1

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A Study of Artificial Intelligence Planning: A Framework for a Planning Development System

by

CARL ROTH, B.ENG.

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

Master of Engineering

**Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University**

Ottawa, Ontario

September 1989

© copyright

1989, Carl Roth

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable 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 without his/her permission.

L'auteur a accordé une licence irrévocable et 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.

ISBN 0-315-54378-7

Canada

Acceptance

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

A Study of Artificial Intelligence Planning: A

Framework for a Planning Development System

submitted by Carl Roth, B.Eng., in partial fulfilment of the requirements for the degree of Master of Engineering.

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
September 1989

Abstract

This thesis is a study of artificial intelligence planning. In doing so, a framework for a planning development system is presented to be used as an aid to evaluating the vast array of planning approaches and techniques. The thesis contains an in depth examination of planning as a field of artificial intelligence and in particular, provides a comparative survey of planning systems that have been important to the field. A Strips style planner implemented by the Author called the Puzzler is discussed as it demonstrates some planning concepts and key techniques. The framework for the planning development system presented is both modular and flexible. It's unifying concept of operators enables a variety of operator types to be invoked using a common control mechanism and global data structure. A separate constraint satisfaction mechanism allows sophisticated planning approaches to be modeled. The standardization and formalization of planning are also discussed.

Acknowledgements

I would like to thank my supervisor, Dr. Bernard Pagurek, for his support, guidance and seasoned wisdom.

I would also like to thank my wife Cathy and my son Karl (who was born during my final year in the program) for their unlimited support and patience.

Financial support was provided by the Department of National Defence.

Table of Contents

Acceptance	ii
Abstract	iii
Acknowledgements	iv
List of Figures	viii
Chapter 1 Introduction	1
1.1 Planning and Artificial Intelligence	1
1.2 Thesis Objectives	4
1.3 Thesis Outline	5
Chapter 2 Planning: A Comparative Background Survey	7
2.1 General	7
2.1.1 Linear and Non-Linear Planners	8
2.1.2 Opportunistic Planners	8
2.1.3 Skeletal, Script and Case Based Planners	9
2.2 Search	10
2.2.1 General	10
2.2.2 Problem Spaces	11
2.2.3 Brute Force Search	12
2.2.4 Heuristic Search	14
2.2.5 Abstraction Spaces	17
2.2.6 Constraint Satisfaction Problems	19
2.2.7 Planning and Search Revisited	20
2.3 Frame Problem	21
2.3.1 General	21
2.3.2 Frame Axioms	22
2.3.3 STRIPS Assumption	22
2.4 Interacting Subgoals	23
2.4.1 Interacting Subgoal Detection	25
2.4.2 Interacting Subgoal Correction	26

2.5 Planning Systems	27
2.5.1 General	27
2.5.2 General Problem Solver (GPS)	27
2.5.3 Stanford Research Institute Problem Solver (STRIPS)	29
2.5.4 Abstraction Based STRIPS (ABSTRIPS)	31
2.5.5 HACKER	33
2.5.6 INTERPLAN	34
2.5.7 Waldinger's System	35
2.5.8 WARPLAN	36
2.5.9 Nets of Action Hierarchies (NOAH)	37
2.5.10 MOLGEN	39
2.5.11 System for Interactive Planning and Execution Monitoring (SIPE)	41
2.5.12 OPM (BB1)	45
2.5.13 TWEAK	49
2.5.14 ZHUGE	52
Chapter 3 Towards Standardizing and Formalizing Planning	55
3.1 General	55
3.2 Standardizing Planning	56
3.2.1 Goal Achievement	56
3.2.2 Goal Interactions	57
3.3 Formalizing Planning	57
Chapter 4 The Puzzler: A Planner for the Puzzle World	59
4.1 Motivation	59
4.2 Problem Description	60
4.3 States in the Puzzle World Space	64
4.4 Operators in The Puzzle World	65
4.4.1 The STRIPS Style Operators	65
4.4.2 Operator Application Descriptions	66
4.4.3 Including a Remove Operator: Interesting Results	69
4.5 Implementation Details of The Puzzler	71
4.5.1 General Execution	71
4.5.2 Analyzing the Grid	72
4.6 Performance Testing and Examples	74
4.7 Limitations of the Puzzler	84

Chapter 5 Planning Techniques Demonstrated by the Puzzler	85
5.1 General	85
5.2 Complex States	86
5.3 Implementing Problem Specific Rules in a Strips Style Planner in a General Way	89
5.4 Goal Analysis	96
5.5 Intelligent Backtracking	96
Chapter 6 A Framework for a Planning Development System	98
6.1 General	98
6.2 Design Considerations	98
6.2.1 Desired Characteristics of the System	98
6.2.2 Planning Technique and Approach Options	100
6.2.3 Planning System Architecture	101
6.2.4 Control Cycles	101
6.3 Hybrid Planning System Architecture	104
6.3.1 Planning System Components	104
6.3.2 Control Structure	106
6.3.3 Global Data Structure (GDS)	112
6.3.4 Operator Structure	116
6.3.5 Constraint Satisfaction Mechanism	120
6.4 Planning Example using the Puzzle World	122
Chapter 7 Conclusions	126
References	129
Appendix A The Puzzler Source Code	133
Appendix B Example Input File	153
Appendix C Simulated Global Data Structure Entries of Problem Example	156

List of Figures

2.1 Sussman Anomaly	25
4.1 Puzzle World	61
4.2 Puzzle World Members	64
4.3 STRIPS Style Operator Descriptions	67
4.4 STRIPS Style Operator Descriptions With Remove	70
4.5 Grid Analysis Example	75
4.6 Example 1 Start State	78
4.7 Example 1 Goal State	79
4.8 Examples 2 through 5	82
4.9 Diagnostics of Examples	83
5.1 Hybrid Planning System Architecture	105
6.2 Control Operators	108
6.3 Operator Activation Record	109
6.4 Planning Node	110
6.5 Node Numbering Scheme	111
6.6 Domain Independent Operators	113
6.7 Global Data Structure Entry	114
6.8 Domain Dependent Operators	119
6.9 Operator Attributes	121
6.10 Operator Rankings	122
6.11 Planning Example	124

Chapter 1

Introduction

1.1 Planning and Artificial Intelligence

"To search is human, but to PLAN — divine!" [the Author]

In his invited talk at AAAI-88, Raj Reddy noted that search compensates for a lack of knowledge and that knowledge eliminates the need for search. Within artificial intelligence, planning is one field that encompasses both of these concepts. At one extreme planning can be accomplished with only brute force search and at the other extreme sophisticated planning techniques can be used to find a solution by dramatically reducing the amount of search required. One of the tasks of planning research can thus be thought of as to develop these techniques to a point where search no longer becomes a limiting factor. This thesis is about planning, the process of generating a sequence of actions to accomplish a goal [1].

Many problems of interest in artificial intelligence involve problems with goals that are achieved by a sequence (or program) of actions, such as robot problems [2]. It has been said that all goal-directed artificial intelligence programs are in essence problem solvers, that is, programs that develop a sequence of actions to solve a goal. Artificial intelligence can therefore be considered a collection of techniques used in problem solving [3]. Not surprisingly, many of the areas of principal concern to artificial

intelligence such as search and choice making, knowledge representation and learning are shared by planning research [4].

During the course of planning research a number of approaches to planning have emerged. These include non-hierarchical, hierarchical, skeletal, script based, case based and opportunistic planning [3]. Each of them has helped to further develop planning research and define planning as a separate field within artificial intelligence. Planning systems have been implemented in a variety of application domains including [4]:

- Robot Control (STRIPS [5]);
- Simple Program Generation (HACKER [6]);
- Mechanical Engineers apprentice supervision (NOAH [7]);
- House Building and Electric Turbine Overhaul (NONLIN [4]);
- Experiment Planning in Molecular Genetics (MOLGEN [8][9][10]);
- Journey Planning (OPM [11][12][13]);
- Job Shop Scheduling for Turbine Production (ISIS-II [14][15]);
- Aircraft Carrier Mission Sequencing (SIPE [1][16]);
- Naval Logistics (NONLIN [4]);
- Voyager Spacecraft Mission Sequencing (DEVISER [17]);

Planning researchers have been able to extract ideas from many areas and use them to incrementally improve the state of the art.

Planning while being a field within artificial intelligence also encompasses techniques that are of use to artificial intelligence in general. These techniques have been developed by examining and finding solutions (complete or otherwise) to the classic planning problems of search, interacting subgoals, and the frame problem. The benefits of planning include [3] :

- reduction of search;
- resolution of goal conflicts;
- providing a basis for error recovery (re-planning or backtracking).

These points are central to the information contained in the thesis.

This thesis is based on the premise that planning is an important component of Artificial Intelligence. In order to determine the magnitude and scope of this importance an in depth investigation of planning is presented that focuses on the techniques used in planning, the classic problems encountered in planning and the planning systems that have made contributions to the field with respect to these points. During the research phase it became apparent that planning is a complex field that uses numerous techniques implemented by numerous planning systems that is best understood by doing rather than just reading. To this end a STRIPS [18] style planner called *The Puzzler* was implemented by the Author in order to demonstrate some of the techniques and experience some of the problems. The Puzzler was also motivated by the need for a planner to provide part of the control function of a robot arm that is under development at the Systems and Computer Engineering Department of Carleton University. The physical constraints imposed by the capabilities of the target robot arm have introduced some interesting knowledge representation and planning issues. The domain of interest is that of problem solving, specifically planning the completion of jigsaw type puzzles. The problem domain selected was defined specifically so that problem difficulty could be increased incrementally. Also, the field of planning is lacking in general techniques for adding problem specific rules to STRIPS style general planners therefore it is the intention of this thesis to outline one such technique developed for use in *The Puzzler*.

Based on the investigation of planning and the experience gained by implementing

the *Puzzler* it also became clear that a planning development system that was capable of modeling the various planning techniques would help provide the experience necessary to further evaluate and develop planning techniques as they apply to the classic planning problems and Artificial Intelligence itself. A framework for a planning development system is presented to provide a tool to aid in the continuing research into the importance of planning as at present the literature does not provide the definitive answer.

1.2 Thesis Objectives

The objectives of this thesis are as follows:

1. provide an in depth examination of planning as a field of artificial intelligence and in particular, provide a comparative survey of planning systems that have been important to the field;
2. implement a planner that demonstrates some planning concepts and key techniques while providing a vehicle to obtain a better understanding of the problems associated with planning;
3. outline a technique that allows the incorporation of problem specific rules in a STRIPS style planner in a general way; and
4. investigate the use of different architectures with a view to developing a framework for a planning development system that could be the basis/environment of future planning research.

The planner implemented, *The Puzzler*, is not intended to compete with the larger systems discussed throughout this thesis but is intended to demonstrate planning concepts and techniques, including one that allows the incorporation of problem specific rules in a STRIPS style planner in a general way.

1.3 Thesis Outline

The thesis begins by providing a comparative background survey of planning in Chapter 2. The comparative survey covers a number of planning systems that have had an impact on the field of planning. The chapter begins with discussion of planning approaches and is followed by an in-depth examination of the classic planning issues, search, interacting subgoals and the frame problem. The chapter continues with a discussion of each planning system focusing on how that particular system deals with each issue.

Chapter 3 presents ideas concerning the standardizing and formalizing of planning. A collection of terms and concepts is presented to help standardize planning. The need to formalize planning is considered in light of one attempt to do so.

Chapter 4 introduces the Strips style planner developed by the author called *The Puzzler*. The areas covered include the domain of interest, representation details, the STRIPS style operators developed, and the implementation details of the planner. A discussion of examples and performance testing information is also included.

In Chapter 5, the planning techniques demonstrated by *The Puzzler* are discussed. These techniques include the use of complex states, implementing problem specific rules in a Strips style planner in a general way, goal analysis and intelligent backtracking.

Chapter 6 presents a framework for a planning development system. This framework is derived from combining a variety of techniques used in the systems compared in Chapter 2, with the goal of providing a development system that can adequately deal with the classic planning issues, yet is flexible enough to become an evolving test bed for old and new ideas.

Lastly, the conclusions are found in Chapter 7.

Chapter 2

Planning: A Comparative Background Survey

2.1 General

This chapter contains a comparative background survey of planning. The planning systems included were chosen for their relevance to the planner described in Chapter 4 (The Puzzler) and the framework for a large scale planning development system proposed in Chapter 6. Emphasis is placed on the classic planning problems and how planning programs have dealt with them. Each system will be described in general and then in detail with respect to the way in which it deals with the classic planning problems. Emphasis will also be placed on the emerging planning techniques that have been developed as a result of these works. The basic planning approaches will be presented prior to any discussion of the classic planning problems.

The classic planning problems include search, the frame problem and interacting subgoals, the latter being by far the most interesting from a planning point of view. The first problem to be discussed is search as it is a common problem within Artificial Intelligence.

2.1.1 Linear and Non-Linear Planners

Planners can be divided into two basic groups, linear and non-linear. Linear planners (GPS, STRIPS, ABSTRIPS, HACKER, WARPLAN, Waldinger's System, INTERPLAN) make the *linearity assumption* and thus solve one goal after the other until all goals have been achieved. The linearity assumption states that subgoals [goals] are independent and thus can be sequentially achieved in an arbitrary order. The success of this method rests on the need for the goals and subgoals to be sufficiently decoupled (if all goals and subgoals were decoupled there would be no interacting subgoals problem).

Non-linear planners (NOAH, MOLGEN, NONLIN, DEVISER, OPM, ISIS, SIPE, TWEAK, ZHUGE) do not make the linearity assumption but adopt a *least commitment* approach. The least commitment approach delays making any specific choices until it is absolutely necessary. Plans are represented as a partially ordered network of actions and goals in which ordering links are only added when necessary. The majority of non-linear planners are of the constraint posting variety. Several non-linear constraint posting planners are examined later in the chapter.

2.1.2 Opportunistic Planners

Opportunistic planners (OPM, MOLGEN —to a degree) are a special type of non-linear planner. They are mentioned separately as they can exhibit all of the characteristics of non-linear constraint posting planners with the exception of their unique control structures (the order in which they solve their goals). Opportunistic planners have the ability to adopt either a goal or data driven approach. A plan is developed piecewise, parts of a plan are developed independently (islands of planning actions — island driving). A current *focus of attention* is determined by the control component of the planner which is then acted upon during that planning cycle. A particular line of reasoning can be suspended

if information is lacking and then resumed when that information becomes available. Information obtained from the cycle can then (but not necessarily) be recorded in the form of constraints.

Opportunistic planners, like OPM, are also of interest to this thesis because of their suitability for implementation using a blackboard architecture. Research by Hayes-Roth also demonstrates the use of a blackboard architecture for the control component of the blackboard itself producing a uniform blackboard architecture in which domain and control problem solving behaviors are characteristically incremental and opportunistic [12].

2.1.3 Skeletal, Script and Case Based Planners

Planners of this variety search for a particular pre-written script or section of skeleton code to be added to a plan to achieve a goal. In a similar vein a planner may have knowledge about a variety of different cases which are used as a basis for comparison or a starting point for planning. In domains where the number of options is limited or of a fixed and known nature this technique may prove to be the most efficient. MOLGEN uses scripts for the actual operations involved in a particular laboratory processes. Each process is a well known set of finite laboratory steps. In general however, the domains envisioned for most planner are quite complicated and as a result the systems discussed in the next section do not include planners solely of this type.

2.2 Search

2.2.1 General

The Handbook of Artificial Intelligence [19][3] states that "In Artificial Intelligence, the terms problem solving and search refer to a large body of core ideas that deal with deduction, inference, planning, common sense reasoning, theorem proving and related processes."; planning in particular can be viewed as a problem solving by search. With this in mind limiting search becomes a critical problem due to efficiency concerns such as the cost of the solution generated, processor time required and memory space constraints. The *combinatorial explosion*¹ plays a key role in planning as the number of possible problem solving operations increases exponentially with the number of available operators [3].

Problem solving systems can be considered in terms of three components, a database, operators, and a control strategy [19][3]. With respect to control strategies Chapman notes that the top level control structure of most domain-independent conjunctive planners² is in fact search, and, search control is the area least understood [20].

Within planning the reduction of search is one of the key motivators for the introduction of new techniques. However, this is tempered by the fact that the domains used by many planners do not require critical search decisions to be made and hence not much time has been devoted to complex search control decisions [20].

The classic search problem types include path-finding problems and constraint satisfaction problems [21]. Search in planning includes examples of each of these

¹ The combinatorial explosion refers to the fact that time complexity grows exponentially with problem size.

² Conjunctive planners are those planners that attempt to achieve conjunctive goals [20]. Conjunctive goals have more than one condition to be satisfied in which order may not have been specified but which may be crucial to the solving of the overall goal.

areas. The following sections will review search as it applies to planning so that subsequent discussions of planning systems will make more sense. The information is based on Korf's excellent survey on the subject [21].

2.2.2 Problem Spaces

Search is considered a key notion of AI due to the *problem space hypothesis*³ which infers that search in a problem space is a completely general model of intelligence [21]. The problem space is of great concern to path-finding problems as they generally involve finding a sequence of operations that map an initial state to a goal state.

A problem space consists of two components, a set of states and a collection of operators. A problem to be solved is a *problem instance* which can be viewed as a problem space with an initial state and a goal state. A goal state can be characterized by explicitly giving the goal state or by giving a test for the desired final state. Since there are usually more than one operator that can be applied at any given state we are left with a path-finding problem (a search in which exactly one operator is applicable to each state can be considered an algorithm).

The standard abstraction used for the problem space is the search graph with the search tree being a special case of the search graph (the tree is generally used as it has no cycles). The performance of most search algorithms is defined in terms of two parameters, *branching factor*⁴ and *depth*⁵. Taken together these parameters can indicate the difficulty of performing search in a particular problem space.

³ A strong version of the hypothesis says that all goal-oriented symbolic activity occurs in a problem space [21].

⁴ The branching factor is the number of choices available at a given node, typically the number of operators that can be applied to produce a new state. The average branching factor is the number generally used.

⁵ The depth of the solution is the length of the shortest path in terms of the number of operator applications.

The difficulty of performing a search is also related to search efficiency which is measured along three dimensions, the cost of the solution generated in terms of arc costs, the time, and the memory required for the search. In order to achieve better performance often the addition of domain specific knowledge is required. A trade off exists between generality and power. The amount of knowledge an algorithm has can be used as a metric for their categorization, specifically, from very general (knowledge poor) to very specific (knowledge rich). In terms of planning this becomes a very useful order in which to present the different search techniques available. The types of searches that will be discussed , in increasing order of knowledge, are brute force, heuristic and various abstraction techniques.

2.2.3 Brute Force Search

A brute force search uses no knowledge about the problem other than the problem space itself. This space includes a set of states, a set of operators, an initial state and a goal state. It is very general but very inefficient.

Brute force search involves expanding (generating all of the children of) all nodes of a graph or tree until the solution path is found, success, or every branch fails, failure. There is however a third case called nontermination, which is important when considering correctness and completeness⁶, in which the space continues to increase. Brute force searches include breadth first , uniform cost, depth first, depth first iterative deepening, backward chaining, and bidirectional . Breadth first search is guaranteed to eventually find the optimal solution if any solution exists but its large memory requirement is a major drawback. The size of its problem space is proportional to the number of nodes saved and in order to generate the next level the previous level must be stored. Uniform

⁶ These points are discussed in section 2.3.3 .

cost search, also known as Dijkstra's Single-Source Shortest Path Algorithm on a graph [21], can be generalized to a breadth first search if all edges have the same weight. It is guaranteed to eventually find the cheapest solution if any solution exists but it also suffers from the same memory problems as the breadth first search. Depth first search involves expansion of the most recently generated or deepest node first. This as well as the breadth first search can be implemented using a list of unexpanded nodes with a first in first out (fifo) strategy producing breadth first and last in first out (lifo) producing depth first. The algorithm only needs to store the stack of the nodes on the path from the root to the current node. The advantage of depth first search is that the space required is proportional to the depth of the search meaning that the memory requirement is only linear in search depth as opposed to exponential for breadth first. The time complexity, or running time, is the same as it generates the same amount of nodes only in a different order. The disadvantage is that the tree does not have a natural termination. The ideal cut-off depth would be the solution depth but usually this is not known before the search begins. If the cut-off depth is not deep enough no solution is found but if it is too deep a non-optimal solution may be found first. Finding all of the solutions up to the cut-off depth and picking the best solution is one costly option available. Depth first iterative deepening is an extension to the depth first search designed by Korf, that solves the problem of choosing a cut-off depth by choosing it dynamically. Initially the cut-off chosen is very small, then it is incrementally increased until a solution is found. In essence it is doing a sequence of depth first searches and therefore inherits all of the advantages of the depth first search. With respect to search time, "regeneration of nodes doesn't affect the asymptotic performance because in an exponentially growing tree, most of the nodes are on the bottom level, ... extra work

in shallower levels doesn't affect the asymptotic complexity"⁷. It is the best brute force algorithm using the brute force assumption of no additional knowledge [21]. Backward chaining involves searching from the goal state to the initial state. It is used extensively in domains where there are relatively few operators that will imply a given statement in a single inference, ie. theorem proving. Backward chaining only works for problems with explicit goal states. When choosing between forward or backward chaining the direction with the lowest branching factor should be used if the depths are the same. Bidirectional search involves searching forward from the initial state and backward from the goal state until the searches meet in the middle [21]. In principle the searches are simultaneous and guarantee eventually finding an optimal solution if any solution exists. The time complexity is significantly less than the other brute force searches as it cuts the exponent of the search time in half but this comes at the cost of an increased memory requirement. In summary it provides a "multiplicative space-time trade-off...[and] ... for problems with machines with sufficient memory time is drastically reduced"⁸.

The limitation of these searches once again rests with the combinatorial explosion. For instance, when the *eight puzzle* is increased to the *fifteen puzzle* brute force searches are no longer adequate. In order to deal with this more knowledge must be added to reduce the complexity. An example of this process using the Puzzler is given in Chapter 4 .

2.2.4 Heuristic Search

In the literature one can find two meanings for heuristic search. Generally it can mean any advice or rule of thumb that is often effective but is not guaranteed to work in

⁷ [21] pp. 208.

⁸ [21] pp. 211.

every case. Specifically it can refer to a heuristic evaluation function. The following is a discussion of the latter.

Heuristic search algorithms add a small amount of additional domain-specific knowledge in the form of a heuristic evaluation function. This function estimates the likelihood of success or distance to the goal of a particular search choice. In this context the function takes a state as an argument and returns a number that is an estimate of the merit of that state with respect to the goal. It has two important properties, firstly, it provides a reasonable estimate such that it never over estimates and secondly, it is inexpensive to compute. Empirical results show that the trade off of computational complexity versus accuracy of heuristic functions is such that giving up a small amount of accuracy often dramatically reduces the complexity of computing an estimate [21].

Heuristic estimates can also include the probability that a node will lead to a solution. The nodes are then ordered by ratio of their probability of success to the cost of realizing it.

The main research issue remains the design of heuristic evaluation functions that effectively use search functions to reduce the time complexity of search [21]. The classic algorithm for single agent heuristic search is A*. A* is a best first search, an optimal search for an optimal solution. It is based on the following cost function: $f^*(n) = g^*(n) + h^*(n)$ such that $g^*(n)$ estimates the minimum cost of a path from the start node to node n and $h^*(n)$ estimates the minimum cost from node n to the goal node, $h^*(n)$ is the carrier of heuristic information. For any n it should hold that $h^*(n)$ is less than or equal to $h(n)$, the actual cost of an optimal path from n to a goal node. This is the admissibility condition. If $h^*(n) = h(n)$ then this is optimal and if $h^*(n) = 0$ then A* reduces to the uniform cost algorithm. In terms of implementation, a list of unexpanded

nodes, called OPEN, sorted by cost is maintained. At first it contains only the initial state but during each cycle a node in OPEN whose cost $f^*(n)$, is lowest, is chosen for expansion and is removed from OPEN. It is expanded by generating each of its children, evaluating them according to the cost function, and inserting them in the OPEN list. This continues until a goal state is chosen for expansion. If the heuristic function never overestimates the actual cost, then upon termination an optimal path to the goal will have been found [21]. It has also been shown that A* is the fastest algorithm for finding optimal solutions, for a given non-overestimating heuristic function. This means that the A* cost function is the best way to combine heuristic and other available information [21]. However, its drawback is that it causes memory to be exhausted quickly. Korf has extended the A* in his Iterative Deepening A* (IDA*). Iterative deepening A* is similar to depth first iterative deepening in that a path is cut off when its total cost $f^*(n)$ exceeds a cost threshold. As in A* if the heuristic function never overestimates the actual cost then IDA* will find an optimal solution. The advantage being that space complexity is linear in the solution depth instead of exponential. Korf states that "combining results on the time optimality of A* with the asymptotic time equivalence of IDA* allows us to conclude that for a given non-overestimating heuristic function IDA* is asymptotically optimal in time and space over all algorithms that are guaranteed to find shortest paths on an exponential tree"⁹.

These algorithms are useful in planning because the use of heuristic evaluation functions reduces the number of nodes that must be examined in order to find a solution and hence allow the planners that use them to solve larger problems within the same time constraints.

⁹ [21] pp. 215.

The advice or rule of thumb heuristics are also effectively used by many planning systems to reduce the search space. Heuristics used in the Puzzler improve the time complexity of the search noticeably. However, optimal heuristic searches only postpone the effects of exponential complexity. Once again it is necessary to add more knowledge to escape these effects.

2.2.5 Abstraction Spaces

In order to reduce the exponential complexity of a problem it is possible to plan in a simpler problem space (one derived from abstracting the origin problem space) first, and then using this abstract plan as a starting point for the more complex task of planning in the original problem space. Sacerdoti's concept of *planning in a hierarchy of abstraction spaces* [22] details an approach designed to augment the power of the heuristic search process by utilizing a means for discriminating between important information and details in the problem space. This allows planning in a hierarchy of abstraction spaces in which successive levels of detail are introduced. The resulting search problems are less complex than the original problem but can still be used to solve it. However, this simplification through abstraction comes at the price of reduced solution optimality. Even if the individual steps are locally optimal, there is no guarantee that the combination will be globally optimal. This loss of solution optimality is an unavoidable cost of reducing complexity.

As may be expected, the abstraction can involve a single or multiple levels. A single level of abstraction involves ignoring low-level details of the problem and concentrating on the essential feature, filling in details later. The performance of search using an abstract problem space depends on the density of the abstract space relative to the original problem space, performance being the time to find a solution not the cost.

The complexity of brute force search is linear in the number of states in the problem space, which is usually an exponential function of the problem size. Hence we have two boundary conditions of density for the search in the abstract space that are equivalent to that of the brute force search. The first being that one space is so sparse that in the limit it does not exist at all, therefore search occurs in the original space and the other being that a space is so dense that it becomes equal to the original space, again the original space is used. The key is to find an optimal level of detail. Mackworth contends that the optimal level of detail is for the number of states in the abstract space to be the square root of the number of states in the base space, having the effect of reducing the running time to find a solution from linear in the number of states, to on the order of the square root of the number of states in the problem space [21].

When discussing multiple hierarchical levels the question that surfaces concerns the optimal number of levels. Korf states that the optimal abstraction hierarchy has $\log n$ levels where n is the number of states in the original space and that in such a hierarchy the ratios of the number of states between successive levels is constant. The running time of problem solving is reduced to logarithmic in the number of states from linear in the number of states [21]. This leads us to Korf's conclusion that using multiple hierarchical levels of abstraction defeats the combinatorial explosion by reducing the complexity as a function of problem size from exponential to polynomial if the number of states is an exponential function of problem size. However, the majority of the planning systems discussed in this thesis determine the number of abstraction levels by doing an analysis of the problem domain in terms of human planning considerations rather than counting the number of states in the original domain space. Planners that employ abstraction levels are termed *hierarchical planners*. The process of determining what should be planned at each level is very difficult and tends to be highly domain dependent.

2.2.6 Constraint Satisfaction Problems

Constraint satisfaction problems include the n queens problem, job shop scheduling and graph coloring to name a few. Constraint satisfaction is a powerful technique that is used in a number of important planning systems¹⁰.

Typically constraint satisfaction problems consist of a set of variables, a set of values and a set of constraints on the values that the variables can be assigned. Constraints can be either unary, a constraint on a variable that specifies a subset of all possible values that can be assigned to that variable, or binary, a constraint between two variables that specifies which possible combinations of assignments to the pair of variables would satisfy the constraint. It has been suggested that the brute force approach to constraint satisfaction is backtracking and the heuristic approach is intelligent backtracking. In backtracking one selects an order for the variables and starts assigning values to the variables one at a time. Assignments are made so that all constraints involving any of the variables that have already been assigned values are satisfied. Once a constraint is violated no assignment to the remaining variables can possibly resatisfy the constraint. If a variable is reached that has no remaining legal assignments then the last variable that was assigned is reassigned to the next legal value. This algorithm continues until a consistent assignment is found or all possible assignments are shown to violate some constraint [21].

Intelligent or heuristic backtracking is backtracking with additional knowledge added. The techniques used to add this knowledge include variable ordering, value ordering, dependency directed backtracking, and constraint recording. Variable ordering is based on the premis that the order in which variables are instantiated effects the efficiency of

¹⁰ Planning systems that use constraints will be discussed later in this chapter.

backtracking. One tries to choose an order likely to cause the least backtracking, ie, instantiate the most tightly; constrained variables first. Value ordering is based on the premis that the order in which values of a given variable are chosen can significantly effect the efficiency of backtracking. One tries to choose the sequence of values for each variable that is likely to minimize backtracking. Ideally the order of values would be from most likely to succeed to least likely to succeed. Dependency directed backtracking, instead of undoing decisions in a chronological order, undoes the decision responsible for the failure. Constraint recording consists of recording any implicit constraints found so that they will not have to be found again. These constraints can be found during backtracking or during a preprocessing phase. Arc consistency is one type of preprocessing and path consistency is a generalization of arc consistency. As can be expected preprocessing reduces the search space.

Constraints are useful because they can be applied to partial assignments of variables. If a constraint is violated in the partial assignment then it follows that no complete assignment can satisfy it either. This ability to prune possible choices early on makes backtracking more efficient than trying all possible combinations. Planners that employ constraints are usually termed *constraint posting planners*.

2.2.7 Planning and Search Revisited

The fact is that problem solving involves search and planning is in effect a special case of problem solving where search is a crucial factor. One of the benefits of planning as a technique in AI is the reduction of search. This allows us to consider planning as collection of high level strategies that implement the search techniques outlined previously as its low level operators. This may seem a bit abstract for some but it emphasizes the importance of search within the field of planning.

The following sections will complete the discussion of the classic planning problems and will be followed by a review of several important planning systems which centers on how they deal with these problems.

2.3 Frame Problem

2.3.1 General

The frame problem is an important problem that effects AI in general. According to Hayes it arises in attempts to formalize the problem solving process involving interactions with a complex world [23]. Every time an action is performed some alteration is usually made to a representation of a world. The frame problem encompasses the difficulty of keeping track of the consequences of these alterations.

Shoham extends this definition by stating that the frame problem arises from the conflicting desires to reason both rigorously and efficiently about the future [24].

Chapman also realizes the usefulness of discovering which *propositions* are left unchanged when trying to determine whether or not a *proposition* holds in a *situation*. He finds the frame problem the main difficulty with extended action representations and concludes that finding an efficiently implementable *truth criterion*¹¹ is the solution, although not simple, to this problem [20].

Related to the frame problem is the qualification problem. The qualification problem is concerned with the possibility of new facts making earlier conclusions invalid and the difficulty of determining which conclusions actually depended on the changed fact.

¹¹ Chapman's Modal Truth Criterion is discussed in more detail later in Chapter 2 and in Chapter 3.

Two notable concepts adopted to defeat the frame problem which have been used in planning systems are outlined below. These are frame axioms and the strips assumption.

2.3.2 Frame Axioms

When a person moves one of many objects on a table to a new position he knows that the other objects do not also move (assuming that he does not touch any of the other objects) but a robot for instance would not know this unless the rule governing the action performed included a portion which stated that the other objects did not move. Such an addition is a frame axiom [25]. The function of the frame axiom is to allow the robot to infer that an event does not affect an assertion. As can be imagined the number of axioms required for complex domains makes their use inefficient and impractical.

Unfortunately the inferences allowed by frame axioms are necessary and therefore must be accomplished in some other way. One such way is by using the strips assumption.

2.3.3 STRIPS Assumption

Waldinger's *STRIPS assumption*¹² is based on the belief that an action leaves all of the relations in a model unchanged unless otherwise specified [26]. This assumption solves the frame problem as all consequences of an action must be explicitly stated. If an object is not mentioned by an action then the action has no effect on it. Since all relationships that hold in the domain are expressed directly systems using the strips assumption have an efficient querying ability.

One reason for insisting upon the use of the strips assumption concerns the completeness and correctness of planners. A planner is said to be complete if finds

¹² STRIPS: Stanford Research Institute Problem Solver was written by Fikes and Nilsson [5].

a solution plan (eventually) whenever one exists. A planner is said to be correct if when a plan is produced it is a solution to the goal(s) being solved for. Chapman's modal truth criterion is strongly based on this assumption. The modal truth criterion can be thought of as a completeness / soundness theorem for a version of the situational calculus¹³[20]. A procedure is sound if any well formed formula (wff) derivable from a set S actually follows from that set. Unfortunately complex domains may not be able to be represented in such a restrictive environment as the addition necessary to allow the use of indirect side-effects for instance would cause the modal truth criterion to fail, that is, it would no longer be complete and possibly not correct (the modal truth criterion will be discussed in more detail later in this thesis).

Although the strips assumption has been around for a long time it has surfaced as a key assumption in the quest to transform planning concepts into theorems.

2.4 Interacting Subgoals

The last of the classic planning problems concerns dealing with interacting subgoals (in this context subgoals includes goals). Subgoal interactions occur when the solving of one goal undoes the solution of another goal. These interactions can occur between conjuncts within the original goal or between conjuncts within subgoals generated in order to solve the original goals (for the purpose of this thesis the original goals of a planner will be called goals and any subgoals generated in order to solve these goals will be called subgoals). In both cases goal and subgoal order becomes an important detail that can not always be optimized before the problem solving begins. To complicate things further not all interactions at the top level can be solved by goal ordering. The techniques used to

¹³ The situational calculus referred to here is based on Chapman's use of situations.

solve this problem are varied but contain a common thread, Tate's Review of Knowledge Based Planning Techniques provides one taxonomy that is both complete and concise [4].

Chapman has introduced four plan modification operations that can be used to describe almost all of the methods used in the planning systems described in this thesis. These are *simple achievement*, *step addition*, *individual promotion* and *block promotion*. Simple achievement consists of determining that a goal is already true in a model. If a goal is not already true and thus has to be made true step addition can be used to insert a step in the plan (if one exists) that has making the desired goal true as one of its effects. If a step is found to change the truth value of a precondition or goal that has already been achieved the step is said to clobber the other step. Individual promotion moves the clobberer forward past the clobbered or one may move the clobbered backward over the clobberer. During this process steps can become separated from the goals that they were to achieve which may require some preconditions to be re-achieved. In order to avoid this block promotion moves the clobberer together with the steps that achieve its preconditions and the steps that achieve their preconditions transitively as a block. Unfortunately block promotion cannot solve problems that cannot be solved by goal re-ordering, such as the Sussman anomaly (see figure 2.1). In the Sussman Anomaly, if the first conjunct of the goal, $on(a,b)$ is achieved first, block c would be unstacked and placed on the table in order to clear block a so that block a could be picked up and stacked on block b. In order to now achieve the second conjunct, $on(b,c)$, the goal $on(a,b)$ would have to be undone to allow block b to be picked up and stacked on block c. If the conjuncts were reversed a similar situation would result, both blocks b and c would have to be removed from block a, undoing $on(b,c)$, so that block a could be picked up and stacked on block b. Therefore the Sussman Anomaly cannot be solved by block promotion. The subtle ability to perform one of the preconditions of solving

Sussman Anomaly

Figure 2.1 Sussman Anomaly

$\text{on}(a,b)$ namely clear a then achieving $\text{on}(b,c)$ prior to achieving the rest of the $\text{on}(a,b)$'s preconditions is required to solve this problem.

2.4.1 Interacting Subgoal Detection

In order to avoid or correct a subgoal interaction it is first necessary to detect it. This can be accomplished by prior analysis of the goals, preprocessing, or by continual monitoring during actual planning. Early linear planners ignored goal/subgoal interactions at the level of the original goals while later linear planners used procedures, sometimes termed *critics*, to detect certain known interactions (HACKER, WARPLAN, Waldinger's

System). Finally linear planners used analysis of data structures, *tick lists*, that maintained a tabular representation of the interactions between conjunctive goals and subgoals to detect interactions (INTERPLAN). The first non-linear planners also used this technique to detect interactions using a *Table Of Multiple Effects*(TOME) (NOAH). Subsequent non-linear planners that employed constraint posting were able to detect interactions during the process of attempting to post, propagate or satisfy constraints (MOLGEN, OPM, TWEAK, ZHUGE) and at least one recent non-linear planner (SIPE) uses critics to analyze a data structure similar to a TOME to detect interactions.

2.4.2 Interacting Subgoal Correction

Interacting subgoal correction can be reduced to two simple concepts [20], one for linear planners and the other for non-linear planners. Linear planners *move steps* and non-linear planners do basically the same thing by *adding constraints*. Early linear planners that ignored interactions simply tested the conjunct of goals at the end of the planning cycle to ensure that all the goals were still achieved. If one was not, it entered the planning cycle again using the current final state as the new initial state but retaining the same original goal. Linear planners that used critics (HACKER) had specific fixes associated with each critic. Other linear planners that detected interaction violations (*protection violations*) were able to move steps by *action regression* (WARPLAN) and *goal regression* (Waldinger's System). Linear planners that analyzed tick list like structures (INTERPLAN) are able to move steps more efficiently as the analysis provided less orderings to try than did the regression techniques.

Non-linear constraint posting planners (MOLGEN, OPM, SIPE, TWEAK, ZHUGE) move plan steps by constraining their temporal position in the plan. In essence all goals are worked on separately with constraints being added as required. Through the

process of constraint satisfaction the appropriate choices are made such that interacting goals/subgoals are either avoided or corrected. A more in depth discussion of how constraints are employed by non-linear planners can be found in the following discussion of the individual planning systems.

2.5 Planning Systems

2.5.1 General

The planning systems discussed in this section are listed individually rather than as a group under the heading of the particular planning issue being discussed. This is done so that the reader can quickly focus a particular system of interest. The systems discussed were chosen based on their applicability to the Puzzler and the framework for a large scale planning development system discussed later in the thesis.

2.5.2 General Problem Solver (GPS)

The General Problem Solver (GPS) was conceived by Newell, Shaw and Simon in 1957 (the latest version of GPS was written by Ernst and Newell in 1969) [27][28][19]. GPS was the product of research into the construction of a general problem solver by using what was then thought of as a general paradigm of problem solving, heuristic search. The original domain of the GPS was theorem proving in the propositional calculus but the final version which was generalized so as to accommodate other domains had changed a key property so that it was no longer able to solve problems in its original domain.

GPS was the first problem solving program to separate its general problem solving methods from knowledge specific to the type of task at hand. It employed a variety

of search related techniques including abstraction spaces, heuristic search, depth first search and constraints.

GPS operated in a single abstraction space formed by replacing all logical connectives by a single abstract symbol (the original problem space defined four logical connectives but many problem solving operators were applicable to any connective). These were treated as detail and abstracted out of the formulation of the problem. The solution was then mapped back into the four connective space.

The heuristic search process was controlled using *means-ends analysis*¹⁴. A single relevant operator was selected at each stage to try to reduce the difference between the current state and the desired state. This would continue depth first as long as chosen operators were applicable and the path shows promise. GPS would attempt to change the current state into a state appropriate as input to the chosen operator. The result was a recursive goal directed program that records the search history in a AND/OR graph with partial development of nodes.

In order to allow GPS to operate on other domains, the representation of objects had to be generalized. To accomplish this the description of desired objects by a list of constraints was introduced.

With respect to the frame problem every action was assumed capable of destroying any relation, therefore frame axioms were required [26].

GPS only operated on problems with a single top goal and thus did not have the ability to deal with subgoal interactions. Subgoals were generated as required but only

¹⁴ Means-ends analysis: selecting the means to be used with a view towards the end to be reached, ie. determine a difference between the current state and the desired state and then choose an operator that might reduce the difference. Differences were defined and classified and operators were classified according to what kinds of differences they might reduce. A domain independent data structure called the Table of Connectives listed the operators relevant to reducing each difference.

simple goal selection was done by the problem solving executive. The problem solving executive had no fixed search strategy built in. Goals were attempted iteratively by working on one goal until deciding to abandon it in order to work on another goal. GPS did have the ability to prevent looping.

2.5.3 Stanford Research Institute Problem Solver (STRIPS)

The Stanford Research Institute Problem Solver (STRIPS) was created by Fikes and Nilsson [5][2][6]. STRIPS attempts to find a sequence of operators in a space of world models to transform a given initial world model in which a given goal formula can be proven true. A world model is represented by an arbitrary collection of first order predicate calculus formulas. It employs a resolution theorem prover¹⁵ to answer questions of particular models and uses means-ends analysis to guide it to the desired goal satisfying model.

The operators used to change from one state to another consist of three parts, a *precondition* formula, a *delete* list, and an *add* formula. The precondition formula is like the antecedent of an implication rule, it is a predicate calculus expression that must logically follow from facts in the state description in order for the operator to be applicable to that state description (model). The delete list is composed of literals (possibly containing free variables) that upon application of the associated operator, are matched with the current state description, and those ground instances obtained are deleted from the old state description as the first step in constructing the new state description. The add formula is a conjunction of literals (possibly containing free variables) that upon application of the delete list, are matched with the current state description and are added to the old state description as the final step in creating a new state description.

¹⁵ The particular type of resolution theorem prover used is not important. For those interested a good discussion of the various types appears in [29].

STRIPS searches a space of world models using a GPS like means-ends analysis strategy. The combination of means-end analysis and formal theorem proving provides a more powerful search heuristic than those found in theorem proving programs alone. STRIPS uses the incomplete proof as the basis for finding differences. The add formula is checked to see if reduces the difference by asserting the literal that has been selected from the uncompleted proof. A heuristic evaluation function is then used to choose between the successful operators. The function considers the number of remaining goals on the goal list, number and types of predicates in the remaining goal formulas and the complexity of the difference attached to the node.

STRIPS overcomes the frame problem by using the precondition / delete / add framework for its operators and by making the STRIPS assumption. STRIPS effectively separates the process of theorem proving from that of searching through a space of world models. Theorem proving is used only with a particular world model and the STRIPS assumption assures (as long as the operators are defined properly) that the world models are in fact complete.

Interacting subgoals are ignored by STRIPS at the level of the original goals. Tests are done during planning to ensure that a particular subgoal is solved properly. After the last goal has been achieved STRIPS checks to ensure that all of the goals are still achieved. If one fails, it backtracks, trying to solve that goal based on the current state description. This often results in repetitive and redundant plans. It is also possible for STRIPS to loop endlessly.

The STRIPS planner, including the STRIPS assumption, and the STRIPS operator convention have played an important part in planning development and according to Chapman, among others, are still very useful and necessary.

2.5.4 Abstraction Based STRIPS (ABSTRIPS)

Abstraction Based STRIPS is a modification of STRIPS that can define an abstraction space hierarchy from the STRIPS representation of a problem space [22]. It uses a hierarchy in solving the problem.

ABSTRIPS combines abstraction spaces and heuristic search. The abstraction approach is designed to augment the power of heuristic search. In ABSTRIPS the precondition wffs in an abstraction space will have fewer literals than in the ground space. The literals omitted will be those that are details in the sense that a simple plan can be found to achieve them once the more critical literals have been achieved. In order to create the abstraction hierarchies each literal is assigned a criticality. This can be done in two ways, manually, representing the definer's intuition about the domain, or automatically, an exhaustive analysis of $n!$ orderings of n literals in order to determine which literals can be achieved once other literals are assumed to be true. In ABSTRIPS, "... a predetermined (partial) ordering of all the predicates used in describing the problem domain was used to specify an order for examining the literals of the precondition wffs of all the operators in the domain"¹⁶. This ordering is achieved using the criticality values that have been assigned. All literals whose truth values could not be changed by an operator in the domain are assigned a maximum criticality value. Each remaining literal is examined in an order to determined by the partial ordering; if a short plan could be found to achieve a literal from a state in which all previously processed literals were assumed to be true then the literal in question was said to be a detail and assigned a criticality equal to its rank in the partial ordering; if no such plan could be found, the literal was assigned a criticality greater than the highest rank in the partial order.

¹⁶ [22] pp. 120.

The executive program of ABSTRIPS is recursive and requires two inputs, a criticality value indicating the abstraction space in which planning is to occur and a list of nodes from the search tree in the higher tree representing a skeleton plan. The search strategy is *length-first*. Length-first search pushes the planning process in each abstraction space all the way to the original goal state before beginning in a lower space. Subproblem failure causes a node in the higher space to be pruned. The major problem with this automatic backtracking failure mechanism is that when a failure of a lower level process is reported the process and the context in which the failure occurred are no longer around for analysis. This causes ABSTRIPS to rely heavily on being able to produce good plans at the highest level.

ABSTRIPS makes two other modifications to STRIPS search. The heuristic evaluation function has itself been made a function of the level of abstraction. At the highest level ABSTRIPS places equal weight to the cost of reaching a given node and to the estimated cost of reaching the goal from that node (STRIPS emphasizes the estimated cost of achieving a goal from the given node and de-emphasizes the cost of arriving at the node from the initial state). This changes incrementally as the level of abstraction decreases until it reaches the old STRIPS function at the level of the problem space. The second modification concerns postponing the selection of one among several equivalent instances of a relevant operator. ABSTRIPS defers the decision and uses a partially instantiated operator until the instantiation is determined. If the selection should fail then the other instantiations are tried by using the backtracking mechanism.

Since the ground space representation is the same as for STRIPS the frame problem is handled in the same way. See 2.5.3 for details.

ABSTRIPS like STRIPS ignores interacting subgoals at the top level and uses backtracking to solve interaction problems.

2.5.5 HACKER

HACKER is a learning system developed by Gerald Sussman to model the process of acquiring programming skills [3]. HACKER develops plans and simulates their execution. It is a complex program that includes a planner (develops plans by pattern directed expansion of planning operators), a critics' gallery (inspects plans for known generalized bugs), a simulator (simulates execution of the plans and checks for errors), a debugger and generalizer (locates and repairs bugs in the plans for later use by critics' gallery) and a generalizer and subroutiner (generalizes plans and installs them in knowledge base).

HACKER contains a simple problem-reduction planner that when given an initial state and a goal attempts to match the goal against its knowledge base of known plans, subroutines and refinement rules. If no plan is found a refinement rule is applied that reformulates the goal as a set of subgoals. Expansion into subgoals continues until an existing plan is found or primitive operators are found that can achieve each of the subgoals. As mentioned previously the search is done by pattern matching.

After HACKER has developed a naive plan it is criticized by the critic's gallery. If a *bug* is detected a fix is applied. As may be expected a key source of bugs is the linearity assumption and as a result HACKER has four bug types related to interaction problems. These are *prerequisite missing*, *prerequisite clobbers brother goal*, *prerequisite conflict brother*, and *strategy conflict brothers*. Prerequisite missing involves adding a step to the plan to make a precondition required by another step true. Prerequisite clobbers brother

goal involves moving a step that has made some other subgoal false to a position that no longer adversely affects other subgoals. Prerequisite conflict brothers involves replacing two steps that adversely affect each others preconditions with a single non-offensive step. Strategy conflict brothers involves a step in one strategy that affects the precondition of another later strategy. Each of these has a specific fix associated with it. Unfortunately HACKER is only able to reorder goals at one level and therefore moving prerequisites of one subgoal to a position in the plan related to another cannot be done.

2.5.6 INTERPLAN

INTERPLAN was developed by Austin Tate and introduced the concept of promotion of preconditions when a *holding period* is broken by a particular precondition [30]. It performs a search for a correct linear ordering by using both debugging and backtracking. It does not create alternate sequences of actions [7].

In order or to deal with interacting subgoals INTERPLAN examines a tabular representation of the interactions between conjunctive goals called a *tick list*. The tick list is a declarative representation to allow protection violations to be detected easily and give the system the relevant information for recovery. Also INTERPLAN abstracted its original goals to determine holding periods over which they could be assumed true. These periods were analyzed with a view toward moving goals around to ease interaction problems [8]. Chapman noted that INTERPLAN employs three main plan modification operations, step addition, individual promotion and block promotion [20].

INTERPLAN demonstrated that a planner can perform reasoning about plans by dealing with information that is much simpler than the plan itself. This concept was used extensively in NOAH.

2.5.7 Waldinger's System

Waldinger's system features two important concepts, that of goal regression and the skeleton model [26]. The system also incorporates Sussman's concept of protection.

Given a relation P and an action F , there is no guarantee that P will be true after performing F . We can find a P' that after executing F guarantees P will be true afterwards. We will demand that P' be the weakest relation that ensures the subsequent truth of P . We call P' the result of passing P back over F . The operation of passing P back is regression. Also, in order to achieve P after executing F_n , it suffices to achieve P' immediately before executing F_n , where P' is the result of passing P back over F_n . Similarly, it suffices to achieve P'' immediately before executing F_{n-1} , where P'' is the result of passing P' back over F_{n-1} . The information required to accomplish this can be given in a set of regression rules which can be regarded as part of the definition of the plan steps.

The skeleton model is a technique in which the partial plan that has been constructed is regarded as a central part of the model. Important relationships and the plan itself are contained in the model explicitly while other relationships can be inferred using the regression rules. Waldinger's regression technique provided a significant means of reducing search.

The frame problem is handled in part by a STRIPS like assumption that is made about regression rules that states that if no information exists as to how to pass a relation back over a plan step it is assumed that the plan step has absolutely no effect on the relation. This reduces the number of rules required as well.

Planners that use the pattern directed function invocation technique can be aware of only one goal at a time whereas regression allows multiple goals to be considered at

one time thus aiding in solving interaction problems. Regression allows a goal that is difficult or impossible to achieve after F has been executed to be moved to an earlier point in the plan where it may be easier to achieve. If achieving P after executing F makes P false, it is possible that planning to achieve P' or P'' earlier will not disturb P at all. This technique will not always prevent goal reordering but will allow solving of some problems that cannot readily be solved regardless of the order in which the goals are attempted like the Sussman anomaly.

The use of skeleton models means that if a relation P is protected at the end of a plan, no modification can be made at any point in the plan that will not leave P true at the end, because, in checking the truth of P, P is percolated back up through the plan, and the interaction between P and the new plan step will be discovered.

2.5.8 WARPLAN

WARPLAN was developed by Warren and employs a similar skeleton model as Waldinger's System [26]. The key feature of WARPLAN is its use of action regression. WARPLAN represents plans as flat orders without a hierarchy and restricts itself to an atomic precondition / delete / add list representation for operators. This in effect implies that the STRIPS assumption was used also.

WARPLAN uses an identical strategy as Waldinger for handling goals which in turn reduces the search space significantly. It deals with interacting subgoals primarily through its use of action regression and addition. It does not make a strong linearity assumption and can solve the Sussman anomaly.

2.5.9 Nets of Action Hierarchies (NOAH)

The Nets of Action Hierarchies (NOAH), developed by Earl Sacerdoti, was a problem solving and execution monitoring system that uses a non-linear representation of a plan [7][22][3]. NOAH employs an information structure called a *procedural net* that represents a plan as a partial ordering of actions with respect to time. This system handles difficult problems by avoiding premature commitment to an ordering.

The planner begins with an oversimplified plan treating goals as parallel and independent. Goals are expanded by adding more detailed subplans in a hierarchical fashion until the most detailed level is reached. At each level the global plan is analyzed by *constructive critics* to determine if there are any potential conflicts as individual subplans for each node (in the procedural net) will be correct but there is no guarantee that the whole plan is correct. The conflicts are resolved by adding linear constraints to an as yet unconstrained plan. This expansion / analysis cycle continues until a completely ordered plan is produced or failure results.

The key search techniques employed by NOAH are the use of abstraction spaces and heuristic search. The hierarchical nature of search in NOAH is based upon the building of the procedural net. NOAH begins with a single abstract goal node which is then expanded by searching for applicable *soup* (Semantics of User Problem) functions. The hierarchy of actions (operators) used allows all of the advantages of abstraction spaces previously discussed. The critics used are based on heuristic planning knowledge and as such are extremely useful in resolving conflicts.

NOAH deals with the frame problem in a way similar to that of STRIPS like planners. Associated with each node in the procedural net is an add and delete list. These are

computed when the node is created and contain symbolic expressions representing the changes to the world model caused by the action that the node represents. In this sense NOAH makes the STRIPS assumption (see 2.5.3).

NOAH assumes conjunctive goals to be independent and additive. Its non-linear representation allows it to apply critics to constructively linearize the plan only when necessary, waiting until it knows the nature of the interactions. The critics used in NOAH are general in nature however specific critics may also be specified. The general critics include *resolve conflicts*, *use existing objects*, and *eliminate redundant preconditions*.

The resolve conflicts critic uses a table of multiple effects (TOME) to detect conflicts between the add and delete lists of the operators concerned. The TOME contains an entry for each expression that was asserted or denied by more than one node in the current plan. A conflict is recognized when an expression that is asserted at some node is denied at a node that is not the asserting node's subgoal. The conflicts are resolved by constraining subgoals to be achieved at different times in the plan (earlier or later).

The use existing objects critic is a type of resource manager. Complete plans must specify the objects that the actions are to manipulate. During planning NOAH will avoid binding a variable to a specific object unless a clear best choice for the binding is available. If this is not the case a generic object called a formal object is generated as a place holder. After a plan has been completed at some level it may become apparent that a the formal object can be replaced by an object used somewhere else in the plan. This replacement function is the job of the critic and may involve merging nodes from different parts of the plan resulting in reordering or partial linearization.

The eliminate redundant preconditions critic is used to remove preconditions that

appear in more than one subgoal. This situation arises because NOAH does not check to see if a subgoal has already been achieved before adding it to the net. Subgoal querying is referred to as simple establishment by Chapman. This is done by examining the TOME.

2.5.10 MOLGEN

MOLGEN was developed by Mark Stefik to plan genetics experiments [8][9][10]. It is a non-linear constraint posting hierarchical planner which employs meta-planning. The hierarchical approach used focuses on the use and interpretation of constraints.

MOLGEN uses various search techniques including abstraction spaces, constraints and heuristics. It compares goals, finds differences and chooses operators to reduce these differences. During each cycle the highest level GPS like difference is identified for reduction, planning is accomplished in a series of cycles until no more differences exist.

Constraints in MOLGEN are used in three ways, as elimination rules (from the perspective of object selection, they constrain the set of allowable bindings), as partial descriptions and commitments (from the perspective of plan refinement a constraint is a partial description of an object that allows MOLGEN to make commitments about partial descriptions of the objects without making specific selections), and as a communications medium for expressing interactions between subproblems (constraints represent an intended relationship between variables, possibly uninstantiated, that are often shared among various subproblems). MOLGEN has the capability to add constraints dynamically unlike previous planners that were restricted to a set of initial constraints. Constraint posting has the advantage of allowing portions of the search tree to be pruned early as constraints from one subproblem are propagated to other subproblems to be used

as initial constraints. They also allow a least commitment strategy to be used.

MOLGEN uses three abstraction spaces and an elementary interpreter to accomplish meta-planning. The lowest or most detailed space is the *domain space*, it contains the actual laboratory operators. The next space is the *design space*, it contains knowledge about designing plans. It defines a set of operators for sketching plans abstractly. Design operators plan by creating and scheduling laboratory steps. In this space planning is viewed as operations on constraints. The *strategy space* is next and contains knowledge about planning strategy. It uses two problem solving approaches, least commitment and heuristic. Strategy operators plan by creating and scheduling design steps. This is done by the *focus operator* which invokes the design operators to search the plan for areas where they would be applicable and then placing these design steps in an agenda. A design step is then chosen from the agenda for execution. Focus iterates through the agenda until it has to invoke the design operators to search the plan again. Steps may be suspended if under-constrained and then resumed when new information is received. Steps can also be undone if they are over-constrained, at which time backtracking is invoked.

When the focus and resume operators have nothing to do the heuristic *guess operator* is used. Thus when the least commitment strategy based on constraints is unable to proceed a heuristic rule is used to allow planning to proceed.

One of the limitations of MOLGEN rest with its solution to the frame problem. In order to predict the results of applying an operator MOLGEN uses procedures that manipulate symbolic representations directly, but the logical interpretation of these representations is not represented explicitly in the knowledge base. The procedures that describe operators combine assertional and operational information. To protect the integrity of the knowledge base MOLGEN requires multiple representations of knowledge

and several distinct procedures are necessary to represent information about the operators. Flexibility to deal with complex domains is gained but simplicity and ease of operator design and modification is lost.

Interacting subgoals are handled by MOLGEN's use of the least commitment strategy and its ability to propagate constraints. When constraints are propagated across partial solutions to other subproblems the planner can anticipate interactions and avoid the generation of fruitless branches by early pruning.

Since MOLGEN was written for a specific domain that was well suited to its meta-planning mechanism it is difficult to predict how it would perform on a problem of interacting goals in other domains. MOLGEN's use of constraint posting is knowledge intensive to the point where missing knowledge about constraint formulation causes the genetics to be modeled incorrectly, leading to experiments that do not work.

2.5.11 System for Interactive Planning and Execution Monitoring (SIPE)

The System for Interactive Planning and Execution Monitoring (SIPE) is a non-linear constraint posting hierarchical planner developed by Wilkins [1]. It includes a general constraint language, the ability to reason about resources, deductive operators, a mechanism for representing plan rationale and a representation scheme that combines a frame based hierarchy with a subset of the predicate calculus. SIPE's interactive feature supports breadth first planning, depth first planning and the construction of planning islands (island driving — opportunistic type planning) that can later be linked together.

Constraints in SIPE are based on those used in MOLGEN but enhancements have been made. The ability to reason about resources (a key contribution of SIPE) aids in

the axiomatization and representation of domains. In SIPE one does not have to list the availability of a resource in the domain operators as a precondition¹⁷. The plan rationale mechanism allows SIPE to represent knowledge on how to achieve goals. Lastly, a restricted form of first order predicate calculus is used to represent properties of domain objects and the relationships among them that may change with the performance of actions (ie. goals and preconditions).

SIPE uses a number of techniques to perform search, such as abstraction levels, a resource mechanism, a plan rationale mechanism, a representation termed a sort hierarchy, critics and a general technique used by each of these techniques to assimilate this search control information, constraints. When automatic search is needed SIPE uses simple depth first search to a given depth without using knowledge for making choices or backtracking. The interactive nature of SIPE also allows the operator to inject knowledge that can be used to prune the search space.

The operators in SIPE are hierarchical, some operators are used to expand the hierarchy to more detailed levels and others concerns actions or goals applicable to a particular level of abstraction. Operators in SIPE contain *effects*, *preconditions*, *purposes*, and *goals*. A precondition in SIPE is a test for the applicability of an operator, conditions that the planner should make true are goals. This distinction is useful for connecting different levels in the hierarchy, a precondition can specify that certain high level goals must be true whereas an operation itself specifies goals at a more detailed level. It is also important for deductive operators.

The resource mechanism in SIPE is a heuristic used to prune the search space. It

¹⁷ For example, in SIPE the precondition of a block being available, *available(B)*, does not appear as a precondition of a pickup operator, *pickup(B)* because the block is considered a resource of the pickup operator. The built in resource mechanism manages the availability of resources.

is a formalism for representing operators that includes a means for specifying that some of the variables associated with an action or goal will actually serve as resources for the action or goal. The declaration of resources associated with an action is one way of saying that one precondition of the action is that the resource be available.

The plan rationale mechanism in SIPE enables the system to determine the following, how long a condition must be maintained, what changes in the world cause problems in the plan and what the relationship is among different levels in the hierarchy. This is important information that is used to aid the search process.

Search in SIPE is also aided through the use of a sort hierarchy. Domain objects and their invariant properties are represented by nodes linked in a hierarchy. This allows a frame base system to be used, and thus nodes have attributes and can inherit properties. In fact all parts of the system are represented as nodes in the hierarchy but the invariant properties are represented separately.

As in NOAH critics are used by SIPE to detect and correct interactions. Their detection and correction plays a key role in reducing the search space of a solution. The critics employ heuristic rules to accomplish their task. They will be discussed in more detail in the section on interacting subgoals.

A key feature of SIPE is its use of constraints. SIPE has extended the constraint posting approach used in MOLGEN by providing a general constraint language for expressing constraints on variable bindings so that they can be encoded as part of the operator. However, this adds considerably to the complexity of the planner as the constraints interact with all parts of the planner.

Generally SIPE deals with the frame problem by using the STRIPS assumption. The

closed world assumption is also made, that is, any predicate is assumed false unless the unnegated form of the predicate is explicitly given in the model or in the effects of an action that has been performed. There are however some reservations connected with this claim. Specifically the treatment of SIPE's deductive operators is said by Chapman to be incomplete and not generally correct. Chapman qualifies his observation by stating "This is reasonable, as SIPE represents the state of the art in engineering, rather than a formal theory of planning"¹⁸. Since the deductive operators add knowledge to the hierarchy it is not certain that their impact will not adversely affect those assumptions made to combat the frame problem.

SIPE uses a combination of general and specific techniques to deal with interacting subgoals. Constraints and the least commitment approach are used to generate correct parallel plans and SIPE's resource mechanism is used to facilitate early detection of problematic interactions. The resource heuristics used replace the need for a TOME like analysis in most cases. Those cases that are not covered by the heuristic rules are handle with a TOME like analysis. SIPE's plan rationale is also used with interactions as it allows the system to recognize if a predicate is not relevant to the plan (extraneous side-effect), if it must be kept permanently true (purpose of the plan), or if it must be kept only temporarily true (precondition for later achievement of a purpose).

SIPE uses a general approach to parallel actions as its overall strategy to combat interacting subgoals. The approach involves four steps: keep as much parallelism as possible, detect and respond to interactions between parallel branches, correct harmful interactions and take advantage of helpful interactions. An interaction is defined to occur if two branches of a plan are in parallel and a goal that is trying to be achieved in

¹⁸ [20] pp. 351.

one branch (at any level in the hierarchy) is made either true or false by an action in another branch. The detection is done by NOAH like critics that are based on interaction heuristics. Interaction detection requires that a goal be involved. This policy attempts to eliminate interactions that do not effect the outcome of the plan. A serious drawback of this policy is that the domain must be encoded so that all important relationships are represented as goals at some level. Wilkins feels that this is reasonable in SIPE.

In order for SIPE to take advantage of helpful interactions the planner must decide whether or not to add ordering constraints to the plan as in some cases these constraints may hinder the plan more than help it. This is a difficult decision for a planner to make and in SIPE a heuristic is used that does not shift actions between two parallel branches of a plan. In fact, the problem must be solvable by leaving things in parallel or placing some number of entire parallel branches sequentially before the others. This, however, prevents some elegant solutions from being found. Wilkins states that this is done for efficiency.

2.5.12 OPM (BB1)

OPM is a blackboard¹⁹ based planner developed by Barbara Hayes-Roth. Her technique of using a blackboard architecture for control was later packaged in a generic blackboard system called BB1 [11][12]. Both are built around features of Hearsay II [33].

OPM treats problem solving as an incremental, opportunistic process of assembling a satisfactory configuration of solution elements. A key feature is its use of a blackboard architecture for control. The blackboard architecture for control explicitly represents domain and control problems, knowledge and solutions. It integrates domain and control

¹⁹ The blackboard model of problem solving is a highly structured special case of opportunistic problem solving. It is usually described as consisting of the following major components: knowledge sources, the blackboard data base and control.[31][32][13].

problem solving in simple basic control loop and articulates, interprets and modifies representations of its own knowledge and behavior. This architecture also has the ability to adapt its problem solving knowledge, its application of that knowledge and its basic control loop to dynamic problem solving situations.

Three basic assumptions are generally made with respect to blackboard systems, the first being that all solution elements generated during problem solving are recorded in a structured, global database called the blackboard. This blackboard has two axes, solution intervals and levels of abstraction. The second assumption is that solution elements are generated and recorded by independent processes called knowledge sources (KS). KS have a condition — action format and their activity is event driven. Each event or change on the blackboard can trigger other KSs. Each triggering produces a Knowledge Source Activation Record (KSAR), which is similar to an item on a task agenda, that represents a unique triggering of a particular KS by a particular blackboard event. When a KSAR is chosen by the scheduling mechanism the KS's action is executed in context of the triggering information. Thirdly, on each problem solving cycle the scheduling mechanism chooses a single KSAR.

This architecture is very flexible as the KSs can apply various inference methods, strategies etc. The key however is the control or scheduling mechanism. Usually these systems employ a sophisticated scheduler. The blackboard architecture for control is an alternative to sophisticated schedulers and extends the above assumptions as follows. The blackboard control architecture defines explicit domain and control blackboards. The control blackboard solution intervals, levels of abstraction and vocabulary are domain-independent and are defined by the architecture. The blackboard control architecture defines explicit domain and control KSs. Some of the KSs are domain-specific and some

are domain-independent but all are represented as data structures that are themselves available for interpretation and modification. The blackboard control architecture defines a simple and adaptive scheduling mechanism to manage both domain and control KSARs.

The basic control KS iterates a three step problem solving cycle:

1. enumerate pending KSAR
2. choose one KSAR
3. execute chosen KSAR's action

It has no specific control knowledge but simply adapts to dynamic solution state information recorded on the domain and control blackboards. In essence the control blackboard dynamically creates a control plan using the heuristics supplied in its KSs.

In addition to opportunistic search, OPM uses abstraction spaces and heuristics to reduce the search space. The key element in OPM's search space control is its problem solving control mechanism, the blackboard architecture for control.

The ability to tailor KSs to any method or strategy allows the blackboard architecture to simulate almost any search control technique. Other blackboard control architectures²⁰ include sophisticated scheduling, solution based focusing and meta-level architectures can also accomplish this but in general they must dedicate their code to a particular strategy. The sophisticated scheduler is a complex program that embodies all of the systems control knowledge. It is used in basic blackboard systems but it does not plan its control strategy and is unmodifiable. Solution based focusing is used in many blackboards and is similar to a sophisticated scheduler except that it does not identify or select among pending KSARs, instead it sequentially selects specific blackboard events and executes KSs triggered by each one. In some implementations all triggered KSs execute in a

²⁰ Comparison information concerning these architectures was taken from [12].

pre-determined order and in others the focusing program uses other aspects of the current solution to determine which subset of triggered KSs to execute and in which order. Hayes-Roth concludes that this method is adequate for some applications but that it still lacks the flexibility of the blackboard architecture for control. Finally in terms of meta-level architectures, Hayes-Roth notes that for blackboard control architectures meta-level is not a defining theoretical construct but simply another potentially useful attribute of KSARs and the decisions they generate (which are available for consideration during scheduling).

OPM's control and domain blackboards have two axis, one of which is levels of abstraction. To illustrate this we will consider the levels of abstraction of the control blackboard of BB1. The most abstract level is the *Problem Level* which contains the problem the system has decided to solve. The next most detailed level is the *Strategy Level* which contains the general sequential plan for solving the problem. Next is the *Focus Level* which contains the local (temporary) problem solving goals. This is followed by the *Policy Level* which contains the global (permanent) scheduling criteria. Next is the *To-Do-Set Level* which contains the sets of pending KSARs, and lastly the most detailed level is the *Chosen-Action Level* which contains the KSARs chosen to execute. On the other end of the scale we have the *Plan* plane of the domain blackboard which like the other domain blackboard planes has four levels of abstraction. This plane indicates actions that the planner intends to take in the world. The most abstract level is the *outcomes level* which indicates what the planner intends to accomplish by executing the finished plan. The next more detailed level is the *designs level* which characterizes the general approach by which the planner intends to achieve the outcomes. Next is the *procedures level* which specifies the specific sequences of actions. Lastly, the most detailed level is the *operations level* which specifies the sequences of more specific actions.

Heuristics information is used by way of the KS. If a particular KS is triggered and executed its heuristic information or deduction is recorded on the appropriate blackboard planes.

The key to solving the frame problem lies in the proper representation of KS actions and management of information on the blackboard. It seems that if required the STRIPS assumption can be made even though it might restrict the expressibility of the domain representation.

OPM and BB1 avoid interactions partially through the use of the search control method mentioned previously and partly through their architecture's ability to incorporate KS that are based on heuristics that are useful for combating the problem of interacting subgoals. Although this point is not dealt with specifically by Hayes-Roth it is apparent that one of the thrusts of the work concerns the flexibility and modifiability of the system to such a point that any of the heuristic methods (ie. Wilkins' Resources) could be encoded as a KS and used to guide the solution.

2.5.13 TWEAK

TWEAK, developed by Chapman, is a non-linear constraint posting planner that is based on a simple and precise algorithm proven to be complete²¹ and correct [20]. Chapman has done a rigorous mathematical reconstruction of previous non-linear planners that has resulted in TWEAK.

TWEAK comprises three layers, plan representation, a way to make a plan achieve a goal and a top level control structure. The first being the most complicated, where the basic operation provided by the representation determines whether a proposition will

²¹ Complete in the sense that all branches of the search tree will be traversed. If a solution exists it will be found.

be true in the world after a part of the plan has been executed. In order to provide an efficient algorithm for this operation it is necessary to depend on a subtle theorem about incomplete plans called the *Modal Truth Criterion* which will be discussed in more detail in Chapter 3. The next layer is accomplished by way of a non-deterministic procedure that is used to transform a plan so that it achieves a goal that it previously did not. This non-determinism is controlled by the top level control structure which uses backtracking search to provide the completeness of TWEAK.

Planning in TWEAK is based on an incomplete plan, a partial specification of a plan that may solve the problem. Depending on the constraints that are added, this partial plan can be completed in a number of ways and therefore represents a class of incomplete plans. An incomplete plan supplies partial knowledge of the complete plan that will eventually be chosen. Planning is finished when all the completions of the incomplete plan solve the given problem. A complete plan is a total order on a finite set of steps, with order representing time and steps actions.

The action representation in TWEAK is simple but restrictive. It does not allow the effects of actions to depend on the situation in which they are applied and all changes made by an action must be explicitly represented as postconditions. These restrictions are made to preserve the correctness of TWEAK.

TWEAK uses a number of search techniques including dependency-directed breadth first search, backtracking, heuristics and constraints. Dependency-directed breadth first search is the basic method used to find goals that are to be achieved and to select the means to do so. Backtracking is used on failure to try alternate solution paths. A heuristic called the *Polynomial Time Algorithm* is used to reduce the search space of possible completions of plans by computing the possible and necessary properties of an

incomplete plan. Constraint posting is also used to prune the search space. In effect parts of the space are progressively removed from consideration. This reduction of arbitrary choice reduces backtracking and allows a least commitment approach to be used.

A basic premise of TWEAK is that all changes made by an action must be explicitly represented as postconditions and as a result adopts an add / delete list type of representation. This in effect embeds the STRIPS assumption in TWEAK. This is not surprising as TWEAK is both complete and correct. Chapman also states that the frame problem can be solved by finding an efficient implementable truth criterion, which he has accomplished.

Interacting subgoals are generally avoided by the use of the search techniques mentioned above but the key to their detection and correction in TWEAK lies in the non-deterministic achievement procedure used. This procedure is used to add the appropriate constraint to the system. It handles the achievement of subgoals through *establishment* and *declobbering*. Establishment is concerned with ensuring that a particular proposition is true when required. This is done by determining (querying) its truth value (simple establishment) and if the truth value is false by making it true by adding a step to the plan that will make it true (step addition). Declobbering is concerned with the correction of interactions and is comprised of *promotion*, *separation* and *demotion* or declobbering by *white knight*. Promotion is accomplished by adding temporal constraints that in effect move the steps that were effected by the clobberer to a position in time before the clobbering step occurs. Separation adds a constraint that ensures that a proposition q does not codesignate²² with P . This works because a step possibly denies P by denying a proposition q which codesignates with P . Demotion is accomplished by adding a white

²² Codesignation is an equivalence relation on variables and constants. To codesignate effectively is to be bound to. [20]

knight step, one that foils the step that would otherwise have become a clobberer.

2.5.14 ZHUGE

ZHUGE developed by Chang is a planner that incorporates two novel features for planning, problem analysis and goal-oriented hierarchical operator representation [34][35]. Problems are analyzed using heuristic rules to reorder the subgoals so that interaction are avoided. The subgoals are then solved using a goal-oriented hierarchical operator representation which avoids the time-consuming operator search process by organizing the operators carefully.

The architecture of ZHUGE is similar to that of many expert systems. It contains a knowledge base, a control structure and a blackboard. A frame structure is used for domain knowledge.

Search in ZHUGE is dealt with using heuristics, constraints and operator abstraction. It is also aided by dividing the knowledge base into three areas, domain knowledge, domain expertise and operator descriptions.

The first phase of the planning process, analysis uses both heuristic rules and constraints. The need for backtracking in the planning process is eliminated by using the heuristic rules to order the subgoals. Constraints are also posted during the analysis phase to ensure that possible causes of backtracking that cannot be dealt with by subgoal ordering are dealt with.

During actual planning an operator abstraction technique termed goal-oriented hierarchical operator representation is used reduce operator search and provide abstraction levels whose benefits were discussed earlier. Operator search time is reduced because

each operator is defined in a goal oriented representation which is named by the goal it will lead to. The operators themselves are hierarchically structured, an abstract operator can be refined into a sequence of less abstract operators. The refinement is conflict free. A goal directed approach is used to select the appropriate operators. Specific knowledge of how to achieve a goal is locally encoded in the operator representation. This knowledge includes how to achieve the assigned goal in various situations.

In ZHUGE a procedural representation is used to enable the use of simple add/delete lists as well as complex effects. This allows richer problem domains to be used. In a procedure trivial add/delete effects can be easily represented by table lookup and complex effects can be defined by using various conditional procedure branches. This has the advantage that it is easy to encode heuristic information but the disadvantage that it is hard to maintain.

ZHUGE eliminates the need to deal with interacting subgoals during planning proper by preprocessing the original goal. During this analysis phase the subgoals are ordered with the aid of heuristic rules. They are analyzed from a global view to find relationships between subgoals. The subgoals are then ordered. Constraints are also posted either to help solve interaction problems where detailed situations can not be foreseen during the analysis phase or to increase the efficiency of the plan.

The heuristics rules used to order the subgoals are carefully ordered as well. They are weighted in an ABSTRIPS fashion except that in ABSTRIPS weights were given to subgoals or preconditions and in ZHUGE weights are given to rules. Some of the rules are concerned with subgoal ordering and others are concerned with constraint posting. Constraint are placed in front of the effected subgoal and are processed during planning. Constraints are also manipulated in order to increase the efficiency of certain plans. If

the heuristic rules are complete and correct subgoal interactions should not occur after the analysis phase has been completed.

Chapter 3

Towards Standardizing and Formalizing Planning

3.1 General

Planning to date has progressed incrementally with the addition of new ideas put forth in systems like those described in Chapter 2, but has suffered from the lack of a unifying vocabulary of terms and concepts. In essence many of the terms used refer to the same procedure or situation. Unfortunately, variations or modifications of previous ideas are given new names, putting the onus on the reader to recognize similarities and differences.

The techniques used in planning have also evolved and it is now evident that certain techniques are more useful than others. Scruffy AI, as it has been called, can be formalized by identifying these techniques and applying some theoretical and logical analysis. Chapman [20] has addressed this issue and his ideas have influenced those presented here.

The purpose of this chapter is to define a set of planning terms that can be used to describe a wide variety of planning techniques and concepts, and to introduce a promising attempt to formalize planning. It is evident from the research performed for this thesis that these areas are important and require more emphasis in general.

3.2 Standardizing Planning

It is clear that certain ideas and concepts in planning that are used frequently, often with different names, should be identified and unified with a common planning terminology. The following paragraphs discuss the planning terms and concepts that this thesis has adopted. It is proposed that they be adopted generally as they meet the requirements specified in the previous section and as such would greatly enhance the readability of future planning research.

The two main concepts of interest are *goal achievement* and *goal interaction*. Planning involves two choices, either achieve the goal or correct the interaction. Interaction detection is also quite useful if it helps prevent interactions from occurring. Interaction detection can occur as part of the first two choices, as is accomplished by using constraints, or it can be done independently through analysis. However, not all situations permit prior analysis.

3.2.1 Goal Achievement

Goal Achievement can be accomplished in three ways:

1. Simple Establishment;
2. Complex Establishment; and
3. Step Addition.

These terms are clear, concise and cover the range of possible actions necessary. Simple establishment refers to the act of determining the truth value of a goal. If a goal is true then it has been achieved. The concept of establishment has been extended to include complex establishment. It has been added for those cases where a goal's truth value cannot be determined by querying the state space directly. Complex establishment allows

complex procedures to be invoked whose success or failure is then used to determine whether or not the goal was established, and thus achieved.

If a goal cannot be established the only way to achieve it is by step addition. Step addition refers to the process of adding a step to the plan that has as one of its effects the addition of the required state to the state space. This however could result in the removal of other states that were the basis of earlier establishments which Chapman calls clobbering. This leads us to the second choice in planning interaction correction.

3.2.2 Goal Interactions

Once a goal interaction is detected a planner has two options, it can add a step to the plan that re-achieves the goal that was removed or one of the steps or goals involved in the interaction can be moved to an alternate place in the plan. This can be done by either adding temporal constraints or by physically moving steps using techniques such as goal or action regression. The process of moving steps is called promotion and can be applied individually or in a block. Individual promotion allows single steps or goals to be moved to correct interactions while still remaining achievable themselves. Block promotion moves the steps including their preconditions as a block so as to avoid further interactions. Unfortunately block promotion cannot solve certain interaction problems such as the Sussman Anomaly. The concept of promotion is not new, however, it deals with the notion of reordering plan steps very well.

3.3 Formalizing Planning

Planning has reached the point where enough is known to standardize terms and concepts. Procedures that accomplish the functions outlined in the previous section exist in a

number of formats and can be accepted as basic techniques within planning. To this end it now becomes useful to formalize the procedures that perform goal achievement and goal interaction detection and correction. Chapman has done this, his non-deterministic achievement procedure [20] performs goal establishment and declobbering by adding constraints to his non-linear constraint posting planner TWEAK. The procedure is based upon his Modal Truth Criterion that is explained and proven in the reference. The structure of the non-deterministic achievement procedure can be represented by a tree with the ends of the branches containing constraints that are added to the system. Alternate paths through the structure are called plan modification operations. It is not the intention to provide a detailed discussion of Chapman's work as it is covered in the reference. The salient point to note is that it is possible to formalize basic planning knowledge. The benefit is an implementable procedure that can perform these techniques in a uniform way, and in Chapman's case using constraint posting. The non-deterministic achievement procedure is limited by its properties of completeness and correctness but it is a starting point from which to build. It is also necessary to determine the formal state of the art from time to time so that the progress of the field in general can be assessed.

Chapter 4

The Puzzler: A Planner for the Puzzle World

4.1 Motivation

In order to better understand state space search, the STRIPS assumption, the effects of the combinatorial explosion and to provide a planner for a robot arm under construction, a planner called *The Puzzler* was implemented. The robot arm is being constructed at Carleton University with two degrees of freedom. This has placed some interesting restrictions on the domain and implementation of the Puzzler. The restriction of movement to only the x and y directions allows the arm to grasp and slide objects on a flat surface. This has prompted the creation of a special planning domain for the arm called the *Puzzle World*. The Puzzle World is described in the next section and is similar to the well known Blocks World. It is simple to understand and visualize yet has enough depth to support problems that are capable of demonstrating the effects of the combinatorial explosion. The Puzzler is a STRIPS style linear planner that uses a state space representation and employs the STRIPS assumption to deal with the frame problem. It incorporates some advanced planning techniques that together with the domain used provides a good vehicle for investigating the above points. The Puzzler was implemented in Logicware's MPROLOG [36] so the author could gain experience with PROLOG in the domain of planning. This experience will prove valuable when

selecting the implementation language for the large scale planning development system introduced in Chapter 6.

4.2 Problem Description

Consider a puzzle consisting of a grid and a number of different flat pieces as shown in Figure 4.1 . The goal is to *complete the puzzle*, that is cover the entire grid with the pieces available. Only as many pieces as required need be used. The plan is to be used by a robot arm²³ that will physically place pieces on a grid. A handle is provided on each piece for the arm to grasp. The robot arm, having only two degrees of freedom, can only move the pieces in either the x or y direction. Since the robot arm cannot pick up the pieces it must slide them into position, making the path used a critical factor. Although the emphasis of this thesis is not concerned with path planning, consideration of the path adds an interesting complexity to the problem. Pieces must have a clear path to their destination and can enter the grid from any of the four sides. To restrict the path planning problem to a reasonable difficulty pieces can only be moved in a straight line. However, non-movable *Pillars* can be placed on the grid to encourage subgoal interactions.

The pieces used to complete the puzzle are referred to as Puzzle World members. Any number of them can be used provided there are enough to solve the problem and any shape can be used provided it has been specified in the planner prior to the beginning of planning. Shapes must however, be made up of the squares used to form the grid. Figure 4.1 shows examples of the pieces used for testing purposes. They are described in more detail in Figure 4.2 . Handles are indicated by a '*' and the orientation is measured in degrees. The pieces are rotated about the axis formed by the handle. Pieces are

²³ The author had no control over the constraints placed on the ability of the robot arm.

Figure 4.1 Puzzle World

slightly smaller than the area they are intended to cover to provide adequate tolerances for manipulation by the robot arm.

The Puzzle World has a number of interesting properties that make it an interesting planning domain. The domain contains a wide range of user definable parameters that allow both simple and complex problems to be defined. The problem size can be varied

by changing the maximum dimensions of the grid and the problem complexity can be varied by introducing more pillars. The number and specification of piece types can also add complexity to the problem. By using the granularity of the grid to control the shape of the pieces a simple coordinate scheme can be used to define piece shape and location on the grid. Also simple formulae can be used to determine the coverage of a particular piece type before and after a rotation.

The grid is also useful for refining the overall goal of *complete the puzzle* into subgoals concerned with covering each individual cell. Subgoal (cell) ordering can now be exploited by the planner. The nature of the problem encourages the use of puzzle solving heuristics. For example heuristics could determine subgoal ordering or piece type selection. Heuristics based on the resource mechanism of SIPE discussed in Chapter Two could also be used. The use of subgoals based on cells of the grid is also interesting because all but one piece type defined in figure 4.2 cover more than one square. The portion of the piece containing the handle becomes the part of the piece to cover the subgoal square. Rotation about the axis of the handle is used to align the piece so that it can be slid into place unobstructed.

The need to determine the coverage of a particular piece type with a particular rotation has prompted the use of *complex states*. Complex states are not states that can have their truth value determined using simple establishment but are predicates that evaluate to true or false after performing a number of complex tests or procedures. Examples of complex states will be discussed in more detail in Chapter 5.

The maximum size of the grid used for the Puzzle World is governed by hardware and MPROLOG implementation details. This will be discussed in more detail later in the chapter.

TYPE	DIMENSION	COVERAGE	ORIENTATION	VIEW
a	2 x 2	(x,y),(x+1,y+1), (x+1,y),(x,y+1)	0
		(x,y),(x+1,y-1) (x+1,y),(x,y-1)	90
		(x,y),(x-1,y-1) (x-1,y),(x,y-1)	180
		(x,y),(x-1,y+1) (x,y+1),(x-1,y)	270
b	1 x 2	(x,y),(x,y+1)	0
		(x,y),(x+1,y)	90
		(x,y),(x,y-1)	180
		(x,y),(x-1,y)	270
c	2 x 1	(x,y),(x+1,y)	0
		(x,y),(x,y-1)	90
		(x,y),(x-1,y)	180
		(x,y),(x,y+1)	270
d	1 x 1	(x,y)	0
e	2 x 2	(x,y),(x,y+1), (x+1,y+1)	0
		(x,y),(x+1,y), (x,y-1)	90
		(x,y),(x-1,y), (x,y-1)	180
		(x,y),(x,y+1), (x-1,y)	270
f	2 x 3	(x,y),(x,y+1), (x,y+2) (x+1,y),(x+1,y+2)	0

Figure 4.2 Puzzle World Members (Continued . . .)

TYPE	DIMENSION	COVERAGE	ORIENTATION	VIEW
		$(x,y),(x,y-1),$ $(x+1,y)$ $(x+2,y),(x+2,y-1)$	90	$\begin{bmatrix} [*] & \square & \square \\ \square & & \square \end{bmatrix}$
		$(x,y),(x-1,y),$ $(x-1,y-2)$ $(x,y-1),(x,y-2)$	180	$\begin{bmatrix} & & \square [*] \\ & & \square \\ & & \square \square \end{bmatrix}$
		$(x,y),(x-2,y+1),$ $(x-2,y)$ $(x-1,Y),(x,y+1)$	270	$\begin{bmatrix} \square & & \square \\ \square \square & [*] & \end{bmatrix}$

Figure 4.2 Puzzle World Members

4.3 States in the Puzzle World Space

States in the puzzle world were determined from the information deemed necessary for planning in the domain. This information includes:

1. The dimension of the puzzle being solved;
2. Which squares are covered;
3. Which squares are uncovered;
4. Which pieces are available;
5. Which piece types are available;
6. The specifications of a piece type;
7. The orientation of a particular piece.
8. The location of a piece that has been placed on the grid, (specifically the location of the handle);
9. Which piece the robot hand (arm) is holding;
10. Whether or not the robot hand (arm) is empty.

These are transformed into the following Prolog fact predicates (examples are related to figure 4.1) :

1. `max_dimension(Dimension) : max_dimension(5,5);`
2. `covered(Square) : covered([4,1]);`
3. `uncovered(Square) : uncovered([4,3]);`
4. `available(Piece) : (Piece = [Type-Number,[Size]]) available([a-1,[2,2]]);`
5. `piece_type(Type) : piece_type(a)`
6. `orientation(Piece,Orientation) : (Orientation = 0,90,180 or 270 ) orientation([f-1,[2,3]],90);`
7. `at(Piece,Square) : at([e-1,[2,2]], [3,3]);`
8. `holding(Piece) : holding([c-2,[2,1]]);`
9. `handempty.`

Although the closed world assumption is made, both covered and uncovered facts are used to make the STRIPS style operators more readable.

4.4 Operators in The Puzzle World

4.4.1 The STRIPS Style Operators

The Puzzler is a STRIPS style linear planner that makes the STRIPS assumption and as such uses STRIPS style operators. The operators consist of three lists, a preconditions list, an add list and a delete list. The preconditions list contains states, (including complex states), that must be true in the world before an operation can be performed. To perform an operation the states in the delete list of the operator are removed from the world and then the states in the add list are added to the world producing a new world that

resulted from performing the operation on the previous or old world. Adding or deleting a complex state also triggers a complex add or delete version of the operator.

The operators used by the Puzzler for the puzzle world are shown in figure 4.3 . These operators only allow pieces to be placed onto the grid, not removed. The object is to plan the placement of the pieces not their removal. The start state of a puzzle includes only the available pieces, their type descriptions and any non-movable pillars that may be present.

4.4.2 Operator Application Descriptions

A typical sequence for placing a piece on the grid includes selecting the piece, rotating it if necessary and then placing it. In order to select a piece $[select(P)]$ it must be available $[available(P)]$ and the robot arm must be empty $[handempty]$. The precondition $first_use(P)$ is a complex state that ensures that only one piece of a particular type will be tried in a particular location. For example, if one does not fit then no other piece of the same type will fit either. Since $first_use(P)$ is a complex state it does not appear as a state in the world. Once a piece is selected, it is no longer available and the robot arm is no longer empty. The state $holding(P)$ is then added to the world.

If a piece will not fit in its current orientation, it may fit if it is rotated about the axis of the handle using a rotation operator. A separate operator is defined for each of the three possible rotation angles from the default position of zero (0) degrees. In order to be rotated the robot arm must be holding the piece $[holding(P)]$, the piece must be in the default orientation $[orientation(P, 0)]$, and the area covered by the piece using this rotation must be clear $[clear_area(P, S, 90)]$. The precondition

Operators	Preconditions	Add List	Delete List
select(P)	handempty first_use(P) available(P)	holding(P)	handempty available(P)
place(P,S,O,D)	holding(P) uncovered(S) uncovered(P,S,O) clear_path(P,O,S,D) non_isolating(P,O, S,D)	covered(S) covered(P,S,O) at(P,S) handempty	uncovered(S) uncovered(P,S,O) holding(P)
rotate_90(P,S)	holding(P) orientation(P,0) clear_area(P,S,90)	orientation(P,90) uncovered(P,S,90)	orientation(P,0)
rotate_180(P,S)	holding(P) orientation(P,0) clear_area(P,S,180)	orientation(P,180) uncovered(P,S,180)	orientation(P,0)
rotate_270(P,S)	holding(P) orientation(P,0) clear_area(P,S,270)	orientation(P,270) uncovered(P,S,270)	orientation(P,0)

Figure 4.3 STRIPS Style Operator Descriptions

`clear_area(P, S, 90)` is a complex state that for a particular piece `P` and the goal square `S` with orientation 90 degrees determines whether or not the other squares covered by the piece are clear. If they are clear, the old default orientation state is removed from the world and the new orientation is added. The complex state `uncovered(P, S, 90)` appears in the add list so that the rotation operators will be considered as applicable when the achievement of `uncovered(P, S, 90)` is attempted. STRIPS style planners select operators if the goal needed to be solved appears in the add list of an operator.

If rotation is not required the operator can be placed on the grid using the place operator. In order to use this operator the robot arm must be holding the piece [`holding(P)`], the goal square must be clear [`uncovered(S)`], the other squares affected by the piece must also be clear [`uncovered(P, S, O)`], the path to the squares must be clear [`clear_path(P, O, S, D)`] and the final location of the piece must not isolate any other clear squares [`non_isolating(P, O, S, D)`]. The precondition `uncovered(P, S, O)` is a complex state that for a piece `P` with orientation `O` and handle on square `S` determines if the other squares covered by the piece are clear. If they are not clear (the precondition is false) a rotation operator may be used to find an orientation that is clear. The precondition `clear_path(P, O, S, D)` is also a complex operator that for a particular piece `P` with orientation `O` covering square `S` determines the direction `D` of a clear path from the square to one of the four edges of the grid if one does exist. The possible directions are north, south, east and west. If no path exists to a clear square then the planner halts signaling an *isolated square — no solution possible* error. Finally, the precondition `non_isolating(P, O, S, D)` is also a complex state that for a particular piece `P` with orientation `O` covering square `S` from direction `D` determines whether the placement will result in a square becoming isolated. Once the preconditions have succeeded the subgoal square is no longer uncovered and the robot

arm is now empty. The squares that are now covered and the location of the piece (its handle) are then added to the world.

4.4.3 Including a Remove Operator: Interesting Results

It may seem at first glance that a remove operator is needed for the puzzle world. Such a capability was added to the puzzle world with the operators shown in figure 4.4²⁴. Tests were done with this modified set of operators which resulted in some interesting observations that lead to the removal operator being abandoned. This discussion is relevant to planning in general as it illustrates the types of problems encountered when designing planning representations.

Using the modified set of operators the planner began to *thrash*²⁵. Specifically, when ever a proposed piece would not fit due to the previous placement of other pieces, the planner would attempt to remove those pieces to make room. This occurred because the precondition of `place/4` that failed, `uncovered/1`, now has an operator that can achieve it. Prior to adding the remove operator this would have caused backtracking to the previous precondition.

When examining the operators with remove it appears that they represent an acceptable operator organization, all required functions are present and all precondition's, add and delete lists are appropriate for their particular operator. However, this is not so and after closer observation and testing it appears that the remove function should remove the piece, re-orient it and put it away all in a single operation. If the current remove is used and the precondition of `place/4`, `uncovered/1`, is achieved, the

²⁴ This is an interim set of operators and as such is not comparable to the final set of operators used.

²⁵ Thrashing is the process of repeatedly achieving, undoing, and re-achieving goals.

Operators	Preconditions	Add List	Delete List
select(P)	handempty first_use(P) available(P)	holding(P)	handempty available(P)
place(P,S,O,D)	holding(P) uncovered(S) uncovered(P,S,O) clear_path(P,O,S,D) non_isolating(P,O, S,D)	covered(S) covered(P,S,O) at(P,S) handempty	uncovered(S) uncovered(P,S,O) holding(P)
rotate_90(P,S)	holding(P) orientation(P,0) clear_area(P,S,90)	orientation(P,90) uncovered(P,S,90)	orientation(P,0)
rotate_180(P,S)	holding(P) orientation(P,0) clear_area(P,S,180)	orientation(P,180) uncovered(P,S,180)	orientation(P,0)
rotate_270(P,S)	holding(P) orientation(P,0) clear_area(P,S,270)	orientation(P,270) uncovered(P,S,270)	orientation(P,0)
remove(P,S)	handempty covered(S) at(P,S) clear_path	uncovered(S) holding(P)	at(P,S) handempty covered(S) covered(P,S,O) uncovered(P,S,O)
put_away(P)	holding(P) orientation(P,0)	handempty available(P)	holding(P)
rotate_90_to_0 (P,S)	holding(P) orientation(P,90)	orientation(P,0)	orientation(P,90)
rotate_180_to_0 (P,S)	holding(P) orientation(P,180)	orientation(P,0)	orientation(P,180)
rotate_270_to_0 (P,S)	holding(P) orientation(P,270)	orientation(P,0)	orientation(P,270)

Figure 4.4 STRIPS Style Operator Descriptions With Remove

last precondition of `place/4`, `holding/1`, is also re-achieved resulting in `place/4` succeeding using the piece that was removed.

The above problem could be solved by redesigning all of the operators but the thrashing problem would require additional logic to be added to the Strips style mechanism. It seems counter-intuitive to design a planner that undoes what it has planned when it should have planned the initial action properly the first time. The Puzzler is not dealing with any unknowns as it has access to all of the facts required to plan and is responsible for all changes to the world. It was for these reasons that the remove operator was not included in the final set of strips style operator descriptions.

4.5 Implementation Details of The Puzzler

4.5.1 General Execution

The Puzzler is a single module program written in Logicware's MProlog. The search used is the default depth first search of prolog. Interactions are handled by using the default backtracking, intelligent backtracking and goal analysis. The source code appears in Appendix A. When executed the program prompts the user for the name of a file that contains the operator definitions and required start state information. An example input file appears in Appendix B. Upon successful reading of this file the predicate `plan/0`²⁶, the main control loop of the planner, is called.

The overall control loop of the planner consists of

1. initializing the world with the start state (`init_start_state/0`);
2. initializing the goal stack (`init_goal_stack/1`);

²⁶ This is a standard Prolog convention for describing predicates, name/arity. The arity is the number of variables the predicate has. Predicates with the same name are considered different as long as they have different arities.

3. analyzing the positions of the non-movable pillars and ordering the subgoal cells based on the analysis heuristics (`perform_analysis/0`);
4. solve the puzzle (`solve/2`);
5. output the plan to the terminal (`write_plan/0`).

The Puzzler uses a list to maintain the stack of goals, subgoals and actions. Stack entries are tagged as either goals and conditions (preconditions) to be achieved, or actions to be performed. The `solve/2` predicate pops the next entry from the stack and calls the `solve/3` predicate to execute the action or achieve the goal or condition using simple establishment or step addition. Unlike simpler STRIPS style planners the Puzzler checks for duplicate entries on the stack that can result in infinite looping using its `check_loop/1` predicate.

When an action entry is encountered its preconditions are checked to ensure that no clobbering has occurred. If they are still true the states in the delete list of the operator are removed from the world and then the states in the add list are added. The operator is then recorded dynamically using the predicate `solution/1`. If clobbering has occurred then the preconditions of the operator are added once again to the stack and planning continues. The analysis performed before solving the puzzle virtually eliminates all major interactions in the domain of the puzzle world.

The `solve/3` predicate also handles the achievement of complex states. A separate clause of the predicate is used for each complex state. When encountered the applicable actions are performed. This will be discussed in detail in Chapter 5.

4.5.2 Analyzing the Grid

Analysis of the non-movable pillar configuration prior to planning performs two functions collectively termed goal analysis:

1. refines the overall goal of *complete_the_puzzle* to that of a list of individual squares that must be covered; and
2. orders the above list so as to ensure little or no interactions due to path planning problems.

The goal analysis is based on a heuristic that categorizes certain squares into two groups, group 1 and group 2. If a square appears in both of these groups it is moved to a third group, group 3. Squares in group 3 are covered first followed by squares in group 2, followed by squares in group 1, and then followed by all those squares not categorized.

The categorization is performed by analyzing all combinations of pillar pairs allowing no repetitions based on different orderings, ie n choose 2, $\binom{n}{2}$. This results in the following number of pairs being tested with n being the number of pillars, $\frac{n!}{2!(n-2)!}$. Pairs that fall into group 1 are of the form $(X,Y),(X1,Y1)$ and produce group 1 squares $(X,Y1),(X1,Y)$. Pairs that fall into group 2 can be one of two forms. The first group 2 form is $(X,Y1),(X,Y2)$ such that $Y1 < Y2$ which produces the following group 2 set: $\{(X,B_1), \dots, (X,B_n) \mid B_1 = Y1+1, B_n = Y2-1, B_{i+1} = B_i+1\}$. The second group 2 form is $(X1,Y),(X2,Y)$ such that $X1 < X2$ which produces the following group 2 set: $\{(B_1,Y), \dots, (B_n,Y) \mid B_1 = X1+1, B_n = X2-1, B_{i+1} = B_i+1\}$. The equations are based on determining which squares are between pillars on the same row or column and which squares are at the intersection points of the rows and columns occupied by pillars. Squares occupied by pillars are removed from consideration after the final ordering is completed. Figure 4.5 shows an example of a 5x5 grid with four pillars. The key cell is

(3,4) as it has only one direction of access, west. If a piece is put on (1,4) or (2,4) that does not also fill (3,4), (3,4) will become isolated. The planner will backtrack placing pieces in different locations in order to prevent this from happening but backtracking is costly. The goal analysis performed identifies (3,4) as a critical (group 3) cell and gives it the highest priority. This analysis is termed a heuristic as it has not been proven to always provide the optimum ordering but tests discussed in the next section show a marked increase in performance when it is used.

4.6 Performance Testing and Examples

This section discusses five example puzzles solved by the puzzler. The examples were run both with and without the use of goal analysis. A typical plan consists of repeated groups of two or three operators depending on whether or not the piece requires rotating. A sample group taken from the beginning of the plan for Example 1 below has piece b-1 (a 1 x 2 sized piece) being selected, then rotated 270 degrees and then placed with the 270 degree orientation onto the grid with the handle on square [5,6] using a straight line path entering from the western edge.

Example 1, shown in figures 4.6 and 4.7, uses the sample input file at Appendix B. Figure 4.6 shows the start state configuration which is a 10 x 10 square puzzle with 20 non-movable pillars. The order of the pieces in the figure represents the order of piece selection in the input file. For this example the Puzzler has the following piece types available: 9 x a, 1 x e, 1 x f, 14 x b, 13 x c and 40 x d for a total of 78 pieces. The puzzle required only 29 pieces be used. The plan produced by the Puzzler, shown in figure 4.7 is as follows:

Pillars	Group 1	Group 3	Group 2	Final Order
(1,2)	(1,5)		(2,2)	(3,4)
(3,2)	(3,2)		(3,3)	(2,2)
(3,5)	(1,4)	(3,4)	(3,4)*	(3,3)
(4,4)	(4,2)			(1,5)
	(3,4)*			(3,2)
	(4,2)			(1,4)
	(3,4)*			(4,2)
	(4,5)			(4,5)
				All other squares

Figure 4.5 Grid Analysis Example

```
enter STRIPS Style operator file in single quotes
*'p810c20.pro'.
'p810c20.pro'.
```

The start state is: ...(removed to reduce code length)

The goal is: complete_the_puzzle

The plan is:

```
select([b-1,[1,2]])
rotate_270([b-1,[1,2]],[5,6])
place([b-1,[1,2]],[5,6],270,west)
select([b-2,[1,2]])
rotate_180([b-2,[1,2]],[5,4])
place([b-2,[1,2]],[5,4],180,south)
select([a-1,[2,2]])
rotate_270([a-1,[2,2]],[5,8])
place([a-1,[2,2]],[5,8],270,north)
select([a-2,[2,2]])
rotate_180([a-2,[2,2]],[3,7])
place([a-2,[2,2]],[3,7],180,west)
select([a-3,[2,2]])
rotate_180([a-3,[2,2]],[3,5])
place([a-3,[2,2]],[3,5],180,west)
select([a-4,[2,2]])
rotate_90([a-4,[2,2]],[7,7])
place([a-4,[2,2]],[7,7],90,east)
select([a-5,[2,2]])
rotate_90([a-5,[2,2]],[7,5])
place([a-5,[2,2]],[7,5],90,east)
select([a-6,[2,2]])
place([a-6,[2,2]],[1,1],0,south)
select([e-1,[2,2]])
place([e-1,[2,2]],[1,3],0,west)
select([b-3,[1,2]])
place([b-3,[1,2]],[1,5],0,north)
select([b-4,[1,2]])
place([b-4,[1,2]],[1,7],0,north)
select([b-5,[1,2]])
place([b-5,[1,2]],[1,9],0,north)
select([b-6,[1,2]])
place([b-6,[1,2]],[2,8],0,north)
select([b-7,[1,2]])
rotate_90([b-7,[1,2]],[2,10])
place([b-7,[1,2]],[2,10],90,north)
select([a-7,[2,2]])
```

```

place([a-7,[2,2]],[3,1],0,south)
select([b-3,[1,2]])
rotate_90([b-8,[1,2]],[4,10])
place([b-8,[1,2]],[4,10],90,north)
select([a-8,[2,2]])
place([a-8,[2,2]],[5,1],0,south)
select([b-9,[1,2]])
rotate_90([b-9,[1,2]],[6,10])
place([b-9,[1,2]],[6,10],90,north)
select([a-9,[2,2]])
place([a-9,[2,2]],[7,1],0,south)
select([b-10,[1,2]])
rotate_90([b-10,[1,2]],[7,9])
place([b-10,[1,2]],[7,9],90,east)
select([b-11,[1,2]])
rotate_90([b-11,[1,2]],[8,10])
place([b-11,[1,2]],[8,10],90,north)
select([f-1,[2,3]])
place([f-1,[2,3]],[9,1],0,south)
select([b-12,[1,2]])
place([b-12,[1,2]],[9,4],0,east)
select([b-13,[1,2]])
place([b-13,[1,2]],[9,6],0,east)
select([b-14,[1,2]])
place([b-14,[1,2]],[9,8],0,east)
select([d-1,[1,1]])
place([d-1,[1,1]],[10,2],0,east)
select([c-1,[2,1]])
rotate_270([c-1,[2,1]],[10,4])
place([c-1,[2,1]],[10,4],270,north)
select([c-2,[2,1]])
rotate_270([c-2,[2,1]],[10,6])
place([c-2,[2,1]],[10,6],270,north)
select([c-3,[2,1]])
rotate_270([c-3,[2,1]],[10,8])
place([c-3,[2,1]],[10,8],270,north)
select([d-2,[1,1]])
place([d-2,[1,1]],[10,10],0,north)

```

Examples 2 through 5 are all 5 x 5 square puzzles with a total of 16 pieces available. The piece type breakdown and selection order is as follows: 3 x a, 3 x b, 2 x c and 8 x d.

Figure 4.7 Example 1 Goal State

due to the isolated square. The message produced by the Puzzler is as follows:

```
isolated square detected during analysis
- no solution possible
```

Example 3 is shown in figure 4.8(b). It is a simple example where the goal squares are covered in the default order, column by column, by increasing row and column. The plan produced by the Puzzler is as follows:

```
enter STRIPS Style operator file in single quotes
*'p85dc8.pro'.
'p85dc8.pro'.
```

The start state is:...(removed to reduce code length)
 The goal is: complete_the_puzzle
 The plan is:
 select([a-1,[2,2]])
 place([a-1,[2,2]],[1,1],0,south)
 select([b-1,[1,2]])
 place([b-1,[1,2]],[1,3],0,north)
 select([d-1,[1,1]])
 place([d-1,[1,1]],[1,5],0,north)
 select([a-2,[2,2]])
 place([a-2,[2,2]],[3,1],0,south)
 select([b-2,[1,2]])
 place([b-2,[1,2]],[5,1],0,south)
 select([b-3,[1,2]])
 place([b-3,[1,2]],[5,3],0,north)
 select([d-2,[1,1]])
 place([d-2,[1,1]],[5,5],0,north)

Example 4 is shown in figure 4.8(d). The goal analysis process is clearly demonstrated by the order of the squares selected for covering by the first two pieces. The plan produced by the Puzzler is as follows:

enter STRIPS Style operator file in single quotes
 *'p85ac8.pro'.
 'p85ac8.pro'.
 The start state is:...(removed to reduce code length)
 The goal is: complete_the_puzzle
 The plan is:
 select([a-1,[2,2]])
 rotate_180([a-1,[2,2]],[4,4])
 place([a-1,[2,2]],[4,4],180,west)
 select([a-2,[2,2]])
 rotate_180([a-2,[2,2]],[2,4])
 place([a-2,[2,2]],[2,4],180,west)
 select([b-1,[1,2]])
 place([b-1,[1,2]],[1,1],0,south)
 select([d-1,[1,1]])
 place([d-1,[1,1]],[1,5],0,north)
 select([b-2,[1,2]])
 rotate_90([b-2,[1,2]],[2,1])
 place([b-2,[1,2]],[2,1],90,south)

```

select([b-3, [1,2]])
rotate_90([b-3, [1,2]], [4,1])
place([b-3, [1,2]], [4,1], 90, south)
select([d-2, [1,1]])
place([d-2, [1,1]], [5,2], 0, east)
select([d-3, [1,1]])
place([d-3, [1,1]], [5,5], 0, north)

```

The final example, Example 5, is shown in figure 4.8(c). Again the analysis process is demonstrated. The plan produced is as follows:

```

enter STRIPS Style operator file in single quotes
*'p85c8.pro' .
'p85c8.pro' .
The start state is:...(removed to reduce code length)
The goal is:  complete_the_puzzle
The plan is:
select([a-1, [2,2]])
rotate_180([a-1, [2,2]], [3,4])
place([a-1, [2,2]], [3,4], 180, west)
select([b-1, [1,2]])
place([b-1, [1,2]], [1,1], 0, south)
select([b-2, [1,2]])
place([b-2, [1,2]], [1,3], 0, north)
select([d-1, [1,1]])
place([d-1, [1,1]], [1,5], 0, north)
select([b-3, [1,2]])
rotate_90([b-3, [1,2]], [2,1])
place([b-3, [1,2]], [2,1], 90, south)
select([c-1, [2,1]])
place([c-1, [2,1]], [4,1], 0, south)
select([c-2, [2,1]])
rotate_270([c-2, [2,1]], [5,2])
place([c-2, [2,1]], [5,2], 270, north)
select([d-2, [1,1]])
place([d-2, [1,1]], [5,4], 0, north)
select([d-3, [1,1]])
place([d-3, [1,1]], [5,5], 0, north)

```

For each of the above examples, with the exception of Example 1, diagnostics where

(a)

(b)

(c)

(d)

Note: Each example started with 16 pieces as follows: 3 x a, 3 x b, 2 x c, 8 x d

Figure 4.8 Examples 2 through 5

Example	Analysis	CPU Time (secs)	Calls	Backtracks
1	Yes	128.99	133,803	10,687
	No	823.32	618,321	65,720
2	Yes	2.23	1,960	221
	No	2.19	1,960	221
3	Yes	8.04	11,443	2,013
	No	8.07	11,441	2,013
4	Yes	10.57	16,262	2,688
	No	49.97	81,880	2,688
5	Yes	10.81	15,956	2,746
	No	39.25	63,934	9,851

Figure 4.9 Diagnostics of Examples

recorded with and without the use of goal analysis. In both cases the analysis was computed but in the latter case it was not used. In the case of Example 1, the f type pieces had to be removed from consideration as without analysis, the Puzzler placed it using an orientation such that it produced an isolated square which caused execution to be halted. Figure 4.9 shows the CPU time in seconds, the number of predicate calls and the number of backtracks performed for each example presented with and without analysis. These figures are meant for comparison purposes only and are not extensive enough to draw any numerical conclusions but they do confirm the expected increase in performance in terms of a decrease in CPU time and backtracking, gained by using the goal analysis technique.

The Puzzler uses a number of techniques to improve performance. Without the

addition of these techniques puzzle sizes of 19 x 19 or greater could not be solved using the default memory configuration of MProlog on a Sun 3/60 work station. The addition of several planning techniques overcame this limitation.

4.7 Limitations of the Puzzler

In addition to the limitations normally associated with linear planners discussed in Chapter 2, the Puzzler has other limitations. These limitations will be discussed in the following paragraphs.

The Puzzler uses a state space representation with add and delete lists combined with the Strips assumption to solve the frame problem. This representation although enhanced by the use of complex states is still somewhat restrictive making it difficult to handle complex problems. One type of complex domain that the Puzzler could not readily deal with are those domains with adversarial agents other than the Puzzler that can change the state space. This would require the Puzzler be able to react to the possibility of changes that might be made, in effect, it would need the ability to plan for contingencies.

Within the own domain of the puzzle world, the Puzzler is limited to straight line paths. Also, the Puzzler does not have the ability to clear a path by removing pieces from the board. Both of these capabilities could be added with a large increase of computational and logical overhead. The limitations of the robot arm have introduced the path planning problem.

The Puzzler was intended to be simple and Strips based. The challenge of improving its planning ability has aided in the authors overall understanding of A.I. planning.

Chapter 5

Planning Techniques Demonstrated by the Puzzler

5.1 General

The Puzzler was a vehicle for investigating planning, and specifically Strips style linear planning for a robot arm. By implementing the Puzzler the following planning techniques were demonstrated, and in the case of the first two techniques, developed especially for the Puzzler:

1. Complex states;
2. Implementing problem specific rules in a Strips style planner in a general way;
3. Goal Analysis; and
4. Intelligent backtracking.

Complex states were introduced in Chapter 4. Their discussion will be expanded to include implementation details as they impact on implementing problem specific rules in a Strips style planner in a general way, a technique developed for use in the Puzzler. The goal analysis technique was also introduced in Chapter 4. It will be discussed in terms of planning using the Puzzler. Lastly, intelligent backtracking which was covered in Chapter 2, is discussed as the Puzzler uses two of the intelligent backtracking techniques to improve its performance.

5.2 Complex States

The simplicity of the state space representation of Strips style planners is not without disadvantages, in particular, certain properties of the puzzle world have prompted the need for additions to the Strips mechanism in this respect. It was important however, to increase the capabilities of the Puzzler without changing the basic execution mechanism of the Strips algorithm. Complex states were developed to provide this increase while requiring only minor additions to the Strips mechanism used. In general, they are useful for performing complex procedures while maintaining the simplicity inherent in Strips style planners.

Complex States do not appear as admissible (simple) states in the state space of the domain but do appear in the precondition, add and delete lists of the operators with simple states. They have variables that can be instantiated and have executable procedures. These procedures can manipulate simple states as well as perform necessary computations. Upon completion of their execution success or failure is returned to the calling procedure. In the Puzzler these procedures are implemented as prolog predicates which either succeed or fail²⁷.

Complex states are also used in add and delete lists to manipulate groups of simple states. In this case special add and delete procedures are required to perform the appropriate functions. Complex states also appear in add lists to cause the selection of operators for step addition in the event that a complex state that is a precondition of another operator fails and must be achieved. It is important to note that care must be taken when using complex states to manipulate the state space. Using the basic Strips algorithm in conjunction with the Strips assumption ensures that the frame problem is

²⁷ The reader is reminded that failure in prolog is based on the closed world assumption.

handled. If complex states are not properly implemented the ability of the planner to deal with the frame problem might be impeded.

The process of adding complex state execution procedures and special add and delete procedures is simplified in the Puzzler by its implementation language, prolog. Once a predicate has been defined, different clauses of that predicate can be added for use depending on the calling situation. In the Puzzler the `solve/3` predicate has different clauses to perform simple establishment, step addition and execute complex state procedures. In order to define a complex state a new clause of the `solve/3` predicate must be added that either performs or calls the execution procedure when the complex state is encountered. By placing the complex state clauses before the simple establishment and step addition clauses their action is performed in lieu of these functions. The ordering of the `solve/3` clauses is actually the basis for the prolog strips algorithm implementation used in the Puzzler.

The following is a list of the complex states defined in the Puzzler:

1. `first_use(Piece);`
2. `clear_path(Piece, Orientation, Square, Direction);`
3. `non_isolating(Piece, Orientation, Square, Direction);`
4. `uncovered(Piece, Square, Orientation);`
5. `covered(Piece, Square, Orientation).`

The first three complex states are used to implement problem specific rules. Their development is discussed later in the chapter. The last two complex states are used to deal with groups of simple states. In the puzzle world a piece can cover more than one square of the grid but the goals that invoke piece placement contain only one square. In order to deal effectively with the side effect of the other squares that are covered complex

2

OF/DE

2

MICRO

states were developed to calculate those squares based on the piece type, the position of the original goal square and the orientation of the piece. The appropriate simple states that concern individual squares can then be added or deleted as the situation warrants. Complex state `uncovered/3` appears as a precondition of the `place/4` operator and in various add and delete lists while `covered/3` appears only in add and delete lists. The following prolog code fragments shows the `solve/3` clauses containing the `uncovered/3` and `clear_path/4` complex state execution procedures:

```
solve(c-X,C,S):-
    X=uncovered([P,D],Sq,Or),
    matching_statement(state(max_dimension(MX,MY))),
    matching_statement(state(orientation([P,D],Or))),
    calc_coverage(P,Sq,MX,MY,Or,Osc),
    handle_covered(check_uncovered,Osc),
    solve(C,S).

solve(c-X,C,S):-
    X=clear_path(P,Or,Sq,Dir),
    !,
    clear_path(P,Or,Sq,Dir),
    !,
    solve(C,S).
```

As can be seen the execution procedure can be included in the clause or can be called using another predicate. The predicate `handle_covered/2` performs simple establishment on the list of other squares covered by the piece. Failure of the simple establishment in this instance does not invoke step addition for an individual square it merely fails the precondition that was the complex state.

The next prolog code fragment is one of the `add_list/1` clauses that is an example of a special add list procedure:

```
add_list([H|T]):-
    H=uncovered([P,D],S,Or),
```

```

!,
matching_statement(state(max_dimension(MX,MY))),
calc_coverage(P,S,MX,MY,Or,Osc),
handle_covered(add_uncovered,Osc),
add_list(T).

```

Again the `handle_covered/2` predicate is used to manipulate the simple states.

Careful use of complex states has enabled complex computations to occur within the framework of a Strips style planner. This addition has aided the implementation of problem specific rules to Strips style planners, which is discussed in detail in the next section.

5.3 Implementing Problem Specific Rules in a Strips Style Planner in a General Way

Linear Strips style planners use a representation based on add/delete lists and the Strips assumption. This coupled with a state space representation allows them to perform planning that is simple and understandable. The domains modeled using the add/delete list format include the blocks world and the puzzle world among others. Generally a domain is encoded so that the basic actions required to solve goals are included. For example the blocks world could include the following operators: `stack(X,Y)`, `unstack(X,Y)`, `pickup(X)` and `putdown(X)`. If the admissible states include `on(X,Y)`, `ontable(X)`, `clear(X)`, `holding(X)` and `handempty` then it is clear that any of the states used as a goal condition could be achieved using the defined operators. The Strips planning architecture allows us to do basic planning in domains like the blocks world, basic meaning using the operators specified to achieve the desired goals. In order to add intelligence to the planner in terms of heuristic or other rules we could change the basic operation of the planner but this would defeat the benefit of using the Strips style

planner, simplicity. In order to add problem specific rules to a Strips style planner in a general way we must modify the operator and state space descriptions without changing their functionality. This requires minor additions to the execution mechanism that will be discussed later in the section.

Prior to implementing any problem specific rules the Puzzler solved problems using only the default depth first backtracking of the prolog implementation of the Strips algorithm developed. It was clear that certain branches of the solution tree could be pruned if certain problem specific rules were implemented. Two of the three rules listed below are not heuristic as they are true²⁸ in all cases. The first rule, however, is based on what appears to be a good rule of thumb for completing puzzles of this type. The following is a list of the domain specific rules implemented in the Puzzler:

1. try pieces in order of size, largest to smallest, cover as much of the grid as possible as soon as possible;
2. if a piece fails then no other piece of the same size will fit and is therefore not even considered (tried);
3. if a piece blocks the only remaining clear path to any uncovered square then it fails and if blockage is the result of a square encircled by pillars execution is halted as no solution exists.

Two methods were used to implement the above rules. The first method used, to implement rule 1, was to order facts in the prolog data base so that the default prolog selection tried pieces in order of descending piece size. This was accomplished by manually ordering the piece descriptions within the start state predicates of the input files (see Appendix B). The `start_state_of_grid/1` predicate contains a list of piece

²⁸ The reader is left to convince himself of this.

types, available pieces and starting piece orientations. The order of the `piece_type/1` facts sets the piece selection order, no other facts in the list need be ordered in any way. This technique is interesting as it can be used to implement a number of piece selection heuristics without modifying the Puzzler code. In fact, if the `piece_type/1` fact is not present for a particular type of piece, the Puzzler will not even consider that type even if `available/1` and `orientation/2` facts exist. This was done by partially instantiating the piece description with its type before searching the data base to instantiate the other part of the piece description, its number (pieces are identified as follows: *d-2* with the letter indicating the piece type and the number that particular instance). By ordering the available instantiations of the piece type part using the `piece_type/1` predicate the selection order can be controlled.

The second method, used to implement rules 2 and 3, was to modify the operator and state space descriptions without altering the basic Strips execution mechanism. The method involves adding complex states to the precondition list of the operators and can be broken down into the following eight steps:

1. Analyze the problem specific rule and identify the key operator and resources or simple states affected including the cause of the effect;
2. Identify the preconditions that determine the instantiation of the resource or simple state;
3. Identify the operators that achieve the above preconditions;
4. Repeat 2 and 3 until the operator and precondition responsible for the lowest level selection of the resource or simple state is reached;
5. Design a draft complex state and associated execution procedure that emulates the behavior required by the problem specific rule;

6. Determine the placement position for the complex state in the precondition list and add it to the list;
7. Add the associated execution procedure to the Strips execution mechanism in accordance with the section on complex states above; and
8. Test and adjust the position of the complex state in the precondition list and / or adjust the complex state execution procedure to ensure that the proper problem specific rule behavior has been achieved.

The above process will be demonstrated using the two problem specific rules implemented as examples. Refer to Figure 4.3 for the operator descriptions used by the Puzzler ignoring the complex states that have already been added.

The first problem specific rule to be implemented states: "if a piece fails then no piece of the same size will fit and is therefore not even considered". The eight steps are outlined below with comments:

Step 1: The rule attempts to stop the checking of all instances of a piece type after it has been determined that the first instance does not fit. Without such a rule all of the other instances would be checked wasting planning effort. The key operator is `place(P, S, O, D)` and the resource is the piece P. The effect is that `place(P, S, O, D)` fails if piece P does not fit. The only choice points available are as a result of achieving the resource.

Step 2: The precondition that determines the instantiation of piece P is `holding(P)`.

Step 3: The operator that achieves `holding(P)` is `select(P)`.

Step 4: The lowest level precondition responsible for determining the instantiation

of piece P is available(P). This is a simple state and therefore could be the source of the prolog choice points for backtracking, choice points that include all pieces.

Step 5: The key is to limit the instantiations of the variable P in available(P) so that on failure another piece type is selected. The simple state `piece_type/1` used to control the piece selection process can be used in conjunction with a complex state `first_use(P)` to accomplish this. The preconditions of `select(P)` prior to any rule implementations were `handempty` and `available(P)`. If we add a precondition, `first_use(P)`, to `select(P)`, that partially instantiates the piece type portion of the piece variable using the `piece_type/1` facts we can force the instantiation of the `available(P)` fact to be of the same piece type. The new prolog choice points for backtracking becomes the list of `piece_type/1` facts. The prolog cut can be used to prevent further backtracking after all of the different piece types have failed.

Step 6: No preconditions should go before the `handempty` precondition and since we are trying to force the instantiation of the `available(P)` precondition, the new precondition `first_use(P)` should go between the two. This would make the new preconditions of `select(P)` be `handempty`, `first_use(P)`, and `available(P)`.

Step 7: The following prolog code fragment is the `solve/3` clause that executes the complex state procedure for `first_use(P)`:

```
solve(c-X,C,S):-
    X=first_use([Y,_]),
    !,
    matching_statement(state(piece_type(Y))),
    solve(C,S).
```

Step 8: Test and adjust.

The above example has been implemented and performs as expected.

The second example implements the problem specific rule that states: “if a piece blocks the only remaining clear path to any uncovered square then it fails, and if the blockage is the result of a square encircled by pillars execution is halted as no solution exists”. This example is different as it is not resource dependent like the previous one.

Step 1: The rule attempts to stop the placement of pieces that block the only clear path to any uncovered square as this will cause unnecessary backtracking resulting in wasted planning effort. If a square is found that is encircled by pillars execution is halted as no solution can exist for this situation in the puzzle world. The key operator is `place (P, S, O, D)`. The effect of the rule is that `place (P, S, O, D)` fails if the placement of the piece results in an isolated uncovered square. The encirclement condition causes complete failure of the main goal.

Step 2: There is no resource involved and a precondition to perform the above test does not exist.

Step 3: n/a.

Step 4: The lowest level precondition responsible for the required test does not exist but should be a precondition of `place (P, S, O, D)`.

Step 5: A complex state, `non_isolating (P, O, S, D)` should be added as a precondition of `place (P, S, O, D)`. This complex state would be true if no isolated uncovered squares resulted from the placement, also, it would halt execution if the pillar encirclement condition was detected.

Step 6: Since the test requires the piece variable to be instantiated and

the path checking performed also requires piece information it seems reasonable that `non_isolating(P,O,S,D)` should be the last precondition. The new preconditions of `place(P,S,O,D)` would be `holding(P)`, `uncovered(S)`, `uncovered(P,S,O)`, `clear_path(P,O,S,D)` (also a complex state), and `non_isolating(P,O,S,D)`.

Step 7: The following prolog code fragments show the `solve/3` clause that executes the complex state procedure for `non_isolating(P,O,S,D)` and the actual execution procedure (the execution halting function is performed by the `check_susp_sqs/7` predicate called by `check_path/6`):

```
solve(c-X,C,S):-
    X=non_isolating(P,Or,Sq,D),
    !,
    non_isolating(P,Or,Sq,D),
    !,
    solve(C,S).

non_isolating(P,Or,Sq,D):-
    matching_statement(state(max_dimension(Mx,My))),
    calc_direction(D,D1,D2,D3),
    (clear_path(P,Or,Sq,D1);
    check_path(P,Or,Sq,D1,Mx,My)),!,
    (clear_path(P,Or,Sq,D2);
    check_path(P,Or,Sq,D2,Mx,My)),!,
    (clear_path(P,Or,Sq,D3);
    check_path(P,Or,Sq,D3,Mx,My)),!.
```

Step 8: Test and adjust.

The above example has been implemented and performs as expected.

These examples illustrate the process used to implement problem specific rules in a Strips style planner in a general way. The procedure is not guaranteed to produce

correct and complete implementations but it does provide a framework to follow. As with many methodologies a great deal depends on the expertise of the person using it. The comments provided are only one instance of those possible and were meant as a guide only. The key thrust of the method is the use of complex states. They allow the Strips mechanism to be used with only minor additions and provide the designer with a degree of flexibility that is useful when a problem tests the limits of a Strips style state space representations.

5.4 Goal Analysis

Goal analysis or pre-processing of goals, is useful as a planning technique because it helps to limit search, however, it is not applicable in all situations. The puzzle world is one domain where it can be of great benefit as the information required to perform the subgoal analysis is contained in the start state. Also, goal analysis is required only once per solution because the state space can only be modified by the robot arm modeled.

The Puzzler uses analysis to reduce the need for backtracking caused by subgoal interactions. This is done by ordering the subgoals (squares to be covered in the grid) to be achieved so that failure of an operator due to the isolation of an uncovered square (discussed in the previous section) is almost eliminated. The heuristic details of the goal analysis were discussed in Chapter 4 and will not be repeated here.

The analysis is performed by the `perform_analysis/0` predicate prior to the beginning of the planning process.

5.5 Intelligent Backtracking

Backtracking is an important component of the search used by the prolog implementation

of the Strips style planner used in the Puzzler. Intelligent backtracking as presented in Chapter 2, uses various methods to add knowledge to backtracking to improve its performance. Two of the techniques, variable ordering and value ordering are used by the Puzzler. Recalling the domain specific rules implemented in the Puzzler listed on page 90, the first rule uses the principle of value ordering and second rule uses the principle of variable ordering. The analysis performed by the puzzler is also based on value ordering.

In general backtracking is important to planning because it is an important search technique and search compensates for a lack of knowledge. From the stand point of formalizing planning, properly implemented backtracking can assure completeness as Chapman points out. Completeness in this context refers to the fact that if a solution exists for a problem, it will eventually be found.

Chapter 6

A Framework for a Planning Development System

6.1 General

This chapter outlines a framework for a planning development system that could be the basis / environment of future planning research. This framework is derived from combining a variety of techniques used in the systems compared in chapter two, with the goal of providing a development system that can adequately deal with the classic planning issues, yet is flexible enough to become an evolving test-bed for old and new ideas.

6.2 Design Considerations

6.2.1 Desired Characteristics of the System

In developing any planning system it is necessary to first develop a set of desired characteristics that the system should display. Some of the characteristics to be discussed below may seem obvious but they are necessary to ensure that initial development decisions are made correctly.

The following is a list of the desired characteristics including brief explanations:

1. modular architecture : This is necessary for software engineering and research purposes. It allows the system to be developed by a team. Modules can be

developed and tested separately and functionality can be added or removed from the system as required.

2. allow different approaches to planning to be modeled : A key requirement of the system is for it to function as a development test bed. This dictates the need for a system capable of supporting a variety of planning approaches. Providing this amount of flexibility will result in a loss of efficiency but the advantages of being able to try different approaches on the same system outweigh efficiency considerations. Suitable implementations can always be recoded for efficiency if stand alone applications are required.
3. general yet rich data / plan representation: Of primary concern with respect to knowledge representation schemes are their flexibility, user friendliness, expressiveness and efficiency of processing . As this is a development system efficiency is not the main concern but it is necessary that the representation allows both simple and complicated domains to be used. The representation employed will be an evolving one. It is necessary to record as much information as possible during planning so that the exact behavior of the system is known.
4. debugging facility: To be useful as a development system a debugging facility is required. The debugger should be able to control execution step by step and provide listings of the planning information recorded during planning maintaining its temporal order. These two basic functions should be simple additions to the user interface.
5. ability to communicate efficiently: Efficient communications are important so that expansion to a distributed environment is possible. This will be limited to the adoption of an architecture that uses a global data structure which could be extended for distributed environments.

6. ability to utilize constraints: Although this is a particular technique used by all of the most promising planning systems its implementation impacts on almost every part of a planning system and therefore is mentioned here to ensure that it is properly planned for. Adding constraints is feasible if a constraint satisfaction mechanism can become a stand alone module that need not be there if a domain can be implemented without constraints. This will allow development to proceed initially without constraints.

6.2.2 Planning Technique and Approach Options

An implicit requirement of the system is that it be able to perform as a stand alone planner. It becomes useful at this point to discuss the various planning techniques available and specifically which of them are the most desirable to incorporate in the system. The system will have the flexibility to use all of the techniques if the required modules are developed. This discussion will define a set of initially desirable techniques and approaches.

Upon consideration of Chapter Two it follows that the state of the art in planning systems use a non-linear constraint posting approach that has the ability to plan in a hierarchy of abstraction spaces.²⁹ To this we would like to add to our system the ability to plan opportunistically, perform goal analysis, use means-ends analysis, use a skeletal-script-case based approach, perform meta-planning (which is partially covered by planning in a hierarchy of abstraction spaces), and the ability to use heuristic rules and heuristic evaluation functions. A particular planner to be implemented need not use all of these techniques but rather a subset consistent with the overall performance requirement.

²⁹ The advantages of the non-linear approach, constraints and abstraction spaces have already been discussed and therefore will not be repeated here.

As might be expected, a planning development system capable of coordinating such a variety of diverse techniques will have an architecture that is of a hybrid nature.

6.2.3 Planning System Architecture

A basic planning system is composed of:

1. data base;
2. operators;
3. control structure.

In terms of a planning development system this translates to:

1. global data structure;
2. the ability to define domain-independent, domain-dependent and control operator structures using abstraction levels;
3. flexible control structure capable of adopting different strategies, based on control operators for instance, dynamically.

This is similar to the blackboard architecture which consists of:

1. blackboard;
2. knowledge sources (KS);
3. control structure.

6.2.4 Control Cycles

An important difference between the planning system and the blackboard architectures is the implementation of the control structure. This thesis has identified three different control cycles from the systems studied in chapter two. From these cycles it is possible

to develop a hybrid cycle that can then be mapped onto a control structure for the planning development system.

A common control cycle used in planning systems consists of the following steps:

1. identify a difference between the desired state and the current state, ie. choose a goal to achieve;
2. choose an operator to achieve or reduce the goal;
3. apply/add the operator to the world/plan;
4. analyze plan for interactions and correct them.

The fourth step is not always defined as a separate step in this way, the interaction detection and correction function may be embedded in the other three in some, usually complicated, fashion. A number of systems do include this function as a separate entity in the form of heuristic rules implemented by critics or as part of the constraint mechanism used. This cycle makes it difficult to dynamically change control strategy. Different control strategies can be specifically included but flexibility to do so simply is lacking.

Chapman has reduced the above cycle to two steps due to the nature of his non-deterministic achievement procedure. These steps are:

1. choose a goal to achieve;
2. achieve the goal.

The non-deterministic achievement procedure incorporates the process of choosing operators to apply, detecting and correcting interactions and applying these processes to the world/plan.

The basic control cycle of a blackboard system contains three steps:

1. enumerate pending knowledge source (KS) activations;

2. choose one of the pending KS activations;
3. execute the chosen KS activation in its context.

Step two can utilize a pre-determined order of KS activations, an order based on the current status of the blackboard or by an ordering specified by a control KS. KS are also used to perform interaction detection and correction. A major advantage of the blackboard cycle is its flexibility and ease of meta-control through the use of the KS with their inherent modularity. This flexibility must be tempered with the high overhead incurred by these systems [12].

The hybrid control cycle incorporates a combination of the simplicity of Chapman's cycle (due to his non-deterministic achievement procedure) and the flexibility and modularity of the blackboard cycle. The desire to have a flexible and modular system overshadows the high overhead of the blackboard approach.

The hybrid planning control cycle contains three steps and is similar to the blackboard control cycle:

1. enumerate possible operator activations;
2. choose one of the pending operator activations;
3. execute the chosen operator activation in its context.

Step one allows all operators that are applicable to the current state (domain-independent, domain-dependent or control) to be considered. Step two chooses an operator based on the control strategy in effect. This could be a default control strategy or one that resulted from the execution of a domain-independent or control operator. Step three then executes the operator. The operator definition is coded to include the chosen strategy to counter any of the classic planning issues and specifically Chapman's non-deterministic

achievement procedure. If the operator chosen is one that operates on a goal the achievement procedure can then be used.

This control cycle is well suited for use within the control component of the planning development system architecture being developed. The cycle allows the use of domain-independent and control operators to provide a flexible control structure.

6.3 Hybrid Planning System Architecture

6.3.1 Planning System Components

The proposed planning system is comprised of four components: (figure 6.1)

1. control structure;
2. operators;
3. global data structure;
4. constraint satisfaction mechanism.

The control structure contains the top level control loop that utilizes applicable control operators to perform the various control functions. It also contains the user interface which adds an interactive component to the control function. The operators are similar in nature to blackboard knowledge sources in that all functions related to control, domain-independent and domain-dependent actions are performed by operators. The constraint satisfaction mechanism is configured as a stand alone module that can be invoked by a constraint operator. Wilkins noted that constraints in SIPE added considerably to the complexity of the planner as they interacted with all components of the system [1]. By keeping the constraint satisfaction mechanism separate it becomes an easily modifiable component. Constraint satisfaction information can be made available to operators

Figure 6.1 Hybrid Planning System Architecture

through the global knowledge structure. Lastly all information about the system is recorded in the global data structure. This facilitates communication between system components and will aid in future work on distributed planning systems.

The overall structure of the proposed system is similar to a blackboard system but is not intended to adopt the blackboard architecture completely. To this end blackboard terms have not been used to name components. The planning system components presented were designed to provide the system with a modular architecture that supports

a unifying structure. The three classes of operations, control, domain independent and domain dependent provide very different functions in a planning system but it helps unify the structure of the system if they differ in class and action only and not in their implementation and invocation/execution mechanisms. Also by limiting the available operators or by modifying the process of selecting an operator activation a variety of planning techniques can be modeled in a consistent and uniform manner.

The concept of the global data structure (gds) is not unique to this application. It does however allow all operators to find system and planning information in one general area and in the future this ease of communication will simplify any research done in the area of distributed planning.

6.3.2 Control Structure

The control structure provides the top level control loop of the planning system. The control loop contains three steps:

1. initialize global data structure with start state and goal information, including user specified control options;
2. execute the hybrid control cycle until there are no remaining items on the operator agenda;
3. output the plan or indicate failure.

The global data structure receives the required start state and goal information from either a prepared file or directly from the user via the user interface. Control options can also be specified at this time to ensure that a certain control plan is followed. These options may or may not be already present as control operators.

Once the global data structure has been initialized the enumerate / choose / execute cycle is started. These three steps are included as control operators but are explicitly called like functions by the control structure by executing their action components without the need for them to be triggered. The initialization and plan output functions are performed the same way using the *initialize* and *output_plan* control operators. A table summarizing the control operators appears at figure 6.2 . The areas referred to in figures 6.2 — 6.9 are areas of the global data structure (gds). The gds is discussed in more detail in section 6.3.3 .

The enumeration portion of the cycle causes the triggers for all operators to be tested by determining their truth values (either true:T or false:F) within the current context (world state). All operators whose trigger is T are added to the *operator_agenda*. At the end of the enumeration cycle the *operator_agenda* contains all of the operators that could be executed in that context (world state). The entries of the *operator_agenda* are called *operator activation records (OAR)*. The information contained in an OAR is shown in figure 6.3 . Each OAR is uniquely numbered for identification and debugging purposes. The cycle that triggered the OAR and the cycle number of the action that added the required information to the gds is also retained. The trigger condition and bindings are recorded as well as the node in the planning tree that is directly affected. A numeric rank is associated with each operator to indicate the relative importance of the operator. Modifiers, either positive or negative, can also be added to aid in the OAR selection process. All of this information is necessary to allow the operator to be executed in the context that it was triggered. It is important to note that the basic gds entry that contains the OAR also contains valuable information that may be required by an operator.

The choose portion of the cycle uses the *select_operator* control operator to choose

NAME	TRIGGER	ACTION
initialize_gds	direct execution by user interface	<ul style="list-style-type: none"> -add initialize_gds fact to control area -read start state and add individual states to state_space area -read goal(s) and add to goal_stack area -add goal node to planning_tree area -read control options and them to control area
enumerate_operator	direct execution by control structure	<ul style="list-style-type: none"> -add enumerate_operator fact to control area -determine truth value of operator triggers and for those that are true (T) : assign OAR # and add to op_agenda list, instantiate OAR and add to operator_agenda area (use old op_agenda as starting point and do not rewrite entries that are still applicable) -add op_agenda list to control area
select_operator	direct execution by control structure	<ul style="list-style-type: none"> -add select_operator fact to control area -add technique fact to control area -if technique specified apply technique operator and add modifiers to operator_agenda area if required -add strategy fact to control area -if strategy specified apply strategy operator and add modifiers to operator_agenda area if required -select OAR with highest rank , if tie choose next unsolved node in planning_tree -add choice(OAR #) fact to control area
execute_operator	direct execution by control structure	<ul style="list-style-type: none"> -add execute_operator fact to control area -execute action of operator selected -add new op_agenda list less executed oar to control area
output_plan	direct execution by control structure	<ul style="list-style-type: none"> -output plan or generate failure report via user interface -add output_plan fact to control area

Figure 6.2 Control Operators

Operator Activation Record	
Attribute	Description
OAR #	each OAR is uniquely numberd
Operator	operator name
Triggering Cycle,Event	cycle no. and event OAR was triggered
Trigger	actual trigger
Triggering Node	event/node that caused triggering
Rank	relative importance of OAR from operator attribute

Figure 6.3 Operator Activation Record

which operator to be executed based on its OAR. The `select_operator` uses a combination of default logic, the current control *technique* and domain *strategy* being used to evaluate each OAR on the operator agenda and choose one for execution. The domain independent technique and domain dependent strategy define operators that apply rank modifiers to gds entries. The operators they define are themselves controlled by the planning process. They will be discussed in more detail in Section 6.3.4 . The default logic selects the OAR with the highest overall rank (rank + modifiers) for execution³⁰. When there is more than one OAR with the same rank selection is based on a default goal achievement priority. Ranking is decided by placement in the directed graph representation of the planning process called the *planning tree* that is stored in the global data structure. The planning tree consists of *planning nodes* which are described in figure 6.4 . The planning tree is an explicit representation of the planning process. The planning tree is used to maintain the relationships between the planning nodes. These relationships are used to

³⁰ The Edinburgh Blackboard Shell by Jones et al also uses numerical ranking to choose among agenda items [37][13].

Planning Node	
Attribute	Description
Number	unique node number
Parent Node #	number of parent node
Type	either goal or operator
Name	of goal or operator
Variable List	[[var_symbol, binding],[,]...] or [[var_symbol, [constraint list], [,[]],...]
Status	-for goal: solved/unsolved -for operator executed/pending

Figure 6.4 Planning Node

help OAR selection and plan debugging. Selection among identical ranks or techniques based on different search orders use the position of nodes in the planning tree to apply modifiers to OAR rankings.

An important part of the planning node is identification. The node numbering scheme proposed allow the operators to quickly determine the position of the node in the tree with respect to its depth level, its parent and its child order. An example of the node numbering scheme is shown in figure 6.5 . Node [1-1-1-1-2] first_use(P) is a fifth level node with parent [1-1-1-1] and child-AND rank 2 (it is the second child of the AND). Nodes with decimal extensions such as [1-1-3-1.1] indicate children of an OR.

The default logic of OAR selection is augmented by the rules/logic of the current control technique being used at the time of the selection process. The control technique includes various search strategies, planning techniques and heuristics that can be used to

Figure 6.5 Node Numbering Scheme

guide the `select_operator` function. However, these techniques are domain independent and appear as domain independent operators. A selection of possible domain independent operators is shown in figure 6.6 . These operators will be discussed later in the chapter.

Once an OAR has been selected for execution, the *execute_operator* control operator executes the action portion of the operator in the context of the OAR.

Lastly, prior to generating a failure report, the system can receive help from the user. This information will be used if possible to generate an operator activation record.

6.3.3 Global Data Structure (GDS)

Throughout the planning process information about each event is recorded in the gds which is divided into eight areas. These areas are:

1. `operator_agenda`;
2. `control`;
3. `plan_steps`;
4. `constraints`;
5. `planning`;
6. `goal_stack`;
7. `state_space`;
8. `planning_tree`.

The gds has an implicit temporal dimension as well. The most current entry in a particular area is the last one that was stored.

A basic gds entry is described in figure 6.7 . The key attributes of an entry are the fact being stored and the area they are stored in. The `operator_agenda` contains OAR

NAME	TRIGGER	ACTION
technique_depth_first	-goal = complete_puzzle and no technique specified	-change technique to depth_first -add technique fact to control area
depth_first	-called by select operator and technique = depth_first	-add depth_first fact to planning area -analyze goals in goal_stack area with respect to nodes in planning_tree area -add modifiers to goals in goal_stack area
simple_establishment	-Truth_Value = U in goal_stack area -state is simple	-add action entry to planning area -check state_space area for current Truth_Value of goal -add new goal entry to goal_stack area with appropriate Truth_Value and status -if Truth_Value = T add new entry to planning_tree area with status = Solved.
complex_establishment	-Truth_Value = U in goal_stack area -state is complex	-add action entry to planning area -execute and record result: T or F -add new goal entry to goal_stack area with appropriate Truth_Value and status -if Truth_Value = T add new entry to planning_tree area with status = Solved
step_addition	truth value = F in goal area and no entry in plan_steps area with oar referencing same goal node	add action entry to planning area -check add list of domain operators and add applicable operators to applicable_ops list -add operators to planning area -add all preconditions of each operator in list to goal_stack area -add nodes to planning_tree area -add each operator in list to plan_steps with status = pending
satisfy_constraints	new constraint added to gds	-add action entry to planning area -invoke constarint satisfaction mechanism -add constraints to constraint area as required
declobber	interaction	-add action entry to planning area -apply appropriate declobbering operation -add gds entries as required

Figure 6.6 Domain Independent Operators

Global Data Structure (gds) Entry	
Attribute	Description
Cycle #	cycle which entry was added
Entry #	entry number within a particular cycle
Area	area of gds used
Fact	substance of entry
Status	Current or Old
Truth_Value	True(T), False(F) or Undetermined(U)
Rank_Modifier	+ or - a value
Node_#	node concerned

Figure 6.7 Global Data Structure Entry

and the `planning_tree` contains `planning_nodes`.

The control area contains entries indicating control actions that have been performed. The `enumerate_operators` control operator produces a control entry called *op_agenda* which contains a list of the OAR numbers produced during that cycle. The `select_operator` control operator produces a control entry called *choice* which contains the OAR selected from the `operator_agenda` for execution. The `execute_operator` control operator produces an entry called *execute* which contains the OAR executed. Other entries, including those concerning initialization and plan output are also recored in the control area.

The `plan_steps` area contains a record of the lowest level domain dependent planning operators considered. It is from these entries that the final plan is extracted. The entries, called *steps*, include the actual domain operator name, the planning node concerned,

a status attribute and the OAR that activated it. A step can be pending, executed or abandoned.

The constraints area contains entries that contain the current list of constraints at the end of a particular cycle. A *cstatus* attribute of satisfied or unsatisfied indicates whether the list has been operated upon by the constraint satisfaction mechanism.

The planning area contains entries that relate to all the non control operator executions. The entry, called an *action*, contains the operator being executed, the name of any related step or goal, the node concerned and the OAR that invoked it.

The *goal_stack* area, which contains entries called *goals*, and the *state_space* area, which contains entries called *states*, are similar in that their entries have the same attribute, the item (goal or state) being recorded.

The following gives example Prolog implementations of the *gds* entries described above:

1. `gds(Cycle_#, Entry_#, Area, Fact, Status, Truth_Value, Rank_Modifier, Node_#)`
2. `gds(Cycle_#, Entry_#, operator_agenda, [OAR_#, Operator, Triggering_Cycle, Trigger, Triggering_Event_Node, Rank], Status, Truth_Value, Rank_Modifier, Node_#)`
3. `gds(Cycle_#, Entry_#, control, op_agenda([001,002]), Status, Truth_Value, Rank_Modifier, Node_#)`
4. `gds(Cycle_#, Entry_#, control, choice(002), Status, Truth_Value, Rank_Modifier, Node_#)`
5. `gds(Cycle_#, Entry_#, control, execute(002), Status, Truth_Value, Rank_Modifier, Node_#)`

6. `gds(Cycle_#, Entry_#, plan_steps, [Item, Node, Pstatus, OAR], Status, Truth_Value, Rank_Modifier, Node_#)`
7. `gds(Cycle_#, Entry_#, constraints, [[Constraint_List], Cstatus], Status, Truth_Value, Rank_Modifier, Node_#)`
8. `gds(Cycle_#, Entry_#, planning, [Operator, SteporGoal, Node, OAR], Status, Truth_Value, Rank_Modifier, Node_#)`
9. `gds(Cycle_#, Entry_#, goal_stack, Goal, Status, Truth_Value, Rank_Modifier, Node_#)`
10. `gds(Cycle_#, Entry_#, state_space, State, Status, Truth_Value, Rank_Modifier, Node_#)`
11. `gds(Cycle_#, Entry_#, planning_tree, [Node_Number, Parent_Node_#, Type, Name, [Variable_List], Node_Status], Status, Truth_Value, Rank_Modifier, Node_#)`

6.3.4 Operator Structure

All changes to the gds are performed by operators, either directly or indirectly via the constraint satisfaction mechanism. A domain independent operator determines the need for constraint satisfaction and invokes the constraint satisfaction mechanism. The constraint satisfaction mechanism uses information from the gds and records its results there as well.

The use of the operator structure allows the operations performed by the system to share a common structure and invocation mechanism. However, the operators remain independent as they rely solely on the information recorded in the gds and do not communicate with each other directly.

The operators are divided into three groups:

1. control;
2. domain independent;
3. domain dependent.

The control group (figure 6.2) provides the operators needed for the hybrid control cycle and any miscellaneous control functions that are required. The domain independent group (figure 6.6) provides the operators necessary for general planning functions such as goal achievement, interaction detection and correction, constraint manipulation and implementing different planning techniques. The domain dependent group provides operators related to specific domains. It includes domain dependent strategies, refinement operators and the low level operators that make up the actual plan steps. A selection of domain dependent operators are shown in figure 6.8 .

NAME	TRIGGER / PRECONDITION	ACTION
strategy_analyze_board	-goal = complete_puzzle and no strategy specified	-change strategy to analyze_board -add strategy fact to control area
analyze_board	called by select_operator / strategy = analyze_board and no analyze_board entry in planning area	-add analyze_board fact to planning area -analyze board and calc modifiers -add goals to goal_stack area with new modifiers -add nodes to planning_tree area with new modifiers

Figure 6.8 Domain Dependent Operators (Continued . . .)

NAME	TRIGGER / PRECONDITION	ACTION
select(P)	<ul style="list-style-type: none"> -select(P) in plan_steps area with status = pending -all preconditions in state_space area with status = Current, Truth_Value = T / available(P), first_use(P), handempty 	<ul style="list-style-type: none"> -add check_preconditions fact to planning area - re-establish preconditions (if there is a failure add preconditions to goal_stack area and exit) -add delete list to state_space area with status = Current, Truth_Value = F -add add list to state_space area with status = Current, Truth_Value = T -add solved node to planning_tree area -change status of operator in plan_step area to executed -change status of other operators in plan_steps area for same node to abandoned -update planning_tree and goal_stack areas of parent node as required
place(P,S,O)	<ul style="list-style-type: none"> -place(P,S,O) in plan_steps area with status = pending -all preconditions in state_space area with status = Current, Truth_Value = T / holding(P), uncovered(S), uncovered(P,S,O), clear_path(P,O,S,D), non_isolating(P,O,S,D) 	<ul style="list-style-type: none"> -add check_preconditions fact to planning area - re-establish preconditions (if there is a failure add preconditions to goal_stack area and exit) -add delete list to state_space area with status = Current, Truth_Value = F -add add list to state_space area with status = Current, Truth_Value = T -add solved node to planning_tree area -change status of operator in plan_step area to executed -change status of other operators in plan_steps area for same node to abandoned -update planning_tree and goal_stack areas of parent node as required
rotate_90(P,S) (rotate_180(P,S) and rotate_270(P,S) are similar, simply substitute the appropriate angle)	<ul style="list-style-type: none"> -rotate_90(P,S) in plan_steps area with status = pending -all preconditions in state_space area with status = Current, Truth_Value = T / holding(P), orientation(P,O), clear_area(P,s,90) 	<ul style="list-style-type: none"> -add check_preconditions fact to planning area - re-establish preconditions (if there is a failure add preconditions to goal_stack area and exit) -add delete list to state_space area with status = Current, Truth_Value = F -add add list to state_space area with status = Current, Truth_Value = T -add solved node to planning_tree area -change status of operator in plan_step area to executed -change status of other operators in plan_steps area for same node to abandoned -update planning_tree and goal_stack areas of parent node as required

Figure 6.8 Domain Dependent Operators (Continued ...)

NAME	TRIGGER / PRECONDITION	ACTION
refine_goal	goal = complete_puzzle and no entry in planning area	-add refine_goal to planning area -add generated goals to goal_stack area -add nodes to planning_tree area
complex_states	goal = name of complex_state	perform action defined by complex_state

Figure 6.8 Domain Dependent Operators

The domain independent operators are important as they allow the flexibility to model different planning techniques. These techniques include different search strategies as well as the basic planning functions of simple establishment, step addition, goal interaction detection and correction, and constraint satisfaction. A domain independent variable named *technique* contains the current technique being used. The domain independent operator that is named as the current technique is used to add modifiers to gds entries containing OARs based on the technique that is to be followed. In this way it affects the OAR selection and ultimately the planning process. The technique operator defined in figure 6.6, *depth_first*, is used to model a depth first criteria for goal selection. It adds modifiers to goals in the goal_stack area so that operator activations are chosen based on the depth first ordering of goals in that area. The information contained in the planning_tree is used for this process.

The domain dependent operators include the low level planning operators used in the domain as well as operators that deal with the domain level strategy. The *strategy* is a domain dependent version of the domain independent technique. It allows domain heuristics and any other domain level rule to be implemented at a level above

that of the low level operators. As with the domain independent technique the rank modifiers of the gds entries are manipulated. The strategy operator defined in figure 6.8, *analyze_board*, is used to model a domain strategy in which the cells are pre-ordered so that piece placement does not result in any isolated cells thus reducing backtracking. The *analyze_board* operator performs this analysis and adds modifiers to each goal so that goal achievement follows the order determined by this analysis.

It should be noted that the low level operators require both a trigger and preconditions. For the STRIPS style operators of the puzzle world the trigger includes the achievement of preconditions and the presence of an entry in the *plan_steps* area indicating that operation was invoked by the *step_addition* operator. The *step_addition* domain independent operator uses the add list information of the STRIPS style operators to determine suitability for invocation.

Operators are similar to Hayes-Roth's knowledge sources and are composed of nine attributes. These are name, *problem_domain*, description, trigger, precondition, variable, action, rank and class. Additional attributes can also be added. A description of the operator attributes appears in figure 6.9 . A key attribute of an operator is its rank. The rank provides the basis for the addition of modifiers. A sample ranking for the operators discussed so far appears in figure 6.10 . Operators with a rank of n/a are called directly and do not need to compete for execution. This can be modified in the future.

6.3.5 Constraint Satisfaction Mechanism

The constraint satisfaction mechanism (csm) is used to satisfy symbolic constraints as they are added to the gds. An operator which has a high ranking is triggered by the addition of new constraints. Its action invokes the csm thereby ensuring that all other

Operator Attributes	
Attribute	Description
Name	name of operator
Problem_Domain	domain of use
Description	characteristic behavior
Trigger	event/state predicate for triggering
Precondition	state based predicates required for execution
Variables	variable bindings or constraints used by operator
Action	program of gds changes
Rank	relative priority of operator
Class	control, domain independent or domain dependent

Figure 6.9 Operator Attributes

operators have the most recent results at their disposal. If an over constrained situation results steps can be taken to backtrack before the next enumeration cycle. This feature provides early pruning of the search space which is one of the advantages of using constraints.

The csm is left as a separate module so that it can be readily adjusted and modified. It is probable that the first implementation of the csm will be inefficient and will become the bottle neck of the system. This is necessary if we are to learn about constraints, however, the system is structured so that domains that can be solved without constraints can be implemented even if the csm is absent. The csm can then be left as a separate development project.

Rank	Operator	Class
n/a	initialize_gds	control
n/a	enumerate_operator	control
n/a	select_operator	control
n/a	execute_operator	control
n/a	output_plan	control
120	refine_goal	domain depend
110	strategy_analyze_board	domain depend
100	technique_depth_first	domain indep
60	satisfy_constraints	domain indep
55	declubber	domain indep
50	simple_establishment	domain indep
50	complex_establishment	domain indep
40	step_addition	domain indep
35	select(P)	domain depend
35	place(P, S, O)	domain depend
35	rotate_90(P, S) etc	domain depend
n/a	analyze_board	domain depend
n/a	depth_first	domain indep
n/a	complex_states	domain depend

Figure 6.10 Operator Rankings

6.4 Planning Example using the Puzzle World

To help understand the operation of the proposed planning system an example from the puzzle word has been simulated. The problem chosen is shown in figure 6.11 . Appendix C contains a listing of global data structure (gds) entries generated during the simulation. The gds entries were produced according to the operator action specifications detailed in

figures 6.2, 6.6, and 6.8 . The `planning_tree` shown in figure 6.5 can also be used as an aid to understanding the example as it contains the solution tree. Also, when referring to Appendix C, the following points should be note:

1. Entries are temporally ordered, from start to finish;
2. Entries have been modeled on prolog style predicates, capital letters denote variables and square brackets [], denote lists;
3. The variable list attribute has been omitted from the `planning_nodes` of the `planning_tree` area;
4. Entries will be referred to by their cycle and entry numbers: 1,1 or 2,1 ;
5. Attributes that seem repetitive or unnecessary have been left intact to provide room for future modifications.

The example begins by performing the first step of the control loop of the system, initialization. The first cycle, entries 1,1 to 1,8 perform this step. The second step, executing the enumerate / select / execute cycle makes up the largest part of the entries, 2,1 to 50,3. Within this group the planner cycles through the three steps a number of times. For example, cycle 2: enumerate, cycle 3: select, cycle 4: execute, cycle 5: enumerate, cycle 6: select and cycle 7: execute. Each entry within a cycle corresponds to the action specified for the operator being executed. The `enumerate_operator`, `select_operator` and `execute_operator` are control operators whose action description appear in figure 6.2 . The last step in the control loop, plan output, is performed at entry 50,4.

We will now discuss a detailed example of the enumerate / select / execute cycle. Consider enumeration cycle 17 and the `enumerate_operator` action listed in figure 6.2. Entry 17,1 records the start of the cycle and 17,2 to 17,6 adds OARs to the `operator_agenda`. These activations are based upon the need to establish the truth value

Figure 6.11 Planning Example

of goal_stack entries 16,3 to 16,7. The final entry of cycle 17, 17,7 records the list of operator activations ready to be executed. The select cycle begins at 18,1 with the recording of the start of the cycle. Entry 18,2 invokes the current technique operator. The current operator is then recorded in the planning area, 18,3. The result of the depth_first operator is the new goal stack entries that have updated rank modifiers received from the technique. The modifiers decrease left to right thus modeling the depth first, left right execution of prolog. Entry 18,14 ends the cycle by recording the highest ranked (rank + modifier) OAR using the choice fact. The select cycle is followed by execute cycle 19. The cycle starts by recording which operator it will be executing followed by an entry in the planning area to record and execute the operator. The result of the

operation is then written to the `goal_stack`. The final entry is 19,4 where the remainder of the `operator_agenda` are saved as they form the basis for the next enumeration cycle. It is recommended that the reader take time to follow the entire example through to completion.

Although the example is not very complicated, it demonstrates that the proposed framework is flexible enough to become a development system due to the nature of the possible operators that could be defined and that some operators could be used to adequately deal with search, the frame problem and interacting subgoals. This flexibility also allows the modeling of various planning techniques and approaches which aids the evaluation process.

Chapter 7

Conclusions

This study of artificial intelligence planning was based on the premise that planning is an important component of artificial intelligence. The investigation presented provides an in depth examination of planning as a field of artificial intelligence and, in particular, provides a comparative survey of planning systems that have been important to the field and has concluded that planning is a complex field that uses numerous techniques and approaches, some of which are more promising than others. Planning also impacts on other important areas within artificial intelligence including search, knowledge representation, learning and problem solving. The most promising planning systems use the non-linear constraint posting and hierarchical approaches. The particular approach and techniques used are effected largely by the classic planning problems of search, interacting subgoals and the frame problem.

Search is a key problem in planning. Due to the large search spaces found in planning problems early pruning of useless choices within the search space is essential. It is important that planners be able to incorporate various search strategies from knowledge poor to knowledge rich. In this respect intelligent backtracking has been found to be a technique that increases the performance of knowledge poor systems. Increasing the search performance of knowledge rich planning systems is best done by focusing on the solution to the interacting subgoals problem. Interacting subgoals seem best handled by constraints but they are complex. However, more experience is required

working with implementations of symbolic constraint satisfaction mechanisms before the technique can be fully evaluated.

The frame problem is also a key problem in planning as its solution affects the complexity of the domains permitted, the knowledge representation used and the extent to which the planning techniques used can be formalized. The Strips assumption and the use of the Strips style add/delete list representation has been found to be a reasonable solution to the frame problem that is both implementable and reliable. It suffers from the requirement to use a restrictive domain representation but its principles can be extended to apply to more complex domains. It is also the basis for one attempt to formalize some planning techniques. Planning has reached the point where proven techniques need to be formalized so that the field can progress. The standardization of planning concepts and terminology also needs to be addressed beyond that which was presented in Chapter 3 to aid future research efforts.

A planner called the Puzzler was implemented by the Author to demonstrate some planning concepts and key techniques while providing a vehicle to obtain a better understanding of the problems associated with planning. This experience has resulted in the outline of a technique that allows the incorporation of problem specific rules in a Strips style planner in a general way. Other techniques demonstrated by the Puzzler included complex states, goal analysis and intelligent backtracking. Complex states were shown to be useful for adding complexity to Strips style planners that would normally be restricted to simpler domains. Goal analysis has proven to be a simple method of preventing interacting subgoals in terms of Strips style planners. These points also support the viewpoint that Strips style planners are still a useful tool for planning research.

This thesis has also lead to the identification of the requirement for a planning

development system that can be used to develop and model planning approaches and techniques. The experience gained while modeling approaches and techniques using such a system is necessary to perform complete evaluations on planning approaches and techniques. The Puzzler demonstrated this in terms of the techniques and approaches it demonstrated. To this end, a framework for a planning development system was presented.

The framework for the planning development system presented is modular and flexible. Its unifying concept of knowledge source like operators enables a variety of operator types (control, domain dependent and domain independent) to be invoked using a common control mechanism and global data structure. The separate constraint satisfaction mechanism allows the technique of constraint posting to be investigated further while allowing the constraint satisfaction mechanism to be modified independently. This enables the planning development system to be used on domains that do not use constraints even if the mechanism is unavailable. Also the use of a global data structure will aid future research in the area of distributed planning.

However, at present only a framework is presented and future research will be needed to turn this framework into a design ready for implementation. Much is still to be gained by planning research as numerous approaches and techniques are supported by different researchers. The existence of a tool such as a planning development system will aid in the complex task of evaluating and developing these techniques and approaches.

References

- [1] D. Wilkins, "Domain-independent planning: Representation and plan generation," *Artificial Intelligence*, vol. 22, pp. 269–301, 1984.
- [2] N. Nilsson, *Principles of Artificial Intelligence*. Los Altos: Morgan Kaufmann, 1980.
- [3] P. Cohen and E. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 3. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1982.
- [4] A. Tate, "A review of knowledge-based planning techniques," in *Expert Systems 85: Proceedings of the 5th Technical Conference of the British Computer Society Specialist Group on Expert Systems*, (Cambridge, UK), pp. 89–111, Cambridge University Press, 1985.
- [5] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, pp. 189–208, 1971.
- [6] A. Barr and E. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 2. Los Altos, CA: William Kaufmann, 1982.
- [7] E. D. Sacerdoti, "The nonlinear nature of plans," in *Advance Papers of the Fourth International Joint Conference of Artificial Intelligence*, (Tbilisi, Georgia, USSR), pp. 206–214, 1975.
- [8] M. Stefik, *Planning With Constraints*. PhD thesis, Stanford University, 1980.
- [9] M. Stefik, "Planning with constraints (molgen: Part 1)," *Artificial Intelligence*, vol. 16, no. 2, pp. 111–139, 1981.
- [10] M. Stefik, "Planning and meta-planning (molgen: Part 2)," *Artificial Intelligence*, vol. 16, no. 2, pp. 141–169, 1981.

- [11]B. Hayes-Roth, F. Hayes-Roth, S. Rosenschein, and S. Cammarata, "Modeling planning as an incremental, opportunistic process," in *Sixth International Joint Conference On Artificial Intelligence*, (Tokyo), pp. 375-383, 1979.
- [12]B. Hayes-Roth, "A blackboard architecture for control," *Artificial Intelligence*, vol. 26, pp. 251-321, 1985.
- [13]R. E. T. Morgan, ed., *Blackboard Systems*. New York: Addison-Wesley Publishing Company, Inc., 1988.
- [14]S. Smith, M. Fox, and P. Ow, "Constructing and maintaining detailed production plans: Investigations into the development of knowledge-based factory scheduling systems," *AI Magazine*, pp. 15-61, Fall 1986.
- [15]S. F. Smith, "A constraint-based framework for reactive management of factory schedules," in *International Conference on Expert Systems and the Leading Edge in Production Planning and Control*, (Charleston), May 1987.
- [16]D. E. Wilkins, "Casual reasoning in planning," *Computational Intelligence*, vol. 4, no. 4, pp. 373-380, 1988.
- [17]S. A. Vere, "Plannint in time: Windows and durations for activities and goals," *I.E.E.E. Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, no. 3, pp. 246-267, 1983.
- [18]R. Fikes, P. Hart, and N. Nilsson, "Learning and executing generalized robot plans," in *Readings in Artificial Intelligence*, pp. 231-249, Palo Alto, CA: Tioga, 1981.
- [19]A. Barr and E. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 1. Los Altos, CA: William Kaufmann, 1981.
- [20]D. Chapman, "Planning for conjunctive goals," *Artificial Intelligence*, vol. 32, pp. 333-377, 1987.

- [21]R. D. Korf, "Search: A survey of recent results," in *Exploring Artificial Intelligence: Survey Talks for the National Conference of Artificial Intelligence* (H. S. . A. eds, ed.), pp. 197–237, Ca: Morgan Kaufman, 1988.
- [22]E. D. Sacerdoti, "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence*, vol. 5, pp. 115–135, 1974.
- [23]P. J. Hayes, "The frame problem and related problems in artificial intelligence," in *Readings in Artificial Intelligence* (B. L. W. N. J. Nilsson, ed.), pp. 223–230, Palo Alto: Tioga, 1981.
- [24]Y. Shoham, "Temporal logics in ai: Semantical and ontological considerations," *Artificial Intelligence*, vol. 33, pp. 89–104, 1987.
- [25]J. McCarthy and P. J. Hayes, "Some philosophical problems form the standpoint of artificial intelligence," in *Readings in Artificial Intelligence* (B. L. W. N. J. Nilsson, ed.), pp. 431–450, Palo Alta, Ca: Tioga, 1981.
- [26]R. Waldinger, "Achieving several goals simultaneously," in *Readings in Artificial Intelligence*, pp. 250–271, Palo Alto, CA: Tioga, 1981.
- [27]G. W. Ernst and A. Newell, *GPS: A Case Study in Generality and Problem Solving*. New York: Academic Press, 1969.
- [28]J. R. Slagle, *Artificial Intelligence: The Heuristic Programming Approach*. New York: McGraw-Hill, 1971.
- [29]M. Genesereth and N. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, CA: Morgan Kaufmann Publishing Inc., 1987.
- [30]A. Tate, "Interacting goals and their use," in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, (Tblisi, Georgia USSR), pp. 215–218, 1975.

- [31]H. Nii, "Part one: Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures," *AI Magazine*, pp. 38–53, Summer 1986.
- [32]H. Nii, "Part two: Blackboard systems: Blackboard application systems, blackboard systems from a knowledge engineering perspective," *AI Magazine*, pp. 82–106, August 1986.
- [33]L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty," *Computing Surveys*, vol. 12, no. 2, pp. 213–253, 1980.
- [34]K. Chang, *A Planning Model with Problem Analysis and Operator Hierarchy*. PhD thesis, University of Cincinnati, Cincinnati, 1986.
- [35]K. Chang and W. G. Wee, "A planning model with problem analysis and operator hierarchy," *I.E.E.E. Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 5, pp. 672–675, September 1988.
- [36]Logicware Inc., *MProlog Language Reference, Release 2.1*. Toronto, Canada: Logicware Inc., September 1985.
- [37]J. Jones and M. Millington, "An edinburgh prolog blackboard shell," tech. rep., University of Edinburgh, Edinburgh, 1986.

Appendix A

The Puzzler Source Code

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PROGRAM:      t8astrips.pro
%
%   VERSION:      4.1
%
%   TITLE:        STRIPS Planner
%
%   AUTHOR:       C. Roth
%
%   PROJECT:      MEng Thesis
%
%   DATE:         March 1989
%
%   SOFTWARE:     MProlog v2.1
%
%   DESCRIPTION:   This is a strips style planner used to solve
%                  problems in a domain called the puzzle world.
%
%   This version includes:
%
%       path checking in four directions North,
%       South, East and West.
%
%       ability to use pieces of different shapes but similar
%       max rectangular area dimensions.
%
%       5 different piece types: a,b,c,d,e,f.
%
%   ****> Goal analysis and reordering prior to solving: -two
%   types of conflict causing subgoal are identified and placed
%   at the head of the subgoal list. Within these two lists the
%   more critical of the two types of subgoals are further
%   ordered. Also, if a completed enclosed square is detected
%   the analysis and plan are halted.
%
%   The following rules have been implemented:
%
%   1.      If a piece fails then no other pieces of the same
%   size will fit and is therefore not even tried.
%
%   2.      If a piece blocks the only remaining path to any
%   uncovered square then it fails. If this blockage is the
%   result of a square encircled by pillars then execution is
%   halted as no solution exists.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
module t8astrips.
all_symbolic.
body.

dynamic (add_list/2).
dynamic (del_list/2).
dynamic (preconditions/2).
dynamic (start_state_of_grid/1).
dynamic (start_state_of_pieces/1).
dynamic (goal_state/1).
dynamic (state/1).
dynamic (solution/1).

```

[illegible]

```

%      PREDICATE:      plan/0
%
%      DESCRIPTION:    overall control loop of planner.  Initializes
%      the start state, initializes the goal stack, analyzes and
%      reorders sub-goals, solves the goal and then writes the plan
%      to the terminal.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
plan:-
    init_start_state,
    init_goal_stack(S),
    perform_analysis,
    !,
    solve(C,S),
    write_plan.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      PREDICATE:      init_start_state/0
%
%      DESCRIPTION:    initializes the start state by adding the
%      start state read in to the current state.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init_start_state:-
    matching_statement(start_state_of_grid(X)),
    add_list(X),
    matching_statement(start_state_of_pieces(Y)),
    add_list(Y),
    !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      PREDICATE:      init_goal_stack/1
%
%      DESCRIPTION:    initializes the goal stack by pushing the
%      goal state read in onto the stack.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init_goal_stack(S):-
    matching_statement(goal_state(X)),
    push(g-X, [], S),
    !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      PREDICATE:      perform_analysis/0
%
%      DESCRIPTION:    performs analysis on covered squares then
%      reorders subgoals before adding uncovered squares to state.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
perform_analysis:-
    matching_statement(start_state_of_grid(X)),
    get_cov_sqs_list(X, [], Cov_sqs_list),
    fss(Cov_sqs_list, [], [], L1, L2, L3),
    remove_cov_sqs(Cov_sqs_list, L1, L2, L3, L1r, L2r, L3r), !,
    rev_order_L3(L3r, L3_rev_ordered),
    prod_pri_uncov_list(L2r, L3_rev_ordered, Pri_uncov_list),

```

```

    matching_statement(state(max_dimension(Mx,My))),
% following predicate is commented out to perform test without
% analysis
    handle_covered(add_uncovered,Pri_uncov_list),
    init_uncov_sqs(norm,Mx,My,1,1).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      get_cov_sqs_list/3
%
%   DESCRIPTION:    returns a list of the covered squares
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

get_cov_sqs_list([],Cov_sqs_list,Cov_sqs_list):-!.
get_cov_sqs_list([covered(S)|T],Tm,Cov_sqs_list):-
    get_cov_sqs_list(T,[S|Tm],Cov_sqs_list).
get_cov_sqs_list([H|T],Tm,Cov_sqs_list):-
    get_cov_sqs_list(T,Tm,Cov_sqs_list).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      fss/6
%
%   DESCRIPTION:    separates potentially difficult sub-goals
%                   into three categories and returns them as lists.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

fss([],L1,L2u,L1,L2,L3):-
    fss3(L1,L2u,[],L3),
    makel2(L2u,L3,L2),!.
fss([H|T],L1,L2,L1f,L2f,L3):-
    fss1(H,T,L1,L2,L1u,L2u),
    fss(T,L1u,L2u,L1f,L2f,L3),!.

```

```

fss3(L1,[],L3,L3):-!.
fss3(L1,[H|T],Tm,L3):-
    member(H,L1),!,
    fss3(L1,T,[H|Tm],L3),!.
fss3(L1,[H|T],Tm,L3):-
    fss3(L1,T,Tm,L3),!.

```

```

makel2(L2,[],L2):-!.
makel2(L2u,[H|T],L2):-
    delete(L2u,H,L2n),!,
    makel2(L2n,T,L2).

```

```

fss1(A,[],L1,L2,L1,L2):-!.
fss1(A,[B|T],L1,L2,L1f,L2f):-
    fss2(A,B,L1,L2,L1u,L2u),
    fss1(A,T,L1u,L2u,L1f,L2f),!.

```

```

fss2([X1,Y],[X2,Y],L1,L2,L1,L2u):-
    (X1<X2,!,
    prody2(X1,X2,Y,L2,L2u);
    X2<X1,!,
    prody2(X2,X1,Y,L2,L2u)).
fss2([X,Y1],[X,Y2],L1,L2,L1,L2u):-
    (Y1<Y2,!,
    prodx2(Y1,Y2,X,L2,L2u);
    Y2<Y1,!,
    prodx2(Y2,Y1,X,L2,L2u)).
fss2([X,Y],[X1,Y1],L1,L2,[X,Y1],[X1,Y]|L1,L2):-!.

```

```
prody2(XL,XH,Y,Tm,L2):-  
    B1 is XL + 1,  
    B1 < XH,  
    prody2(B1,XH,Y,[{B1,Y}|Tm],L2),!.  
prody2(XL,XH,Y,L2,L2):-!.  
prodx2(YL,YH,X,Tm,L2):-  
    B1 is YL + 1,  
    B1 < YH,  
    prodx2(B1,YH,X,[{X,B1}|Tm],L2),!.  
prodx2(YL,YH,X,L2,L2):-!.
```

#####

```
% . PREDICATE:      remove_cov_sqs/7  
% DESCRIPTION:     returns three lists with covered squares removed  
#####
```

```
remove_cov_sqs([],L1,L2,L3,L1,L2,L3):-!  
remove_cov_sqs([H|T],L1,L2,L3,L1r,L2r,L3r):-  
    delete(L1,H,L1t),  
    delete(L2,H,L2t),  
    delete(L3,H,L3t),  
    remove_cov_sqs(T,L1t,L2t,L3t,L1r,L2r,L3r),!.
```

#####

```
% PREDICATE:      rev_order_L3/2  
% DESCRIPTION:     order the list according to analysis rules,  
%                 list returned is in reverse order as it is reversed again by  
%                 prod_pri_uncov_list/3.  
#####
```

```
rev_order_L3(L3,L3_rev_ordered):-  
    weight_L3(L3,[],L3_weighted),  
    weightsort(L3_weighted,L3_sorted),  
    remove_weight(L3_sorted,L3_rev_ordered),!.  
rev_order_L3(_,_) :-  
    write("isolated square detected during analysis"),nl,  
    write(" - no solution possible"),nl,fail.
```

#####

```
% PREDICATE:      weight_L3/3  
% DESCRIPTION:     returns a weighted list  
#####
```

```
weight_L3([],X,X):-!  
weight_L3([H|T],Lt,L3_weighted):-  
    get_weight(H,W),  
    weight_L3(T,[{H,W}|Lt],L3_weighted),!
```

#####

```
% PREDICATE:      get_weight/2  
% DESCRIPTION:     returns the weight of a square  
#####
```

```

get_weight([X,Y],W):-
    matching_statement(start_state_of_grid(L)),
    geth1(xh,X,Y,L,Xh),
    geth1(xl,X,Y,L,Xl),
    geth1(yh,X,Y,L,Yh),
    geth1(yl,X,Y,L,Yl),
    (Xh=open;Xl=open;Yh=open;Yl=open),
    calc_weight(xh,L,X,Y,Xh,0,Xhw),
    calc_weight(xl,L,X,Y,Xl,0,Xlw),
    calc_weight(yh,L,X,Y,Yh,0,Yhw),
    calc_weight(yl,L,X,Y,Yl,0,Ylw),
    W is Xhw + Xlw + Yhw + Ylw,
    !.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      geth1/5
%
%      DESCRIPTION:    returns the next highest or lowest x or y
%                      coord of surrounding covered square on the same row or col
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

geth1(xh,X,Y,L,Xh):-
    member(covered([Xh,Y]),L),
    Xh > X,!.
geth1(xl,X,Y,L,Xl):-
    member(covered([Xl,Y]),L),
    X > Xl,!.
geth1(yh,X,Y,L,Yh):-
    member(covered([X,Yh]),L),
    Yh > Y,!.
geth1(yl,X,Y,L,Yl):-
    member(covered([X,Yl]),L),
    Y > Yl,!.
geth1(_,_,_,_open):-!.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      calc_weight/7
%
%      DESCRIPTION:    returns number of open squares between square
%                      and next covered square, in a given direction.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

calc_weight(_,_,_,_open,_0):-!.
calc_weight(xh,L,X,Y,Xh,Xwo,Xwn):-
    Xt is X+1,
    Xt < Xh,
    not member(covered([Xt,Y]),L),!,
    Xwt is Xwo+1,
    calc_weight(xh,L,Xt,Y,Xh,Xwt,Xwn),!.
calc_weight(xl,L,X,Y,Xl,Xwo,Xwn):-
    Xt is X-1,
    Xt > Xl,
    not member(covered([Xt,Y]),L),!,
    Xwt is Xwo+1,
    calc_weight(xl,L,X,Y,Xl,Xwt,Xwn),!.
calc_weight(yh,L,X,Y,Yh,Xwo,Xwn):-
    Yt is Y+1,

```

```

        Yt < Yh,
        not member(covered([X,Yt]),L),!,
        Xwt is Xwo+1,
        calc_weight(yh,L,X,Yt,Yh,Xwt,Xwn),!.
    calc_weight(Yl,L,X,Y,Yl,Xwo,Xwn):-
        Yt is Y-1,
        Yt > Yl,
        not member(covered([X,Yt]),L),!,
        Xwt is Xwo+1,
        calc_weight(Yl,L,X,Yt,Yl,Xwt,Xwn),!.
    calc_weight(_,_,_,_,_X,X):-!.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      weightsort/2
%
%      DESCRIPTION:    sorts list ascending by weight
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

weightsort([X|Xs],Ys):-weightsort(Xs,Zs),insert(X,Zs,Ys).
weightsort([],[]).
insert(X,[],[X]).
insert([X,Xn],[Y,Yn|Ys],[Y,Yn|Zs):-
    Xn>Yn, insert([X,Xn],Ys,Zs).
insert([X,Xn],[Y,Yn|Ys],[X,Xn],[Y,Yn|Ys):-
    Xn<=Yn.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      remove_weight/2 & 3
%
%      DESCRIPTION:    removes weight from squares in list and
%                      reverses the list as prod_pri_uncov_list/3 reverses it again.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

remove_weight(With,Without):-
    remove_weight(With,[],Without).
remove_weight([X,W|T],Temp,Without):-
    remove_weight(T,[X|Temp],Without).
remove_weight([],Without,Without):-!.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      prod_pri_uncov_list/3
%
%      DESCRIPTION:    returns list of sub-goals that should be
%                      solved first.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

prod_pri_uncov_list(X,[],X):-!.
prod_pri_uncov_list(L2,[H|T],Pri):-
    prod_pri_uncov_list([H|L2],T,Pri).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      init_uncov_sqs/5
%
%      DESCRIPTION:    uses max dimension of puzzle to add uncovered
%                      squares to state (that have not previously been added).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

init_uncov_sqs(norm,Mx,My,Cx,Cy):-

```

```

Cx <= Mx,
Cy <= My,
((not matching_statement(state(uncovered([Cx,Cy])))),
not matching_statement(state(covered([Cx,Cy])))),
add_statement(state(uncovered([Cx,Cy]))),
Cynext is Cy + 1;
(Cynext is Cy + 1)),
init_uncov_sqs(norm,Mx,My,Cx,Cynext),!.
init_uncov_sqs(norm,Mx,My,Cx,Cy):-
Cx <= Mx,
Cxnext is Cx + 1,
init_uncov_sqs(norm,Mx,My,Cxnext,1),!.
init_uncov_sqs(norm,Mx,My,Cx,Cy):-!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      solve/2
%
%      DESCRIPTION:    pops the next item from the stack and then
%                      calls solve/3 to solve it.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

solve(C,S):-
pop(Y,S,N),
!,
solve(Y,C,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      solve/3
%
%      DESCRIPTION:    solves stack entry, using different predicate
%                      for goals, preconditions and actions.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

solve(g-X,C,S):-
push(g-X,S,N),
matching_statement(state(uncovered(Y))),
push(c-covered(Y),N,Ns),
solve(C,Ns).

solve(g-X,C,S):-
matching_statement(state(uncovered(_))),
!,fail.

solve(g-X,C,S).

solve(c-X,C,S):-
X=clear_area([P,D],Sq,Or),
matching_statement(state(max_dimension(MX,MY))),
!,
calc_coverage(P,Sq,MX,MY,Or,Osc),
!,
handle_covered(check_uncovered,Osc),
!,
solve(C,S).

solve(c-X,C,S):-
X=uncovered([P,D],Sq,Or),
matching_statement(state(max_dimension(MX,MY))),

```



```

    matching_statement(state(orientation([P,D],Or))),
    calc_coverage(P,Sq,MX,MY,Or,Osc),
    handle_covered(check_uncovered,Osc),
    solve(C,S).

solve(c-X,C,S):-
    X=clear_path(P,Or,Sq,Dir),
    !,
    clear_path(P,Or,Sq,Dir),
    !,
    solve(C,S).

solve(c-X,C,S):-
    X=non_isolating(P,Or,Sq,D),
    !,
    non_isolating(P,Or,Sq,D),
    !,
    solve(C,S).

% if first_use not used then this must be reactivated
% solve(c-X,C,S):-
%     X=available(Y),
%     !,
%     matching_statement(state(available(Y))),
%     !,
%     solve(C,S).

solve(c-X,C,S):-
    X=first_use([Y-,_]),
    !,
    matching_statement(state(piece_type(Y))),
    solve(C,S).

solve(c-X,C,S):-
    matching_statement(state(X)),
    !,
    solve(C,S).

solve(c-X,C,S):-
    get_operator(X,G),
    % with f rule found pre-conditions must be found and added to stack
    get_preconditions(G,Y),
    % ensure that condition is not already on stack- ie. stop looping
    check_loop(Y),
    push(c-X,S,Ns),
    push(a-G,Ns,Ns1),
    push_list(Y,Ns1,Ns2),
    solve(C,Ns2).

solve(a-X,C,S):-
    get_preconditions(X,Y),
    check_state(Y),
    !,
    get_del_list(X,DEL),
    delete_list(DEL),
    get_add_list(X,ADD),
    add_list(ADD),
    add_operator(X),
    solve(C,S).

solve(a-X,C,S):-
    get_preconditions(X,Y),
    check_loop(Y),

```

```

push(a-X,S,Ns),
push_list(Y,Ns,Ns1),
solve(C,Ns1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      handle/1
%
%      DESCRIPTION:
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

handle(clear_area([P,D],S,Or):-
    matching_statement(state(max_dimension(MX,MY))),
    calc_coverage(P,S,MX,MY,Or,Osc),
    handle_covered(check_uncovered,Osc).
handle(uncovered([P,D],S,Or):-
    matching_statement(state(max_dimension(MX,MY))),
    calc_coverage(P,S,MX,MY,Or,Osc),
    handle_covered(check_uncovered,Osc).
handle(clear_path(P,Or,Sq,Dir):-
    clear_path(P,Or,Sq,Dir).
handle(first_use(P)).
handle(non_isolating(P,Or,Sq,D)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      write_plan/0
%
%      DESCRIPTION:    Writes plan to terminal by finding all of the
%                      solution/1 predicates that have been asserted.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

write_plan:-
    write('The start state is:'),nl,
    matching_statement(start_state_of_grid(X)),
    write(X),nl,
    matching_statement(start_state_of_pieces(X1)),
    write(X1),nl,
    write('The goal is: '),
    matching_statement(goal_state(Y)),
    write(Y),nl,
    write('The plan is:'),nl,
    matching_statement(solution(Z)),
    write(Z),nl,
    fail.
write_plan.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      add_list/1
%
%      DESCRIPTION:    adds a list of states to the current state by
%                      calling cstate/2.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_list([]).
add_list([H|T]):-
    H=covered([P,D],S,Or),
    !,
    matching_statement(state(max_dimension(MX,MY))),
    calc_coverage(P,S,MX,MY,Or,Osc),
    handle_covered(add_covered,Osc),
    add_list(T).

```

```

add_list([H|T]):-
    H=uncovered([P,D],S,Or),
    !,
    matching_statement(state(max_dimension(MX,MY))),
    calc_coverage(P,S,MX,MY,Or,Osc),
    handle_covered(add_uncovered,Osc),
    add_list(T).
add_list([H|T]):-
    cstate(add,H),
    add_list(T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      delete_list/1
%
%      DESCRIPTION:    deletes a list of states from the current state by
%                      calling cstate/2.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

delete_list([]).
delete_list([H|T]):-
    H=covered([P,D],S,Or),
    !,
    matching_statement(state(max_dimension(MX,MY))),
    calc_coverage(P,S,MX,MY,Or,Osc),
    handle_covered(delete_covered,Osc),
    delete_list(T).
delete_list([H|T]):-
    H=uncovered([P,D],S,Or),
    !,
    matching_statement(state(max_dimension(MX,MY))),
    calc_coverage(P,S,MX,MY,Or,Osc),
    handle_covered(delete_uncovered,Osc),
    delete_list(T).
delete_list([H|T]):-
    cstate(delete,H),
    delete_list(T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      handle_covered/2
%
%      DESCRIPTION:    kluge for 'other squares covered'
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

handle_covered(_,[ ]):-!.
handle_covered(add_covered,[H|T]):-
    cstate(add,covered(H)),
    handle_covered(add_covered,T).
handle_covered(add_uncovered,[H|T]):-
    cstate(add,uncovered(H)),
    handle_covered(add_uncovered,T).
handle_covered(delete_covered,[H|T]):-
    cstate(delete,covered(H)),
    handle_covered(delete_covered,T).
handle_covered(delete_uncovered,[H|T]):-
    cstate(delete,uncovered(H)),
    handle_covered(delete_uncovered,T).
handle_covered(check_covered,[H|T]):-
    check_state(covered(H)),
    handle_covered(check_covered,T).
handle_covered(check_uncovered,[H|T]):-
    check_state(uncovered(H)),
    handle_covered(check_uncovered,T).

```

[illegible]

```
push(Item,Old_stack,[Item|Old_stack]).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      pop/3
%
%   DESCRIPTION:    pops the appropriate condition off the stack.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
pop(Item,[Item|Old_stack],Old_stack).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      push_list/3
%
%   DESCRIPTION:    calls push/3 to push a list of conditions
%                   onto the stack.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
push_list([],S,S).
push_list([X|T],Os,Ns):-
    push(c-X,Os,Ns),
    push_list(T,Ns,Ns).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      check_loop/2
%
%   DESCRIPTION:    fails if sub-goal is already on stack.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
check_loop([],S).
check_loop([H|T],S):-
    not member(H,S),
    check_loop(T,S).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      add_operator/1
%
%   DESCRIPTION:    adds operator to solution via dynamic
%                   solution/1.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
add_operator(X):-
    add_statement_b(solution(X)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      check_state/1
%
%   DESCRIPTION:    succeeds if true in current state.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
check_state([]).
check_state([H|T]):-
    handle(H),
    check_state(T).
check_state([H|T]):-
```

```

        matching_statement(state(H)),
        check_state(T).
check_state(X):-
    matching_statement(state(X)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      calc_coverage/6
%
%      DESCRIPTION:    determines other squares covered by a
%      particular shape
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

calc_coverage(a_, [X,Y],MX,MY,0,[[X,YP1],[XP1,Y],[XP1,YP1]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX,
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY.

calc_coverage(a_, [X,Y],MX,MY,90,[[XP1,Y],[X,YM1],[XP1,YM1]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX,
    YM1 is Y - 1, YM1 >= 1, YM1 <= MY.

calc_coverage(a_, [X,Y],MX,MY,180,[[XM1,Y],[XM1,YM1],[X,YM1]]):-
    XM1 is X - 1, XM1 >= 1, XM1 <= MX,
    YM1 is Y - 1, YM1 >= 1, YM1 <= MY.

calc_coverage(a_, [X,Y],MX,MY,270,[[X,YP1],[XM1,Y],[XM1,YP1]]):-
    XM1 is X - 1, XM1 >= 1, XM1 <= MX,
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY.

calc_coverage(b_, [X,Y],MX,MY,0,[[X,YP1]]):-
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY.

calc_coverage(b_, [X,Y],MX,MY,180,[[X,YP1]]):-
    YP1 is Y - 1, YP1 >= 1, YP1 <= MY.

calc_coverage(b_, [X,Y],MX,MY,90,[[XP1,Y]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX.

calc_coverage(b_, [X,Y],MX,MY,270,[[XP1,Y]]):-
    XP1 is X - 1, XP1 >= 1, XP1 <= MX.

calc_coverage(c_, [X,Y],MX,MY,0,[[XP1,Y]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX.

calc_coverage(c_, [X,Y],MX,MY,180,[[XM1,Y]]):-
    XM1 is X - 1, XM1 >= 1, XM1 <= MX.

calc_coverage(c_, [X,Y],MX,MY,90,[[X,YM1]]):-
    YM1 is Y - 1, YM1 >= 1, YM1 <= MY.

calc_coverage(c_, [X,Y],MX,MY,270,[[X,YP1]]):-
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY.

calc_coverage(d_, [X,Y],MX,MY,0,{}).

calc_coverage(e_, [X,Y],MX,MY,0,[[X,YP1],[XP1,Y]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX,
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY.

calc_coverage(e_, [X,Y],MX,MY,90,[[XP1,Y],[X,YM1]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX,
    YM1 is Y - 1, YM1 >= 1, YM1 <= MY.

```

```

calc_coverage(e-_, [X,Y],MX,MY,180,[[X,M1],[X,YM1]]):-
    XM1 is X - 1, XM1 >= 1, XM1 <= MX,
    YM1 is Y - 1, YM1 >= 1, YM1 <= MY.

calc_coverage(e-_, [X,Y],MX,MY,270,[[X,YP1],[X,M1]]):-
    XM1 is X - 1, XM1 >= 1, XM1 <= MX,
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY.

calc_coverage(f-_, [X,Y],MX,MY,0,[[X,YP1],[X,YP2],[XP1,Y],[XP1,YP2]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX,
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY,
    YP2 is Y + 2, YP2 >= 1, YP2 <= MY.

calc_coverage(f-_, [X,Y],MX,MY,90,[[X,YM1],[XP1,Y],[XP2,Y],[XP2,YM1]]):-
    XP1 is X + 1, XP1 >= 1, XP1 <= MX,
    YM1 is Y - 1, YM1 >= 1, YM1 <= MY,
    XP2 is X + 2, XP2 >= 1, XP2 <= MX.

calc_coverage(f-_, [X,Y],MX,MY,180,[[X,M1],[X,M2],[X,YM1],[X,YM2]]):-
    XM1 is X - 1, XM1 >= 1, XM1 <= MX,
    YM1 is Y - 1, YM1 >= 1, YM1 <= MY,
    YM2 is Y - 2, YM2 >= 1, YM2 <= MY.

calc_coverage(f-_, [X,Y],MX,MY,270,[[X,M2,YP1],[X,M2,Y],[X,M1,Y],[X,YP1]]):-
    YP1 is Y + 1, YP1 >= 1, YP1 <= MY,
    XM1 is X - 1, XM1 >= 1, XM1 <= MX,
    XM2 is X - 2, XM2 >= 1, XM2 <= MX.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% PREDICATE:      non_isolating/4
%
% DESCRIPTION:    determines whether placing a piece in a
%                 particular position blocks the only remaining clear path to
%                 any uncovered square.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

non_isolating(P,Or,Sq,D):-
    matching_statement(state(max_dimension(Mx,My))),
    calc_direction(D,D1,D2,D3),
    (clear_path(P,Or,Sq,D1);
    check_path(P,Or,Sq,D1,Mx,My)),!,
    (clear_path(P,Or,Sq,D2);
    check_path(P,Or,Sq,D2,Mx,My)),!,
    (clear_path(P,Or,Sq,D3);
    check_path(P,Or,Sq,D3,Mx,My)),!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% PREDICATE:      calc_direction/4
%
% DESCRIPTION:    when given one of north, south, west, east it
%                 returns the other 3 directions.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

calc_direction(south,west,east,north).
calc_direction(north,west,east,south).
calc_direction(west,south,north,east).
calc_direction(east,south,north,west).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      check_path/6
%
%   DESCRIPTION:    fails if there is an uncovered square in
%                   this direction that has no clear path.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

check_path(P,Or,Sq,D,Mx,My):-
    find_susp_sqs(P,Or,Sq,D,Mx,My,Susp_list),
    calc_direction(D,D1,D2,D3),
    check_susp_sqs([d_,[1,1]],0,Susp_list,D,D1,D2,D3),
    !.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      find_susp_sqs/7
%
%   DESCRIPTION:    returns a list of uncovered squares in the
%                   particular direction excluding those covered by the proposed
%                   piece.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

find_susp_sqs([P,[X,Y]],Or,[A,B],D,Mx,My,Susp_list):-
    calc_limits(Or,D,Mx,My,X,Y,A,B,X1,X2,Y1,Y2),
    calc_coverage(P,[A,B],Mx,My,Or,Osc),
    x_list(X1,X2,[],X_list),
    x_list(Y1,Y2,[],Y_list),
    path_list(X_list,Y_list,[],Path_list),
    susp_list(Path_list,[[A,B]|Osc],[],Susp_list),
    !.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      susp_list/4
%
%   DESCRIPTION:    given a list of squares returns a list of
%                   those squares that are uncovered exculding those that will be
%                   covered by the proposed piece.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

susp_list([],Sqs_cov,X,X):- !.
susp_list([H|T],Sqs_cov,Old_list,Susp_list):-
    not member(H,Sqs_cov),
    check_state(uncovered(H)),
    susp_list(T,Sqs_cov,[H|Old_list],Susp_list).
susp_list([H|T],Sqs_cov,Old_list,Susp_list):-
    susp_list(T,Sqs_cov,Old_list,Susp_list).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   PREDICATE:      check_susp_sqs/7
%
%   DESCRIPTION:    succeeds if path is clear in either direction
%                   for all squares in list.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

check_susp_sqs(P,Or,[],D,D1,D2,D3):- !.
check_susp_sqs(P,Or,[H|T],D,D1,D2,D3):-
    (clear_path(P,Or,H,D1);
    clear_path(P,Or,H,D2);
    clear_path(P,Or,H,D3)),

```



```

!,
    check_susp_sqs(P,Or,T,D,D1,D2,D3).
check_susp_sqs(P,Or,[H|T],D,D1,D2,D3):-
    !,
    not clear_path(P,Or,H,D3),
    write('isolated square - no solution possible'),nl,
    halt.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      clear_path/4
%
%      DESCRIPTION:    returns direction of a clear path for the
%                      piece to be slid.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

clear_path([P,[X,Y]],Or,[A,B],Direction):-
    matching_statement(state(max_dimension(Mx,My))),
    !,
    calc_limits(Or,Direction,Mx,My,X,Y,A,B,X1,X2,Y1,Y2),
    x_list(X1,X2,[],X_list),
    x_list(Y1,Y2,[],Y_list),
    path_list(X_list,Y_list,[],Path_list),
    adj_ath_list(Path_list,P,A,B,Or,Direction,Mod_path_list),
    handle_covered(check_uncovered,Mod_path_list).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      calc_limits/12
%
%      DESCRIPTION:    returns the x and y limits of the path needed
%                      by a particular type of piece.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

calc_limits(0,south,Mx,My,X,Y,A,B,A,X2,Y1,1):-
    Y1 is B+(Y-1), X2 is A+(X-1).

```

```

calc_limits(0,north,Mx,My,X,Y,A,B,A,X2,B,My):-
    X2 is A+(X-1).

```

```

calc_limits(0,west,Mx,My,X,Y,A,B,X1,1,B,Y2):-
    X1 is A+(X-1), Y2 is B+(Y-1).

```

```

calc_limits(0,east,Mx,My,X,Y,A,B,A,Mx,B,Y2):-
    Y2 is B+(Y-1).

```

```

calc_limits(90,south,Mx,My,X,Y,A,B,A,X2,B,1):-
    X2 is A+(Y-1).

```

```

calc_limits(90,north,Mx,My,X,Y,A,B,A,X2,Y1,My):-
    Y1 is B-(X-1), X2 is A+(Y-1).

```

```

calc_limits(90,west,Mx,My,X,Y,A,B,X1,1,B,Y2):-
    X1 is A+(Y-1), Y2 is B-(X-1).

```

```

calc_limits(90,east,Mx,My,X,Y,A,B,A,Mx,B,Y2):-
    Y2 is B-(X-1).

```

```

calc_limits(180,south,Mx,My,X,Y,A,B,A,X2,B,1):-
    X2 is A-(X-1).

```

```

calc_limits(180,north,Mx,My,X,Y,A,B,A,X2,Y1,My):-

```

```

Y1 is B-(Y-1), X2 is A-(X-1).

calc_limits(180,west,Mx,My,X,Y,A,B,A,1,B,Y2):-
    Y2 is B-(Y-1).

calc_limits(180,east,Mx,My,X,Y,A,B,X1,Mx,B,Y2):-
    X1 is A-(X-1), Y2 is B-(Y-1).

calc_limits(270,south,Mx,My,X,Y,A,B,A,X2,Y1,1):-
    Y1 is B+(X-1), X2 is A-(Y-1).

calc_limits(270,north,Mx,My,X,Y,A,B,A,X2,B,My):-
    X2 is A-(Y-1).

calc_limits(270,west,Mx,My,X,Y,A,B,A,1,B,Y2):-
    Y2 is B+(X-1).

calc_limits(270,east,Mx,My,X,Y,A,B,X1,Mx,B,Y2):-
    X1 is A-(Y-1), Y2 is B+(X-1).

#####
%
%      PREDICATE:      path_list/4
%
%      DESCRIPTION:    given the two x and two y limits returns a
%                      list of the squares that make up the path.
%
%#####

path_list([],_,X,X):-!.
path_list([H:T],Y,Old,New):-
    x_coord(H,Y,Old,New1),
    path_list(T,Y,New1,New).

#####
%
%      PREDICATE:      x_coord/4
%
%      DESCRIPTION:    given an x coordinate and a list of y
%                      coordinates returns a list of all x,y pairs.
%
%#####

x_coord(_,[],X,X):-!.
x_coord(X,[H:T],Old,New):-
    x_coord(X,T,[[X,H]|Old],New).

#####
%
%      PREDICATE:      x_list/4
%
%      DESCRIPTION:    given two numerical whole number limits
%                      returns a list of the whole number between the two.
%
%#####

x_list(X,X,Old_list,[X|Old_list]):-!.
x_list(X1,X2,Old_list,New_list):-
    X1 < X2,
    !,
    X3 is X1 + 1,
    x_list(X3,X2,[X1|Old_list],New_list).

```

```
x_list(X1,X2,Old_list,New_list):-
    X3 is X2 + 1,
    _list(X3,X1,[X2|Old_list],New_list).
```

```

%*****
%
%      PREDICATE:      adj_path_list/7
%
%      DESCRIPTION:    adjusts path list to accomodate irregular
%      piece shapes.
%
%*****

```

```
adj_path_list(Path_list,P,A,B,Or,D,Mod_path_list):-
    adj_sqs(P,Or,D,A,B,Adj_sqs),
    remove_sqs(Path_list,Adj_sqs,Mod_path_list),
    !.
```

```

*****
%
%      PREDICATE:      adj_sqs/6
%
%      DESCRIPTION:    returns list of squares to be adjusted/removed
%
*****

```

```
adj_sqs(e_,0,south,A,B,[[X,Y]]):-
    X is A+1, Y is B+1.
```

```
adj_sqs(e_,0,west,A,B,[(X,Y)):-
    X is A+1, Y is B+1.
```

```
adj_sqs(e_, 90, north, A, B, [[X, Y]]):-
    X is A+1, Y is B-1.
```

```
adj_sqs(e_, 90, west, A, B, [[X, Y]]):-
    X is A+1, Y is B-1.
```

```
adj_qs(e_,180,north,A,B,[[X,Y]]):-
    X is A-1, Y is B-1.
```

```
adj_sqs(e_,180,east,A,B,[[X,Y]]):-
    X is A-1, Y is B-1.
```

```
adj_sqs(e_,270,south,A,B,[[X,Y]]):-
    X is A-1, Y is B+1.
```

```
adj_sqs(e_,270,east,A,B,[[X,Y]]):-
    X is A-1, Y is B+1.
```

```
adj_sqs(f_-,0,west,A,B,[[X,Y]]):-
    X is A+1. Y is B+1.
```

```
adj_sqs(f_,90,north,A,B,[[X,Y]]):-
    X is A+1, Y is B-1.
```

```
adj_sqsf(_,180,east,A,B. [[X,Y]]):-
    X is A-1, Y is B-1.
```

```
adj_sqs(f--,270,south,A,B,[(X,Y)]):-
    X is A-1, Y is B+1.
```

adj_sqs(, , , , {}).

.....

```

%      PREDICATE:      remove_sqs/3
%
%      DESCRIPTION:    returns list path_list with adj_sqs removed
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
remove_sqs(Mod_path_list, [], Mod_path_list) :- !.
remove_sqs(Path_list, [H|T], Mod_path_list) :-
    delete(Path_list, H, New_path_list), !,
    remove_sqs(New_path_list, T, Mod_path_list).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      gcl/0
%
%      DESCRIPTION:    performs necessary garbage collection of
%                      dynamic predicates
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gcl:-
    del_all_statements(solution/1),
    del_all_statements(state/1),
    del_all_statements(add_list/2),
    del_all_statements(del_list/2),
    del_all_statements(preconditions/2),
    del_all_statements(start_state_of_grid/1),
    del_all_statements(start_state_of_pieces/1),
    del_all_statements(goal_state/1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      PREDICATE:      Miscellaneous Predicates
%
%      DESCRIPTION:    Self Explanatory
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

member(X, [X|_]).
member(X, [_|_]) :- member(X, _).

delete([X|Xs], X, Ys) :- delete(Xs, X, Ys).
delete([X|Xs], Z, [X|Ys]) :- X \= Z, delete(Xs, Z, Ys).
delete([], X, []).

reverse(Xs, Ys) :- reverse(Xs, [], Ys).
reverse([X|Xs], Acc, Ys) :- reverse(Xs, [X|Acc], Ys).
reverse([], Ys, Ys).

endmod /*t8astrips*/.

```

Appendix B

Example Input File

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      TITLE:      p810c20.pro
%
%      AUTHOR:      C. Roth
%
%      PROJECT:      Strips Planner
%
%      DATE:      March 1989
%
%      DESCRIPTION:  This is the strips style operator description
%                    file used by the strips style planner 'The Puzzler' .
%
%      Puzzle Size:  10 x 10
%
%      Goal Order:   [1,1]-[1,10] - [10,1]-[10,10]
%
%      No. of Pieces: 78
%
%      Piece Order:  Largest to smallest
%
%      Sqs. Covered: [3,3],[3,8],[3,9],[4,3],[4,4],[4,5],[4,7],
%                    [5,5],[5,7],[6,3],[6,4],[6,5],[6,6],[6,7],
%                    [6,8],[6,9],[7,3],[7,8],[8,3],[8,8]
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

preconditions(select(Piece),[available(Piece),first_use(Piece),handempty])).
preconditions(place(Piece,Square,Orientation,Direction),[
    non_isolating(Piece,Orientation,Square,Direction),
    clear_path(Piece,Orientation,Square,Direction),
    uncovered(Piece,Square,Orientation),uncovered(Square),holding(Piece)]).
preconditions(rotate_90(Piece,Square),[clear_area(Piece,Square,90),
    orientation(Piece,0),holding(Piece)]).
preconditions(rotate_180(Piece,Square),[clear_area(Piece,Square,180),
    orientation(Piece,0),holding(Piece)]).
preconditions(rotate_270(Piece,Square),[clear_area(Piece,Square,270),
    orientation(Piece,0),holding(Piece)]).

```

```

add_list(select(Piece),[holding(Piece)]).
add_list(place(Piece,Square,Orientation,Direction),[covered(Square),
    covered(Piece,Square,Orientation),at(Piece,Square),handempty])).
add_list(rotate_90(Piece,Square),[orientation(Piece,90),
    uncovered(Piece,Square,90)]).
add_list(rotate_180(Piece,Square),[orientation(Piece,180),
    uncovered(Piece,Square,180)]).
add_list(rotate_270(Piece,Square),[orientation(Piece,270),
    uncovered(Piece,Square,270)]).

```

```

del_list(select(Piece),[handempty,available(Piece)]).
del_list(place(Piece,Square,Orientation,Direction),[uncovered(Square),
    uncovered(Piece,Square,Orientation),holding(Piece)]).
del_list(rotate_90(Piece,Square),[orientation(Piece,0)]).
del_list(rotate_180(Piece,Square),[orientation(Piece,0)]).
del_list(rotate_270(Piece,Square),[orientation(Piece,0)]).

```

```

start_state_of_grid([max_dimension(10,10),handempty,
covered([3,3]),covered([3,8]),covered([3,9]),covered([4,3]),covered([4,4]),
covered([4,5]),covered([4,7]),covered([5,5]),covered([5,7]),covered([6,3]),
covered([6,4]),covered([6,5]),covered([6,6]),covered([6,7]),covered([6,8]),
covered([6,9]),covered([7,3]),covered([7,8]),covered([8,3]),covered([8,8])]).

start_state_of_pieces([
piece_type(a),piece_type(e),piece_type(f),piece_type(b),piece_type(c),
piece_type(d),
available([a-1,[2,2]]),available([a-2,[2,2]]),available([a-3,[2,2]]),
available([a-4,[2,2]]),available([a-5,[2,2]]),available([a-6,[2,2]]),
available([a-7,[2,2]]),available([a-8,[2,2]]),available([a-9,[2,2]]),
available([b-1,[1,2]]),available([b-2,[1,2]]),available([b-3,[1,2]]),
available([b-4,[1,2]]),available([b-5,[1,2]]),available([b-6,[1,2]]),
available([b-7,[1,2]]),available([b-8,[1,2]]),available([b-9,[1,2]]),
available([b-10,[1,2]]),available([b-11,[1,2]]),available([b-12,[1,2]]),
available([b-13,[1,2]]),available([b-14,[1,2]]),
available([c-1,[2,1]]),available([c-2,[2,1]]),available([c-3,[2,1]]),
available([c-4,[2,1]]),available([c-5,[2,1]]),available([c-6,[2,1]]),
available([c-7,[2,1]]),available([c-8,[2,1]]),available([c-9,[2,1]]),
available([c-10,[2,1]]),available([c-11,[2,1]]),available([c-12,[2,1]]),
available([c-13,[2,1]]),
available([d-1,[1,1]]),available([d-2,[1,1]]),available([d-3,[1,1]]),
available([d-4,[1,1]]),available([d-5,[1,1]]),available([d-6,[1,1]]),
available([d-7,[1,1]]),available([d-8,[1,1]]),available([d-9,[1,1]]),
available([d-10,[1,1]]),available([d-11,[1,1]]),available([d-12,[1,1]]),
available([d-13,[1,1]]),available([d-14,[1,1]]),available([d-15,[1,1]]),
available([d-16,[1,1]]),available([d-17,[1,1]]),available([d-18,[1,1]]),
available([d-19,[1,1]]),available([d-20,[1,1]]),available([d-21,[1,1]]),
available([d-22,[1,1]]),available([d-23,[1,1]]),available([d-24,[1,1]]),
available([d-25,[1,1]]),available([d-26,[1,1]]),available([d-27,[1,1]]),
available([d-28,[1,1]]),available([d-29,[1,1]]),available([d-30,[1,1]]),
available([d-31,[1,1]]),available([d-32,[1,1]]),available([d-33,[1,1]]),
available([d-34,[1,1]]),available([d-35,[1,1]]),available([d-36,[1,1]]),
available([d-37,[1,1]]),available([d-38,[1,1]]),available([d-39,[1,1]]),
available([d-40,[1,1]]),
available([e-1,[2,2]]),available([f-1,[2,3]]),
orientation([a-1,[2,2]],0),orientation([a-2,[2,2]],0),orientation([a-3,[2,2]],0),
orientation([a-4,[2,2]],0),orientation([a-5,[2,2]],0),orientation([a-6,[2,2]],0),
orientation([a-7,[2,2]],0),orientation([a-8,[2,2]],0),orientation([a-9,[2,2]],0),
orientation([b-1,[1,2]],0),orientation([b-2,[1,2]],0),orientation([b-3,[1,2]],0),
orientation([b-4,[1,2]],0),orientation([b-5,[1,2]],0),orientation([b-6,[1,2]],0),
orientation([b-7,[1,2]],0),orientation([b-8,[1,2]],0),orientation([b-9,[1,2]],0),
orientation([b-10,[1,2]],0),orientation([b-11,[1,2]],0),orientation([b-12,[1,2]],0),
orientation([b-13,[1,2]],0),orientation([b-14,[1,2]],0),
orientation([c-1,[2,1]],0),orientation([c-2,[2,1]],0),orientation([c-3,[2,1]],0),
orientation([c-4,[2,1]],0),orientation([c-5,[2,1]],0),orientation([c-6,[2,1]],0),
orientation([c-7,[2,1]],0),orientation([c-8,[2,1]],0),orientation([c-9,[2,1]],0),
orientation([c-10,[2,1]],0),orientation([c-11,[2,1]],0),
orientation([c-12,[2,1]],0),orientation([c-13,[2,1]],0),
orientation([d-1,[1,1]],0),orientation([d-2,[1,1]],0),orientation([d-3,[1,1]],0),
orientation([d-4,[1,1]],0),orientation([d-5,[1,1]],0),orientation([d-6,[1,1]],0),
orientation([d-7,[1,1]],0),orientation([d-8,[1,1]],0),orientation([d-9,[1,1]],0),
orientation([d-10,[1,1]],0),orientation([d-11,[1,1]],0),
orientation([d-12,[1,1]],0),orientation([d-13,[1,1]],0),
orientation([d-14,[1,1]],0),orientation([d-15,[1,1]],0),
orientation([d-16,[1,1]],0),orientation([d-17,[1,1]],0),
orientation([d-18,[1,1]],0),orientation([d-19,[1,1]],0),
orientation([d-20,[1,1]],0),orientation([d-21,[1,1]],0),
orientation([d-22,[1,1]],0),orientation([d-23,[1,1]],0),
orientation([d-24,[1,1]],0),orientation([d-25,[1,1]],0),
orientation([d-26,[1,1]],0),orientation([d-27,[1,1]],0),
orientation([d-28,[1,1]],0),orientation([d-29,[1,1]],0),

```

```
orientation([d-30,[1,1]],0),orientation([d-31,[1,1]],0),  
orientation([d-32,[1,1]],0),orientation([d-33,[1,1]],0),  
orientation([d-34,[1,1]],0),orientation([d-35,[1,1]],0),  
orientation([d-36,[1,1]],0),orientation([d-37,[1,1]],0),  
orientation([d-38,[1,1]],0),orientation([d-39,[1,1]],0),  
orientation([d-40,[1,1]],0),  
orientation([e-1,[2,2]],0),orientation([f-1,[2,3]],0)).  
  
goal_state(complete_the_puzzle).
```

Appendix C

Simulated Global Data Structure Entries of Problem Example

```

gds(1, 1, control, initialize_gds, c, t, _, _ )
gds(1, 2, state_space, max_dimension(1,1), c, t, _, _ )
gds(1, 3, state_space, handempty, c, t, _, _ )
gds(1, 4, state_space, piece_type(d), c, t, _, _ )
gds(1, 5, state_space, available([d-1,[1,1]]), c, t, _, _ )
gds(1, 6, state_space, orientation([d-1,[1,1]],0), c, t, _, _ )
gds(1, 7, goal_stack, complete_puzzle, c, f, _, 0 )
gds(1, 8, planning_tree, [0, _, g, complete_puzzle, u ], c, f, _, 0 )
gds(2, 1, control, enumerate_operator, c, t, _, _ )
gds(2, 2, operator_agenda, [001, technique_depth_first, [1,7], 0, 100], c, f, _, 0)
gds(2, 3, operator_agenda, [002, strategy_analyze_board, [1,7], 0, 110], c, f, _, 0)
gds(2, 4, operator_agenda, [003, refine_goal, [1,7], 0, 120], c, f, _, 0)
gds(2, 5, control, op_agenda([001,002,003]), c, t, _, _ )
gds(3, 1, control, select_operator, c, t, _, _ )
gds(3, 2, control, choice(003), c, f, _, 0)
gds(4, 1, control, execute_operator(003), c, f, _, 0)
gds(4, 2, planning, [refine_goal, complete_puzzle, 0, 003], c, f, _, 0)
gds(4, 3, goal_stack, covered([1,1]), c, u, _, 1)
gds(4, 4, planning_tree, [1, 0, g, covered([1,1]), u], c, f, _, 1)
gds(4, 5, control, op_agenda([001,002]), c, t, _, _ )
gds(5, 1, control, enumerate_operator, c, t, _, _ )
gds(5, 2, operator_agenda, [004, simple_establishment, [4,3], 1, 50], c, u, _, 1)
gds(5, 3, control, op_agenda([001,002,004]), c, t, _, _ )
gds(6, 1, control, select_operator, c, t, _, _ )
gds(6, 2, control, choice(002), c, f, _, 0)
gds(7, 1, control, execute_operator(002), c, f, _, 0)
gds(7, 2, control, strategy=analyze_board, c, t, _, _ )
gds(7, 3, control, op_agenda([001,004]), c, t, _, _ )
gds(8, 1, control, enumerate_operator, c, t, _, _ )
gds(8, 2, control, op_agenda([001,004]), c, t, _, _ )

```



```

gds(9, 1, control, select_operator, c, t, _, _)
gds(9, 2, control, strategy, c, t, _, _)
gds(9, 3, planning, [analyze_board, _, _, 002], c, t, _, _)
gds(9, 4, goal_stack, covered([1,1]), c, u, +10, 1)
gds(9, 5, planning_tree, [1, 0, g, covered([1,1]), u], c, f, +10, 1)
gds(9, 6, operator_agenda, [004, simple_establishment, [9,3], 1, 50], c, u, +10, 1)
gds(9, 7, control, choice(001), c, f, _, 0)
gds(10, 1, control, execute_operator(001), c, f, _, 0)
gds(10, 2, control, technique=depth_first, c, t, _, _)
gds(10, 3, control, op_agenda([004]), c, t, _, _)
gds(11, 1, control, enumerate_operator, c, t, _, _)
gds(11, 2, control, op_agenda([004]), c, t, _, _)
gds(12, 1, control, select_operator, c, t, _, _)
gds(12, 2, control, choice(004), c, u, +10, 1)
gds(13, 1, control, execute_operator(004), c, u, +10, 1)
gds(13, 2, planning, [simple_achievement, covered([1,1]), 1, 004], c, u, +10, 1)
gds(13, 3, goal_stack, covered([1,1]), c, f, +10, 1)
gds(13, 4, control, op_agenda([1]), c, t, _, _)
gds(14, 1, control, enumerate_operators, c, t, _, _)
gds(14, 2, operator_agenda, [005, step_addition, [13,3], 1, 50], c, f, +10, 1)
gds(14, 3, control, op_agenda([005]), c, t, _, _)
gds(15, 1, control, select_operator, c, t, _, _)
gds(15, 2, control, choice(005), c, f, +10, 1)
gds(16, 1, control, execute_operator(005), c, f, +10, 1)
gds(16, 2, planning, [step_addition, place(P, [1,1], O), 1-1, 005], c, _, _, 1-1)
gds(16, 3, goal_stack, holding(P), c, u, _, 1-1-1)
gds(16, 4, goal_stack, uncovered([1,1]), c, u, _, 1-1-2)
gds(16, 5, goal_stack, uncovered(P, [1,1], O), c, u, _, 1-1-3)
gds(16, 6, goal_stack, clear_path(P, O, [1,1], D), c, u, _, 1-1-4)
gds(16, 7, goal_stack, non_isolating(P, O, [1,1], D), c, u, _, 1-1-5)
gds(16, 8, planning_tree, [1-1, 1, o, place(P, [1,1], O), p], c, _, _, 1-1)
gds(16, 9, planning_tree, [1-1-1, 1-1, g, holding(P), u], c, u, _, 1-1-1)
gds(16, 10, planning_tree, [1-1-2, 1-1, g, uncovered([1,1]), u], c, u, _, 1-1-2)
gds(16, 11, planning_tree, [1-1-3, 1-1, g, uncovered(P, [1,1], O), u], c, u, _, 1-1-3)
gds(16, 12, planning_tree, [1-1-4, 1-1, g, clear_path(P, O, [1,1], D), u], c, u, _, 1-1-4)

```

```

gds(16, 13, planning_tree, [1-1-5, 1-1, g, non_isolating(P, O, [1,1], D), u], c, u, _, 1-1-5)
gds(16, 14, plan_steps, [place(P, [1,1], O), 1-1, p, 005], c, _, _, 1-1)
gds(16, 15, control, op_agenda([]), c, t, _, _)
gds(17, 1, control, enumerate_operator, c, t, _, _)
gds(17, 2, operator_agenda, [006, simple_establishment, [16,3], 1-1-1, 50], c, u, _,
    1-1-1)
gds(17, 3, operator_agenda, [007, simple_establishment, [16,4], 1-1-2, 50], c, u, _,
    1-1-2)
gds(17, 4, operator_agenda, [008, complex_establishment, [16,5], 1-1-3, 50], c, u, _,
    1-1-3)
gds(17, 5, operator_agenda, [009, complex_establishment, [16,6], 1-1-4, 50], c, u, _,
    1-1-4)
gds(17, 6, operator_agenda, [010, complex_establishment, [16,7], 1-1-5, 50], c, u, _,
    1-1-5)
gds(17, 7, control, op_agenda([006,007,008,009,010]), c, t, _, _)
gds(18, 1, control, select_operator, c, t, _, _)
gds(18, 2, control, technique, c, t, _, _)
gds(18, 3, planning, [depth_first, _, _, _], c, t, _, _)
gds(18, 4, goal_stack, holding(P), c, u, +5, 1-1-1)
gds(18, 5, goal_stack, uncovered([1,1]), c, u, +4, 1-1-2)
gds(18, 6, goal_stack, uncovered(P, [1,1], O), c, u, +3, 1-1-3)
gds(18, 7, goal_stack, clear_path(P,O, [1,1], D), c, u, +2, 1-1-4)
gds(18, 8, goal_stack, non_isolating(P, O, [1,1], D), c, u, +1, 1-1-5)
gds(18, 9, operator_agenda, [006, simple_establishment, [18,4], 1-1-1, 50], c, u, +5,
    1-1-1)
gds(18, 10, operator_agenda, [007, simple_establishment, [18,5], 1-1-2, 50], c, u, +4,
    1-1-2)
gds(18, 11, operator_agenda, [008, complex_establishment, [18,6], 1-1-3, 50], c, u, +3,
    1-1-3)
gds(18, 12, operator_agenda, [009, complex_establishment, [18,7], 1-1-4, 50], c, u, +2,
    1-1-4)
gds(18, 13, operator_agenda, [010, complex_establishment, [18,8], 1-1-5, 50], c, u, +1,
    1-1-5)
gds(18, 14, control, choice(006), c, u, +5, 1-1-1)
gds(19, 1, control, execute_operator(006), c, u, +5, 1-1-1)
gds(19, 2, planning, [simple_establishment, holding(P), 1-1-1, 006], c, u, +5, 1-1-1)

```

```

gds(19, 3, goal_stack, holding(P), c, f, +5, 1-1-1)
gds(19, 4, control, op_agenda([007,008,009,010]), c, t, _, _)
gds(20, 1, control, enumerate_operator, c, t, _, _)
gds(20, 2, operator_agenda, [011, step_addition, [19,3], 1-1-1, 40], c, f, +5, 1-1-1)
gds(20, 3, control, op_agenda([007,008,009,010,011]), c, t, _, _)
gds(21, 1, control, select_operator, c, t, _, _)
gds(21, 2, control, technique, c, t, _, _)
gds(21, 3, planning, [depth_first, _, _, _], c, t, _, _)
gds(21, 4, goal_stack, holding(P), c, f, +5+15, 1-1-1)
gds(21, 5, operator_agenda, [011, step_addition, [21,4], 1-1-1, 40], c, f, +20, 1-1-1)
gds(21, 6, control, choice(011), c, f, +20, 1-1-1)
gds(22, 1, control, execute_operator(011), c, f, +20, 1-1-1)
gds(22, 2, planning, [step_addition, select(P), 1-1-1-1, 011], c, _, _, 1-1-1-1)
gds(22, 3, goal_stack, available(P), c, u, _, 1-1-1-1-1)
gds(22, 4, goal_stack, first_use(P), c, u, _, 1-1-1-1-2)
gds(22, 5, goal_stack, handempty, c, u, 1-1-1-1-3)
gds(22, 6, planning_tree, [1-1-1-1, 1-1-1, o, select(P), p], c, u, _, 1-1-1-1)
gds(22, 7, planning_tree, [1-1-1-1-1, 1-1-1-1, g, available(P), u], c, u, _, 1-1-1-1-1)
gds(22, 8, planning_tree, [1-1-1-1-2, 1-1-1-1, g, first_use(P), u], c, u, _, 1-1-1-1-2)
gds(22, 9, planning_tree, [1-1-1-1-3, 1-1-1-1, g, handempty, u], c, u, _, 1-1-1-1-3)
gds(22, 10, plan_steps, [select(P), 1-1-1-1, p, 011], c, _, _, 1-1-1-1)
gds(22, 11, control, op_agenda([007,008,009,010]), c, t, _, _)
gds(23, 1, control, enumerate_operator, c, t, _, _)
gds(23, 2, operator_agenda, [012, simple_establishment, [22,3], 1-1-1-1-1, 50], c, u, _,
    1-1-1-1-1)
gds(23, 3, operator_agenda, [013, complex_establishment, [22,4], 1-1-1-1-2, 50], c, u,
    _, 1-1-1-1-2)
gds(23, 4, operator_agenda, [014, simple_establishment, [22,5], 1-1-1-1-3, 50], c, u,
    _, 1-1-1-1-3)
gds(23, 5, control, op_agenda([007,008,009,010,012,013,014]), c, t, _, _)
gds(24, 1, control, select_operator, c, t, _, _)
gds(24, 2, control, technique, c, t, _, _)
gds(24, 3, planning, [depth_first, _, _, _], c, t, _, _)
gds(24, 4, goal_stack, available(P), c, u, +8, 1-1-1-1-1)
gds(24, 5, goal_stack, first_use(P), c, u, +7, 1-1-1-1-2)
gds(24, 6, goal_stack, handempty, c, u, +6, 1-1-1-1-3)

```

gds(24, 7, operator_agenda, [012, simple_establishment, [24,4], 1-1-1-1-1, 50], c, u, +8,
 1-1-1-1-1)
 gds(24, 8, operator_agenda, [013, complex_establishment, [24,5], 1-1-1-1-2, 50], c, u,
 +7, 1-1-1-1-2)
 gds(24, 9, operator_agenda, [014, simple_establishment, [24,6], 1-1-1-1-3, 50], c, u,
 +6, 1-1-1-1-3)
 gds(24, 10, control, choice(012), c, u, +8, 1-1-1-1-1)
 gds(25, 1, control, execute_operator(012), c, u, +8, 1-1-1-1-1)
 gds(25, 2, planning, [simple_establishment, available(P), 1-1-1-1-1, 012], c, u, +8,
 1-1-1-1-1)
 gds(25, 3, goal_stack, available(P), c, t, +8, 1-1-1-1-1)
 gds(25, 4, planning_tree, [1-1-1-1-1, 1-1-1-1, g, available(P), s], c, t, _, 1-1-1-1-1)
 gds(25, 5, control, op_agenda([007,008,009,010,013,014]), c, t, _, _)
 gds(26, 1, control, enumerate_operator, c, t, _, _)
 gds(26, 2, control, op_agenda([007,008,009,010,013,014]), c, t, _, _)
 gds(27, 1, control, select_operator, c, t, _, _)
 gds(27, 2, control, choice(013), c, u, +7, 1-1-1-1-2)
 gds(28, 1, control, execute_operator(013), c, u, +7, 1-1-1-1-2)
 gds(28, 2, planning, [complex_establishment, first_use(P), 1-1-1-1-2, 013], c, u, +7,
 1-1-1-1-2)
 gds(28, 3, goal_stack, first_use(P), c, t, +7, 1-1-1-1-2)
 gds(28, 4, planning_tree, [1-1-1-1-2, 1-1-1-1, g, first_use(p), s], c, t, _, 1-1-1-1-2)
 gds(28, 5, control, op_agenda([007,008,009,010,014]), c, t, _, _)
 gds(29, 1, control, enumerate_operator, c, t, _, _)
 gds(29, 2, control, op_agenda([007,008,009,010,014]), c, t, _, _)
 gds(30, 1, control, select_operator, c, t, _, _)
 gds(30, 2, control, choice(014), c, u, +6, 1-1-1-1-3)
 gds(31, 1, control, execute_operator(014), c, u, +6, 1-1-1-1-3)
 gds(31, 2, planning, [simple_establishment, handempty, 1-1-1-1-3, 014], c, u, +6,
 1-1-1-1-3)
 gds(31, 3, goal_stack, handempty, c, t, +6, 1-1-1-1-3)
 gds(31, 4, planning_tree, [1-1-1-1-3, 1-1-1-1, g, handempty, s], c, t, _, 1-1-1-1-3)
 gds(31, 5, control, op_agenda([007,008,009,010]), c, t, _, _)
 gds(32, 1, control, enumerate_operator, c, t, _, _)
 gds(32, 2, operator_agenda, [015, select(P), [[25,3],[28,3],[31,3],[22,10]], 1-1-1-1, 35],
 c, u, +20, 1-1-1-1)

```

gds(32, 3, control, op_agenda([007,008,009,010,015]), c, t, _, _)
gds(33, 1, control, select_operator, c, t, _, _)
gds(33, 2, control, choice(015), c, u, +20, 1-1-1-1)
gds(34, 1, control, execute_operator(015), c, u, +20, 1-1-1-1)
gds(34, 2, planning, [check_preconditions, select(P), 1-1-1-1, 015], c, u, _, 1-1-1-1)
gds(34, 3, state_space, handempty, c, f, _, _)
gds(34, 4, state_space, available([d-1,[1,1]]), c, f, _, _)
gds(34, 5, state_space, holding([d-1,[1,1]]), c, t, _, _)
gds(34, 6, planning_tree, [1-1-1-1, 1-1-1, o, select(P), e], c, t, _, 1-1-1-1)
gds(34, 7, plan_steps, [select(P), 1-1-1-1, e, 015], c, _, _, 1-1-1-1)
gds(34, 8, planning_tree, [1-1-1, 1-1, g, holding(P), s], c, t, _, 1-1-1)
gds(34, 9, goal_stack, holding(P), c, t, +1, 1-1-1)
gds(34, 10, control, op_agenda([007,008,009,010]), c, t, _, _)
gds(35, 1, control, enumerate_operator, c, t, _, _)
gds(35, 2, control, op_agenda([007,008,009,010]), c, t, _, _)
gds(36, 1, control, select_operator, c, t, _, _)
gds(36, 2, control, choice(007), c, u, +4, 1-1-2)
gds(37, 1, control, execute_operator(007), c, u, +4, 1-1-2)
gds(37, 2, planning, [simple_establishment, uncovered([1,1]), 1-1-2, 007], c, u, +4,
    1-1-2)
gds(37, 3, goal_stack, uncovered([1,1]), c, t, +4, 1-1-2)
gds(37, 4, planning_tree, [1-1-2, 1-1, g, uncovered([1,1]), s], c, t, _, 1-1-2)
gds(37, 5, control, op_agenda([008,009,010]), c, t, _, _)
gds(38, 1, control, enumerate_operator, c, t, _, _)
gds(38, 2, control, op_agenda([008,009,010]), c, t, _, _)
gds(39, 1, control, select_operator, c, t, _, _)
gds(39, 2, control, choice(008), c, u, +3, 1-1-3)
gds(40, 1, control, execute_operator(008), c, u, +3, 1-1-3)
gds(40, 2, planning, [complex_establishment, uncovered(P, [1,1],O), 1-1-3, 008], c, u,
    +3, 1-1-3)
gds(40, 3, goal_stack, uncovered(P, [1,1], O), c, t, +3, 1-1-3)
gds(40, 4, planning_tree, [1-1-3, 1-1, g, uncovered(P, [1,1], O), s], c, t, _, 1-1-3)
gds(40, 5, control, op_agenda([009,010]), c, t, _, _)
gds(41, 1, control, enumerate_operator, c, t, _, _)
gds(41, 2, control, op_agenda([009,010]), c, t, _, _)
gds(42, 1, control, select_operator, c, t, _, _)

```

```

gds(42, 2, control, choice(009), c, u, +2, 1-1-4)
gds(43, 1, control, execute_operator(009), c, u, +2, 1-1-4)
gds(43, 2, planning, [complex_establishment, clear_path(P, O, [1,1], D), 1-1-4, 009], c,
    u, +2, 1-1-4)
gds(43, 3, goal_stack, clear_path(P, O, [1,1], D), c, t, +2, 1-1-4)
gds(43, 4, planning_tree, [1-1-4, 1-1, g, clear_path(P, O, [1,1], D), s], c, t, _, 1-1-4)
gds(43, 5, control, op_agenda([010]), c, t, _, _)
gds(44, 1, control, enumerate_operator, c, t, _, _)
gds(44, 2, control, op_agenda([010]), c, t, _, _)
gds(45, 1, control, select_operator, c, t, _, _)
gds(45, 2, control, choice(010), c, u, +1, 1-1-5)
gds(46, 1, control, execute_operator(010), c, u, +1, 1-1-5)
gds(46, 2, planning, [complex_establishment, non_isolating(P, O, [1,1], D), 1-1-5, 010],
    c, u, +1, 1-1-5)
gds(46, 3, goal_stack, non_isolating(P, O, [1,1], D), c, t, +1, 1-1-5)
gds(46, 4, planning_tree, [1-1-5, 1-1, g, non_isolating(P, O, [1,1], D), s], c, t, _, 1-1-5)
gds(46, 5, control, op_agenda([]), c, t, _, _)
gds(47, 1, control, enumerate_operator, c, t, _, _)
gds(47, 2, operator_agenda, [016, place(P, [1,1], O), [[34,9],[37,3],[40,3],[43,3],[46,3]],
    1-1, 35], c, u, _, 1-1)
gds(47, 3, control, op_agenda([016]), c, u, _, 1-1)
gds(48, 1, control, select_operator, c, t, _, _)
gds(48, 2, control, choice(016), c, u, _, 1-1)
gds(49, 1, control, execute_operator(016), c, u, _, 1-1)
gds(49, 2, planning, [check_preconditions, place(P, [1,1], O), 1-1, 016], c, u, _, 1-1)
gds(49, 3, state_space, uncovered([1,1]), c, f, _, _)
gds(49, 4, state_space, holding(P), c, f, _, _)
gds(49, 5, state_space, covered([1,1]), c, t, _, _)
gds(49, 6, state_space, at(P, [1,1]), c, t, _, _)
gds(49, 7, state_space, handempty, c, t, _, _)
gds(49, 8, planning_tree, [1-1, 1, o, place(P, [1,1], O), s], c, t, _, 1-1)
gds(49, 9, plan_steps, [place(P, [1,1], O), 1-1, e, 005], c, t, _, 1-1)
gds(49, 10, planning_tree, [1, 0, g, covered([1,1]), s], c, t, _, 1)
gds(49, 11, goal_stack, covered([1,1]), c, t, _, 1)
gds(49, 12, control, op_agenda([]), c, t, _, _)
gds(50, 1, control, enumerate_operators, c, t, _, _)

```

```
gds(50, 2, control, op_agenda([]), c, t, _, _)  
gds(50, 3, goal_stack, complet_puzzle, c, t, _, 0)  
gds(50, 4, control, output_plan, c, t, _, _)  
gds(50, 5, control, end, c, t, _, _)
```

END

13|09|90

FIN