
p60	15001	105.4	0	15001	186.73	0	15001	104.82	0
p61	6	0.01	1	6	0	1	6	0.01	1
p62	10	0.04	2	10	0.03	2	10	0.03	2
p63	15001	67.02	0	15001	168.45	0	22	0.06	6
p64	15001	92.00	0	15001	199.31	0	53	0.26	9
p65	22	0.13	5	15001	84.02	0	15001	105.75	0
p66	10	0.03	2	10	0.02	2	10	0.03	2
p67	6	0.01	1	6	0.01	1	6	0.01	1
p68	15001	105.8	0	15001	150.76	0	33	0.15	6
p69	19	0.04	3	19	0.05	3	19	0.04	3
p70	6	0.01	1	6	0.02	1	6	0.02	1
p71	19	0.10	11	86	0.63	15	38	0.2	7
p72	20	0.05	3	20	0.05	3	20	0.05	3
p73	15001	153.8	0	15002	296.63	0	15001	163.04	0
p74	6	0.02	1	6	0.02	1	6	0.01	1
p75	15001	106.0	0	15001	127.64	0	43	0.24	8
p76	19	0.10	11	87	0.68	15	43	0.28	11
p77	6	0.01	1	6	0.02	1	6	0	1
p78	19	0.06	4	19	0.05	4	19	0.03	4
p79	10	0.04	2	10	0.03	2	10	0.03	2
p80	15001	128.6	0	15001	158.38	0	15001	128.63	0
p81	15001	114.9	0	15001	184.45	0	15001	114.82	0
p82	6	0.01	1	6	0.01	1	6	0.02	1
p83	6	0.01	1	6	0.02	1	6	0.01	1
p84	14	0.05	6	32	0.12	6	32	0.12	6
p85	29	0.08	5	29	0.08	5	29	0.07	5
p86	14	0.06	3	18	0.06	4	18	0.07	3
p87	21	0.07	4	22	0.06	4	20	0.05	4
p88	10	0.04	2	17	0.05	4	24	0.08	2
p89	10	0.04	2	17	0.04	4	32	0.11	3
p90	3	0.01	0	3	0	0	3	0.01	0
p91	3	0.00	0	3	0.01	0	3	0	0
p92	6	0.01	1	9	0.02	2	6	0.02	1
p93	15001	276.8	0	15001	299.22	0	105	0.88	19
p94	3	0.00	0	3	0	0	3	0.01	0
p95	6	0.01	1	6	0.01	1	6	0.01	1
p96	10	0.04	2	10	0.03	2	10	0.03	2
p97	6	0.01	1	6	0	1	6	0	1
p98	6	0.00	1	6	0.01	1	6	0.01	1
p99	3	0.01	0	3	0.01	0	3	0.01	0
p100	3	0.00	0	3	0.01	0	3	0.01	0

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p101	3	0.01	0	3	0.01	0	3	0	0
p102	6	0.01	1	6	0.01	1	6	0.02	1
p103	10	0.03	2	9	0.02	2	13	0.03	2
p104	6	0.01	1	6	0.01	1	6	0.01	1
p105	3	0.00	0	3	0.01	0	3	0	0
p106	15001	80.49	0	116	0.95	17	25	0.12	5
p107	13	0.03	3	13	0.04	3	13	0.03	3
p108	6	0.01	1	6	0.02	1	6	0	1
p109	6	0.01	1	6	0.01	1	6	0.02	1
p110	6	0.00	1	6	0.02	1	6	0.01	1
p111	10	0.03	2	10	0.03	2	10	0.03	2
p112	15001	108.3	0	15001	242.25	0	15001	112.86	0
p113	6	0.01	1	6	0.01	1	6	0	1
p114	6	0.01	1	6	0.01	1	6	0.02	1
p115	15001	110.3	0	15001	109.83	0	15001	109.58	0
p116	6	0.02	1	6	0	1	6	0.01	1
p117	10	0.03	2	10	0.03	2	10	0.03	2
p118	17	0.09	9	46	0.23	9	46	0.25	9
p119	6	0.01	1	6	0	1	6	0.01	1
p120	23	0.06	4	23	0.06	4	23	0.07	4
p121	3	0.00	0	3	0	0	3	0	0
p122	6	0.01	1	6	0.01	1	6	0.01	1
p123	15001	71.97	0	15001	72.03	0	33	0.13	6
p124	15001	83.12	0	15001	185.26	0	39	0.22	8
p125	6	0.01	1	6	0.01	1	6	0.01	1
p126	10	0.02	2	10	0.03	2	10	0.02	2
p127	3	0.00	0	3	0.01	0	3	0.01	0
p128	70	0.30	9	70	0.28	9	73	0.3	9
p129	6	0.01	1	6	0.01	1	6	0.01	1
p130	10	0.04	2	10	0.04	2	10	0.03	2
p131	15001	100.3	0	15001	175.94	0	15001	99.62	0
p132	6	0.01	1	6	0	1	6	0.01	1
p133	19	0.10	11	48	0.3	11	35	0.18	9
p134	6	0.01	1	6	0.01	1	6	0.01	1
p135	3	0.01	0	3	0	0	3	0	0
p136	15001	200.7	0	15001	89.95	0	15001	85.06	0
p137	6	0.01	1	6	0.02	1	6	0.01	1
p138	6	0.00	1	6	0.01	1	6	0.01	1
p139	10	0.03	2	10	0.03	2	10	0.01	2
p140	15001	73.20	0	15001	173.71	0	31	0.1	8
p141	3	0.00	0	3	0	0	3	0	0
p142	15001	153.7	0	15001	154.38	0	15001	154.65	0
p143	10	0.04	2	17	0.06	4	32	0.12	3
p144	15001	86.03	0	15001	193.03	0	15001	110.62	0
p145	10	0.01	2	10	0.03	2	10	0.04	2
p146	15001	133.4	0	15001	289.47	0	15001	124.4	0
p147	3	0.00	0	3	0.01	0	3	0	0
p148	10	0.03	2	10	0.04	2	10	0.01	2
p149	24	0.07	4	24	0.08	4	24	0.07	4
p150	6	0.00	1	9	0.02	2	6	0.01	1

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p151	6	0.01	1	9	0.01	2	6	0.01	1
p152	18	0.09	10	56	0.27	10	39	0.2	9
p153	21	0.06	4	21	0.07	4	21	0.05	4
p154	6	0.01	1	6	0.01	1	6	0.01	1
p155	15001	92.16	0	15001	199.63	0	53	0.3	9
p156	6	0.01	1	6	0.02	1	6	0.01	1
p157	10	0.02	2	10	0.02	2	10	0.03	2
p158	15001	128.1	0	15001	196.78	0	15001	82.69	0
p159	10	0.03	2	10	0.04	2	10	0.03	2
p160	15001	67.18	0	15001	175.01	0	31	0.18	8
p161	19	0.10	11	15001	188.93	0	44	0.24	10
p162	6	0.02	1	6	0	1	6	0.01	1
p163	6	0.01	1	6	0.01	1	6	0.01	1
p164	19	0.10	11	48	0.34	11	43	0.23	10
p165	6	0.00	1	6	0.02	1	6	0.01	1
p166	6	0.01	1	6	0.01	1	6	0	1
p167	6	0.02	1	6	0.01	1	6	0.01	1
p168	15001	65.67	0	64	0.34	11	15001	73.11	0
p169	15001	97.16	0	15001	97.17	0	15001	81.86	0
p170	33	0.13	6	18	0.05	4	40	0.14	6
p171	19	0.06	3	19	0.05	3	19	0.04	3
p172	15001	211.9	0	15001	115.45	0	15001	157.73	0
p173	67	0.42	12	67	0.4	12	42	0.21	8
p174	10	0.03	2	10	0.03	2	10	0.02	2
p175	10	0.04	2	10	0.02	2	10	0.02	2
p176	15001	91.38	0	15001	200.11	0	15001	144.72	0
p177	13	0.04	3	13	0.04	3	13	0.04	3
p178	15001	83.74	0	15001	189.33	0	37	0.25	10
p179	138	0.58	13	139	0.57	13	28	0.1	5
p180	10	0.04	2	10	0.02	2	10	0.01	2
p181	15001	104.3	0	15001	96.24	0	15001	99.29	0
p182	22	0.05	4	22	0.06	4	22	0.06	4
p183	14	0.06	6	29	0.11	6	37	0.23	7
p184	15001	72.81	0	112	0.87	16	38	0.26	10
p185	15001	128.6	0	15001	240.86	0	15001	127.64	0
p186	10	0.03	2	10	0.03	2	10	0.04	2
p187	6	0.02	1	9	0.02	2	6	0.01	1
p188	3	0.01	0	3	0	0	3	0	0
p189	6	0.01	1	6	0.01	1	6	0.01	1
p190	31	0.18	13	15001	231.92	0	42	0.3	11
p191	6	0.00	1	6	0.01	1	6	0.01	1
p192	18	0.08	10	73	0.32	10	15001	71.51	0
p193	15001	84.42	0	15001	190.16	0	39	0.23	8
p194	15001	109.8	0	15001	155.24	0	25	0.12	5
p195	6	0.01	1	6	0.01	1	6	0.01	1
p196	6	0.02	1	6	0.02	1	6	0	1
p197	19	0.06	4	19	0.05	4	19	0.05	4
p198	6	0.00	1	6	0.01	1	6	0.01	1
p199	15001	67.32	0	15001	69.07	0	15001	66.92	0
p200	15001	99.19	0	15001	214.93	0	50	0.39	13

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p201	15001	90.27	0	15001	91.69	0	15001	140.95	0
p202	10	0.03	2	13	0.03	3	10	0.03	2
p203	10	0.02	2	10	0.02	2	10	0.03	2
p204	14	0.05	6	32	0.13	6	15001	75.68	0
p205	19	0.05	3	19	0.04	3	19	0.04	3
p206	10	0.03	2	10	0.03	2	10	0.03	2
p207	6	0.02	1	9	0.03	2	6	0.01	1
p208	15001	96.40	0	15001	107.26	0	15001	74.06	0
p209	6	0.01	1	6	0.02	1	6	0.01	1
p210	3	0.01	0	3	0	0	3	0.01	0
p211	15001	128.7	0	15001	197.87	0	15001	82	0
p212	6	0.02	1	6	0.01	1	6	0.01	1
p213	6	0.01	1	6	0	1	6	0.01	1
p214	6	0.02	1	6	0.01	1	6	0.02	1
p215	19	0.09	11	60	0.43	12	62	0.39	11
p216	6	0.01	1	9	0.01	2	6	0	1
p217	6	0.01	1	6	0.01	1	6	0.01	1
p218	15001	92.37	0	15001	103.78	0	15001	91.62	0
p219	10	0.03	2	15001	82.82	0	10	0.02	2
p220	20	0.04	3	20	0.06	3	20	0.06	3
p221	6	0.02	1	6	0.01	1	6	0.01	1
p222	3	0.00	0	3	0	0	3	0.01	0
p223	10	0.03	2	10	0.03	2	10	0.03	2
p224	19	0.04	3	19	0.05	3	19	0.05	3
p225	10	0.02	2	13	0.03	3	13	0.02	2
p226	6	0.01	1	6	0.01	1	6	0.01	1
p227	17	0.05	4	17	0.06	4	17	0.04	4
p228	6	0.01	1	6	0.01	1	6	0.01	1
p229	27	0.11	5	27	0.1	5	27	0.11	5
p230	15001	105.5	0	15001	198.87	0	38	0.22	7
p231	15001	75.13	0	15001	75.21	0	33	0.15	6
p232	20	0.05	3	20	0.05	3	20	0.05	3
p233	15001	181.8	0	15001	279.42	0	15001	81.03	0
p234	6	0.01	1	6	0	1	6	0.01	1
p235	19	0.10	11	60	0.46	12	62	0.4	11
p236	19	0.05	4	19	0.06	4	19	0.04	4
p237	6	0.01	1	6	0.01	1	6	0.01	1
p238	15	0.04	3	15	0.04	3	15	0.03	3
p239	15001	150.3	0	15001	276.97	0	15001	193.6	0
p240	10	0.04	2	10	0.03	2	10	0.01	2
p241	6	0.01	1	6	0.01	1	6	0.01	1
p242	6	0.01	1	6	0.01	1	6	0.02	1
p243	10	0.02	2	10	0.02	2	10	0.04	2
p244	3	0.01	0	3	0.01	0	3	0.01	0
p245	3	0.01	0	3	0	0	3	0	0
p246	15001	88.56	0	15001	106.23	0	15001	94.8	0
p247	3	0.01	0	3	0.01	0	3	0	0
p248	15001	71.73	0	15001	72.11	0	15001	71.49	0
p249	10	0.03	2	13	0.04	3	13	0.02	2
p250	19	0.03	4	19	0.06	4	19	0.05	4

Appendix D

The kitchen domain

The kitchen domain simulates a one armed mobile robot that can make different beverages. Chapter ?? contains a more detailed description of the kitchen domain. The kitchen domain consists of 51 objects and 51 operators.

All experiments were run on an IBM PC compatible computer with 16 MB of RAM and a 486-DX66 processor. The tables below compare DoLittlerunning in Prodigy emulation mode (abstractions off, no general operators, relevant operator selection method) to two different multi-strategy planners (PC-MSP-O and DoLittle) and three single strategy planners (Cases, macros, abstractions), based on the learning methods described in chapter ??.

Since DoLittle's planning bias learners currently only learn from success, the difficulty of the problems in the training phase and test phase was gradually increased. The test phase consisted of 150 problems.

D.1 Randomly generating problems in the kitchen

The following algorithm was used to create a random problem in the kitchen domain. The procedure uses one parameter DIFFICULTY, which ranges from 1 to 5. After every 30 problems in the training set and every 50 problems in the test set, the parameter DIFFICULTY is incremented.

In the first group (DIFFICULTY=1), goals were restricted to disallow any goals asking for preparation of beverages (e.g., milk, tea, coffee). So, (CONTAINS CUP1 TEA) was not allowed, but (HOLDING CUP1) is a possible goal. In the second group (DIFFICULTY=2), the problems may contain problems that require filling a cup with milk or water. The third group (DIFFICULTY=3) includes problems that require making tea, instant coffee, coffee. The fourth group (DIFFICULTY=4) includes problems that have a beverage and one ingredient, e.g., tea with milk. The fifth group (DIFFICULTY=5) contains problems with a drink and two ingredients, e.g., tea with milk and honey.

First the position of the cups and glasses is calculated. Each cup and glass is with probability 0.5 on the table and otherwise in the cupboard. Next the status of the doors is computed similarly (a door is open with probability 0.5). The number of drinks is a uniformly distributed random variable between 1 and 3. Each drink has an associated container ((CUP1),(CUP2),(CUP3)). For each container, the type of drink is determined based on the value of the parameter DIFFICULTY. The possible goals for the first group contain the single literal (CONTAINS CUPX NOTHING). The drinks of the second group contain the ones of the first group and also ((CONTAINS CUPX WATER), (CONTAINS CUPX MILK)). The third group adds drinks that contain a simple beverage with no ingredients. The beverages are tea, instant coffee, and coffee. In the fourth group, a beverage contains one ingredient (milk, sugar, ice, and honey). In the fifth group, a beverage may contain two in-

gredients such as cream and sugar. With probability 0.3, the goal (HOLDING CUPX) is added. With probability 0.3, the conjunct (IS-AT ROBBY LOCATION) is added to the goal, where (LOCATION) is selected at random from the four locations of the domain ((AT-SINK), (AT-TABLE), (AT-STOVE), and (AT-FRIDGE)).

D.2 Domain specification of the kitchen domain

The following is the description of the kitchen domain in DOLITTLE's representation language.

```
(create-problem-space 'kitchen :current t)

(ptype-of object :top-type)

(ptype-of movable object)

(ptype-of silverware movable)
(ptype-of fillable movable)

(ptype-of microwavable fillable)

(ptype-of cup microwavable)
(ptype-of glass microwavable)

(ptype-of can fillable)

(ptype-of container movable)
(ptype-of kettle movable)
(ptype-of cutable movable)
(ptype-of location :top-type)
(ptype-of door object)
(ptype-of robot object)
(ptype-of drink object)
```

```
(ptype-of ingredients object)

(pinstance-of knife silverware)
(pinstance-of fork silverware)
(pinstance-of spoon silverware)
(pinstance-of scissors silverware)

; Declare the instances of the different types

(pinstance-of cup1 cup)
(pinstance-of cup2 cup)
(pinstance-of cup3 cup)

(pinstance-of glass1 glass)
(pinstance-of glass2 glass)
(pinstance-of glass3 glass)

(pinstance-of kettle1 kettle)
(pinstance-of coffee-can can)

(pinstance-of tea-box container)
(pinstance-of instant-coffee-jar container)
(pinstance-of coffee-jar container)
(pinstance-of sugar-box container)
(pinstance-of honey-jar container)

(pinstance-of microwave door)
(pinstance-of drawer door)
(pinstance-of cupboard door)
(pinstance-of fridge door)

(pinstance-of milk-carton cutable)
(pinstance-of tea-bag movable)
(pinstance-of old-tea-bag movable)

(pinstance-of at-sink location)
(pinstance-of at-table location)
(pinstance-of at-stove location)
(pinstance-of at-fridge location)
```

```
(pinstance-of robby robot)

(pinstance-of water drink)
(pinstance-of hot-water drink)
(pinstance-of milk drink)
(pinstance-of hot-milk drink)
(pinstance-of ice drink)
(pinstance-of tea drink)
(pinstance-of iced-tea drink)
(pinstance-of instant-coffee drink)
(pinstance-of coffee drink)
(pinstance-of instant-coffee drink)
(pinstance-of iced-coffee drink)
(pinstance-of iced-instant-coffee drink)
(pinstance-of nothing drink)

(pinstance-of table object)
(pinstance-of sink object)
(pinstance-of stove object)
(pinstance-of shelf object)
(pinstance-of coffee-maker object)
(pinstance-of garbage-can object)

(pinstance-of honey ingredients)
(pinstance-of sugar ingredients)
(pinstance-of little-milk ingredients)


; ---------- Move the robot around ----------
(OPERATOR MOVE-ROBOT
          (params <robot-loc> <new-loc>)
          (preconds
           ((<robot-loc> location)
            (<new-loc> location))
           (and (next-to <robot-loc> <new-loc>)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (is-at robby <new-loc>))
            (del (is-at robby <robot-loc>)))))
```

```
; ---------- Use the water tab ----------
(OPERATOR FILL-WITH-WATER
          (params <object>)
          (preconds
           ((<object> fillable)
            (<robot-loc> location))
           (and
            (is-reachable sink <robot-loc>)
            (arm-empty)
            (~ (water-on))
            (contains <object> nothing)
            (is-at robby <robot-loc>)
            (is-in <object> sink)
            ))
          (effects
           ()
           ((add (water-on))
            (del (contains <object> nothing))
            (add (contains <object> water)))))



(OPERATOR TURN-WATER-OFF
          (params)
          (preconds
           ((<robot-loc> location))
           (and
            (is-reachable sink <robot-loc>)
            (is-at robby <robot-loc>)
            (water-on)
            (arm-empty)))
          (effects
           ()
           ((del (water-on)))))

; ---------- Using the silverware ----------
(OPERATOR CUT
          (params <object>)
          (preconds
```

```
           ((<object> cutable)
            (<tool> silverware)
            (<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (holding <tool>)
                (is-at robby <robot-loc>)
                (is-on <object> table)
                (~ (is-open <object>))))
          (effects
           ()
           ((add (is-open <object>)))))

(OPERATOR STIR
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (holding spoon)
                (is-at robby <robot-loc>)
                (is-on <object> table)
                (to-stir <object>)))
          (effects
           ()
           ((del (to-stir <object>)))))



; ---------- Use the microwave ----------
(OPERATOR HEAT-WATER-IN-MICROWAVE
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and
            (is-reachable microwave <robot-loc>)
            (~ (is-open microwave))
            (is-in <object> microwave)
            (is-at robby <robot-loc>)
            (contains <object> water)
```

```
            (arm-empty)))
          (effects
           ()
           ((add (is-hot <object>))
            (del (contains <object> water))
            (add (contains <object> hot-water)))))

(OPERATOR HEAT-MILK-IN-MICROWAVE
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and
            (is-reachable microwave <robot-loc>)
            (~ (is-open microwave))
            (is-in <object> microwave)
            (is-at robby <robot-loc>)
            (contains <object> milk)
            (arm-empty)))
          (effects
           ()
           ((add (is-hot <object>))
            (del (contains <object> milk))
            (add (contains <object> hot-milk)))))

; ---------- Stove ----------
(OPERATOR USE-STOVE
          (params <object>)
          (preconds
           ((<object> kettle)
            (<robot-loc> location))
           (and
            (is-reachable stove <robot-loc>)
            (is-at robby <robot-loc>)
            (arm-empty)
            (contains <object> water)
            (is-on <object> stove)))
          (effects
           ()
           ((add (contains <object> hot-water))
```

```
            (del (contains <object> water)))))

; ---------- Picking things up and putting them down ----------
(OPERATOR PICK-UP-FROM-TABLE
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (arm-empty)
            (is-at robby <robot-loc>)
            (is-on <object> table)))
          (effects
           ()
           ((add (holding <object>))
            (del (arm-empty))
            (del (is-on <object> table)))))

(OPERATOR PUT-ON-TABLE
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (holding <object>)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (is-on <object> table))
            (add (arm-empty))
            (del (holding <object>)))))

(OPERATOR PICK-UP-FROM-DRAWER
          (params <object>)
          (preconds
           ((<object> silverware)
            (<robot-loc> location))
           (and
            (is-reachable drawer <robot-loc>)
```

```
           (arm-empty)
           (is-open drawer)
           (is-at robby <robot-loc>)
           (is-in <object> drawer)))
         (effects
          ()
          ((add (holding <object>))
           (del (arm-empty))
           (del (is-in <object> drawer)))))

 (OPERATOR PUT-IN-DRAWER
           (params <object>)
           (preconds
            ((<object> silverware)
             (<robot-loc> location))
            (and
             (is-reachable drawer <robot-loc>)
             (holding <object>)
             (is-open drawer)
             (is-at robby <robot-loc>)))
           (effects
            ()
            ((add (is-in <object> drawer))
             (add (arm-empty))
             (del (holding <object>)))))

 (OPERATOR PICK-UP-FROM-SINK
           (params <object>)
           (preconds
            ((<object> movable)
             (<robot-loc> location)
            (and
             (is-reachable sink <robot-loc>)
             (arm-empty)
             (is-at robby <robot-loc>)
             (is-in <object> sink)))
           (effects
            ()
            ((add (holding <object>))
             (del (arm-empty))
```

```
             (add (sink-empty))
             (del (is-in <object> sink)))))

  (OPERATOR PUT-IN-SINK
            (params <object>)
            (preconds
             ((<object> movable)
              (<robot-loc> location))
             (and (is-reachable sink <robot-loc>)
                  (holding <object>)
                  (sink-empty)
                  (is-at robby <robot-loc>)))
            (effects
             ()
             ((add (is-in <object> sink))
              (del (sink-empty))
              (add (arm-empty))
              (del (holding <object>)))))

  (OPERATOR PICK-UP-FROM-FRIDGE
            (params <object>)
            (preconds
             ((<object> movable)
              (<robot-loc> location))
             (and
              (is-reachable fridge <robot-loc>)
              (arm-empty)
              (is-at robby <robot-loc>)
              (is-open fridge)
              (is-in <object> fridge)))
            (effects
             ()
             ((add (holding <object>))
              (del (arm-empty))
              (del (is-in <object> fridge)))))

(OPERATOR PUT-IN-FRIDGE
          (params <object>)
          (preconds
           ((<object> movable)
```

```
            (<robot-loc> location))
           (and (is-reachable fridge <robot-loc>)
                (holding <object>)
                (is-open fridge)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (is-in <object> fridge))
            (add (arm-empty))
            (del (holding <object>)))))

(OPERATOR PICK-UP-FROM-MICROWAVE
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and
            (is-reachable microwave <robot-loc>)
            (arm-empty)
            (is-at robby <robot-loc>)
            (is-open microwave)
            (is-in <object> microwave)))
          (effects
           ()
           ((add (holding <object>))
            (del (arm-empty))
            (add (microwave-empty))
            (del (is-in <object> microwave)))))

(OPERATOR PUT-IN-MICROWAVE
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and
            (is-reachable microwave <robot-loc>)
            (holding <object>)
            (is-open microwave)
            (microwave-empty)
            (is-at robby <robot-loc>)))
```

```
          (effects
           ()
           ((add (is-in <object> microwave))
            (add (arm-empty))
            (del (microwave-empty))
            (del (holding <object>)))))

(OPERATOR PICK-UP-FROM-CUPBOARD
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and
            (is-reachable cupboard <robot-loc>)
            (arm-empty)
            (is-at robby <robot-loc>)
            (is-open cupboard)
            (is-in <object> cupboard)))
          (effects
           ()
           ((add (holding <object>))
            (del (arm-empty))
            (del (is-in <object> cupboard)))))

(OPERATOR PICK-UP-FROM-STOVE
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and
            (is-reachable stove <robot-loc>)
            (arm-empty)
            (is-on <object> stove)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (is-on <object> stove))
            (del (arm-empty))
              (add (stove-empty))
              (add (holding <object>)))))
```

```
(OPERATOR PUT-ON-STOVE
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and (is-reachable stove <robot-loc>)
                (holding <object>)
                (stove-empty)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (is-on <object> stove))
            (add (arm-empty))
            (del (stove-empty))
            (del (holding <object>)))))

(OPERATOR PICK-UP-FROM-SHELF
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and
              (is-reachable shelf <robot-loc>)
              (arm-empty)
              (is-at robby <robot-loc>)
              (is-on <object> shelf)))
          (effects
           ()
           ((add (holding <object>))
            (del (arm-empty))
            (del (is-on <object> shelf)))))

(OPERATOR PUT-ON-SHELF
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and (holding <object>)
                (is-reachable shelf <robot-loc>)
```

```
               (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (is-on <object> shelf))
            (add (arm-empty))
            (del (holding <object>)))))

; ---------- Ice tray ----------
(OPERATOR GET-ICE
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and
            (is-reachable fridge <robot-loc>)
            (is-at robby <robot-loc>)
            (contains <object> nothing)))
          (effects
           ()
           ((add (contains <object> ice))
            (del (contains <object> nothing)))))

(OPERATOR MAKE-ICED-TEA
          (params <cup1> <cup2>)
          (preconds
           ((<cup1> fillable)
            (<cup2> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (contains <cup1> tea)
            (contains <cup2> ice)
            (is-on <cup2> table)
            (holding <cup1>)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <cup2> ice))
            (del (contains <cup1> tea))
            (add (contains <cup2> iced-tea)))))
```

```
(OPERATOR MAKE-ICED-COFFEE
          (params <cup1> <cup2>)
          (preconds
           ((<cup1> fillable)
            (<cup2> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (contains <cup1> coffee)
            (contains <cup2> ice)
            (is-on <cup2> table)
            (holding <cup1>)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <cup2> ice))
            (del (contains <cup1> coffee))
            (add (contains <cup2> iced-coffee)))))

(OPERATOR MAKE-ICED-INSTANT-COFFEE
          (params <cup1> <cup2>)
          (preconds
           ((<cup1> fillable)
            (<cup2> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (contains <cup1> instant-coffee)
            (contains <cup2> ice)
            (is-on <cup2> table)
            (holding <cup1>)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <cup2> ice))
            (del (contains <cup1> instant-coffee))
            (add (contains <cup2> iced-instant-coffee)))))

; ---------- Things to do with a tea bag ----------
```

```
(OPERATOR GET-TEA-BAG
          (params)
          (preconds
           ((<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (is-at robby <robot-loc>)
                (arm-empty)
                (is-on tea-box table)
                (is-open tea-box)))
          (effects
           ()
           ((add (holding tea-bag))
            (del (arm-empty)))))

(OPERATOR MAKE-TEA
          (params <object>)
          (preconds
           ((<object> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (contains <object> hot-water)
            (holding tea-bag)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <object> hot-water))
            (add (contains <object> tea))
            (add (holding old-tea-bag))
            (del (holding tea-bag)))))

; ---------- Handling instant coffee ---------
(OPERATOR SCOOP-INSTANT-COFFEE
          (params)
          (preconds
           ((<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (holding spoon)
                (is-on instant-coffee-jar table)
```

```
               (is-open instant-coffee-jar)
               (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (contains spoon instant-coffee)))))

(OPERATOR POURS-INSTANT-COFFEE
          (params <object>)
          (preconds
           ((<object> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (contains <object> hot-water)
            (holding spoon)
            (contains spoon instant-coffee)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <object> hot-water))
            (del (contains spoon instant-coffee))
            (add (to-stir <object>))
            (add (contains <object> instant-coffee)))))

; ---------- Honey ---------
(OPERATOR SCOOP-HONEY
          (params)
          (preconds
           ((<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (holding spoon)
                (is-on honey-jar table)
                (is-open honey-jar)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (contains spoon honey)))))

(OPERATOR ADD-HONEY-TO-MILK
```

```
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (contains <object> hot-milk)
            (holding spoon)
            (contains spoon honey)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains spoon honey))
            (add (contains <object> honey))
            (add (to-stir <object>)))))

(OPERATOR ADD-HONEY-TO-TEA
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (contains <object> tea)
            (holding spoon)
            (contains spoon honey)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains spoon honey))
            (add (contains <object> honey))
            (add (to-stir <object>)))))

; ---------- Sugar ----------
(OPERATOR GET-SUGAR
          (params)
          (preconds
           ((<robot-loc> location))
```

```
           (and (is-reachable table <robot-loc>)
                (is-on sugar-box table)
                (is-open sugar-box)
                (holding spoon)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (contains spoon sugar)))))

(OPERATOR ADD-SUGAR
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (or (contains <object> hot-milk)
                (contains <object> tea)
                (contains <object> coffee)
                (contains <object> instant-coffee))
            (holding spoon)
            (contains spoon sugar)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains spoon sugar))
            (add (contains <object> sugar))
            (add (to-stir <object>)))))

; --------- Milk ---------

(OPERATOR POUR-MILK
          (params <object>)
          (preconds
           ((<object> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
```

```
            (contains <object> nothing)
            (holding milk-carton)
            (is-open milk-carton)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <object> nothing))
            (add (contains <object> milk)))))


(OPERATOR ADD-MILK
          (params <object>)
          (preconds
           ((<object> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (is-open milk-carton)
            (or (contains <object> tea)
                (contains <object> coffee)
                (contains <object> instant-coffee))
            (holding milk-carton)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (contains <object> little-milk))
            (add (to-stir <object>)))))

; ---------- Coffee-maker ----------

(OPERATOR SCOOP-COFFEE
          (params)
          (preconds
           ((<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (holding spoon)
                (is-on coffee-jar table)
                (is-open coffee-jar)
                (is-at robby <robot-loc>)))
```

```
          (effects
           ()
           ((add (contains spoon coffee)))))

(OPERATOR ADD-WATER-TO-COFFEE-MAKER
          (params <object>)
          (preconds
           ((<object> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on coffee-maker table)
            (~ (contains coffee-maker water))
            (holding <object>)
            (contains <object> water)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <object> water))
            (add (contains coffee-maker water)))))


(OPERATOR ADD-COFFEE-TO-COFFEE-MAKER
          (params)
          (preconds
           ((<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (is-on coffee-maker table)
                (~ (contains coffee-maker coffee))
                (holding spoon)
                (contains spoon coffee)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains spoon coffee))
            (add (contains coffee-maker coffee)))))


(OPERATOR USE-COFFEE-MAKER
          (params)
```

```
          (preconds
           ((<robot-loc> location))
           (and (is-reachable table <robot-loc>)
                (is-at robby <robot-loc>)
                (contains coffee-maker coffee)
                (contains coffee-maker water)
                (is-in coffee-can coffee-maker)
                (contains coffee-can nothing)
                (arm-empty)))
          (effects
           ()
           ((del (contains coffee-can nothing))
            (del (contains coffee-maker water))
            (del (contains coffee-maker coffee))
            (add (contains coffee-can coffee)))))

(OPERATOR PUT-IN-COFFEE-MAKER
          (params <object>)
          (preconds
           ((<object> can)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (holding <object>)
            (coffee-maker-empty)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (is-in <object> coffee-maker))
            (del (coffee-maker-empty))
            (add (arm-empty))
            (del (holding <object>)))))


(OPERATOR PICK-UP-FROM-COFFEE-MAKER
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and
```

```
            (is-reachable table <robot-loc>)
            (arm-empty)
            (is-at robby <robot-loc>)
            (is-in <object> coffee-maker)))
          (effects
           ()
           ((add (holding <object>))
            (del (arm-empty))
            (add (coffee-maker-empty))
            (del (is-in <object> coffee-maker)))))


(OPERATOR POUR-COFFEE
          (params <object>)
          (preconds
           ((<object> fillable)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (contains <object> nothing)
            (holding coffee-can)
            (contains coffee-can coffee)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains <object> nothing))
            (add (contains <object> coffee)))))

; ---------- Kettle ----------
(OPERATOR POUR-FROM-KETTLE
          (params <object>)
          (preconds
           ((<object> microwavable)
            (<x> drink)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (contains <object> nothing)
```

```
            (holding kettle1)
            (contains kettle1 <x>)
            (is-at robby <robot-loc>)))
          (effects
           ()
           ((del (contains kettle1 <x>))
            (add (contains <object> <x>)))))

; ---------- Garbage-can ----------
(OPERATOR PUT-IN-GARBAGE-CAN
          (params <object>)
          (preconds
           ((<object> movable)
            (<robot-loc> location))
           (and (is-reachable garbage-can <robot-loc>)
                (holding <object>)
                (is-at robby <robot-loc>)))
          (effects
           ()
           ((add (is-in <object> garbage-can))
            (add (arm-empty))
            (del (holding <object>)))))



; ---------- Opening and closing things ----------
(OPERATOR OPEN-DOOR
          (params <object>)
          (preconds
           ((<object> door)
            (<robot-loc> location))
           (and
            (is-reachable <object> <robot-loc>)
            (is-at robby <robot-loc>)
            (arm-empty)
            (~ (is-open <object>))))
          (effects
           ()
           ((add (is-open <object>)))))
```

```
(OPERATOR CLOSE-DOOR
          (params <object>)
          (preconds
           ((<object> door)
            (<robot-loc> location))
           (and
            (is-reachable <object> <robot-loc>)
            (is-at robby <robot-loc>)
            (arm-empty)
            (is-open <object>)))
          (effects
           ()
           ((del (is-open <object>)))))

(OPERATOR OPEN-CONTAINER
          (params <object>)
          (preconds
           ((<object> container)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (is-at robby <robot-loc>)
            (arm-empty)
            (~ (is-open <object>))))
          (effects
           ()
           ((add (is-open <object>)))))

(OPERATOR CLOSE-CONTAINER
          (params <object>)
          (preconds
           ((<object> container)
            (<robot-loc> location))
           (and
            (is-reachable table <robot-loc>)
            (is-on <object> table)
            (is-at robby <robot-loc>)
            (arm-empty)
            (is-open <object>)))
```

```
          (effects
           ()
           ((del (is-open <object>)))))


(INFERENCE-RULE INFER-STIRRED
                (params <ob>)
                (preconds
                 ((<ob> object))
                 (~ (to-stir <ob>)))
                (effects
                 ()
                 ((add (stirred <ob>)))))
```

The following is an example problem from the kitchen domain.

```
(setf (current-problem)
      (create-problem
       (name get-hot-water)
       (objects )
       (state
        (and
         (is-on tea-box shelf)
         (is-on instant-coffee-jar shelf)
         (is-on coffee-jar shelf)
         (is-on honey-jar shelf)
         (is-on sugar-box shelf)

         (contains cup1 nothing)
         (contains cup2 nothing)
         (contains cup3 nothing)
         (contains glass1 nothing)
         (contains glass2 nothing)
         (contains glass3 nothing)
         (contains kettle1 nothing)

         (next-to at-sink at-table)
         (next-to at-table at-sink)
```

```
(next-to at-table at-stove)
(next-to at-stove at-table)
(next-to at-stove at-fridge)
(next-to at-fridge at-stove)

(is-reachable sink at-sink)
(is-reachable garbage-can at-sink)
(is-reachable shelf at-sink)
(is-reachable cupboard at-table)
(is-reachable drawer at-table)
(is-reachable stove at-stove)
(is-reachable microwave at-stove)
(is-reachable fridge at-fridge)

(is-in scissors drawer)
(is-in knife drawer)
(is-in spoon drawer)

(is-in cup1 cupboard)
(is-in cup2 cupboard)
(is-in cup3 cupboard)

(is-in glass1 cupboard)
(is-in glass2 cupboard)
(is-in glass3 cupboard)

(is-on kettle1 stove)

(is-on coffee-maker table)
(is-in coffee-can coffee-maker)
(contains coffee-can nothing)

(is-at robby at-table)

(is-in milk-carton fridge)

(arm-empty)
(sink-empty)
(stove-empty)
(microwave-empty)
```

```
    (coffee-maker-empty)))

  (goal (and (contains cup1 hot-tea))
             (contains cup1 sugar))
))
```

D.3 Empirical results in the kitchen domain

This section summarizes the results of the experiments in the kitchen domain. The table contains the following columns:

- **Prob** is the problem number.
- **Nodes** is the total number of nodes expanded during the search. A maximum limit of $15,000$ nodes was used in all tests. Because the planner synchronizes only after certain node types, there is a chance that slightly more nodes are expanded.
- **Time** is the total CPU time used. There was a time limit of 600 CPU seconds imposed. However, in these experiments the node limit was the determining factor, since it was exceeded first.
- **Len** is the number of primitive operators in the solution. It is 0 if no solution exist or no solution was found within the resource limit.

Prob	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p1	282	3.42	13	451	6.86	0	307	4.21	16
p2	382	5.23	16	374	4.62	11	234	3.1	18
p3	599	6.97	17	310	4.02	11	262	3.66	19
p4	420	5.03	0	335	4.35	0	15146	223.87	0
p5	15001	206.77	10	337	3.87	13	294	3.9	8
p6	546	6.50	15	464	7.60	12	361	4.9	10
p7	339	3.88	9	416	6.02	13	162	2.09	10
p8	497	6.25	14	435	7.05	10	419	6.49	17
p9	579	7.25	10	476	8.13	0	226	3.48	13
p10	399	4.62	9	398	4.90	0	181	2.36	9
p11	443	5.50	11	266	3.44	11	329	4.64	13
p12	668	8.63	8	394	6.35	10	199	2.95	19
p13	540	6.40	14	232	2.73	0	445	6.2	12
p14	471	5.43	15	468	6.85	9	214	2.81	14
p15	685	8.17	11	263	4.23	13	306	4.13	11
p16	548	6.78	15	358	4.57	15	270	3.86	14
p17	411	5.02	16	287	4.39	0	272	4.14	16
p18	638	8.72	0	349	5.69	12	272	4	12
p19	15001	185.51	13	258	3.95	16	160	2.15	15
p20	505	6.76	12	298	4.36	0	435	6.18	11
p21	473	5.71	19	438	6.81	0	454	5.94	14
p22	741	9.05	15	566	8.43	11	15100	211.03	0
p23	348	4.54	18	284	4.53	8	294	4.17	11
p24	708	8.45	21	442	6.99	9	585	7.73	18
p25	354	4.46	0	414	5.96	0	446	5.82	11
p26	15001	183.10	11	374	4.29	12	435	6.61	12
p27	583	7.40	20	543	6.19	10	275	3.56	14
p28	477	6.54	14	356	4.62	11	585	8.18	15
p29	510	6.23	16	369	4.53	0	227	2.96	14
p30	677	8.48	16	357	4.36	9	537	8.07	18
p31	627	8.05	0	577	8.85	0	476	6.45	15
p32	15001	199.40	22	361	5.06	10	262	3.85	14
p33	511	5.98	16	432	5.51	12	233	3.56	11
p34	599	7.85	17	252	3.74	11	505	7.13	13
p35	723	9.45	15	279	3.35	9	596	8.8	13
p36	708	9.79	13	374	4.54	0	572	8.74	18
p37	705	9.04	12	512	7.05	0	295	4.52	17
p38	743	9.46	19	413	6.20	0	557	8.39	13
p39	388	4.47	15	235	2.87	0	256	3.97	16
p40	459	5.45	11	350	5.24	13	527	7.5	11
p41	497	6.34	21	360	5.96	8	300	4.24	11
p42	745	8.81	13	353	5.51	0	235	3.03	16
p43	571	6.83	14	316	3.85	0	446	5.88	12
p44	438	5.02	10	497	7.77	10	15263	230.04	0
p45	578	7.84	0	396	5.61	13	598	7.88	14
p46	15001	191.34	19	299	4.82	0	232	3.45	10
p47	361	4.28	18	333	5.69	0	341	4.45	16
p48	540	7.32	17	459	6.19	11	512	6.74	9
p49	655	8.64	22	344	3.93	11	385	5.23	9
p50	619	8.32	20	426	6.42	22	219	3.3	29

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p51	1076	14.46	0	1172	15.66	0	349	5.05	24
p52	15001	199.81	0	550	6.59	0	371	4.87	34
p53	15001	187.12	28	593	8.37	21	359	4.68	21
p54	1441	18.90	0	1035	14.29	20	641	9.22	20
p55	15001	215.51	33	912	12.31	0	477	7.12	38
p56	739	8.53	0	1227	19.65	21	672	10.26	26
p57	15001	199.08	0	845	12.68	0	15263	211.24	0
p58	15001	223.09	25	930	11.63	0	878	12.44	27
p59	1363	17.11	18	840	13.57	0	510	7.53	23
p60	668	8.64	0	625	9.45	22	633	8.48	29
p61	15001	190.70	0	509	8.07	20	638	8.31	31
p62	15001	223.58	0	1020	12.54	0	691	9.02	20
p63	15001	218.39	0	506	7.91	0	732	9.81	18
p64	15001	215.52	0	944	14.75	0	492	7.03	30
p65	15001	224.60	24	835	12.42	18	746	11.41	21
p66	735	9.37	36	884	11.72	0	815	11.79	33
p67	1002	11.45	0	824	10.48	0	573	8.12	22
p68	15001	212.53	24	609	7.44	22	329	4.68	25
p69	1326	17.35	0	538	8.56	26	690	10.74	27
p70	15001	223.29	0	502	6.81	17	325	4.67	27
p71	15001	222.03	0	905	15.19	0	304	4.34	36
p72	15001	176.05	0	1134	17.86	31	755	11.09	35
p73	15001	214.21	29	502	8.40	0	643	8.87	30
p74	1317	16.49	0	685	10.48	0	516	6.71	28
p75	15001	219.46	0	1165	18.57	0	768	10.22	23
p76	15001	184.09	0	502	8.16	26	565	8.68	38
p77	15001	180.75	20	573	8.31	0	888	11.89	26
p78	1448	17.58	22	803	12.11	17	15099	203.74	0
p79	1353	16.13	0	949	13.63	0	765	11.46	28
p80	15001	209.05	33	607	7.47	22	540	8.26	20
p81	1012	12.46	0	895	13.72	0	775	11.21	28
p82	15001	200.63	0	701	9.09	25	712	10.05	26
p83	15001	212.30	22	1057	15.74	0	753	10.01	25
p84	660	7.84	0	1060	12.82	22	15272	216.69	0
p85	15001	177.62	18	980	11.59	22	15116	209.59	0
p86	1449	18.94	0	1136	18.11	0	476	6.22	24
p87	15001	186.73	24	842	10.02	17	734	10.17	23
p88	759	9.63	26	570	9.24	0	418	6.12	21
p89	1043	12.54	0	921	12.57	0	462	6.13	37
p90	15001	215.54	0	1161	17.22	0	445	6.63	19
p91	15001	200.18	23	937	11.90	0	517	7.46	31
p92	1483	18.52	44	1234	19.87	23	301	4.06	30
p93	724	10.06	0	614	7.32	20	823	12.09	24
p94	15001	220.69	0	732	9.89	0	542	6.98	30
p95	15001	175.14	0	1342	20.10	0	556	7.72	27
p96	15001	192.68	26	691	10.07	0	487	7.03	24
p97	891	11.50	0	775	11.97	21	704	10.22	29
p98	15001	176.84	42	561	9.26	0	852	12.6	21
p99	985	12.30	0	562	6.89	0	770	12.01	28
p100	15001	216.21	0	727	11.25	0	368	5	34

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p101	15001	217.48	0	539	7.99	0	940	14.08	44
p102	15001	223.23	0	587	8.27	0	1865	26.8	54
p103	15001	206.08	0	1898	22.15	0	1426	20.14	47
p104	15001	199.85	0	1696	20.08	0	1463	18.8	58
p105	15001	197.61	0	1333	20.31	0	1085	15.38	42
p106	15001	193.59	37	1032	12.57	51	1999	26.18	40
p107	1868	25.92	0	2157	25.41	0	2183	29.25	45
p108	15001	196.26	52	15001	183.09	0	1046	14.78	48
p109	1164	14.91	0	2498	33.06	0	1071	16.33	34
p110	15001	205.81	0	1592	18.24	0	15213	241.23	0
p111	15001	184.23	0	1221	17.66	57	879	13.46	56
p112	15001	216.71	0	991	15.93	0	2094	27.59	50
p113	15001	202.14	76	1237	15.75	0	2184	32.41	79
p114	2994	39.65	0	1420	17.07	0	1600	21.6	48
p115	15001	222.97	0	1272	21.33	0	1176	16.21	62
p116	15001	222.88	60	15001	187.80	0	1440	18.72	46
p117	1057	14.63	59	15001	159.02	0	2087	28.47	46
p118	2301	31.82	0	15001	170.43	0	1402	19.94	81
p119	15001	183.74	0	1072	14.00	0	1352	20.95	79
p120	15001	216.22	0	2901	40.37	0	15209	234.24	0
p121	15001	178.28	0	774	12.63	0	1494	22.73	74
p122	15001	189.14	0	868	14.60	0	1663	21.96	46
p123	15001	184.41	0	1212	17.10	0	1752	26.69	42
p124	15001	185.52	0	1118	13.47	0	1721	24.04	62
p125	15001	200.35	0	15001	168.90	0	2245	31.07	76
p126	15001	175.88	0	1275	21.19	0	1066	14.31	57
p127	15001	185.42	0	1318	20.09	0	1334	18.81	58
p128	15001	214.90	0	1179	19.95	48	15242	207.01	0
p129	15001	184.64	0	1253	17.13	51	1017	13.11	66
p130	15001	197.15	49	580	9.05	42	2135	32.91	70
p131	2557	33.26	0	1208	18.90	0	1092	16.85	34
p132	15001	216.49	0	2000	30.66	0	1683	22.07	37
p133	15001	223.92	73	1424	19.31	0	15133	201.83	0
p134	1071	14.09	0	2249	30.41	0	1781	25.48	58
p135	15001	207.36	0	2200	33.09	0	772	11.78	51
p136	15001	209.29	0	1236	17.88	0	823	12.17	54
p137	15001	178.91	0	2129	25.18	0	1503	19.72	81
p138	15001	187.30	0	1894	28.20	0	1133	14.52	79
p139	15001	206.66	0	1622	18.89	0	993	14.32	58
p140	15001	210.43	0	15001	156.04	0	1732	23.16	44
p141	15001	177.51	0	1737	20.40	0	1021	14.61	42
p142	15001	222.58	0	1587	23.38	38	1262	16.88	61
p143	15001	185.27	0	2106	24.49	0	1415	18.28	71
p144	15001	220.85	0	2340	28.30	0	15213	198.2	0
p145	15001	185.85	0	1045	12.52	0	1225	15.74	54
p146	15001	203.03	0	1011	12.02	0	1205	18.43	64
p147	15001	183.70	0	15001	176.51	0	1478	21.45	77
p148	15001	183.96	0	1716	19.91	44	1930	26.96	50
p149	15001	175.34	0	1530	22.83	0	1085	16.14	62
p150	15001	208.92	0	15001	161.52	0	883	13.13	119

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p151	15001	183.21	0	3830	52.52	0	3243	48.64	120
p152	15001	212.23	0	3358	42.05	0	15081	233.17	0
p153	15001	197.72	0	2171	24.78	0	3422	46.4	105
p154	15001	188.86	0	2286	34.24	86	1873	26.15	121
p155	15001	191.27	0	3880	58.14	0	2914	41.3	140
p156	15001	190.00	0	15001	190.00	0	3392	49.88	74
p157	15001	175.95	0	15001	190.00	0	2815	41.83	106
p158	15001	223.71	0	15001	190.00	0	3170	44.53	92
p159	15001	195.38	0	15001	167.25	0	2645	36.34	118
p160	15001	222.87	0	3167	51.51	0	2705	35.64	118
p161	15001	192.85	0	2866	41.05	100	1308	20.22	81
p162	15001	217.79	0	2126	30.81	82	1294	16.91	73
p163	15001	175.32	165	3756	45.77	0	2774	41.35	108
p164	5033	67.38	0	15001	196.38	0	1798	24.75	115
p165	15001	209.50	0	2992	36.51	0	1724	22.77	112
p166	15001	189.44	0	2584	36.85	0	3460	46.82	123
p167	15001	186.21	0	15001	152.31	0	2514	33.7	132
p168	15001	180.62	0	3430	55.54	0	1205	16.09	104
p169	15001	205.65	0	4618	55.02	0	3009	39.21	125
p170	15001	188.95	0	5091	62.85	0	2091	28.98	149
p171	15001	181.54	0	15001	151.57	0	1925	26.54	89
p172	15001	178.43	0	5689	89.26	0	15036	232.54	0
p173	15001	202.00	0	2716	35.00	0	1737	22.71	124
p174	15001	206.87	0	15001	168.23	0	3412	52.64	97
p175	15001	199.69	0	15001	167.94	0	2868	39.45	139
p176	15001	190.17	0	4380	60.67	0	2597	36.52	131
p177	15001	216.55	0	15001	164.01	0	15066	200.17	0
p178	15001	194.66	0	2051	30.65	0	2104	30.13	104
p179	15001	190.17	0	15001	156.33	0	1218	18.09	85
p180	15001	176.69	0	15001	188.16	0	2955	41.19	139
p181	15001	213.16	0	15001	174.05	0	2719	35.69	106
p182	15001	222.71	0	2127	27.14	0	2289	30.98	90
p183	15001	212.37	0	15001	167.89	0	3162	46.13	122
p184	15001	180.32	0	4590	61.33	0	2186	28.11	122
p185	15001	204.09	0	4240	48.85	0	2824	42.8	95
p186	15001	211.41	0	2302	38.26	0	3218	47.99	111
p187	15001	198.65	0	15001	167.26	0	2390	34.74	84
p188	15001	198.59	0	15001	157.04	0	15029	196.64	0
p189	15001	180.46	0	2087	28.59	0	3465	48.89	104
p190	15001	202.45	0	15001	154.99	0	1740	25.3	95
p191	15001	224.45	0	4017	51.09	0	2191	30.16	81
p192	15001	200.40	0	4555	55.67	0	2192	28.71	116
p193	15001	175.50	0	4816	60.76	0	2996	42.5	111
p194	15001	208.75	0	15001	179.37	0	2565	33.98	150
p195	15001	195.62	0	3902	58.89	0	3317	46.02	84
p196	15001	211.48	0	3769	54.05	0	3422	52.64	103
p197	15001	192.82	0	4349	64.10	0	2067	31.66	89
p198	15001	199.36	0	15001	179.47	0	2082	30.68	119
p199	15001	179.05	0	15001	191.02	0	1318	19.39	98
p200	15001	200.00	0	15001	193.83	0	2383	34.47	90

Prob.	PRODIGY-DL			PC-MSP-O			DoLittle		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p201	15001	215.68	0	8956	148.82	0	6841	106.82	130
p202	15001	178.85	205	9425	122.20	0	5257	79.98	212
p203	3554	40.71	0	15001	156.72	0	4249	57.34	184
p204	15001	223.74	0	15001	157.03	0	5189	75.6	127
p205	15001	206.35	0	15001	181.15	0	6881	93.29	94
p206	15001	199.15	0	15001	194.32	0	2874	38.94	108
p207	15001	180.81	0	15001	179.37	0	6809	93.1	123
p208	15001	176.44	0	15001	168.33	0	5794	76.85	99
p209	15001	215.69	0	9752	116.86	0	6041	83.25	108
p210	15001	189.82	0	15001	169.65	0	3358	46.14	156
p211	15001	216.45	0	6384	90.16	127	6217	92.12	102
p212	15001	187.86	0	9463	122.16	0	3870	54.51	179
p213	15001	193.24	0	4992	76.16	0	6753	102.63	137
p214	15001	209.44	0	15001	194.43	0	5868	86.24	90
p215	15001	210.16	0	15001	199.35	0	7413	102.78	132
p216	15001	181.91	0	15001	182.99	0	6420	99.82	93
p217	15001	214.39	0	15001	197.91	0	4049	53.52	115
p218	15001	190.38	0	6261	88.56	0	2750	39.03	165
p219	15001	196.04	0	7191	113.95	0	3044	41.64	143
p220	15001	208.37	0	15001	167.58	0	5750	87.27	121
p221	15001	196.50	0	15001	179.05	0	15169	208.4	0
p222	15001	208.25	0	15001	172.56	0	15236	205.56	0
p223	15001	192.44	0	8850	121.92	0	4463	57.9	116
p224	15001	185.85	0	6484	79.54	0	2719	41.45	114
p225	15001	175.11	0	9480	134.51	0	3089	41.19	83
p226	15001	176.51	0	15001	183.72	0	3205	44.47	133
p227	15001	221.55	0	4945	73.69	0	3920	55.5	186
p228	15001	186.78	0	4870	74.64	0	3546	51.73	123
p229	15001	199.12	0	7624	95.34	0	3841	57.48	110
p230	15001	217.58	0	15001	195.38	0	4998	73	115
p231	15001	199.39	0	9297	148.78	0	2823	39.16	102
p232	15001	190.46	0	15001	180.87	0	3570	54.49	133
p233	15001	185.28	0	6736	92.20	0	15296	235.01	0
p234	15001	191.14	0	15001	167.53	0	15130	233.24	0
p235	15001	202.35	0	15001	172.20	0	3682	56.37	75
p236	15001	198.15	0	15001	164.63	0	15085	225.98	0
p237	15001	184.95	0	5137	80.94	0	4796	74.59	85
p238	15001	222.43	0	7872	101.30	0	6164	82.33	113
p239	15001	199.02	0	15001	150.58	0	5797	74.35	152
p240	15001	181.96	0	15001	156.81	0	5628	85.83	124
p241	15001	214.35	0	15001	167.71	0	5079	78.41	138
p242	15001	200.35	0	10543	120.46	0	15301	224.86	0
p243	15001	194.13	0	15001	199.20	0	6451	94.26	87
p244	15001	202.66	0	15001	183.83	0	5650	85.85	137
p245	15001	202.74	0	8819	117.96	0	4977	70.24	164
p246	15001	185.20	0	15001	168.32	0	6639	96.58	158
p247	15001	197.85	0	15001	153.56	0	5744	78.93	135
p248	15001	223.07	0	15001	193.65	0	7296	96.48	113
p249	15001	203.22	0	15001	182.33	0	5042	68.85	147
p250	15001	182.86	0	5946	85.27	0	5708	74.21	108

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p1	537	7.03	12	15018	222.94	0	451	6.86	13
p2	374	4.62	15	578	7.95	11	595	9.28	16
p3	310	4.02	11	742	11.23	15	409	6.5	13
p4	335	4.35	13	15065	223.86	0	659	10.87	13
p5	337	3.87	18	450	6.48	13	706	11.91	18
p6	573	7.21	15	651	10.30	12	464	7.6	15
p7	541	6.62	20	416	6.02	14	747	10.66	13
p8	544	6.63	10	514	7.93	16	435	7.05	13
p9	15001	191.95	0	503	7.12	10	476	8.13	13
p10	398	4.90	16	15035	211.30	0	686	10.52	17
p11	266	3.44	12	594	8.22	11	399	6.28	12
p12	468	6.19	17	480	7.17	12	394	6.35	10
p13	232	2.73	14	15289	200.74	0	324	4.67	10
p14	547	6.29	22	703	9.60	18	468	6.85	9
p15	430	5.65	16	263	4.23	13	632	10.59	18
p16	358	4.57	17	596	8.32	18	625	10.56	15
p17	15001	150.57	0	287	4.39	12	400	6.1	14
p18	471	6.48	13	449	6.36	19	349	5.69	12
p19	493	6.02	16	258	3.95	16	549	8.74	18
p20	526	6.54	12	15270	215.48	0	298	4.36	11
p21	15001	191.89	0	704	11.49	20	438	6.81	13
p22	593	7.79	16	637	8.99	15	566	8.43	11
p23	343	4.72	20	531	7.65	10	284	4.53	8
p24	592	7.99	9	442	6.99	13	504	7.38	11
p25	575	7.28	14	414	5.96	12	15249	237.17	0
p26	374	4.29	15	423	6.12	12	446	6.86	13
p27	543	6.19	10	663	10.91	13	746	11.03	13
p28	356	4.62	11	732	11.15	13	567	8.96	14
p29	369	4.53	12	15101	227.85	0	15105	247.22	0
p30	357	4.36	9	560	7.89	12	735	11.2	14
p31	15001	198.01	0	657	10.78	15	577	8.85	16
p32	541	6.27	10	361	5.06	14	660	9.87	13
p33	432	5.51	13	585	9.11	12	454	6.42	13
p34	414	5.56	11	520	7.25	11	252	3.74	19
p35	279	3.35	12	730	10.32	12	452	7.47	9
p36	374	4.54	20	471	7.73	12	15291	243.31	0
p37	512	7.05	12	15138	248.11	0	578	8.64	13
p38	439	5.01	10	413	6.20	12	15225	254.43	0
p39	235	2.87	11	15145	202.41	0	422	6.41	9
p40	464	5.70	14	350	5.24	20	674	9.93	13
p41	596	7.52	11	360	5.96	8	714	12.2	13
p42	590	6.74	12	353	5.51	13	15185	237.03	0
p43	316	3.85	15	15023	213.29	0	573	8.31	13
p44	530	6.05	21	497	7.77	10	554	8.32	18
p45	480	5.74	14	687	10.28	13	396	5.61	15
p46	15001	194.12	0	299	4.82	15	405	6.49	16
p47	426	5.56	15	15214	214.13	0	333	5.69	15
p48	459	6.19	11	671	9.28	14	501	7.48	12
p49	344	3.93	11	399	5.43	14	369	5.43	14
p50	1109	14.16	25	921	13.98	29	426	6.42	22

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p51	1172	15.66	33	15146	248.47	0	15096	243.59	0
p52	550	6.59	26	15090	217.88	0	711	10.89	27
p53	922	12.32	46	1383	21.36	21	593	8.37	21
p54	1439	19.09	39	1035	14.29	31	1163	16.55	20
p55	912	12.31	26	1144	18.32	26	15134	219.61	0
p56	1391	17.44	21	1285	18.48	33	1227	19.65	28
p57	15001	151.03	0	1255	17.08	22	845	12.68	22
p58	930	11.63	29	15067	207.59	0	15017	236.99	0
p59	15001	168.29	0	1460	22.41	33	840	13.57	22
p60	956	11.70	33	723	9.85	32	625	9.45	22
p61	814	9.85	20	1093	15.67	26	509	8.07	28
p62	1020	12.54	16	1279	20.31	29	15052	254.92	0
p63	1423	17.56	22	15010	200.60	0	506	7.91	33
p64	1117	14.99	23	944	14.75	38	15299	222.95	0
p65	846	10.65	35	862	12.87	18	835	12.42	18
p66	884	11.72	26	1443	22.99	24	15193	225.02	0
p67	824	10.48	27	15250	226.21	0	1403	22.22	24
p68	609	7.44	22	722	11.12	29	767	12.5	35
p69	1382	16.39	33	538	8.56	26	1460	21.54	31
p70	502	6.81	36	516	8.33	17	1191	17.38	26
p71	1398	16.53	47	15127	204.98	0	905	15.19	37
p72	1192	15.33	37	1216	17.98	31	1134	17.86	31
p73	1023	12.40	23	15201	221.84	0	502	8.4	26
p74	15001	163.73	0	685	10.48	18	1393	20.46	28
p75	1349	17.13	33	1165	18.57	19	15135	241.15	0
p76	805	9.72	37	502	8.16	26	963	14.19	32
p77	15001	165.77	0	573	8.31	29	15190	231.18	0
p78	1073	14.50	20	949	14.73	26	803	12.11	17
p79	15001	186.50	0	1239	18.62	22	949	13.63	26
p80	607	7.47	22	902	13.74	25	812	13.19	23
p81	1338	15.32	18	15272	227.52	0	895	13.72	31
p82	701	9.09	25	933	13.90	25	726	10.56	31
p83	1496	19.20	25	15193	211.45	0	1057	15.74	29
p84	1060	12.82	22	1489	22.22	27	1251	18.7	24
p85	980	11.59	36	1364	20.31	22	1440	23.49	33
p86	15001	150.72	0	1136	18.11	24	15197	221.47	0
p87	842	10.02	26	877	12.46	26	1089	16.52	17
p88	15001	162.79	0	15228	229.90	0	570	9.24	30
p89	1428	17.31	28	921	12.57	19	15003	234.43	0
p90	15001	175.29	0	1161	17.22	29	1255	17.6	25
p91	937	11.90	19	15242	245.40	0	1044	14.74	30
p92	1256	16.19	23	1469	22.37	26	1234	19.87	40
p93	614	7.32	27	799	11.93	20	661	9.79	31
p94	732	9.89	27	15276	207.24	0	15143	212.18	0
p95	1348	17.12	30	15256	237.55	0	1342	20.1	20
p96	1295	17.69	23	691	10.07	22	15026	223.99	0
p97	900	10.80	35	775	11.97	21	1465	24.29	33
p98	15001	189.71	0	803	11.59	33	561	9.26	25
p99	562	6.89	21	1413	22.36	24	15199	232.57	0
p100	1901	21.85	53	15246	229.29	0	727	11.25	48

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p101	15001	153.31	0	15030	235.23	0	539	7.99	61
p102	2672	32.82	55	15154	218.20	0	587	8.27	75
p103	1898	22.15	49	15274	240.61	0	15186	209.47	0
p104	1696	20.08	50	1896	27.31	41	15256	217.69	0
p105	15001	193.39	0	2091	34.09	69	1333	20.31	52
p106	1032	12.57	79	2541	41.74	51	1189	16.65	53
p107	2157	25.41	71	15119	246.49	0	15011	212.73	0
p108	15001	183.09	0	15106	207.63	0	15251	219.57	0
p109	2498	33.06	43	15148	220.84	0	15223	220.01	0
p110	1592	18.24	44	15249	207.70	0	15075	216.16	0
p111	1507	18.87	60	2012	32.85	64	1221	17.66	57
p112	1961	22.62	44	15045	202.02	0	991	15.93	38
p113	1237	15.75	55	15265	218.08	0	15124	251.56	0
p114	1420	17.07	69	15245	211.37	0	15098	233.3	0
p115	15001	165.43	0	2781	40.20	61	1272	21.33	62
p116	15001	187.80	0	15111	201.93	0	15279	211.27	0
p117	15001	159.02	0	15199	232.25	0	15016	226.34	0
p118	15001	170.43	0	15194	210.35	0	15049	221.45	0
p119	1072	14.00	75	2326	38.05	69	15033	213.49	0
p120	15001	190.53	0	2901	40.37	66	15115	244.43	0
p121	2131	26.49	45	15002	230.52	0	774	12.63	62
p122	2579	33.03	70	15282	237.53	0	868	14.6	69
p123	15001	150.05	0	2098	30.02	38	1212	17.1	42
p124	1118	13.47	72	15172	245.85	0	1198	17.4	67
p125	15001	168.90	0	15140	248.64	0	15130	245.73	0
p126	2643	32.58	82	15235	238.68	0	1275	21.19	47
p127	15001	155.46	0	15101	231.83	0	1318	20.09	46
p128	2467	32.93	71	2385	38.78	58	1179	19.95	48
p129	1253	17.13	66	2952	43.80	55	1416	22.48	51
p130	2393	29.60	73	2344	37.42	42	580	9.05	69
p131	2983	35.37	85	15075	222.19	0	1208	18.9	47
p132	15001	172.62	0	15260	225.07	0	2000	30.66	38
p133	1424	19.31	62	2920	42.64	35	15129	240.41	0
p134	2249	30.41	96	15203	203.10	0	15080	217.56	0
p135	2377	31.09	42	15199	219.06	0	2200	33.09	38
p136	1629	18.66	57	1236	17.88	40	15020	209.18	0
p137	2129	25.18	64	15261	249.94	0	15150	241.9	0
p138	2573	31.37	68	15004	242.20	0	1894	28.2	42
p139	1622	18.89	54	15248	214.95	0	15278	237.98	0
p140	15001	156.04	0	15094	225.37	0	15009	230.58	0
p141	1737	20.40	72	15122	239.13	0	15161	225.46	0
p142	2769	33.75	42	2064	32.59	48	1587	23.38	38
p143	2106	24.49	46	2789	42.22	52	15283	205.1	0
p144	2340	28.30	82	15074	205.27	0	15052	241.7	0
p145	1045	12.52	65	15121	223.75	0	15284	237.72	0
p146	1011	12.02	57	15252	242.06	0	15215	213.83	0
p147	15001	176.51	0	15057	249.96	0	15064	206.58	0
p148	1716	19.91	83	2618	43.00	44	2552	36.05	50
p149	15001	174.72	0	1530	22.83	52	15075	229.66	0
p150	15001	161.52	0	15118	217.53	0	15099	216.24	0

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p151	5821	80.04	167	3830	52.52	105	15127	227.16	0
p152	3358	42.05	88	4817	68.85	122	15047	240.39	0
p153	2171	24.78	128	4880	73.74	90	15154	245.78	0
p154	3927	49.15	146	3624	52.82	86	2286	34.24	105
p155	15001	196.83	0	15145	231.64	0	3880	58.14	78
p156	15001	190.00	0	15071	211.47	0	15261	229.3	0
p157	15001	190.00	0	15222	221.85	0	15114	222.17	0
p158	15001	190.00	0	15037	216.65	0	15197	243.49	0
p159	15001	167.25	0	15217	249.64	0	15155	212.28	0
p160	15001	190.00	0	3167	51.51	102	15265	244.75	0
p161	3457	40.61	100	5057	71.69	109	2866	41.05	110
p162	4432	57.49	82	2126	30.81	87	5448	87.8	103
p163	3756	45.77	115	15094	229.42	0	15075	248.5	0
p164	15001	196.38	0	15155	213.12	0	15175	239.07	0
p165	2992	36.51	84	15138	206.42	0	15060	206.47	0
p166	15001	167.72	0	2584	36.85	120	15280	206.5	0
p167	15001	152.31	0	15243	228.54	0	15205	244.92	0
p168	15001	193.18	0	3430	55.54	90	15012	219.38	0
p169	4618	55.02	87	15132	240.19	0	15159	236.6	0
p170	5091	62.85	108	15230	231.57	0	15227	242.15	0
p171	15001	151.57	0	15218	243.47	0	15214	231.04	0
p172	15001	164.70	0	5689	89.26	128	15146	251.81	0
p173	2716	35.00	118	15247	243.46	0	4114	59.69	92
p174	15001	168.23	0	15123	236.61	0	15168	226.77	0
p175	15001	167.94	0	15203	217.13	0	15224	239.37	0
p176	4380	60.67	104	15060	241.17	0	15144	254.69	0
p177	15001	164.01	0	15092	236.15	0	15162	235.39	0
p178	15001	158.52	0	15284	234.27	0	2051	30.65	150
p179	15001	156.33	0	15025	229.53	0	15213	246.2	0
p180	15001	188.16	0	15264	210.57	0	15058	243.23	0
p181	15001	174.05	0	15162	243.55	0	15234	250.15	0
p182	2127	27.14	156	15187	248.10	0	3044	44.79	115
p183	15001	167.89	0	15165	218.95	0	15189	247.12	0
p184	4590	61.33	104	15075	249.71	0	15227	228.05	0
p185	4240	48.85	154	15199	212.66	0	15231	231.73	0
p186	4143	49.16	119	15166	201.43	0	2302	38.26	109
p187	15001	167.26	0	15151	249.99	0	15220	234.6	0
p188	15001	157.04	0	15212	240.51	0	15147	225.71	0
p189	2087	28.59	150	15109	237.85	0	15192	224.53	0
p190	15001	154.99	0	15184	216.84	0	15059	213.5	0
p191	4017	51.09	103	15245	207.00	0	15149	226.05	0
p192	4555	55.67	77	5106	72.55	93	15100	247.18	0
p193	4816	60.76	91	15139	243.84	0	15218	250.8	0
p194	15001	179.37	0	15097	238.50	0	15267	222.94	0
p195	15001	190.22	0	3902	58.89	93	15167	225.76	0
p196	15001	159.35	0	15220	207.34	0	3769	54.05	126
p197	15001	186.82	0	15092	231.01	0	4349	64.1	138
p198	15001	179.47	0	15140	216.70	0	15194	238.15	0
p199	15001	191.02	0	15159	246.95	0	15076	226.64	0
p200	15001	193.83	0	15280	245.23	0	15241	211	0

Prob	Case-L			Macro-L			Abstract-L		
Num.	Nodes	Time	Len	Nodes	Time	Len	Nodes	Time	Len
p201	15001	160.60	0	15236	227.76	0	8956	148.82	118
p202	9425	122.20	187	15100	237.04	0	15270	217.22	0
p203	15001	156.72	0	15025	234.23	0	15067	253.45	0
p204	15001	157.03	0	15048	245.99	0	15255	248.99	0
p205	15001	181.15	0	15176	232.04	0	15053	252.31	0
p206	15001	194.32	0	15273	213.44	0	15049	240.79	0
p207	15001	179.37	0	15233	214.82	0	15116	209.77	0
p208	15001	168.33	0	15059	213.90	0	15205	234.15	0
p209	9752	116.86	121	15068	241.52	0	15173	217	0
p210	15001	169.65	0	15017	223.74	0	15124	214.61	0
p211	10279	139.43	153	6384	90.16	190	9572	141.23	127
p212	9463	122.16	84	15152	215.08	0	15234	247.55	0
p213	15001	199.53	0	4992	76.16	166	15200	249.6	0
p214	15001	194.43	0	15186	210.44	0	15264	225.14	0
p215	15001	199.35	0	15132	213.42	0	15007	233.43	0
p216	15001	182.99	0	15081	215.85	0	15201	231.05	0
p217	15001	197.91	0	15299	231.19	0	15140	241.93	0
p218	10337	141.19	206	6261	88.56	177	15242	228.95	0
p219	15001	188.16	0	15109	223.12	0	7191	113.95	129
p220	15001	167.58	0	15134	236.47	0	15281	206.61	0
p221	15001	179.05	0	15220	211.06	0	15261	240.16	0
p222	15001	172.56	0	15056	206.33	0	15095	237.19	0
p223	8850	121.92	243	15131	219.49	0	15145	249.21	0
p224	6484	79.54	120	15040	245.46	0	15262	222.53	0
p225	15001	170.56	0	15207	246.74	0	9480	134.51	138
p226	15001	183.72	0	15187	233.65	0	15220	226.64	0
p227	15001	177.28	0	4945	73.69	172	15014	226.17	0
p228	15001	199.76	0	4870	74.64	162	15052	228.47	0
p229	7624	95.34	217	15129	224.12	0	15223	243.68	0
p230	15001	195.38	0	15002	224.21	0	15177	216.4	0
p231	15001	160.57	0	15107	209.47	0	9297	148.78	111
p232	15001	180.87	0	15296	200.18	0	15149	212.87	0
p233	6736	92.20	105	15270	214.87	0	15008	245.29	0
p234	15001	167.53	0	15028	200.64	0	15075	221.81	0
p235	15001	172.20	0	15007	238.57	0	15163	225.26	0
p236	15001	164.63	0	15169	235.35	0	15097	208.09	0
p237	15001	151.43	0	15031	221.03	0	5137	80.94	130
p238	7872	101.30	156	15128	236.11	0	15171	251.4	0
p239	15001	150.58	0	15229	202.36	0	15242	237.56	0
p240	15001	156.81	0	15029	212.12	0	15105	221.04	0
p241	15001	167.71	0	15138	228.72	0	15124	209.04	0
p242	10543	120.46	90	15160	205.35	0	15159	225.79	0
p243	15001	199.20	0	15148	224.72	0	15198	218.64	0
p244	15001	183.83	0	15088	229.37	0	15069	210.6	0
p245	8819	117.96	224	15125	226.01	0	15083	249.75	0
p246	15001	168.32	0	15069	236.96	0	15157	210.11	0
p247	15001	153.56	0	15039	216.24	0	15077	244.11	0
p248	15001	193.65	0	15172	244.67	0	15156	214.91	0
p249	15001	182.33	0	15290	211.89	0	15073	218.48	0
p250	15001	169.31	0	15258	201.44	0	5946	85.27	195