

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

**A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600**



Université d'Ottawa • University of Ottawa

**A Planning System Based on Plan Re-Use
and
Its Application to Geographical Information Systems and
Remote Sensing**

by

Daniel Charlebois

**A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree**

Ph.D. of Computer Science

**Ottawa-Carleton Institute of Computer Science
Department of Computer Science
Faculty of Science
University of Ottawa**

January 1997

© 1997, Daniel Charlebois, Ottawa, Canada



**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced with the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-20994-6

Abstract

This dissertation integrates the use of both transformational and derivational analogy into a general problem solving system. As the system faces new problems, cases are retrieved, adapted and subsequently generalized in order to enhance its performance. Empirical results show that the system can successfully address problems in simple domains and scales up smoothly to create solutions to problems in more complex domains including the management and processing of remote sensing and geographic information systems data for natural resource applications.

Most case-based reasoning systems rely on sophisticated indexing schemes and adaptation rules to find solutions to new problems. As a result, they expend considerable effort in retrieving and adapting cases to new problems. The approach presented in this dissertation introduces the use of generalization to case-based reasoning. Once a case has been retrieved and adapted to a new problem, the system will generalize the old case with the new case by using an algorithm similar to least general generalization. As the system gains experience, the case-base is generalized and, as is shown by the experimental results, the number of cases required to solve problems is significantly reduced.

To show that the approach scales up to real world problems, the system, dubbed PALERMO (Planning and LEarning for Resource Management and Organization), has been implemented and integrated into the SEIDAM environment. SEIDAM (System of Experts for Intelligent DATA Management) is a complex system that uses several AI approaches to manage large quantities of remote sensing and geographic data. It draws on expert system technology, software agents and case-based reasoning to gather and process remote sensing and digital geographic data. One of the goals of SEIDAM is to use remote sensing data to update digital forest cover maps to assist in land use decision making.

Keywords: analogical reasoning, case-based reasoning, derivational analogy, transformational analogy, planning, general problem solving, machine learning, remote sensing, geographical information systems (GIS)

Acknowledgments

There are several people to whom I must extend my deepest appreciation. The first is my advisor Dr. Stan Matwin. Believe it or not, Stan has endured a decade of me. It all began in 1986 when I started my Master of Computer Science degree. We had a few ideas that, for some reason, we thought would make it interesting for us to work together. Little did we know that I would eventually be drawn to the Ph.D. program where many things were left for us to explore. We've seen each other go through small personal crises as well as more difficult ones. To him, I am grateful for the continuous support, not to mention patience, through thick and thin.

Then, about half way through my association with Stan, we met Dr. David Goodenough. At the time, David was at the Canada Centre for Remote Sensing in Ottawa and needed someone to develop tools that would allow a Prolog interpreter to connect to an XWindows interface (at the time, there did not exist such a beast). As luck would have it, David and I hit it off. We became friends and colleagues. During the development of the Prolog/XWindows interface, we came up with the ideas that eventually gave birth to the work presented in this dissertation. During our association in Ottawa, David suddenly departed for the more moderate climate of Victoria, British Columbia, to work at the Pacific Forestry Centre. Unwilling to end our collaboration, David offered me the opportunity to come to Victoria to complete the research we had begun. Throughout, we have been involved in many discussions that have helped me mold my ideas into something that could handle very difficult real problems. I have learned many things from David and owe him great thanks.

Several people have come and gone at the forestry centre: coop students, contractors and other scientists. All have provided me with insight at one time or another as to how I could improve my work. To them, thank you for the fun we've had in the lab!

Colleagues from work and university obviously have a stake in my work, but much of the real support, that I think each one of us needs, came from my friends and family. At one point in time, my friends considered me insufferable. Graduate studies had gone to my head! They learned to deal with me and eventually calmed my insecurities. This allowed me to deal much better with my work and helped provide focus and determination to finally finish what I had begun.

My family has also provided me with incredible support. When I began my university studies, all I wanted was a bachelor's degree. I've gone much further than I had planned, and throughout the 15 years that it has taken to finally reach the end of my

student career, the one constant in my universe was my family. They've always been there for me and I greatly appreciate their love. I'm speechless!

Finally, thank you Nico for being there for me. Your love and support has helped me through the final hurdles of producing this dissertation.

In the immortal words of Winnie the Pooh's friend Tigger, TTFN (ta! ta! for now)!!!

Table of contents

Table of contents	i
Tables and figures	iii
Glossary	v
Abbreviations and acronyms	vi
 Chapter 1: Introduction	 1
1.1 Goal Regression, Derivational Analogy and CBR	3
1.2 Definitions	3
1.3 Problem Solving with Agents	7
1.4 Contributions and Overview	9
Chapter 2: Literature Review	11
2.1 Geographical Information Systems and STRIPS	12
2.2.1 Geographical Information Systems	13
2.2.2 STRIPS	13
2.2 Partial Orderings on Plans	16
2.3 Derivational Analogy	22
2.4 Transformational Analogy	27
2.5 Operator Ordering and Merging	30
2.6 Least General Generalization	33
2.7 Other work	34
Chapter 3: The Problem Solver	36
3.1 The Search Algorithm	39
3.1.1 Formal definition of the search algorithm	40
3.1.2 Example of the application of the search algorithm ...	42
3.2 Case-based reasoning in goal regression	44
3.2.1 Definition of the retrieve algorithm	45
3.2.1.1 Goal directed generalization	47
3.2.1.2 Definition of goal directed generalization	48
3.2.2 Example of the application of the retrieve algorithm .	51
3.2.3 Definition of the short replay algorithm	54
3.2.4 Example of the application of the short replay algorithm	55
3.3 Merging plans	57
3.3.1 Definition of the merge algorithm	58
3.3.2 Example of the application of the merge algorithm	61
3.4 Plan repair	64
3.4.1 Definition of plan repair	65
3.4.2 Example of plan repair	66
Chapter 4: Empirical Results	69
4.1 Achieving Non-Linearity	70
4.2 Scaling up - The logistics transportation domain	75
4.2.1 Generating problems	75
4.2.2 Rate of accumulation of one conjunct goal statement problems	75
4.2.3 Analogy versus no analogy	77
4.2.4 Generalization versus no generalization	79
4.2.5 Learning from multiple conjunct goal statements	81
Chapter 5: SEIDAM: The Application	86
5.1 The problem domain	87
5.2 Problem solving in SEIDAM	88
5.3 SEIDAM: An example	91
5.4 SEIDAM agents: An overview	98

Chapter 6: Conclusion	100
6.1 System components	100
6.2 Results and contributions	101
6.3 Future work	103
References	105
Appendix A: One-way rocket and extended STRIPS domain operators ..	110
Appendix B: Goal directed generalization: a detailed example	114
Appendix C: Empirical results	119
Appendix D: SEIDAM planning operators	160

Tables and Figures

Table 4.1.1	74
Table 5.2.1	90
Figure 1.1 - The one-way rocket domain	3
Figure 1.2 - Initial world and goal statement	4
Figure 1.3 - Solution for conjunct <i>at(obj1, loc2)</i>	4
Figure 1.4 - Solution for the goal statement <i>at(obj1, loc2)</i> and <i>at(obj2, loc2)</i>	6
Figure 1.5 - Solution for conjunct <i>at(obj2, loc2)</i>	7
Figure 2.1 - NOAH planning	19
Figure 2.2 - NOAH planning	20
Figure 2.3 - NOAH planning	20
Figure 2.4 - NOAH planning	20
Figure 2.5 - NOAH planning	21
Figure 2.6 - NOAH planning	21
Figure 2.7 - Merging plans and operators	32
Figure 3.1.1 - Search procedure	41
Figure 3.1.2 - One-way rocket reasoning trace	43
Figure 3.2.1 - One-way rocket solution for goal <i>at(obj1, loc2)</i>	44
Figure 3.2.2 - Search procedure	45
Figure 3.2.3 - Retrieval procedure	46
Figure 3.2.4 - Generalized one-way rocket case	47
Figure 3.2.5 - Goal directed generalization	49
Figure 3.2.6 - Substitution computation procedure	51
Figure 3.2.7 - Modified one-way rocket	52
Figure 3.2.8 - Adapted case	53
Figure 3.2.9 - Adapted case	53
Figure 3.2.10 - Generalized case	54
Figure 3.2.11 - Short replay procedure	55
Figure 3.2.12 - New one-way rocket specification	56
Figure 3.2.13 - Case retrieved for goal <i>at(obj(2), loc(2))</i>	57
Figure 3.3.1 - Merging non-empty plans	60
Figure 3.3.2 - Two cases to merge	61
Figure 3.3.3 - Merge: <i>load_rocket(obj2, loc1)</i> is added to the current plan	62
Figure 3.3.4 - Current planning state after adding <i>load_rocket(obj2, loc1)</i> to the plan	62
Figure 3.3.5 - Merge: <i>load_rocket(obj1, loc1)</i> is added to the current plan	63
Figure 3.3.6 - Current planning state after adding <i>load_rocket(obj1, loc1)</i> to the plan	63
Figure 3.3.7 - Merge: <i>move_rocket</i> is redundant and discarded	64
Figure 3.3.8 - Current planning state after discarding <i>move_rocket</i> to the plan	64
Figure 3.3.9 - Current planning state after adding <i>move_rocket</i> to the plan	64
Figure 3.4.1 - Merging and plan repair	65
Figure 3.4.2 - Two cases from the extended STRIPS domain	66
Figure 3.4.3 - The current world after adding the operators necessary for the robot to reach room <i>room(3)</i>	67
Figure 3.4.4 - The current planning state	67
Figure 3.4.5 - The current planning state	68
Figure 3.4.6 - Plan repair	68
Figure 4.1.1 - Reasoning trace for the one-way rocket problem while using analogy	71
Figure 4.1.2 - Reasoning trace for the one-way rocket	

problem without using analogy	72
Figure 4.1.3 - Reasoning trace for the one-way rocket problem while using analogy and a non-empty generalized case-base	73
Figure 4.2.1 - Automatically generated initial world (118 literals) ...	76
Figure 4.2.2 - Accumulation of cases for one conjunct goal statement problems	77
Figure 4.2.3 - Average solve time per problem for 1000 problems	78
Figure 4.2.4 - Average number of search node per problem	78
Figure 4.2.5 - Accumulation of cases while using generalization versus accumulation of cases without using generalization	79
Figure 4.2.6 - Adaptation effort	80
Figure 4.2.7 - Average solve time per problem	80
Figure 4.2.8 - Accumulation of cases for multiple conjunct goal statement problems	81
Figure 4.2.9 - Use of cases to solve goal statement conjuncts	82
Figure 4.2.10 - Adaptation effort	82
Figure 4.2.11 - Adaptation effort	83
Figure 4.2.12 - Average time to solve one conjunct	84
Figure 4.2.13 - Average merge time per problem	84
Figure 4.2.14 - Average retrieval time per goal statement conjunct	85
Figure 5.2.1 - SEIDAM Architecture	89
Figure 5.3.1 - Integration of LANDSAT TM imagery and GIS forest cover resulting from execution of plan	94-97
Figure 5.4.1 - Agent architecture	98

Index

agents	2, 7, 87, 98, 102
case	2, 22, 27, 44
case based reasoning	2, 27, 37, 44
derivational analogy	1, 12, 22, 37, 54, 69, 101
eager replay	34, 103
eager reuse	9, 69, 102
explanation based generalization	9, 48
explanation based learning	48
footprint	46
geographical information system	8, 12
goal directed generalization	47
goal regression	3, 13, 37, 39, 70, 100
least general generalization	33, 47, 49, 101
linear planning	3, 9, 16, 17, 36, 101
merging strategy	58
noah	4, 11, 18, 36
nolimit	4, 23, 73
non-linear planning	3, 4, 18, 37, 70, 101
operator merging	12, 30, 31, 32, 104
partial order planning	3, 16, 30, 103
plan based state representation	36
plan merging	7, 30, 37, 45, 57, 65, 84, 100
plan repair	2, 39, 64, 82, 101
pre-condition clobbering	18, 21, 60, 65
relevance classes	49
remote sensing	8, 34, 86, 102
SEIDAM	10, 86
state based state representation	36
sussman anomaly	21, 32, 60, 65
thematic mapper	87, 91
total order planning	3, 9, 16, 69, 103
transformational analogy	6, 12, 27, 37, 69, 100

Abbreviations and acronymes

AVIRIS:	Airborne Visible to Infra-Red Imaging Spectrometre
CBR:	Case-Based Reasoning
DEM:	Digital Elevation Model
DTM:	Digital Terrain Model
EBG:	Explanation-Based Generalization
EBL:	Explanation-Based Learning
EOS:	Earth Observing System
GIS:	Geographical Information System
LGG:	Least General Generalization
NOAH:	Network Of Abstraction Hierarchies
PALERMO:	Planning and LEarning for ResOurce Management and Organization
PAROT:	PALERMO Agent ResOurce Trainer
SEIDAM:	System of Experts for Intelligent Data Management
TM:	Thematic Mapper
TRIM:	Terrain Resource Information Map

Chapter 1

Introduction

Problem solving is an artificial intelligence approach whose goal is to assemble a set of actions that, if applied to a given initial world state, will produce a new state where a desired goal is satisfied. Newell and Simon stipulated that problem solving could be accomplished by a heuristic search of a state space [Newell and Simon 1972]. Search as a problem solving mechanism has proven to have several drawbacks, the most important of which is lack tractability. To address this problem, search control mechanisms have been introduced in the hope reducing the amount of search required to produce a solution. Unfortunately, the result of introducing search control mechanisms, e.g. means-ends analysis, is to narrow the scope of search and thus often to overlook possible solutions.

We are hence faced with two possibilities: define search control mechanisms that reduce the size of the explored search space and apply them to problem solvers that use exhaustive search, or design approaches that will extend the size of the search space explored by problem solvers that already use search control mechanisms. Derivational analogy [Carbonell 1986] is a search control mechanism that has been implemented in the PRODIGY system [Carbonell, Knoblock et al. 1990; Veloso 1992]. It uses past experience of problem solving episodes to guide the problem solver during its search for a solution to a new problem. The approach, as implemented in PRODIGY, uses exhaustive search to generate solutions to problems. Once a solution has been found, the decisions that led to it are remembered and can be re-used when another similar problem occurs. When analogy fails, the system will still use exhaustive search to find a solution. The advantage to such an approach is that, when an analog exists, the solution to a problem is found in a relatively short time, and if an analog does not exist, the solution can still be successfully assembled through exhaustive search when feasible.

Case-based reasoning (CBR), as described by Hammond [Hammond 1989], forgoes search altogether and relies on existing solutions to old problems to solve new problems. When a new problem occurs, an old solution is retrieved and adapted to account for the differences between the old problem and the new problem. If the solution fails during its application, the cause of the failure is remembered and a repair is attempted. The advantage of such a system is that no search is performed by the problem solver. The approach to problem solving presented in this thesis aims at integrating the use of CBR, derivational analogy with a search control mechanism, goal-regression as implemented in STRIPS [Fikes, Hart et al. 1971]. By taking advantage of both CBR and derivational analogy, it is possible to extend the scope of the search performed by a goal-regression problem solver.

Throughout the thesis, examples are drawn from different domains to illustrate the behaviour of the system as well as measure its performance. These domains, ranging from a very simple *one-way-rocket* [Veloso 1992] domain to a much more complex *logistics transportation* [Veloso 1992] domain, present significant challenges to a reasoning system, but they still do not function in a real world production system. The integration of the reasoning system presented in this thesis with other knowledge-based systems is presented in chapter 5, and deals with the maintenance and processing of extremely large data sets for forest management decision support. The approach presented here does not presume to solve the scalability problem, however, it does offer a novel approach to working within a real world application by using simple *software agents* (also known as *apprentices* [Mitchell, Mahadevan et al. 1985; Mitchell, Caruana et al. 1994] and *softbots* [Etzioni and Weld 1994]). Briefly, for each planning operator used by the problem solver, there exists a corresponding software agent. Agents can intelligently perform simple tasks that are assigned to them by the problem solver. In order to do this, agents must provide the problem solver with a planning operator that describes its requirements (pre-conditions, information that must be at the agent's disposal by being stored in a knowledge-base) and its effects (information that the agent will add or remove from the knowledge-base). Since agents accomplish their tasks intelligently by drawing on information stored in a knowledge-base and by modifying the contents of the knowledge-base, planning can be done at a higher level of abstraction thus reducing the amount of work performed by the problem solver.

This chapter is divided into four sections. Section 1 gives four definitions of planning vernacular. Section 2 motivates problem solving work contained in this thesis with a simple example drawn from [Veloso 1992] used to show the differences between this approach, CBR and derivational analogy. Section 3 briefly outlines the integration of simple *software agents* and the problem solving system to help address more complex, real world problems. Section 4 introduces the thesis and its scientific contributions.

1.1 Definitions

Throughout this manuscript, I use four expressions that are well known in the planning community. However, there are often subtle differences in the interpretation of these expressions. I have therefore decided to outline my interpretation here:

- Linear planning: a planning method where a solution is found by solving goals in a *linear* order, i.e. if two or more goals exist then one goal is selected and all of its subgoals are achieved before other goals are considered.
- Non-linear planning: a planning method a solution is found by solving goals in a *non-linear* order, i.e. all goals are placed in a *pending goal* set and if two or more goals exist then one is selected and all of its subgoals are placed in the same *pending goal* set.
- Total order plan: the operators in the plan created by the planner are totally ordered, i.e. permutations of operators that would not affect the outcome of applying the plan are considered different plans by a total order planner.
- Partial order plan: the operators in the plan created by the planner are partially ordered, i.e. permutations of operators that would not affect the outcome of applying the plan are considered the same plan by a partial order planner.

1.2 Goal Regression, Derivational Analogy and CBR

One of the main problems introduced by search control mechanisms such as goal regression is incompleteness. Consider the STRIPS operators in Figure 1.1 below that define the *one-way rocket* domain as introduced in [Veloso 1992].

```
load_rocket(Obj, Loc) ::
if at(Obj, Loc) and at(rocket, Loc)
then
  delete
    at(Obj, Loc)
  add
    inside(Obj, rocket)

move_rocket ::
if at(rocket, loc1)
then
  delete
    at(rocket, loc1)
  add
    at(rocket, loc2)

unload_rocket(Obj, Loc) ::
if inside(Obj, rocket) and at(rocket, Loc)
then
  delete
    inside(Obj, rocket)
  add
    at(Obj, Loc)
```

Figure 1.1 - The *one-way-rocket* domain

When applied, the operator *load_rocket* can load an object into the rocket if both the object and the rocket are at the same location. The operator *unload_rocket* will unload an object from the rocket at the rocket's location. Finally, the operator *move_rocket* moves the rocket from location *loc1* to *loc2* only.

Consider the problem of moving two objects, *obj1* and *obj2*, from location *loc1* to location *loc2* given the initial state described in Figure 1.2 where all objects and the rocket are at location *loc1*.

Initial state:

```
at(obj1, loc1)
at(obj2, loc1)
at(rocket, loc1)
```

Goal statement:

```
at(obj1, loc2) and
at(obj2, loc2)
```

Figure 1.2 - Initial world and goal statement

If the goal statement above is submitted to a goal regression problem solver, the system would start by attempting to solve the first conjunct, *at(obj1, loc2)*. A goal regression problem solver will attempt to solve one goal conjunct completely before attempting to solve the other conjuncts. It would therefore produce the plan in Figure 1.3 for the first conjunct and then attempt to solve the second conjunct, *at(obj2, loc2)*. Since a search based problem solver simulates the applications of the operators it includes in a plan by means of the add and delete lists, after finding a solution for the first conjunct, the problem solver starts from the state reached after applying the operator *unload_rocket(obj1, loc2)*. At this point, the rocket is at location *loc2* and it is therefore not possible to satisfy the second goal. The same situation would arise if the problem solver attempted to solve the second conjunct first.

```
load_rocket(obj1, loc1)
move_rocket
unload_rocket(obj1, loc2)
```

Figure 1.3 - Solution for conjunct *at(obj1, loc2)*

Several authors have introduced approaches to address some of the problems of linear problem solvers. One of the earliest was Tate who created a system called INTERPLAN [Tate 1975], a planning system based on interaction analysis between subgoals, and another called NONLIN [Tate 1977] based on a hierarchy of abstraction levels. Yet another was a system called NOAH [Sacerdoti 1975] created by Sacerdoti and presented in chapter 2.

An alternative approach presented by Veloso uses a non-linear search engine coupled with an analogical component. The discussion below describes the behaviour of NoLimit [Veloso 1992], the non-linear search engine that integrates a goal regression engine with exhaustive search,

i.e. at each planning step, the problem solver either considers a subgoal to regress from or applies an applicable operator. This system is presented because of the integration of analogy with the search engine.

For the *one-way-rocket* example above, NoLimit could behave as follows:

- if there is more than one applicable operator in the current state, remember this state as backtrack point 1,
- select a conjunct for regression, e.g. *at(obj1, loc2)*,
- select the operator *unload_rocket(obj1, loc2)* as the operator that satisfies the conjunct,
- set the new goal to be the pre-conditions the operator *unload_rocket(obj1, loc2)*, i.e. *inside(rocket, obj1)* and *at(rocket, loc2)*,
- select a conjunct for regression, *inside(rocket, obj1)*,
- select the operator *load_rocket(obj1, A)* as the operator that satisfies the conjunct,
- set the new goal to be the pre-conditions the operator *load_rocket(obj1, A)*, i.e. *at(rocket, A)* and *at(obj1, A)*,
- select a conjunct for regression, *at(rocket, A)*,
- since the fact *at(rocket, loc1)* holds in the initial world, the variable *A* is bound to the value *loc1*,
- since both pre-conditions of *load_rocket(obj1, loc1)* are satisfied, apply the operator and create a new state (*at(rocket, loc1)* and *at(obj2, loc1)* and *inside(rocket, obj1)*),
- select the next conjunct for regression, *at(rocket, loc2)*,
- since there is only one applicable operator in this state, i.e. *move_rocket*, and its pre-conditions are satisfied, apply the operator and create a new state (*at(rocket, loc2)* and *at(obj2, loc1)* and *inside(rocket, obj1)*),
- since all pre-conditions for operator *unload_rocket(obj1, loc2)*, are now satisfied, apply the operator and create a new state (*at(rocket, loc2)* and *at(obj2, loc1)* and *at(obj1, loc2)*),
- since there still exists an unsatisfied conjunct, *at(obj2, loc2)*, are no other operators are applicable backtrack to last backtrack point, select an applicable operator, e.g. *load_rocket(obj2, loc1)*, and apply it thus creating the new state *at(rocket, loc1)* and *at(obj1, loc1)* and *inside(rocket, obj2)*,
- redo all of the previous steps to reach the state *at(rocket, loc2)* and *at(obj1, loc2)* and *inside(rocket, obj2)*,
- select a conjunct for regression, *at(obj2, loc2)*,
- select the operator *unload_rocket(obj2, loc2)* as the operator that satisfies the conjunct,
- since all of the operators pre-conditions are satisfied, apply the operator at create the new state *at(rocket, loc1)* and *at(obj1, loc1)* and *at(obj2, loc2)*,
- the successful plan is shown in Figure 1.4.

By remembering backtrack points and the applicable operators at those points, the NoLimit problem solver performs an exhaustive search thereby insuring the production of a solution if one exists. For the *one-way-rocket* example, NoLimit solves the problem in a reasonable amount of

time. However, when solving problems in domain where there are more operators, NoLimit requires much more time to find solutions.

```
load_rocket(obj2, loc1)
load_rocket(obj1, loc1)
move_rocket
unload_rocket(obj1, loc2)
unload_rocket(obj2, loc2)
```

Figure 1.4 - Solution for the goal statement *at(obj1, loc2) and at(obj2, loc2)*

Analogy can be used in two ways to improve the performance of this type of reasoning system. The first is to use derivational analogy to help at each decision point during planning. The second is to use transformational analogy to create new problem solutions from previously constructed solutions.

Derivational analogy was added to NoLimit to help it select among competing search paths at each decision point. Although the new system still required a significant amount of time to solve new problems from scratch, its performance was greatly improved when similar problems had been seen before since every planning step was guided by the decisions made in the old solution. For the example above, when derivational analogy is used the backtrack point is remembered. However, since NoLimit has a case on which to base its decisions, it would include the operator *load_rocket(obj2, loc1)* before proceeding to build the complete plan thereby avoiding backtracking at the later planning step.

If transformational analogy had been added to the NoLimit problem solver, then it would have to have been given a past similar case, plan modification rules and a similarity matrix that would enable it to adapt the case to the new problem. Like derivational analogy, the effect would have been to reduce the amount of search that NoLimit would have to carry out while attempting to find solutions to new problems.

The approach presented here, dubbed PALERMO (Planning And LEarning for Resource Management and Organization), integrates the use of goal regression with both derivational and transformational analogy [Carbonell 1986]. The approach is to use a narrow search control mechanism as the basic problem solver and to expand its coverage of the search space by using analogy. Derivational and transformational analogy are used together. Rather than remembering the decisions made at each planning step as is described above, a case for PALERMO's analogical component is akin to a case in transformational analogy, i.e. a case is completely ground and was created to solve a specific goal. As PALERMO's experience grows, cases can be generalized and re-used much more efficiently. Once a case has been generalized, a replay mechanism drawn from the derivational analogy paradigm is used to provide bindings for the generalized components of the case.

In the example below, the solutions to each of the goal conjuncts are symmetrical. However, the purpose of this example is to show that PALERMO learns and re-uses cases more eagerly. As discussed in chapter 3, this allows the system to find solutions for problems that cannot be solved by linear problem solvers.

Given the operators in Figure 1.1, the initial world and goal statement in Figure 1.2, and an empty case-base, PALERMO would select the first conjunct, *at(obj1, loc2)*, of the goal statement and solve it completely using goal regression. Under normal circumstances, a goal regression problem solver would then fail to solve the second conjunct. However, before attempting to solve the second conjunct, PALERMO remembers the solution for the first conjunct (Figure 1.3) in its case-base and thus makes it immediately available for re-use. It then selects the next conjunct, *at(obj2, loc2)*, and proceeds to find a solution. Since there is now an existing case in the case-base, PALERMO will try to adapt it for the current subgoal. As stated above, this case is completely ground and is tailored to the first conjunct. The first step is to compute the substitution that would transform the goal for the case, *at(obj1, loc2)*, into the new subgoal, *at(obj2, loc2)*. The resulting substitution is *[obj1/obj2]*. The second step is to apply the substitution to the case and create the plan in Figure 1.5.

```
load_rocket(obj2, loc1)
move_rocket
unload_rocket(obj2, loc2)
```

Figure 1.5 - Solution for conjunct *at(obj2, loc2)*

The next step is to merge the two plans together while considering the effects of interleaving the operators from one plan on the operator pre-conditions of the other. If the merge is successful, then the final step is to generalize the case with respect to the substitution computed above. Although this example is extremely simple in that there is full symmetry between the subgoals, it does show that goal regression can be used to solve individual subgoals separately and their solutions can subsequently be merged. By taking this approach, the weakness of the linear nature of regression can be addressed.

1.3 Problem solving with Agents

Marvin Minsky [Minsky 1994] stated that the expression *software agent* is used to describe a system that can serve as a go-between, because of possessing specialized skills. The term *agent* has also been used to describe systems ranging from adaptive user interfaces to systems that use planning algorithms to generate shell scripts [Kautz, Selman et al. 1994]. These systems have been designed to unobtrusively observe the manner in which they are used, adapt to the tasks for which they are used, as well as learn from the circumstances of their use [Mitchell, Mahadevan et al. 1985; Etzioni and Weld 1994; Mitchell, Caruana et al. 1994; Matwin, Charlebois et al. 1995]. These systems operate in settings in which there is an abundance of examples of their use.

Standard inductive learning mechanisms can then adapt agents to the environment in which they operate based on the repetitive examples of their operation. Recent interest in agents resulted in systems that assist people in a variety of tasks: the use of Internet, planning of meetings, and scheduling (e.g. a series of interviews in an organization [Kautz, Selman et al. 1994]).

Agents perform tasks for users in a deterministic fashion, i.e. the agent's behaviour can be represented by finite state machine and can therefore be examined by graph traversal algorithms. They also process and modify information contained in a knowledge base by adding and deleting objects. These characteristics allow us to construct planning operators for each agent that describes the tasks they perform. Planning operators are composed of a pre-condition list and add and delete lists. By giving agents the ability to describe the task they perform to PALERMO, it becomes possible for the reasoning system to assemble seemingly simple plans that satisfy complex goals. This results in allowing PALERMO to do higher level reasoning since the agents are themselves responsible for providing the lower level services that could be otherwise expensive to reason about.

This approach has been implemented in a prototype system that performs digital forest cover map update. By processing satellite imagery, it is possible to detect changes in forest cover as a result of logging or other types of forest cover depletion. The process by which a forest cover map must be updated is:

- extraction of a digital elevation model (DEM) and forest cover map from a database into a geographical information system (GIS) format,
- creation of a digital terrain model (DTM) using the DEM,
- selection of the best satellite imagery for this problem,
- radiometric and atmospheric correction of satellite imagery,
- registration of the satellite imagery to the forest cover map or GIS file,
- overlaying of the satellite imagery on the forest cover map,
- enhancements to the satellite imagery to highlight clear cuts and other harvested areas,
- creation of the new harvested area boundaries,
- highlight the areas where the enhanced satellite image and the map disagree on forest cover,
- create and store the updated forest cover GIS file and map.

The PAROT system (**P**ALERMO **A**gent/**R**es**O**urce **T**rainer) has been used to train a set of 25 agents that are able to perform the tasks listed above. Each agent performs a task domain experts consider to be one processing step for forest cover update. Agents manipulate information in a shared knowledge-base, i.e. they retrieve, delete and add object attribute values from the knowledge-base. These object manipulations can be expressed in planning operators as pre-conditions and effects, and can therefore provide PALERMO with the necessary information to assemble plans in the forest cover update domain.

1.4 Contributions and Overview

The contributions of the work presented in this thesis are:

- to machine learning:

Goal-directed generalization is a learning approach that is similar to explanation based generalization. As the name suggests, goal-directed-generalization uses the goal statement to focus its generalization efforts. The focus of the generalization procedure is based on the definition of relevance classes, section 3.2.1.2, that serve to generalize cases and to assess the suitability of a case for new problems.

- to case-based reasoning:

The utility problem, in case-based reasoning in particular and in machine learning in general, occurs when remembering the new information causes a system's knowledge base to grow so that its performance degrades. The results presented in chapter 4 show that the design of cases introduced in this thesis can help address this problem. The main feature that distinguishes cases from other systems is that each case can only satisfy one conjunct goal statements. For example, in the logistics transportation domain, other systems have required an initial case-base of over 1000 cases. The system presented here starts with an empty case-base and, after thousands of tests, grows to a upper bound of approximately 35 cases.

- to planning:

By adding an analogical component to a linear, total-order planner, it becomes possible to create plans that can only be handled by non-linear problem solvers. Indeed, as was shown in section 1.1, by re-using cases "eagerly", goal-regression coupled with case-based reasoning suddenly becomes a non-linear planning system.

- scalability:

Chapter 5 discusses a real world application. The domain for this application integrates the use of complex processes to manipulate geographical information system data, remote sensing data and databases. The introduction of software agents able to describe their behaviour in a computer-understandable way enables the integration of a reasoning system into a complex real world domain. The agents' descriptions are STRIPS-like planning operators that list the agents' pre-conditions (e.g. what data must exist) and effects (e.g. what data is deleted and what data is added).

The remainder of the thesis is organized as follows: chapter 2 discusses other approaches that are relevant to problem solving and case-based reasoning related to my work; chapter 3 outlines the design and implementation of the reasoning system; chapter 4 presents a discussion of an empirical evaluation of the system; chapter 5 describes how the problem solver is integrated into the SEIDAM system to deal with real world problems integrating geographical information data and remote sensing data for forest information management; finally, chapter 6 concludes the thesis and discusses future work.

Chapter 2

Related work

Research in planning and problem solving has taken two main directions over the last 25 years. The first approaches were based on the exploration of a search space characterized by a set of vertices and edges. Each vertex contained a set of facts describing a world state for a given domain. Each edge connected two vertices such that traversing the edge from one vertex to the other represented an acceptable action in the domain that changes the world from one state to another. The second type of approach to planning is case-based reasoning. The two main motivations for the use of case-based reasoning are: (1) to control the amount of search performed by either reducing the number of vertices examined or to eliminate search altogether; (2) to promote the re-use of existing plans in order to facilitate the planning task.

Approaches such as goal regression and means-end analysis gained much recognition during the early 70s since they rendered search-based planning tractable by significantly reducing the number of vertices explored during search. The obvious drawback of such approaches is that they often tend to ignore large sections of the search space and thus overlook possible solutions. Much effort has since gone into improving planning systems by using other AI techniques. The goal of some work was to increase the number of vertices explored by search engines such as goal regression. Other work has focused on reducing the number of vertices explored by search engines that perform a complete traversal of the search space.

One of the early attempts at increasing the number of explored vertices was the NOAH [Sacerdoti 1975] system designed by Sacerdoti. When NOAH tries to find a solution to multiple goals, it merges plans that can solve individual goals separately. This is accomplished by

examining the operators that were used to produce the plans and attempting to resolve the conflicts between operators.

Rather than increasing the number of vertices explored during planning, Veloso [Veloso 1992] took another tack by attempting to reduce the number of search nodes explored by a non-linear, complete search engine. The approach, called derivational analogy, uses cases representing successful planning decisions made in the past and applies them in new situations. Derivational analogy remains a search-based planning approach, however, since the use of past experience suggests which edges should be used to reach other vertices in the search space, the system performs a much more focused search.

Hammond [Hammond 1989] also uses case-based reasoning to produce solutions for new problems. However, his approach is based on transformational analogy where solutions are retrieved based on a similarity matrix and subsequently adapted to account for the differences between the problem for which the case was originally created and the new problem being solved. One of the notable contributions of this work is that the system creates solutions without performing any search.

While Yang's [Yang 1992] work was not mainly directed at search space traversal, it did nonetheless introduce important contributions to plan and operator merging. His concern is mainly to find optimal plans to satisfy goals. One of the main contributions is that the approach allows operators belonging to the same equivalence class to be merged into a single operator.

Since the problem solver presented in this thesis is used in the forest information management domain, examples were drawn from this domain to highlight the finer points of the work introduced above as well as the problems they do not address. For forest information management, spatial databases, called geographical information systems or GIS, are used to store information relating to forest attributes (main species over a given area, timber volume of secondary species, under story vegetation type, etc.).

2.1 Geographical Information Systems and STRIPS

Before further discussion is possible about the works presented above, there are two topics that must be addressed. The first introduces the basic concepts of Geographical Information Systems (GIS). The second is a brief description of the STRIPS approach [Fikes 81]. GISs will be the application for the examples used to show the strengths of the approaches that will be discussed. STRIPS is introduced to provide a specific example with which different approaches will be contrasted.

2.1.1 Geographical Information Systems

A Geographical Information System integrates traditional database management systems with graphical user interfaces and sophisticated data structures to store maps. The GIS system on which the examples will be based is the PAMAP GIS. The distinction is made because the vocabulary in the GIS field does not always agree between different approaches.

In the PAMAP GIS a map is a data structure that contains graphical representations of the actual map and database pointers that refer to relevant information about the map's area:

```
map_structure = layers: array of layer;
layer = record
  case layer_type: ( surface_cover, polygon_cover ) of
    surface_cover: ( raster_image );
    polygon_cover: ( polygons, database_tags )
  end
```

A map covers a geographical surface and contains information about that surface. For instance, a map that represents elevation has line work that delimit areas on the map where each area has a different elevation. In the GIS, each area is called a polygon. Each polygon on a map has a database tag pointing to a database record containing information that should be consistent across the polygon. For instance, in a forest cover map, a database record associated to a given polygon will identify the primary, secondary and tertiary species, their respective percentage of coverage over the polygon surface and possibly other information. This type of map is called a polygon cover map. A surface cover map is a raster image where each pixel has a numerical value. For instance, a surface cover elevation map would not have line work that delimits areas and thus is not associated to a database record. Each pixel on the image has a value that is the elevation at that point on the map.

Also, each map can have a variety of different layers. Each layer contains information about different geographical surface features, for instance:

- forest cover maps (polygon cover map that describes the forest),
- rivers (polygon cover map showing hydrological information),
- elevation.

Moreover, by overlaying maps, it is possible to extract surface information pertaining to intersections between polygons from different layers, etc. For instance, by overlaying an elevation map and a river map, it is possible to determine the direction in which a river flows.

2.1.2 STRIPS

STRIPS [Fikes, Hart et al. 1971; Fikes, Hart et al. 1981] is a problem solving system based on means-ends analysis and goal regression. The problem can be described as follows:

GIVEN:

- an initial world state described by facts,
- a goal describing facts that must hold in the final world state,
- knowledge about how the world evolves (this is a general description that accounts for case-based reasoning, problem solving from first principle, etc.)

FIND:

- a sequence of "steps" that will transform the initial world state into a state where the goal is satisfied.

Facts are predicates that represent object/attribute/value triplets that hold in the world at a given time. The initial world is the set of facts that are true when the problem solving exercise begins. Problem solving requires that the initial world be changed such that the goal statement is true in the final world. The goal statement, hereafter the goal, is expressed as a fact or a conjunction of facts. For the goal to be satisfied, each conjunct of the goal must hold after the initial world has been changed.

The steps that change the world are operators that must respect the following BNF definitions:

```
<operator> -->    <operator-name> [ ( <parameter-list> ) ] ::  
                    [ parms          <parameter-bindings> ]  
                    if                <pre-conditions-list>  
                    then delete      <deleted-fact-list>  
                    add              <added-fact-list>  
  
<pre-condition-list> --> <fact-list>  
<deleted-fact-list>  --> <fact-list>  
<added-fact-list>    --> <fact-list>  
  
<fact-list>          --> <fact> [ and <fact-list> ]  
  
<parameter-bindings> --> <object> is_a <object-type>  
                        [ and <parameter-bindings> ]
```

The following example was adapted from [Veloso 1992] and shows the initial world and the goal statement as sets of facts.

```
initial world: { at( pc, uofo ), at( mac, uofo ), at( rocket, uofo ) }
```

```
goal: { at( rocket, ucarleton ) }
```

The operators that can be used to modify this world are:

```
load_rocket( Object, Location ) ::
    if                at( Object, Location ) and
                      at( rocket, Location )
    then delete      at( Object, Location )
                   add   inside( Object, rocket )

unload_rocket( Object, Location ) ::
    if                inside( Object, rocket ) and
                      at( rocket, Location )
    then delete      inside( Object, rocket )
                   add   at( Object, Location )

move_rocket ::
    if                at( rocket, uofo )
    then delete      at( rocket, uofo )
                   add   at( rocket, ucarleton )
```

The operators shown here have three important components. The first is the pre-condition following the "if" keyword. A pre-condition is a set of facts that must hold in the world before the operator can be applied. The other two components are the "delete list" and the "add list". The delete list is the set of facts that do not hold in the world after the operator has been applied. The add list is the set of facts that hold after the operator has been applied. If we examine the load_rocket operator, it states that the load_rocket(Object, Location) can be done if the rocket and the object are at the same location. As a result of applying the operator, the fact at(Object, Location) is deleted from the world and the fact inside(Object, rocket) is added to the world.

For example, to satisfy the goal statement above, a problem solver must select an operator and try to apply it. Let us assume that it selects the move_rocket operator. Given the initial world:

```
{ at( pc, uofo ), at( mac, uofo ), at( rocket, uofo ) }
```

the pre-conditions of the operator hold, i.e. at(rocket, uofo) is in the set of facts representing the initial world. Since the pre-conditions are satisfied, we can perform the changes described by the delete and add lists. The first is to delete the fact at(rocket, uofo) resulting in the world:

```
{ at( pc, uofo ), at( mac, uofo ) }
```

The next action is to add the facts in the add list to the world, thus producing:

```
{ at( pc, uofo ), at( mac, uofo ), at( rocket, ucarleton ) }
```

Hence the solution to the problem above is to apply the operator move_rocket.

For problem solving, the system performs a problem solving step by selecting an operator and applying it. In general problem solving, operator selection can be done arbitrarily, thus resulting in intractable exercises.

STRIPS can produce this solution by means-ends analysis and goal-regression. The basic STRIPS algorithm is:

```
procedure solve( Gs: list of goals; var Solution )
repeat
  select a goal G from Gs
  if G is not true in the current world then
    repeat
      find a operator O that has G in its add list
      status<-solve(preconditions of O, PreConditionSolution)
    until status = success or no rules can be found
    if not ( status = success ) then exit( failure ) endif
    Solution <- Solution U { O } U PreConditionSolution
    apply the actions in operator O's delete and add lists
  endif
until Gs is empty
exit( success ).
```

The operator selection mechanism in STRIPS is the basis of goal regression. At each problem solving step, an operator is selected only if one of the goal statement's conjunct is a member of its add list. This operator selection bias is based solely on syntax.

2.2 Partial Orderings on Plans

The behaviour of a linear planner such as STRIPS is to select one goal at a time, attempt to satisfy it, and, if successful, protect it before trying to solve the next goal. To protect a goal means that any operator that tries to remove the goal from the world once it has been achieved cannot be selected for application.

The following example shows how a linear planner would not succeed in forming a plan to solve a conjunction of goals. The example was created on a GIS system called PAMAP. In PAMAP, to create a new polygon level the system needs new line work stored on a vector level, a set of database tags and database records. The line work should be a set of closed areas that will form polygons after processing by the GIS. This line work is stored as a set of vectors on a vector cover level. The database tags are pointers placed inside each polygon, they are references to database records that contain information describing their corresponding polygons. If these three parameters are available, then it is possible to successfully execute the *create polygons* command that will create a new polygon cover level. The four operators listed below will start and stop the

GIS, create a new polygon cover level and remove line work from a vector level used to create a polygon cover level.

```
initial world: { forest_level is_a polygon_cover_level,
                new_line_work( clayoquot, working_vector_level ),
                db_tags( clayoquot, working_vector_level ),
                database( clayoquot, working_vector_level ),
                clayoquot has working_vector_level }
```

operators:

```
create_polygons( Map, Level ) ::
    parms          Level is_a polygon_cover_level
    if             new_line_work( Map, VectorLevel ) and
                   db_tags( Map, VectorLevel ) and
                   database( Map, Level ) and
                   running( pamap )
    then add       Map has Level.

reset( Map, Level ) ::
    if             new_line_work( Map, VectorLevel ) and
                   db_tags( Map, VectorLevel ) and
                   database( Map, Level ) and
                   running( pamap )
    then delete    new_line_work( Map, Level ) and
                   db_tags( Map, Level ) and
                   database( Map, Level ) and
    add            clear( Map, Level ).

start( Program ) ::
    parms          Program is_a program
    if             not( running( _ ) )
    then add       running( Program ).

stop( Program ) ::
    parms          Program is_a program
    if             running( Program )
    then delete    all( current_map( Program, _ ) ) and
                   running( Program ).

user goal:        { clear( clayoquot, working_vector_level ),
                   clayoquot has forest_level }
```

In this example above, the GIS database already contains a map called "clayoquot" that has some new line work, database tags and a database on a working level. This information constitutes the pre-conditions of two of the operators above, i.e. the reset operator and the create_polygons operator. Given the user goal as stated above, a linear planner would first try to find a solution for the first conjunct. The solution it would find is to apply the operator "reset(clayoquot, working_vector_level)". It would, then, attempt to find a solution to the second conjunct. Since

the application of the "reset" operator deletes the pre-conditions of the "create_polygons" operator, the planner would fail to find a solution.

If each of the goals were addressed separately, a linear planner would produce the following solutions:

To satisfy:

```
goal 1: clear( clayoquot, working_vector_level )
```

apply plan:

```
start( pamap )  
reset( clayoquot, working_vector_level )  
stop( pamap )
```

```
goal 2: clayoquot has forest_level
```

apply plan:

```
start( pamap )  
create_polygons( clayoquot, forest_level )  
stop( pamap )
```

An alert reader should realize that simply reordering the goals would allow a linear planner to solve this particular problem. In this case, it should be obvious that the goals are not independent since they share resources.

To address this problem, Sacerdoti introduced NOAH (Networks of Action Hierarchies) [Sacerdoti 1975], a planner that addresses non-linear planning problems by merging plans. The approach is based on imposing a partial ordering on plan operators to ensure that the application of one operator does not prevent the application of another. In the merged plan, the operator whose application is threatened, is applied before the operator that threatens it.

In NOAH, plans are remembered in a hierarchy of nets. Plan formation is accomplished by expanding nodes to lower levels and applying critics that resolve conflicts between operators, provide bindings for variables in the plan under construction and eliminate redundant operators from the plan. Once all of the critics have been applied, there is an ordering that is imposed on the operators.

The NOAH algorithm:

- 1- simulate the most detailed plan in the procedural net. Each plan in the procedural net is stored in a node. When expandable, the node with the most detailed plan in the net is expanded thus producing a new, more detailed net.
- 2- apply the resolve conflict critic. The resolve conflict critic examines precondition clobbering and imposes an ordering on the operators.
- 3- apply the existing object critic. Binds variables in operators to the best known object. if no object is clearly the best choice, the planner generates a formal object to bind the variable until a clear choice can be made
- 4- apply the eliminate redundant pre-condition critic. This has the effect of eliminating redundant pre-conditions that can result from node expansion.
- 5- if all nodes have been expanded, return plan
else goto 1.

In NOAH, plans are built up in a data structure called a procedural net. Initially, the procedural net contains one node. This node contains a plan to satisfy the user goals. The plan contains symbolic expressions that represent the changes to the world model caused by the action that the node represents. NOAH visits each node in the net and simulates the action represented therein. As a result, new nodes are created representing a more detailed plan. Each in turn, three critics examine the new nodes in the procedural net. Their task is to impose an ordering on the nodes and eliminate any redundant operations. NOAH terminates when it does not expand any of the new nodes.

In the example given above, NOAH would first represent the plan in the most abstract way by using a single node shown in Figure 2.1.

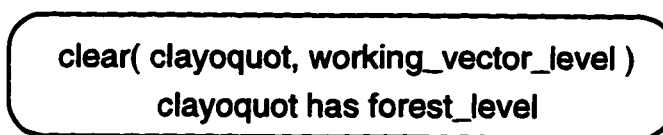


Figure 2.1

NOAH then expands this node and produces the network shown Figure 2.2.

Given this network, applying the critics does not change the network since they cannot detect any conflicts or redundancy. However, NOAH does not terminate since it is possible to expand these nodes into Figure 2.3.

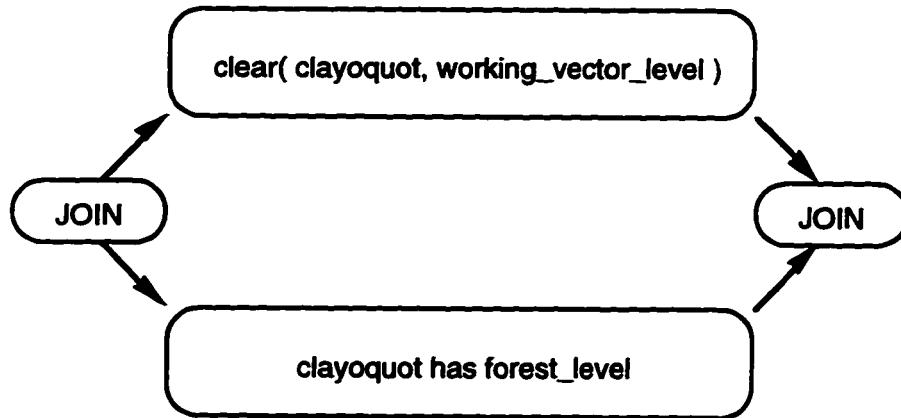


Figure 2.2

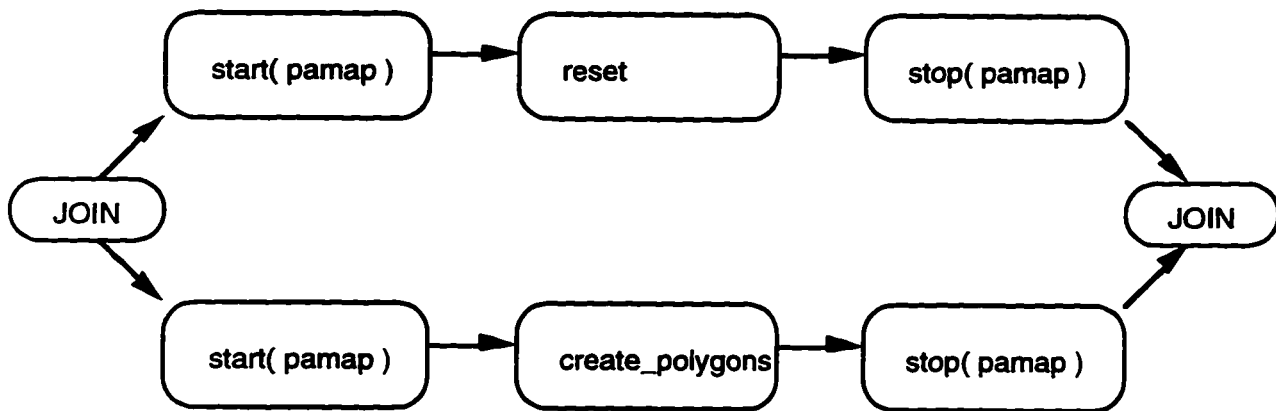


Figure 2.3

To begin, the eliminate redundant pre-conditions critic would notice that, in this plan, the pre-conditions satisfied by the operator "start(pamap)" to clear the working vector level also satisfies the same pre-conditions to create a new polygon level, thus the two operators are deemed redundant and can be replaced by a single occurrence of the same operator shown in Figure 2.4.

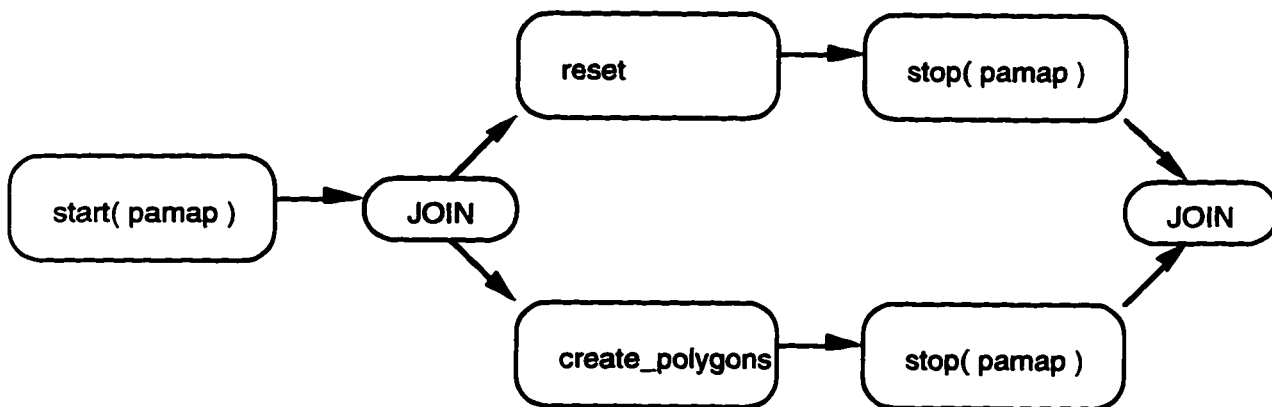


Figure 2.4

The resolve conflict critic would then find that there is an ordering that must be imposed on this network. The preconditions to resetting a map database and to creating a polygon vector level are identical. However, after resetting the map database, the pre-conditions of the operator `create_polygon` are no longer true. It becomes necessary to impose an ordering on the nodes that would place the polygon creation operator before the reset operator:

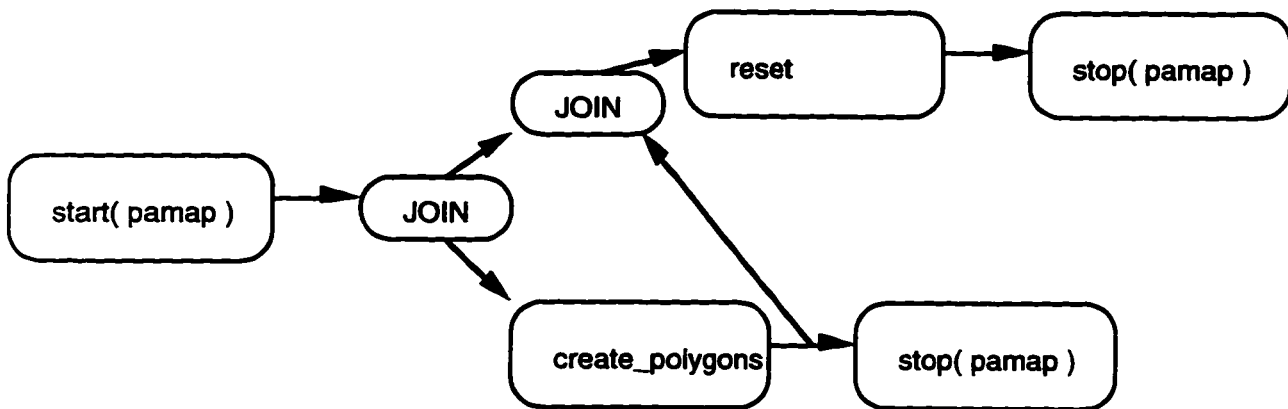


Figure 2.5

Again, the eliminate redundant pre-conditions critic would then notice that, in this plan, the pre-conditions satisfied by the operator `stop( pamap )` to reset the working vector level also satisfies the same pre-conditions to create the polygons and can be replaced by a single occurrence of the same operator:

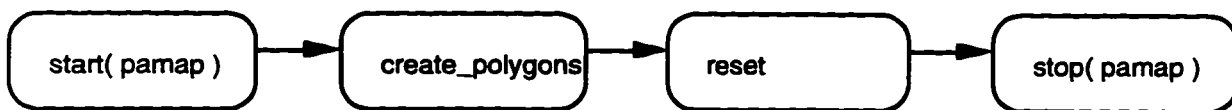


Figure 2.6

In addition to specifying the operators that must be applied, a plan also determines the objects that the plan must manipulate. The third critic puts off this decision until the last possible moment with the hope that waiting will allow NOAH to select the best object for the operation.

The main problem that NOAH cannot resolve is the Sussman anomaly [Sussman 1973] involving mutual pre-condition clobbering when trying to impose an order on operators. Briefly, if a plan has two operators that must be applied where each operator changes the state of the world such that the other operators pre-conditions are no longer true, then NOAH will fail to form a plan.

2.3 Derivational Analogy

In the example given in the last section, it was noted that a simple reordering of the goals would have allowed a linear planner to find a solution to the goal conjunction. However, there are cases where goal reordering would not suffice.

In the following example, the user goal is to display two maps on a monitor. The two maps are currently stored on tape and must be transferred to disk before being displayed. Given the resources that this system has at its disposal, in order to copy the maps from tape to disk, there must not be any programs running other than the *get map* program. With only the operators listed at its disposal, a linear planner would not be successful at creating a plan to address the user goal.

```
initial world: { tape_map( clayoquot ),
                tape_map( sooke ) }

operators:
    display_map( Map ) ::
        if
            then delete
            add
            disk_map( Map ) and
            running( mapper )
            disk_map( Map )
            displayed( Map ).

    start( Program ) ::
        parms
        if
        then add
        Program is_a program
        not( running( _ ) )
        running( Program ).

    get_map( Map ) ::
        if
        then delete
        add
        tape_map( Map )
        tape_map( Map )
        disk_map( Map ).

user goal: { displayed( clayoquot ),
            displayed( sooke ) }
```

Given these operators, once the mapper program has started, it cannot be stopped. A linear planner would first solve the goal "displayed(clayoquot)". Since solving this goal also starts the mapper program, it is impossible to solve the goal "displayed(sooke)". Likewise, if a linear planner tries to solve the second goal first, it would encounter the same situation, i.e.. since the program has already started to display the second map, it cannot solve the first goal. The obvious solution is to store both maps on tape before starting the mapper program. This implies that the initial world must somehow be changed. To change the initial world, one can simply select an applicable operator, one whose preconditions are satisfied, and apply it. For instance, if the operator "get_map(sooke)" were applied before trying to solve the user goal, a linear planner

would be able to find a solution. Applying the operator to the initial world is a non-linear step. Hence, to solve the problem:

```
goal: { displayed( clayoquot ), displayed( sooke ) }
```

Apply plan:

```
get_map( sooke )           - non-linear step
get_map( clayoquot )
start( mapper )
display( clayoquot )
display( sooke )
```

In [Veloso 1992], the NoLimit algorithm performs such a step based on cases of past experience:

GIVEN:

- initial world state
- set of operators (STRIPS-like)
- derivational experience (cases representing past decisions that led to solutions or better solutions)
- user goals

DETERMINE:

the best solution to achieve the user goals

ALGORITHM:

- 1- check if the goal statement is true in the current state, or if there is a reason to suspend the current search path (e.g. failure to find a solution via this path in a previous problem solving episode).
if yes then return the final plan or backtrack
- 2- compute the set of pending goals G , and the set of possible applicable operators A .
- 3- do one of:
 - 1- A-chose a goal g from G
B- expand goal g , i.e. get set O of relevant instantiated operators for the goal g ,
C- choose an operator o from O ,
D- solve pre-conditions of o (i.e. recursive call to NoLimit)
 - 2- A- select an operator a from A that is directly applicable. This operator is selected based on analogical derivation, i.e. their operator selection is supported by a case in the case base
B- apply a
- 4- go to step 1

The approach taken in the NoLimit system is essentially to perform plan formation from first principle by doing a depth first search of the problem solution space. This is done by selecting an operator at each step in the search algorithm. The selection of an operator is done in one of two methods. The first method is done in a STRIPS-like fashion, i.e. the planner selects an operator that will add to the world one of the unsolved goals in a pending goal list. The second method for selecting an operator is guided by two criteria. The first is that the operator must be applicable in the current search state, i.e. all of the operator's preconditions must be true in the current state and, there must exist evidence that the application of the operator will somehow change the state of the world such that the new state is "closer" to a state satisfying the goal. This evidence and the measure of closeness is represented as a case in the derivational analogy paradigm.

In the example above, the knowledge used by the NoLimit system includes the operators as they are expressed as well as a case base. The system behaves in much the same way as STRIPS, in that it performs goals regression to form plans. There is one extra action done at each planning step. The behaviour of the system could be as follows:

- the pending goals are initially the user goals: { displayed(clayoquot) and displayed(sooke) },
- since they are not true and there is no reason to suspend the search (i.e. some operators have not yet been tried), NoLimit will go to step 2,
- the set of pending goals contains the user goals, the set of applicable operators is { start(pamap), get_map(sooke), get_map(clayoquot) },
- the third step is a non-deterministic step:
 - do one of:
 - select a goal G from the pending goal list; or
 - select an operator A from the applicable operator list.

For this example, let us assume that NoLimit selects the goal: displayed(clayoquot),

- since a goal was chosen, the next step is to expand it, i.e. select a relevant operator (one that has the goal G on its add list); the only relevant operator is display_map(clayoquot); update the goal statement according to the operator's pre-conditions, i.e. replace the goal G on the current goal statement with the operator's pre-conditions; the new goal statement is: { disk_map(clayoquot), running(mapper) }
- solve the new goal statement
- since the goal statement is not true compute a set of pending goals and applicable operators:
 - goals: { disk_map(clayoquot), running(mapper) },
 - operators: { start(pamap), get_map(sooke), get_map(clayoquot) },

operators selected thus far: { display(clayoquot) }

If we continue with this behaviour, i.e. selecting goals rather than applicable operators according to the syntactic bias, NoLimit will fail and eventually could backtrack to this state. In this event, since goal regression failed the only alternative in this state is to select an applicable operator. If we accepted that the operator could be selected arbitrarily, then the search would become intractable. However, the case base now provides the knowledge based bias for the operator selection. In this episode, since the pending goals are { disk_map(clayoquot), running(mapper), displayed(sooke) }, the case base would suggest applying the get_map(sooke) operator. This suggestion is based on knowledge of past experience or a case provided to the system from the outset. In either case, the selection is not based on a syntactic bias. Once this operator has been applied, NoLimit can resume planning via goal regression. Because of this operator's application, NoLimit would be successful from the second step:

- world state: { tape_map(clayoquot), disk_map(sooke) }
- goals: { disk_map(clayoquot), running(mapper), displayed(sooke) },
- applicable operators: { start(pamap), get_map(clayoquot) }

operators selected thus far: { get_map(sooke), display(clayoquot) }

- step 3: select a goal from G or an operator from A; the choice would be to select a goal,
- find a relevant operator for the goal disk_map(clayoquot); the only operator is get_map(clayoquot),
- since the operator's pre-conditions are true, the goal statement is now: { running(mapper), displayed(sooke) },

operators selected thus far: { get_map(sooke), get_map(clayoquot), display(clayoquot) }

- compute pending goals and applicable operators:
 - goals: { running(mapper), displayed(sooke) },
 - operators: { start(mapper) }
- in this case, let us assume that the operator is applied resulting in:
 - world state: { running(mapper), disk_map(clayoquot), disk_map(sooke) }

operators selected thus far: { get_map(sooke), get_map(clayoquot), start(mapper), display(clayoquot) }

- at step one, NoLimit continues,
- the set of pending goals has remained unchanged (since we chose to apply the operator):

- world state: { running(mapper), disk_map(clayoquot), disk_map(sooke) }
- pending goals: { running(mapper), displayed(sooke) }
- applicable operators: { display(sooke) }

Since the mapper program is already running, the first goal in the pending goal list is satisfied. The next step would simply add the remaining operator to the solution thus resulting in:

operators selected thus far: { get_map(sooke), get_map(clayoquot), start(mapper), display(sooke), display(clayoquot) }

The main advantage of this type of system is that, if we relax the requirement that evidence exists for the selection of an operator, we can make the search space traversal complete. However, this type of unrestricted search becomes intractable given the potential exponential size of the search space. By using derivational analogy, the hope is to extend the number of nodes a pure linear planner might visit by providing justification for visiting nodes that would otherwise not be by a linear problem solver.

The approach can also provide a linear planner with a "preferred" thread through the search space that will yield better solutions, more quickly. This behaviour is achieved by using cases to select a preferred operator amongst several competing applicable operators. As experience grows, the system remembers which operators were more successful and will select them before others.

It was pointed out in [Veloso 1992] that NoLimit can easily handle conjunctions for which a linear planner cannot find a solution (e.g.: the one way rocket problem). However, if we restrict goals, temporarily, to one literal goals only, i.e. no conjunctions allowed, and we provide a linear planner with the same operators, it will often be successful in trying to find a solution. The main advantage to using analogical reasoning in this case would be in finding solutions more efficiently and in creating better plans. These measures are domain dependent and the knowledge required for this behaviour is represented as cases.

The main problem that must be addressed when using derivational analogy is the acquisition of cases of which there are two types:

- 1- the "preferred solution" type,
- 2- the "leads to a solution" type.

Cases of type 1 are used to select between competing operators when performing a linear search. This type of case allows the planner to select the "better" operator. Acquiring the knowledge that will identify the preferred solution can be done by experimentation. The experimental results can be put at the problem solver's disposal by storing them in a preference matrix.

Cases of type 2 are somewhat more difficult to provide. The acquisition of type 2 cases can be done similarly to cases of type 1. However, the problem solver must actually find a solution before acquiring a case, in effect, performing a non-linear search. The alternative is to supply the problem solver with an initial case-base.

To solve conjunctions, our approach will be to solve each conjunct independently and to merge the solutions. The approach also allows us to take full advantage of type 1 cases.

2.4 Transformational Analogy

Hammond [Hammond 1989] takes a completely different approach to plan formation. Rather than creating a plan from first principle, the approach requires a set of existing plans, a case-base, upon which the reasoning system can draw. The input to the system, CHEF, is a set of goals that have to be satisfied by a single integrated plan. CHEF then tries to find a plan in the case-base which satisfies as many of goals as possible. Once a plan is found, it is modified to match any of the goals that are not satisfied. CHEF works in the Szechwan cooking domain, however, the approach can be applied to others. The basic behaviour of CHEF can be illustrated with the following example:

```
goal: { query( parson, timber_volume ), accuracy( greater, 75 ) }
```

Since the case-base does not have a plan that matches all the goals exactly, a CHEF-type CBR system could retrieve the plan to query timber volume in the Sooke test site and make the appropriate changes.

```
plan:
name: query( timber_volume( parson ) )
operators:
  get_image( parson_image )
  max1_classification( parson, enhanced_mss_image )
  start( mapper )
  get_map( mapper, parson )
  set_active_level( parson, working_vector_level )
  overlay( parson, enhanced_mss_image )
  digitize_vectors( parson, working_vector_level )
  place_db_tags( parson, working_vector_level )
  stop( mapper )
  start( create_database )
  create_database( parson, forest_level )
  stop( create_database )
  start( vector_to_polygon )
  create_polygons( parson, forest_level )
  stop( vector_to_polygon )
  start( mapper )
  get_map( mapper, parson )
  get_cover_level( parson, forest_level )
  query_polygon_cover( parson, timber_volume )
```

This plan is the result of replacing all references to Sooke in the plan retrieved from the case-base with Parson. Of course, such changes do not always assure the formation of successful plans. In CHEF, there is additional knowledge used when forming plans. It can have knowledge of the world that requires, for example, the query must satisfy an accuracy percentage.

Thus the plan above could fail to satisfy CHEF's understanding of the world in that the enhancement of a multi spectral image (enhanced_mss_image) may not provide the level of accuracy that a user requires. CHEF can only discover the failure after having an agent execute the plan. It would then have to remember the failure to avoid the problem in the future. An alternative for the example above would be to use segmentation classification rather than maximum likelihood classification. Segmentation is a classification algorithm that uses an edge detection approach. Once edges have been identified, they are assembled into closed areas called segments and subsequently classified. Maximum likelihood scans the image with a fixed window size to classify each pixel.

Obviously, the problems related to linear planners are completely bypassed since the reasoner does not traverse a search space of world states in search of a solution.

The CHEF algorithm:

GIVEN:

- an initial world state
- a case-base of past problem solutions
- a similarity matrix
- modification rules
- user goals

DETERMINE:

- a plan that satisfies the user goals

ALGORITHM:

- 1.1 anticipate possible problems from user goals based on past successes and failures
- 1.2 using user goals and a similarity matrix, retrieve a base plan that avoids the anticipated problems
- 1.3 apply modification rules and plan critics to modify the base plan so that it accounts for all unsatisfied goals
- 1.4 store plan in case base by indexing with user goals
- 2.1 upon failure, to repair plan:
 - retrieve failed plan
 - retrieve repair strategies (repair strategies are built-in but require domain knowledge in order to be applied)
 - repair plan
- 2.2 store the original plan as a "bad" plan
 - store the repaired plan as a "good" plan
- 2.3 use explanation of failure to create plan failure predictors used in step 1.1

The approach taken in CHEF, the transformational analogy paradigm, is to form plans by using existing, complete plans (cases) and modifying them. Rather than remembering operator selection knowledge, as in derivational analogy, this type of system remembers complete operator sequences and the modifications performed to satisfy new problems. This type of plan formation does not require solution space traversal. The approach forgoes the search space completely. As a result, the system does not need to know any details of the operators, i.e. STRIPS like add and delete lists are not necessary.

The very nature of this approach is its main advantage: it does not require search through a state space. The efficiency of the reasoning system is not based on a search mechanism but rather on the performance of its retrieval and adaptation components.

Also, this type of system can easily solve problems that cannot be solved by pure linear planners. A linear planner must select an operator that will allow it to perform a planning step. This operator selection is biased by the requirement that the operator add to the world a fact that represents a pending goal. For a transformational analogy-based system, all that is needed is a case that solves a similar problem. For these same reasons, this type of reasoning can easily be used to solve goal conjunctions.

The problem that a case-base designer must face is the same as other types of systems. How to acquire knowledge, i.e. cases, modification rules similarity matrices, etc. Contrary to derivational analogy, this type of system relies heavily on an external knowledge source to provide the initial cases, similarity matrices, etc.

Yet another problem that transformational analogy must deal with is failure. A plan can result in failure for two main reasons. The first is that the plan was doomed from the outset, i.e. it is not possible for the plan to satisfy the goals for which it was created. The second is that the plan cannot be applied for lack of resource availability during plan execution, i.e. a plan may be correct but cannot succeed in the current state of the world. By relying on search type planning, it is possible to avoid the first type of failure altogether. It would be possible to avoid the second type of failure by creating conditional plans that can, for instance, ensure the availability of certain resources and take appropriate action when they are not free. This type of plan formation requires knowledge of the operators preconditions as well as the effects it has on the world.

The advantages are clear when attempting to solve problems that are encountered often. The solutions are readily available and can be adjusted through simple modifications. We can expect that, when the distance between problems is small enough, transformational analogy will provide solutions quickly and efficiently. However, a question that begs to be answered is: when does transformational analogy cease to be the best approach to solving a problem? As a corollary, is the success of transformational analogy somehow domain dependent (e.g. is a *divide and conquer* repair strategy applicable to all domains given relevant knowledge)?

The use of transformational analogy is very desirable for our approach since it can quickly provide solutions to often encountered problems. We must define an assessment heuristic that determines whether to try or forgo transformational analogy as a first step.

2.5 Operator Ordering and Merging

Yang [Yang 1992] shows how operator merging can successfully assist in the formation of optimal plans. The example below is drawn once again from the GIS domain. However, at the beginning of the problem solving exercise, the maps are already on disk.

initial world: { disk_map(clayoquot), disk_map(sooke) }

operators:

```
display_map( Map ) ::
    if
        current_map( Map ) and
        running( mapper )
    then add
        display( Map ).

get_map( Map ) ::
    if
        disk_map( Map ) and
        running( mapper )
    then delete
        all( current_map( _ ) )
    add
        current_map( Map ).

get_map( none ) ::
    if
        true
    then delete
        all( current_map( _ ) ).
```

user goal: { display(clayoquot), display(sooke) }

In the operators above, the get_map operator can serve two purposes, the first is to get a map and the second is to remove from the world any fact referring to the current_map.

Attention should be drawn to the merging of existing plans that satisfy subgoals independently. Thus the objective for this example is to merge these two plans:

Goal:

display(clayoquot)	display(sooke)
----------------------	------------------

Plan:

get_map(clayoquot)	get_map(sooke)
display_map(clayoquot)	display_map(sooke)
get_map(none)	get_map(none)

(1)

(2)

In the plans, the `current_map` is set to none after having displayed both the Clayoquot and Sooke maps. If plan (2) were appended to plan (1), the resulting plan would first set the current map to none and then set the current map to sooke. By merging the operators "`get_map( none )`" and "`get_map( sooke )`", the result is a plan that would set the current map to sooke rather than setting the current map to none and then to sooke:

```

get_map( clayoquot )
display_map( clayoquot )
get_map( sooke )
display_map( sooke )
get_map( none )

```

The approach is based on three type of operator interactions:

- An action-precedence interaction is an interaction that requires that an action a in some plan P_i must occur before an action b in some other plan P_j .
- An identical-action interaction occurs when an action in one plan must be identical to an action in one of the other plans.
- A simultaneous-action interaction occurs when action a in plan P_i must happen at the same time as action b in some other plan P_j .

Let A be a set of action $\{ a_1, a_2, a_3, \dots, a_n \}$. Then there may be a merged action $M(A)$ capable of accomplishing the useful effects¹ of all actions in A while leaving the resultant plan correct.

- 1- set $S = \{ P_1, P_2, \dots, P_n \}$ containing one plan P_i for each goal G_i
- 2- set I = interactions among the actions in the plans (precedence-interactions, simultaneous-interactions, identical-interactions)
- 3-
 - for each plan P in S create a graph representing P as a Hasse diagram
 - create a sorted linear index, L , of the actions in the plan
 - for each action-precedence interaction in interaction list I modify the graph by creating a precedence arc between the actions named in the interactions
 - for each simultaneous-action interaction in interaction list I modify the graph by creating a simultaneous arc between the actions named in the interaction
 - for each identical-action interaction in interaction list I combine the actions named in the interaction into a single action by sorting the interaction list and comparing index L (i.e. any identical-action interaction between operators from different plans can be collapsed into a single operator)
 - call the result 'combine(S)'

¹ For planning systems that use STRIPS-type planning operators, the useful effects of a merged action would be that the current world after applying the merged operator would be identical to the current world after applying the set of action in A .

- 4- partition all the actions in S into equivalence classes of actions $E_1, E_2, \dots, E_n$, such that for every set of actions A , the actions in A are merge able iff they are in the same equivalence class
- 5- for each mergeability class E_i of actions in $\text{combine}(S)$ do merge the actions in E_i

This work is very closely related to Sacerdoti's work in NOAH. The main concern is to merge several plans into one. The thrust of the work has to do with recognizing several types of operator interactions and merging plans while accounting for them. The basic idea is to place two sequences of actions, plans, next to each other. Graphically, one can imagine two parallel directed graphs, Figure 2.7 between which there are no connecting edges. As the algorithm progresses, edges are placed between action nodes that interact. When completed, the resulting graph shows the (partial) ordering required to assemble the merged plan. In addition to creating the graph, equivalence classes are defined where each class of operators can be merged into one operator, in effect deleting several operators and changing parameters for others.

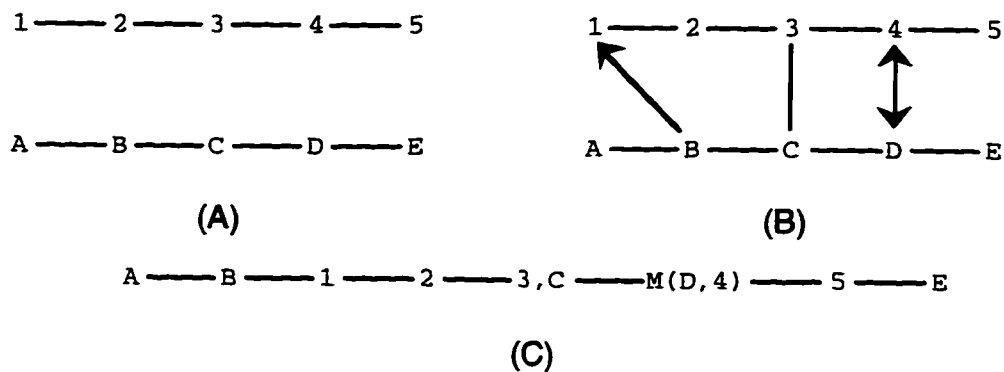


Figure 2.7 - Merging plans and operators

In Figure 2.7, (A) shows two plans before adding the interactions between operators. (B) shows an action-precedence interaction between nodes B and 1 where B must precede 1 in a merged plan. (B) shows a simultaneous-action interaction between nodes C and 3. (B) also shows an identical-action interaction between nodes D and 4. (C) shows one possible merge where operators C and 3 must be applied simultaneously and M(D,4) stands for a merge of operators D and 4.

The advantage to this approach is that it has been shown to successfully merge plans that satisfy independent goals and share resources. However, to be successful, the system must have at its disposal, from the outset, all possible solutions for each goal conjunct. As stated in [Yang 1992], the motivation is that the system must have these plans to allow it create the best merged plan. In addition, the system must have at its disposal of all these plans so that it can succeed. Certain types of anomalies, e.g. Sussman's anomaly, can cause the merge to fail and have the system attempt a new merge with other plans.

A problem solving system could very easily take advantage of this approach by using a case-based reasoner to select plans to merge. Successful merges can be remembered. Unsuccessful merges would have to be explained to avoid similar failures in future.

Also, during merging it may not be possible for this algorithm to succeed. For example, there may exist only one operator that can achieve a given goal, but the application of this operator can leave the world in a state such that the operators in the other plan cannot be applied. In these instances, the system must include the problem operator in the final plan, but it must also change the world state resulting from the application of the operator to a state that would render the other operators applicable, i.e. it must plan.

2.6 Learning - Least General Generalization

Generalization is an important Machine Learning technique that can contribute significantly to a case-based reasoning system. Rather than investing a considerable effort to adapt cases to new situations, the planning effort can be significantly reduced if cases are generalized. Upon retrieval, the unbound elements of the case can be ground to information derived from the current planning episode. When generalized cases are not applicable, the system tries to repair the plan or relies on a search engine. The generalization approach that inspired the generalization procedure integrated into the reasoning system presented in chapter 3 is called *least general generalization* (lgg) [Plotkin 1971]. This approach has also been used in other fields including extensive research in inductive logic programming [Lavrac and Dzeroski 1994; Bergadano and Gunetti 1995]. A generalization is, intuitively, a clausal description of a concept that subsumes one or more instances of the concept. The lgg of a concept is the minimal possible generalization of two or more instances of a concept. The approach was investigated by [Plotkin 1971] and led to lgg under θ -subsumption:

Definition: Clause C θ -subsumes (or is a generalization of) a clause D , if there exists a substitution θ such that $C\theta \models D$. Written $C \preceq D$. C is the least general generalization under θ -subsumption if $C\theta \models D$ and, for every other E such that $E\theta \models D$, it is also the case that $E\theta \models C$. [Bergadano, 1995]

The computation of the lgg of two clauses results from computing the lgg of each term in either clause and is defined as follows:

The lgg of two terms $t_1(l_1, l_2, \dots, l_n)$ and $t_2(m_1, m_2, \dots, m_n)$ is a new variable v if $f_1 \neq f_2$ and is defined as $t_1(\text{lgg}(l_1, m_1), \text{lgg}(l_2, m_2), \dots, \text{lgg}(l_n, m_n))$ if $f_1 = f_2$.

The lgg of two clauses $C_1 = \{t_1, t_2, \dots, t_n\}$ and $C_2 = \{s_1, s_2, \dots, s_m\}$ is defined as:

$\text{lgg}(C_1, C_2) = \{ \text{lgg}(l, s) \mid l \in C_1 \wedge s \in C_2 \mid \text{lgg}(l, s) \text{ is defined} \}.$

The following example shows the output of lgg when applied to two clauses:

$$C_1 = \{ \text{at}(\text{obj1}, \text{loc1}), \text{at}(\text{obj2}, \text{loc1}) \}$$

$$C_2 = \{ \text{at}(\text{obj1}, \text{loc2}), \text{at}(\text{obj1}, \text{loc2}) \}$$

$$\text{lgg}(C_1, C_2) = \{ \text{at}(\text{obj1}, \text{Loc}), \text{at}(\text{obj1}, \text{Loc}), \text{at}(\text{Obj}, \text{Loc}), \text{at}(\text{Obj}, \text{Loc}) \}$$

The lgg procedure will create a new generalized clause whose cardinality is equal to $|C_1| + |C_2|$. After further simplifications, the final clause is:

$$\text{lgg}(C_1, C_2) = \{ \text{at}(\text{obj1}, \text{Loc}), \text{at}(\text{Obj}, \text{Loc}) \}$$

This generalization procedure will be used in PALERMO to generalize solutions retrieved from the case-base to address new problems.

2.7 Other work

Eager derivational replay [Ihrig and Kambhampati 1994] is an approach to derivational analogy where cases are completely replayed upon retrieval and any attempt at plan repair is delayed until replay is complete. Any unsatisfied subgoals are processed after replay by calling on the search engine to modify the plan. For total order planners, eager replay often leads to plan repair failure. This is mainly due to the fact that total order planners can only add operators to the beginning or the end of a plan and that unsatisfied subgoals may require additional operators in the middle of the plan. On the other hand, partial order planners have benefited from eager replay by improved performance. The short replay mechanism presented in 3.2.3 is very similar eager replay in that it eagerly replays the past solution. However, if any unsatisfiable subgoals are encountered then replay is abandoned. As the discussion in the future work section of chapter 6 points out, when the search mechanism for PALERMO is replaced by a partial order engine, a full implementation of eager replay will be used.

The COLLAGE system [Lansky 1991; Lansky and Philpot 1993] provides a framework in which it is possible to integrate several search-based problem solving algorithms. Its main concern is to perform planning by localizing the search activities in restricted areas of the search space and selecting *preferred* algorithms to address different types of problems. For COLLAGE, planning is viewed as a constraint satisfaction problem. It shows the importance of integrating different problem solving approaches to solving complex problems. Moreover, the application described therein is of particular interest since it deals with the use of remote sensing data analysis for land cover information management.

In [Langley et al. 1992], the authors present a mixture of hybrid approaches to planning including the use of macro-operators, search control rules, derivational analogy and transformational analogy. The main argument in the paper is that, traditionally, it has been

assumed that a single planning method is desirable for a given domain, whether it be a hybrid approach, a purely search-based approach or an analogical approach. The authors believe that “there are domains for which mixed approaches are more desirable than any pure method”. A mixed approach would rely on domain-specific knowledge to define classes of problems that would be *better* solved by analogy, by search or by re-use of macro-operators. Of course, determining what is *better* remains the challenge.

Chapter 3

The Problem Solver

Search-based planning is a process whereby a problem solver steps through a space of world states in search of a state where a given goal holds. Minton has identified two classes in which to situate methods used for conducting search: *state-based* search and *plan-based* search [Minton and Zweben 1993]. Classical AI systems such as GPS [Newell and Simon 1963; Ernst and Newell 1969] and STRIPS [Fikes, Hart et al. 1971], are state-based planning systems. In this approach, the problem solver searches for a state where a goal is satisfied by attempting to apply operators to produce new states. The problem solver maintains a *current state* representation where each state results from the application of the operators included in the plan so far. In plan-based search, the problem solver maintains the *current plan* rather than a *current state*. It starts with an empty plan and adds operators to it until a solution is found. When an operator is added, a new partial ordering is imposed on the plan to insure that there are no conflicts between the operators. The NOAH system [Sacerdoti 1975] is a plan-based search-based problem solver.

For both classes of problem solvers, the problem solving processes can use forward search from an initial state or they can use a more sophisticated, goal-directed approach, e.g. means-ends analysis. The main reason for using a goal-directed approach is to focus the search on a much smaller portion of the search space thus rendering tractability to the problem solving mechanism since planning, in the worst case, is undecidable [Chapman 1987] and therefore intractable. Unfortunately, the consequence of reducing the coverage of the search space is to often overlook possible solutions. *Linear planning* is a goal-directed search paradigm where: plans are totally ordered and the solution is found by solving the goals in a *linear* order (i.e. if two or more goals exist then one goal and all of its subgoals are achieved before other goals are considered.) *Non-*

linear planning, i.e. where no search control mechanism is applied, is simply an exhaustive exploration of the whole search space.

Derivational analogy [Carbonell 1986; Veloso 1992] was introduced as a means of providing guidance during search so that the problem solver can be focused when solving problems similar to previously encountered situations. During re-use in derivational analogy (derivational replay [Veloso 1992; Ihrig and Kambhampati 1994; Veloso 1994]), for each decision point, the problem solver uses the successful decision made previously to select the next action to perform. However, when a new problem occurs for which there does not exist an analog, the problem solver must revert to an unfocused, potentially exhaustive search-based process. For PRODIGY/ANALOGY [Veloso 1992] for instance, the search is performed by a non-linear mechanism that covers the search space completely.

Case-based reasoning (CBR) [Bain 1986; Kolodner and Riesbeck 1986; Schank 1986; Collins 1987; Hammond 1989; Riesbeck and Schank 1989] is a non-search-based problem solving approach that forgoes search altogether. Rather than creating a plan from first principle, the problem solver uses a set of existing plans, a case-base, from which the reasoning system can draw. The input to the system is a set of goals that have to be satisfied by a single integrated plan. It then tries to find a plan in the case-base which satisfies as many goals as possible. Once a plan is found, it is modified to match any of the goals that are not satisfied. Rather than remembering operator selection knowledge, as in derivational analogy, this type of system remembers complete operator sequences and the modifications performed to satisfy new problems. Moreover, the system does not need to know any details of the operators, i.e. STRIPS like add and delete lists are not necessary.

The problem solver presented in this chapter integrates the use of the following artificial intelligence techniques:

- goal-regression,
- case-based reasoning,
- plan merging [Sacerdoti 1975; Foulser, Li et al. 1992; Yang 1992],
- derivational analogy,
- machine learning.

Goal-regression is the basic search-based problem solving method used, however, rather than maintaining a representation of the *current state*, the problem solver maintains a *current plan* from which all the states along the derivation path can be readily recalled, including the current state. CBR, or transformational analogy, was then added to the problem solver thus enabling it to explore areas of the search space unreachable via goal-directed search. However, the use of cases within the goal-directed, backward chaining algorithm, required a sophisticated merging mechanism. In fact, there are two *replay* steps involved in integrating a case into a plan under construction by goal-regression: *short replay* and *analogical plan merge*. Short replay is akin to the adaptation of a transformational case, whereas *analogical plan merge* is similar to *derivational*

replay. Finally, when re-use of an existing case for a different problem is successful, then *goal-directed generalization*, derived from machine learning techniques, is used to generalize the case to increase its coverage and to improve the performance of the retrieval mechanism.

Throughout this chapter 2 examples will be used to show how different components of the problem solver behave. The first example is the *one-way-rocket* problem [Veloso 1992] described in chapter 1 and repeated below. The second is the *shaky-the-robot* problem also known as the extended STRIPS domain. A complete specification of these two domains is given in Appendix A.

initial world:

```
at(obj1,loc1)
at(obj2,loc1)
at(rocket,loc1)
```

goal statement:

```
at(obj1,loc2) and
at(obj2,loc2)
```

operators:

<pre>load_rocket(Obj, Loc) :: if at(Obj, Loc) and at(rocket, Loc) then delete at(Obj, Loc) add inside(Obj, rocket)</pre>	<pre>unload_rocket(Obj, Loc) :: if inside(Obj, rocket) and at(rocket, Loc) then delete inside(Obj, rocket) add at(Obj, Loc)</pre>	<pre>move_rocket :: if at(rocket, loc1) then delete at(rocket, loc1) add at(rocket, loc2)</pre>
--	---	---

case:

```
case(
  goal:at(obj1,loc2),
  foot_print:[
    at(rocket,loc1),
    at(obj1,loc1)
  ],
  search_tree:[
    op(load_rocket(obj1,loc1),inside(obj1,rocket)),
    op(move_rocket,at(rocket,loc2)),
    op(unload_rocket(obj1,loc1),at(obj1,loc2))
  ]
)
```

In the *one-way-rocket* problem there is an initial world, a goal statement, a set of three planning operators and a case. The meaning of each of these structures will be explained in more detail in this chapter. Some details are given here to serve as a guide to the chapter. In section 3.1, the problem solver's search mechanism will be laid out. To perform its task, it requires a goal statement, an initial world from which to start and a set of planning operators that describe how changes can be made to the world in order to satisfy the goal statement. In section 3.2, I will show how case-based reasoning, CBR, can be integrated with a search-based problem solver. A case-based reasoning system needs a case base and a goal statement to perform its task. The first step in CBR is retrieval of a relevant case and its adaptation to a new goal. While considering the goal

statement, the problem solver will first use goal regression to solve the first conjunct, *at(obj1,loc2)*. As a result, it will have learned the case shown above. When it attempts to solve the second conjunct, *at(obj2,loc2)*, the retrieval procedure, section 3.2.1, will retrieve the case and make the necessary changes. If an adapted case is successful at satisfying a goal, then it is generalized using goal-directed generalization as is described in section 3.2.2. The adapted case will then be submitted to a replay mechanism called *short-replay*, section 3.2.3, whose task is to ensure that the solution in the adapted case does not violate any of the bindings that would be imposed by the planning operators. Once the changes have been made and the problem solver has a solution for both conjuncts, it will call on the merge procedure, section 3.3, to assemble the complete solution. If a retrieved plan fails to satisfy its goal as a result of merging with another plan, then plan repair, as described in section 3.4, is attempted. Each section begins with a brief discussion of a particular component of the system, followed by a description of the design and implementation of the system component, and ending with an example highlighting the behaviour of the system.

3.1 The search algorithm

Goal regression is a classical problem-solving mechanism that has many incarnations. In the field of Artificial Intelligence, one of the most widely known, implemented systems is STRIPS. I have therefore used this problem solver as the framework upon which to start the integration of the techniques listed above. It should be noted, however, that the integration of case-based reasoning, derivational analogy and machine learning need not be restricted to a STRIPS-like problem solver. In fact, any search-based problem solver could benefit from this type of integration. The main reason for this choice is to show that, although the search space covered with goal-regression is quite narrow and therefore incomplete, it is possible to widen the scope of the coverage of the search while taking advantage of the efficiency of regression.

The search procedure uses a goal-directed approach and is backward-chaining. Actions that can be performed by the planner are expressed in simple operators represented by pre-conditions and effects. The pre-conditions are defined in first-order predicate logic and can contain conjunctions, disjunctions and negation. The effects are represented as add and delete lists. The basic backward-chaining was presented in section 2.1.2.

Basically, if a given goal is true in the current state, the planner exits successfully, otherwise it selects a relevant operator (for a positive goal, a relevant operator adds it to the current state, for a negative goal, the operator deletes it from the current state). If all of the operator's pre-conditions hold in the current state, then it is applied and the current state is modified to reflect the effects specified in the add and delete lists. If the operator's pre-conditions do not hold in the current state, then they become subgoals and the solve procedure is called recursively. As a result, all of the pre-conditions are addressed before any other goal in the goal set. In this way, the problem solver is said to *simulate* the application of the operators. This implementation is a *state-based* planner and, as a result, maintains a current state of the world as opposed to maintaining a current plan. Since the objective is to merge plans created in this fashion with plans created by a

case-based system, it is necessary to have a compatible representation. In consideration of this situation, the basic problem solving algorithm above was modified to maintain a current plan.

3.1.1 Formal definition of the search algorithm

Since the goal of search-based problem solving is to explore a space of states in search of a state where a given goal is true, it is necessary to define states and how they are connected. Let a *state* be a set F of facts where each fact $f \in F$ is a predicate stating that f is true, e.g. $at(obj1, loc1)$ is a fact that states that the object *obj1* is at location *loc1*. Also, the problem solver reasons within the closed-world assumption. A goal is expressed as a conjunction of *desired* facts that must hold true. Given this definition, a plan P is the triple (G, I, O) where:

- G is the goal that requires satisfying,
- I is the *initial state* of the world, i.e. the set of facts that are true when a goal is submitted to the problem solver,
- O contains the operators that, if applied to I , will produce sequence of states; in the last state of the sequence, G holds. O is a search tree represented as a directed acyclic graph or DAG (V, E) [Aho, Hopcroft et al. 1974] where:
 - the set of vertices V represents operator nodes; each operator node $v \in V$ is expressed as a pair (o, g) where o is the complete specification of the operator that was used to satisfy subgoal g (the specification includes the pre-conditions and add and delete lists, this is done to preserve a history of the bindings that led to the solution,)
 - the set of edges E depict the order in which the operators must be applied.

The current state of the world W is the pair (I, Q) where:

- I is the *initial state* of the world,
- Q is the tree already expanded containing operator nodes and edges representing the portion of the space searched, i.e. a possible sub-tree of O .

A problem solving state S is the pair (W, G) where W is the state of the world and G is a goal that must be satisfied. The search procedure, shown in Figure 3.1.2, maps a problem solving state into a problem solving state. This search procedure, implemented in Prolog, is based on the STRIPS algorithm. There are two components of the procedure that should be pointed out: the goal stack and the protected stacks.

The goal stack is used to remember which goals the procedure is currently trying to achieve. If a subgoal has been placed on the stack, then the procedure is currently working on the goal and

will not try to proceed if the expansion of a search node leads to trying to solve the same subgoal. Once the subgoal has been achieved, it is removed from the goal stack.

```

Input: goal G; current state of the world W.
Output: state of the world after satisfying the goal, NewW

solve( empty, W, W ).
solve( G, W, NewW ) :-
    select_goal( H, G, J ),
    solve_one_goal( H, W, MidW ),
    solve( J, MidW, NewW ).

solve_one_goal( G, W, W ) :-
    true_in_world( G, W ),
    protect( G ).
solve_one_goal( G, W, NewW ) :-
    not true_in_world( G, W ),
    not stacked( goal_stack, G ),
    push( goal_stack, G ),
    regress( G, W, NewW ),
    pop( goal_stack ),
    protect( G ).

regress( G, W, NewW ) :-
    relevant_operator( G, op( Operator, PCs, Dels, Adds ) ),
    make_protected_stack( prot( G, Operator ) ),
    solve( PCs, W, MidWorld ),
    protected_goals_ok( Dels ),
    remove_protected_stack( prot( G, Operator ) ),
    append( MidWorld, [ op( Operator, G, PCs, Dels, Adds ) ], NewW ).

```

Figure 3.1.1 - Search procedure

The protected stacks are used to prevent the system from un-doing work it has done. When the procedure retrieves an operator to solve a subgoal, it creates a new set of subgoals, the operator's pre-conditions, that must be achieved. All of the pre-conditions must be true at the same time in order to apply the operator. To prevent the procedure from removing some pre-conditions after having achieved them, they are placed on a protected stack. Once all of the pre-conditions have been achieved, the effects of the operator are applied and the protected stack is removed thus un-protecting the pre-conditions.

This is a typical goal-regression algorithm implemented in Prolog. There are two main predicates that perform the search: *solve* / 3 and *solve_one_goal* / 3. The *solve* procedure receives as input a goal *G* and the state of the world *W*. From this input, it searches for a state of the world *NewW* where the goal is true. Since this implementation maintains a *current plan*, the solution is returned with the output state. The *solve_one_goal* procedure first tests if the goal is true in the current world. If it is, no further search is needed. If it is not, then to prevent *goal loops* (i.e. trying to solve a goal that the procedure is already working on), stack the current goal thus

recording that it is being working on and the regression procedure is called. The regression procedure selects a relevant operator for the current goal and calls the *solve* procedure recursively with the relevant operator's pre-conditions as the goal.

3.1.2 Example of the search algorithm

Let us reconsider the *one-way-rocket* problem from chapter 1. Figure 3.1.3 shows the reasoning trace that the search algorithm above would follow given the goal statement *at(obj1, loc2)* and *at(obj2, loc2)*. Let us assume, for the moment, that the procedure *select_goal* will select a goal from the goal statement on a first-come-first-serve basis with no backtracking (i.e. it will try to solve *at(obj1, loc2)* first and will then try to solve *at(obj2, loc2)* iff a solution to *at(obj1, loc2)* was found.) In the figure, the “->” arrow indicates a procedure call and the “<-” arrow indicates the successful completion of the procedure. The resulting bindings are shown in the successful procedure calls. Also, to simplify the trace, the nodes in the search tree contain only the operator names and their respective parameter bindings rather than the complete operator specifications.

The initial call to the *solve* procedure has three parameters: the first is the initial goal statement (i.e. user goal), the second is the initial world *I*, and the third is the variable that will contain the final world, including the solution, if the search is successful. *Solve* then calls *select_goal* with three parameters: the first is the subgoal which will be expanded, the second is the goal from which it must chose a subgoal, and the third returns the remaining subgoals (i.e. the subgoals that must be solved if a solution is found for the first subgoal.) Once a subgoal has been selected for expansion, the *solve_one_goal* procedure is called. For sake of brevity, the first call to *solve_one_goal* will not be described, suffice it to say that it returns successfully thus producing the mid-state that contains the search tree to solve the first subgoal, the initial world and the resulting *mid-world*. As stated before, a search-based reasoning system must simulate the application of operators included in its solution, therefore, the search for a solution for the second subgoal begins from the *mid-world*. The recursive call to *solve* is done with the subgoal *at(obj2, loc2)* and the initial world *I'* (*mid-world*). The next call to *select_goal* returns the remaining subgoal, *at(obj2, loc2)*.

Solve_one_goal first ensures that the current subgoal is not satisfied and that attempting to resolve the subgoal will not result in a goal loop. It then places the subgoal on a goal stack for subsequent goal loop checks. The next step is to expand the regression from the subgoal. Since the *rocket* is at *loc2* already, this subgoal cannot be satisfied because the *rocket* can move in only one direction. If we relax the assumption that the *select_goal* procedure always selects the first subgoal from a conjunction, then the result would be that before failing, the problem solver would simply attempt to solve the subgoals in a different order, i.e. *at(obj2, loc2)* before *at(obj1, loc2)*.

The algorithm and the example described above show the inherent incompleteness of goal regression. It is a result of its single mindedness in solving only one subgoal at a time and thus ignoring the interactions between subgoals. However, it is possible to increase the scope of the

search space coverage of the algorithm by adding to it the use of case-based reasoning while still preserving that single mindedness that provides focus for the search.

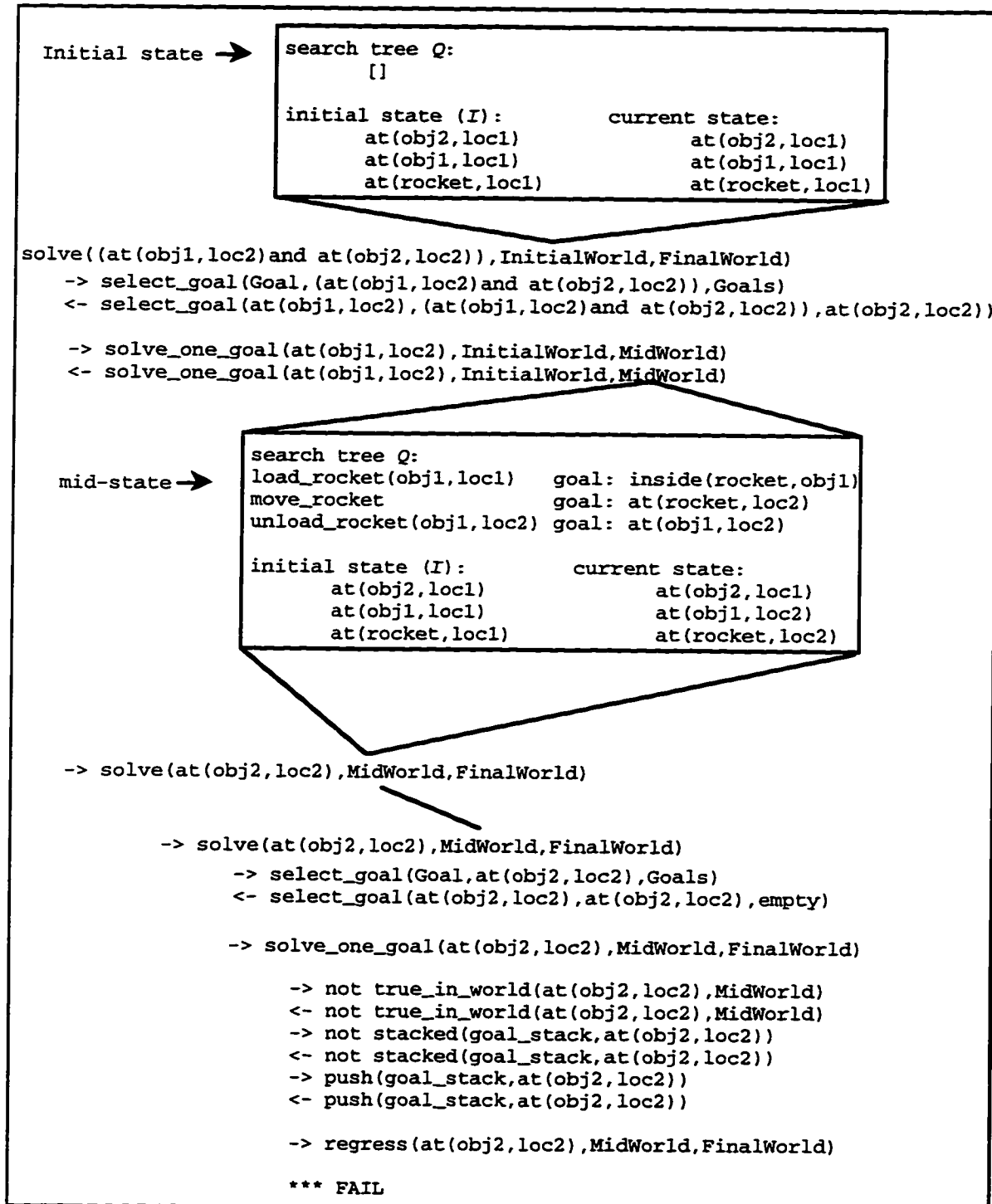


Figure 3.1.2 - One-way-rocket reasoning trace

3.2 Case-based reasoning in goal-regression

The main advantage of case-based reasoning systems is that they solve problems by retrieving existing solutions and adapting them, thereby concentrating on *tweaking* or *massaging* plans into acceptable solutions for new, similar goals. The main disadvantage of CBR systems is that if they are called upon to solve a goal that is not *similar enough* to an existing case, then the system will fail. By adding CBR capabilities to the search algorithm of section 3.1.1, I will show how the disadvantages of either approach can be addressed.

```
plan(  
  goal:at(obj1,loc2),  
  foot_print:[  
    at(rocket,loc1),  
    at(obj1,loc1)  
  ],  
  search tree:[  
    op(load_rocket(obj1,loc1),inside(obj1,rocket)),  
    op(move_rocket,at(rocket,loc2)),  
    op(unload_rocket(obj1,loc1),at(obj1,loc2))  
  ]  
)
```

Figure 3.2.1 - *One-way-rocket* solution for goal *at(obj1, loc2)*

Let us recall the *one-way-rocket* problem. If the search algorithm was given the goal statement *at(obj1, loc2)*, then it would produce the plan shown in Figure 3.2.1 below (as in section 3.1.2, the search tree only includes the operator names and parameter bindings rather the complete operator specifications.) This plan contains a *foot print* [Veloso 1992] that specifies which facts from the initial world are relevant to the plan, i.e. the set of facts that, to be proven to hold, did not require the application of an operator during problem solving. They are the relevant features of a plan, used for indexing and retrieval of cases.

The next step is to give the problem solver access to a case base that already has the case in Figure 3.2.1. Case retrieval, in the context of the search procedure, should be performed when the goal that is selected for expansion is known not to be true in the current world. The changes to the search procedure to allow it to use the case base are shown in Figure 3.2.2. Rather than trying to solve the goal by regression, the *solve_one_goal* procedure first attempts to find a case that satisfies the current goal *G* given the initial world *IW*. If it is successful, then the case must be merged with the current world to produce the solution. As the results in section 4.2.3 show, once the system has learned a sufficient number of cases, analogy outperforms regression thereby justifying attempting to use analogy before search.

```

solve_one_goal( G, InitialWorld, W, NewW ) :-
    \+ true_in_world( G, W ),
    \+ stacked( goal_stack, G ),
    push( goal_stack, G ),
    (
        analogy( G, InitialWorld, Case ),
        merge( InitialWorld, W, Case, NewW )
    ;
        regress( G, InitialWorld, W, NewW )
    ),
    pop( goal_stack ),
    protect( G ).

analogy( Goal, InitialWorld, Case ) :-
    retrieve( InitialWorld, Goal, Footprint, Plan ),
    short_replay( InitialWorld, Plan, Case ).

```

Figure 3.2.2 - Search procedure

If the goal statement *at(obj1, loc2)* were submitted to the problem solver, then the retrieval of this case would be quite straightforward since the goal statement and the goal remembered in the case match. However, if the goal statement is *at(obj2, loc2)* then this case must be retrieved and modified. The *analogy* procedure is responsible for retrieving and adapting a case. The first step is retrieval of the case and is discussed in section 3.2.1. The second and final step in the *analogy* procedure is *short_replay*, described in section 3.2.2. The purpose of this procedure is to verify that the changes made to the case in order to adapt it to the new goal do not violate operator pre-conditions or effects in the retrieved solution.

3.2.1 Definition of the retrieval algorithm

One of the first steps in any case-based reasoning system is retrieval. Most systems rely on similarity matrices that allow them to determine whether two goals are *similar* or not. For instance, a measure of similarity can be the result of climbing an object class hierarchy where instances of objects found under the same class are considered to be similar. This approach is taken in CHEF [Hammond 1989]. Yet another approach is to submit the complete goal statement and initial state to the retrieval mechanism. The similarity measure is the result of summing the number of literals that match between the goal statement and the initial state, and those found in the retrieved case. If the similarity measure is above a given threshold, then the case is considered similar. This is the approach taken in PRODIGY/ANALOGY [Veloso 1992].

In order to define the retrieval procedure, it is necessary to define what a case is and how two objects are similar. A case is defined with respect to the goal-regression search algorithm. Let us recall the definition of a plan $P = (G, I, O)$ where:

- G is the goal,
- I is the initial state of the world,
- O is the DAG representing a search tree for the solution to goal G given a satisfied footprint I .

Definition: a plan P is a case iff the goal statement in the plan contains only one literal, i.e. $|G| = 1$, and the set of facts in I is equal to the footprint.

If we recall the definition of a plan, each node in the search tree O is a pair (o, g) where o is the operator that was chosen to satisfy subgoal g . The specification of the operator in the node includes the operator's pre-conditions, and add and delete lists. For a case, however, the operator contained in the node is reduced to its name and parameter bindings. The plan shown in Figure 3.2.1 in section 3.2 satisfies the definition of a case since the goal has only one literal and the footprint contains only those facts from the initial world relevant to the plan.

Definition: two expressions, $e1$ and $e2$, are similar iff a substitution [Chang and Lee 1973] θ can be found such that if applied to $e1$ will produce $e2$, and the substitution $(e1 / e2)$ is not in θ

Therefore, two goals are similar if the expressions used to represent them have the same functor and arity. The retrieval procedure is shown in Figure 3.2.3. There are two options available: the first looks for exact goal matches, i.e. if cases exist that match the goal exactly, then retrieve those first and verify that the footprint holds in the initial world; the second will retrieve cases, attempt to adapt them to satisfy the current goal given the initial world and subsequently check the footprint.

```

retrieve( World, Goal, FP, Plan ) :-
    get_case( case( goal:Goal, footprint:FP, plan:Plan ) ),
    check_footprint( Goal, FP, World ).
retrieve( World, Goal, Footprint, Plan ) :-
    get_case( Case ),
    gdg( World, Goal, Case, Footprint, Plan ),
    check_footprint( Goal, Footprint, World ).

```

Figure 3.2.3 - Retrieval procedure

When there exists a case that requires no adaptation, the retrieval is straight forward. As well as ensuring that the footprint holds in the initial world, the `check_footprint` call will provide variable bindings for the case.

When there are no cases that match the new goal exactly, and there are cases in the case-base, the retrieval procedure will call the goal directed generalization procedure to attempt to adapt and generalize cases.

3.2.1.1 Goal directed generalization

As it was shown for the *one-way-rocket* example, the solution for the subgoals *at(obj1, loc2)* and *at(obj2, loc2)* are virtually identical save the operator parameters. If the solution were generalized to the case in Figure 3.2.4 then the problem solver would benefit in two ways:

- retrieval would be more efficient,
- adaptation would not be necessary.

```
case(  
  goal:at(A,loc2),  
  foot_print:[  
    at(rocket,loc1),  
    at(A,loc1)  
  ],  
  search_tree:[  
    op(load_rocket(A,loc1),inside(A,rocket)),  
    op(move_rocket,at(rocket,loc2)),  
    op(unload_rocket(A,loc1),at(A,loc2))  
  ]  
).
```

Figure 3.2.4 - Generalized *one-way-rocket* case

Retrieval would be more efficient since, rather than retrieving and examining all of the cases in the case base, the problem solver would rely on Prolog's unification mechanism to directly retrieve the relevant case. Adaptation would not be necessary since at retrieval time the variables in the case would automatically be substituted by the appropriate values by Prolog unification.

Two machine learning techniques inspired the generalization mechanism created for this problem solver. The first is explanation-based learning (EBL) [DeJong and Mooney 1990; Mitchell, Keller et al. 1990] and the second is least-general-generalization (lgg).

EBL uses domain knowledge, an operability criterion and a single example to learn a generalized operational description of a concept. To produce an operational description of a concept, EBL creates a search tree and uses the operability criterion to determine the depth of its search. The search tree leaves are then collected to form the concept description. Explanation-based generalization (EBG) [Mitchell, Keller et al. 1990] can then be applied to this concept

description to produce a more general description that will recognize the example as well as other concept instances that satisfy the description. EBG systems typically perform the generalization during the creation of the search tree.

Least general generalization (lgg) is the minimal possible generalization of two or more instances of a concept. To compute the lgg of two clauses, we must compute the lgg of each literal from the first clause with each literal from the second. The lgg of two literals is a variable if the literals have different functors or arity. The lgg of two literals have the same functor and arity is a new literal with the same functor and arity, and the lgg of their parameters (see section 2.6 for more details).

The problem solver, as described so far, has all of the components necessary to perform EBL. Planning operators serve two purposes: they serve as background knowledge; they also serve as the operability criterion since they each have an executable counterpart (i.e. an agent that executes processes described by the planning operator). The other component required to perform EBL is an example concept. For this problem solver, the example concept is the goal that it must satisfy. What remains is to generalize the cases that are created.

3.2.1.2 Definition of goal directed generalization

When there are no cases that can satisfy the goal without adaptation, the retrieval procedure will attempt to adapt and generalize existing cases. Adaptation and generalization both depend on the successful creation of a substitution that can be applied to the retrieved case such that it accounts for differences between the old and new goals as well as the differences between the case's footprint² and the current world. Goal directed generalization is performed in three steps: computation of a substitution; application of the substitution to the case; and least general generalization. The procedure is shown in Figure 3.2.5.

The substitution computation is the most important step of gdg. Two main concerns must be addressed: first, when a case has been partially generalized, its retrieval will include new bindings with respect to the new goal that must not be undone by applying the new substitution; second, the search must focus on the literals in the footprint that require changes with respect to the new goal and the new initial state of the world.

Variables contained in the case's goal and footprint will bind with information in the current world that holds. When computing a substitution to adapt the case, it is only necessary to substitute ground subterms within the footprint literals. This problem can be addressed by creating a list of footprint literals paired with the variables contained within each literal. This new list of literals can then be used to prevent substituting subterms achieved by grounding variables. Each

² A foot print is the set of facts that must hold in the current world for a case to be applicable.

element of the list is the expression `fact( Fact, Vars )` where *Fact* is the original literal from the footprint and *Vars* is a list of variables contained within the literal.

```
gdg(
    World,
    Goal,
    case( goal:G, footprint:OldFP, plan:OldPlan ),
    Footprint,
    Plan
) :-
    substitution( [ G, Goal ], GoalSub ),
    \+ member( G / _, GoalSub ),

    copy_term(
        case( goal:G, footprint:OldFP, plan:OldPlan ),
        OriginalCase
    ),

    substitution( World, G, Goal, OldFP, InitialSub, Substitution ),
    make_changes( Substitution, OldFP, Footprint ),
    make_changes( Substitution, OldPlan, Plan ),
    lgg(
        [ OriginalCase,
          case( goal:Goal, footprint:Footprint, plan:Plan )
        ],
        case( LGGGoal, LGGFootprint, LGGPlan ),
        Substitution
    ),
    forget_cases( LGGGoal, LGGFootprint, LGGPlan ),
    remember_a_case( LGGGoal, LGGFootprint, LGGPlan ).
```

Figure 3.2.5 - Goal directed generalization

To focus the search for a substitution on literals in the footprint that require change, it is necessary to determine the relevance class of each footprint literal to the goal. The extended example in Appendix B shows why it is necessary to focus the search on certain footprint literals rather than others. Some footprint literals might not hold in the current state of the world, but arbitrarily modifying them in order to make them hold can easily lead to changes that are inconsistent with the case and would therefore have to be undone via backtracking. It is much more desirable to find literals that are apt to lead to an acceptable substitution. This is achieved by the use of relevance classes.

Definition: a footprint literal belongs to the *most* relevant class if the goal and the literal share subterms, i.e. their subterms intersect.

Definition: a footprint literal belongs to the *least* relevant class if it is a negative literal.

Definition: a footprint's relevance class is *undetermined* if it is not negative and the goal and the literal do not share subterms, i.e. the subterm intersection is empty.

A footprint literal that belongs to the *most* relevant relevance class will typically require fewer changes than literals belonging to the other relevance classes since, by definition, some of the subterms should not be changed. Moreover, changes made to literals belonging to the most relevant relevance class create substitutions that must be applied to other literals in the footprint. Once the substitution is applied to the other literals, the new literals may require more changes and therefore the procedure seeks these literals out to compute new substitutions. A footprint literal that belongs to the *least* relevant relevance class cannot be used to compute the substitution. Thus, footprint literals belonging to the *most* relevant class are processed first, then the footprint literals belonging to the *undetermined* relevance class are processed, and finally the *least* relevant literal class is processed.

The substitution computation procedure is shown in Figure 3.2.6. The call to the procedure has five parameters: the current state of the world, the new goal, the footprint, the current substitution and the new substitution. The current state of the world is used to check if the current literal holds as well as provide the facts that the footprint literals must be adapted to. The new goal allows the procedure to determine the relevance class of footprint literals. The footprint literals are modified using the current substitution, the fourth parameter, and checked against the current state of the world. When literals require changes in order to hold in the current state of the world, a new substitution is computed and returned in the last parameter.

The procedure will first determine if the current footprint literal (taken from the beginning of the list) belongs to the *most* relevant relevance class. This is achieved by: checking whether it is a negative literal, applying the current substitution to the literal and finally by checking if the goal literal and the modified footprint literal have identical subterms. If the modified literal does belong to the *most* relevant relevance class then the procedure checks that the literal holds in the current state of the world. If it holds in the current state of the world then the procedure proceeds with the rest of the footprint. If it does not hold in the current state of the world, a new substitution is computed (`more_subs/5`) by replacing subterms that are not in the current substitution or in the subterm intersection with variables. Once replaced by variables, the new literal is matched against the current state of the world in order to find a literal that holds and compute a new substitution. This new substitution is then applied to the goal to ensure that substitution will not affect the goal (`make_changes(MidSub,Goal,Goal)`). The procedure then continues to compute a substitution recursively and upon completion it is applied to the original footprint literal and checked in the current state of the world. This last step is necessary because the final substitution could modify the literal more than the intermediate substitution resulting in a literal that does not hold in the current state of the world. In the event that the literal does not belong to the *most* relevant relevance class, the procedure simply processes the rest of the footprint literals first. The example presented in section 3.2.2 shows how the use relevance classes is applied.

The two remaining steps of `gdg` are substitution application and generalization of the retrieved case and the adapted case. The application of the substitution does not require further explanations. The generalization of the cases is performed by a modified `lgg` algorithm that uses

the newly created substitution to direct the generalization. The procedure uses the substitution to replace disagreeing subterms with variables.

```

substitution( World, Goal, [], Sub, Sub ).
substitution( World, Goal, [ fact( Fact, Vars ) | FP ], Sub, NewSub ) :-
    ( \+ Fact = not( _ ),
      change( Sub, Fact, Vars, NewFact, NewVars ),
      parameter_intersect( Goal, NewFact ) ->
        ( true_in_world( NewFact, World ) ->
            substitution( World, Goal, FP, Sub, NewSub );
          otherwise ->
            more_subs( World, NewFact, NewVars, Sub, MidSub ),
            make_changes( MidSub, Goal, Goal ),
            substitution( World, Goal, FP, MidSub, NewSub ),
            change( NewSub, Fact, NewVars, NewerFact, _ ),
            true_in_world( NewerFact, World )
          );
    otherwise ->
      substitution( World, Goal, FP, Sub, MidSub ),
      make_changes( MidSub, Goal, Goal ),
      change( MidSub, Fact, Vars, NewFact, NewVars ),
      ( true_in_world( NewFact, World ) ->
          NewSub = MidSub;
        \+ Fact = not( _ ) ->
          more_subs( World, NewFact, NewVars, MidSub, Sub2 ),
          make_changes( Sub2, Goal, Goal ),
          substitution( World, Goal, FP, Sub2, NewSub ),
          change( NewSub, Fact, NewVars, NewerFact, _ ),
          true_in_world( NewerFact, World )
        )
    ),
    make_changes( NewSub, Goal, Goal ).

```

Figure 3.2.6 - Substitution computation procedure

3.2.2 Example of the application of the retrieval algorithm

Two examples show the behaviour of the retrieval, adaptation and generalization procedures. The first is a short example based on a slightly modified *one-way-rocket* problem. The second, in Appendix B, shows a more complex example drawn from the logistics transportation domain.

To demonstrate the necessity for a more complex substitution computation, the *one-way-rocket* domain is now composed as shown in Figure 3.2.7.

initial world:

```
at(obj1,loc1)
at(obj2,loc1)
at(rocket,loc1)
```

goal statement:

```
at(obj1,loc2)
```

operators:

```
load_rocket(Obj, Loc) ::      unload_rocket(Obj, Loc) ::      move_rocket ::
if at(Obj, Loc) and          if inside(Obj, rocket) and      if at(rocket, Loc) and
    at(rocket, Loc)          at(rocket, Loc)          not Loc = loc2
then                          then                          then
    delete                  delete                  delete
    at(Obj, Loc)            inside(Obj, rocket)        at(rocket, Loc)
    add                     add                      add
    inside(Obj, rocket)      at(Obj, Loc)              at(rocket, loc2)

case(
    goal:at(obj1,loc2),
    foot_print:[
        at(rocket,loc1),
        at(obj1,loc1)
    ],
    search tree:[
        op(load_rocket(obj1,loc1),inside(obj1,rocket)),
        op(move_rocket,at(rocket,loc2)),
        op(unload_rocket(obj1,loc1),at(obj1,loc2))
    ]
)
```

Figure 3.2.7 - Modified *one-way-rocket*.

The difference is that the `move_rocket` operator can now move from any location to location *loc2*. The case in Figure 3.2.7 was created by submitting the goal statement to the reasoning system while starting from the initial world also given in the Figure. In fact, this specification, including the changes to the operator does not affect in any way the examples given thus far. To show the substitution computation, let us begin with the initial world and goal statement below:

initial world:

```
at(obj1,loc1)
at(obj2,loc3)
at(rocket,loc3)
```

goal statement:

```
at(obj2,loc2)
```

Since the case base does not contain a case that matches this goal exactly, the retrieval procedure will attempt to adapt and generalize the case shown in Figure 3.2.7. As a first step, the procedure computes the substitution required to change the old case into the new case, i.e. *[obj1 / obj2]*. If this substitution is applied to the case, it produces the case in Figure 3.2.8. The application of the plan in this new case will move object *obj2* from location *loc1* to location *loc2*. Obviously, since object *obj2* is in location *loc3* in the initial world, there is a need for more changes to the case.

```

case(
  goal:at(obj2,loc2),
  foot_print:[
    at(rocket,loc1),
    at(obj2,loc1)
  ],
  search tree:[
    op(load_rocket(obj2,loc1),inside(obj2,rocket)),
    op(move_rocket,at(rocket,loc2)),
    op(unload_rocket(obj2,loc1),at(obj2,loc2))
  ]
)

```

Figure 3.2.8 - Adapted case.

The next step in gdg is to compute a substitution that accounts for the differences between the initial world and the footprint. The substitution procedure is called with the initial world, the goal in the case, the new goal, the footprint and the substitution computed to adapt the goal. The procedure will either fail or return a substitution that can be applied to the footprint as well as to the plan.

```

case(
  goal:at(obj2,loc2),
  foot_print:[
    at(rocket,loc3),
    at(obj2,loc3)
  ],
  search tree:[
    op(load_rocket(obj2,loc3),inside(obj2,rocket)),
    op(move_rocket,at(rocket,loc2)),
    op(unload_rocket(obj2,loc2),at(obj2,loc2))
  ]
)

```

Figure 3.2.9 - Adapted case.

The first literal taken from the footprint list is *fact(at(rocket, loc1), [])*. This shows that there were no variables found in the footprint literal. Since this is not a negative literal, the current substitution, *[obj1 / obj2]*, is applied producing the new literal *at(rocket, loc1)*. The new literal is of undetermined relevancy (i.e. the parameter intersection with the goal is empty) and thus further processing of the literal is delayed. The substitution procedure is recursively called with the remaining footprint literal *at(obj1, loc1)*. This literal is not negative and after applying the substitution the system produces *at(obj2, loc1)*. This literal is of most relevance since its parameter intersects with the goal is *[obj2]*. Processing of this literal is immediate. The literal does not hold in the initial world, therefore it is necessary to find a new substitution that will make

it true. The *obj2* subterm cannot be touched since it was introduced by a previous substitution, on the other hand the subterm *loc1* can be replaced by a variable and used to find the value that would make *at(obj2, X)* hold in the initial world. Once this variable is ground, the substitution [*loc1* / *loc3*] has been computed and is then added to the initial substitution to produce [*obj1* / *obj2*, *loc1* / *loc3*]. When applied to the current footprint literal, this substitution ensures that the new literal, *at(obj2, loc3)*, holds in the current world. The substitution procedure terminates its recursive call and now has a substitution that it applies to the first footprint literal, *at(rocket, loc3)*. Since there are no other literals remaining in the footprint, the substitution computation is complete and returns the substitution [*obj1* / *obj2*, *loc1* / *loc3*] which *gdg* now applies to the original footprint and to the original plan to create the new case in Figure 3.2.9.

```

case(
  goal:at(A,loc2),
  foot_print:[
    at(rocket,B),
    at(A,B)
  ],
  search_tree:[
    op(load_rocket(A,B),inside(A,rocket)),
    op(move_rocket,at(rocket,loc2)),
    op(unload_rocket(A,loc2),at(A,loc2))
  ]
)

```

Figure 3.2.10 - Generalized case.

The final step is to apply *lgg* to the two cases to create the generalized case shown in 3.2.10.

3.2.3 Definition of the short replay algorithm

The purpose of replay (also known as *derivational replay* and *analogical replay* [Veloso 1992; Ihrig and Kambhampati 1994; Veloso 1994]) in derivational analogy is to *re-apply* the decisions contained in the old case to the new problem. PALERMO has two replay mechanisms, the first of which is introduced here and is called *short replay*, Figure 3.2.11. The purpose of *short replay* is to ensure that the substitutions that resulted from retrieval do not violate any bindings that would normally be imposed by planning operators and to provide additional bindings that result from binding the complete operator specification with a subgoal.

For every operator in the case, *short replay* will retrieve the complete operator specification, i.e. pre-conditions and effects, and provide bindings for variables found in the add and delete lists. If successful, then the expanded case is considered to be an acceptable case for the new goal and will be returned by the *analogy* procedure to the solve procedure .

```

short_replay(
    Goal,
    [ op( Operator, Goal ) ],
    [ op( Operator, Goal, PCs, Dels, Adds ) ]
) :-
    relevant_operator( Subgoal, op( Operator, PCs, Dels, Adds ) ),
    ( Subgoal = not( NegGoal ),
      member( NegGoal Dels );
      member( Subgoal, Adds ) )
    ).
short_replay(
    Goal,
    [ op( Operator, Subgoal ) | Case ],
    [ op( Operator, Subgoal, PCs, Dels, Adds ) | ExpandedTree ]
) :-
    relevant_operator( Subgoal, op( Operator, PCs, Dels, Adds ) ),
    ( Subgoal = not( NegGoal ),
      member( NegGoal Dels );
      member( Subgoal, Adds ) )
    ),
    short_replay( Goal, Case, ExpandedTree ).

```

Figure 3.2.11 - Short replay procedure

3.2.4 Example of the application of the short replay algorithm

To better illustrate how the *short replay* procedure behaves, consider the changes to the *one-way-rocket* problem specification below. The difference is that locations and objects are specified with a parameterized expression, e.g. *loc(l)* rather than *loc1*.

This expressive power can be used to include implicit pre-conditions and effects in operators. For instance, by using this approach it is possible to impose type constraints on the objects manipulated by the operators as in the operator:

```

load_rocket( obj( Obj ), loc( Loc ) ) ::
if
    at( obj( Obj ), loc( Loc ) ) and
    at( rocket, loc( Loc ) )
then
    delete at( obj( Obj ), loc( Loc ) )
    add
        inside( rocket, obj( Obj ) )

```

The advantage of using this operator over the one specified in chapter 1 is that when the search algorithm tries to find a relevant operator to move the rocket (i.e. add the fact that the rocket is at location *loc2*), the operator above will not be considered thus adding the implicit constraint that only objects matching *obj(Object)* will result in applying the *load_rocket* operator.

Initial state: at(obj(1), loc(1)) at(obj(2), loc(1)) at(rocket, loc(1))	Goal statement: at(obj(1), loc(2))
--	---------------------------------------

```

case(
  goal:at(obj(2),loc(2)),
  foot_print:[
    at(rocket,loc(1)),
    at(obj(2),loc(1))
  ],
  search tree:[
    op(load_rocket(obj(2),loc(1)),inside(obj(2),rocket)),
    op(move_rocket,at(rocket,loc(2))),
    op(unload_rocket(obj(2),loc(1)),at(obj(2),loc(2)))
  ]
).

```

Figure 3.2.12 - New *one-way-rocket* specification

On the other hand, this type of implicit constraint introduces a different kind of problem: substitutions computed by the retrieval procedure may result in binding violations. For instance, given the *one-way-rocket* specification in Figure 3.2.12, if the retrieval procedure is called with the goal *at(obj(1), loc(2))*, then the substitution necessary to transform the goal statement into the goal satisfied by the case is *[2 / 1]*. Since this satisfies the similarity criteria defined in section 3.2.1, the retrieval procedure will return the case in Figure 3.2.13. Let us recall the *move_rocket* operator. If we cast it in the same form as the *load_rocket* operator above, we have:

```

move_rocket ::
if                at( rocket, loc( 1 ) )
then delete at( rocket, loc( 1 ) )
   add          at( rocket, loc( 2 ) )

```

As stated before, the result of applying this operator is to move the rocket from location *loc(1)* to location *loc(2)*. The retrieved case shown in Figure 3.2.13 specifies that the *move_rocket* operator is used to satisfy the goal *at(rocket, loc(1))*. The *short replay* procedure will attempt to unify the *move_rocket* operator above with the *move_rocket* operator in the case. Since the *move_rocket* operator cannot add the fact *at(rocket, loc(1))* to the world, this case violates the bindings provided by the operator specification and will therefore be rejected. Moreover, as can be seen in the last operator in the search tree, the object *obj(1)* is not unloaded at location *loc(2)* but rather at location *loc(1)*. Of course this does not satisfy the goal statement and the case should be rejected.

```

case(
  goal:at(obj(1),loc(2)),
  foot_print:[
    at(rocket,loc(1)),
    at(obj(1),loc(1))
  ],
  search_tree:[
    op(load_rocket(obj(1),loc(1)),inside(obj(1),rocket)),
    op(move_rocket,at(rocket,loc(1))),
    op(unload_rocket(obj(1),loc(1)),at(obj(1),loc(1)))
  ]
).

```

Figure 3.2.13 - Case retrieved for goal *at(obj(2), loc(2))*

3.3 Merging plans

The purpose of case retrieval is to find cases that can solve new problems that are similar to ones encountered before. The purpose of short-replay is to expand cases so that they satisfy the definition of plans discussed in section 3.1.1 and to verify variable bindings to ensure that operator pre-conditions are not violated. Cases are returned as complete plans so that they can be merged with other plans when more than one goal must be solved. Plan merging, the second replay mechanism, combines two or more plans into one plan while considering the interactions among subgoals and between operators [Sussman 1973; Sacerdoti 1975; Veloso 1992; Yang 1992]. Interactions between subgoals means that each subgoal in a goal statement is either completely independent of the other subgoals, completely dependent of the other subgoals, or shares resources with the other subgoals (the term resource should be interpreted as to include any process, object, operator, etc. referred to in a solution.)

Selecting a merging strategy is directly related to the type of goal interaction the problem solver is faced with. The three types of goals listed above are:

Independent goals: these are goals that can be solved completely separately and, when merged, all of the vertices and edges in their respective search trees are included in the final solution.

Dependent goals: these are goals where achieving one goal results in satisfying other goals thereby allowing the problem solver to forgo explicitly trying to solve them.

Resource sharing goals: these are goals that can be solved completely separately, however, since some vertices may have the same subgoals, they may be redundant and can be discarded along with the edges that lead to them when they are merged.

There are three merging strategies: serial, round-robin and exploratory [Veloso 1993]. In derivational analogy, these three strategies are used to select the decision nodes that will be expanded upon during replay. In serial merging, cases are randomly selected and the decision nodes are all replayed before selecting a second case to merge. In round-robin merging, replay is done by following a step of each case at a time. In exploratory merging, cases are replayed in arbitrary order and each step is chosen from a randomly selected case.

Since both the search-based and analogical components of PALERMO are plan based, they both return plans rather than decision nodes. These plans are ordered by the edges included in their respective search trees. The merge performed here results in the union of the sets of vertices in the search tree and the selection of edges from both trees such that every vertex is reached. Each edge included in the final solution must reflect the interactions between operators from either plan. In NOAH [Sacerdoti 1975], the selected edges are submitted to two critics: the eliminate redundant pre-conditions critic and the resolve conflict critic. Yet another approach is to identify operator interactions such as action-precedence, identical-action or simultaneous-action [Yang 1992].

The plan merge algorithm follows both serial and exploratory type merging with a intelligent casual commitment strategy [Minton, Knoblock et al. 1989] while considering operator interactions. Casual commitment allows the problem solver to commit to a certain ordering on a working solution and to backtrack to try another ordering whenever failure is encountered. Independent goals are merged using a serial approach. Resource sharing goals are merged using an exploratory approach. Finally, let us consider two dependent goals g and h , where whenever g is achieved, h is achieved. If g is solved first, then h will already be true when submitted to the problem solver. If h is solved first, then both the set of vertices and set of edges from its search tree will be a subset of the set of vertices and the set of edges from the search tree for the solution to g and can therefore be discarded.

3.3.1 Definition of the merge algorithm

The objective of plan merging is to combine two or more plans into one. More specifically, a successful plan merge will combine the search trees of each plan into one search tree. Let us therefore recall the definition of a search tree. A search tree O is represented as a directed acyclic graph (V, E) where:

- the set of vertices V represents operator nodes; each operator node $v \in V$ is expressed as a pair (o, g) where o is the complete specification of the operator that was used to satisfy subgoal g ,
- the set of edges E represents the order in which the operators must be applied.

As discussed in section 3.1.1, edges represent the order in which operators are included in a plan. When merging two plans, the operators taken from one plan can affect the applicability of operators in the other plan. They can also make some operators redundant or have other effects

when the plans are merged. Hence in order to outline the merge algorithm, it is necessary to define the following: current operators, redundant operators, applicable operators and threatening operators. At any point during merging, there are always two current operators, one from either plan, unless one plan has already been completely merged into the final solution.

Definition: an operator is current when the last edge of the plan from which it was taken leads to its vertex.

Definition: an operator is redundant if:

- it is identical to the current operator from the other plan, or
- the subgoal in its vertex has already been satisfied.

Definition: an operator is applicable iff all of its pre-conditions are satisfied in the current world.

Definition: an operator threatens another iff its application results in removing some of the pre-conditions of the other.

Since the problem solver maintains a *current plan* rather than a *current state*, the merge procedure can be expressed by the formula $P \times P \times P \rightarrow P$, i.e. the merge procedure maps three problem solving states into a problem solving state. The three problem solving states are three plans that must be merged into one. The two first plans are used to satisfy subgoals that are unsatisfied in the current world (the third plan.) The plan that defines the current world is correct, i.e. each operator o such that $(o, g) \in V$ is applicable with respect to the ordering represented by the edges contained in E . Moreover, the plans that satisfy the subgoals are initially correct with respect to the current world (this was achieved by either *short replay* or the search algorithm.) As the merge proceeds, operator vertices are added to the current world's search tree and can thus affect the correctness of the remaining plans.

The first part of the merge procedure merges two non-empty plans and the current state of the world into a new state of the world. There are three different situations that must be dealt with: the first is when one of the two current operators is considered redundant, the second is when one operator is applicable in the current state and the other is not, and the final situation is when both are applicable and neither is redundant.

The intuition behind the merge algorithm is that if an operator is redundant it should be discarded. If the current operator from one plan is applicable and the current operator from the other is not, then add the applicable operator to the current world and proceed with the merge. Hopefully, as we add operators to the current world, its state will be changed in such a way as to render the current inapplicable operator applicable. Finally, if both current operators are applicable then we must select one to apply. Applying one operator might affect the applicability of applying another, for instance, applying *move_rocket* before applying *load_rocket*, will make the *load_rocket* operator's pre-conditions false, i.e. it is not possible to load an object into the rocket if the object and the rocket are at different locations. If the application of one operator affects the applicability of the second but the application of the second does not affect the applicability of the first, then the choice is to add the second operator to the current world and proceed with the merge.

A problem occurs when both the application of the first operator affects the applicability of the second and the application of the second affects the applicability of the first. The problem is called mutual pre-condition clobbering [Sussman 1973] and is well known as the Sussman anomaly. To address the Sussman anomaly, one operator is chosen and added to the current world, and the merge proceeds. The hope is that, after adding operators to the world, the clobbered pre-conditions will become true again thus allowing the threatened operator to be applied.

```
merge([Operator1|Case1],[Operator2|Case2],World,NewWorld):-
    redundant(Operator2,Operator1,World),
    merge(Case2,[Operator1|Case1],World,NewWorld).

merge([Operator1|Case1],[Operator2|Case2],World,NewWorld):-
    redundant(Operator1,Operator2,World),
    merge([Operator2|Case2],Case1,World,NewWorld).

merge([Operator1|Case1],[Operator2|Case2],World,NewWorld):-
    not applicable(Operator2,World),
    applicable(Operator1,World),
    append(World,[Operator1],MidWorld),
    merge([Operator2|Case2],Case1,MidWorld,NewWorld).

merge([Operator1|Case1],[Operator2|Case2],World,NewWorld):-
    not applicable(Operator1,World),
    applicable(Operator2,World),
    append(World,[Operator2],MidWorld),
    merge(Case2,[Operator1|Case1],MidWorld,NewWorld).

merge([Operator1|Case1],[Operator2|Case2],World,NewWorld):-
    applicable(Operator1,World),
    applicable(Operator2,World),
    not redundant(Operator2,Operator1,World),
    not redundant(Operator1,Operator2,World),
    ( not threatens(Operator2,Operator1),
      append(World,[Operator2],MidWorld),
      merge(Case2,[Operator1|Case1],MidWorld,NewWorld);
      not threatens(Operator1,Operator2),
      append(World,[Operator1],MidWorld),
      merge([Operator2|Case2],Case1,MidWorld,NewWorld);
      threatens(Operator2,Operator1),
      append(World,[Operator2],MidWorld),
      merge(Case2,[Operator1|Case1],MidWorld,NewWorld);
      threatens(Operator1,Operator2),
      append(World,[Operator1],MidWorld),
      merge([Operator2|Case2],Case1,MidWorld,NewWorld)
    ).
```

Figure 3.3.1 - Merging non-empty plans.

Figure 3.3.1 shows the implementation of the merge procedure that reflects the discussion above. There are five clauses in the procedure. The first two deal with redundant operators by discarding them. The next two deal with the case where one operator is applicable and the other is

not. The applicable operator is added to the current world, thus creating a *mid-world* which will in turn become the current world for the next merge step.

The last clause in Figure 3.3.1 handles the case when both operators are applicable and neither is redundant. These conditions are checked first. Then if one operator threatens the other but is itself not threatened, its application is postponed until the threatened operator has been added to the current world. Finally, if both threaten each other, then one is selected and the merge proceeds. The two last options tested in the procedure, i.e. *threatens(Operator2, Operator1)* and *threatens(Operator1, Operator2)*, attempt to resolve the Sussman anomaly by selecting one operator over the other. If the procedure fails with the first operator it selects, it will try with the second operator.

3.3.2 Example of the application of the merge algorithm

Consider the cases in Figure 3.3.2. For simplicity, only the plans are shown. Case A is the plan retrieved for the goal statement *at(obj1, loc2)* and case B is the case retrieved for the goal statement *at(obj2, loc2)*. In this section, the merge procedure is given these two plans to merge.

Case A:

```
goal: at( obj1, loc2 )

load_rocket( obj1, loc1 )
move_rocket
unload_rocket( obj1, loc2 )
```

Case B

```
goal: at( obj2, loc2 )

load_rocket( obj2, loc1 )
move_rocket
unload_rocket( obj2, loc2 )
```

Figure 3.3.2 - Two cases to merge

The merge begins as shown in Figure 3.3.3. Case A is bound with the variable *Case1* and case B is bound with the variable *Case2*. The plan representing the current world is empty. The current operator for case A is *load_rocket(obj1, loc1)* and the current operator for case B is *load_rocket(obj2, loc1)*. The clause that is used for the first step of the merge is the last one shown in Figure 3.3.1. This occurs since both current operators are not redundant and both are applicable. Figure 3.3.3 shows that these redundancy and applicability tests occur. Then it is shown that the application of *load_rocket(obj2, loc1)* in the current (empty) world would not threaten the applicability of the operator *load_rocket(obj1, loc1)*. As a result, the operator *load_rocket(obj2, loc1)* is added to the current world and the merge procedure is called again with the original case A, the remainder of case B and the new current world.

Figure 3.3.4 shows the planning state after the *load_tocket(obj2, loc1)* has been added to the current world. Figure 3.3.5 shows the next call to the merge procedure given the current planning state in Figure 3.3.4.

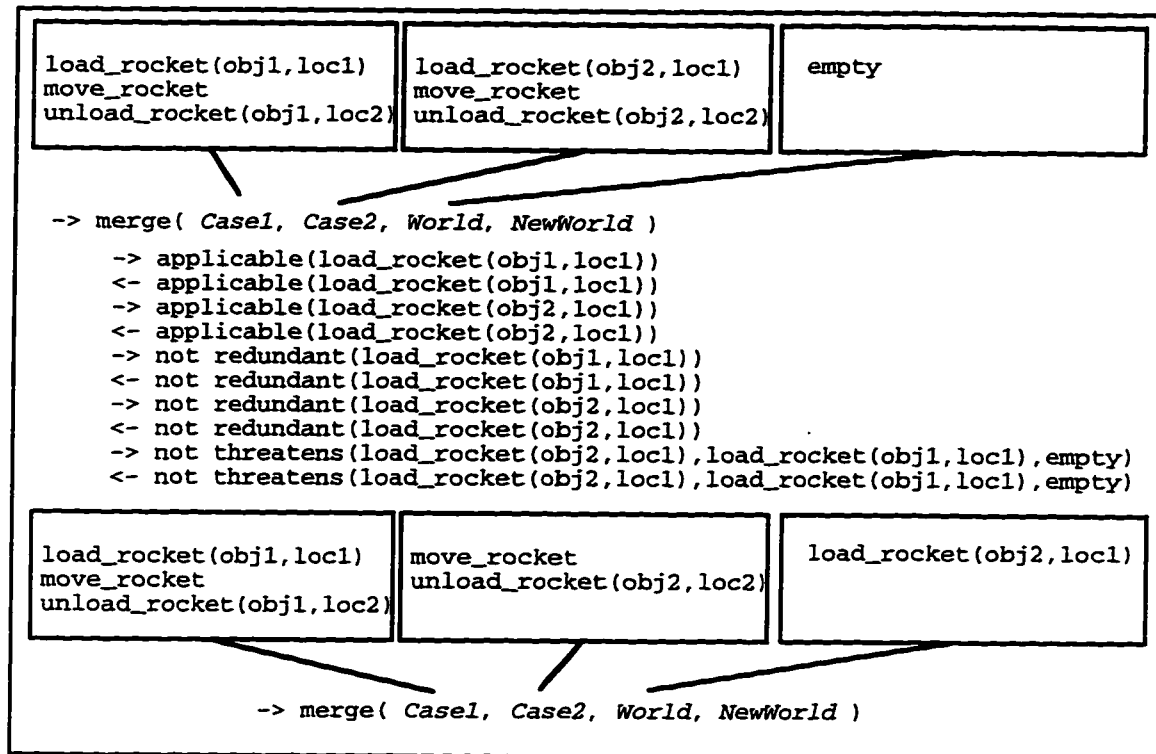


Figure 3.3.3 - Merge: *load_rocket(obj2, loc1)* is added to the current plan

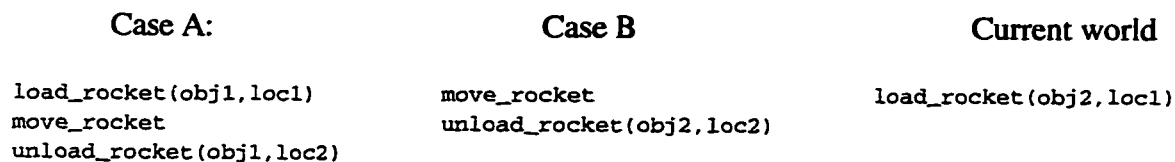


Figure 3.3.4 - Current planning state after adding *load_rocket(obj2, loc1)* to the plan.

The current operator for case A is *load_rocket(obj1, loc1)* and the current operator for case B is *move_rocket*. The clause that is used for the second step of the merge is again the last one shown in Figure 3.3.1. This occurs since both current operators are not redundant and both are applicable. Figure 3.3.5 shows that these redundancy and applicability tests occur. Then it is shown that the application of *move_rocket* in the current world would threaten the applicability of the operator *load_rocket(obj1, loc1)*. Since this test fails, the next test is to check if the application of *load_rocket(obj1, loc1)* would threaten *move_rocket*. This test succeeds and as a result, the operator is added to the current world and the merge proceeds.

Figure 3.3.6 shows the planning state after the *load_tocket(obj1, loc1)* has been added to the current world. Figure 3.3.7 shows the next call to the merge procedure given the current planning state.

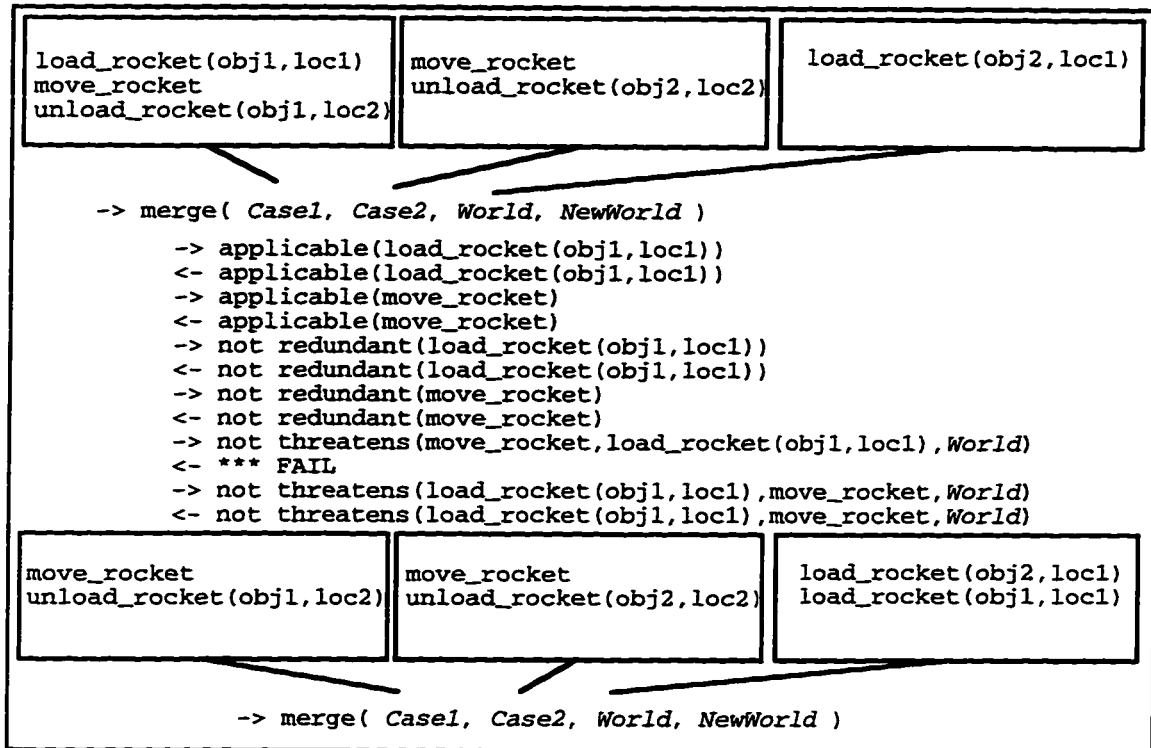


Figure 3.3.5 - Merge: *load_rocket(obj1, loc1)* is added to the current plan

Case A:	Case B	Current world
move_rocket unload_rocket(obj1,loc2)	move_rocket unload_rocket(obj2,loc2)	load_rocket(obj2,loc1) load_rocket(obj1,loc1)

Figure 3.3.6 - Current planning state after adding *load_rocket(obj1, loc1)* to the plan.

The current operator for case A is now *move_rocket* and the current operator for case B is *move_rocket*. The clause that is used for the third step of the merge is the first one shown in Figure 3.3.1. The redundancy test performed on the *move_rocket* operator from case A succeeds and, as shown in Figure 3.3.7, the operator is discarded resulting in the planning state shown in Figure 3.3.8.

The current operator for case A is now *unload_rocket(obj1, loc2)* and the current operator for case B is *move_rocket*. Since the neither operator is redundant, and the current operator from case A is applicable and the current operator from case B is not applicable, fourth clause shown in Figure 3.11 is used. The effect is to add the *move_rocket* operator to the current world (Figure 3.3.9).

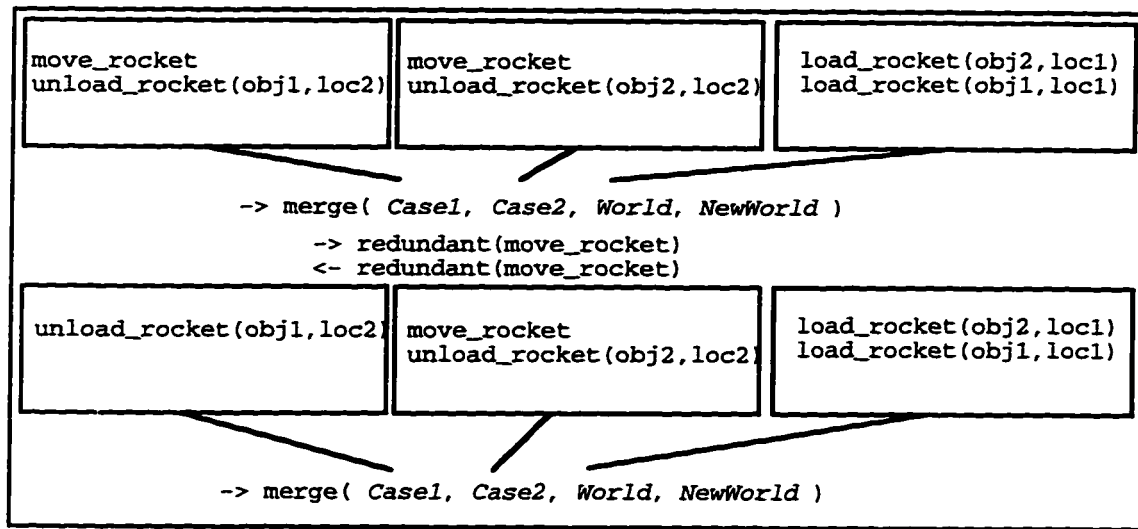


Figure 3.3.7 - Merge: *move_rocket* is redundant and discarded

Case A:	Case B	Current world
unload_rocket(obj1, loc2)	move_rocket unload_rocket(obj2, loc2)	load_rocket(obj2, loc1) load_rocket(obj1, loc1)

Figure 3.3.8 - Current planning state after discarding *move_rocket* from case A.

Case A:	Case B	Current world
unload_rocket(obj1, loc2)	unload_rocket(obj2, loc2)	load_rocket(obj2, loc1) load_rocket(obj1, loc1) move_rocket

Figure 3.3.9 - Current planning state after adding *move_rocket*.

Finally, the rest of the merge proceeds much like it did while adding the *load_rocket* operators to the current world. Both the remaining *unload_rocket* operators are not redundant, both are applicable and each of them does not threaten the application of the other. Moreover, it is worth noting that the problem solver is not committed to the order of the resulting plan. Thanks to the backtracking mechanism of Prolog, the merge algorithm can undo this ordering in favour of another if a failure is encountered or if an external critic, e.g. human critic or a critic module, is not satisfied with the plan.

3.4 Plan repair

It was shown in sections 3.3.1 and 3.3.2 that plan merging could combine two plans into one while taking into account subgoal interaction between operators from either plan. The example showed that the “repairs” needed to successfully merge the plans were to simply discard the

redundant operator *move_rocket*. However, more serious problems can occur during merge. As discussed in 3.3.1, when the Sussman anomaly is encountered, the strategy is to select one of the plans and add its operators to the final solution until the clobbered pre-conditions of the other plan become satisfied once again. The merge algorithm in Figure 3.3.1 will behave in this manner. If, in the event of a Sussman anomaly, the merge algorithm adds all of the operators from the selected plan to the final solution and the clobbered pre-conditions are still unsatisfied, then there are two options: abandon the merge or attempt to satisfy the clobbered pre-conditions.

3.4.1 Definition of the plan repair

PALERMO will attempt to repair the plan by satisfying the clobbered pre-conditions. The merge procedure is expressed by the formula $P \times P \times P \rightarrow P$, i.e. the merge procedure maps three problem solving states into a problem solving state. Viewing the repair strategy in a similar notation simplifies the mapping to the formula $P \times P \rightarrow P$ that maps two problem solving states into one problem solving state. In [Hammond 1989], for instance, there are several repair strategies. One strategy, dubbed *divide and conquer*, will split the steps listed in a plan according to criteria provided by a domain expert.

A plan is in need of repair when a subgoal taken from a case is not satisfied. A simple approach to repairing is to submit the unsatisfied subgoal to the problem solver. It will either have a case that can be merged to the current world or it will rely on search. This discussion is reflected in the remaining clauses (Figure 3.4.1) of the merge algorithm. When merging two cases into the current plan, if one of them is empty then if the current operator is not applicable then solve its pre-conditions and proceed with the merge.

```
merge( Case1, [], World, NewWorld ) :-
    merge( Case1, World, NewWorld ).
merge( [], Case2, World, NewWorld ) :-
    merge( Case2, World, NewWorld ).

merge( [], World, World ).
merge( [ Op | Case ], World, NewWorld ) :-
    ( applicable( Op, World ) ->
        append( World, [ Op ], MidWorld ),
        merge( Case, MidWorld, NewWorld );
        otherwise ->
            get_subgoal( Op, SubGoal ),
            solve( SubGoal, World, MidWorld ),
            merge( Case, MidWorld, NewWorld )
    ).
```

Figure 3.4.1 - Merging and plan repair.

3.4.2 Example of plan repair

The examples that have been presented above were all based on the *one-way-rocket* problem. However, to illustrate how other problems arise and are addressed, let us consider the cases in Figure 3.4.2 drawn from the extended STRIPS domain. Case A solves the goal *setting(light(1), on)*, i.e. the problem is to have the robot turn the light *light(1)* on. Similarly, case B solves the goal *setting(light(2), on)*. Some details have been omitted for the sake of space, namely the subgoals which prompted the use of the operators specified in the cases.

Case A	Case B
<pre> case(goal:setting(light(1),on), foot_print:[onfloor, not(closed(door(a))), not(closed(door(b))), not(nextto(robot,box(1))), not(nextto(robot,door(a))), not(nextto(robot,door(b))), pushable(box(1)), inroom(robot,room(1)), inroom(box(1),room(3)), inroom(door(a),room(1)), inroom(door(b),room(2)), inroom(door(b),room(3)), inroom(light(1),room(2)), connects(door(a),room(1),room(2)), connects(door(b),room(2),room(3))], search tree:[goto(door(a),room(1)), gothru(door(a),room(1),room(2)), goto(door(b),room(2)), gothru(door(b),room(2),room(3)), goto(box(1),room(3)), pushto(box(1),door(b),room(3)), pushthru(door(b),box(1),room(3),room(2)), pushto(box(1),light(1),room(2)), climbon(box(1)), turn(light(1),on)]) </pre>	<pre> case(goal:setting(light(2),on), foot_print:[onfloor, not(closed(door(a))), not(closed(door(b))), not(closed(door(c))), not(nextto(robot,box(2))), not(nextto(robot,door(a))), not(nextto(robot,door(b))), not(nextto(robot,door(c))), pushable(box(2)), inroom(robot,room(1)), inroom(box(2),room(4)), inroom(door(a),room(1)), inroom(door(b),room(2)), inroom(door(c),room(3)), inroom(light(2),room(4)), connects(door(a),room(1),room(2)), connects(door(b),room(2),room(3)), connects(door(c),room(3),room(4))], search tree:[goto(door(a),room(1)), gothru(door(a),room(1),room(2)), goto(door(b),room(2)), gothru(door(b),room(2),room(3)), goto(door(c),room(3)), gothru(door(c),room(3),room(4)), goto(box(2),room(4)), pushto(box(2),light(2),room(4)), climbon(box(2)), turn(light(2),on)]) </pre>

Figure 3.4.2 - Two cases from the extended-STRIPS domain.

Merging the two plans would proceed much like it did in the *one-way-rocket* example. Both plans are identical up to the operator *gothru(door(b), room(2), room(3))* for entering room *room(3)*. The operators up to that point are either discarded because they are redundant or they are added to the *current plan* (shown in Figure 3.4.3).

The two current operators are: *goto(box(1), room(3))* from case A and *goto(door(c), room(3))* from case B. Both of these operators are applicable and neither threatens the pre-conditions of the other, therefore, one is selected and added to the current world. If the chosen operator is *goto(door(c), room(3))* then the two new current operators are: *goto(box(1), room(3))* from case A and *gothru(door(c), room(3), room(4))* from case B. A Sussman anomaly has occurred since going through the door will clobber the pre-conditions of going next to the box and going to the box will clobber the pre-conditions of going through the door. Had the chosen operator been *goto(box(1), room(3))*, then a Sussman anomaly would also have occurred, therefore let us continue with the example as it has been described so far. The current planning state is shown in Figure 3.4.4.

```
goto(door(a),room(1)),
gothru(door(a),room(1),room(2)),
goto(door(b),room(2)),
gothru(door(b),room(2),room(3)),
```

Figure 3.4.3 - The current world after adding the operators necessary for the robot to reach room *room(3)*.

Case A

```
goto(box(1),room(3)),
pushto(box(1),door(b),room(3)),
pushthru(door(b),box(1),room(3),room(2)),
pushto(box(1),light(1),room(2)),
climbon(box(1)),
turn(light(1),on)
```

Case B

```
gothru(door(c),room(3),room(4)),
goto(box(2),room(4)),
pushto(box(2),light(2),room(4)),
climbon(box(2)),
turn(light(2),on)
```

Current world

```
goto(door(a),room(1)),
gothru(door(a),room(1),room(2)),
goto(door(b),room(2)),
gothru(door(b),room(2),room(3)),
goto(door(c),room(3)),
```

Figure 3.4.4 - The current planning state.

When a Sussman anomaly occurs, a case is chosen and its current operator is added to the current world. Let us assume that the chosen operator is *gothru(door(c), room(3), room(4))*. After adding this operator to the current world, the pre-conditions for the operator *goto(box(1), room(3))* are no longer satisfied. The merge procedure will therefore continue and add the operators from case B to the current world. At each step, the merge algorithm will check the pre-conditions of the current operator from case A. Since the pre-conditions for the operator *goto(box(1), room(3))* will never be satisfied, all of the operators from case B will be added to the current world. As a result, there are no operators left in the case that could satisfy the clobbered pre-conditions. This state is shown in Figure 3.4.5.

As shown in Figure 3.4.1, when a case is empty the merge proceeds by merging one case into the current plan. Since one of the pre-conditions of the current operator from case A is *inroom(robot, room(3))* and is not satisfied, the merge algorithm will attempt to repair this problem by solving this subgoal in the current world (Figure 3.4.5). If a case does not exist to satisfy this subgoal, then search is used thus producing the solution in Figure 3.4.6(a) and the repair is remembered as a new case, Figure 3.4.6(b). Repairing a plan can result in the system learning new cases.

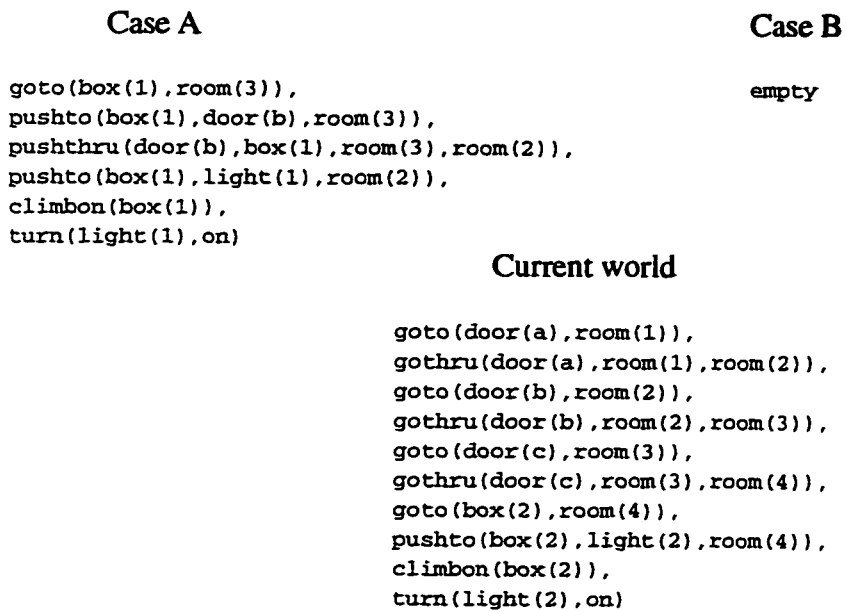


Figure 3.4.5 - The current planning state.

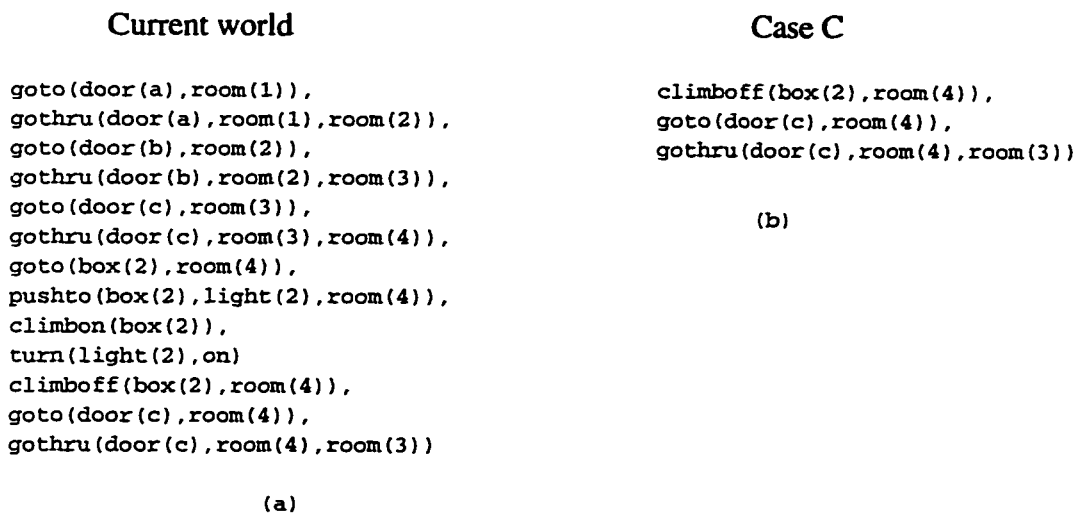


Figure 3 .4.6 - Plan repair.

Chapter 4

Empirical Results

The previous chapters have described the design and implementation of a problem solving system. The design was based on the integration of several different problem solving approaches in order to take advantage of each of their strengths while attempting to avoid their weaknesses. The problem solver is able to reason from first principle by using a domain theory as well as perform analogical reasoning based on past experience. The main features of the problem solver are:

- performs non-linear planning,
- creates total order plans,
- draws on both transformational and derivational analogy,
- uses a plan-based search state representation.

This chapter presents the empirical results obtained after running the problem solver on four domains: the *one-way-rocket* domain, the *extended-STRIPS* domain, the *logistics transportation* domain and the *digital forest cover map update* domain [Charlebois, Goodenough et al. 1994; Goodenough, Charlebois et al. 1994; Matwin, Charlebois et al. 1994]. The *one-way-rocket* domain was used to show how non-linearity could be achieved by a linear problem solver by adding the eager use of analogy. The *extended-STRIPS* domain was used to ensure that the problem solver could perform independently of the application domain. The *logistics transportation* domain was used in [Veloso 1992] and was the main evaluation domain for PRODIGY/Analogy and is the main evaluation domain for this problem solver. Finally, the *digital forest cover map update* domain is the real world application where the problem solver will be used. There are currently only a few results that can be presented in this domain since the application itself is the subject of ongoing research. This chapter's aim is to show the contribution of each component of

the system. The rest of the chapter is organized as follows: section 4.1 discusses achieving non-linearity and the systems behaviour compared to another non-linear planning system; section 4.2 shows how the system scales up to a more complex domain by outlining the contributions of the system components (i.e.: use of analogy and use of generalization).

4.1 Achieving Non-Linearity

Chapter 3 introduced the use of case-based reasoning as a preferred problem solving method. Since the problem solver remembers the solution to each goal statement conjunct as soon as it is created, these solutions are made available to solve other conjuncts within the same goal statement. This approach transforms the goal regression algorithm into a non-linear problem solver³. Let us recall the *one-way-rocket* problem with the goal statement *at(obj1, loc2) and at(obj2, loc2)*. Figure 4.1.1 shows the reasoning trace to solve this problem while using analogy and starting with an empty case-base. Figure 4.1.2 shows the reasoning trace to solve this problem by using the approach taken by simulating the NoLimit system without analogy. Figure 4.1.3 shows the reasoning trace to solve the same problem while using analogy with a case-base containing one generalized case.

The first trace shows the goal regression behaviour of the problem solver with the additional attempt to solve goals via analogy before regression. For the first conjunct of the goal statement, *at(obj1, loc2)*, the system first tries analogy. Since the case-base is empty, analogy fails and the system must perform regression. It therefore must find an operator (*unload(obj1, loc2)*) that adds the desired fact to the world and attempt to satisfy the operator's pre-conditions (*obj1 is_a package and inside(obj1, rocket) and at(rocket, loc2)*). The first conjunct, *obj1 is_a package*, in the pre-condition holds. The second conjunct, *inside(obj1, rocket)*, does not. Since the case-base is still empty, analogy fails and the system performs regression by retrieving the *load_rocket(obj1, loc1)* operator. Since both of its pre-conditions are true, the operator's effects are applied and the operator is added to the current world. This continues until a solution for the first conjunct is found. This solution is then stored in the case-base and made available for immediate reuse. When the system attempts to solve the second conjunct in the goal statement, it retrieves the case, computes the substitution required to adapt the case, generalizes the case base and finally merges the solution for the second conjunct with the solution for the first conjunct and terminates successfully.

While attempting to solve the first conjunct of the goal statement, the second trace is identical to the first with the exception that there is no attempt at using analogy. Once the solution for the first conjunct is found, the system fails because, as operators are added to the current world, their effects are applied resulting in the rocket being at location *loc2* and there is no way to return it

³ As was stated in chapter 3, *linear planning* is a goal-directed search paradigm where: plans are totally ordered and the solution is found by solving the goals in a *linear* order (i.e. if two or more goals exist then one goal and all of its subgoals are achieved before other goals are considered.)

to location *loc1*. Upon failure, the system will try to apply any applicable operator in order to proceed. In this case, the only available, applicable operator is *load_rocket(obj1, loc2)*. The system quickly realizes that this simply undoes the work that it has done thus far and abandons

```
Trying analogy for: at(obj1,loc2)
Analogy failed: at(obj1,loc2)
To solve at(obj1,loc2) try unload_rocket(obj1,loc2)
  obj1 is_a package is true already
Trying analogy for: inside(obj1,rocket)
Analogy failed: inside(obj1,rocket)
To solve inside(obj1,rocket) try load_rocket(obj1,A)
  obj1 is_a package is true already
  at(obj1,loc1) is true already
  at(rocket,loc1) is true already
  Doing action(s) - del: [at(obj1,loc1)]
                    add: [inside(obj1,rocket)]
Trying analogy for: at(rocket,loc2)
Analogy failed: at(rocket,loc2)
To solve at(rocket,loc2) try unload_rocket(rocket,loc2)
  Trying analogy for: rocket is_a package
  Analogy failed: rocket is_a package
  Could not prove rocket is_a package
Failed to solve at(rocket,loc2)
To solve at(rocket,loc2) try move_rocket
  at(rocket,loc1) is true already
  Doing action(s) - del: [at(rocket,loc1)]
                    add: [at(rocket,loc2)]
Doing action(s) - del: [inside(obj1,rocket)]
                    add: [at(obj1,loc2)]
Trying analogy for: at(obj2,loc2)
Substitution: at(obj2,loc2) and at(obj1,loc2) <== [obj1/obj2]
Substituting: [obj1/obj2]
Generalizing: at(obj2,loc2) and at(obj1,loc2)
Generalization: goal:at(A,loc2)
Trying case for : at(obj2,loc2)
  foot print      : [at(obj2,loc1),at(rocket,loc1),obj2 is_a package]
  Operator: load_rocket(obj2,loc1) for goal: inside(obj2,rocket)
  Operator: move_rocket for goal: at(rocket,loc2)
  Operator: unload_rocket(obj2,loc2) for goal: at(obj2,loc2)
  Current operators: (load_rocket(obj1,loc1)) - (load_rocket(obj2,loc1))
  + Inserting load_rocket(obj2,loc1) before load_rocket(obj1,loc1)
  Current operators: (load_rocket(obj1,loc1)) - (move_rocket)
  + Inserting load_rocket(obj1,loc1) before move_rocket
  Current operators: (move_rocket) - (move_rocket)
  + Removing redundant: move_rocket
  Current operators: (move_rocket) - (unload_rocket(obj2,loc2))
  + Inserting move_rocket before unload_rocket(obj2,loc2)
  Current operators: (unload_rocket(obj1,loc2)) - (unload_rocket(obj2,loc2))
  + Inserting unload_rocket(obj2,loc2) before unload_rocket(obj1,loc2)
  + Appending unload_rocket(obj1,loc2)
Succeeded at: 0.255335 secs.
```

Figure 4.1.1 - Reasoning trace of the *one-way-rocket* problem while using analogy.

```

To solve at(obj1,loc2) try unload_rocket(obj1,loc2)
obj1 is_a package is true already
To solve inside(obj1,rocket) try load_rocket(obj1,A)
obj1 is_a package is true already
at(obj1,loc1) is true already
at(rocket,loc1) is true already
Doing action(s) - del: [at(obj1,loc1)]
                  add: [inside(obj1,rocket)]
To solve at(rocket,loc2) try unload_rocket(rocket,loc2)
Could not prove rocket is_a package
Failed to solve at(rocket,loc2)
To solve at(rocket,loc2) try move_rocket
at(rocket,loc1) is true already
Doing action(s) - del: [at(rocket,loc1)]
                  add: [at(rocket,loc2)]
Doing action(s) - del: [inside(obj1,rocket)]
                  add: [at(obj1,loc2)]
To solve at(obj2,loc2) try unload_rocket(obj2,loc2)
obj2 is_a package is true already
To solve inside(obj2,rocket) try load_rocket(obj2,A)
obj2 is_a package is true already
at(obj2,loc1) is true already
To solve at(rocket,loc1) try unload_rocket(rocket,loc1)
Could not prove rocket is_a package
Failed to solve at(rocket,loc1)
Could not prove at(rocket,loc1)
NoLimit - trying load_rocket(obj1,loc2)
NoLimit - retracting load_rocket(obj1,loc2)
- at(obj2,loc1)
- obj2 is_a package
Failed to solve inside(obj2,rocket)
Could not prove inside(obj2,rocket)
- obj2 is_a package
Failed to solve at(obj2,loc2)
Could not prove at(obj2,loc2)
Retracting - del: [inside(obj1,rocket)]
              add: [at(obj1,loc2)]
Retracting - del: [at(rocket,loc1)]
              add: [at(rocket,loc2)]
- at(rocket,loc1)
NoLimit - trying load_rocket(obj2,loc1)
Doing action(s) - del: [at(obj2,loc1)]
                  add: [inside(obj2,rocket)]
at(rocket,loc1) is true already
Doing action(s) - del: [at(rocket,loc1)]
                  add: [at(rocket,loc2)]
Doing action(s) - del: [inside(obj1,rocket)]
                  add: [at(obj1,loc2)]
To solve at(obj2,loc2) try unload_rocket(obj2,loc2)
obj2 is_a package is true already
inside(obj2,rocket) is true already
at(rocket,loc2) is true already
Doing action(s) - del: [inside(obj2,rocket)]
                  add: [at(obj2,loc2)]
Succeeded at: 0.285471 secs.

```

Figure 4.1.2 - Reasoning trace of the *one-way-rocket* problem without using analogy.

the attempt (these two steps are labeled *NoLimit* in the trace). Since there are no other alternatives, the system backtracks and undoes all of its work until it reaches the state where the rocket is at location *loc1* and the object *obj1* is inside the rocket. At this backtrack point, the system finds the applicable operator *load_rocket(obj2, loc1)* and applies it. The next step is to satisfy the precondition *at(rocket, loc2)* of the operator *unload_rocket(obj1, loc2)*. This can proceed normally with regression as it did before. When the first conjunct in the goal statement is satisfied the state of the world is: *at(obj1, loc2) and at(rocket, loc2) and inside(obj2, rocket)*. The system can now easily solve the second conjunct of the goal statement by using regression.

The third trace shows that to solve the first conjunct of the goal statement, the system retrieves a case from the case-base. Since the case has been generalized, there is no need to compute a substitution to adapt the case. The case is then merged into the current (empty) plan to produce the solution for the conjunct. The system then retrieves the same case to solve the second conjunct of the goal statement and proceeds to merge it with the current plan to produce the final solution.

```
Trying analogy for: at(obj1,loc2)
Trying case for : at(obj1,loc2)
  foot print      : [at(obj1,loc1),at(rocket,loc1),obj1 is_a package]
  Operator: load_rocket(obj1,loc1) for goal: inside(obj1,rocket)
  Operator: move_rocket for goal: at(rocket,loc2)
  Operator: unload_rocket(obj1,loc2) for goal: at(obj1,loc2)
    + Appending load_rocket(obj1,loc1)
    + Appending move_rocket
    + Appending unload_rocket(obj1,loc2)
Trying analogy for: at(obj2,loc2)
Trying case for : at(obj2,loc2)
  foot print      : [at(obj2,loc1),at(rocket,loc1),obj2 is_a package]
  Operator: load_rocket(obj2,loc1) for goal: inside(obj2,rocket)
  Operator: move_rocket for goal: at(rocket,loc2)
  Operator: unload_rocket(obj2,loc2) for goal: at(obj2,loc2)
  Current operators: (load_rocket(obj1,loc1)) - (load_rocket(obj2,loc1))
    + Inserting load_rocket(obj2,loc1) before load_rocket(obj1,loc1)
  Current operators: (load_rocket(obj1,loc1)) - (move_rocket)
    + Inserting load_rocket(obj1,loc1) before move_rocket
  Current operators: (move_rocket) - (move_rocket)
    + Removing redundant: move_rocket
  Current operators: (move_rocket) - (unload_rocket(obj2,loc2))
    + Inserting move_rocket before unload_rocket(obj2,loc2)
  Current operators: (unload_rocket(obj1,loc2)) - (unload_rocket(obj2,loc2))
    + Inserting unload_rocket(obj2,loc2) before unload_rocket(obj1,loc2)
    + Appending unload_rocket(obj1,loc2)
Succeeded at: 0.115437 secs.
```

Figure 4.1.3 - Reasoning trace of the *one-way-rocket* problem with analogy and a non-empty, generalized case-base.

The CPU time required to solve the problem with analogy is just over 0.25 seconds when starting with an empty case-base. The CPU time required to solve the same problem by simulating the *NoLimit* system without using analogy is just below 0.29 seconds. Finally, the CPU time

required to solve the same problem with a non-empty case-base containing one generalized case is just over 0.11 seconds.

Table 4.1.1 highlights the performance differences between the NoLimit simulation without analogy and the approach presented here with analogy beginning with an empty case-base and with analogy beginning with a non-empty case-base. Each of the three types of tests listed were applied to two, three and four conjunct goal statements. Since the Prolog engine selects terms from its knowledge base in an ordered fashion, it was simple to determine the problems that would influence the performance of the NoLimit system. By re-ordering the conjuncts in the goal statement, it was possible to create the best and worst case scenarios that produced the results shown in the table. For NoLimit, the first line shows the best case and the second line shows the worst case. For PALERMO, the system performed equally well regardless of the order of the goal statement conjuncts. There are two numbers entered in the table for each experiment: the number of search nodes and the CPU time required to solve the problem (*<search nodes> / <CPU time>*).

	2 subgoals	3 subgoals	4 subgoals
NoLimit (no analogy)	9 / 0.3 57 / 3.37	15 / 0.69 58 / 3.53	22 / 1.19 59 / 3.48
Analogy (empty case-base)	5 / 0.255	6 / 0.345	7 / 0.511
Analogy (non-empty case-base)	2 / 0.115	3 / 0.228	4 / 0.378

Table 4.1.1

NoLimit, like goal regression, is a goal-directed search based problem solving approach. However, rather than exclusively expanding a pending subgoal at each search node, NoLimit either selects a subgoal to expand or applies an operator applicable at the current search node. The implementation used for the results presented here has two modifications: the first is a preference for expanding subgoals before applying applicable operators, the second is that an applicable operator is only applied when another operator's pre-conditions (a subset of the pending goals) have failed to be satisfied. These two modifications have made for a more efficient implementation. In [Veloso 1992], the time required by NoLimit to solve a four subgoal problem in the *one-way-rocket* domain without using analogy is 117.5 seconds. In the worst case, the NoLimit simulation solved the four subgoal problem in 3.48 seconds while exploring 59 search nodes. Without the second enhancement listed above, the NoLimit simulation took approximately 50 to 60 seconds in the worst case for a four subgoal problem.

These results show that a search-based linear planner such as goal regression can be modified to use case-based reasoning to produce solutions that it would otherwise overlook. The search engine is still linear in nature in that it attempts to solve all the pre-conditions of any operator chosen to expand a search node, however, if it uses a case to solve a subgoal, the case is merged into the current state of the world. In the *one-way-rocket* domain, goal-regression with analogy significantly outperforms the NoLimit simulation both in time and in the number of search nodes.

4.2 Scaling up - The logistics transportation domain

In this section, the costs and the benefits of using an analogical reasoning system as a complement to search-based planning is examined to determine whether the system has gained by their integration. In order to evaluate the problem solver's behaviour, a problem generator for the logistics transportation domain has been implemented. The domain is composed of locations, airports and post-offices, spread out across different cities. Planes are used to travel between cities and trucks are used to travel between locations in the same city. Although there are a few other types of goals possible in this domain (e.g.: that airplane *plane051* be located at location *airport076*), the only type of goals that were generated were to deliver packages from one location to another. In fact, moving a truck or a plane from one location to another requires only one operator and therefore these plans would not satisfy the definition of a case. When there is more than one conjunct in the goal statement, the system will learn repair cases that will include satisfying goals such as loading a package into a plane or a truck.

The empirical results presented in this section show the scaling up properties of the problem solver as well as its ability to solve problems in a complex domain while using a relatively small case-base which is initially empty.

4.2.1 Generating problems

The problem generator must create a goal statement and an initial world from which the problem solver must reason to satisfy the goal. The generator generates a random number of cities (four to 10), a random number of post-offices per city (two to four), a random number of trucks per city (one to four), one airport per city, a random number of airplanes (two to 10) and a random number of packages (10 to 15). The packages are distributed randomly across all possible locations (airports and post-offices), the airplanes are distributed randomly between the airports and the trucks are distributed randomly between the locations within their respective cities. Finally, a goal is generated that specifies a destination for one or more of the packages. For all of the experiments we started with an empty case-base.

4.2.2 Rate of accumulation of one conjunct goal statement problems

The object of the first experiment was to determine if the problem solver would learn the expected number of cases for goal statements having only one conjunct. Given the implementation of the problem solver and domain operators, there is a total of 32 different possible cases (the cases for single conjunct goal statements have been enumerated and cover all possible problems in this domain; moreover, it is possible to reduce this number by relaxing certain constraints in the

generalization procedure). For this experiment, we generated 1000 problems, restricting them to one truck per city, one plane and one package, and submitted them to the problem solver. Selecting one truck per city and one plane ensures that all possible situations are more likely to occur. When we generate 1000 problems, there is still between 0.040 and 0.096 probability that certain situations may not occur given these parameters.

```

at(obj(obj1206),airport(ap4676))
at(obj(obj1475),airport(ap3968))
at(obj(obj1587),airport(ap2084))
at(obj(obj1981),post_office(po8970))
at(obj(obj2440),post_office(po579))
at(obj(obj4437),airport(ap8837))
at(obj(obj460),post_office(po8970))
at(obj(obj5719),post_office(po4887))
at(obj(obj8179),post_office(po7822))
at(obj(obj8499),airport(ap9254))
at(obj(obj8920),post_office(po8970))
at(obj(obj8981),airport(ap2084))
at(obj(obj9203),post_office(po3189))
at(obj(obj9589),post_office(po7822))
at(obj(obj9967),airport(ap2084))
at(plane(p1244),airport(ap9254))
at(plane(p2377),airport(ap9254))
at(plane(p4893),airport(ap9254))
at(truck(truck1279),airport(ap9254))
at(truck(truck2067),airport(ap4676))
at(truck(truck2068),post_office(po7822))
at(truck(truck2729),airport(ap4676))
at(truck(truck3328),airport(ap3968))
at(truck(truck3395),post_office(po1370))
at(truck(truck4306),post_office(po334))
at(truck(truck472),airport(ap3968))
at(truck(truck5228),airport(ap9254))
at(truck(truck5823),post_office(po1370))
at(truck(truck6238),post_office(po7822))
at(truck(truck6454),post_office(po334))
at(truck(truck6745),post_office(po7822))
at(truck(truck7566),airport(ap3968))
at(truck(truck7886),airport(ap4676))
at(truck(truck8101),post_office(po334))
at(truck(truck8348),post_office(po7822))
at(truck(truck8674),post_office(po4887))
at(truck(truck8856),airport(ap8837))
at(truck(truck959),airport(ap3968))
city(airport(ap2084),city(city7411))
city(airport(ap3968),city(city688))
city(airport(ap4676),city(city7246))
city(airport(ap8837),city(city5362))
city(airport(ap9254),city(city4064))
city(obj(obj1206),city(city7246))
city(obj(obj1475),city(city688))
city(obj(obj1587),city(city7411))
city(obj(obj1981),city(city4064))
city(obj(obj2440),city(city5362))
city(obj(obj4437),city(city5362))
city(obj(obj460),city(city4064))
city(obj(obj5719),city(city7246))
city(obj(obj8179),city(city7411))
city(obj(obj8499),city(city4064))
city(obj(obj8920),city(city4064))
city(obj(obj8981),city(city688))
city(obj(obj9203),city(city688))
city(obj(obj9589),city(city7411))
city(obj(obj9967),city(city7411))
city(post_office(po1370),city(city4064))
city(post_office(po1645),city(city7411))
city(post_office(po1846),city(city4064))
city(post_office(po2914),city(city688))
city(post_office(po3189),city(city688))
city(post_office(po334),city(city5362))
city(post_office(po4831),city(city7411))
city(post_office(po4887),city(city7246))
city(post_office(po5007),city(city688))
city(post_office(po579),city(city5362))
city(post_office(po6183),city(city7246))
city(post_office(po6806),city(city5362))
city(post_office(po7822),city(city7411))
city(post_office(po8107),city(city7246))
city(post_office(po8970),city(city4064))
city(truck(truck1279),city(city4064))
city(truck(truck2067),city(city7246))
city(truck(truck2068),city(city7411))
city(truck(truck2729),city(city7246))
city(truck(truck3328),city(city688))
city(truck(truck3395),city(city4064))
city(truck(truck4306),city(city5362))
city(truck(truck472),city(city688))
city(truck(truck5228),city(city4064))
city(truck(truck5823),city(city4064))
city(truck(truck6238),city(city7411))
city(truck(truck6454),city(city5362))
city(truck(truck6745),city(city7411))
city(truck(truck7566),city(city688))
city(truck(truck7886),city(city7246))
city(truck(truck8101),city(city5362))
city(truck(truck8348),city(city7411))
city(truck(truck8674),city(city7246))
city(truck(truck8856),city(city5362))
city(truck(truck959),city(city688))
airport(ap2084)is_a airport
airport(ap2084)is_a location
airport(ap3968)is_a airport
airport(ap3968)is_a location
airport(ap4676)is_a airport
airport(ap4676)is_a location
airport(ap8837)is_a airport
airport(ap8837)is_a location
airport(ap9254)is_a airport
airport(ap9254)is_a location
post_office(po1370)is_a location
post_office(po1645)is_a location
post_office(po1846)is_a location
post_office(po2914)is_a location
post_office(po3189)is_a location
post_office(po334)is_a location
post_office(po4831)is_a location
post_office(po4887)is_a location
post_office(po5007)is_a location
post_office(po579)is_a location
post_office(po6183)is_a location
post_office(po6806)is_a location
post_office(po7822)is_a location
post_office(po8107)is_a location
post_office(po8970)is_a location

```

Figure 4.2.1 - Automatically generated initial world (118 literals).

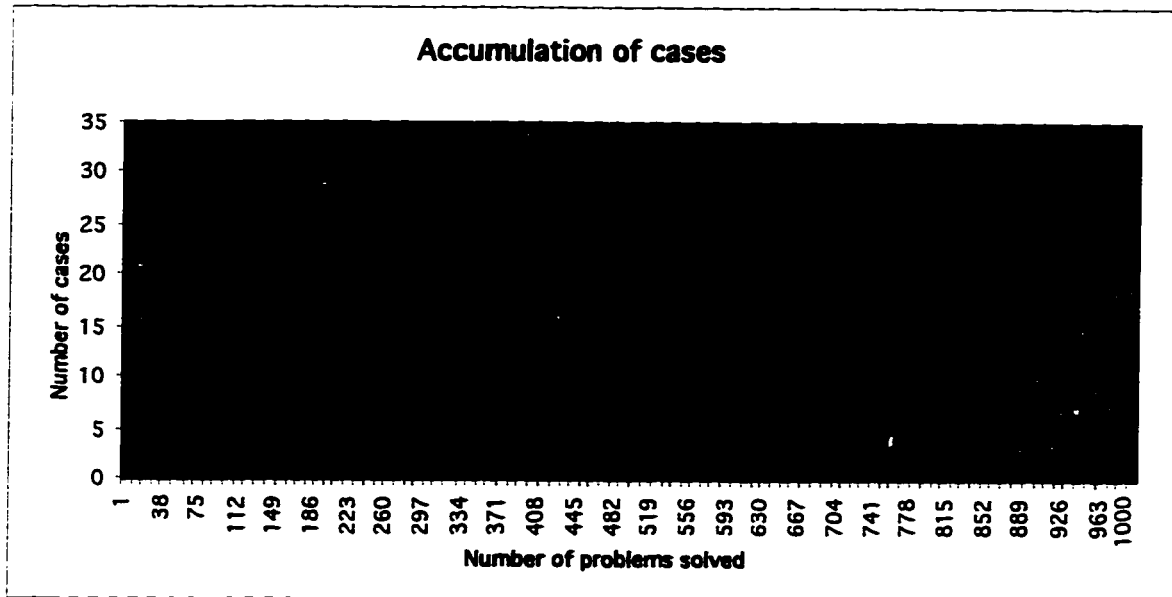


Figure 4.2.2 - Accumulation of cases for one conjunct goal statement problems

The results are shown in Figure 4.2.2. Since problem solver starts with an empty case-base, it is expected that the number of cases will rise rapidly. This is very evident in the figure where the *learning curve* is very steep for the first 100 to 150 problems. In fact, this behaviour has been very consistent during other similar experiments. After approximately 200 problems, the curve levels out to between 30 and 33 cases. In general, the small fluctuations in the curve are due to a delay in generalization, which is at times effective only after more problems are submitted to the system.

4.2.3 Analogy versus no Analogy

The object of the second experiment was to compare the performance of system, using all of its analogical and generalization features, to a goal-regression-based planner while using the identical problem generation parameters as for the first experiment. Since the problem solver's search mechanism is goal-regression, we simply disabled the analogical component and generated a second set of problems. In the graph in Figure 4.2.3, the problems were grouped into ten 100-problem sets. On the X axis, the numbers represent each set as they were submitted to the system. The solid line Y axis represents the average time in seconds that the system required to solve each problem while learning and re-using cases (with analogy), and subsequently generalizing them. The dashed line represents the average time required to solve each problem while using goal-regression only (without analogy).

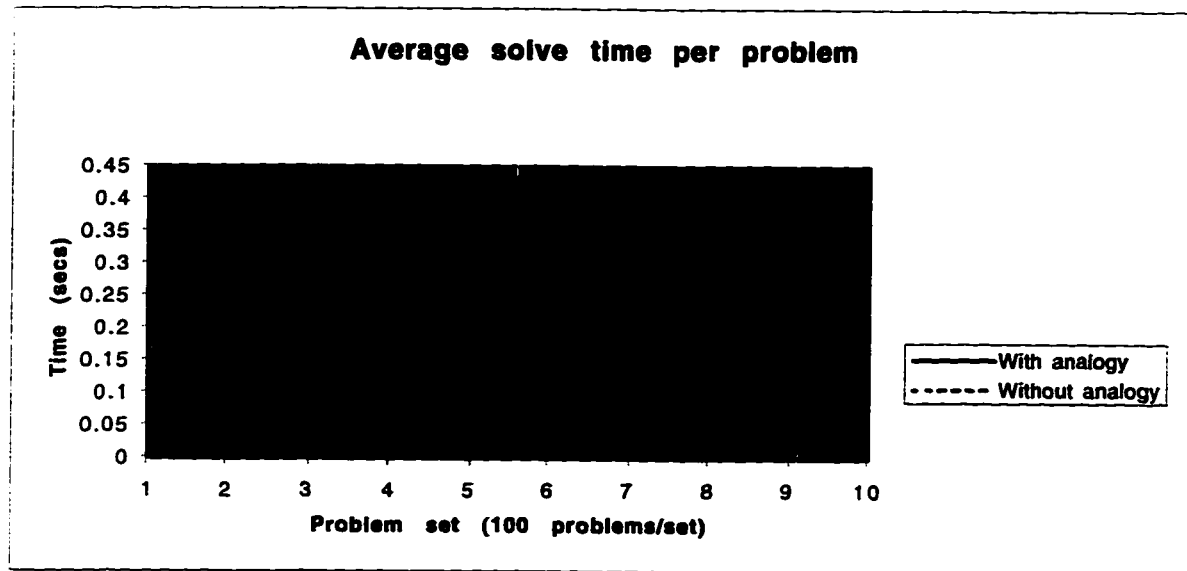


Figure 4.2.3 - Average solve time per problem for 1000 problems

When using analogy, the system begins by spending much of its time creating, retrieving, adapting and generalizing cases. This accounts for the extra amount of time it requires over goal-regression when its case base is empty. If we refer to Figure 4.2.2, once the system has reached the top of the learning curve, it begins to improve its performance and eventually outperforms goal-regression (Figure 4.2.3) when solving problems with single conjunct goal statements. This result indicates that the system does not suffer from the utility problem. Had we begun with good representational cases, the average solve time with analogy could have outperformed regression from the outset.

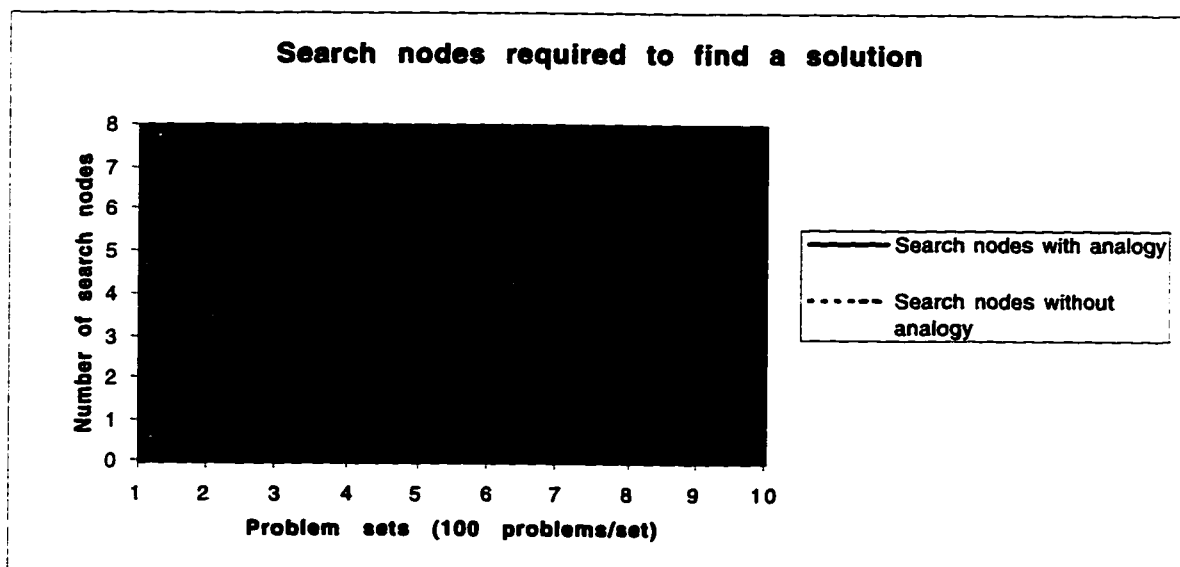


Figure 4.2.4 - Average number of search nodes per problem

The graph in Figure 4.2.4 shows the average number of search nodes the system explored for each problem solved. This graph is arranged in the same way as the graph in Figure 4.2.3, i.e. the problems are grouped in sets of 100 and the average for each set. As stated above, when using analogy, the system begins by spending much of its time creating cases. This justifies the use of extra search nodes when the case-base is empty. Once it has learned a sufficient number of cases, the number of search nodes required should level out at one because all possible problems in the domain should be handled by a case. When the system does not use analogy, it requires an average of 7.5 search nodes per problem.

4.2.4 Generalization versus no Generalization

The purpose of the next experiment was to compare the performance of the analogical component of the system with and without generalization while using the identical problem generation parameters as for the previous experiments (i.e.: one conjunct goal statements, one truck per city, etc.). Figure 4.2.5 shows the growth of the case base. The dotted line shows the system's accumulation of cases while using generalization (taken from Figure 4.2.2). The solid line shows the effect of turning generalization off. Since the system does not use generalization, it has no opportunity of refining the case base and therefore accumulates more cases whenever it fails to adapt the existing ones. Although failure occurs less frequently as the number of problems solved increases, the case base continues to grow and would eventually cause the utility problem.

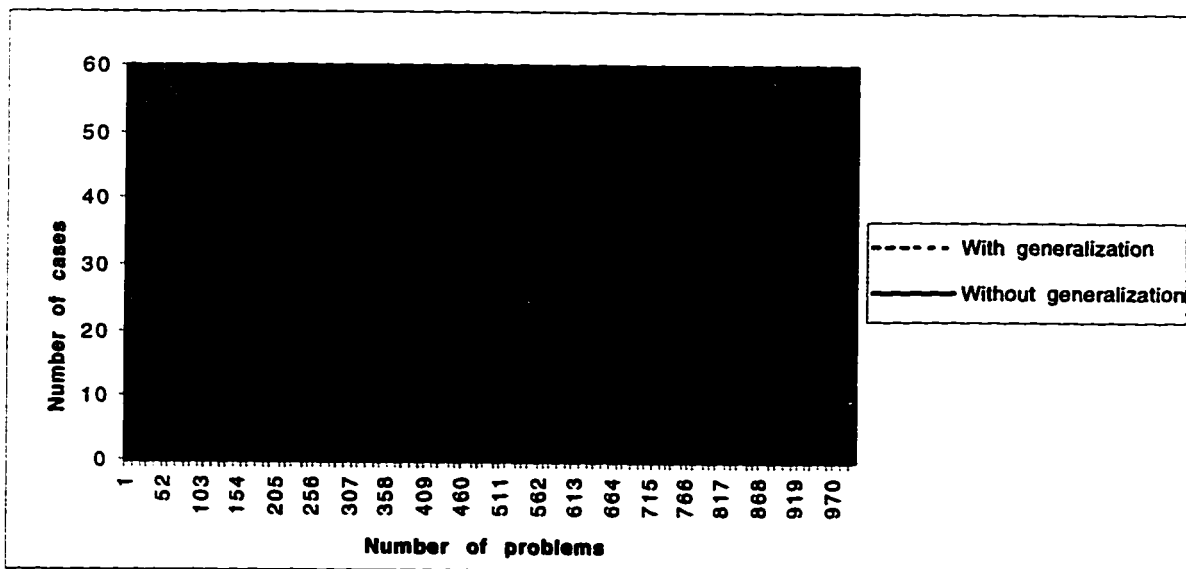


Figure 4.2.5 - Accumulation of cases while using generalization versus accumulation of cases without using generalization

As well as having a significant impact on the size of the case base, the generalization procedure also affects the effort that the system must expend in attempting to adapt cases to new problems. If generalization does not occur, all cases in the case base contain ground subterms that

must be replaced with other terms relevant to the new problem. The effort required to adapt and generalize a case is a function of the number of recursive calls that must be made to the generalization procedure whenever subterms of the case and the new problem do not agree. Figure 4.2.6 shows the average effort required by the system to adapt cases. The system performs much of the adaptation and generalization within the first 100 problems. Clearly, the effort required to adapt cases is considerably less when generalization is used. In fact, after having solved 300 problems, the adaptation effort is negligible, whereas if generalization is not used, the effort is consistently expensive.

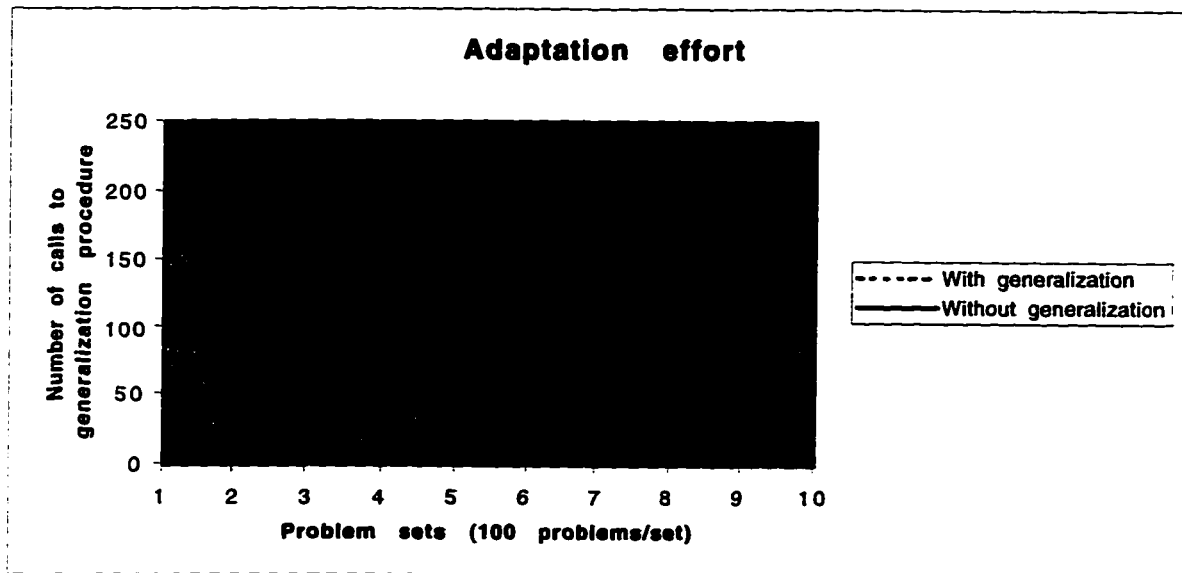


Figure 4.2.6 - Adaptation effort

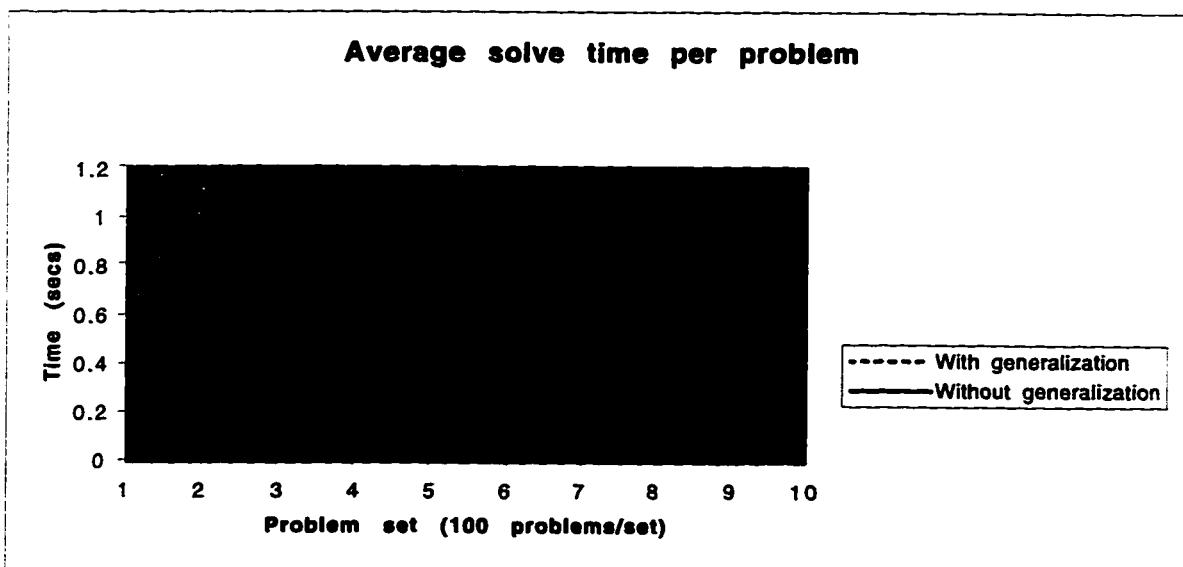


Figure 4.2.7 - Average solve time per problem.

As a result, when using generalization, the problem solver can reduce the time it requires to solve problems by more than 75 percent as is shown in Figure 4.2.7.

4.2.5 Learning from multiple conjunct goal statements

The experiments above were done mainly to determine whether the problem solver would behave in a predictable manner. The objective of restricting the number of conjuncts in the goal statement and the number and nature of literals describing the initial was to allow the calculation of probabilities to maximize the possibility of all problems to occur. The next set of experiments were conducted to measure the performance of the system when faced with multiple conjunct goal statements (i.e.: more complex problems) and a wider pool of resources to draw upon (i.e.: more available trucks and airplanes).

Ten problem sets of 1000 problems were generated for this experiment. Each set was further divided into ten subsets of 100 problems. The first subset was composed of 100 one conjunct goal statements. The second subset was composed of 100 two conjunct goal statements. The third was composed of three conjunct goal statement and so on to the tenth subset composed of 100 problems with ten conjunct goal statements. The total number of conjuncts solved is 5500 for each problem set. The results produced from one problem set are presented in this section. The results produced from the other nine problem sets can be found in appendix C. Other than Figure 4.2.8, the X axis on all graphs are divided into 100 problem subsets. Label 1 is for the 100 one conjunct goal statement, label 2 is for the 100 two conjunct goal statements, etc.

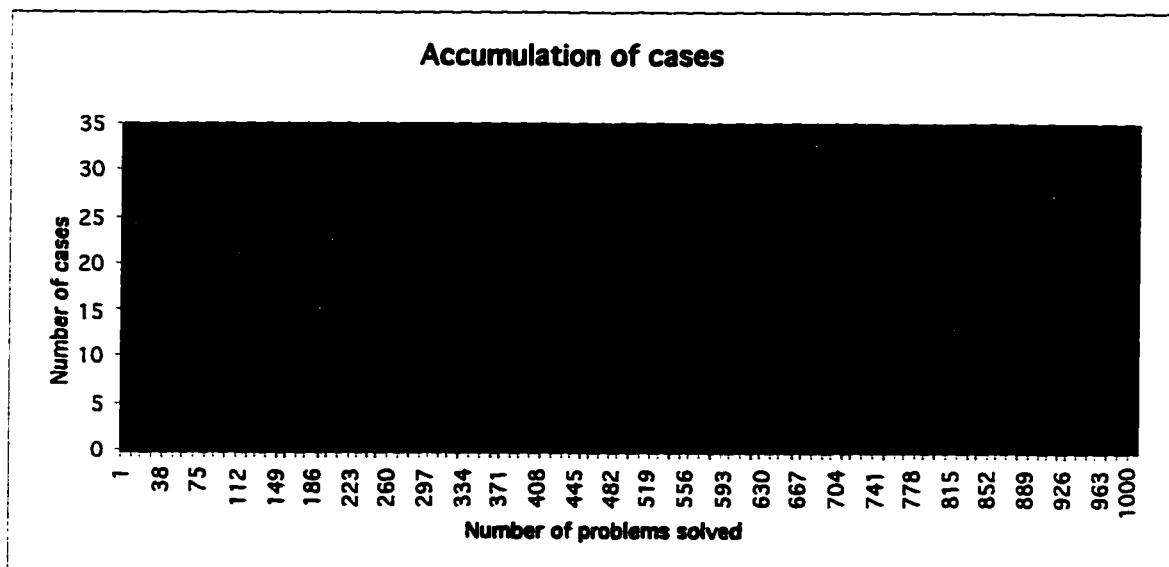


Figure 4.2.8 - Accumulation of cases for multiple conjunct goal statement problems.

The first graph, Figure 4.2.8, shows a similar learning curve to those in Figures 4.2.2 and 4.2.5. Most of the learning occurs early. After approximately 150 problems, the number of cases in the case base has reached 30 and thereafter increases only slightly. It is expected that the system should learn the same set of cases as in previous experiments. It should also learn some repair cases which accounts for the curve leveling at 35 cases.

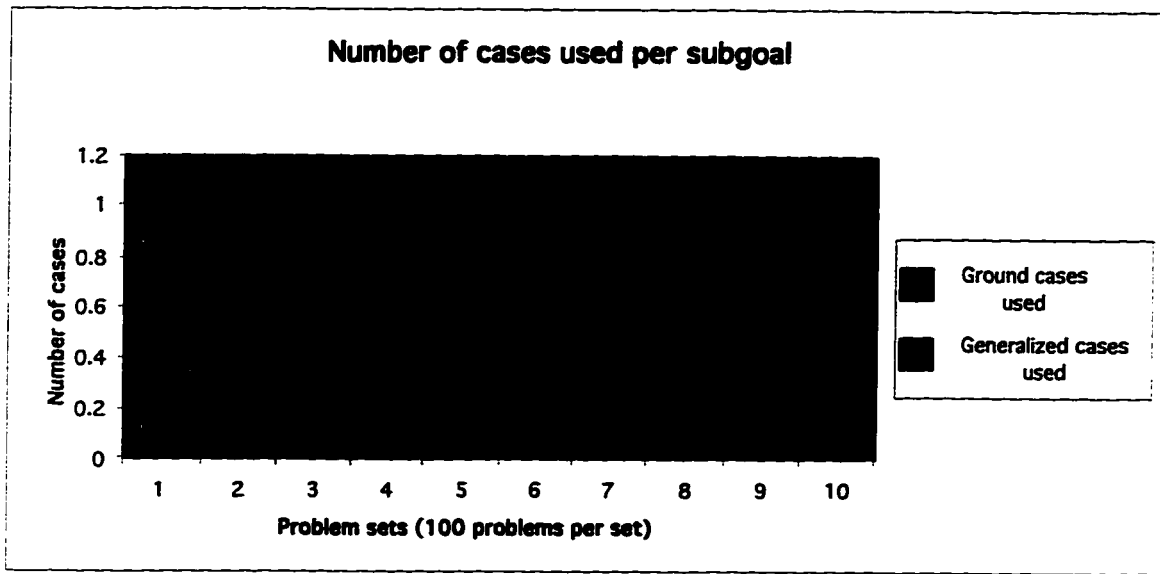


Figure 4.2.9 - Use of cases to solve goal statement conjuncts.

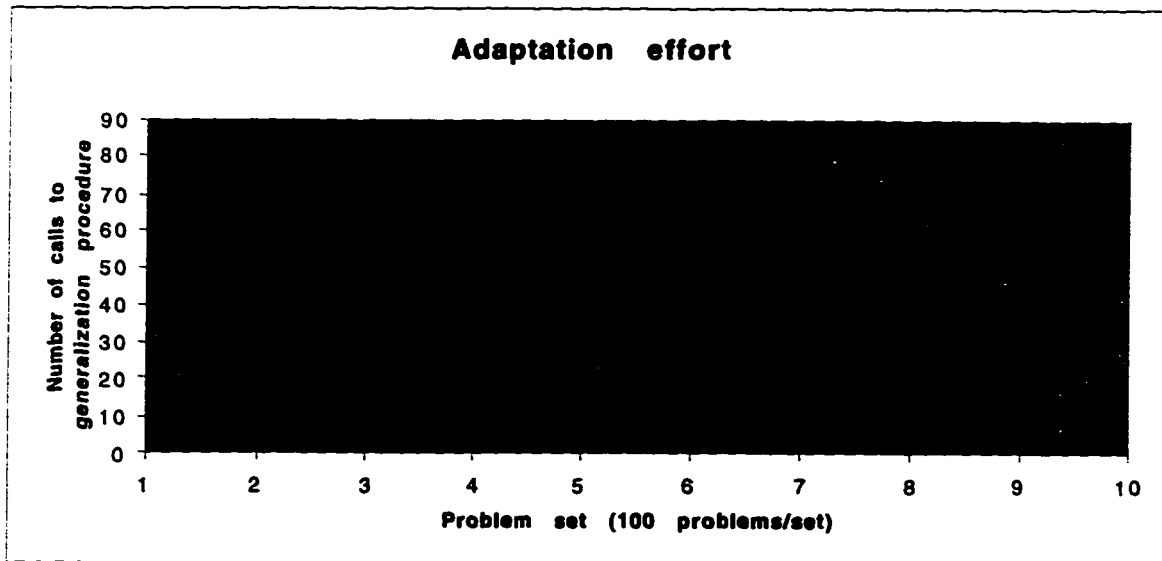


Figure 4.2.10 - Adaptation effort.

The next graph, Figure 4.2.9, shows the average number of cases used to solve each conjunct. It also shows the proportion of generalized cases used versus non-generalized cases. For instance, the first subset shows that for the 100 problems solved, 73% were solved using a

case (therefore 27 were solved from scratch). Of the 73%, 70% were solved using a generalized case (the lighter shade of gray) and 30% were solved using a non-generalized case (the darker shade of gray). This indicates that most of the generalization occurs very early during problem solving. Once the system starts solving the second and third subsets, very little generalization occurs. As the experiment continues, the average number of cases used to solve each conjunct rises above one. This is the result of using cases to carry out repairs during merging of solutions.

The adaptation effort described in section 4.2.3 describes the impact of generalization on the system's performances in terms of number calls to the adaptation procedure. The results for multiple conjunct goal statements are no less dramatic than they are for single conjunct goal statements as is shown in Figure 4.2.10. The system expends much effort early during problem solving to generalize solutions. As it gains experience, its case-base is generalized and it eventually requires virtually no adaptation effort.

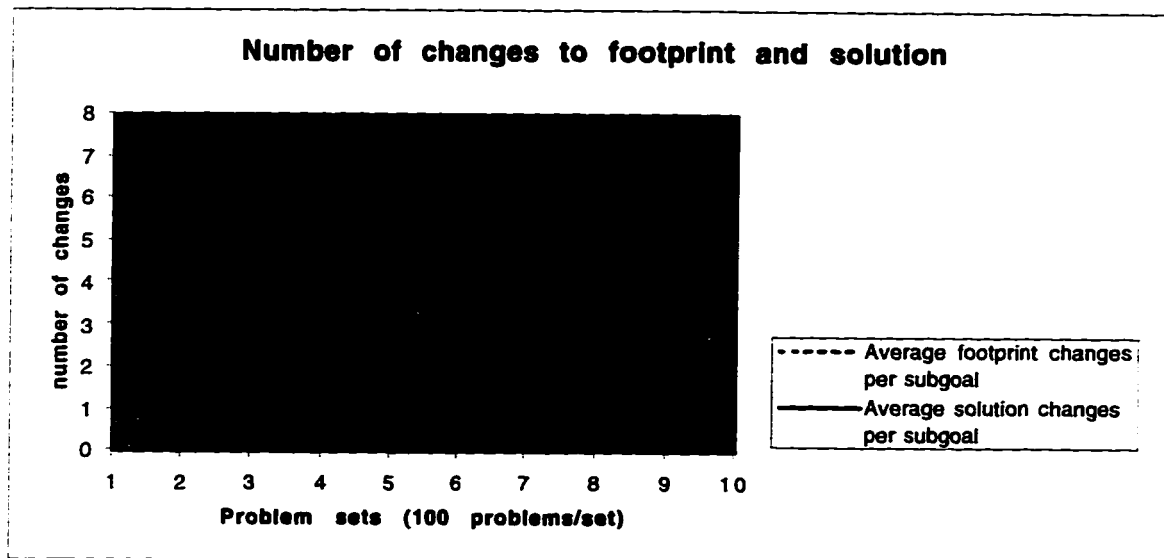


Figure 4.2.11 - Adaptation effort.

As a further measure of the adaptation effort, the graph in Figure 4.2.11 shows the average number of changes applied to case footprints and solutions when adapting them to new problems. After generalization, the number of changes required is very low since the generalized case contains variables that can be bound to world facts rather than ground literals that must be substituted for world facts.

Figure 4.2.12 shows the average time required to solve each conjunct in the goal statements. For the first 100 problem subset, the time required is greater than other subsets because this is when generalization occurs. As the subsets grow in the number of conjuncts in the goal statements, additional time is required to merge the solutions and carry out repairs.

Figure 4.2.13 clearly shows that, with an increasing number of conjuncts in goal statements, the system must spend considerable effort in merging and repairing cases.

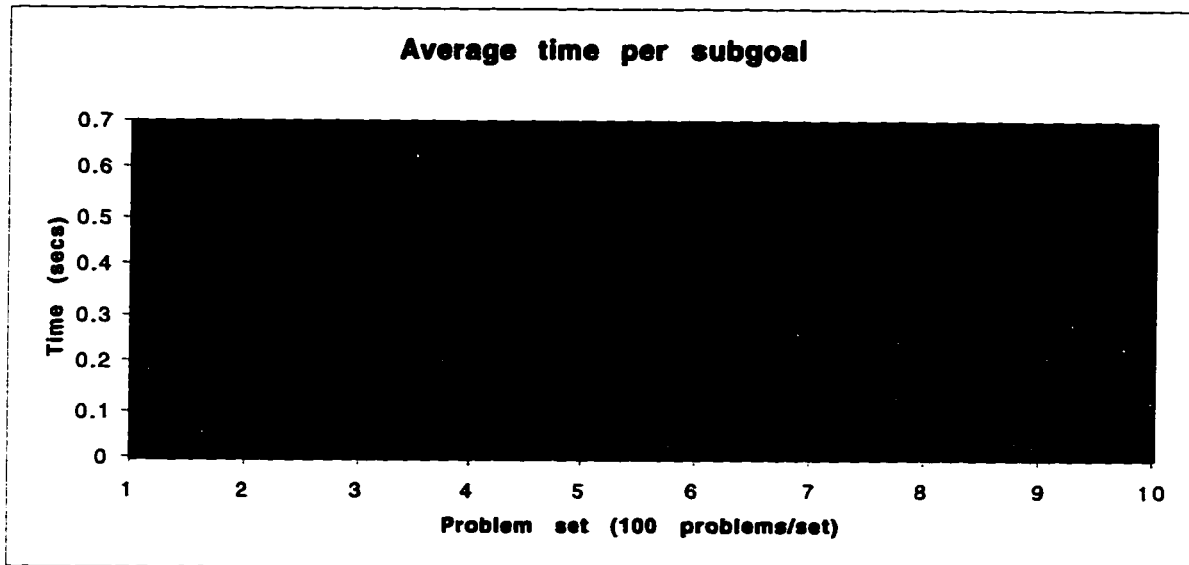


Figure 4.2.12 - Average time to solve one conjunct.

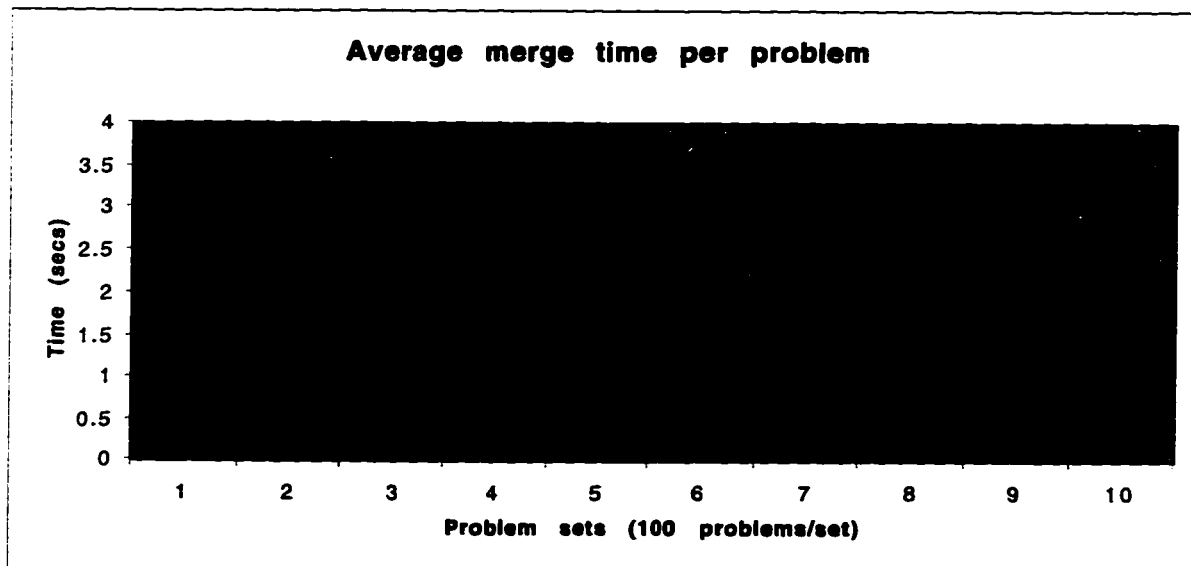


Figure 4.2.13 - Average merge time per problem.

The final graph for this problem set, Figure 4.2.14, shows the CPU time spent by the system retrieving cases from the case-base. Since the generalization procedure and retrieval procedure are so tightly coupled, the adaptation effort measured in time is included in the retrieval time. The system clearly requires more time for retrieval when the case-base has not yet been generalized. Once generalization has occurred, retrieval time for each conjunct is relatively constant.

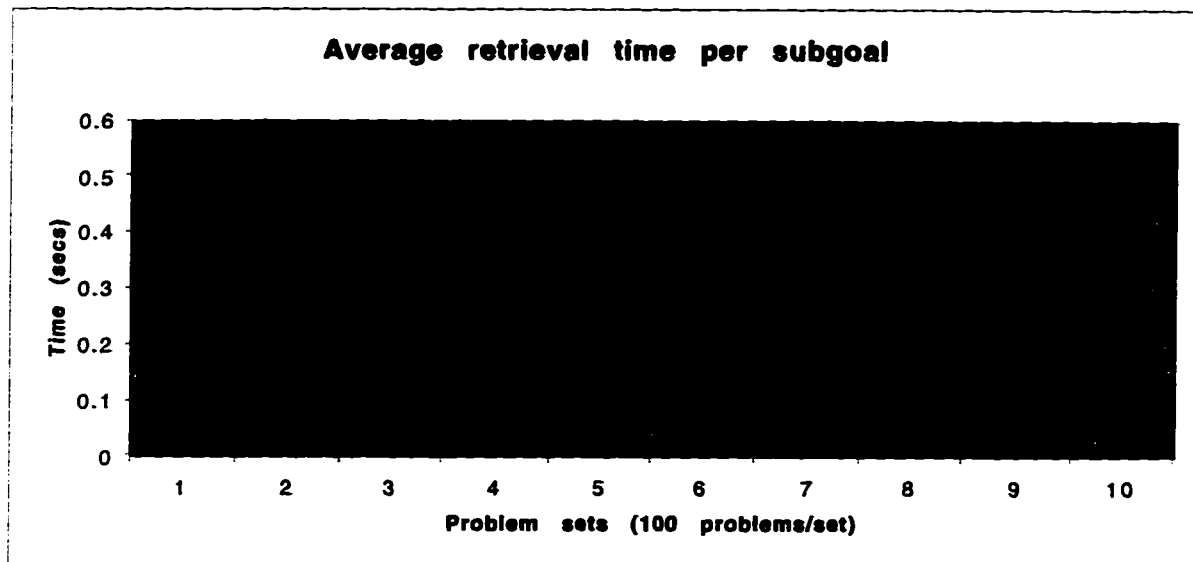


Figure 4.2.14 - Average retrieval time per goal statement conjunct.

Chapter 5

SEIDAM: The Application

One of the main challenges that AI in general and problem solving in particular face is scaling up to real world problems. The empirical results presented in chapter 4 dealt with toy worlds (the *one-way-rocket* domain and the *extended-STRIPS* domain) and a domain (the *logistics transportation* domain) that showed the scalability of the system to more complex (near) real world domains. The issues addressed in that chapter related mainly to performance of the reasoning system when presented with thousands of problems with multiple conjunct goal statements. Although the system behaved well, it did not face the complexities of sophisticated domains.

One such domain is the processing and management of extremely large data sets built from geographical information systems (GIS) data, remote sensing data and databases. As was discussed in chapter 2, a GIS file is a spatial database in which maps can be stored. A map covers a geographical surface and contains information about that surface. For instance, a map that represents elevation has line work that delimits areas on the map where each area has a different elevation. In the GIS, each area is called a polygon. Each polygon on a map has a database pointer to a database record containing information that should be consistent across the polygon (e.g. main tree species in the polygon is Douglas Fir). Processing of GIS information is typically based on the polygon areas. On the other hand, remote sensing data is pixel based, i.e. the image is not connected to a database and each pixel in the image grid represents a feature value at that particular geographic point. For instance, on an image grid representing elevation, the value of each pixel is the elevation at the pixel's geographic coordinates. A remote sensing image is divided into several spectral bands and the value for each pixel is a combination of these spectral bands representing the spectral signature of the corresponding geographic coordinate. For instance, a LANDSAT TM (Thematic Mapper) image is divided into seven spectral bands ranging from infra-red to violet and

each pixel covers a 25 metre by 25 metre surface. By combining spectral bands three, four and five it is possible to easily highlight different types of forest (old growth forest, new growth forest and areas of denudation). Hyper-spectral sensors are divided into many more spectral bands. For example, AVIRIS (Airborne Visible to Infra-Red Imaging Spectrometre) is divided into 224 spectral bands covering the same spectrum as TM.

When used together, all of these data sets can be used for many specialized applications. For instance, land use decision makers must consider many environmental issues when performing their duties. In order to accomplish their tasks, it is crucial that accurate up to date information be at their disposal. Forest cover change is one type of information that is essential. Forest cover inventories are stored in digital form in GISs. Remote sensing data (satellite imagery, air photos, radar imagery, etc.) can be processed and compared to GIS forest cover data in order to detect changes. The processing that must be applied to integrate the GIS and remote sensing data includes many complex steps. Given the appropriate description of these steps, a problem solving system can formulate plans to create data products that would satisfy land use decision makers' needs.

This chapter introduces SEIDAM (System of Experts for Intelligent Data Management) [Charlebois, Goodenough et al. 1994; Goodenough, Bhogal et al. 1994; Matwin, Charlebois et al. 1994] for forest and environmental monitoring, a system currently being developed at the Pacific Forestry Centre in Victoria British Columbia. The discussion will show how problem solving techniques as introduced herein can be used to formulate plans for SEIDAM. These plans address problems such as data integration for forest inventory update and calibration of remote sensing data. The use of software agents and how they interact with the problem solver is also described in this chapter. Software agents, (also known as *apprentices* [Mitchell, Mahadevan et al. 1985; Mitchell, Caruana et al. 1994] and *softbots* [Kautz, Selman et al. 1994]) are systems that unobtrusively observe the manner in which they are used, adapt to the tasks for which they are used, as well as learn from the circumstances of their use.

5.1 The problem domain

In recent years, vast data repositories useful for understanding the current state of the biosphere and the trends in vegetation over large areas became available. Use and analysis of such data can lead to better, sustainable utilization of resources. Timely understanding of changes in the information on the forest cover in the areas where utilization of forests is an economic reality can also promote the efforts leading to maintenance of bio-diversity and sustainable development. The problem with the access to, use, and analysis of these data is their diversity and complexity. This diversity is reflected in a variety of formats, media, and granularity of data. The problem is compounded by the complexity and heterogeneity of the computing environments in which the data reside, as well as the multitude of software tools which are needed to extract the required information from data. In forestry, a typical information system integrates forest cover descriptions, topographic maps, remote sensing, and application knowledge. Data grows incrementally, as update data is acquired

automatically from space-borne and air-borne remote sensing devices. The volume of the incoming data is huge; it is anticipated that the rate of data acquisition will reach one terabyte per day by the year 2000. Data management systems for resources will need to make intelligent selections of the incoming data in order to respond to users' goals, reduce complexity, and be more adaptable.

Typically, in order to analyze these data, they require sophisticated processing to correct for distortions created by the atmosphere, by surface topography, or by platform movement. Calibration must be carried out to adjust for sensor behaviour and signal degradation. And, they must undergo several transformations between different formats to allow the remote sensing data to be integrated with the GIS data. These processing and transformation problems lend themselves well to planning and case-based reasoning since the steps involved are fixed and must follow certain sequences.

5.2 Problem solving in SEIDAM

The main objective of the system is to provide data and knowledge representation to allow sophisticated processing of the data in order to facilitate land use decisions. There are two types of data that can be processed: remote sensing image data and GIS data. Given the large volumes of these data, SEIDAM must rely on remote sensing image and GIS metadata databases. These metadata databases contain information regarding the state of the image and GIS data. For instance, a record of the processing applied to remote sensing images is stored in the metadata database. These data will indicate such information as the type of processing undergone by the imagery. This information is critical since an image that has not been submitted to topographic relief correction cannot be used for forest cover change detection.

SEIDAM components are shown in Figure 5.2.1. In this diagram, the metadata databases, and the raw and processed image and GIS data repositories are shown on the outside of the gray box. Inside the box, there are seven individual components: the SEIDAM expert system, the reasoning system, the smart access software agents, the image processing software agents, the GIS software agents, the SEIDAM knowledge base and the case base.

The SEIDAM expert system is responsible for the overall control of the components. It oversees the activation of the reasoning system as well as all of the software agents.

The reasoning system draws on two sources to create plans: a case base of previously solved problems and the SEIDAM knowledge base. The case base begins empty, and grows as the system solves new problems. The SEIDAM knowledge base contains the planning operators and a copy of the metadata data relevant to the current user goal statement (the initial world).

The image processing agents and the GIS agents will place planning operators in the SEIDAM knowledge-base. These planning operators are subsequently used by the PALERMO to solve problems submitted to it by the SEIDAM expert system. The planning operators placed in the

knowledge-base by the image processing and GIS agents describe the inputs that each agent requires (pre-conditions) and the output they produce (effects: add and delete lists).

The image processing agents and GIS agents perform tasks such as creating a digital elevation model (DEM) from point elevation data. Point elevation data is a grid that has irregularly distributed elevation information, i.e. not every cell in the grid contains data. The GIS agents can take point elevation data and process it to produce a DEM. A DEM can also be stored in a grid. The difference with point elevation is that a DEM grid is full, i.e. every cell in the grid must have data. When translated from the GIS domain to the image domain, a DEM can be used to carry out topographic relief correction to a remotely sensed image.

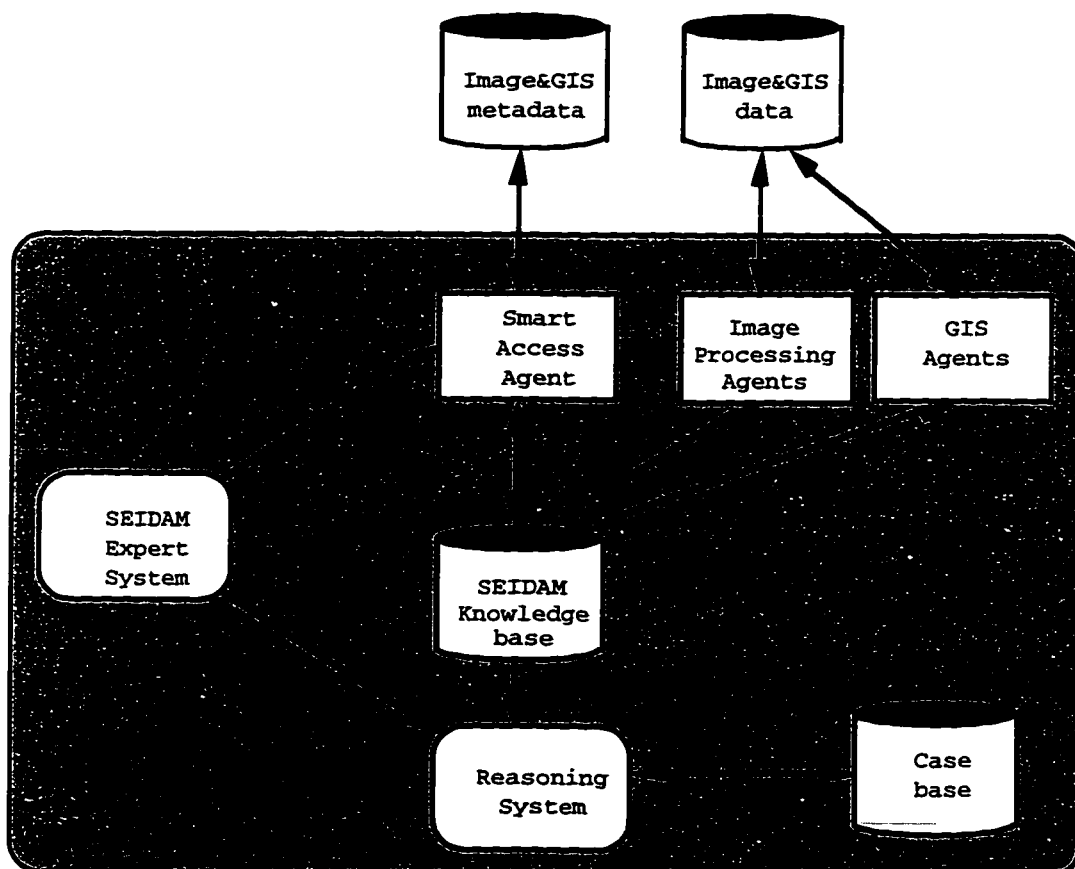


Figure 5.2.1 - SEIDAM architecture.

The Smart Access software agents utilize the user's goal statement to create SQL queries into the image and GIS metadata databases. The objective of these queries is to extract all of the information contained in the databases that is relevant to the user's goal. This information is placed in the SEIDAM knowledge base and is used by the reasoning system as its initial world.

Table 5.2.1 describes how the SEIDAM environment maps into the requirements of a planning system. Let us recall the definition of planning:

Given:

- an initial world described by a set of facts (object-attribute-value triplets),
- a set of planning operators that have a pre-condition list, a delete list and an add list that describe: what facts must hold for the operator to be applicable, what facts will be deleted and what facts will be added,
- a goal statement containing the facts that must hold to satisfy the user's goal.

Find a partially ordered set of operators that, if successfully applied, will ensure that the goal statement is satisfied.

Planning	SEIDAM	Example
goal statement	product description	updated forest cover map
planning operators	SEIDAM agents	trim2dem: extracts a digital elevation model from a terrain resource information map <pre> trim2dem :: if frame_get(current_map_sheet,name,A) and frame_get(working_directory,path,B) then delete [map_sheet(A,dem_file,C)] add [trim_to_dem, map_sheet(A,dem_file,dem)]</pre>
initial world	image and GIS metadata	digital elevation model filename /dwarfs/robot/data/gis/092f013/dem

Table 5.2.1 - Problem solving in SEIDAM

In the context of SEIDAM, the initial world is described by the information contained in the metadata databases (e.g.: a digital elevation model exists in the file /dwarfs/robot/data/gis/092f013/dem). Each operator in the set of planning operators describes how a software agent behaves (e.g.: the trim2dem software agent requires a terrain resource information map, TRIM, and produces a digital elevation model). The user's goal statement describes a

product that would allow her to make a land use decision (e.g.: requires an updated forest cover map to determine whether or not to allow logging in a certain area; to create this product the goal would be that a forest cover file exists).

In a typical scenario for SEIDAM, a user selects a product used to make land use decisions (e.g. emit a logging license for a given site) via a graphical user interface. A product can be a digital forest cover map updated with TM imagery in order to determine change in the total clear-cut area. The creation of this product becomes the goal that SEIDAM will submit to the reasoning system. SEIDAM's main expert system will then activate the smart access software agent that retrieves the relevant metadata from the remote sensing image and GIS metadata databases and stores the information in SEIDAM's knowledge-base. SEIDAM then submits the goal statement describing the product to the reasoning system. The reasoning system then creates a plan to satisfy the user's goal. If the reasoning system is successful, the plan is executed. The execution of the plan entails activating each agent in the order prescribed by the ordering constraints included in the plan. As the agents execute, they access, process and create new image and GIS data. After the successful activation of all of the agents, a product satisfying the user's goal has been created, and there is new information contained in the knowledge-base that must be stored in the metadata databases. SEIDAM will therefore activate the smart access agent to update the metadata databases.

5.3 SEIDAM: an example

One of the more complex problems SEIDAM must solve is the update of a digital forest cover map by using remote sensing images to detect forest depletion due to logging or other phenomena. This task can be carried out by overlaying a GIS digital forest cover map on a segmented LANDSAT-TM image and allowing a user to help identify new areas of denudation, thus creating updated polygons that can be added to the GIS. A geocoded TM image must be corrected for topographic relief before it can be used for this purpose. Digital elevation models (DEMs) are used as the basis to correct the image for topographic relief. The example below describes how a TM image is prepared by using a DEM created from 1:20000 TRIM irregularly spaced elevation data. TRIM (Terrain Resource Information Map) data is the standard used by the province of British Columbia Ministry of Environment, Lands and Parks.

The process is initiated by the user selecting from a list of functions the forest inventory update function in the SEIDAM forestry products window. Subsequently, the user specifies the spatial selection in the Spatial Specifications window. The administrative mapping structure for two provincial ministries guides the user to selecting digital GIS files either symbolically or by map sheet number. Once the map set selection is complete, SEIDAM activates the Smart Access software agents. Their task is to start the relational DBMS where the image and GIS metadata reside and submit SQL queries that will identify all of the GIS data available, as well as all of the

TM imagery covering the selected maps⁴. The metadata returned by the SQL queries will include the processing the data have undergone. The Smart Access agents then place this information in the SEIDAM knowledge base and shut down the DBMS.

SEIDAM then activates the case-based reasoning system to which it submits the user goal “update the forest GIS file using TM imagery over the selected map set”. The reasoning system polls each of the remote sensing and GIS agents for their planning operator descriptions. It then attempts to find a case that solved a forest GIS file update problem. If a case is found, it is checked to ensure that it can be applied to the current problem by matching the current problem’s goal and initial world (e.g.: GIS map number) with the case’s goal and footprint. Although the retrieved case may match the current problem’s goal and initial world, it can still require adaptation to account for differences between the case and the current problem. If a matching case cannot be found or if adaptation is not successful, the reasoning system uses goal regression to find a solution. The solution will consist of an ordered list of processing tasks that must be executed to update the forest cover. Each processing task corresponds to one remote sensing or GIS software agent. The reasoning system assesses the processing states of the remote sensing and GIS data and only includes in the solution agents for which processing is required.

The reasoning system then activates each of the processing agents listed in the solution. As each agent executes, it modifies the contents of SEIDAM’s knowledge base by adding and deleting information and performing the necessary processing. Upon successful execution, the information added and deleted from the knowledge base should agree with the add and delete lists described by the planning operator provided by each agent. This allows the reasoning system to ensure that the processing carried out by each agent is successful before activating the next one. The solution for the forest update problem is:

```
copy_files_to_working_dir,  
trim_to_dlg,  
import_hydrology,  
erdascp_lakes,  
sieve_lakes,  
lakes_to_bitmap,  
trim_to_dem,  
create_tin,  
erdascp_dem,  
copy_bitmap,  
set_georeference_tm_large,  
create_tm_small,  
dem_to_pix_image_match,  
set_georeference_tm_small,  
tercom,  
export_clear_cuts_to_pci,  
image_segmentation,  
export_clear_cuts_to_arc,
```

⁴ In the current SEIDAM prototype, the update of a forest cover map is a process that uses TM imagery by default. Work is currently being carried out to determine how to automate the use of other types of remote sensing data.

- **copy_files_to_working_dir:** this agent will copy the TM image and GIS files from the robotic mass storage device to a working directory on disk (the map in Figure 5.3.1 A and the image in Figure 5.3.1. B are copied to a working directory).
- **trim_to_dlg:** this agent translates TRIM GIS data into the digital line graph standard readable by Arc/Info.
- **import_hydrology:** this agent creates an Arc/Info hydrology coverage.
- **erdascp_lakes:** this agent translates the hydrology coverage into a PCI image file.
- **sieve_lakes:** since the hydrology data contains many small lakes, a smoothing filter is used to remove any lake smaller than nine pixels or 5500 m².
- **lakes_to_bitmap:** convert the smoothed hydrology image into a bit map.
- **trim_to_dem:** creates a point elevation file readable by Arc/Info from TRIM data.
- **create_tin:** create a DEM from a triangulated irregular network constructed with the point elevation file and place the result in an Erdas file format for input to PCI.
- **erdascp_dem:** create a PCI image file from the Erdas DEM file.
- **copy_bitmap:** copy the hydrology bit map into the DEM image file.
- **set_georeference_tm_large:** get geo-reference information from the TM image file and DEM image file and place it in SEIDAM's knowledge base,
- **create_tm_small:** create a small TM file that fits over the current map and copy a subset image from the original TM image file.
- **dem_to_pix_image_match:** copy DEM into small TM file.
- **set_georeference_tm_small:** add geo-reference information to small TM file.
- **tercom:** apply topographic relief to small TM file.
- **export_clear_cuts_to_pci:** export old clear cut vectors from Arc/Info to PCI.
- **image_segmentation:** create a gradient image (Figure 5.3.1 E) and apply segmentation to the image thus producing the image in 5.3.1 F (segmentation is an image classification algorithm that allows the user to point out delineated homogeneous areas of an image and label them with one of several classes: old growth forest, new growth forest, denudation, water, etc.)
- **export_clear_cuts_to_arc:** export the new clear cut vectors from PCI to Arc/Info (use mask produced from forest cover, Figure 5.3.1 C, and segmented image to identify new cutblocks, Figure 5.3.1 G).

As SEIDAM activates each of the agents in the plan above, the images below, Figure 5.3.1, either serve as input to some of the processes or are output from some of the processes. Image A is the original forest cover map around the Greater Victoria Watershed District (GVWD) on Vancouver Island in southwestern British Columbia. The royal blue in the middle of the image is Sooke lake, the black area bounding the image is outside the administrative zone managed by the GVWD and the other polygons are forested areas differing in main species. Image B is a LANDSAT TM image with channels three (red), four (green) and five (blue) combined. The darker areas of green are old growth forests, the lighter shades of green are areas of reforestation and the pink areas result from bare areas (natural rock surfaces, clear-cuts, forest fire, etc.) Image C is a mask derived from the forest cover in image A, that is then used to select the area of interest on the TM image. Image D, although not produced by the plan described above, shows the area on the TM image that is of interest for the current problem. Image E is the gradient image created

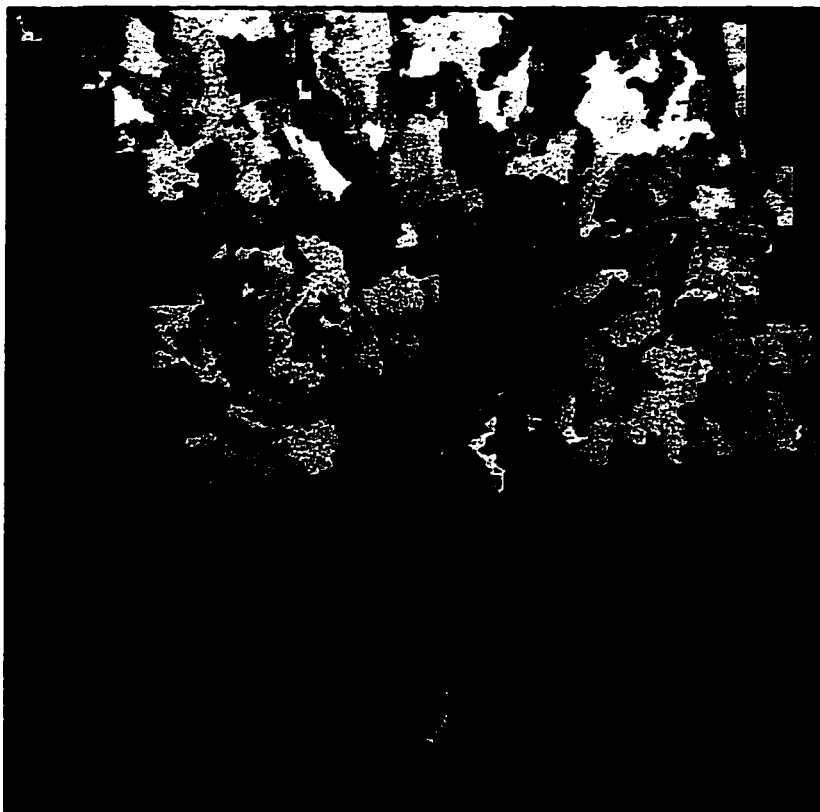


Image A - Forest cover map with GVWD administrative boundaries

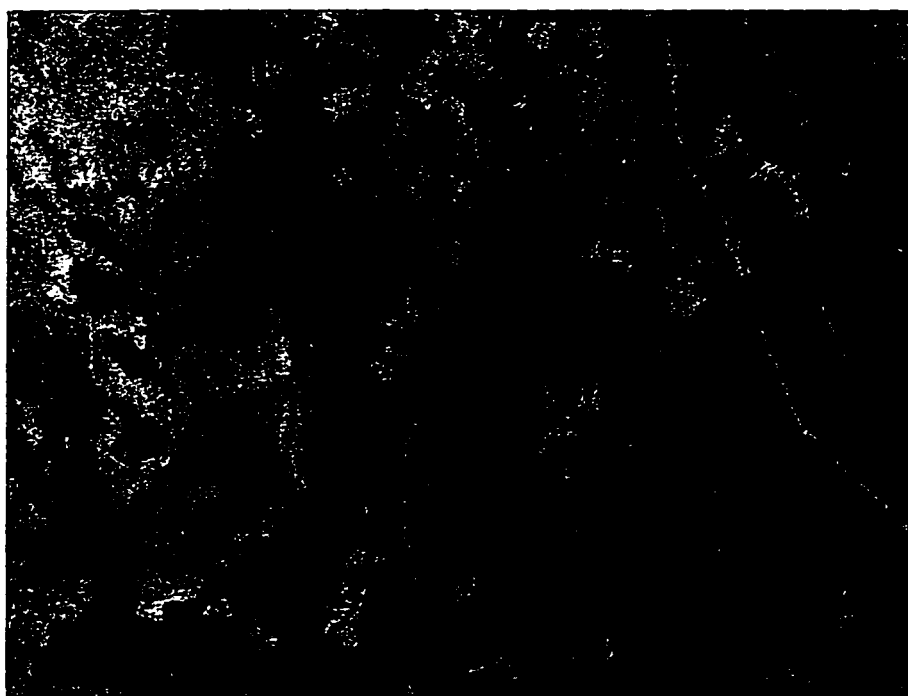


Image B - LANDSAT TM image over GVWD, channels three, four and five.



Image C - Mask used to identify areas of interest on the classified image.



Image D - Mask overlaid on TM image.

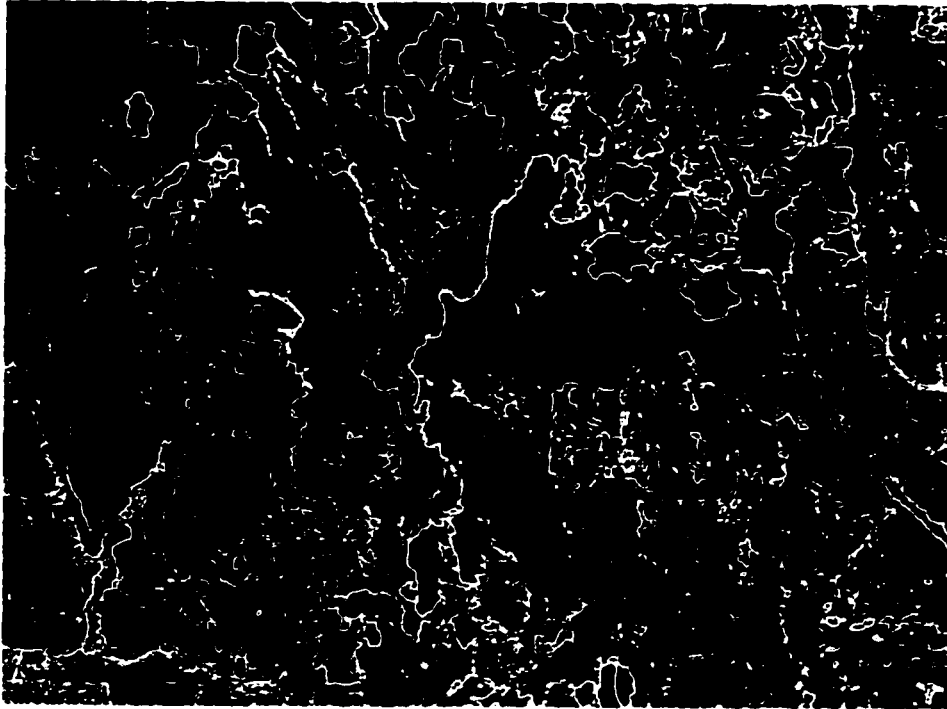


Image E - Gradient image created from image B.



Image F - Segmented image created from gradient image and user assistance (pink: clear-cuts; light green: new growth; dark green: old growth; blue: water; other: unclassified).



Image G - Forest cover mask overlaid on classified image.

Figure 5.3.1 Integration of LANDSAT TM imagery and GIS forest cover resulting from execution of plan.

during the segmentation. Image F is the image resulting from classifying the segmented image with the user's assistance. Image G is the map overlay of the forest cover mask onto the classified image. The final step in the plan will export the newly identified denuded areas from the image analysis format to the GIS format and update the polygon attribute database.

After the agents have all executed, SEIDAM, once again, activates the Smart Access software agents. Their task now is to start the relational DBMS and submit Data Manipulation Language (DML) commands to update the image and GIS metadata databases to reflect the processing that has been carried out on the image and GIS files. Once this has been completed, the Smart Access agents will move the image and GIS data into the appropriate locations on the robotic data storage device.

Domain experts were involved in the design of the knowledge bases used by the reasoning system and consider this plan very useful for the problem at hand. In fact, after the system had created the plan, SEIDAM successfully executed it to their satisfaction.

5.4 SEIDAM Agents: An overview

SEIDAM agents must be able to use image analysis, database management and GIS software. For instance, in order to use remotely sensed data with GIS data, it is necessary to translate remote sensing data into a GIS format or GIS data into an image format. This is usually a simple task that can be carried out by using either the image analysis software or the GIS software. Under normal circumstances, a human user would provide the data file names and the necessary parameters to the software to accomplish the transformation. SEIDAM uses a software agent to perform this task. In order to do so, the software agent must appear as a human user to the software. Therefore, the software's input and output streams are redirected to TCP/IP sockets and fed to a software agent. The software agent can then send instructions to the software on its input stream as well as observe its behaviour on its output stream.

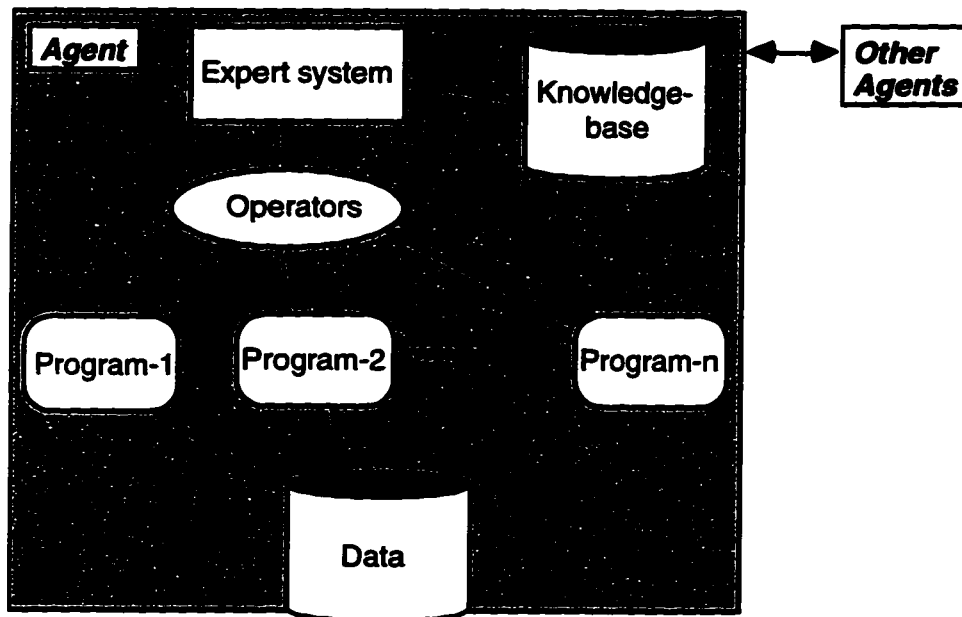


Figure 5.4.1 - Agent architecture.

Software agents used by SEIDAM were specifically designed to work in conjunction with the problem solver presented in this thesis. Each software agent, Figure 5.4.1, is composed of: an expert system, a knowledge-base, a Prolog module containing a set of operators and a set of programs (software packages such as PCI image analysis or ESRI Arc/Info GIS). All of the agents were created with a system called PAROT⁵. This ensures that all agents are consistent with the requirements of PALERMO. PAROT uses a graphical user interface to guide a domain expert through the creation steps of an agent, i.e. agents are the result of training. Once complete,

⁵ PAROT currently only has online documentation restricted to Natural Resources Canada at the Pacific Forestry Centre's Advanced Forest Technologies group. The web page URL is:
http://www.aft.pfc.forestry.ca/library/Experts_Guide/parot/index.html

PAROT uses its internal model of the agent to generate each of the components shown in Figure 5.4.1.

The expert system component of the agent consists of a set of procedural rules and a set of relational rules. In the current implementation, an agent's expert system is composed of procedural rules only. Relational rules will eventually be used to express knowledge that the agents learn during their use. Procedural rules are sequences of steps that represent states and state transitions. A procedural rule normally fires an operator and then sets the next state according to the reaction of the program (a transition). This behaviour is a finite state machine (FSM) where each transition is deterministic.

The expert system draws on a knowledge base where its rules, as well as the objects it manipulates, are stored. Objects are stored in a class hierarchy where an object can inherit behaviour associated to its class via daemons connected to the object's attributes.

Operators in the prolog module are used by the expert system to send commands to programs as well as to receive their responses. A response can be the report of an error, the display of file header information or simply the program prompt indicating successful execution of the operator.

Within the Prolog module is a STRIPS operator that lists the agents pre-conditions and effects. This operator was created by PAROT when domain experts saved the agent after completing the training. By stepping through the expert system rules, PAROT can identify different execution threads as well as determine all knowledge-base accesses. Knowledge-base accesses translate into pre-conditions and effects. For each execution thread there exists pre-conditions and effects which allows PAROT to generate a STRIPS operator. The STRIPS operators for SEIDAM are listed in Appendix D.

Chapter 6

Conclusion

The thesis has described the design, implementation and evaluation of a problem solving system that successfully integrates search-based planning, analogical reasoning and machine learning components. This chapter summarizes the contributions of each component of the system, presents a brief discussion of the results and discusses directions for future work.

6.1 System components

The main components of the system are: the search-based planner, the case-based reasoning system, the goal-directed generalization procedure, and the short replay and plan merging algorithms.

As has been discussed in chapter 2, goal regression is incomplete by virtue of using a linear approach to solving goals and subgoals resulting from operator selection and expansion. The main advantage to such a search mechanism is that it is highly focused and efficient. The main disadvantage is that, because of its incompleteness, solutions may be overlooked but they needn't be. By complementing goal-regression with an analogical reasoning system, it is possible to expand the search space coverage as was discussed in section 4.1.

The analogical reasoning component is based mainly on transformational analogy, i.e. complete cases are retrieved and adapted for the new problem. This type of case-based reasoning system normally suffers from the inability to solve problems that do not satisfy pre-defined similarity criteria. Since this system has planning operators at its disposal, the use of a similarity

heuristic coupled with the applicability verification of the operators ensures that the adaptation of cases can be made without allowing unacceptable substitutions. Also, if the system does not have a case to address the new problem, it cannot rely on search to attempt to solve it. By allowing the system to use a search mechanism when cases do not exist or can not be adapted, the transformational analogy system can benefit through covering a larger set of problems in the application domain.

Case-based reasoning systems normally rely on retrieval and adaptation of cases. The approach taken to the implementation of the analogical component presented in this thesis was to generalize cases. This was accomplished by creating a new case via adaptation of an existing case. Since two cases were readily available, the original case from the case base and the case resulting from the adaptation, least general generalization was applied to the two. This generalization was subsequently placed in the case base replacing the case that had been adapted. Generalized cases in the case base therefore covered more than problems for which they were created or adapted thus reducing the size of the case base and addressing the utility problem.

The short replay algorithm was mainly inspired by the derivational replay mechanism found in derivational analogy systems. The main difference was that for derivational analogy the choices made during past planning episodes are replayed whereas for the system presented here, the replay algorithm performs two tasks: it matches the footprint against the current state of the world; it ensures that all bindings in the case do not conflict with the current state of the world by replaying the plan in the case. As a result of replay, the algorithm expands the operators stored in the case to include pre-condition lists and operator effects in order to merge the case into the current plan.

Solutions for individual goal statement conjuncts are created by either search-based planning or analogical reasoning as described above. Upon successful creation of solutions for individual conjuncts, the new plans must be integrated into the current world. The system calls on the merge procedure to accomplish this task. The merge algorithm is based on other approaches as discussed in the literature review. The main concern of this component is to merge two cases into a given world. When this succeeds, a new world is created where the goals statement conjuncts are all satisfied and all possible conflicts between operators have been resolved either by imposing an ordering on the operators or by effecting repairs where conflicts cannot be resolved.

6.2 Results and contributions

The results presented in chapter 4 introduce several contributions derived from integrating search-based planning with analogical reasoning.

The results show that a linear planning system such as goal regression can be used as the base search mechanism. Once the search engine has been tightly coupled to a case-based reasoning system, it becomes a non-linear problem solver. As was shown in section 4.1, eager re-use of cases allows the system to create solution that are achievable only by a non-linear approach.

For case-based reasoning, one of the main results is that a simple definition for a case can impact the system's overall memory requirements. Indeed, by restricting each case to being able to solve single conjunct goal statements, the case-base covers a very large set of problems with very few cases. The results reported here show that for the logistics transportation domain, a case-base of approximately 35 cases suffices to solve goal statements having up to ten conjuncts. The number of conjuncts reported here is not a restriction on the number of conjuncts but rather an arbitrarily chosen upper bound. The total number of cases created by the system is very significant since other work attempting to solve problems in the logistics transportation domain reported starting the problem solver with approximately 1000 cases at its disposal. These case-bases contained solutions for one subgoal problems, two subgoal problems, and so on up to the maximum number of allowable conjuncts in the goal statements. To create the results herein, the case-base was always empty from the outset of the problem solving exercise.

The second result shows that, after generating a case-base that covers the logistics transportation domain, performance of the case-based reasoning system is slightly better than regression alone. The point here is that the integration of the different problem solving approaches presented here does not degrade the performance with respect to goal regression. This is mainly due to the generalization of cases since, once generalized, no adaptation is required.

The system also scales up very well. Experiments in the *one-way-rocket* and the *extended-STRIPS* domains showed that the system can work well in "toy" domains and exhibits the desired behaviour during problem solving. By generating and testing the system on the *logistics transportation* domain, I have shown that the approach is applicable to more complex domains.

By creating simple software agents that describe their behaviour with a STRIPS planning operator, I have shown that the system can indeed solve difficult problems in a complex, real world domain such as data integration for digital forest inventory update using remote sensing imagery. Many experiments have been conducted to show how collaboration between the problem solving system and software agents can address very complex problems in the environmental information system domain. These problems include:

- calibration of AIRSAR data [Bhogal, Goodenough et al. 1996] (the AIRSAR sensor is an air-borne synthetic aperture radar system that is also carried on the space shuttle and will be mounted on a satellite as part of NASA's Earth Observing System, EOS; this sensor can be used to compute biomass and forest timber volume, important indicators for international environmental agreements),
- management of large remote sensing and GIS data sets for environmental monitoring [Matwin, Charlebois et al. 1994; Matwin, Charlebois et al. 1995; Charlebois, Goodenough et al. 1996; Charlebois, Goodenough et al. 1996],
- such as the calibration of AVIRIS data [Goodenough, Bhogal et al. 1995; Goodenough, Charlebois et al. 1995] (the AVIRIS sensor is an air-borne visible to infra-red imaging spectrometer that will also be include in the EOS; this is a high spectral resolution, 224

channels, sensor that can be used, for instance, to determine water vapor distribution in the atmosphere),

- integration of multi-temporal imaging Spectrometre data for forest health monitoring [Bhogal, Goodenough et al. 1996],
- fusion of multi-temporal data-sets for forest cover change detection [Goodenough, Bhogal et al. 1995] (by fusing Thematic Mapper and Multi-Spectral Scanner imagery from the LANDSAT satellites with forest cover GIS files, it is possible to detect forest cover change over time and plot the results graphically), and forest cover depletion detection [Charlebois, Goodenough et al. 1993; Charlebois, Goodenough et al. 1994; Goodenough, Charlebois et al. 1994],
- imaging Spectrometre analysis for forestry [Goodenough, Charlebois et al. 1995],

Finally, although no measurements are presented, the definition of relevance of footprint literals for goal-directed generalization had a significant impact on the performance of the system. Before the definition was introduced, the time required to compute an acceptable substitution was an order of magnitude greater. Footprint literal relevance was also used in the footprint verification procedure. As a consequence, cases that were not applicable were more quickly discarded.

6.3 Future work

There are several areas of future work that could provide opportunities to improve the problem solving system as well as modifications to other components that would likely have an impact on the system's behaviour and performance.

The search engine chosen for the current implementation of the problem solver is a goal regression search algorithm. It was chosen because of its simple implementation and its highly focused behaviour leading to relatively efficient performance. There are two issues that have been addressed by other researchers with respect to the system's ability to create plans: linearity versus non-linearity and total-order versus partial-order planning. Systems such as TO [Minton, Bresina et al. 1994] have shown that non-linear, partial-order planning need not unacceptably sacrifice the goal-directed approach inherent to goal-regression. Since these systems cover a larger slice of the search space from the outset, evaluating the impact of replacing goal-regression by this type of search-engine is in order.

Eager-replay was introduced in [Ihrig and Kambhampati 1994]. It showed that partial-order planners using derivational analogy could greatly improve their performance. Although the system presented here uses transformational analogy as its underpinning, since planning states are represented as a current plan, it should be straightforward to evaluate the possible benefits of integrating eager replay to eager case reuse. In combination with a partial-order planner such as TO, this new approach could have a significant impact on the overall performance.

In the introduction to chapter 3, I suggested that researchers have taken two main tacks to search-based problem solving. The first is to start with a problem solver that covers the search

space completely and introduce search control to reduce the number of nodes visited. The second approach is to start with a restrictive search engine, such as goal-regression, and increase the coverage of the search space. Derivational analogy, as introduced by Veloso [Veloso 1992], starts with a complete search engine and uses analogy as a knowledge-based search control mechanism. The problem solver presented here could profit from the use of derivational analogy to expand its coverage of the search space by replaying derivations that would not occur in a linear planner.

The merge algorithm is based on approaches aimed at merging plans. As was discussed in chapter 2, there also is research being conducted in the area of operator merging. The impact of these approaches is important in that they tend to remove reduce the number of operators in a plan without substituting the remaining operators with new ones. Obviously, the cost of executing plans that have merged operators is less than executing the original plans. In application domains such as forest inventory update and environmental monitoring, the impact has the potential of being very important since many processing steps tend to be CPU intensive. The application could therefore greatly benefit from a reduced number of processing tasks resulting from operator merging.

The impact of integration of the problem solver into the SEIDAM environment where it benefited greatly from its interaction with software agents is undeniable. However, the current implementation of the software agents is one where the agents do not perform any learning. If the agents were given this ability, impact on the reasoning system would certainly be significant. Although the agents could be adapted to properly create a STRIPS operator more dynamically when it has learned new knowledge, the problem solver would need to assess the state of the case-base with respect to the changes made to the operator.

The results presented in this thesis were, for the most part, based on the *logistics transportation* domain. Although this served as a good starting point to showcase the system's behaviour, its ultimate test will be how it performs in an operational environment such as the *forest information management* domain. As was discussed in chapter 5, plans produced by the reasoning system in the *forest information management* domain were qualitatively assessed and satisfied the requirements of the SEIDAM system. Upon completion of the SEIDAM prototype, an new empirical evaluation should be carried out on the reasoning system in order to ensure that the system behaves well in a real world environment.

References

- Aho A. V., Hopcroft J. E. and Ullman J. D. (1974). The Design and Analysis of Computer Algorithms. Reading Massachusetts, Addison-Wesley.
- Bain W. M. (1986). Case-Based Reasoning: A Computer Model of Subjective Assessment, Yale University.
- Bergadano F. and Gunetti D. (1995). Inductive Logic Programming: From Machine Learning to Software Engineering. London, England, The MIT Press.
- Bhogal A. S. P., Goodenough D. G., Charlebois D., Matwin S. and Niemann O. (1996). Intelligent Fusion and Analysis of AIRSAR Data for SEIDAM. IGARSS'96, Lincoln NB, pp. 1873-1876.
- Bhogal A. S. P., Goodenough D. G., Charlebois D., Matwin S., Portugal F., Barclay H., Thomson A. and Niemann O. (1996). SEIDAM for Forestry: Intelligent Fusion and Analysis of Multi-Temporal Imaging Spectrometer Data 26th International Remote Sensing Symposium, Vancouver, BC, Canada, pp. 59-62.
- Carbonell J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. Machine Learning: An Artificial Intelligence Approach. R. S. Michalski, J. G. Carbonell and T. M. Mitchell, Morgan Kaufman. II.
- Carbonell J. G., Knoblock C. A. and Minton S. (1990). Prodigy: An integrated architecture for planning and learning. Architecture for Intelligence. K. VanLehn. Erlbaum, Hillsdale, NJ.
- Chang C.-L. and Lee R. C.-T. (1973). Symbolic Logic and Mechanical Theorem Proving. New York NY, Academic Press.
- Chapman D. (1987). "Planning for conjunctive goals." Artificial Intelligence 32: 333-337.
- Charlebois D., Goodenough D. G., Bhogal A. S. P. and Matwin S. (1996). Case-Based Reasoning and Software Agents of Intelligent Forest Information Management. IGARSS'96, Lincoln NB, pp.2303-2306.
- Charlebois D., Goodenough D. G. and Matwin S. (1993). Machine Learning from Remote Sensing Analysis. IGARSS'93, Tokyo, Japan, pp. 165-172.
- Charlebois D., Goodenough D. G. and Matwin S. (1994). Case-Based Reasoning in an Intelligent Information System for Forestry. SPIE'94, Orlando, FL, pp. 64-74.
- Charlebois D., Goodenough D. G., Matwin S., Bhogal A. S. P. and Barclay H. (1996). Planning and Learning in a Natural Resource Information System. Canadian AI, Toronto ON Canada, pp. 187-199.

- Collins G. C. (1987). Plan Creation: Using Strategies as Blueprints, Yale University.
- DeJong G. and Mooney R. (1990). Explanation-Based Learning: An Alternative View. Readings in Machine Learning. J. W. Shavlik and T. G. Dietterich. San Mateo, CA, Morgan-Kaufmann: 452-467.
- Ernst G. W. and Newell A. (1969). GPS: A Case Study in Generality and Problem Solving. New York, NY, Academic Press.
- Etzioni O. and Weld D. (1994). "A Softbot-Based Interface to the Internet." Communications of the ACM **37**(7): 72-79.
- Fikes R. E., Hart P. E. and Nilsson N. J. (1971). "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." Artificial Intelligence **2**: 189-208.
- Fikes R., Hart P. and Nilsson N. (1981). Learning and Executing Generalized Robot Plans. Readings in AI. B. Webber and N. Nilsson. Palo Alto CA.
- Foulser D. E., Li M. and Yang Q. (1992). "Theory and algorithms for plan merging." Artificial Intelligence **57**: 143-181.
- Goodenough D. G., Bhogal P., Charlebois D., Heyd M., Matwin S. and Portigal F. (1995). Knowledge-based Imaging Spectrometer Analysis. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Victoria BC Canada, pp. 287-291.
- Goodenough D. G., Bhogal P., Charlebois D., Matwin S. and Niemann O. (1995). Intelligent Data Fusion for Environmental Monitoring. IGARSS'95, Florence Italy, pp. 2157-2160.
- Goodenough D. G., Charlebois D., Bhogal P., Heyd M., Matwin S., Niemann O. and Portigal F. (1995). Knowledge-Based Imaging Spectrometer Analysis and GIS for Forestry. IGARSS'95, Florence Italy, pp. 464-467.
- Goodenough D. G., Charlebois D. and Matwin S. (1994). "Automating Reuse of Software for Expert Systems Analysis of Remote Sensing Data." IEEE Trans on GRS **32**(3): 525-533.
- Goodenough D. G., Charlebois D., Matwin S., MacDonald D. and Thomson A. J. (1994). Queries and Their Application to Reasoning with Remote Sensing and GIS. IGARSS'94, Pasadena CA, pp. 1199-1203.
- Goodenough D. G., Bhogal A. S. P., Charlebois D. and Holloway D. (1994). System of Experts for Intelligent Data Management (SEIDAM). NASA-AISRP Workshop, Boulder CO.
- Hammond K. J. (1989). Case-Based Planning: Viewing Planning as a Memory Task. Boston MA, Academic Press.

- Ihrig L. H. and Kambhampati S. (1994). Derivation Replay for Partial-Order Planning. AAAI'94, Seattle, WA, pp. 992-997.
- Kautz H., Selman B., Coen M., Ketchpel S. and Ramming S. (1994). An experiment in the design of software agents. AAAI'94, Seattle, WA, pp. 438-443.
- Kolodner J. L. and Riesbeck C. K. (1986). Experience, Memory and Reasoning. Hillsdale, NJ, Lawrence Erlbaum Associates.
- Langley P., Matwin S. and Allen J. A. (1992). A Unified Framework for Domain-Specific Planning. Workshop on Incremental Modification - Stanford Spring Symposium Series, pp. 7-12.
- Lansky A. L. (1991). Localized Search for Multiagent Planning. IJCAI'91, Darling Harbour, Sydney, Australia, pp. 252-258.
- Lansky A. L. and Philpot A. (1993). AI-Based Planning for Data Analysis Tasks. Conference on Artificial Intelligence Applications, pp. 390-398.
- Lavrac N. and Dzeroski S. (1994). Inductive Logic Programming: Techniques and Applications. Ljubljana, Slovenia, Ellis Horwood Limited (Hertfordshire, England).
- Matwin S., Charlebois D. and Goodenough D. G. (1994). Machine Learning and Planning for Data Management in Forestry. AAAI'94: Workshop for Environmental Applications, Seattle, WA, pp. 83-90.
- Matwin S., Charlebois D. and Goodenough D. G. (1995). Training Agents in a Complex Environment. IEEE: Conference on Artificial Intelligence Applications, Los Angeles, California, pp. 94-100.
- Matwin S., Charlebois D., Goodenough D. G. and Bhogal P. (1995). "Machine Learning and Planning for Data Management in Forestry." IEEE Expert Systems 10(6): 35-41.
- Minsky M. (1994). "A Conversation with Marvin Minsky About Agents." Communications of the ACM 37(7): 23-29.
- Minton S., Bresina J. and Drummond M. (1994). "Total-Order and Partial-Order Planning: A Comparative Analysis." Journal of Artificial Intelligence Research(December 1994): 227-262.
- Minton S., Knoblock C. A., Kuokka D. R., Gil Y., Joseph R. L. and Carbonell J. G. (1989). PRODIGY 2.0: The manual and tutorial, School of Computer Science, Carnegie Mellon University.
- Minton S. and Zweben M. (1993). Learning, Planning, and Scheduling: An Overview. Machine Learning Methods for Planning. S. Minton. San Mateo, CA, Morgan Kaufmann Publishers Inc.: 1-29.

- Mitchell T. M., Keller R. M. and Kedar-Cabelli S. T. (1990). Explanation-Based Generalization: A Unifying View. Readings in Machine Learning. J. W. Shavlik and T. G. Dietterich. San Mateo, CA, Morgan-Kaufmann: 435-451.
- Mitchell T., Caruana R., Freitag D., McDermott J. and Zabowski D. (1994). "Experience with a learning personal assistant." Communications of the ACM 37(7): 80-91.
- Mitchell T., Mahadevan S. and Steinberg L. (1985). LEAP: A learning apprentice for VLSI design. Ninth IJCAI, LA, pp. 573-580.
- Newell A. and Simon H. A. (1963). GPS: A program that simulates human thoughts. Computers and Thought. E. A. Feigenbaum and J. Feldman. New York, NY, McGraw-Hill.
- Plotkin G. (1971). Automatic Methods of Inductive Inference, Edinburgh.
- Riesbeck C. K. and Schank R. C. (1989). Inside Case-Based Reasoning. Hillsdale NJ, Lawrence Erlbaum Associates.
- Sacerdoti E. D. (1975). The Non-Linear Nature of Plans. IJCAI'75, pp. 206-213.
- Schank R. C. (1986). Explanation Patterns: Understanding Mechanically and Creatively. Hillsdale, NJ, Lawrence Erlbaum Associates.
- Sussman G. J. (1973). A Computational Model of Skill Acquisition, MIT.
- Tate, A. (1975). Interacting Goals and Their Use, IJCAI-75, Tbilisi, USSR. pp.215-218.
- Tate, A. (1977). Project Planning Using a Hierarchical Non-linear Planner, Department of Artificial Intelligence Report 25, Edinburgh University.
- Veloso M. (1992). Learning by Analogical Reasoning in General Problem Solving, Carnegie Mellon University.
- Veloso M. (1993). Planning for complex Tasks: Replay and Merging of Multiple Plans. Working notes of the AAAI Spring Symposium on "Foundations of Automatic Planning: The Classical Approach and Beyond", Stanford, CA, pp. 146-150.
- Veloso M. (1994). Flexible Strategy Learning: Analogical Replay of Problem Solving Episodes. AAAI'94, Seattle, WA, pp. 595-600.
- Yang Q. (1992). "Merging Separately Generated Plans with Restricted Interactions." Computational Intelligence 8(4): 648-676.

Appendix A

One-Way Rocket and Extended STRIPS Domain Operators

One-Way Rocket Domain:

```
initial_world(  
    at( obj1, loc1 ) and  
    at( obj2, loc1 ) and  
    at( obj3, loc1 ) and  
    at( obj4, loc1 ) and  
    at( rocket, loc1 )  
).  
  
obj1 is_a package.  
obj2 is_a package.  
obj3 is_a package.  
obj4 is_a package.  
loc1 is_a location.  
loc2 is_a location.  
  
load_rocket( Object, Location ) ::  
    if      Object is_a package and  
            at( Object, Location ) and  
            at( rocket, Location )  
    then delete at( Object, Location )  
            add  inside( Object, rocket ).  
  
unload_rocket( Object, Location ) ::  
    if      Object is_a package and  
            inside( Object, rocket ) and  
            at( rocket, Location )  
    then delete inside( Object, rocket )  
            add  at( Object, Location ).  
  
move_rocket ::  
    if      at( rocket, loc1 )  
    then delete at( rocket, loc1 )  
            add  at( rocket, loc2 ).
```

Extended STRIPS Domain:

```
initial_world(  
    connects( door( a ), room( 1 ), room( 2 ) ) and  
    connects( door( b ), room( 2 ), room( 3 ) ) and  
    connects( door( c ), room( 3 ), room( 4 ) ) and  
    inroom( robot, room( 1 ) ) and  
    inroom( light( 1 ), room( 2 ) ) and  
    inroom( box( 1 ), room( 3 ) ) and  
    inroom( light( 2 ), room( 4 ) ) and  
    inroom( box( 2 ), room( 4 ) ) and  
    inroom( door( a ), room( 1 ) ) and  
    inroom( door( a ), room( 2 ) ) and  
    inroom( door( b ), room( 2 ) ) and  
    inroom( door( b ), room( 3 ) ) and  
    inroom( door( c ), room( 3 ) ) and  
    inroom( door( c ), room( 4 ) ) and  
    pushable( box( 1 ) ) and  
    pushable( box( 2 ) ) and  
    setting( light( 1 ), off ) and  
    setting( light( 2 ), off ) and  
    onfloor  
).  
  
goto( Object, Room ) ::  
    if  
        onfloor and  
        inroom( Object, Room ) and  
        inroom( robot, Room ) and  
        not( nexttto( robot, Object ) )  
    then delete all( nexttto( robot, _ ) )  
        add nexttto( robot, Object ).  
  
pushto( Obj1, Obj2, Room ) ::  
    if  
        onfloor and  
        pushable( Obj1 ) and  
        inroom( Obj2, Room ) and  
        inroom( Obj1, Room ) and  
        nexttto( robot, Obj1 )  
    then delete all( nexttto( Obj1, _ ) ) and  
        all( nexttto( _, Obj1 ) )  
        add nexttto( robot, Obj1 ) and  
        nexttto( Obj1, Obj2 ) and  
        nexttto( Obj2, Obj1 ).  
  
gothru( Door, Room1, Room2 ) ::  
    if  
        connects( Door, Room2, Room1 ) or  
        connects( Door, Room1, Room2 ) and  
        not( closed( Door ) ) and  
        onfloor and  
        inroom( robot, Room1 ) and  
        nexttto( robot, Door )  
    then delete all( nexttto( robot, _ ) ) and  
        inroom( robot, Room1 )  
        add inroom( robot, Room2 ) and  
        nexttto( robot, Door ).  
  
pushthru( Door, Obj, Room1, Room2 ) ::  
    if  
        pushable( Obj ) and  
        onfloor and  
        inroom( Obj, Room1 ) and  
        connects( Door, Room2, Room1 ) or  
        connects( Door, Room1, Room2 ) and  
        not( closed( Door ) ) and  
        inroom( robot, Room1 ) and  
        nexttto( Obj, Door ) and  
        nexttto( robot, Obj )  
    then delete inroom( robot, Room1 ) and  
        inroom( Obj, Room1 ) and  
        all( nexttto( Obj, _ ) ) and  
        all( nexttto( _, Obj ) )  
        add inroom( robot, Room2 ) and
```

```

        inroom( Obj, Room2 ) and
        nextto( robot, Obj ).

climb( box( B ) ) ::
    if      nextto( robot, box( B ) ) and
            onfloor
    then delete onfloor
        add  on( robot, box( B ) ).

climboff( box( B ) ) ::
    if      on( robot, box( B ) )
    then delete on( robot, box( B ) )
        add  onfloor and
            nextto( robot, box( B ) ).

turn( light( S ), Setting ) ::
    if      nextto( box( S ), light( S ) ) and
            on( robot, box( S ) )
    then delete setting( light( S ), _ )
        add  setting( light( S ), Setting ).

```

Appendix B

Goal Directed Generalization: A Detailed Example

The example given here demonstrates how the goal directed generalization (GDG) procedure behaves on a complex problem. Given the new goal statement “at(obj(obj4),post_office(po0))” and an existing case for the goal “at(obj(obj3),post_office(po0))”, which satisfies the similarity criteria presented in chapter three, the problem is to find a substitution that, if applied to the case, would produce a plan to satisfy the new goal statement. Table B.1 shows the case retrieved from the case base on the right and the desired plan and relevant facts from the new problem’s initial world on the left.

As was shown in chapter three, GDG has three steps: find a substitution that can be applied to the case’s footprint such that it will produce a new footprint where all the literals hold in the new problem’s initial world; apply the substitution to both the footprint and the plan in the case; apply least general generalization to the original case and the adapted plan to produce a generalized case.

Case	New problem
goal: at(obj(obj3),post_office(po0)) footprint: at(obj(obj3),post_office(po6)), at(plane(p3),airport(ap3)), at(truck(truck0),post_office(po0)), at(truck(truck3),post_office(po6)), city(airport(ap0),city(city0)), city(airport(ap3),city(city3)), city(obj(obj3),city(city3)), city(post_office(po0),city(city0)), city(post_office(po6),city(city3)), city(truck(truck0),city(city0)), city(truck(truck3),city(city3)) plan: load_truck(obj3),truck3,po6) drive_truck(truck3,po6,ap3) unload_truck(obj3,truck3,ap3) load_plane(obj3,p3,ap3) fly_plane(p3,ap3,ap0) unload_plane(obj3,p3,ap0) drive_truck(truck0,po0,ap0) load_truck(obj3,truck0,ap0) drive_truck(truck0,ap0,po0) unload_truck(obj3,truck0,po0)	goal: at(obj(obj4),post_office(po0)) relevant facts from initial world: at(obj(obj4),post_office(po8)), at(plane(p4),airport(ap4)), at(truck(truck0),post_office(po0)), at(truck(truck4),post_office(po8)), city(airport(ap0),city(city0)), city(airport(ap4),city(city4)), city(obj(obj4),city(city4)), city(post_office(po0),city(city0)), city(post_office(po8),city(city4)), city(truck(truck0),city(city0)), city(truck(truck4),city(city4)) desired plan: load_truck(obj4),truck4,po8) drive_truck(truck4,po8,ap4) unload_truck(obj4,truck4,ap4) load_plane(obj4,p4,ap4) fly_plane(p4,ap4,ap0) unload_plane(obj4,p4,ap0) drive_truck(truck0,po0,ap0) load_truck(obj4,truck0,ap0) drive_truck(truck0,ap0,po0) unload_truck(obj4,truck0,po0)

Table B.1

current substitution: [obj3/obj4] current footprint: 1. at(obj(obj4),post_office(po6)), 2. at(plane(p3),airport(ap3)), 3. at(truck(truck0),post_office(po0)), 4. at(truck(truck3),post_office(po6)), 5. city(airport(ap0),city(city0)), 6. city(airport(ap3),city(city3)), 7. city(obj(obj4),city(city3)), 8. city(post_office(po0),city(city0)), 9. city(post_office(po6),city(city3)), 10. city(truck(truck0),city(city0)), 11. city(truck(truck3),city(city3))	The first and 7th literals in the footprint are false. We must therefore compute a substitution that will make the first literal true, [po6/po8], and add it to the original substitution.
--	---

Table B.2

The substitution that would result in the plan on the left in table B.1 if it were applied to the case in the right column is: [obj3/obj4, po6/po8, truck3/truck4, city3/city4, ap3/ap4, p3/p4].

When the case is retrieved, the literals that are members of the footprint are all hold in the initial world, but are not relevant to the new goal statement.

current substitution: [obj3/obj4,po6/po8] current footprint: 1. at(obj(obj4),post_office(po8)), 2. at(plane(p3),airport(ap3)), 3. at(truck(truck0),post_office(po0)), 4. at(truck(truck3),post_office(po8)), 5. city(airport(ap0),city(city0)), 6. city(airport(ap3),city(city3)), 7. city(obj(obj4),city(city3)), 8. city(post_office(po0),city(city0)), 9. city(post_office(po8),city(city3)), 10. city(truck(truck0),city(city0)), 11. city(truck(truck3),city(city3))	The 4th, 7th and 9th literals in the footprint are now false. The system therefore computes a substitution to make the 4th literal true, i.e. [truck3/truck4]. It is added to the substitution.
--	---

Table B.3

The GDG procedure begins computing the substitution to change the case's goal statement to the new goal statement: [obj3/obj4]. It will then apply this substitution to the case's footprint shown in table B.2. As a result, the first and seventh literals in the footprint no longer hold in the initial world. GDG will select one of the false literals and compute a new substitution that will make it hold with respect to the new goal statement. The substitution [po6/po8] will make the first literal hold in the initial world.

current substitution: [obj3/obj4,po6/po8,truck3/truck4] current footprint: 1. at(obj(obj4),post_office(po8)), 2. at(plane(p3),airport(ap3)), 3. at(truck(truck0),post_office(po0)), 4. at(truck(truck4),post_office(po8)), 5. city(airport(ap0),city(city0)), 6. city(airport(ap3),city(city3)), 7. city(obj(obj4),city(city3)), 8. city(post_office(po0),city(city0)), 9. city(post_office(po8),city(city3)), 10. city(truck(truck0),city(city0)), 11. city(truck(truck4),city(city3))	The 7th, 9th and 11th literals in the footprint are now false. The substitution to make the 7th literal true is [city3/city4]. It is added to the substitution.
--	---

Table B.4

After applying the [po6/po8] substitution, the footprint in table B.3 is created. In addition to the seventh literal, now the fourth and ninth literals no longer hold.

GDG selects the fourth literal and computes the [truck3/truck4] substitution and applies it to produce the new footprint in table B.4

The seventh, ninth and eleventh literals do not hold in the initial world. GDG selects the seventh literal and computes the [city3/city4] substitution and applies it to produce the new footprint in table B.5.

current substitution: [obj3/obj4,po6/po8,truck3/truck4,city3/city4] current footprint: 1. at(obj(obj4),post_office(po8)), 2. at(plane(p3),airport(ap3)), 3. at(truck(truck0),post_office(po0)), 4. at(truck(truck4),post_office(po8)), 5. city(airport(ap0),city(city0)), 6. city(airport(ap3),city(city4)), 7. city(obj(obj4),city(city4)), 8. city(post_office(po0),city(city0)), 9. city(post_office(po8),city(city4)), 10. city(truck(truck0),city(city0)), 11. city(truck(truck4),city(city4))	The 7th, 9th and 11th literals are now true. The 6th literal, however, is now false. The substitution to make the 6th literal true is [ap3/ap4].
--	---

Table B.5

The three literals that did not hold previously, now hold in the initial world. However, the sixth literal no longer holds. GDG selects it and computes the [ap3/ap4] substitution and applies it to produce the new footprint in table B.6.

current substitution: [obj3/obj4,po6/po8,truck3/truck4,city3/city4,ap3/ap4] current footprint: 1. at(obj(obj4),post_office(po8)), 2. at(plane(p3),airport(ap4)), 3. at(truck(truck0),post_office(po0)), 4. at(truck(truck4),post_office(po8)), 5. city(airport(ap0),city(city0)), 6. city(airport(ap4),city(city4)), 7. city(obj(obj4),city(city4)), 8. city(post_office(po0),city(city0)), 9. city(post_office(po8),city(city4)), 10. city(truck(truck0),city(city0)), 11. city(truck(truck4),city(city4))	The 6th literal is now true but the 2nd is not. The substitution to make the 2nd literal true is [p3/p4].
--	--

Table B.6

Since the second literal no longer holds, GDG selects it and computes the [p3/p4] substitution and applies it to produce the new footprint in table B.7.

current substitution: [obj3/obj4,po6/po8,truck3/truck4,city3/ city4,ap3/ap4,p3/p4] current footprint: 1. at(obj(obj4),post_office(po8)), 2. at(plane(p4),airport(ap4)), 3. at(truck(truck0),post_office(po0)), 4. at(truck(truck4),post_office(po8)), 5. city(airport(ap0),city(city0)), 6. city(airport(ap4),city(city4)), 7. city(obj(obj4),city(city4)), 8. city(post_office(po0),city(city0)), 9. city(post_office(po8),city(city4)), 10. city(truck(truck0),city(city0)), 11. city(truck(truck4),city(city4))	All of the literals now hold in the initial world.
--	--

Table B.7

The substitution computation is now complete. The next step is to apply the substitution to the case's plan to create the new plan shown in Table B.1. GDG now has at its disposal both the initial case and the modified case. Before terminating, GDG will compute the least general generalization, figure B.1, that subsumes both plans. This new generalized plan will replace the old case in the case-base.

```

goal: at(obj(A),post_office(po0))
footprint:
  at(obj(A),post_office(C)),
  at(plane(E),airport(D)),
  at(truck(truck0),post_office(po0)),
  at(truck(B),post_office(C)),
  city(airport(ap0),city(city0)),
  city(airport(D),city(F)),
  city(obj(A),city(F)),
  city(post_office(po0),city(city0)),
  city(post_office(C),city(F)),
  city(truck(truck0),city(city0)),
  city(truck(B),city(F))
plan:
  load_truck(obj(A),truck(B),post_office(C))
  drive_truck(truck(B),post_office(C),airport(D))
  unload_truck(obj(A),truck(B),airport(D))
  load_plane(obj(A),plane(E),airport(D))
  fly_plane(plane(E),airport(D),airport(ap0))
  unload_plane(obj(A),plane(E),airport(ap0))
  drive_truck(truck(truck0),post_office(po0),airport(ap0))
  load_truck(obj(A),truck(truck0),airport(ap0))
  drive_truck(truck(truck0),airport(ap0),post_office(po0))
  unload_truck(obj(A),truck(truck0),post_office(po0))

```

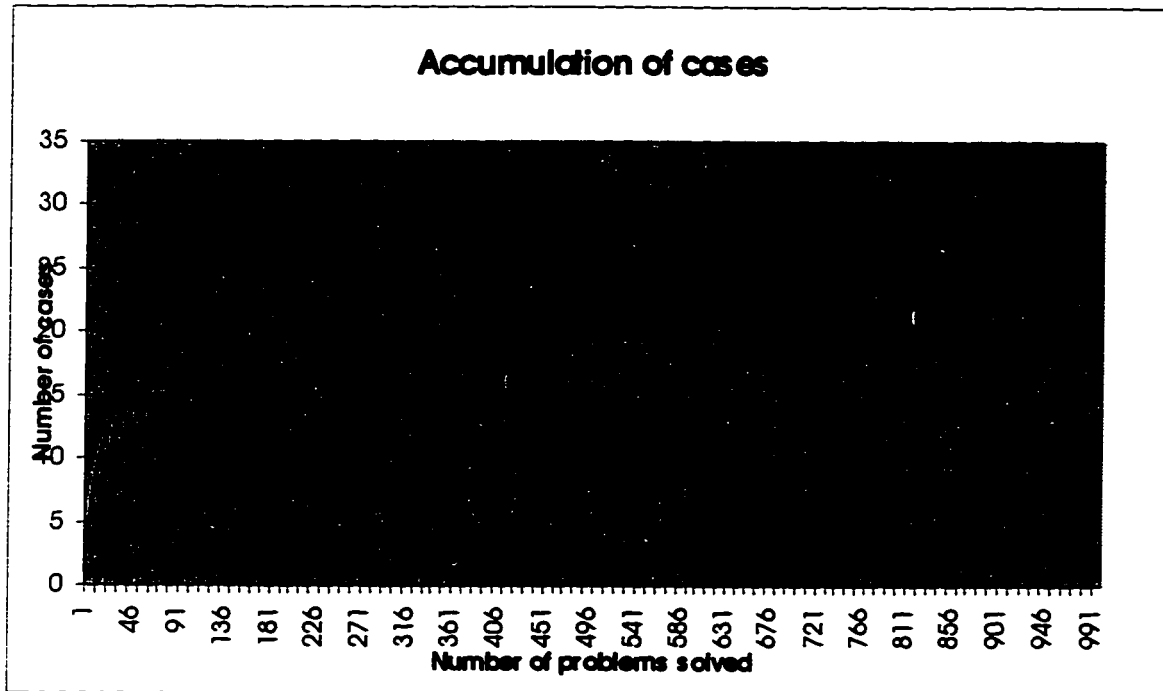
Figure B.1 - Generalized plan

Appendix C

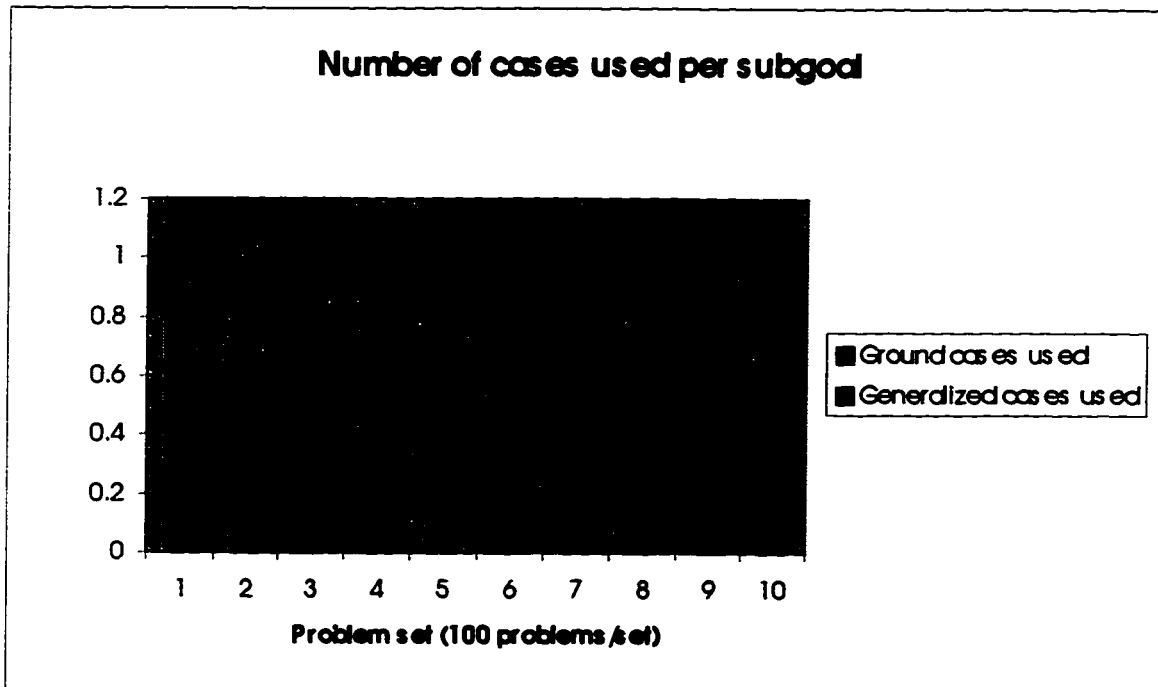
Empirical Results

Chapter 4 presented empirical results for the reasoning system. This appendix contains ten extra sets of experiments conducted in the logistics transportation domain. The graphs presented in section 4.2.4 showing how the system behaves when solving multiple conjunct goals have all been reproduced below. For each set of experiments, the system solved 1000 problems beginning with an empty case base. For each set of 1000 problems, the first 100 were one conjunct goal statements, the second 100 were two conjunct goal statements, up to the last 100 which were ten conjunct goal statements. The graphs are numbered by appendix letter, set and measure (e.g. graph C.1.1: appendix C, problem set 1, graph 1). Graph 1 in each set shows the accumulation of cases in the case-base. Graph 2 shows the average number of cases required to solve each conjunct in a goal statement. Graph 3 shows the adaptation effort measured in the number of recursive calls made to the generalization procedure. Graph 4 shows the adaptation effort in terms of the number modifications necessary to adapt a case to the current problem. Graph 5 shows the average solve time for each goal statement conjunct. Graph 6 plots the average merge time per problem (as the number of conjuncts grows, so does the time required to merge the solutions into the final plan). Graph 7 contains the average retrieval time per goal statement conjunct.

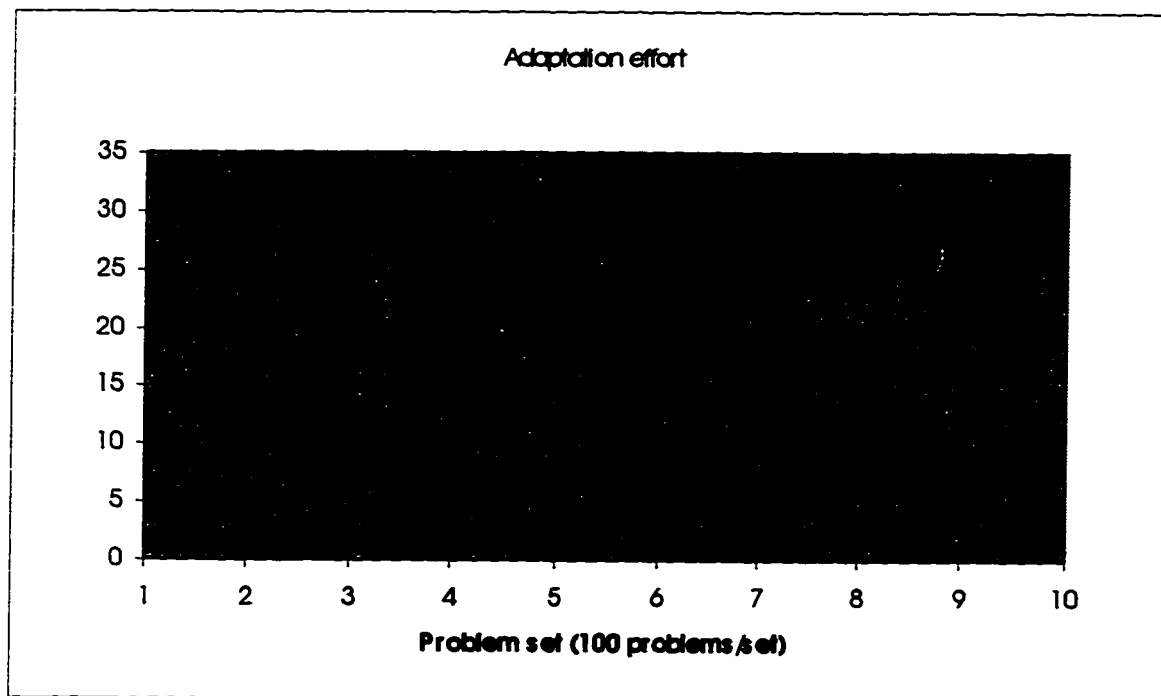
Problem set 1



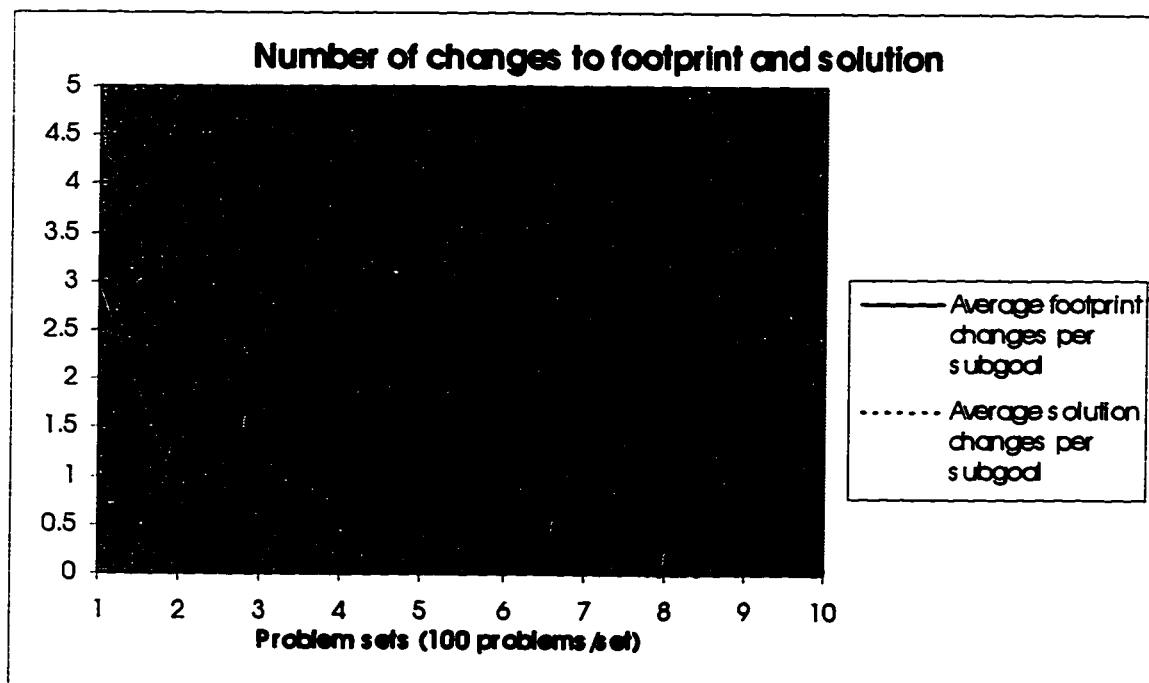
C.1.1 - Accumulation of cases for multiple conjunct goal statement problems.



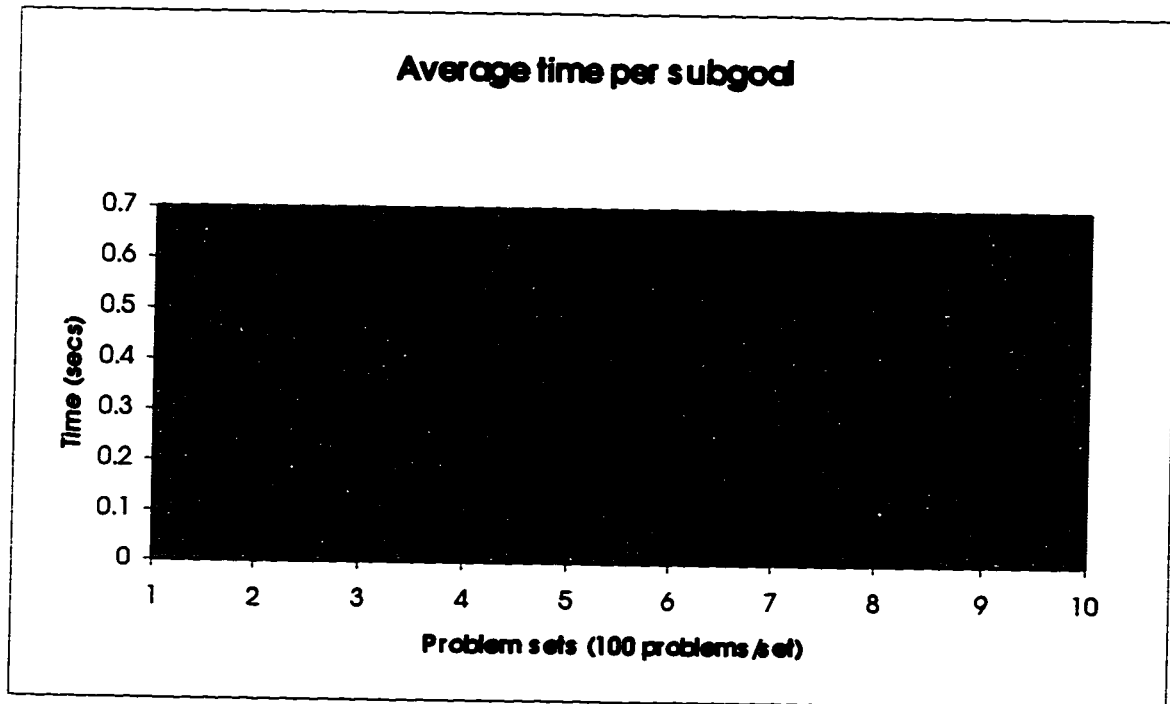
C.1.2 - Use of cases to solve goal statement conjuncts.



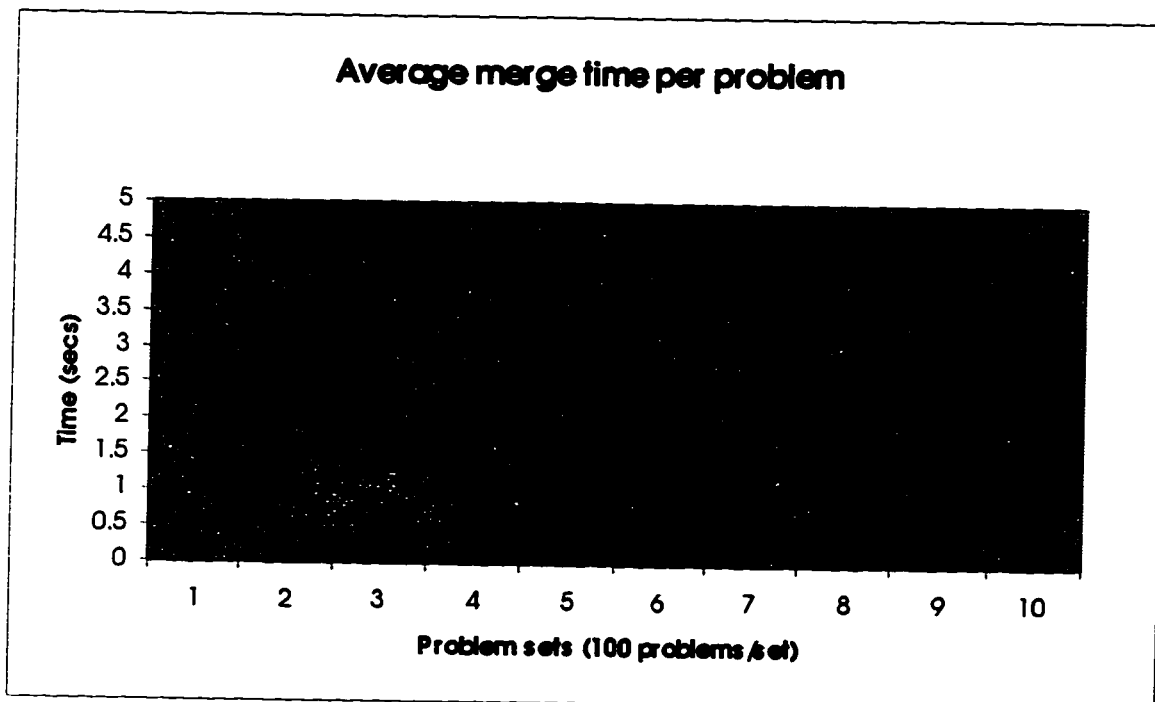
C.1.3 - Adaptation effort.



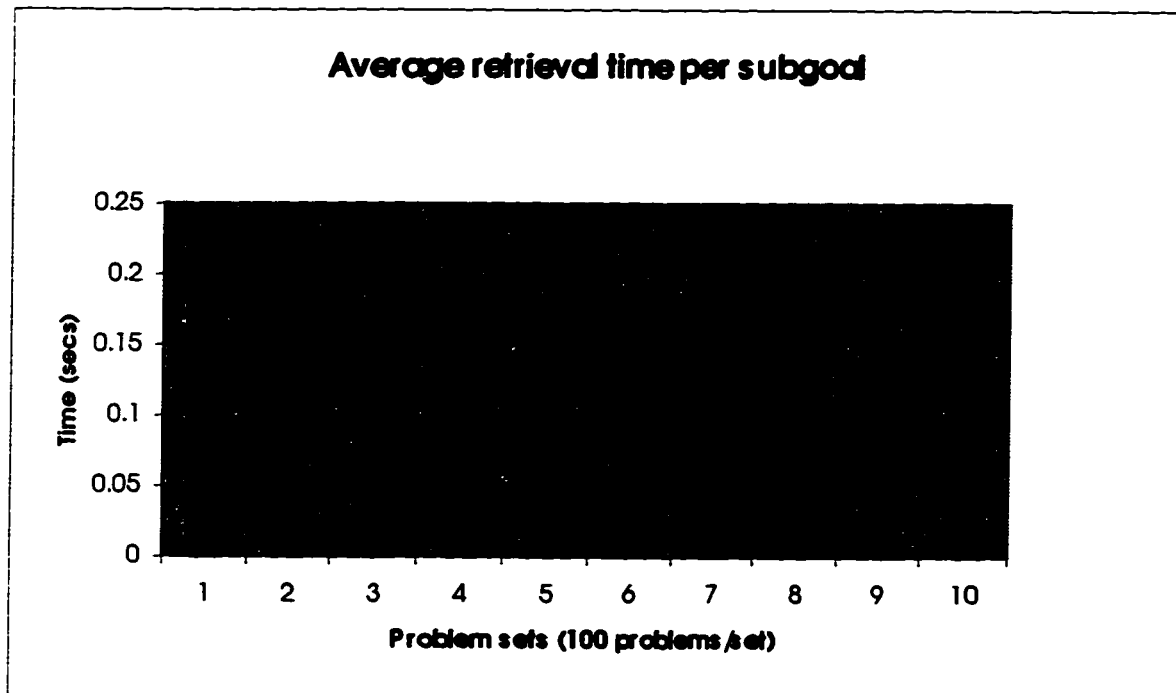
C.1.4 - Adaptation effort.



C.1.5 - Average time to solve one conjunct.

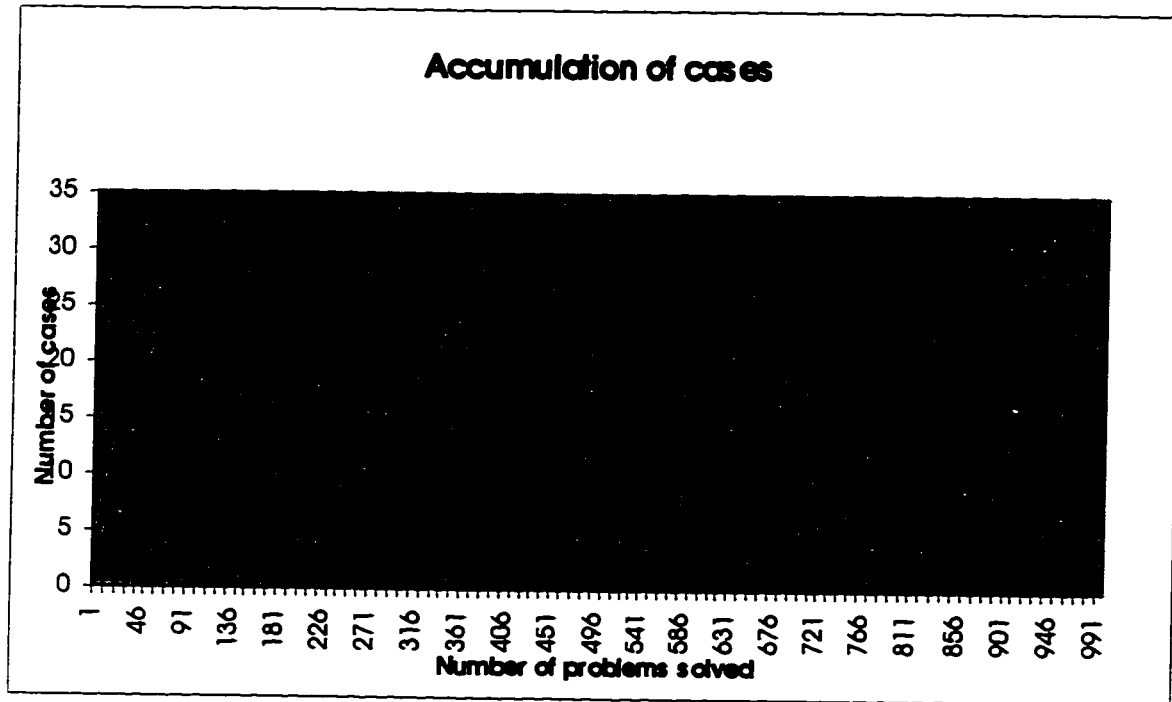


C.1.6 - Average merge time per problem.

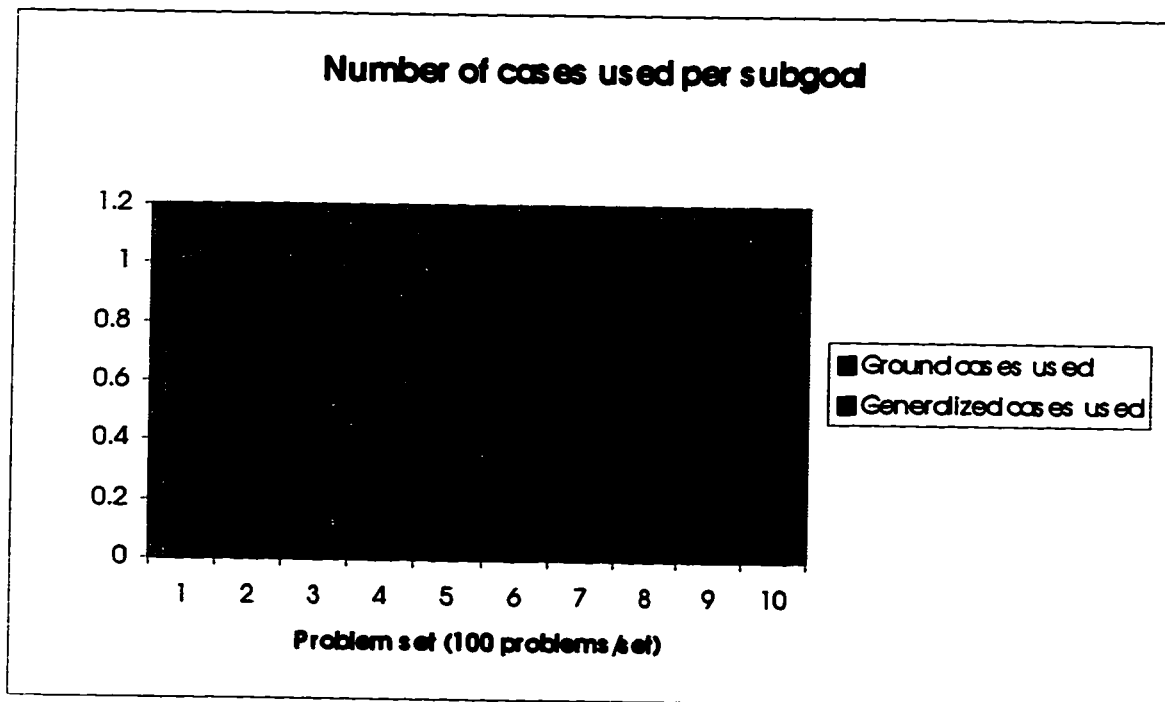


C.1.7 - Average retrieval time per goal statement conjunct.

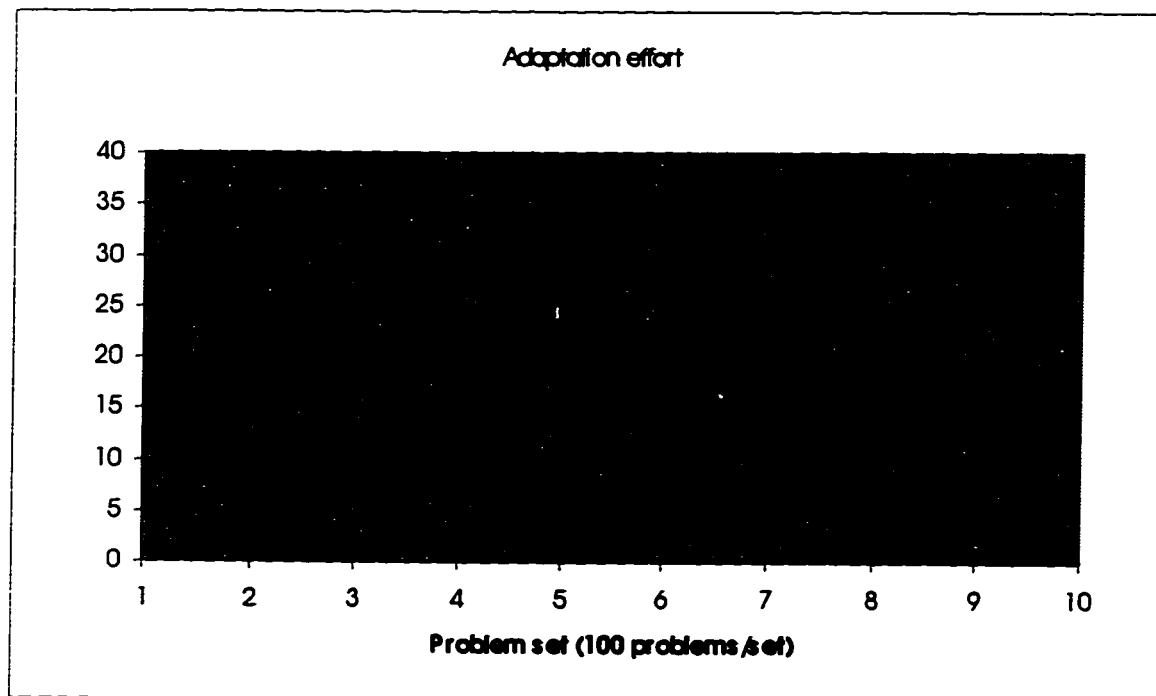
Problem set 2



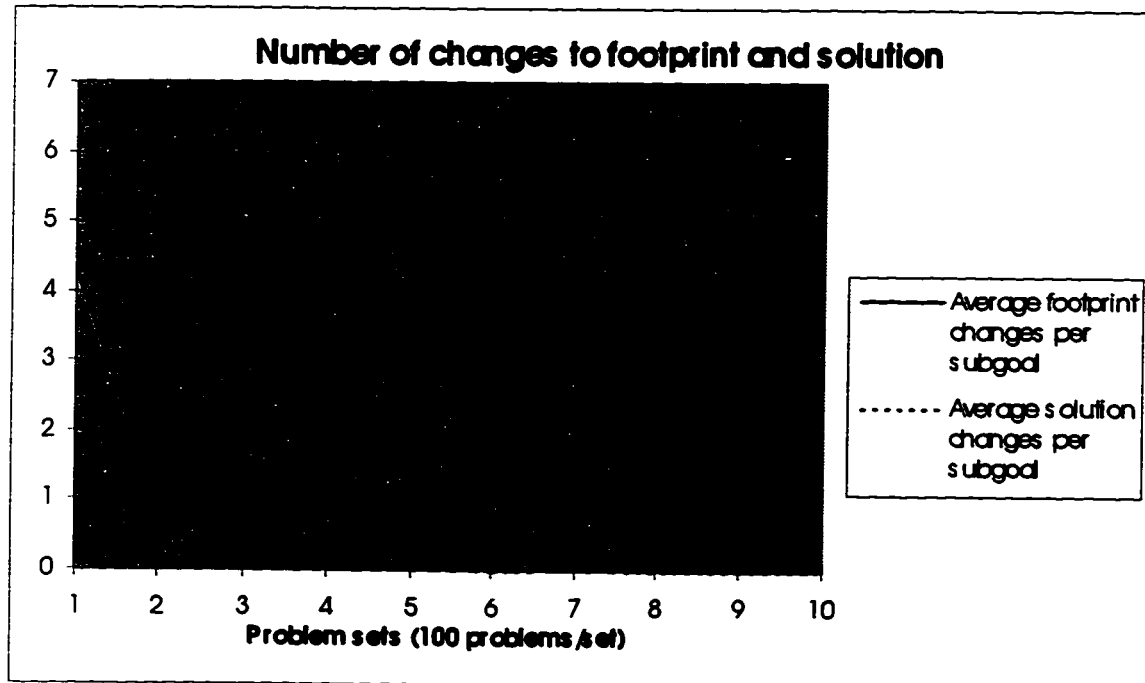
C.2.1 - Accumulation of cases for multiple conjunct goal statement problems.



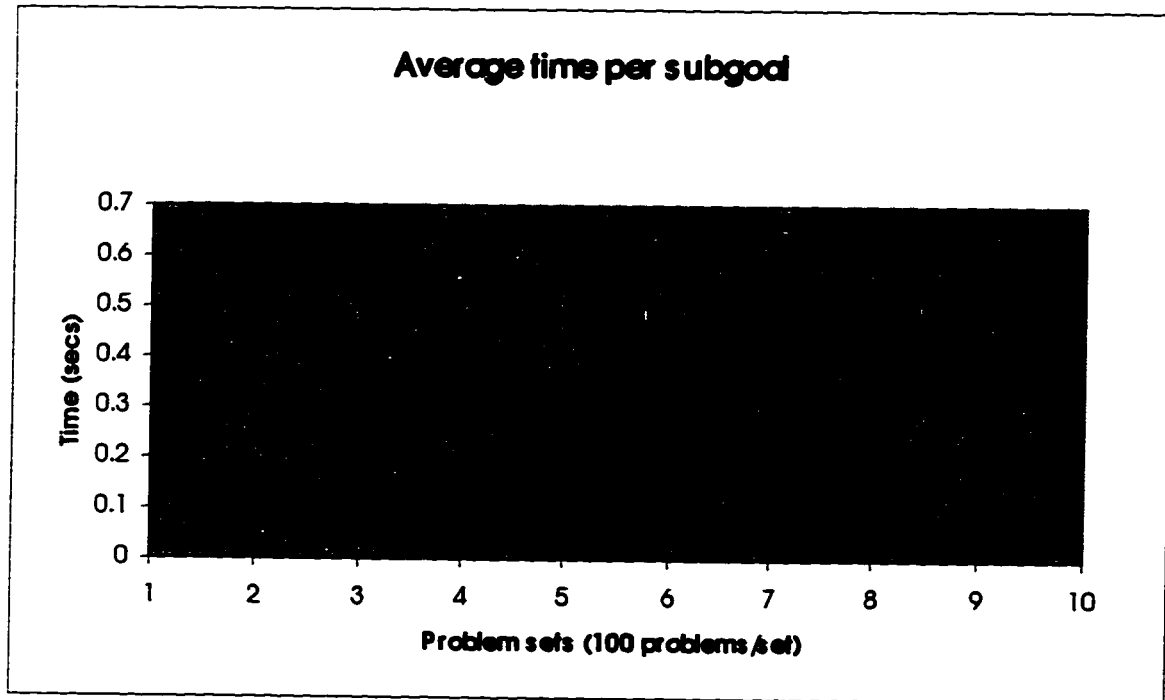
C.2.2 - Use of cases to solve goal statement conjuncts.



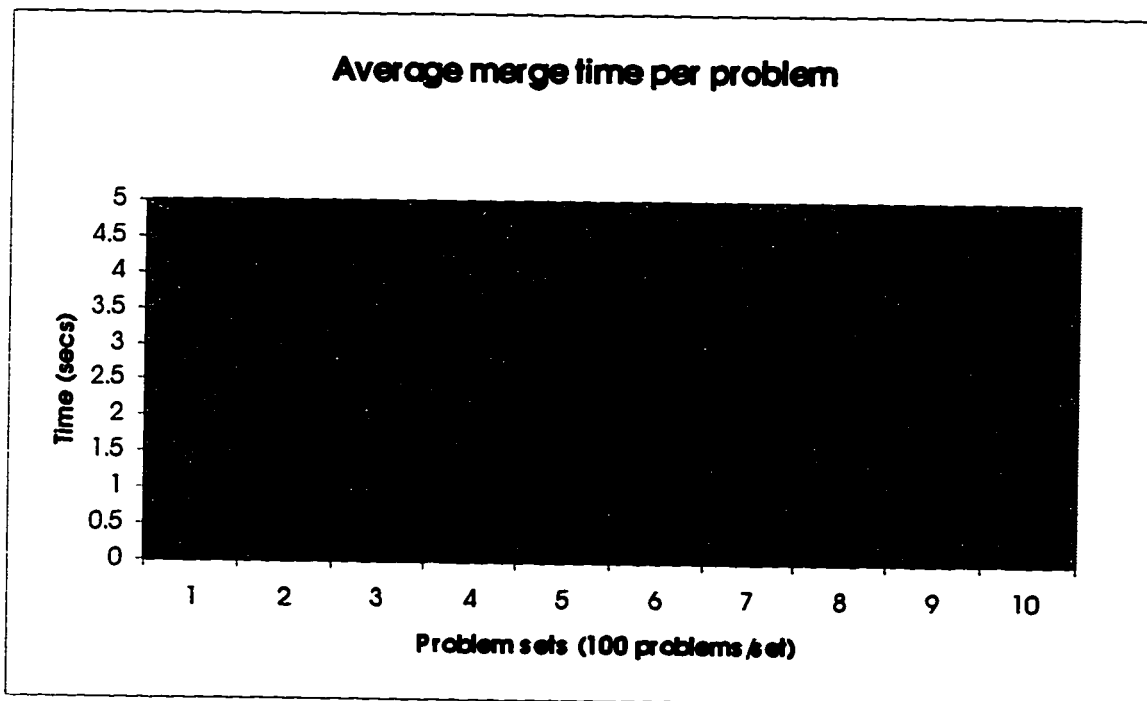
C.2.3 - Adaptation effort.



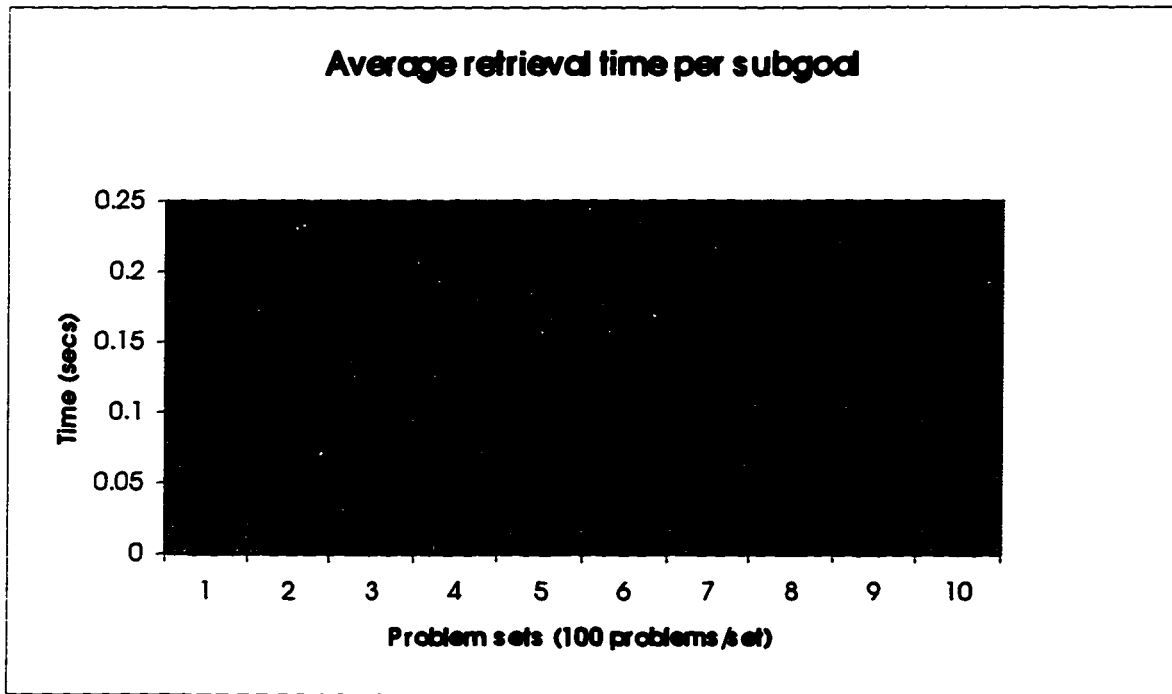
C.2.4 - Adaptation effort.



C.2.5 - Average time to solve one conjunct.

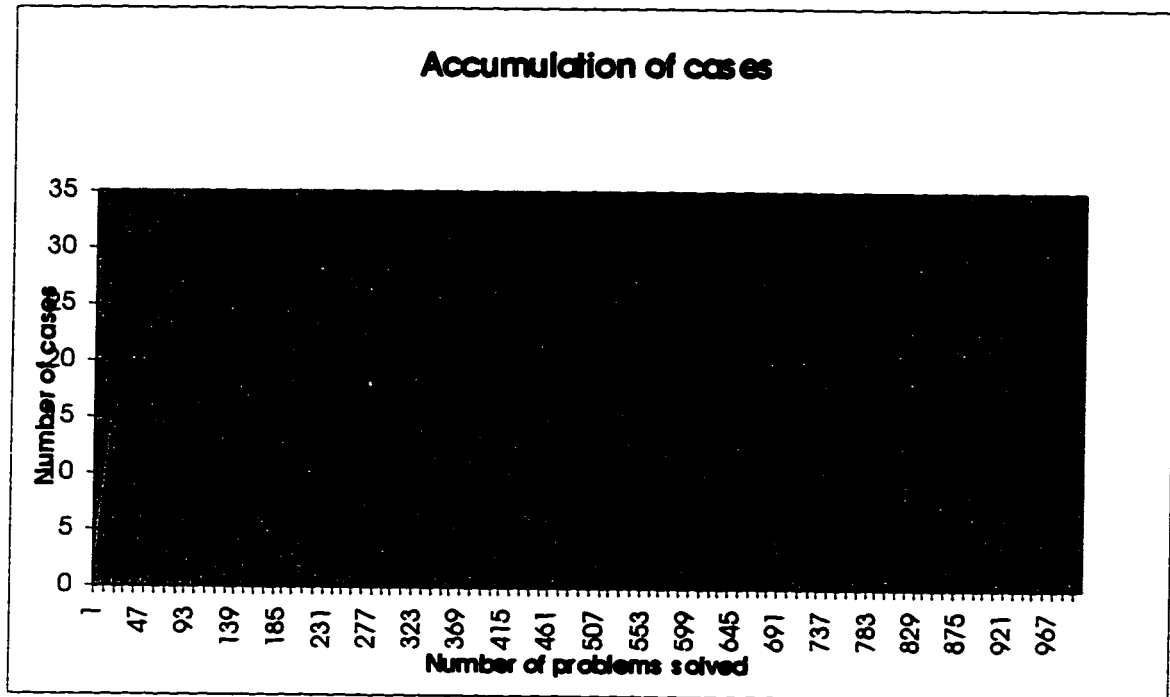


C.2.6 - Average merge time per problem.

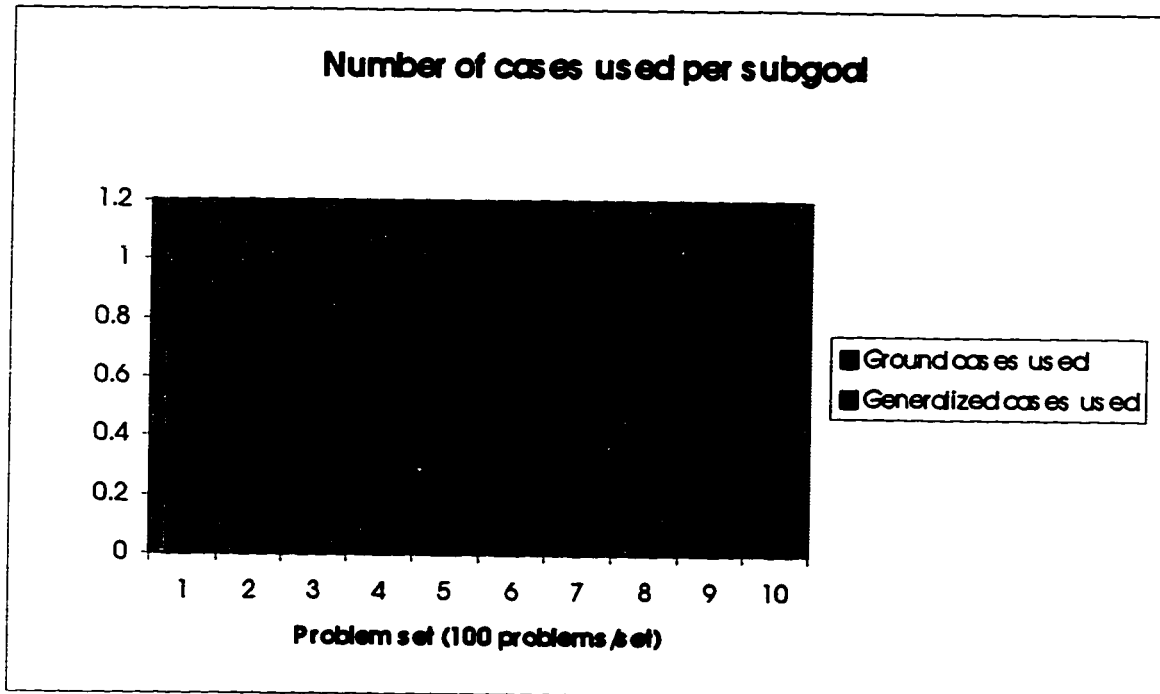


C.2.7 - Average retrieval time per goal statement conjunct.

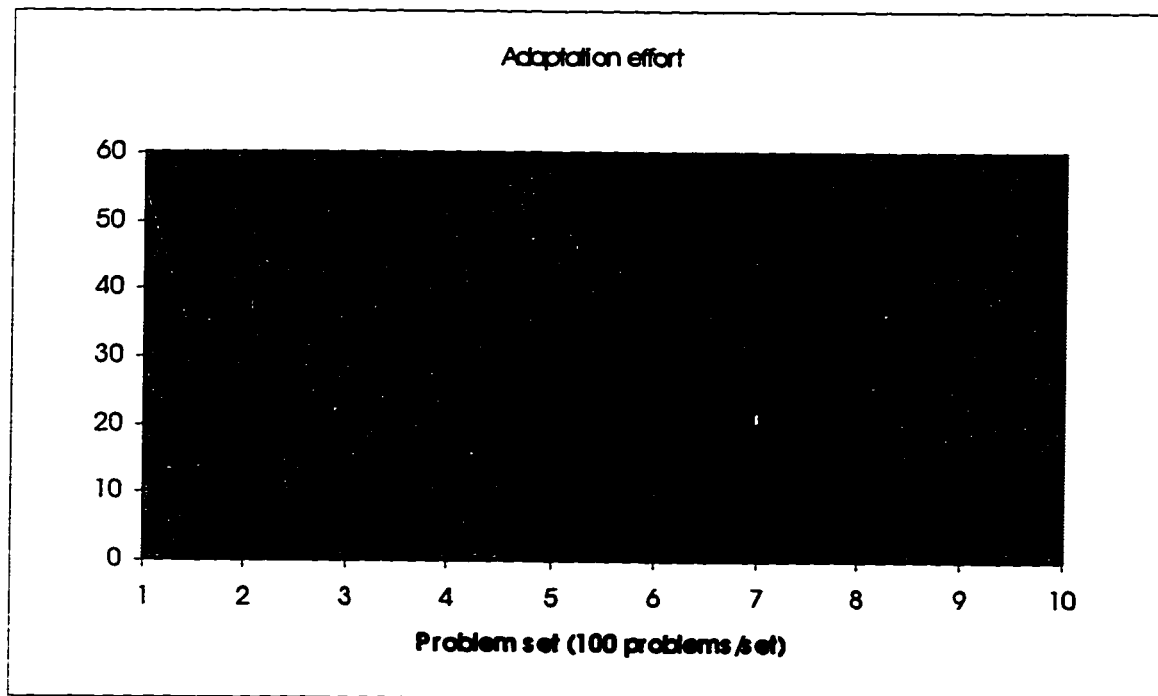
Problem set 3



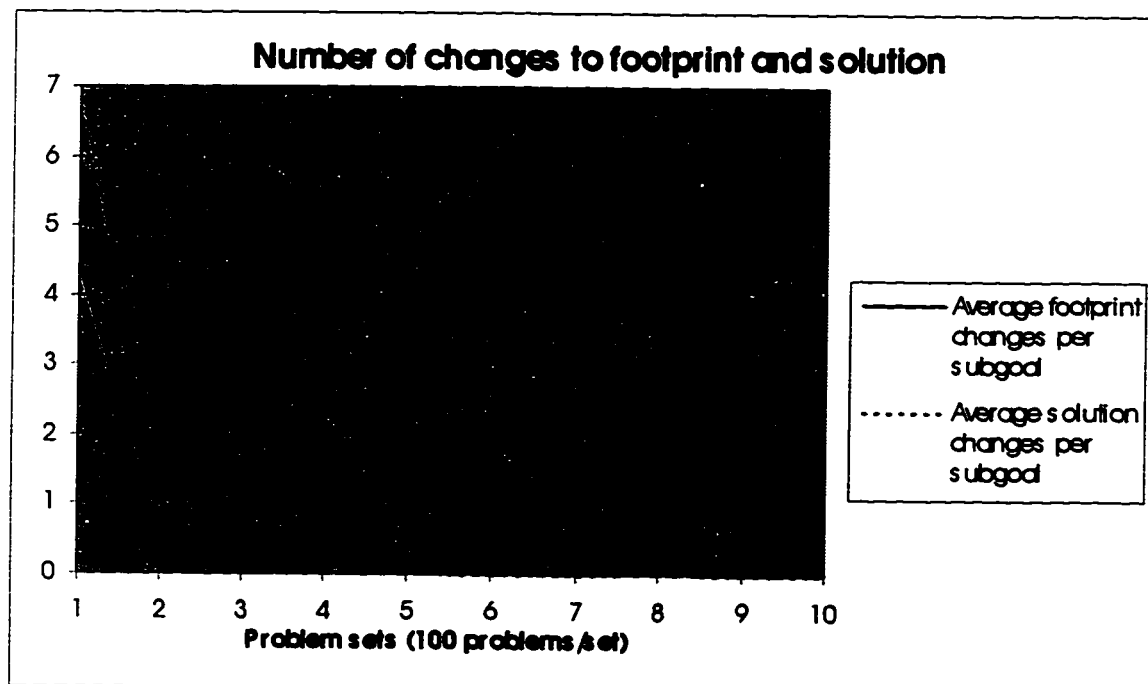
C.3.1 - Accumulation of cases for multiple conjunct goal statement problems.



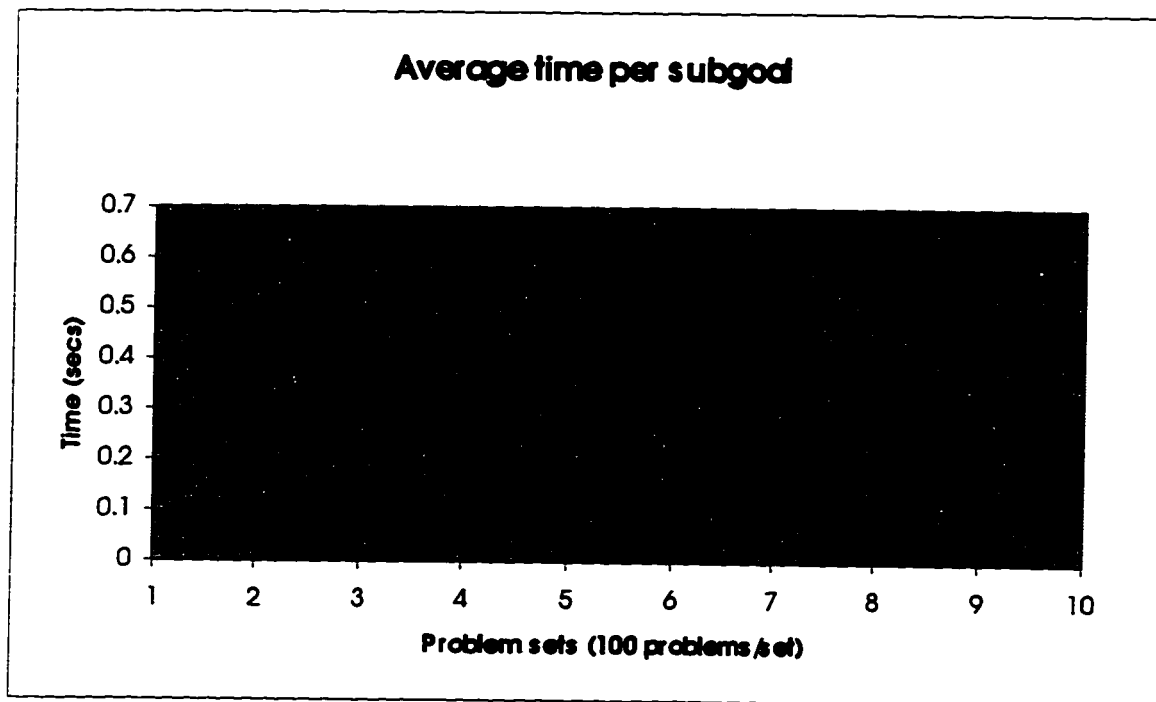
C.3.2 - Use of cases to solve goal statement conjuncts.



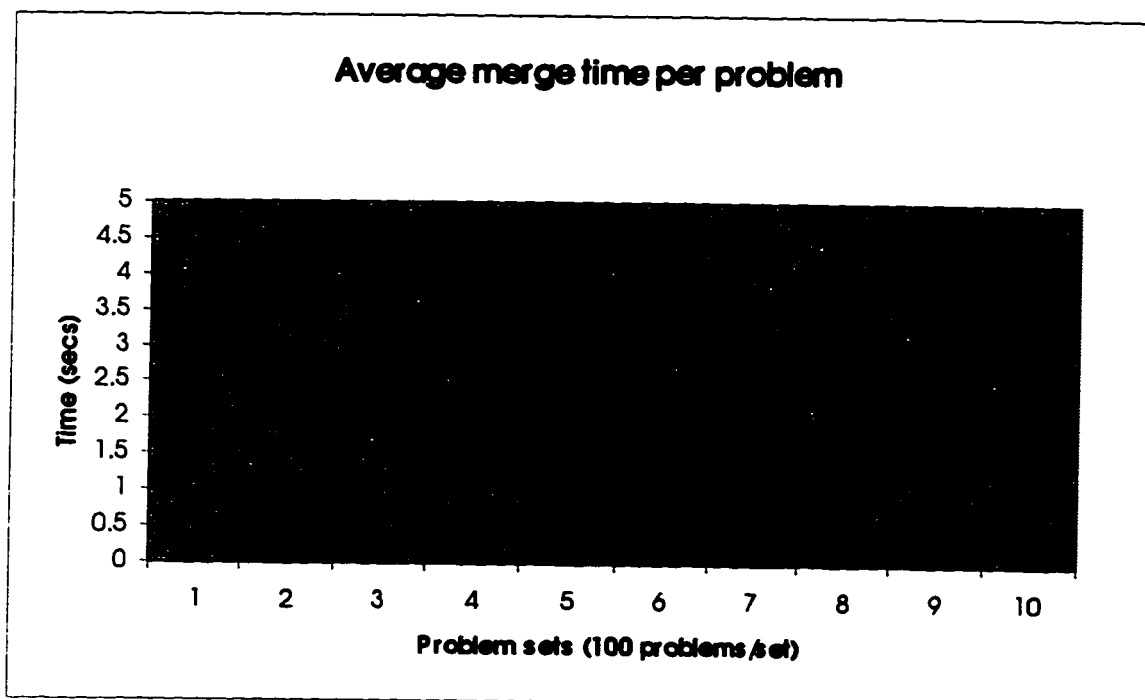
C.3.3 - Adaptation effort.



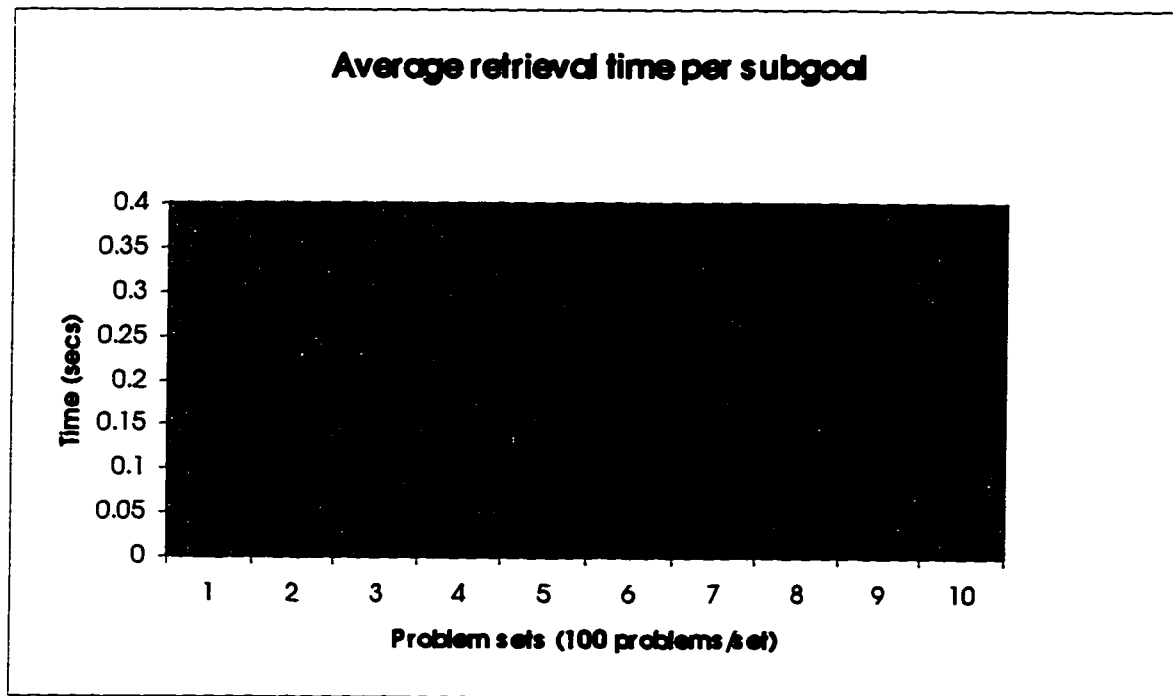
C.3.4 - Adaptation effort.



C.3.5 - Average time to solve one conjunct.

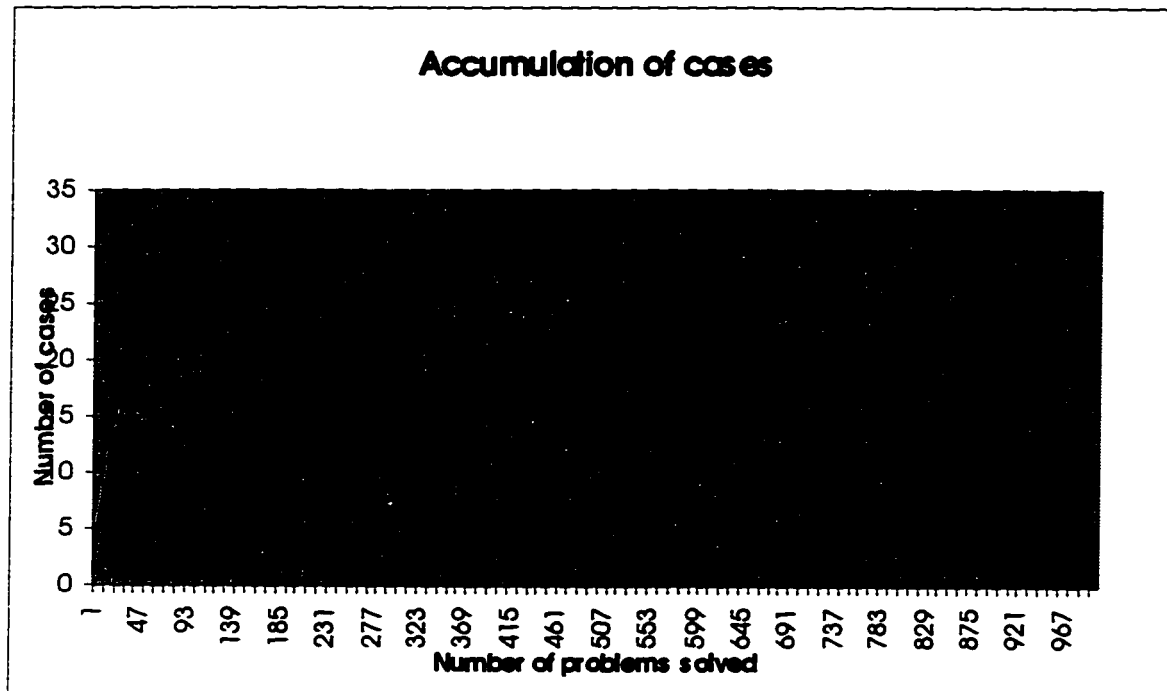


C.2.6 - Average merge time per problem.

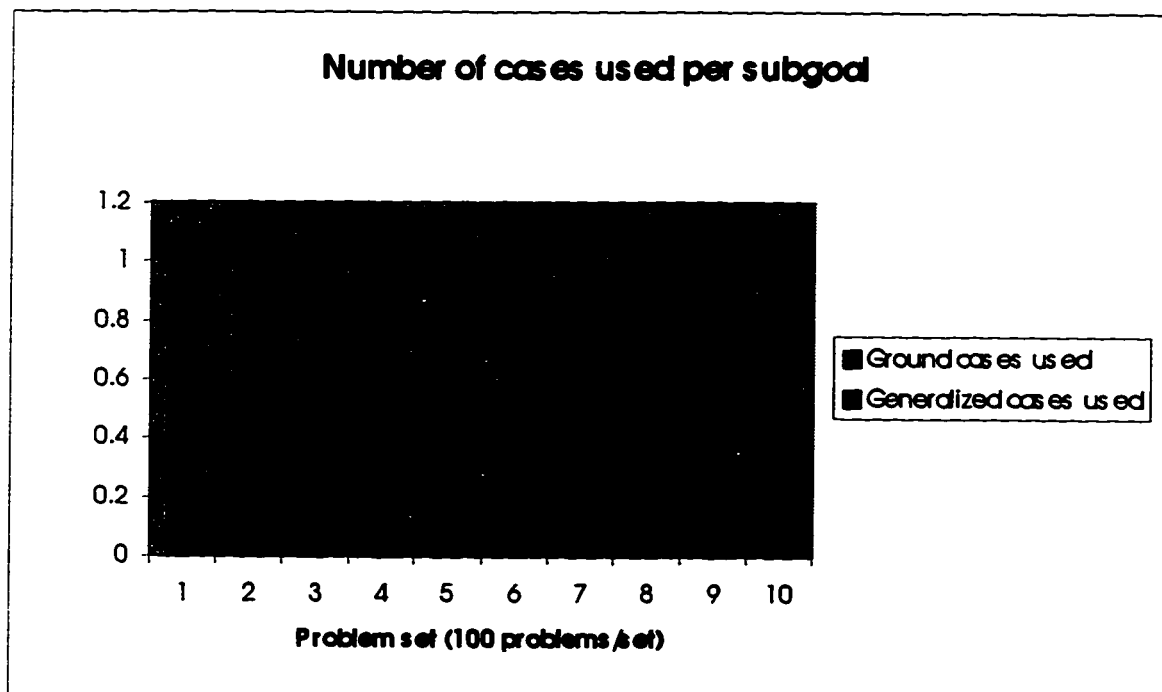


C.3.7 - Average retrieval time per goal statement conjunct.

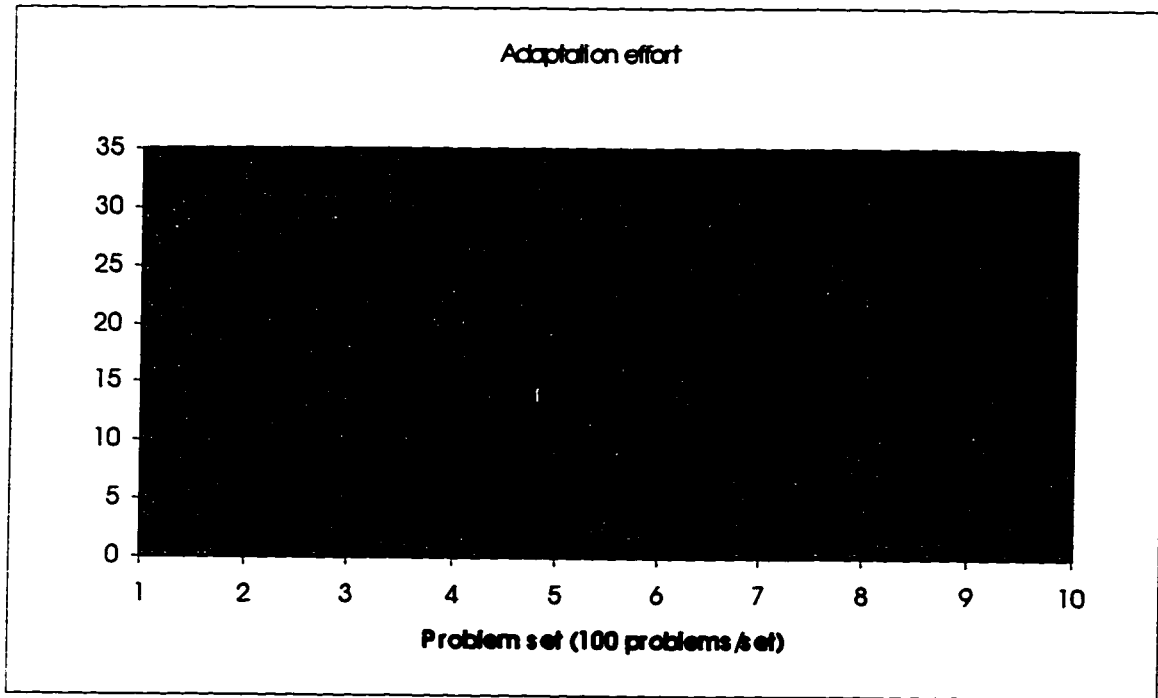
Problem set 4



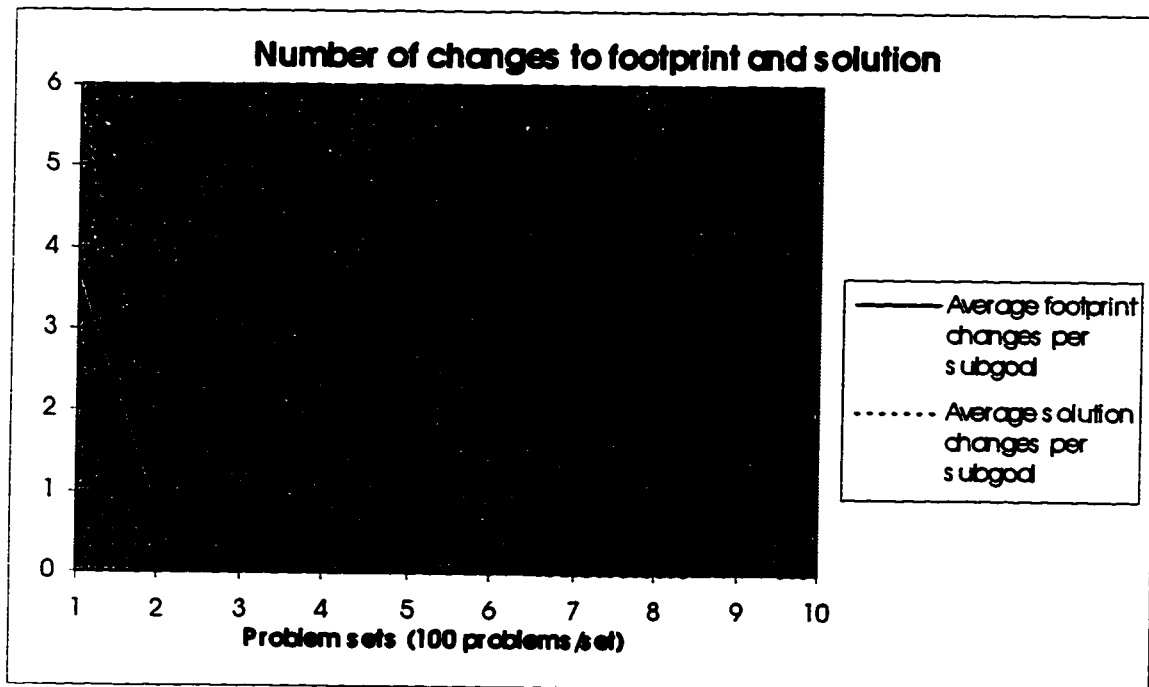
C.4.1 - Accumulation of cases for multiple conjunct goal statement problems.



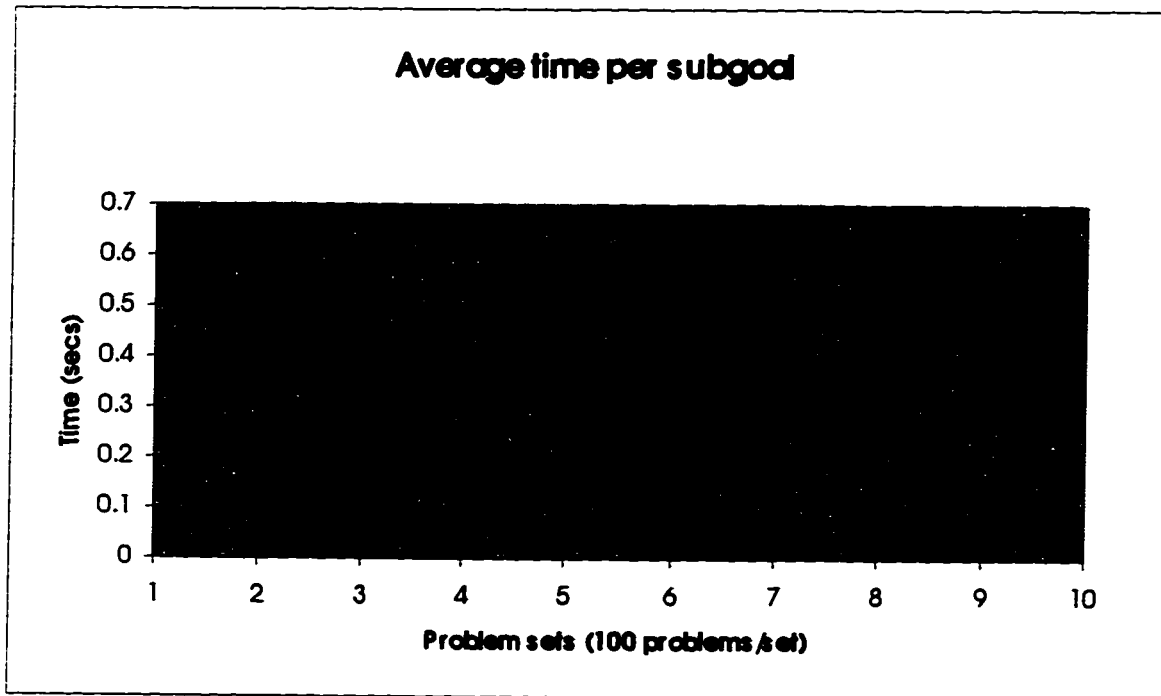
C.4.2 - Use of cases to solve goal statement conjuncts.



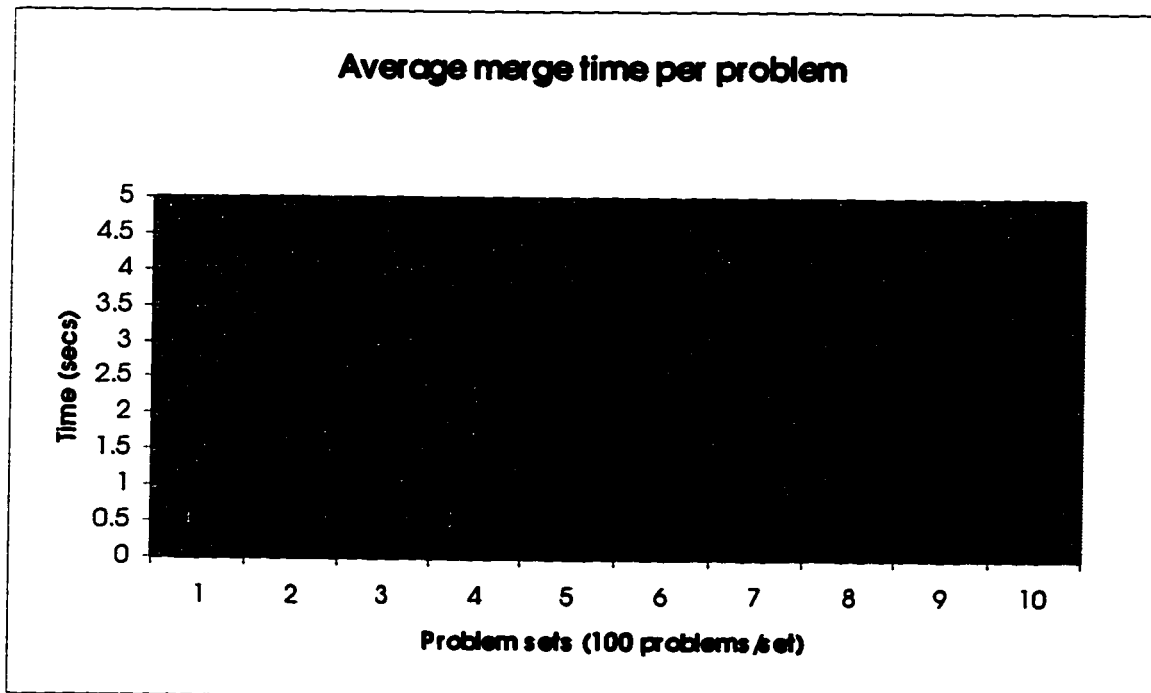
C.4.3 - Adaptation effort.



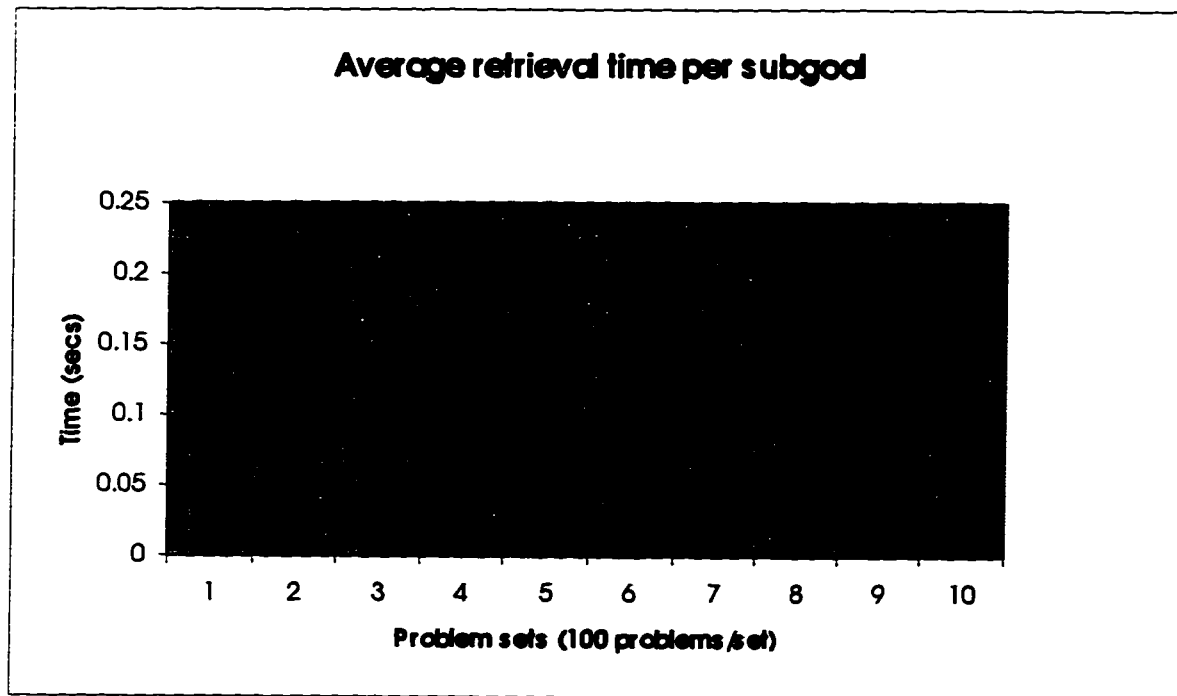
C.4.4 - Adaptation effort.



C.4.5 - Average time to solve one conjunct.

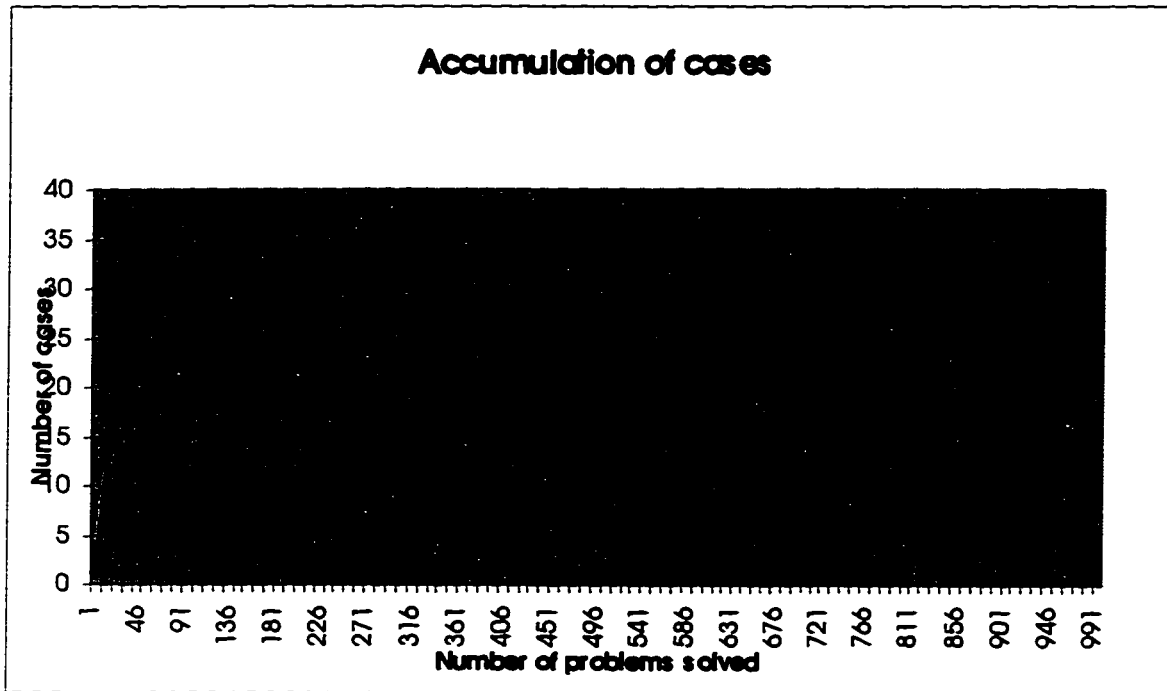


C.4.6 - Average merge time per problem.

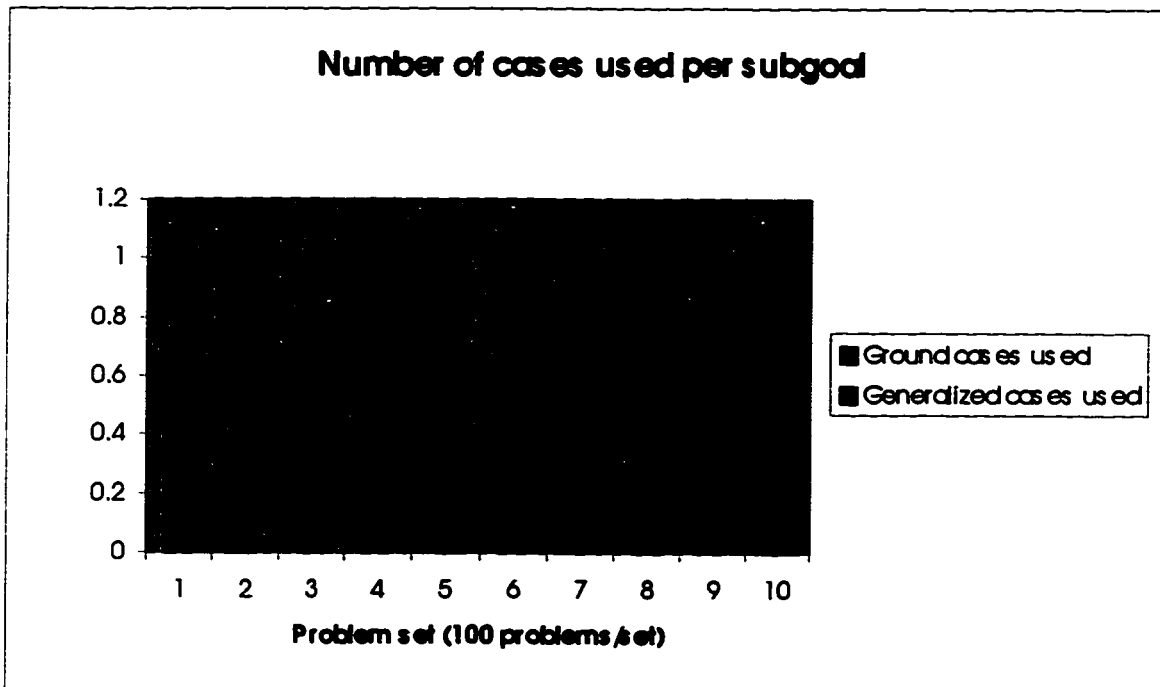


C.4.7 - Average retrieval time per goal statement conjunct.

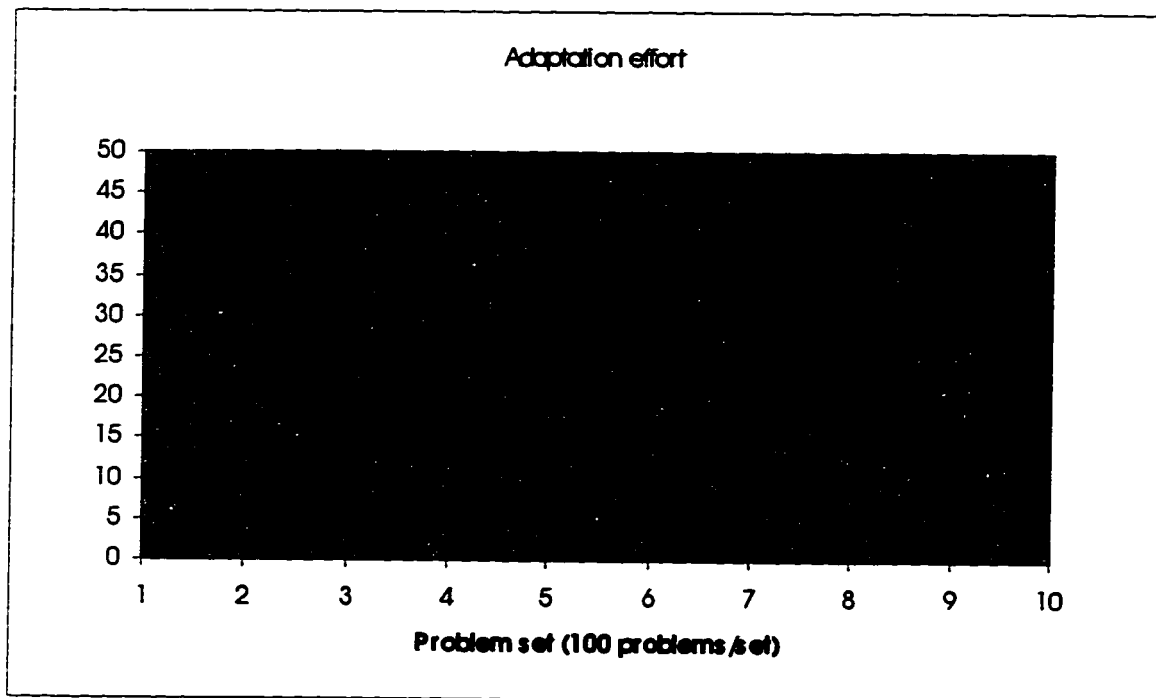
Problem set 5



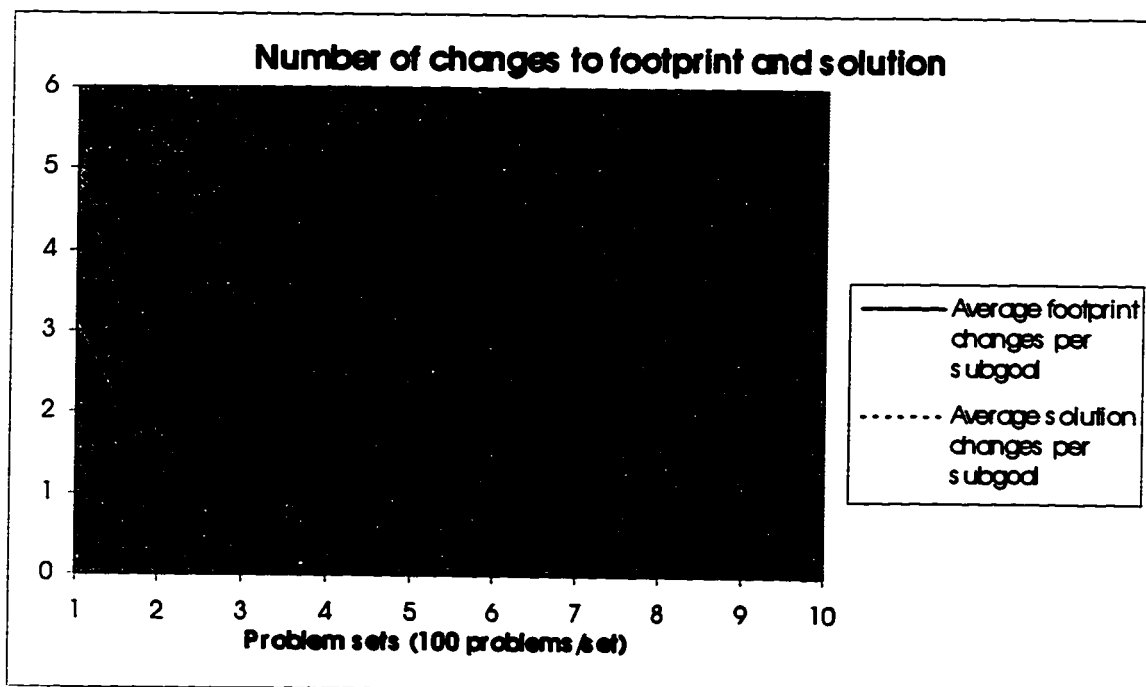
C.5.1 - Accumulation of cases for multiple conjunct goal statement problems.



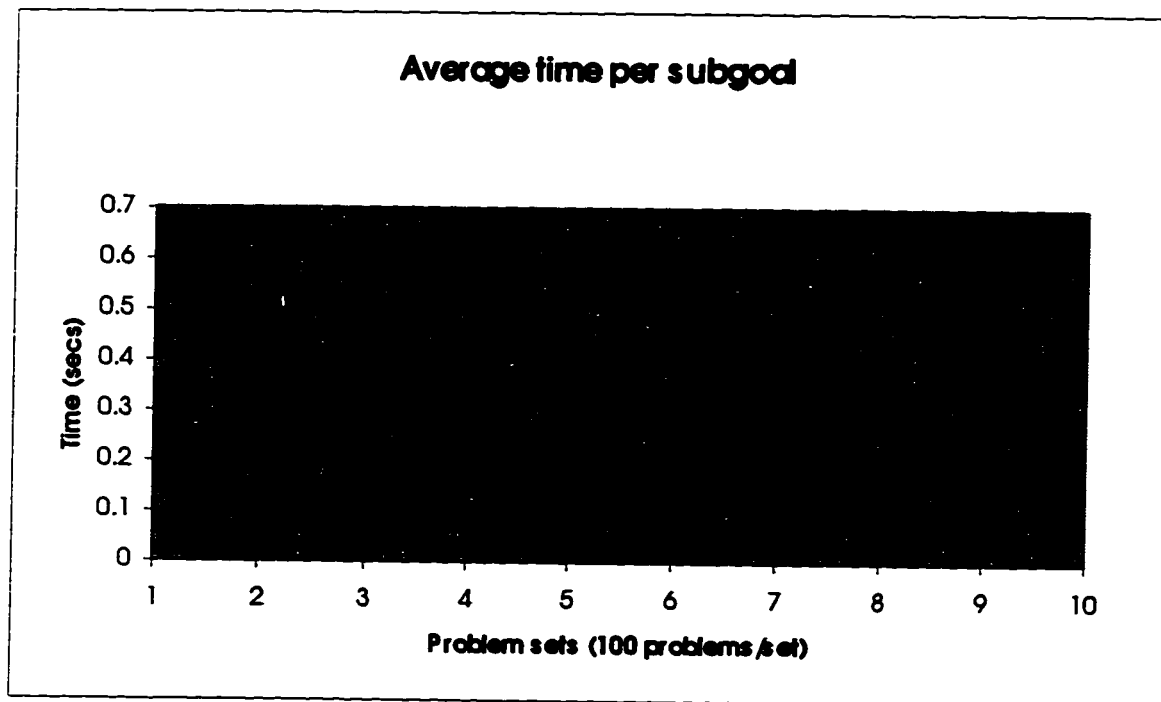
C.5.2 - Use of cases to solve goal statement conjuncts.



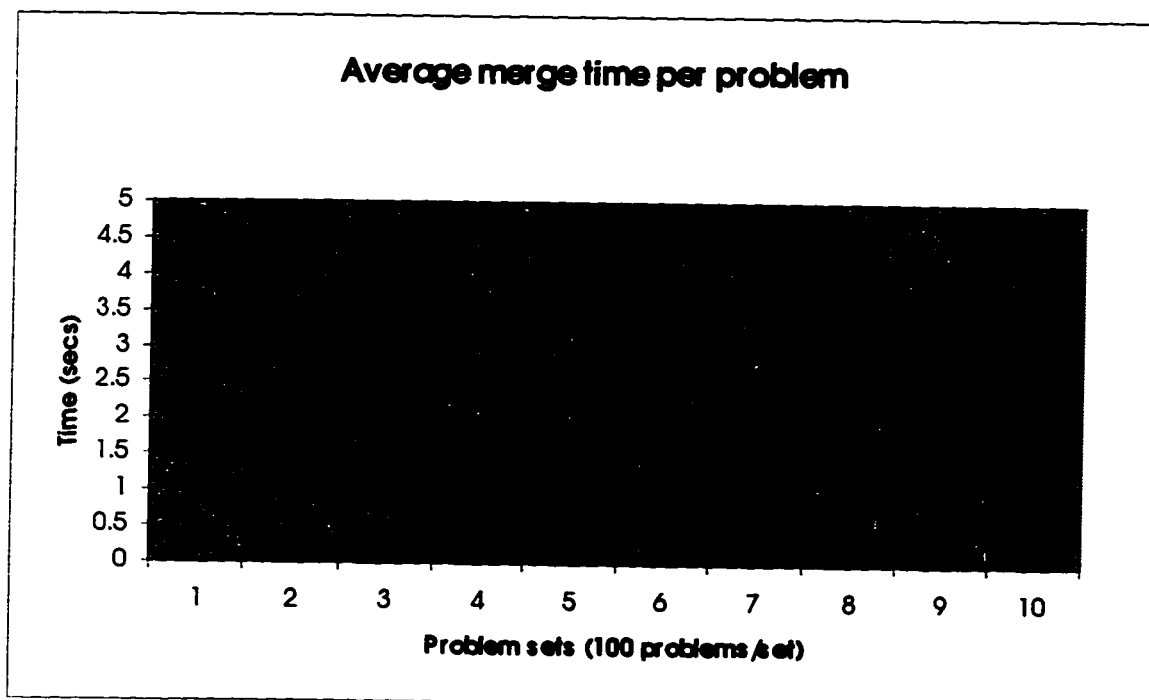
C.5.3 - Adaptation effort.



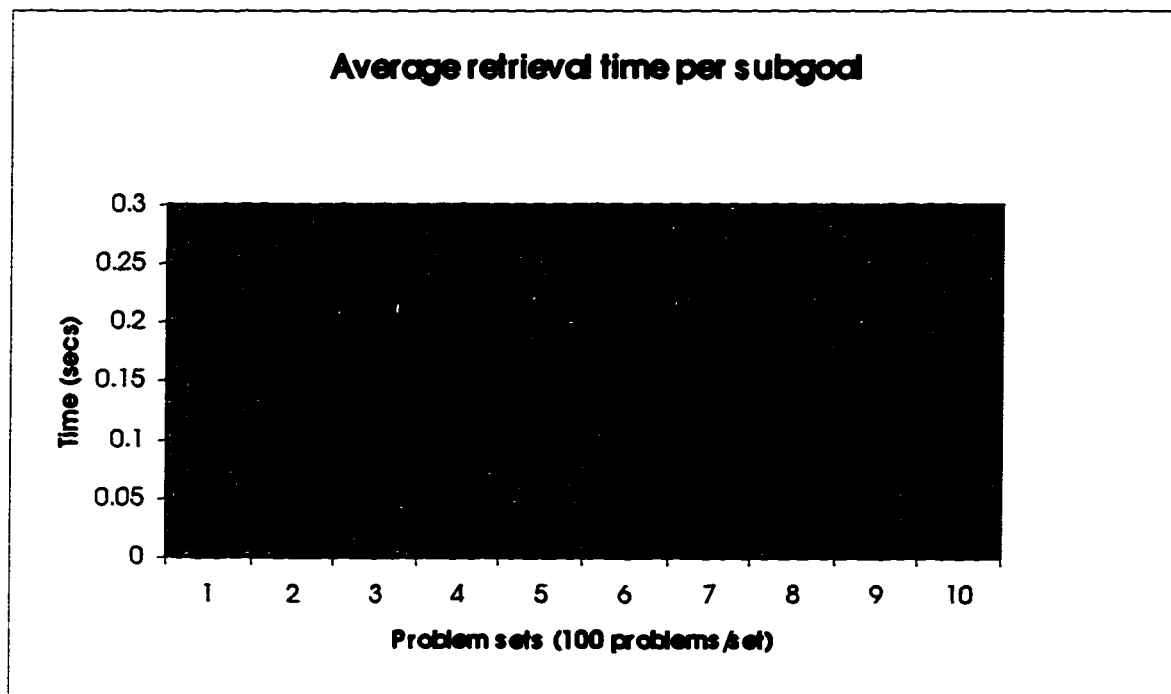
C.5.4 - Adaptation effort.



C.5.5 - Average time to solve one conjunct.

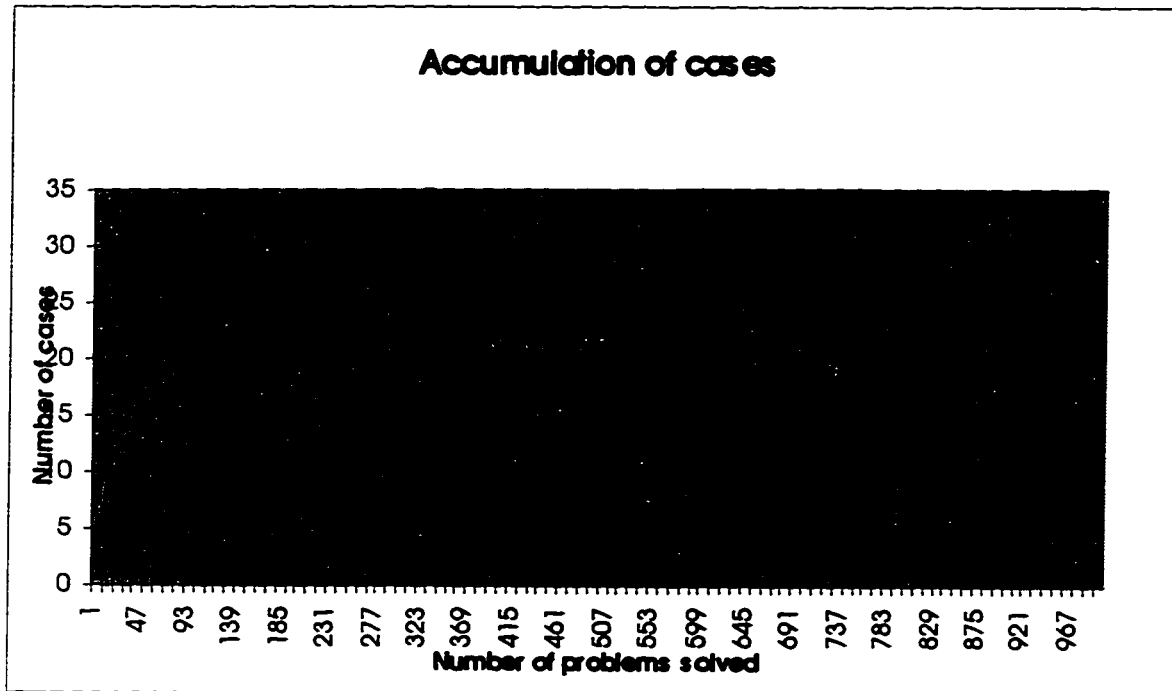


C.5.6 - Average merge time per problem.

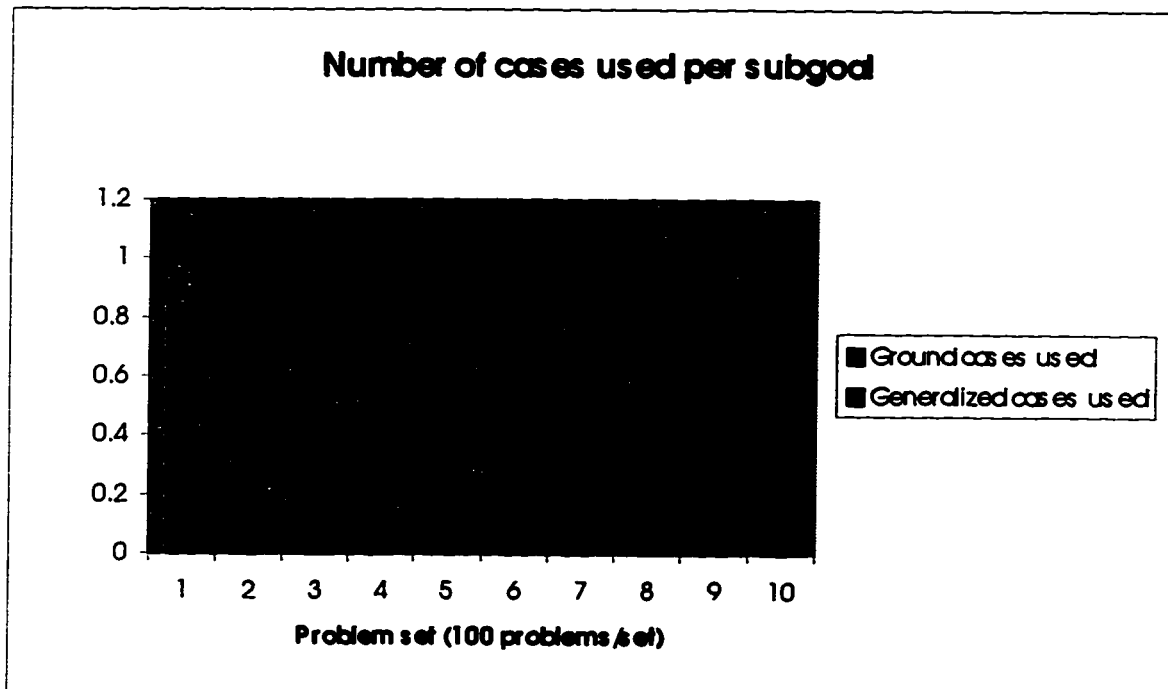


C.5.7 - Average retrieval time per goal statement conjunct.

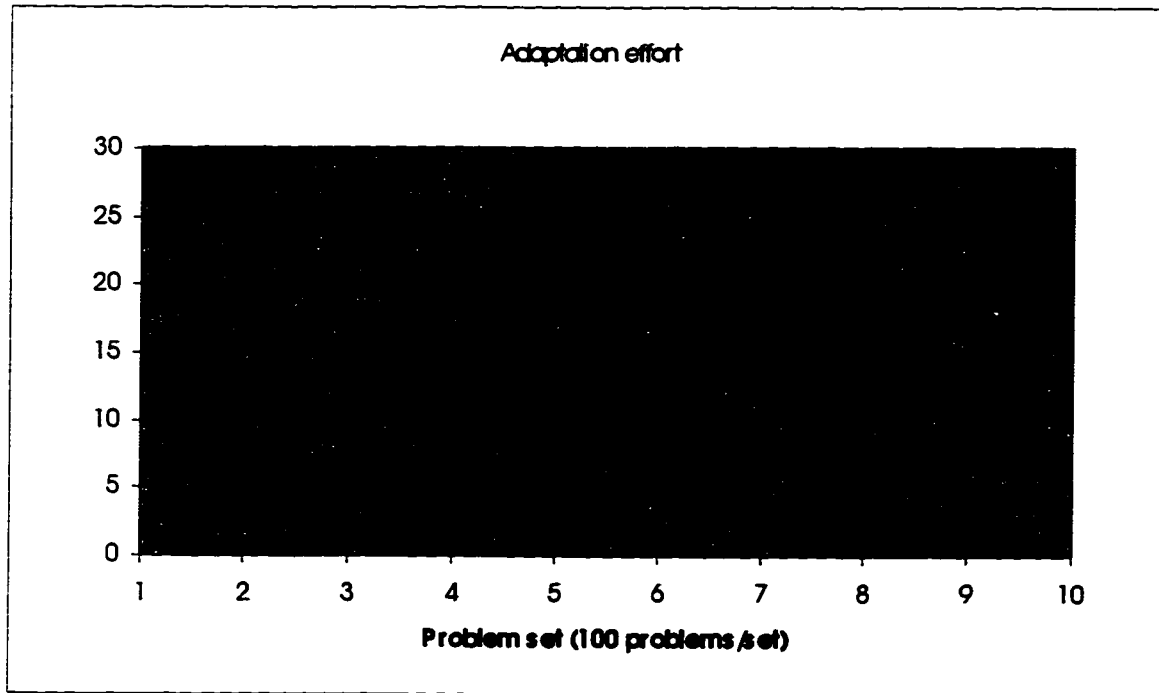
Problem set 6



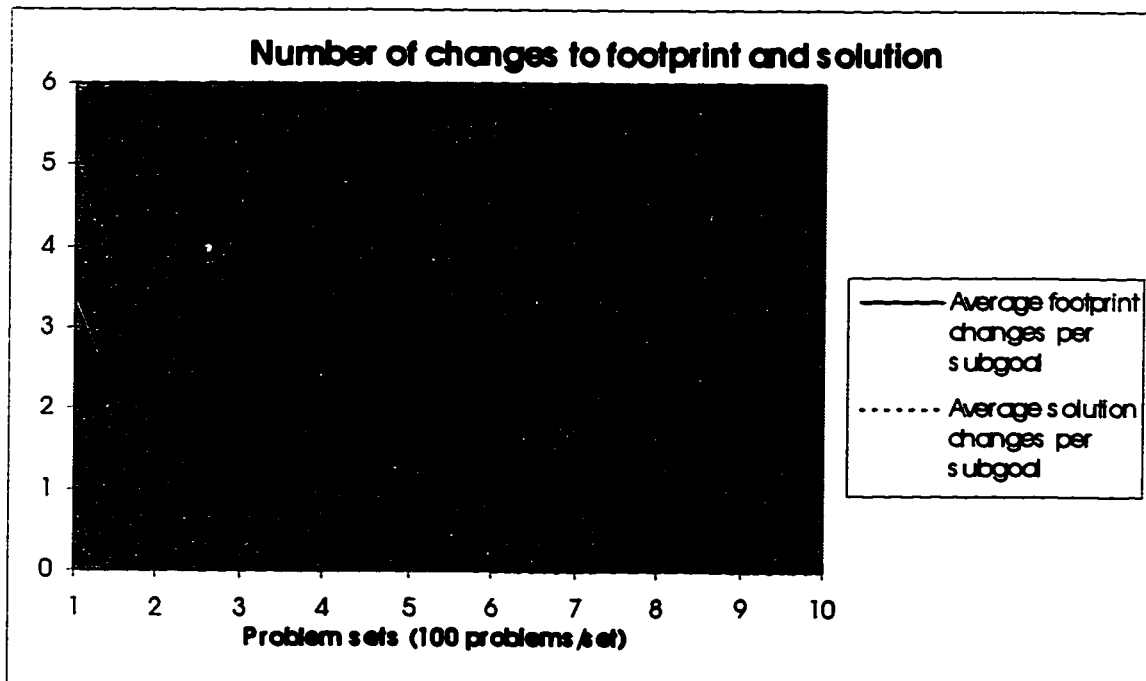
C.6.1 - Accumulation of cases for multiple conjunct goal statement problems.



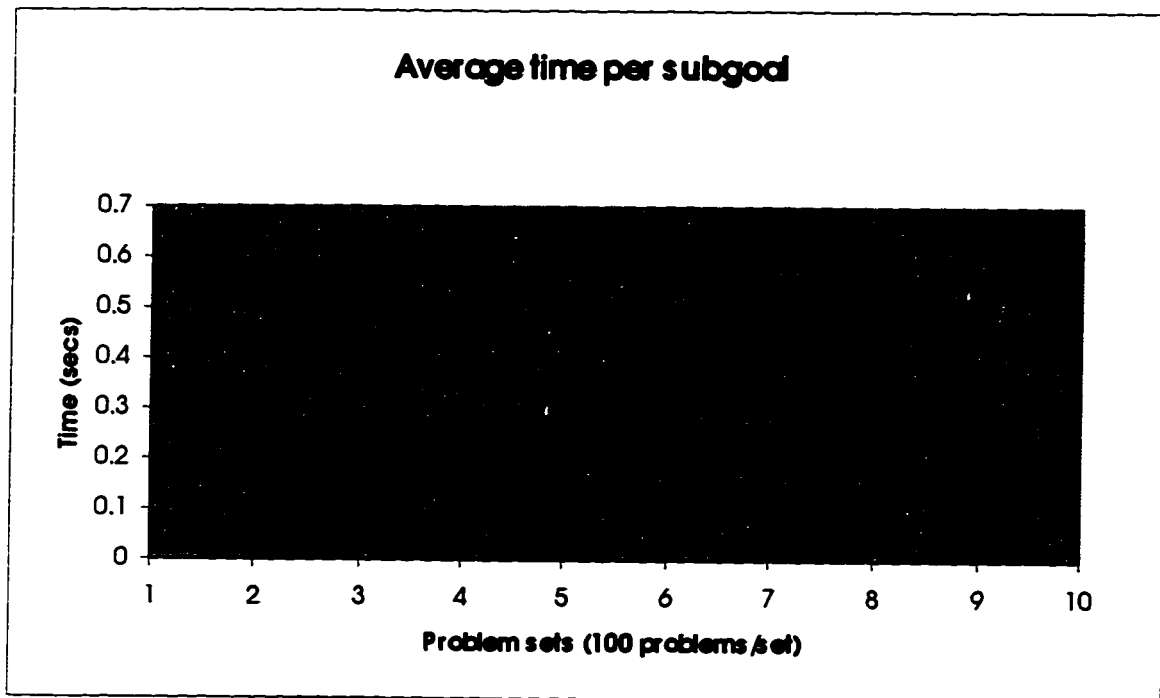
C.6.2 - Use of cases to solve goal statement conjuncts.



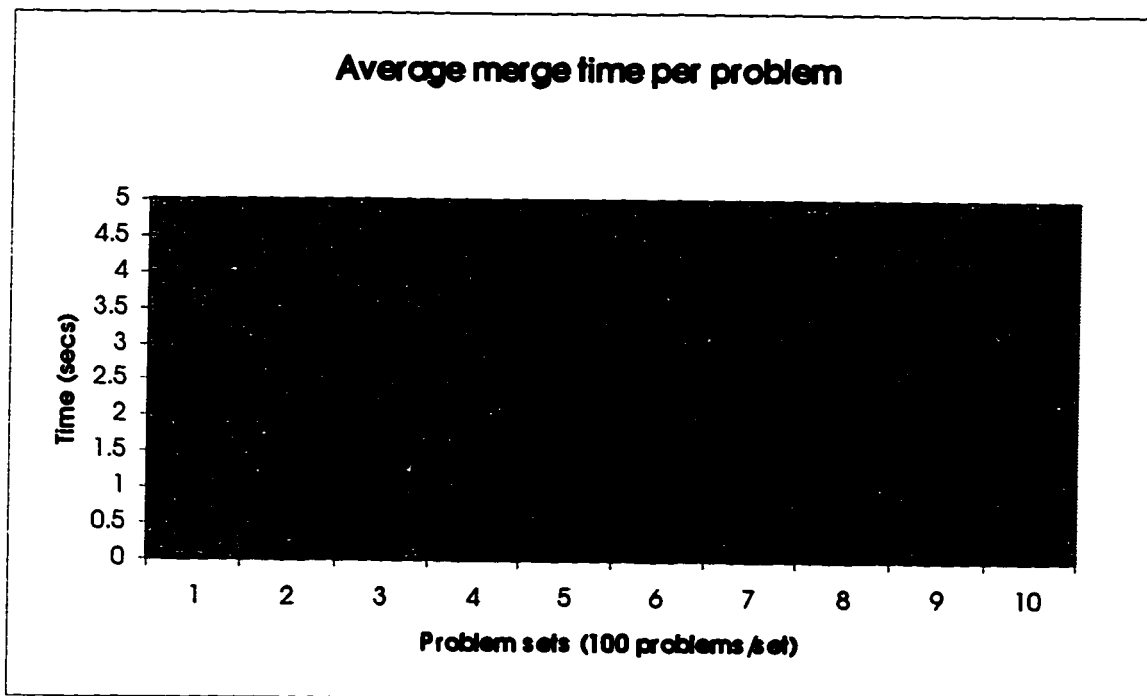
C.6.3 - Adaptation effort.



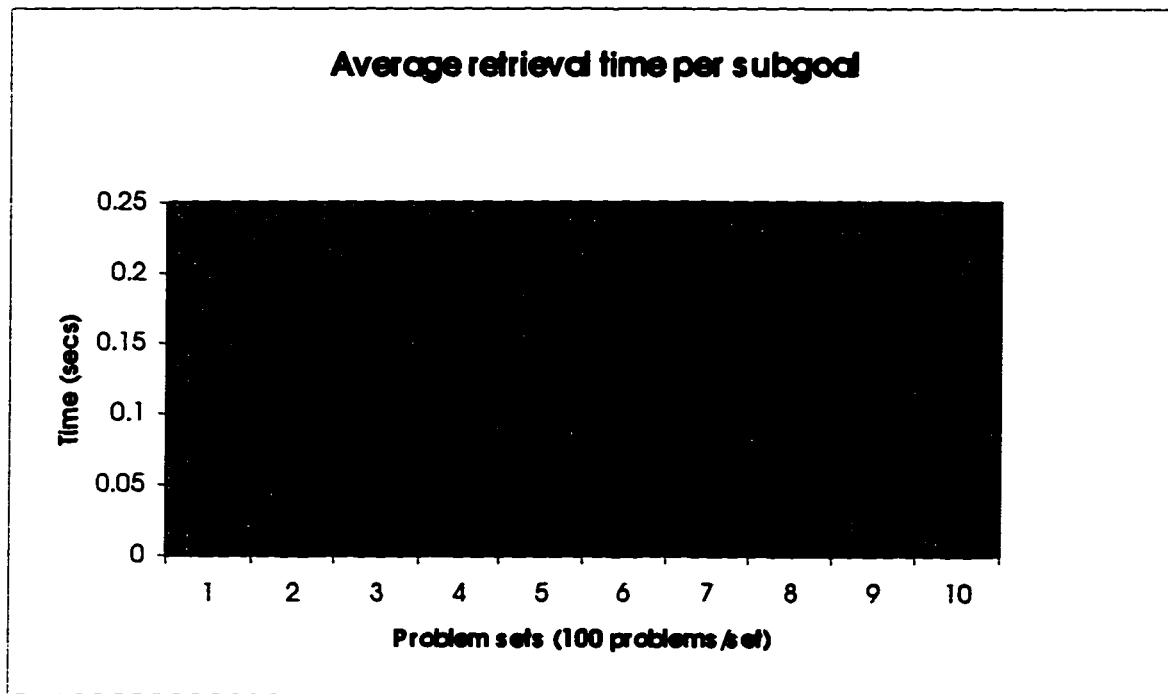
C.6.4 - Adaptation effort.



C.6.5 - Average time to solve one conjunct.

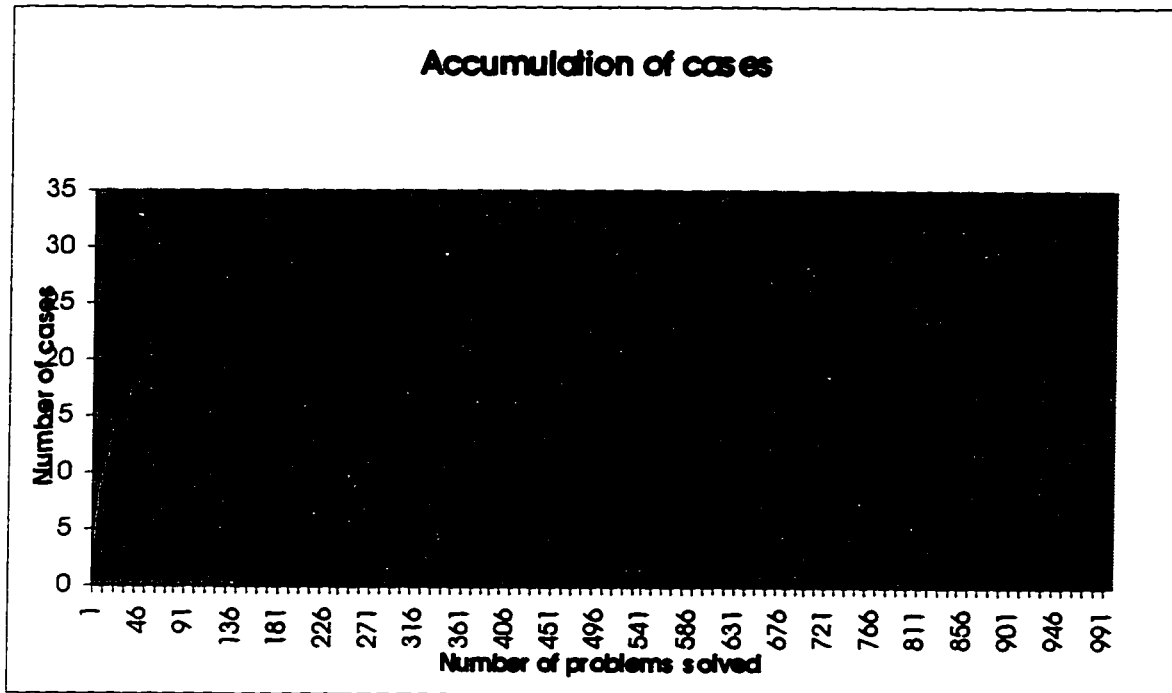


C.6.6 - Average merge time per problem.

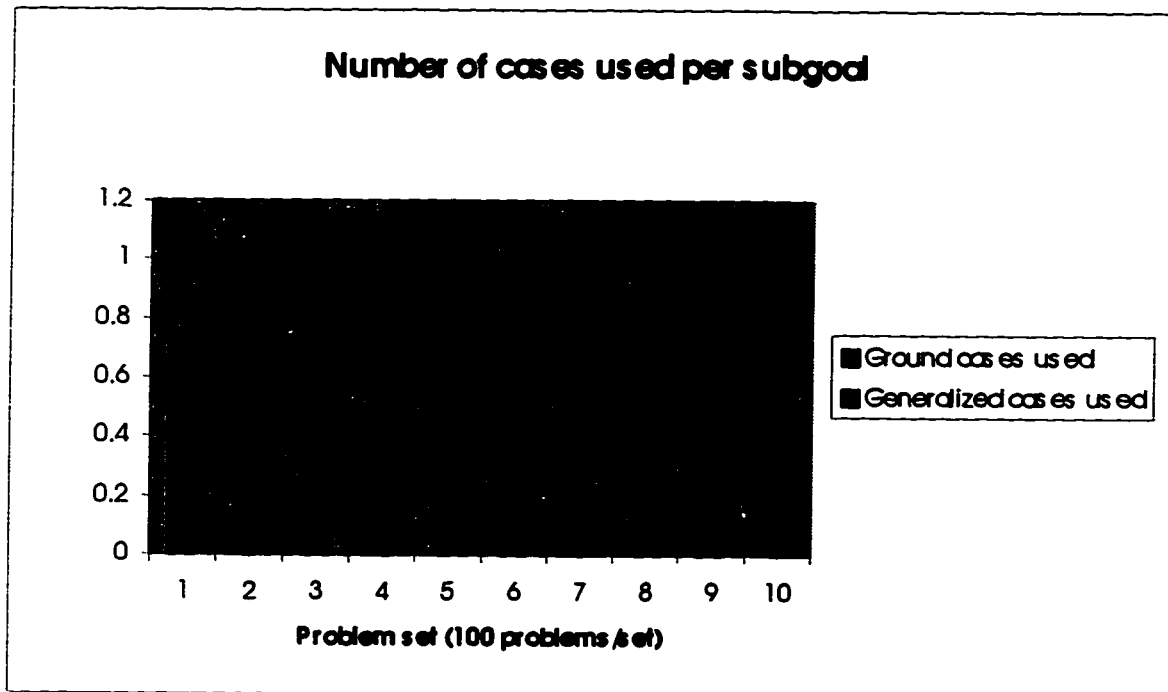


C.6.7 - Average retrieval time per goal statement conjunct.

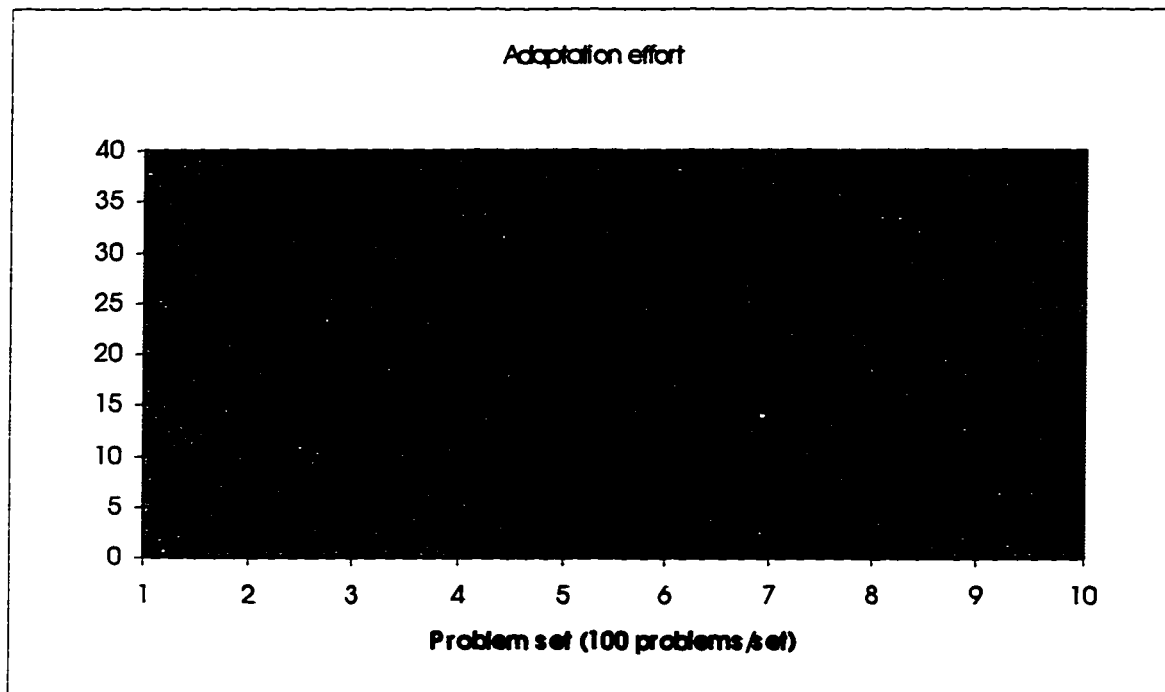
Problem set 7



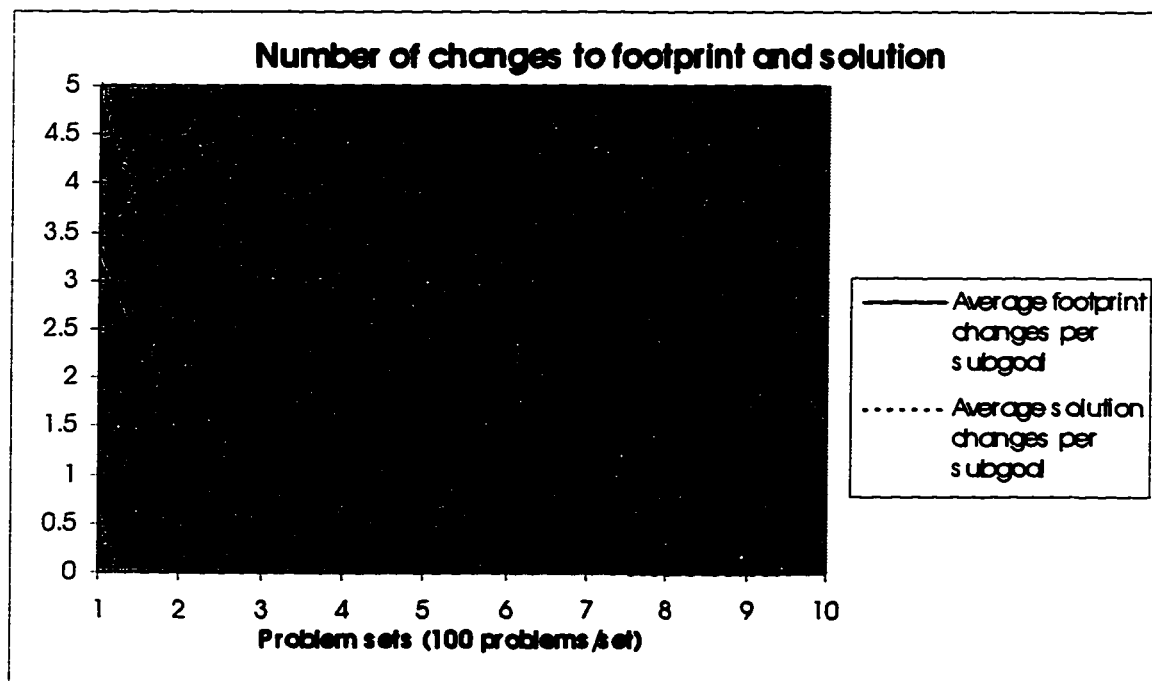
C.7.1 - Accumulation of cases for multiple conjunct goal statement problems.



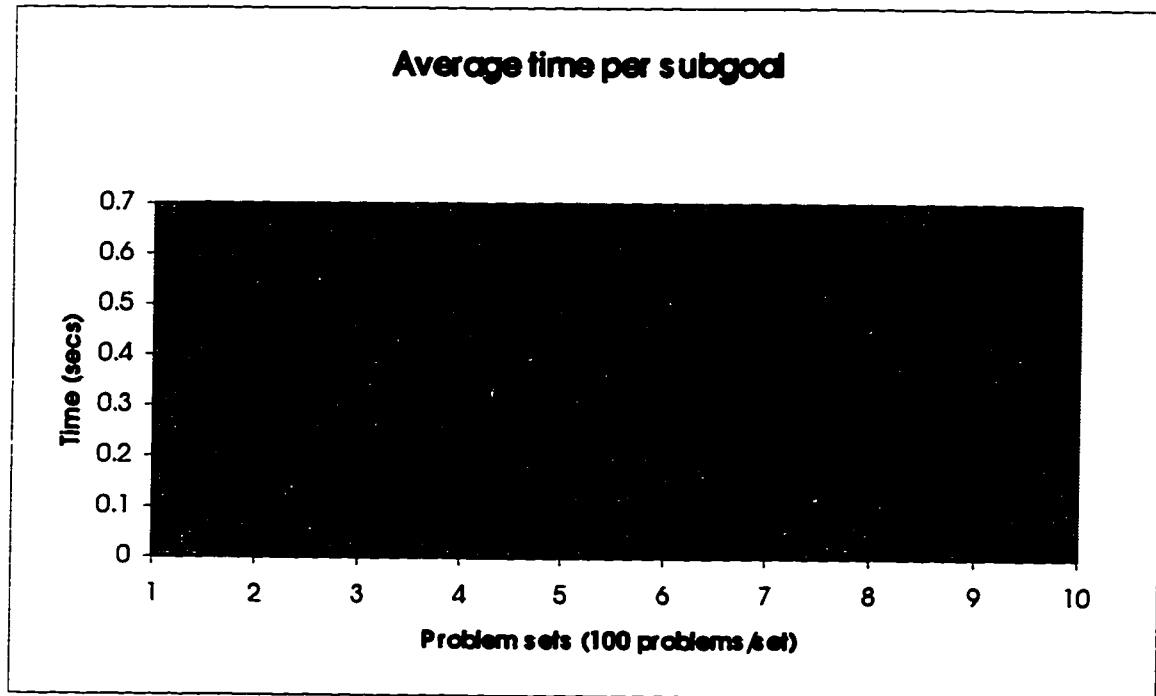
C.7.2 - Use of cases to solve goal statement conjuncts.



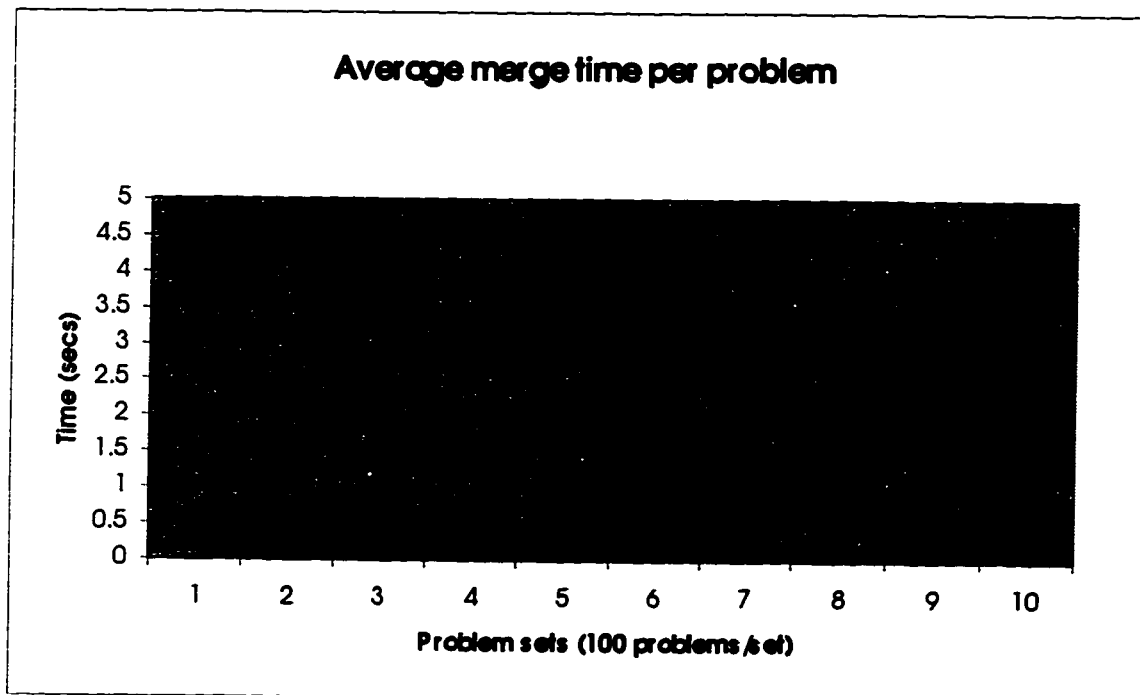
C.7.3 - Adaptation effort.



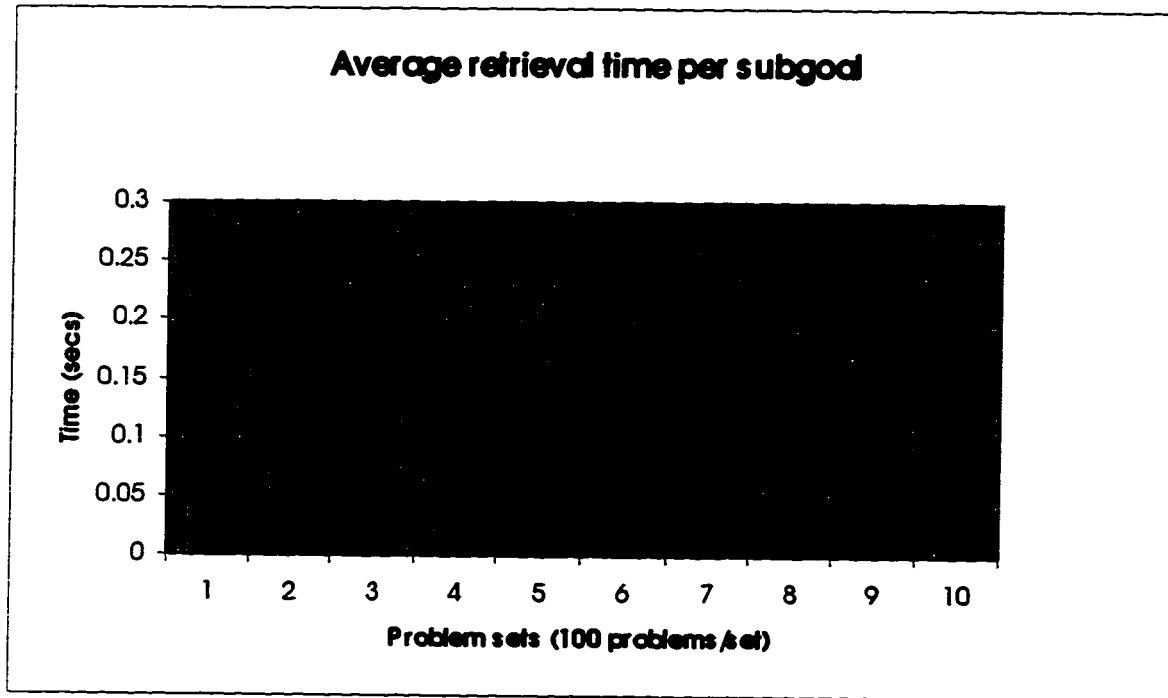
C.7.4 - Adaptation effort.



C.7.5 - Average time to solve one conjunct.

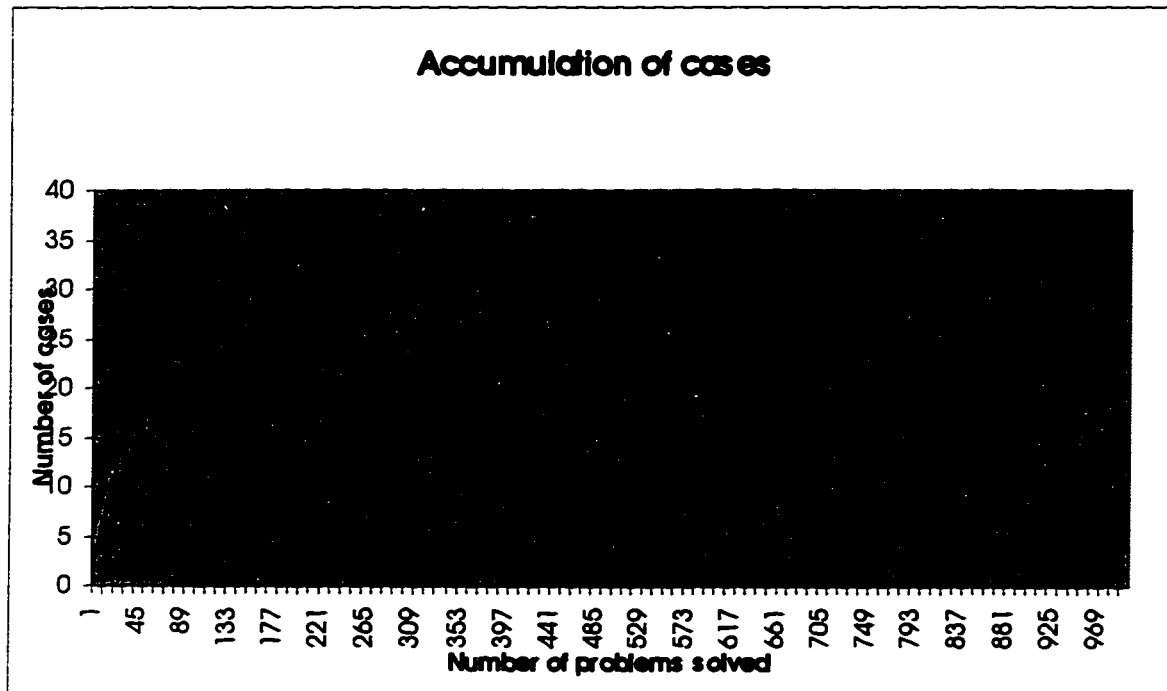


C.7.6 - Average merge time per problem.

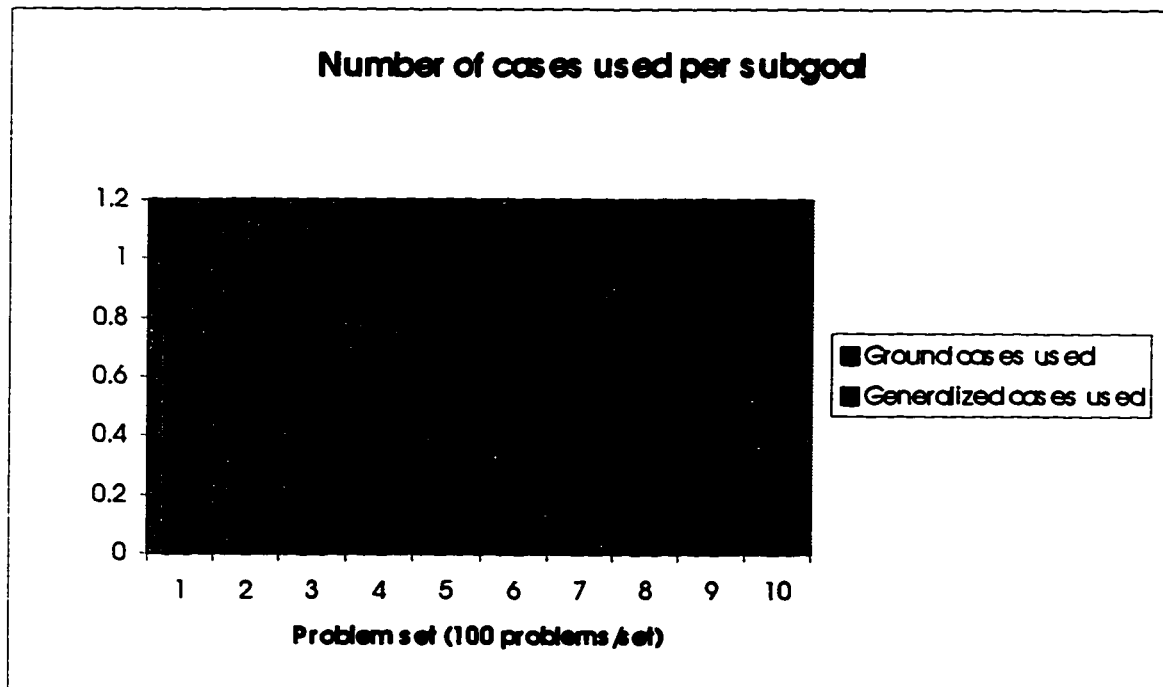


C.7.7 - Average retrieval time per goal statement conjunct.

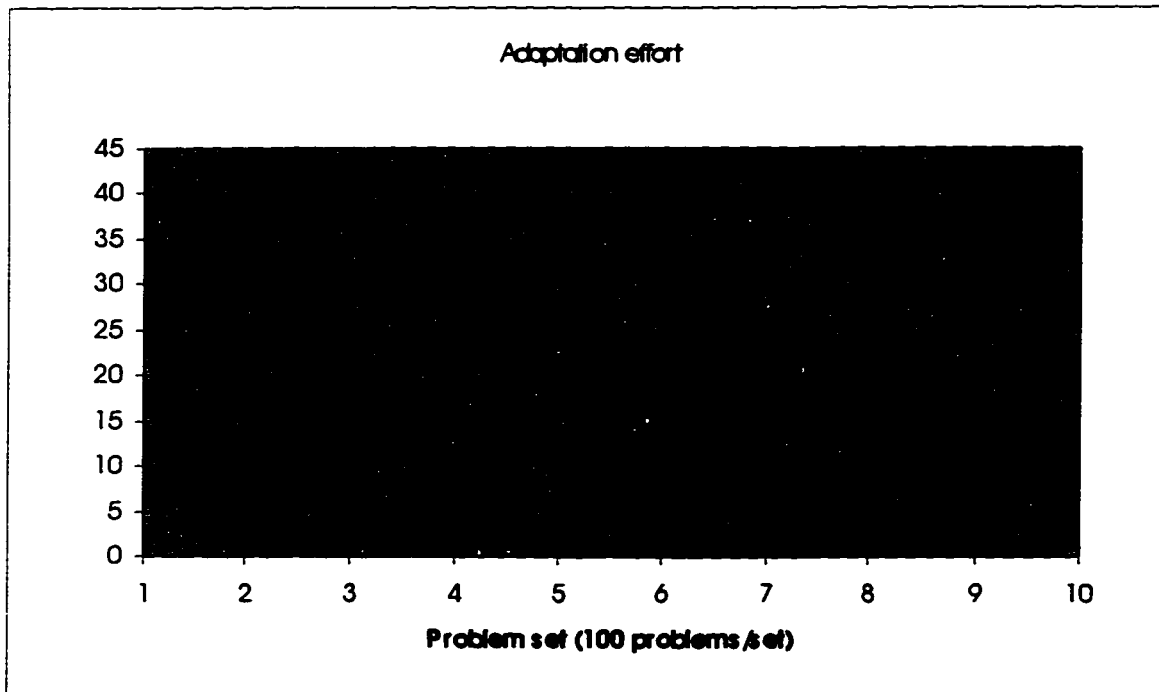
Problem set 8



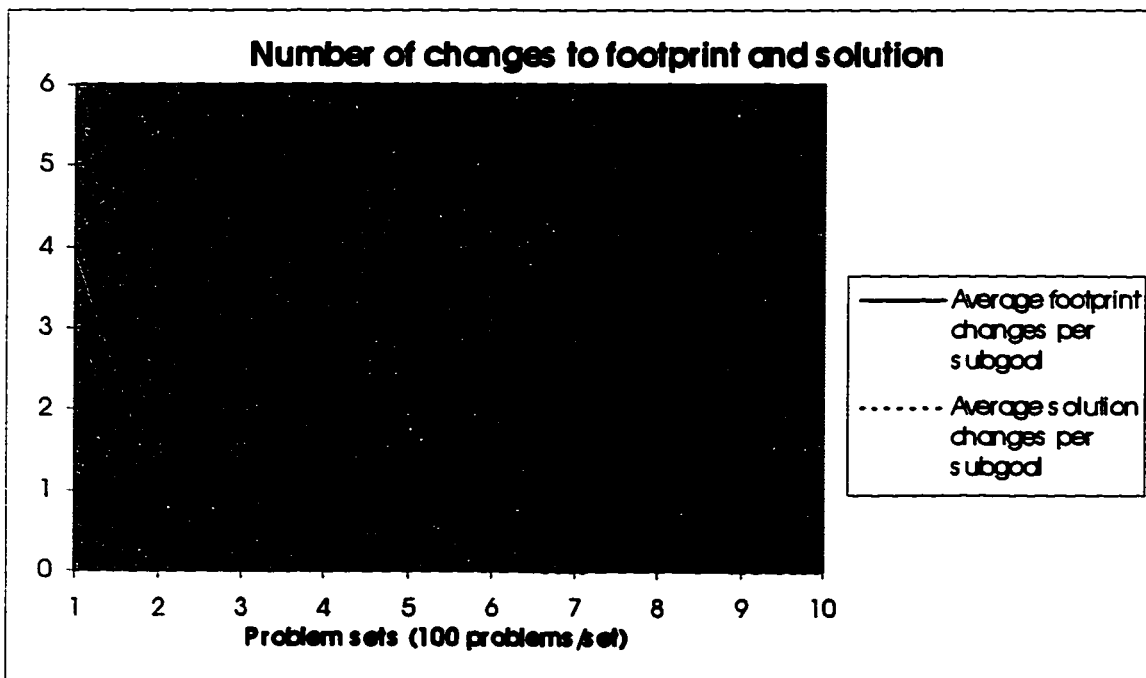
C.8.1 - Accumulation of cases for multiple conjunct goal statement problems.



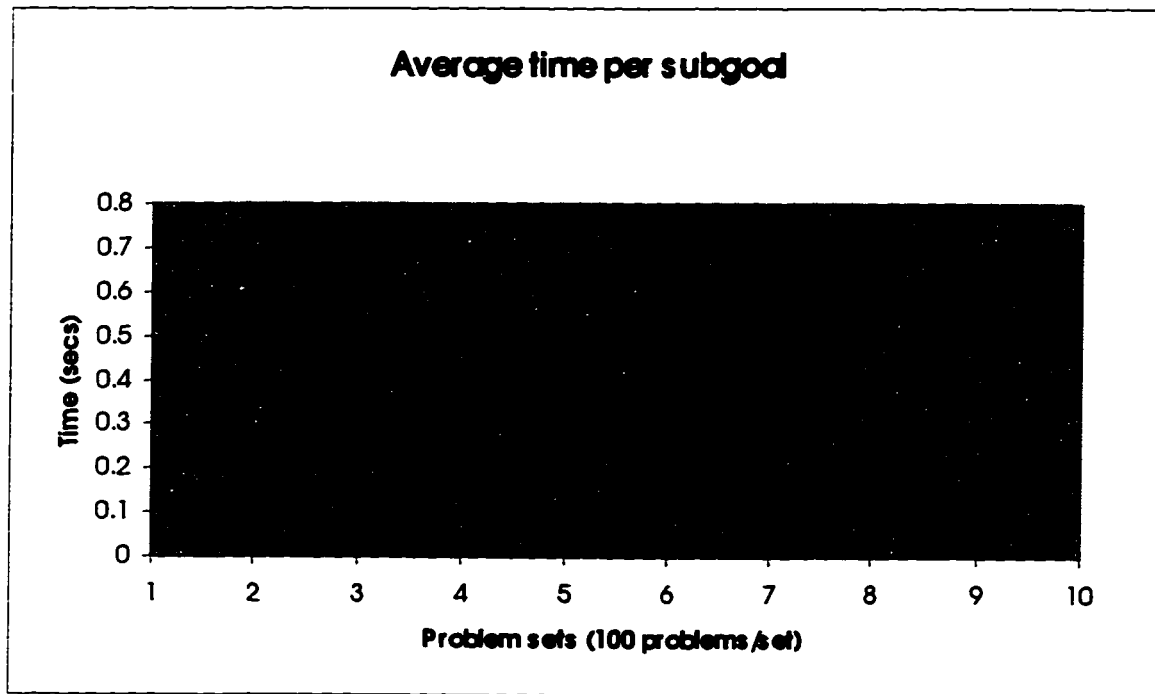
C.8.2 - Use of cases to solve goal statement conjuncts.



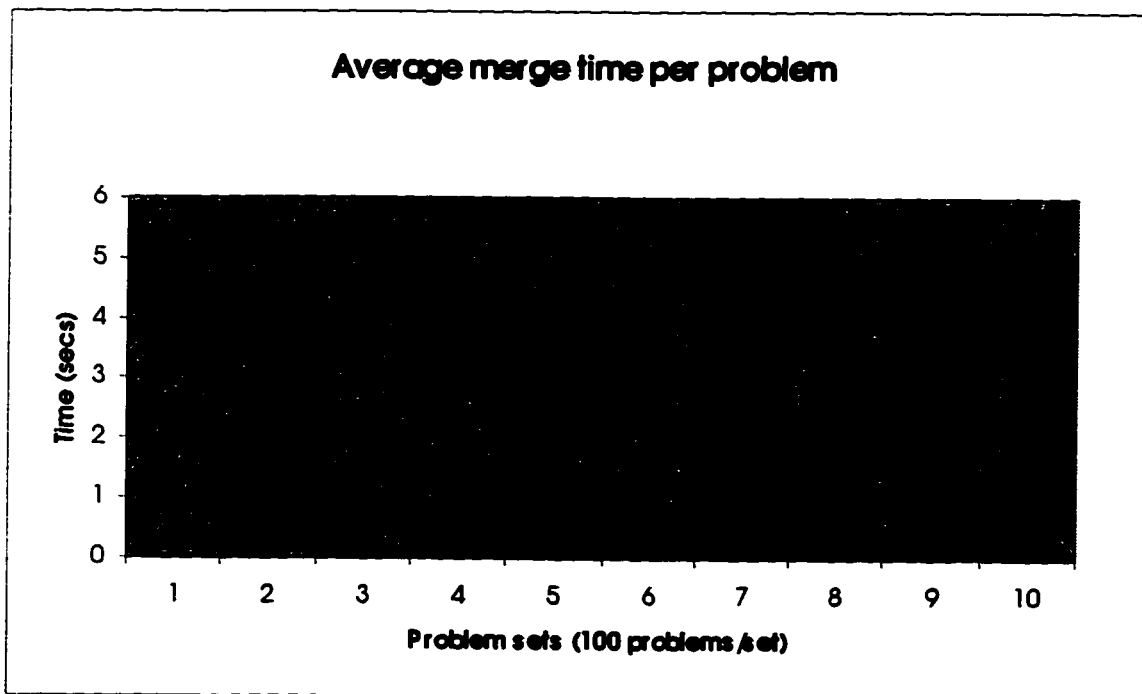
C.8.3 - Adaptation effort.



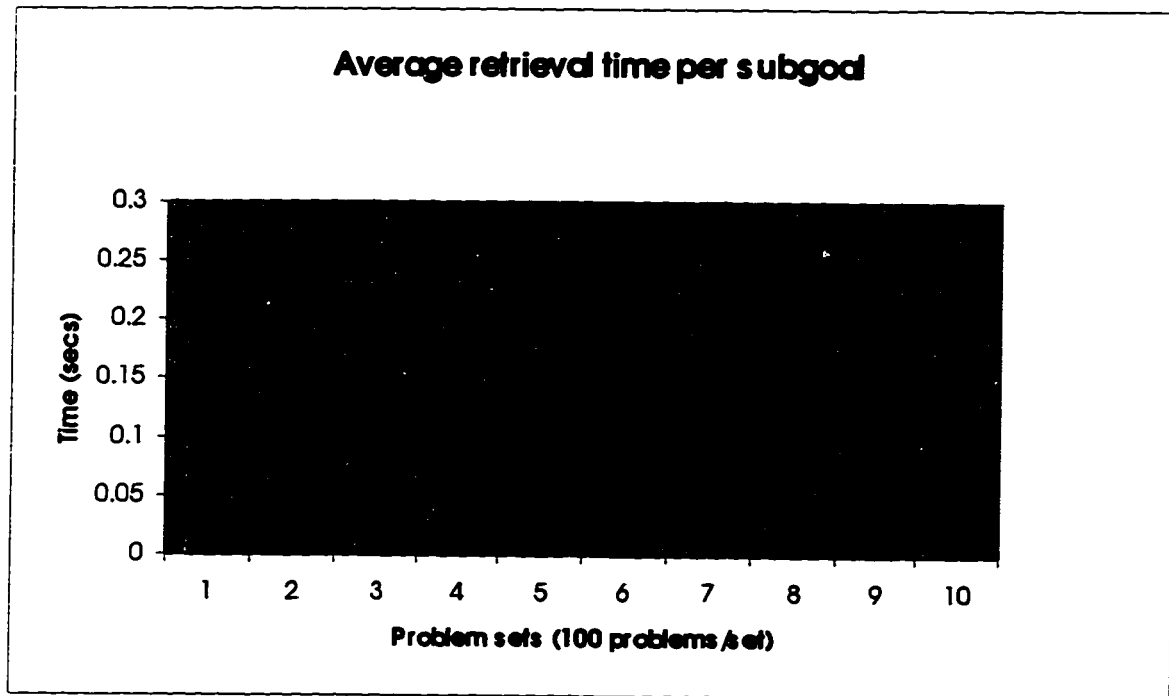
C.8.4 - Adaptation effort.



C.8.5 - Average time to solve one conjunct.

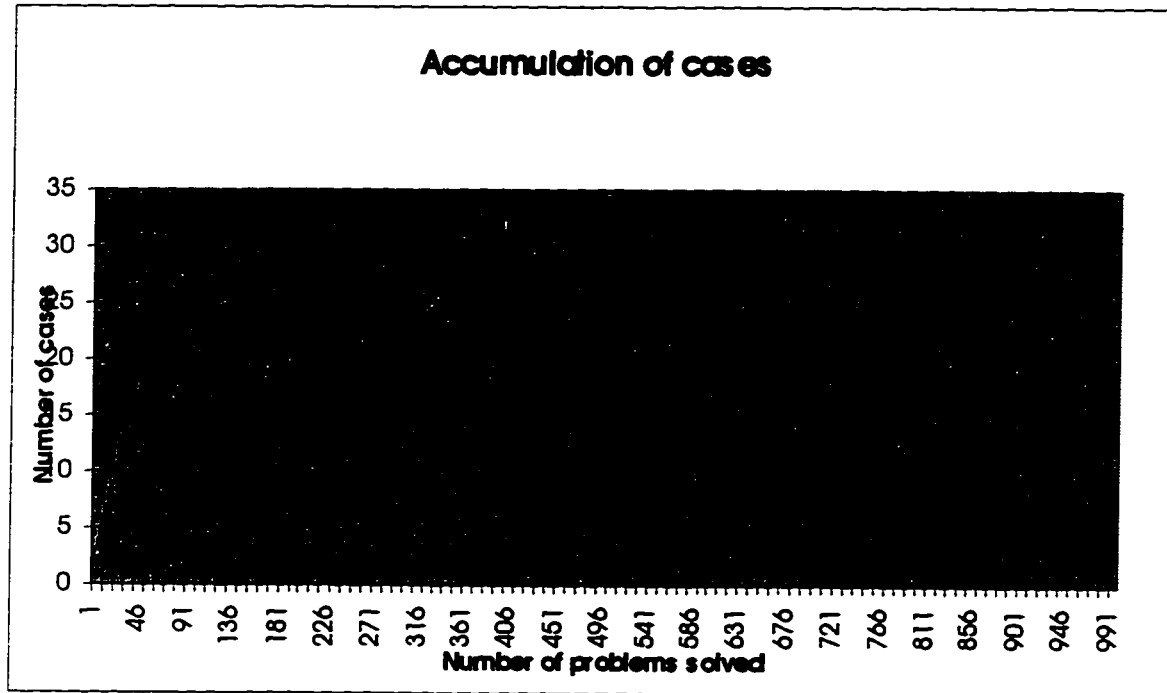


C.8.6 - Average merge time per problem.

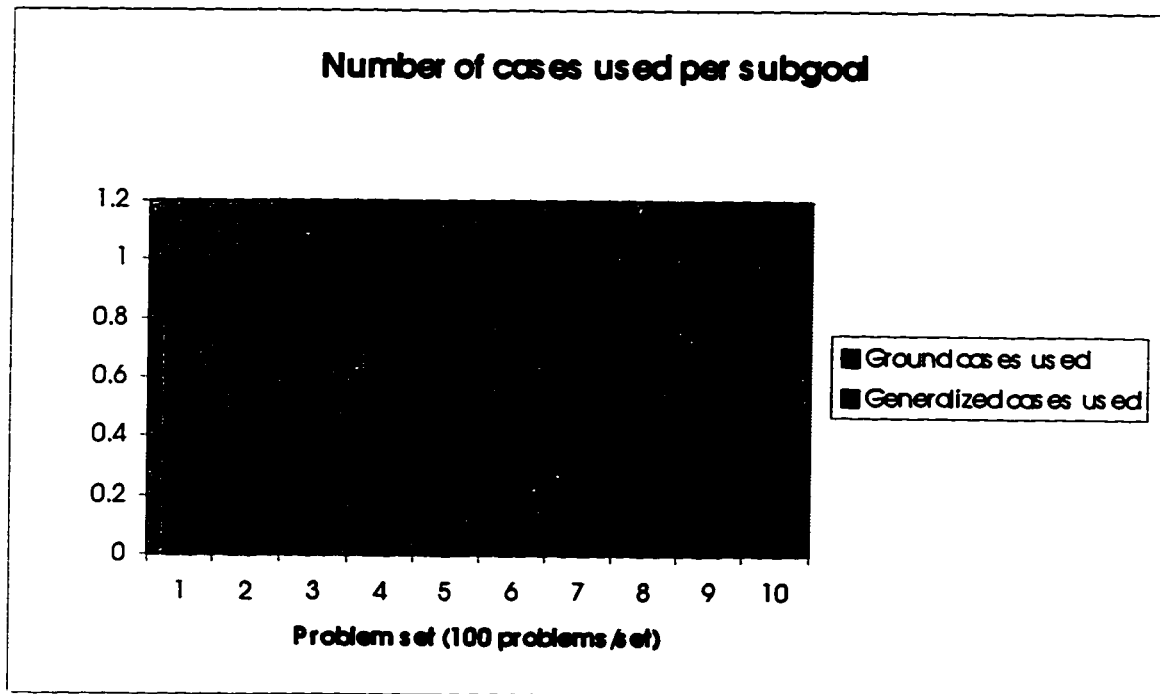


C.8.7 - Average retrieval time per goal statement conjunct.

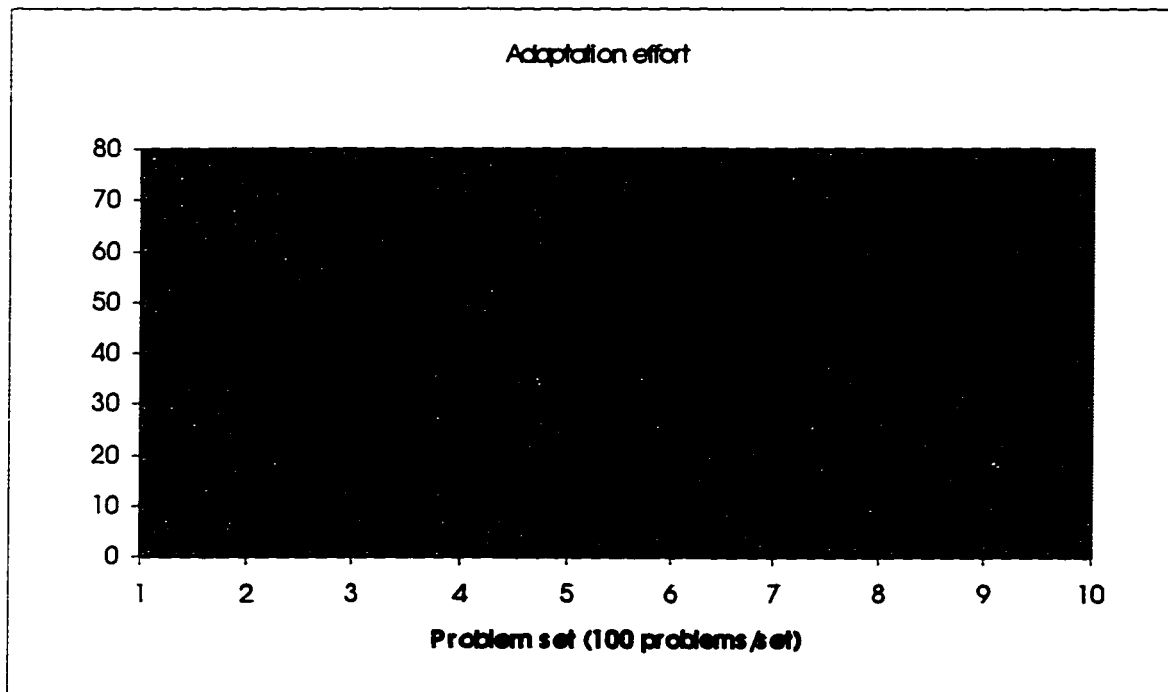
Problem set 9



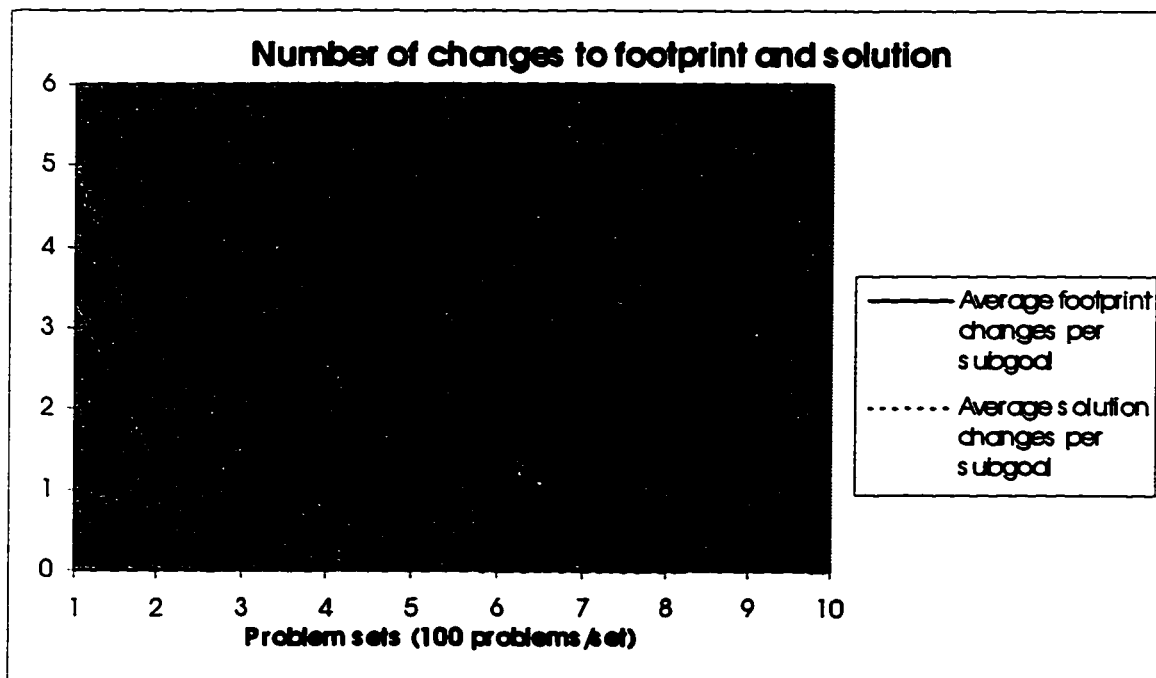
C.9.1 - Accumulation of cases for multiple conjunct goal statement problems.



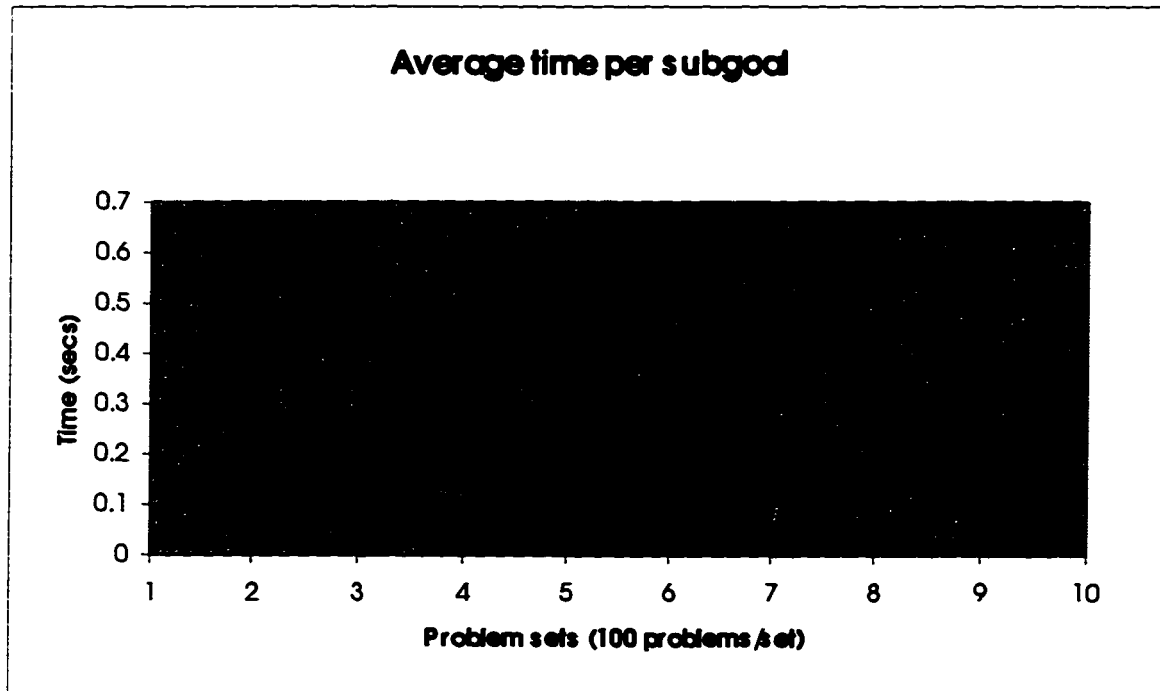
C.9.2 - Use of cases to solve goal statement conjuncts.



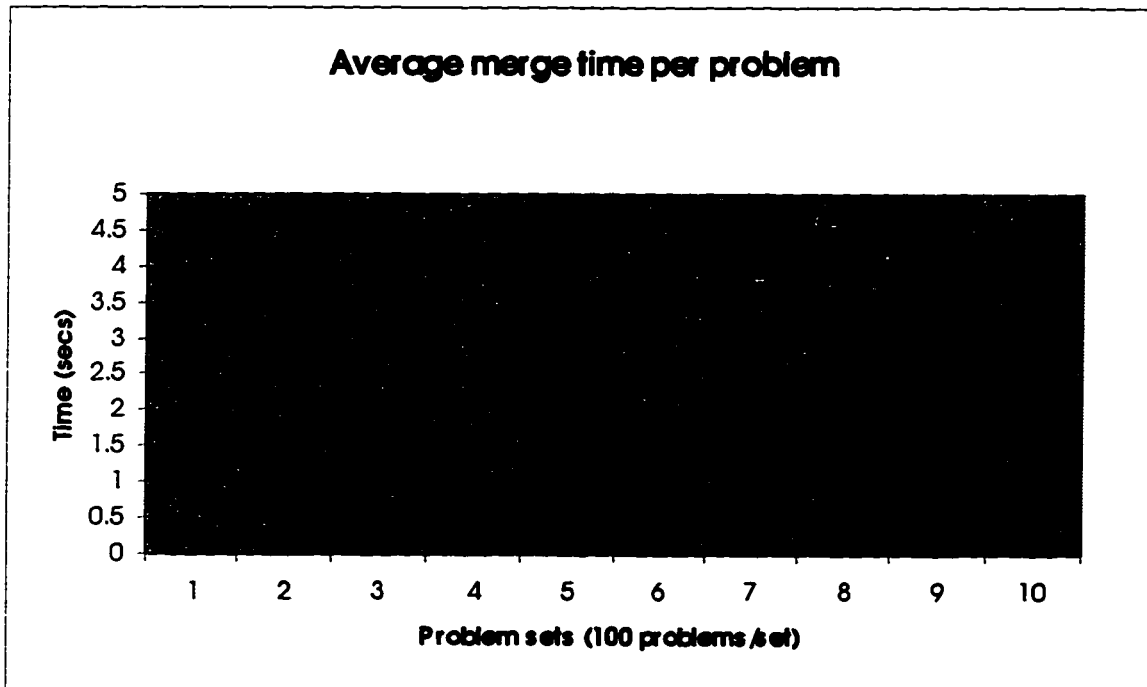
C.9.3 - Adaptation effort.



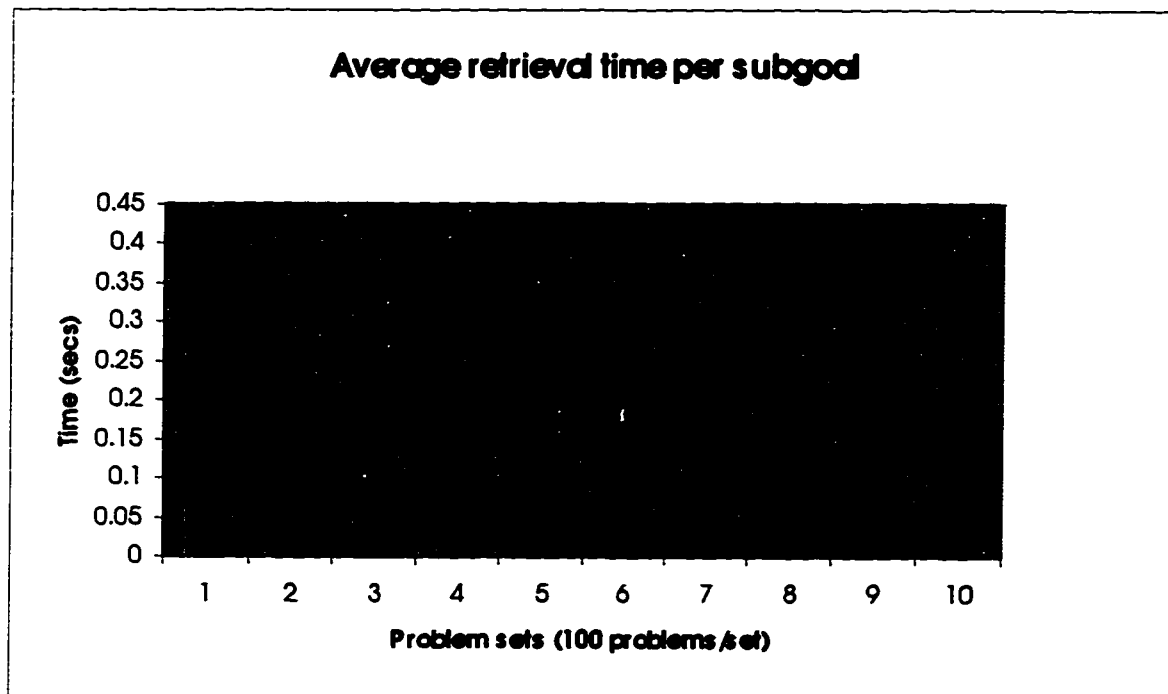
C.9.4 - Adaptation effort.



C.9.5 - Average time to solve one conjunct.

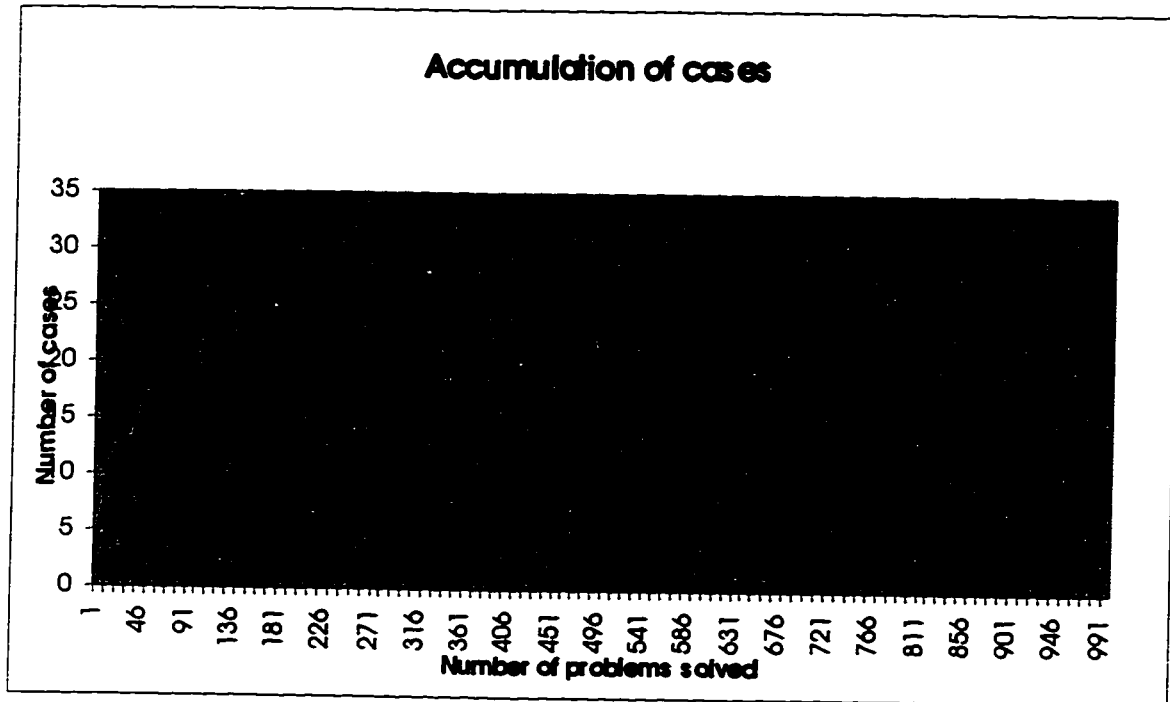


C.9.6 - Average merge time per problem.

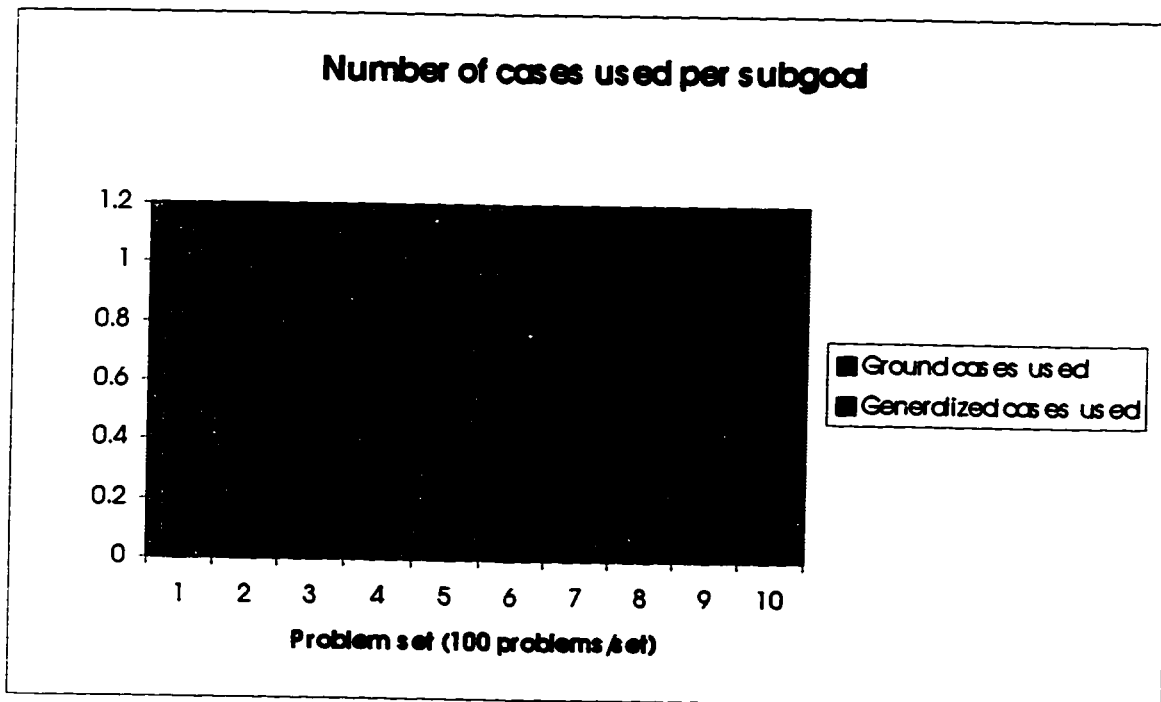


C.9.7 - Average retrieval time per goal statement conjunct.

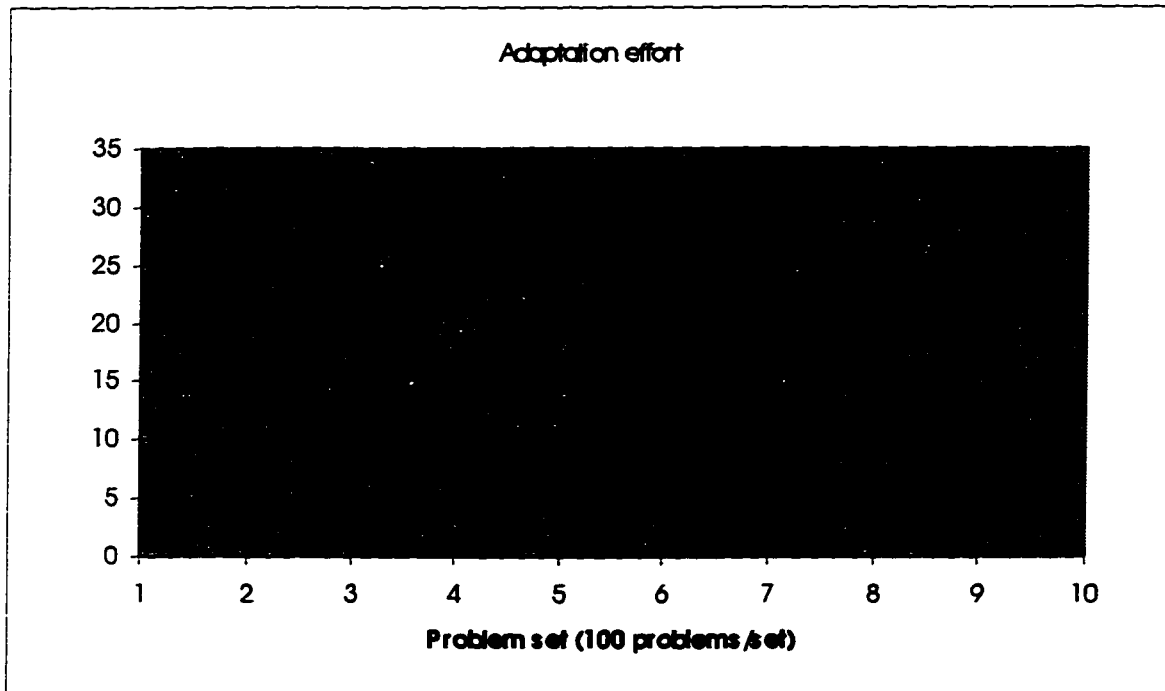
Problem set 10



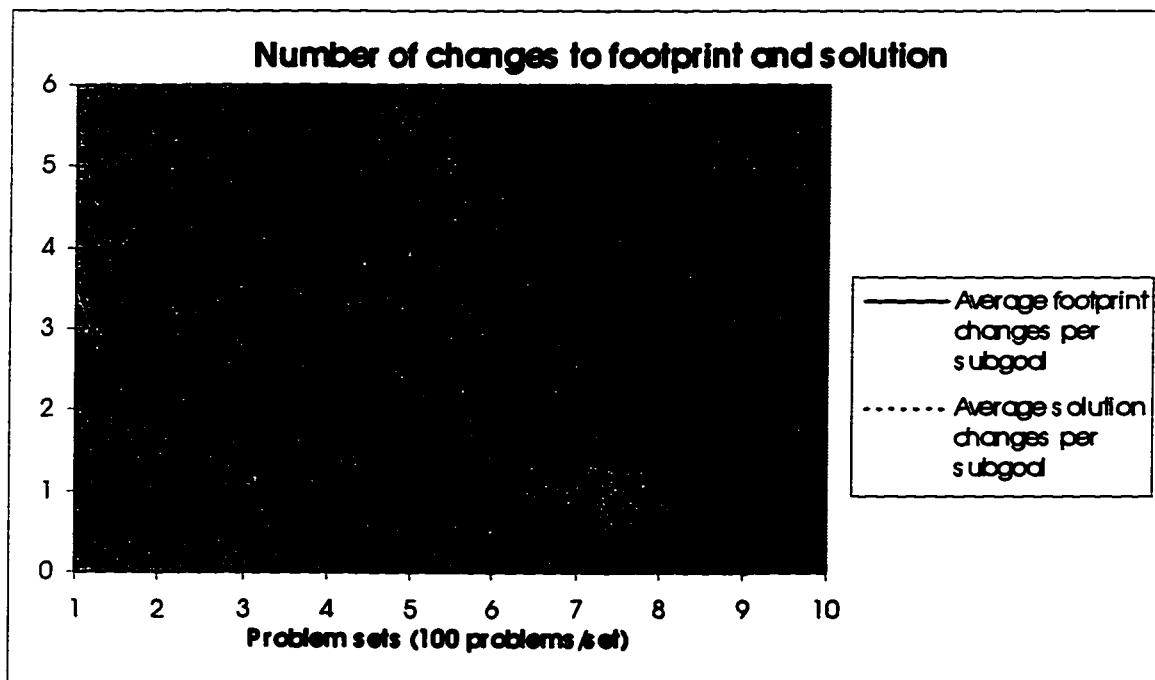
C.10.1 - Accumulation of cases for multiple conjunct goal statement problems.



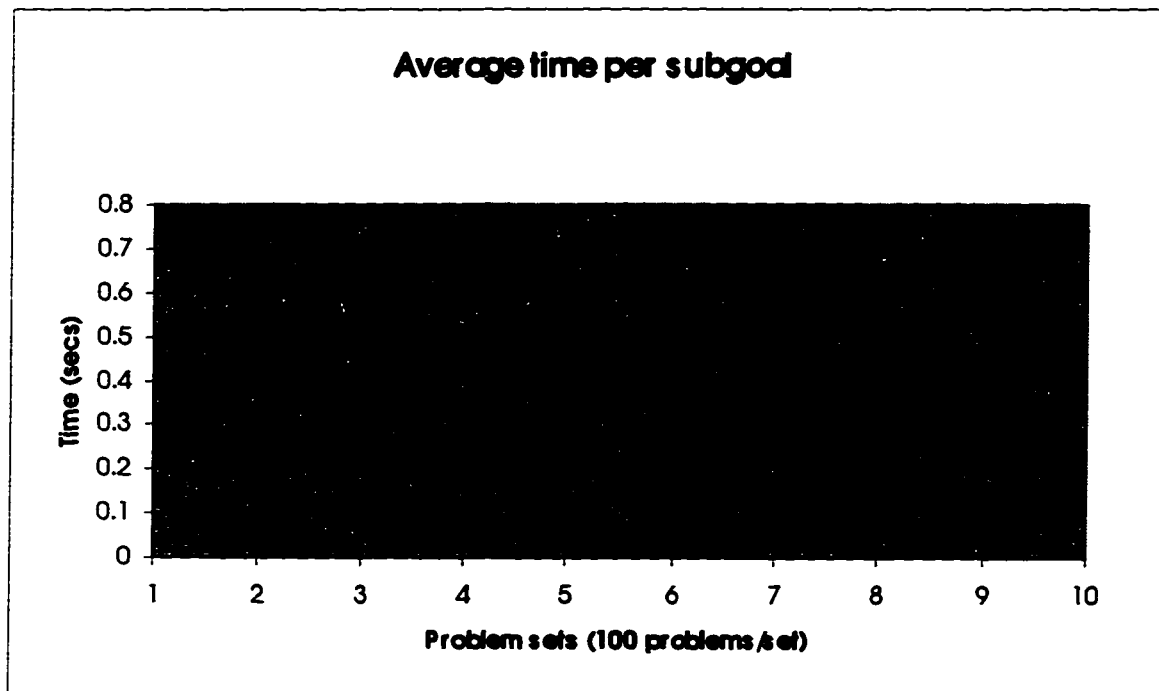
C.10.2 - Use of cases to solve goal statement conjuncts.



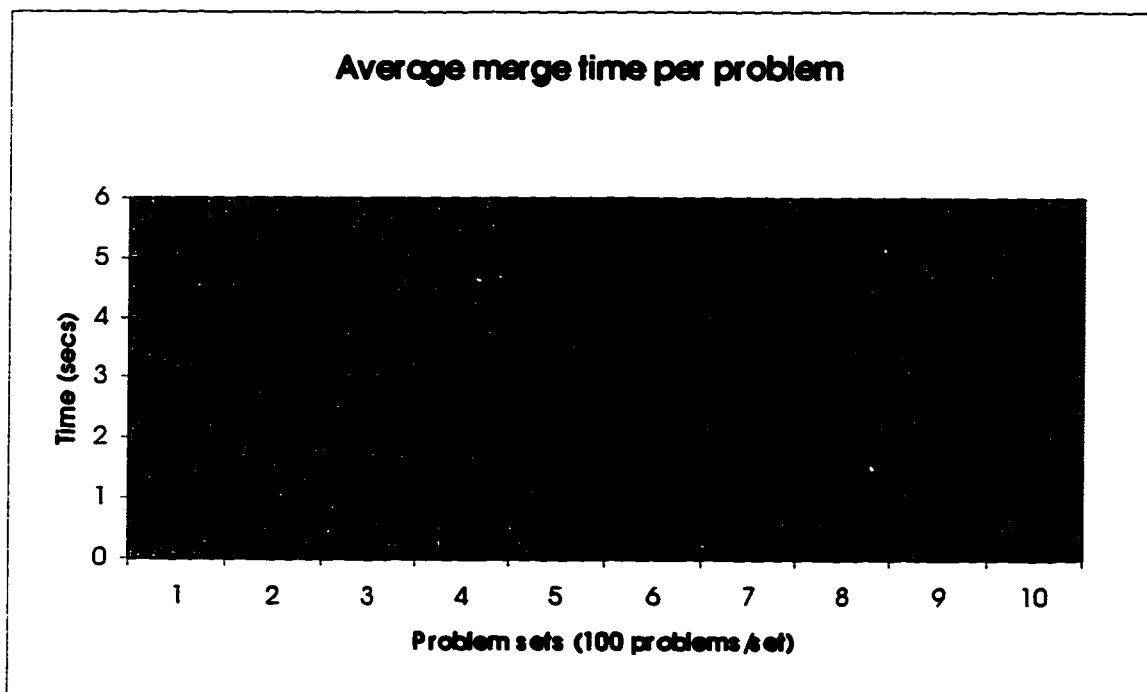
C.10.3 - Adaptation effort.



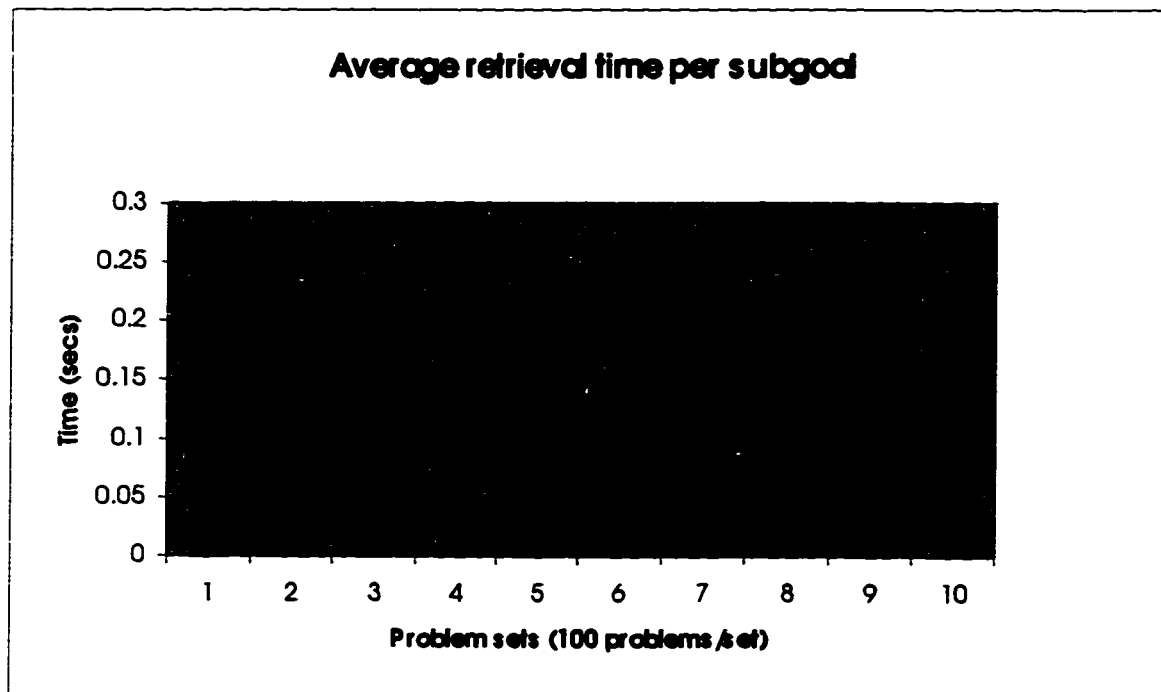
C.10.4 - Adaptation effort.



C.10.5 - Average time to solve one conjunct.



C.10.6 - Average merge time per problem.



C.10.7 - Average retrieval time per goal statement conjunct.

Appendix D

SEIDAM Planning Operators

```

aviris_calib
::
if aviris_scene(current_aviris_scene,bil_image_file,A)
and frame_get(current_aviris_scene,forced_fit_function,B)
and aviris_scene(current_aviris_scene,num_lines,C)
and aviris_scene(current_aviris_scene,gssflt_zero_file,D)
and aviris_scene(current_aviris_scene,gssflt_ten_file,E)
and frame_get(current_aviris_scene,image_directory,F)
then delete[]add[aviris_calib]

```

```

aviris_convolve_output
::
if aviris_scene(current_aviris_scene,optic_model_zero_file,A)
and aviris_scene(current_aviris_scene,opticmodel_ten_file,B)
and aviris_scene(current_aviris_scene,scene_number,C)
and frame_get(current_aviris_scene,image_directory,D)
then delete []
add [
    aviris_convolve_output,
    aviris_scene(current_aviris_scene,gssflt_ten_file,E),
    aviris_scene(current_aviris_scene,gssflt_zero_file,F)
]

```

```

aviris_copy_information
::
if frame_get(current_aviris_scene,image_directory,A)
then delete []
add [
    aviris_copy_information,
    hardware_information(hardware_info,expert_executables,B)
]

```

```

aviris_extractor
::
if true
then delete [
    aviris_images(current_aviris_images,image_file_name,A),
    aviris_images(current_aviris_images,names,B),
    class(image_full_filename,name,C)
]
add [
    aviris_extractor,
    'Directory'('File_Path',path,D),
    aviris_images(current_aviris_images,names,E),
    aviris_images(current_aviris_images,image_file_name,F)
]

```

```

aviris_moments_rad
::
if aviris_scene(current_aviris_scene,bil_image_file,A)
and aviris_scene(current_aviris_scene,num_lines,B)
and frame_get(current_aviris_scene,image_directory,C)
then delete[]add[aviris_moments_rad]

```

```

aviris_moments_refl
::
if aviris_scene(current_aviris_scene,num_lines,A)
and aviris_scene(current_aviris_scene,scene_number,B)
and frame_get(current_aviris_scene,image_directory,C)
then delete []
add [
    aviris_moments_refl,

```

```

    aviris_scene(current_aviris_scene,calibrated_file,D),
    aviris_scene(current_aviris_scene,bad_bands_file,E)
]

```

aviris_optic_model

```

::
if aviris_scene(current_aviris_scene,scene_number,A)
and frame_get(current_aviris_scene,day_of_year,B)
and frame_get(current_aviris_scene,image_directory,C)
and frame_get(current_aviris_scene,latitude_degrees,D)
and frame_get(current_aviris_scene,latitude_minutes,E)
and frame_get(current_aviris_scene,latitude_seconds,F)
and frame_get(current_aviris_scene,longitude_degrees,G)
and frame_get(current_aviris_scene,longitude_minutes,H)
and frame_get(current_aviris_scene,longitude_seconds,I)
and frame_get(current_aviris_scene,time_hour,J)
and frame_get(current_aviris_scene,time_minute,K)
and frame_get(current_aviris_scene,time_second,L)
then delete []
add [
    aviris_optic_model,
    aviris_scene(current_aviris_scene,constraint_ten,M),
    aviris_scene(current_aviris_scene,constraint_zero,N),
    aviris_scene(current_aviris_scene,optic_model_zero_file,O),
    aviris_scene(current_aviris_scene,opticmodel_ten_file,P),
    aviris_scene(current_aviris_scene,spectral_calibration_file,Q),
    aviris_scene(current_aviris_scene,temperature,R),
    aviris_scene(current_aviris_scene,water_vapour_scaling,S),
    aviris_scene(current_aviris_scene,ozone_scaling,T),
    aviris_scene(current_aviris_scene,co2_scaling,U),
    aviris_scene(current_aviris_scene,co_scaling,V),
    aviris_scene(current_aviris_scene,ch4_scaling,W),
    aviris_scene(current_aviris_scene,n2o_scaling,X),
    aviris_scene(current_aviris_scene,o2_scaling,Y),
    aviris_scene(current_aviris_scene,nh3_scaling,Z),
    aviris_scene(current_aviris_scene,no_scaling,AA),
    aviris_scene(current_aviris_scene,no2_scaling,AB),
    aviris_scene(current_aviris_scene,so2_scaling,AC),
    aviris_scene(current_aviris_scene,unknown_scaling,AD)
]

```

aviris_plot_optic_model

```

::
if aviris_scene(current_aviris_scene,optic_model_zero_file,A)
and aviris_scene(current_aviris_scene,opticmodel_ten_file,B)
and aviris_scene(current_aviris_scene,scene_number,C)
and frame_get(current_aviris_scene,image_directory,D)
and hardware_information(hardware_info,avs_machine,E)
and hardware_information(hardware_info,avs_machine_secondary,F)
then delete [host_computer_class(work_station_host,name,G)]
add [
    aviris_plot_optic_model,
    aviris_scene(current_aviris_scene,opticmodel_postscript_file,H),
    host_computer_class(work_station_host,name,I)
]

```

aviris_s2n

```

::
if aviris_scene(current_aviris_scene,scene_number,A)
and frame_get(current_aviris_scene,frame_number,B)
and frame_get(current_aviris_scene,image_directory,C)
and hardware_information(hardware_info,display_machine,D)
then delete [
    aviris_scene(current_aviris_scene,day_of_year,E),
    aviris_scene(current_aviris_scene,latitude_degrees,F),
    aviris_scene(current_aviris_scene,latitude_minutes,G),

```

```

    aviris_scene(current_aviris_scene,latitude_seconds,H),
    aviris_scene(current_aviris_scene,longitude_degrees,I),
    aviris_scene(current_aviris_scene,longitude_minutes,J),
    aviris_scene(current_aviris_scene,longitude_seconds,K),
    aviris_scene(current_aviris_scene,num_lines,L),
    aviris_scene(current_aviris_scene,time_hour,M),
    aviris_scene(current_aviris_scene,time_minute,N),
    aviris_scene(current_aviris_scene,time_second,O)
]
add [
    aviris_s2n,
    aviris_scene(current_aviris_scene,bil_image_file,P),
    aviris_scene(current_aviris_scene,num_lines,Q),
    aviris_scene(current_aviris_scene,s2n_postscript_file,R),
    aviris_scene(current_aviris_scene,s2n_source_file,S),
    hardware_information(hardware_info,avs_machine,T),
    hardware_information(hardware_info,avs_machine_secondary,U),
    aviris_scene(current_aviris_scene,day_of_year,V),
    aviris_scene(current_aviris_scene,latitude_degrees,W),
    aviris_scene(current_aviris_scene,latitude_minutes,X),
    aviris_scene(current_aviris_scene,latitude_seconds,Y),
    aviris_scene(current_aviris_scene,longitude_degrees,Z),
    aviris_scene(current_aviris_scene,longitude_minutes,AA),
    aviris_scene(current_aviris_scene,longitude_seconds,AB),
    aviris_scene(current_aviris_scene,time_hour,AC),
    aviris_scene(current_aviris_scene,time_minute,AD),
    aviris_scene(current_aviris_scene,time_second,AE),
    hardware_information(hardware_info,expert_source,AF)
]

```

```

aviris_tape2disk
::
if frame_get(current_aviris_scene,image_directory,A)
and frame_get(hardware_info,tape_name,B)
and frame_get(hardware_info,remote_tape_machine,C)
then delete[]add[aviris_tape2disk]

```

```

avs
::
if frame_get(avs_host,name,A)
and frame_get(current_avs_image,filename,B)
and frame_get(current_ip_module,name,C)
then delete [
    avs_module(current_ip_module,port_number,D),
    host_computer_class(work_station_host,name,E)
]
add [
    avs,
    avs_module(current_ip_module,port_number,F),
    host_computer_class(work_station_host,name,G)
]

```

```

calibrate_aviris_scene
:: if true then delete[]add[calibrate_aviris_scene]

```

```

change_lakes_to_zero
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,lakes_map_dbib,B)
and map_sheet(A,copy_hydrology_to_dem,C)
and map_sheet(A,dem_pci_file,D)
and map_sheet(A,lakes_sieve_dboc,E)
and map_sheet(A,lakes_sieve_dbic,F)
and frame_get(working_directory,path,G)
then delete [map_sheet(A,change_lake_elevation,H)]
add [change_lakes_to_zero,map_sheet(A,change_lake_elevation,0)]

```

```

cleanup
::
if frame_get(current_map_sheet,name,A)
and frame_get(working_directory,path,B)
then delete [
    map_sheet(A,arc_lakes_file,C),
    map_sheet(A,clear_cut_all,D),
    map_sheet(A,clear_cut_arc_file,E),
    map_sheet(A,clear_cut_none,F),
    map_sheet(A,clear_cut_only_file,G),
    map_sheet(A,clear_cut_part,H),
    map_sheet(A,clear_cut_poly_file,I),
    map_sheet(A,copy_dem_file_into_small_tm_file,J),
    map_sheet(A,create_small_tm_file,K),
    map_sheet(A,creating_isoclus_themes,L),
    map_sheet(A,dem_easting_distance,M),
    map_sheet(A,dem_erdas_file,N),
    map_sheet(A,dem_file,O),
    map_sheet(A,dem_grid_file,P),
    map_sheet(A,dem_loright_easting,Q),
    map_sheet(A,dem_loright_northing,R),
    map_sheet(A,dem_no_lakes_file,S),
    map_sheet(A,dem_northing_distance,T),
    map_sheet(A,dem_pci_file,U),
    map_sheet(A,dem_pci_num_lines,V),
    map_sheet(A,dem_pci_num_pixels,W),
    map_sheet(A,dem_tin_file,X),
    map_sheet(A,dem_tin_max_easting,Y),
    map_sheet(A,dem_tin_max_northing,Z),
    map_sheet(A,dem_tin_mesh_points,AA),
    map_sheet(A,dem_tin_min_easting,AB),
    map_sheet(A,dem_tin_min_northing,AC),
    map_sheet(A,digitize_road_network,AD),
    map_sheet(A,display_arc_coverage,AE),
    map_sheet(A,dlg_lakes_file,AF),
    map_sheet(A,dlg_roads_file,AG),
    map_sheet(A,erdas_lakes_file,AH),
    map_sheet(A,grid_lakes_file,AI),
    map_sheet(A,highlight_roads,AJ),
    map_sheet(A,isoclus_channel,AK),
    map_sheet(A,kclus_channel,AL),
    map_sheet(A,lakes_dlg_file,AM),
    map_sheet(A,lakes_map_dbib,AN),
    map_sheet(A,lakes_num_lakes,AO),
    map_sheet(A,lakes_thr_dbib,AP),
    map_sheet(A,last_isoclus_thr_dbib,AQ),
    map_sheet(A,mas_blue_channel,AR),
    map_sheet(A,mas_dem_channel,AS),
    map_sheet(A,mas_filename,AT),
    map_sheet(A,mas_green_channel,AU),
    map_sheet(A,mas_red_channel,AV),
    map_sheet(A,pca_output_1,AW),
    map_sheet(A,pca_output_2,AX),
    map_sheet(A,pca_output_3,AY),
    map_sheet(A,pci_lakes_file,AZ),
    map_sheet(A,roads_arc_file,BA),
    map_sheet(A,roads_dlg_file,BB),
    map_sheet(A,roads_lin_file,BC),
    map_sheet(A,segment_roads,BD),
    map_sheet(A,segment_clear_cuts,BE),
    map_sheet(A,sieve_isoclus,BF),
    map_sheet(A,sieve_kclus,BG),
    map_sheet(A,sieve_lakes_dbic,BH),
    map_sheet(A,sieve_lakes_dboc,BI),
    map_sheet(A,sieve_lakes_file,BJ),
    map_sheet(A,sieve_lakes_sthresh,BK),
    map_sheet(A,sieved_isoclus_channel,BL),
    map_sheet(A,sieved_kclus_channel,BM),
    map_sheet(A,source_files_copied,BN),
    map_sheet(A,threshold_isoclus,BO),

```

```

map_sheet(A,tm_file,BP),
map_sheet(A,tm_large_file,BQ),
map_sheet(A,tm_large_line_size,BR),
map_sheet(A,tm_large_pixel_size,BS),
map_sheet(A,tm_large_start_line,BT),
map_sheet(A,tm_large_start_pixel,BU),
map_sheet(A,tm_upleft_easting,BV),
map_sheet(A,tm_upleft_northing,BW),
map_sheet(A,tm_loright_easting,BX),
map_sheet(A,tm_loright_northing,BY),
map_sheet(A,tm_small_channel_16s,BZ),
map_sheet(A,tm_small_channel_16u,CA),
map_sheet(A,tm_small_channel_32r,CB),
map_sheet(A,tm_small_channel_8u,CC),
map_sheet(A,tm_small_dem_channel,CD),
map_sheet(A,tm_small_easting_distance,CE),
map_sheet(A,tm_small_file,CF),
map_sheet(A,tm_small_loright_easting,CG),
map_sheet(A,tm_small_loright_northing,CH),
map_sheet(A,tm_small_northing_distance,CI),
map_sheet(A,tm_small_num_lines,CJ),
map_sheet(A,tm_small_num_pixels,CK),
map_sheet(A,tm_small_upleft_easting,CL),
map_sheet(A,tm_small_upleft_northing,CM),
map_sheet(A,tm_small_with_dem_file,CN),
map_sheet(A,updated_clear_cut_coverage,CO),
map_sheet(A,updated_clear_cut_segment,CP),
map_sheet(A,updated_clear_cut_point_file,CQ),
map_sheet(A,updated_clear_cut_vector_file,CR),
map_sheet(A,updated_coverage_file,CS),
map_sheet(A,updated_road_point_file,CT),
map_sheet(A,updated_road_segment,CU),
map_sheet(A,updated_road_vector_file,CV)
]
add [cleanup]

concat_map_sheets
::
if frame_get(arc_host,name,A)and frame_get(working_directory,path,B)
then delete[add[concat_map_sheets]]

copy_bitmap
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,sieve_lakes_file,B)
and map_sheet(A,lakes_thr_dbib,C)
and frame_get(working_directory,path,D)and map_sheet(A,dem_pci_file,E)
then delete [map_sheet(A,lakes_map_dbib,F)]
add [
copy_bitmap,
map_sheet(A,dem_no_lakes_file,E),
map_sheet(A,lakes_map_dbib,G)
]

copy_files_to_working_dir
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and frame_get(gis_directory,path,C)
and frame_get(rs_directory,path,D)
and frame_get(B,tm_image_name,E)and frame_get(working_directory,path,F)
then delete [map_sheet(B,source_files_copied,G),map_sheet(B,tm_large_file,H)]
add [
copy_files_to_working_dir,
map_sheet(B,source_files_copied,yes),
map_sheet(B,tm_large_file,'tm_large.pix')
]

```

```

create_tin
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,arc_lakes_file,C)
and map_sheet(B,dem_file,D) and frame_get(working_directory,path,E)
then delete [
    map_sheet(B,dem_tin_min_northing,F),
    map_sheet(B,dem_tin_max_northing,G),
    map_sheet(B,dem_tin_min_easting,H),
    map_sheet(B,dem_tin_max_easting,I)
]
add [
    create_tin,
    map_sheet(B,dem_grid_file,J),
    map_sheet(B,dem_erdas_file,K),
    map_sheet(B,dem_tin_file,L),
    map_sheet(B,dem_tin_mesh_points,M),
    map_sheet(B,dem_tin_min_northing,N),
    map_sheet(B,dem_tin_max_northing,O),
    map_sheet(B,dem_tin_min_easting,P),
    map_sheet(B,dem_tin_max_easting,Q)
]

create_tm_small
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_large_pixel_size,B)
and map_sheet(A,tm_large_line_size,C)
and map_sheet(A,dem_no_lakes_file,D)
and map_sheet(A,tm_large_file,E)
and frame_get(working_directory,path,F)
and map_sheet(A,dem_upleft_easting,G)
and map_sheet(A,dem_upleft_northing,H)
then delete [map_sheet(A,dem_pci_num_pixels,I),map_sheet(A,dem_pci_num_lines,J)]
add [
    create_tm_small,
    map_sheet(A,tm_small_file,K),
    map_sheet(A,tm_small_channel_8u,L),
    map_sheet(A,tm_small_num_lines,M),
    map_sheet(A,tm_small_num_pixels,N),
    map_sheet(A,tm_small_channel_32r,O),
    map_sheet(A,tm_small_channel_16u,P),
    map_sheet(A,tm_small_channel_16s,Q),
    map_sheet(A,tm_large_start_pixel,R),
    map_sheet(A,tm_large_start_line,S),
    map_sheet(A,dem_pci_num_pixels,T),
    map_sheet(A,dem_pci_num_lines,U)
]

dem_to_pix_image_match
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,dem_pci_num_pixels,B)
and map_sheet(A,dem_pci_num_lines,C)
and map_sheet(A,dem_no_lakes_file,D)
and frame_get(working_directory,path,E)
and map_sheet(A,tm_small_file,F)
and map_sheet(A,tm_small_channel_8u,G)
and map_sheet(A,tm_small_channel_16s,H)
and map_sheet(A,tm_small_channel_16u,I)
then delete []
add [
    dem_to_pix_image_match,
    map_sheet(A,tm_small_with_dem_file,F),

```

```

    map_sheet(A,tm_small_dem_channel,J)
]

```

```

digitize_clear_cuts

```

```

::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,segment_clear_cuts,B)
and map_sheet(A,tm_small_num_pixels,C)
and map_sheet(A,tm_small_num_lines,D)
and map_sheet(A,tm_file,E)
and map_sheet(A,tm_small_upleft_easting,F)
and map_sheet(A,tm_small_upleft_northing,G)
and frame_get(working_directory,path,H)
and map_sheet(A,tm_large_line_size,I)
and map_sheet(A,tm_large_pixel_size,J)
then delete [map_sheet(A,updated_clear_cut_segment,K)]
add [
    digitize_clear_cuts,
    map_sheet(A,updated_clear_cut_segment,L),
    map_sheet(A,tm_small_easting_distance,M),
    map_sheet(A,tm_small_northing_distance,N)
]

```

```

digitize_clear_cuts_tercom

```

```

::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,segment_clear_cuts,B)
and map_sheet(A,tm_small_num_pixels,C)
and map_sheet(A,tm_small_num_lines,D)
and map_sheet(A,tm_small_file,E)
and frame_get(A,dem_upleft_easting,F)
and frame_get(A,dem_upleft_northing,G)
and map_sheet(A,last_isoclus_thr_dbib,H)
and frame_get(working_directory,path,I)
and map_sheet(A,tm_large_line_size,J)
and map_sheet(A,tm_large_pixel_size,K)
then delete [
    map_sheet(A,updated_clear_cut_segment,L),
    map_sheet(A,digitize_road_network,M),
    map_sheet(A,clear_cut_all,N),
    map_sheet(A,clear_cut_part,O),
    map_sheet(A,clear_cut_none,P)
]
add [
    digitize_clear_cuts_tercom,
    map_sheet(A,updated_clear_cut_segment,Q),
    map_sheet(A,digitize_road_network,done),
    map_sheet(A,dem_easting_distance,R),
    map_sheet(A,dem_northing_distance,S),
    map_sheet(A,clear_cut_all,T),
    map_sheet(A,clear_cut_part,T),
    map_sheet(A,clear_cut_none,T),
    map_sheet(A,tm_small_northing_distance,U),
    map_sheet(A,tm_small_easting_distance,V)
]

```

```

digitize_clear_cuts_with_clustering

```

```

::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_num_pixels,B)
and map_sheet(A,tm_small_num_lines,C)
and map_sheet(A,tm_small_file,D)
and map_sheet(A,segment_clear_cuts,E)
and map_sheet(A,last_isoclus_thr_dbib,F)
and frame_get(working_directory,path,G)
and map_sheet(A,tm_large_line_size,H)

```

```

and map_sheet(A,tm_large_pixel_size,I)
then delete [
    map_sheet(A,clear_cut_all,J),
    map_sheet(A,clear_cut_part,K),
    map_sheet(A,clear_cut_none,L)
]
add [
    digitize_clear_cuts_with_clustering,
    map_sheet(A,clear_cut_all,M),
    map_sheet(A,clear_cut_part,M),
    map_sheet(A,clear_cut_none,M),
    map_sheet(A,tm_small_northing_distance,N),
    map_sheet(A,tm_small_easting_distance,O)
]

```

```

digitize_roads
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,segment_roads,B)
and map_sheet(A,tm_small_num_pixels,C)
and map_sheet(A,tm_small_num_lines,D)
and map_sheet(A,tm_small_upleft_easting,E)
and map_sheet(A,tm_small_upleft_northing,F)
and map_sheet(A,tm_small_file,G)
and map_sheet(A,highlight_roads,H)
and map_sheet(A,pca_output_3,I)
and map_sheet(A,pca_output_2,J)
and frame_get(working_directory,path,K)
and map_sheet(A,tm_large_line_size,L)
and map_sheet(A,tm_large_pixel_size,M)
then delete [
    map_sheet(A,updated_road_segment,N),
    map_sheet(A,digitize_road_network,O)
]
add [
    digitize_roads,
    map_sheet(A,updated_road_segment,P),
    map_sheet(A,digitize_road_network,done),
    map_sheet(A,tm_small_easting_distance,Q),
    map_sheet(A,tm_small_northing_distance,R)
]

```

```

disk_to_display
::
if frame_get(database_display,database_input_channels,A)
and frame_get(image_handler_0,database_file_path,B)
then delete[]add[disk_to_display]

```

```

display_arc_coverage
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,updated_coverage_file,C)
and frame_get(working_directory,path,D)
then delete [
    host_computer_class(work_station_host,name,E),
    map_sheet(B,display_arc_coverage,F)
]
add [
    display_arc_coverage,
    host_computer_class(work_station_host,name,G),
    map_sheet(B,display_arc_coverage,done)
]

```

```

display_arc_coverage_c

```

```

::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,updated_clear_cut_coverage,C)
and frame_get(working_directory,path,D)
then delete [
    host_computer_class(work_station_host,name,E),
    map_sheet(B,display_arc_coverage_c,F)
]
add [
    display_arc_coverage_c,
    host_computer_class(work_station_host,name,G),
    map_sheet(B,display_arc_coverage_c,done)
]

```

erdascp_dem

```

::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,dem_erdas_file,B)and frame_get(working_directory,path,C)
then delete[]add[erdascp_dem,map_sheet(A,dem_pci_file,D)]

```

erdascp_lakes

```

::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,erdas_lakes_file,B)
and frame_get(working_directory,path,C)
then delete[]add[erdascp_lakes,map_sheet(A,pci_lakes_file,D)]

```

export_clear_cuts_to_arc

```

::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_file,B)
and map_sheet(A,updated_clear_cut_segment,C)
and frame_get(working_directory,path,D)
then delete []
add [
    export_clear_cuts_to_arc,
    map_sheet(A,updated_clear_cut_point_file,E),
    map_sheet(A,updated_clear_cut_vector_file,F)
]

```

export_clear_cuts_to_pci

```

::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,tm_file,C)and frame_get(working_directory,path,D)
then delete [map_sheet(B,segment_clear_cuts,E)]
add [
    export_clear_cuts_to_pci,
    map_sheet(F,roads_lin_file,G),
    map_sheet(B,clear_cut_poly_file,H),
    map_sheet(B,clear_cut_only_file,I),
    map_sheet(B,clear_cut_arc_file,J),
    map_sheet(B,segment_clear_cuts,K)
]

```

export_roads_to_arc

```

::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_file,B)
and map_sheet(A,updated_road_segment,C)
and map_sheet(A,digitize_road_network,D)
and frame_get(working_directory,path,E)
then delete []

```

```

add [
    export_roads_to_arc,
    map_sheet(A,updated_road_point_file,F),
    map_sheet(A,updated_road_vector_file,G)
]

export_roads_to_pci
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,tm_file,C)
and map_sheet(B,roads_arc_file,D)and frame_get(working_directory,path,E)
then delete [map_sheet(B,segment_roads,F)]
add [
    export_roads_to_pci,
    map_sheet(B,roads_lin_file,G),
    map_sheet(B,segment_roads,H)
]

import_clear_cuts_to_arc
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,updated_clear_cut_vector_file,C)
and map_sheet(B,updated_clear_cut_point_file,D)
and frame_get(working_directory,path,E)
then delete []
add [import_clear_cuts_to_arc,map_sheet(B,updated_clear_cut_coverage,F)]

import_dlg_to_arc
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,dlg_roads_file,C)and frame_get(working_directory,path,D)
then delete[]add[import_dlg_to_arc,map_sheet(B,roads_arc_file,E)]

import_hydrology
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,dlg_lakes_file,C)and frame_get(working_directory,path,D)
then delete []
add [
    import_hydrology,
    map_sheet(B,grid_lakes_file,E),
    map_sheet(B,erdas_lakes_file,F),
    map_sheet(B,arc_lakes_file,G)
]

import_roads_in_arc
::
if frame_get(arc_host,name,A)
and frame_get(current_map_sheet,name,B)
and map_sheet(B,updated_road_vector_file,C)
and map_sheet(B,updated_road_point_file,D)
and frame_get(working_directory,path,E)
then delete[]add[import_roads_in_arc,map_sheet(B,updated_coverage_file,F)]

invent_fly
::
if frame_get(current_map_sheet,name,A)
and frame_get(working_directory,path,B)

```

```

and map_sheet(A,tm_small_dem_channel,C)
then delete []
add [
    invent_fly,
    map_sheet(A,mas_filename,D),
    map_sheet(A,mas_blue_channel,E),
    map_sheet(A,mas_green_channel,F),
    map_sheet(A,mas_red_channel,G),
    map_sheet(A,mas_dem_channel,C)
]

```

```

isoclus_on_tm_small_c
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_file,B)
and map_sheet(A,isoclus_channel,C)and frame_get(working_directory,path,D)
then delete[]add[isoclus_on_tm_small_c]

```

```

klus_on_tm_small_c
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_file,B)and frame_get(working_directory,path,C)
then delete []
add [
    kclus_on_tm_small_c,
    map_sheet(A,kclus_channel,D),
    map_sheet(E,isoclus_channel,F)
]

```

```

lakes_to_bitmap
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,sieve_lakes_file,B)
and map_sheet(A,sieve_lakes_dboc,C)
and frame_get(working_directory,path,D)
then delete [map_sheet(A,lakes_thr_dbib,E)]
add [
    lakes_to_bitmap,
    map_sheet(A,lakes_num_lakes,F),
    map_sheet(A,lakes_thr_dbib,G)
]

```

```

pca_on_tm_small
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,dem_pci_num_pixels,B)
and map_sheet(A,dem_pci_num_lines,C)
and map_sheet(A,tm_file,D)and frame_get(working_directory,path,E)
then delete [map_sheet(A,highlight_roads,F)]
add [
    pca_on_tm_small,
    map_sheet(A,pca_output_1,G),
    map_sheet(A,pca_output_2,H),
    map_sheet(A,pca_output_3,I),
    map_sheet(A,highlight_roads,done)
]

```

```

pca_on_tm_small_c
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,dem_pci_num_pixels,B)
and map_sheet(A,dem_pci_num_lines,C)
and map_sheet(A,tm_small_file,D)and frame_get(working_directory,path,E)

```

```

then delete [map_sheet(A,highlight_roads,F)]
add [
    pca_on_tm_small_c,
    map_sheet(A,pca_output_1,G),
    map_sheet(A,pca_output_2,H),
    map_sheet(A,pca_output_3,I),
    map_sheet(A,highlight_roads,done)
]

seidam_backup::if true then delete[]add[seidam_backup]

seidam_palermo
::
if frame_get(current_user,goal,A)
then delete [
    application(reshell_application,instance_of,B),
    application(reshell_application,plan,C),
    application(reshell_application,current_agent,D)
]
add [
    seidam_palermo,
    application(reshell_application,instance_of,application),
    application(reshell_application,plan,E),
    application(reshell_application,current_agent,F)
]

set_georeference_tm_large
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_large_file,B)
and map_sheet(A,dem_no_lakes_file,C)
and frame_get(working_directory,path,D)
then delete [
    map_sheet(A,tm_large_pixel_size,E),
    map_sheet(A,tm_large_line_size,F),
    map_sheet(A,tm_upleft_northing,G),
    map_sheet(A,tm_upleft_easting,H),
    map_sheet(A,tm_loright_northing,I),
    map_sheet(A,tm_loright_easting,J),
    map_sheet(A,dem_upleft_northing,K),
    map_sheet(A,dem_upleft_easting,L),
    map_sheet(A,dem_loright_northing,M),
    map_sheet(A,dem_loright_easting,N)
]
add [
    set_georeference_tm_large,
    map_sheet(O,tm_large_pixel_size,P),
    map_sheet(Q,tm_large_line_size,R),
    map_sheet(A,tm_upleft_northing,S),
    map_sheet(A,tm_upleft_easting,T),
    map_sheet(A,tm_loright_northing,U),
    map_sheet(A,tm_loright_easting,V),
    map_sheet(A,dem_upleft_northing,W),
    map_sheet(A,dem_upleft_easting,X),
    map_sheet(A,dem_loright_northing,Y),
    map_sheet(A,dem_loright_easting,Z)
]

set_georeference_tm_small
::
if frame_get(current_map_sheet,name,A)
and frame_get(working_directory,path,B)
and map_sheet(A,tm_small_with_dem_file,C)
and map_sheet(A,dem_upleft_northing,D)
and map_sheet(A,dem_upleft_easting,E)
then delete []

```

```

add [
    set_georeference_tm_small,
    map_sheet(A,tm_small_loright_northing,F),
    map_sheet(A,tm_small_loright_easting,G),
    map_sheet(A,tm_small_upleft_easting,H),
    map_sheet(A,tm_small_upleft_northing,I),
    map_sheet(A,tm_file,C)
]

```

```

set_georeference_tm_small_tercom
::
if frame_get(current_map_sheet,name,A)
and frame_get(A,dem_upleft_easting,B)
and frame_get(A,dem_upleft_northing,C)
and map_sheet(A,dem_loright_easting,D)
and map_sheet(A,dem_loright_northing,E)
and map_sheet(A,tm_file,F)
and frame_get(working_directory,path,G)
and map_sheet(H,dem_upleft_northing,I)
and map_sheet(J,dem_upleft_easting,K)
then delete []
add [
    set_georeference_tm_small_tercom,
    map_sheet(L,tm_small_loright_northing,M),
    map_sheet(N,tm_small_loright_easting,O),
    map_sheet(H,tm_small_upleft_northing,P),
    map_sheet(J,tm_small_upleft_easting,Q)
]

```

```

sieve_isoclus
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_file,B)and frame_get(working_directory,path,C)
then delete [map_sheet(A,sieve_isoclus,D)]
add [
    sieve_isoclus,
    map_sheet(A,isoclus_channel,E),
    map_sheet(A,sieved_isoclus_channel,F),
    map_sheet(A,sieve_isoclus,done)
]

```

```

sieve_kclus
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_file,B)
and map_sheet(A,kclus_channel,C)and frame_get(working_directory,path,D)
then delete [map_sheet(A,sieve_kclus,E)]
add [
    sieve_kclus,
    map_sheet(A,sieved_kclus_channel,F),
    map_sheet(A,sieve_kclus,done)
]

```

```

sieve_lakes
::
if frame_get(current_map_sheet,name,A)
and frame_get(working_directory,path,B)and map_sheet(A,pci_lakes_file,C)
then delete []
add [
    sieve_lakes,
    map_sheet(A,sieve_lakes_sthresh,D),
    map_sheet(A,sieve_lakes_dboc,E),
    map_sheet(A,sieve_lakes_dbic,F),
    map_sheet(A,sieve_lakes_file,C)
]

```

]

tercom

```
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_large_pixel_size,B)
and map_sheet(A,tm_large_line_size,C)
and map_sheet(A,tm_upleft_northing,D)
and map_sheet(A,tm_large_file,E)
and map_sheet(A,dem_pci_file,F)
and map_sheet(A,tm_upleft_easting,G)
and frame_get(A,dem_upleft_northing,H)
and frame_get(A,dem_upleft_easting,I)
and frame_get(working_directory,path,J)
and map_sheet(A,dem_upleft_northing,K)
and map_sheet(A,dem_upleft_easting,L)
then delete [
    map_sheet(A,create_small_tm_file,M),
    map_sheet(A,copy_dem_file_into_small_tm_file,N),
    map_sheet(A,dem_pci_num_pixels,O),
    map_sheet(A,tm_small_num_pixels,P),
    map_sheet(A,dem_pci_num_lines,Q),
    map_sheet(A,tm_small_num_lines,R),
    map_sheet(A,tm_small_upleft_northing,S),
    map_sheet(A,tm_small_loright_northing,T),
    map_sheet(A,tm_small_upleft_easting,U),
    map_sheet(A,tm_small_loright_easting,V)
]
add [
    tercom,
    map_sheet(A,tm_small_num_lines,W),
    map_sheet(A,tm_small_num_pixels,X),
    map_sheet(A,tm_small_file,Y),
    map_sheet(Z,tm_small_channel_32r,AA),
    map_sheet(AB,tm_small_channel_16u,AC),
    map_sheet(AD,tm_small_channel_16s,AE),
    map_sheet(A,create_small_tm_file,done),
    map_sheet(A,copy_dem_file_into_small_tm_file,done),
    map_sheet(A,dem_pci_num_pixels,X),
    map_sheet(A,dem_pci_num_lines,W),
    map_sheet(A,tm_small_upleft_northing,AF),
    map_sheet(A,tm_small_loright_northing,AG),
    map_sheet(A,tm_small_upleft_easting,AH),
    map_sheet(A,tm_small_loright_easting,AI),
    map_sheet(A,tm_large_start_line,AJ),
    map_sheet(A,tm_large_start_pixel,AK)
]
```

threshold_isoclus

```
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,tm_small_file,B)and frame_get(working_directory,path,C)
then delete [
    map_sheet(A,last_isoclus_thr_dbib,D),
    map_sheet(A,creating_isoclus_themes,E),
    map_sheet(A,threshold_isoclus,F)
]
add [
    threshold_isoclus,
    map_sheet(A,sieved_isoclus_channel,G),
    map_sheet(A,last_isoclus_thr_dbib,H),
    map_sheet(A,creating_isoclus_themes,done),
    map_sheet(A,threshold_isoclus,done)
]
```

tm_extractor

```

::
if frame_get(tape_drive_dev,name,A)
and frame_get(tape_drive_host,name,B)
and frame_get(working_directory,path,C)
then delete [
    class(tm_class,tape_name,D),
    class(tm_class,year,E),
    tm_class(tm_image,tape_name,F),
    tm_class(tm_image,year,G)
]
add [
    tm_extractor,
    tm_class(tm_image,tape_name,H),
    tm_class(tm_image,year,I)
]

```

```

tm_tape2disk
::
if frame_get(exabyte_dribe,host,A)
and frame_get(exabyte_drive,name,B)
and frame_get(working_directory,path,C)
then delete[]add[tm_tape2disk]

```

```

trim_to_dem
::
if frame_get(current_map_sheet,name,A)
and frame_get(working_directory,path,B)
then delete [map_sheet(A,dem_file,C)]
add [
    trim_to_dem,
    map_sheet(A,source_files_copied,D),
    map_sheet(A,dem_file,E)
]

```

```

trim_to_dlg
::
if frame_get(current_map_sheet,name,A)
and map_sheet(A,source_files_copied,B)
and frame_get(working_directory,path,C)
then delete []
add [
    trim_to_dlg,
    map_sheet(A,dlg_roads_file,D),
    map_sheet(A,dlg_lakes_file,E)
]

```

```

vdyp_agent
::
if frame_get(current_map_sheet,name,A)
and frame_get(working_directory,path,B)
and frame_get(vdyp_out_var,start_age,C)
and frame_get(vdyp_out_var,finish_age,D)
and frame_get(vdyp_out_var,age_increment,E)
and frame_get(ingres_host,name,F)and frame_get(vdyp_host,name,G)
then delete []add[vdyp_agent,vdyp(H,var_list,[]),vdyp(vdyp_out_var,var_list,I)]

```

```

vdyp_display_output
::
if frame_get(current_mapsheet,name,A)
and frame_get(ingres_host,name,B)and frame_get(working_directory,path,C)
then delete [host_computer_class(work_station_host,name,D)]
add [vdyp_display_output,host_computer_class(work_station_host,name,E)]

```