An Analysis of Conjunctive-Goal Planning

David E. Joslin
John W. Roach

TR 86-34

AN ANALYSIS OF CONJUNCTIVE-GOAL PLANNING

David Eugene Joslin

Committee Chairman: John Roach
Computer Science

(ABSTRACT)

This thesis develops a formal theory of planning, using a simple paradigm of planning that has been previously explored in work such as GPS, HACKER, STRIPS and NOAH. This thesis analyzes the goal interactions that occur when the goal is stated as a conjunction of sub-goals. In this analysis we assume that the problem has a finite state space, and that operators are reversible.

Graph theory can be used to characterize these sub-goal interactions. The entire state space is treated as a graph, and each sub-goal or conjunction of sub-goals defines a subgraph. Each subgraph is composed of one or more connected components. Solving each sub-goal by choosing a connected component that contains a final goal state is a necessary and sufficient condition for solving any planning problem.

In the worst case, analyzing goal interactions is shown to be no more effective than enumerating the state space and searching. This complexity proves that no complete algorithm can solve all planning problems in linear time.

The technique of goal ordering is analyzed, along with several extensions to that technique. While a generalization of goal ordering is possible, in the worst case generating the goal order requires as much computation as solving the problem by a brute-force search.

A technique called capability analysis, derived from the connected component results, uses first-order logic to find the constraints that must apply as sub-goals are achieved. A partial implementation uses counterfactual logic to identify the components of a world state that prevent the remaining sub-goals from being achieved.

Contents

1 Introduction **1**

1.1 An outline of the thesis 2

2 Previous work **3**

2.1 GPS 3

2.2 STRIPS 4

2.3 LAWALY and ABSTRIPS 5

2.4 JANITOR and HACKER 6

2.5 NOAH and NONLIN 8

2.6 GAGS 9

2.7 MACRO-OPS 9

2.8 TWEAK 10

2.9 Summary 10

3 Theoretical development **12**

3.1 State spaces as graphs 12

3.2 Sub-goals as subgraphs 13

3.3 Intuitive analysis 15

3.4 Connected components 16

3.5 Summary 17

4 The complexity of planning **18**

4.1 A lower bound on complexity 19

4.2 An upper bound on complexity 19

4.3 A note on representations 21
4.4 Complexity from goal interactions 22
4.5 Summary 25

5 Analysis of goal ordering **26**
5.1 Goal ordering 26
5.2 Enhancements to goal ordering 27
5.3 Limitations to goal ordering 29
5.4 Summary 30

6 An algorithm using logic **31**
6.1 Representing connected components 32
6.2 Capabilities and consistency 33
6.3 Using the constraints 34
6.4 Constraints and counterfactuals 35
6.5 Summary 38

7 Summary and conclusions **40**
7.1 How this thesis fits into the world of AI 40
7.2 What this thesis accomplished 41
7.3 Directions for further research 42

List of Figures

2.1 STRIPS operator example 5
2.2 The CAB problem 7

3.1 The two rooms problem 14

4.1 STRIPS-like description of maze problem 20
4.2 Card arrangement for tic-tac-toe isomorphism 22
4.3 Maze problem with goal interactions 24

6.1 Counterfactual program for CAB problem 36
6.2 Results for CAB problem with counterfactuals 37

Chapter 1

Introduction

> Every time you have a task before you, examine it carefully, take exact measure of what is expected of you. Then make your plan and, in order to execute it properly, create for yourself a method, never improvise.
>
> —MARSHAL FERDINAND FOCH

> There is nothing better in life than to have a goal and be working toward it.
>
> —GOETHE

The most basic type of planning is the ordering of actions to achieve a stated goal. The subject of planning also includes a broad spectrum of problems such as time reasoning, resource management, spatial reasoning and path planning.

This thesis addresses a much more limited type of planning. We start with a known *initial world state*, then apply *operators* that transform one state into another. A successful plan is a sequence of operators that transforms the initial world state to some *goal state*.

Even with this limited view of planning, the problem of *goal interactions* is not a simple one. Goal interactions may occur when the final goal is stated as a conjunction of simple sub-goals. A plan that achieves one particular sub-goal may interfere with a plan that achieves another sub-goal.

A number of techniques for handling these interactions have been attempted, most with only partial success. One of the earliest planners, STRIPS, could only handle problems that could be solved in a few steps and was inefficient even for those. The idea of goal ordering, originating with GPS and later refined by LAWALY and ABSTRIPS, appeared at first to be a powerful technique. Soon after, though, the discovery of a class of problems called *non-linear* problems revealed some serious flaws in that technique. Later planning systems such as NOAH were

Chapter 2

Previous work

When a thing is done, its done. Don't look back.
Look forward to your next objective.
—GEORGE C. MARSHALL

Plans fail for lack of counsel,
but with many advisers they succeed.
—PROVERBS 15:22 (NIV)

This chapter reviews some previous research in planning, and also introduces some basic concepts and terminology. Some of the more significant planning systems are described, along with some of their achievements and weaknesses. These systems and basic concepts are analyzed in later chapters after a theoretical framework is developed.

2.1 GPS

One of the earliest general-purpose planning systems was GPS [Ernst 69]. GPS used a technique called *means-ends analysis*. GPS is described as a problem solving system, rather than a planning system, but in this case the two ideas mean the same thing. GPS derives a sequence of steps to solve a problem, and that sequence of steps can be called a plan.

The GPS system established several ideas in the field of planning. A particularly important one is the idea of solving a compound problem by solving each of its component sub-goals independently.

Another idea fundamental to GPS was that of *differences*. The difference between the current world state and the goal state is ideally the distance from the

current state to the goal. In GPS and almost all of the systems we discuss here, the idea of differences is used to help choose the appropriate sub-goal to achieve next.

In GPS, differences were grouped into classes that were ranked according to difficulty. The ranking is intended to arrange the sub-goals so that it is always possible to solve lower-ranked goals without disturbing any previously solved higher-ranked goals. This idea of *goal ordering* was also used in the systems LAWALY and ABSTRIPS, and will be discussed in more detail later.

2.2 STRIPS

In 1971, Fikes and Nilsson published their work on STRIPS [Fikes 71]. This work established a technique for describing world states and operators that strongly influenced many other efforts. The world state is a database of well-formed formulas (wffs) in first-order logic, such as $atrobot(a)$ or $nextto(robot, box1)$. Things that are true in the world are represented in the database by positive atomic wffs. STRIPS uses a closed world assumption, meaning that if an atomic wff is not in the database, its negation is true. Thus if $nextto(robot, box2)$ is *not* in the database, then we know that the robot is not next to $box2$.

An operator has three components, the *preconditions*, the *delete list* and the *add list*. The precondition is an expression that must be satisfied in order for the operator to be applied. If the operator is applied, then all of the wffs in the delete list are deleted from the world state and all of the wffs in the add list are added to the world state.

For example, a *gothru* operator, used when the robot goes through a door into another room, is shown in Figure 2.1. The preconditions state that the robot must be next to the door before going through the door, that the door must connect the room the robot is currently in with the destination room, and that the door must be open. The delete list specifies that after the operator is applied the robot will no longer be in the current room and will have moved away from any object it was next to. The add list says that after the operator is applied the robot will be in the room on the other side of the door.

Another contribution of STRIPS was the formalization of the idea of differences. GPS used problem-dependent functions to determine differences. STRIPS formalized this idea by defining the differences to be the atomic wffs required by the goal state but not satisfied in the current state.

STRIPS could only solve simple problems and was computationally expensive. STRIPS did not have an effective method for choosing the best sub-goal to solve next, and consequently had to do a lot of searching. STRIPS used a theorem prover

```
gothru(d,m,n): Go from room m to room n via door d

Preconditions:  NEXTTO_ROBOT(d)
                CONNECTS(d,m,n)
                STATUS(d,open)
                INROOM(m)

Delete list:    NEXTTO_ROBOT(*)
                INROOM(m)

Add list:       INROOM(n)
```

Figure 2.1: STRIPS operator example

to conduct the search, but the theorem prover suffered from exponential blowup and as a result the system could not handle problems that required more than six steps or so to solve.

2.3 LAWALY and ABSTRIPS

LAWALY [Siklóssy 72] [Siklóssy 73] in 1972 used a technique called *goal ordering* to choose the sub-goal to solve next. Goal ordering, as mentioned with GPS, consists of ranking the goals in such a way that solving a lower-priority goal will not affect a previously solved higher-priority goal.

For example, suppose we have the two goals of turning a light switch on and stopping in room A. If the light switch is not in room A, then we have to turn on the light switch before going to room A. The state of the light switch does not affect our ability to go to a particular room, but if we go to a particular room and stay there we can only affect a light switch in that room. If we have a compound goal consisting of any light switch goal and any location goal, we can always solve them in that order.

Finding an effective goal order is not possible for all problems, though. The effectiveness of goal ordering is analyzed in more detail in Chapter 5.

LAWALY recognized that a sub-goal that has been previously achieved can sometimes be undone by the plan to achieve another sub-goal unless care is taken to prevent it. This idea has been termed *goal protection,* because a goal is protected from being undone after it has been achieved.

ABSTRIPS [Sacerdoti 74], a system similar to LAWALY, also used goal ordering. ABSTRIPS was based on STRIPS. ABSTRIPS did not use goal protection;

it assumed that previously solved sub-goals cannot be affected by plans for later sub-goals, and did not detect that a sub-goal had been undone. This assumption is not valid for all problems so the system sometimes gave incorrect solutions.

2.4 JANITOR and HACKER

A *non-linear* or *non-serializable* problem is one that cannot be solved by goal ordering. The first non-linear problem was described in a paper on the JANITOR robot planning system in 1972 [Roach 72]. This non-linear problem is described and analyzed in Chapter 3 (Figure 3.1).

The JANITOR system used goal ordering and so could not handle this non-linear problem. Up to this point goal ordering was thought to be an extremely powerful technique. This problem began to show some of the limitations of the technique.

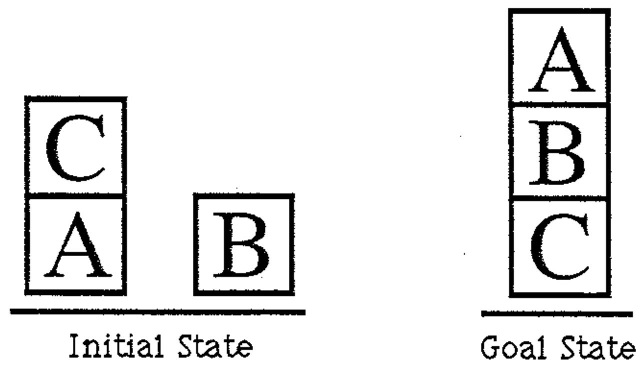
Sussman's HACKER [Sussman 75], originally published in 1973, was an early attempt to solve some non-linear problems. HACKER, like STRIPS, was a linear planner — a planner that assumed each sub-goal can be solved in turn, without regard to the sub-goals not yet solved. HACKER attempted to handle non-linear problems by trying to solve them linearly, then patching the plan after it was developed to take care of non-linear situations. This approach was partially successful.

Sussman described a non-linear problem that could not be solved by HACKER. This problem is illustrated in Figure 2.2(a). We will refer to this problem as the CAB problem, since that is what the blocks spell in the initial configuration. This problem is sometimes called the *Sussman anomaly*.

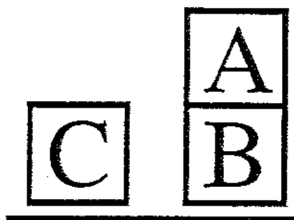
The problem uses three blocks, A, B and C. A block can be picked up only if no other block is on top of it. Initially block C is on block A, and blocks A and B are on the table. The goal is to have block A on block B, and block B on block C.

If we attempt to put block A on block B first, we reach the state shown in Figure 2.2(b). Block C is removed from block A so that block A can be picked up; block A is then picked up and placed on block B. We cannot put block B on block C now without undoing the goal that we have already achieved, block A on block B.

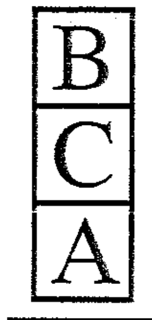
A similar situation arises when we try to put block B on block C first. This can be done in one step, but if we want to then put block A on block B we have to remove block B from block C first. This is an example of a non-linear problem because if we solve either sub-goal first in the most direct manner, we must undo that sub-goal in order to achieve the remaining sub-goal.



(a) Problem description



(b) Achieving $on(A, B)$ in the most direct way



(c) Achieving $on(B, C)$ in the most direct way

Figure 2.2: The CAB problem

2.5 NOAH and NONLIN

A linear planner assumes that the sub-goals can be solved in some order that will never require a previously solved sub-goal to be undone. We have seen some problems are non-linear, and cannot be solved under this assumption. This realization led to the development of non-linear planners that did not make the assumption of linearity.

Sacerdoti's NOAH system [Sacerdoti 75][Sacerdoti 77] is a non-linear planner. It is also called a *hierarchical* planning system. The hierarchy begins with a single node representing the compound goal. This node is expanded into sub-nodes corresponding to each sub-goal, so that the top level goal is achieved when all of its sub-nodes are achieved. Similarly, the second-level goals can be broken down further.

In effect, NOAH generates a plan for each sub-goal, then finds the ordering that has to be imposed between operators. This ordering is imposed when an operator in one sub-goal's plan interferes with an operator for another sub-goal's plan. Once this ordering is established additional operators may need to be inserted. An advantage of this approach is that no assumptions are made about the order of achieving goals until goal interactions force an ordering. Delaying the ordering was one technique that gave NOAH more flexibility than earlier systems.

NOAH was structured with a number of *critics* that refined the plan as it was developed. Some of the critics have been analyzed in [Chapman 85]. These critics were able to handle many of the non-linear problems that other systems had failed to solve, but were unable to solve some other simple problems.

A key concept in allowing the flexibility of joining individual sub-goal plans was that of *delayed binding*. For example, if a sub-goal is that the robot be in room Z, then we may be able to go through any of several doors to achieve that goal. We can specify that a "go through door" operator will be used to achieve this goal without specifying which door to go through. Another sub-goal may make one of the choices more desirable than the others, so the decision is delayed as long as possible.

This flexibility allowed NOAH to solve many problems without having to backtrack. NOAH was able to solve the CAB problem, and Sacerdoti made the following observation [Sacerdoti 77, p. 40]:

> The planning system has chosen the correct ordering for the subgoals, without backtracking or wasted computation. By avoiding a premature commitment to a linear plan, the system never had to undo a random choice made on the basis of insufficient information.

In fact, NOAH made no provision for backtracking. This prevented NOAH from solving some problems, but NOAH was useful in investigating the kinds of problems that can be solved in linear time by a hierarchical planner.

Sacerdoti recognized that backtracking would be necessary. He noted that search with backtracking "will have to be done if NOAH is to be truly competent at solving problems" [Sacerdoti 77, p. 91]. This was done by Tate in 1976, in a system called NONLIN [Tate 77]. NONLIN enhanced NOAH by adding a top-level backtracking control structure.

2.6 GAGS

In 1975, Giorgio Levi and Franco Sirovich published their work on generalized and/or graphs (GAGs) [Levi 75][Levi 76]. Their algorithm allows the solution of many problems, including problems that involve consumable resources.

An and/or graph is equivalent to a context-free or type 2 grammar [Hall 68]. A context-free grammar has production rules with left-hand sides consisting of only a non-terminal symbol. A generalized and/or graph is shown to be equivalent to the most general type of grammar, a type 0 grammar. Production rules in a type 0 grammar may be length-contracting, with the right-hand side having fewer symbols than the left-hand side. The productions may also be context-sensitive. Conceptually, a generalized and/or graph is able to represent the interdependencies between sub-goals that would not be shown by a simple and/or graph.

Levi and Sirovich give an algorithm that they prove to be able to solve any problem that can be expressed in GAGs. The system is apparently not implemented, and details for its implementation are not given. The distinguishing factor in this work is its formalism.

2.7 MACRO-OPS

In the decade or so between NON-LIN (1977) and MACRO-OPS (1985), planning research paid little attention to the extension of conjunctive-goal planning as such. Some planning systems during this period addressed the need for planning with resource management or time management. More recently there has been some renewed interest in the simpler domain that we are studying here.

Korf [Korf 85] describes a system called MACRO-OPS that uses macro operators to solve problems. A macro operator is formed by concatenating several primitive operators. Korf based his work on some previous planners that used

macro operators, but was able to extend and improve the technique. In particular, Korf was able to give a formal explanation for the applicability and performance of his system.

In MACRO-OPS, the macro operators are used to generalize the idea of goal protection. Rather than requiring that a goal be protected once it has been achieved, macro operators are created that might undo a goal temporarily, but that restore the goal before the macro operation is completed. Thus the goal is protected at the macro operator level, but not at the level of the primitive operators.

The MACRO-OPS system is able to solve complicated problems such as the Rubik's cube. The Rubik's cube state space has approximately 4×10^{19} states, and the average length of a solution is about 95 moves. The pre-processing time required to learn the macro operators for Rubik's cube is less than 15 cpu minutes on a VAX 11/785, using about 200K words of memory.

The macro operators transform a non-linear problem into a linear one, solvable with goal ordering. Once the macro operators have been learned, the pre-processing does not have to be repeated.

2.8 TWEAK

Chapman's system, TWEAK[Chapman 85], is similar to NOAH or NON-LIN in that it generates a partial ordering on operators, and allows variables to be bound as late as possible. As with NON-LIN, the top level of TWEAK does the searching.

Chapman describes his planning algorithm in a rigorous manner that allows him to prove that it is correct and complete. In this sense, TWEAK is similar to MACRO-OPS. Both improved upon some partially-successful planning techniques, then described the generalized technique in a rigorous way. Chapman is able to use his theoretical framework to analyze a number of previous algorithms, including HACKER, NOAH and NONLIN.

2.9 Summary

As we review the previous work in this paradigm of planning, we see progress in the types of problems that can be solved. Early planners suffered from an incorrect assumption of linearity. Sussman tried to solve this problem in HACKER by patching linear plans when the assumption of linearity failed. With NOAH, Sacerdoti got away from the idea of linear planning with his hierarchical planning technique. NONLIN was an improvement on this, adding a necessary search

component. Chapman's TWEAK improved on these systems even further, and Chapman was able to prove that his algorithm was complete and correct.

Few systems have had much of a theoretical basis. GAGS was based on formal languages, but implementation details have never been reported. Chapman's system is perhaps the first implemented system to be proven complete. This thesis is even more theoretical because it only attempts to analyze the problem of planning, rather than to create a new planning algorithm. Some of the successes and failures of these systems will be explained in the following chapters.

Chapter 3

Theoretical development

Here arises a puzzle that has disturbed scientists of all periods. How is it that mathematics, a product of human thought that is independent of experience, fits so excellently the objects of physical reality?
—ALBERT EINSTEIN

Our experience hitherto justifies us in believing that nature is the realization of the simplest conceivable mathematical ideas.
—ALBERT EINSTEIN

In this chapter we use graph theory to develop a formal representation of planning problems. Later chapters apply this representation to analyze previous work and to develop new solution techniques. In order to use graph theory for this analysis we have to assume that the problem has discrete states, and has a finite number of states. We will make a few other simplifying assumptions as we go along. The assumptions that we will be making are consistent with the problems that have been studied in the research efforts described in the previous chapter.

3.1 State spaces as graphs

If our problem domain has a finite number of discrete states, we can view the state space as a graph by considering each legal world state to be a node. If any operator changes one state to another, we have a directed arc from the first state to the second. If we allow that at most one operator can cause a given state transition, we have a 1–graph. A 1-graph is a graph that has at most one arc from any given initial node to any given destination node.

The idea of treating a problem space as a graph is not new. Both Berge [Berge 73] and Carre [Carre 79], among others, make use of it. Treating the entire

state space as a graph is useful only for small worlds; in many worlds the number of states is much too large to make this practical. Nevertheless, this representation is useful for analysis.

As an example we can consider the simple problem shown in Figure 3.1(a), first described in [Roach 72]. The initial state has a robot in room A, not next to anything, and the door between rooms A and B is closed. The goal state has the door closed and the robot next to a sink that is in room B. We will refer to this problem as the *two rooms* problem.

The door can only be opened or closed if the robot is next to the door. The robot can go into the next room only if it is next to the door and the door is open. The robot must be in the same room as something before it can go next to it. The door is considered to be in both rooms.

The door has two states, open and closed. The robot can be in four places. In room A, the robot is either next to the door or not next to anything. In room B, the robot is either next to the door or next to the sink. In Figure 3.1(a) these locations are numbered one through four. The initial location is labeled number one, and the final location is number four.

The entire state space, shown in Figure 3.1(b), has eight states. Each state is identified by the number for the robot's location, and by the letter *O* or the letter *C* for the door's state of open or closed. The initial state is in the lower left, labeled *1C*. The goal state is in the lower right, labeled *4C*. Note that the arcs are not directed arcs since in this particular problem the transitions are all reversible.

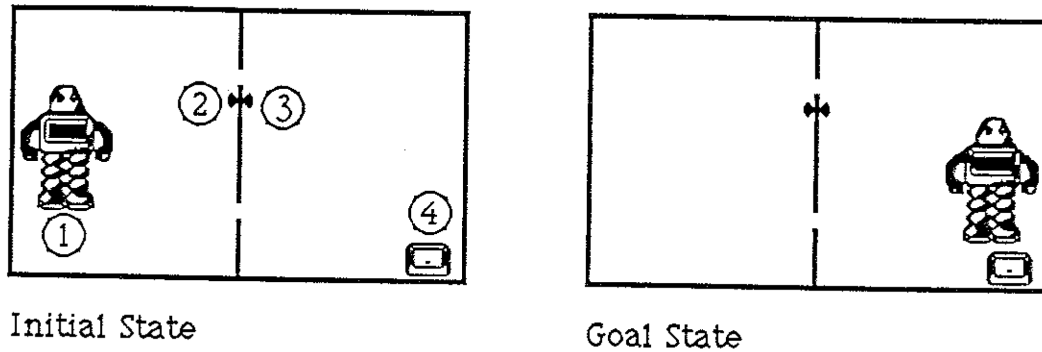
3.2 Sub-goals as subgraphs

Having defined the state space as a graph, we see that each sub-goal defines a subgraph. A subgraph can be defined by specifying a subset of the nodes of a graph; a sub-goal specifies the subset of world states where the sub-goal is satisfied. A conjunction of sub-goals defines a subgraph where all of the sub-goals in the conjunction are satisfied.

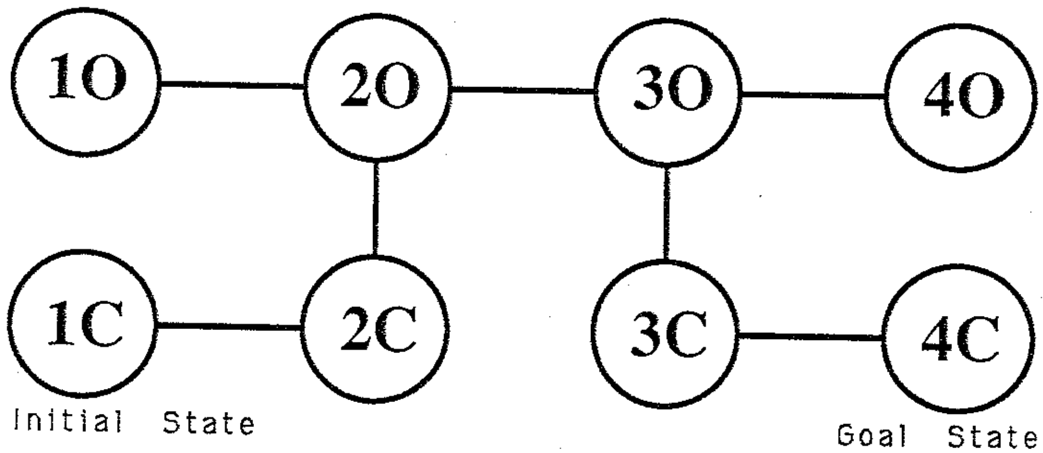
In our two rooms example, the subgraph that has the door closed is shown in Figure 3.1(c). We see that as long as the door is closed the robot is constrained to be either in room A (locations one and two) or room B (locations three and four).

Likewise, the goal of having the robot next to the sink defines a subgraph. This subgraph consists of two isolated nodes. In Figure 3.1(b), these nodes are labeled *4O* and *4C*.

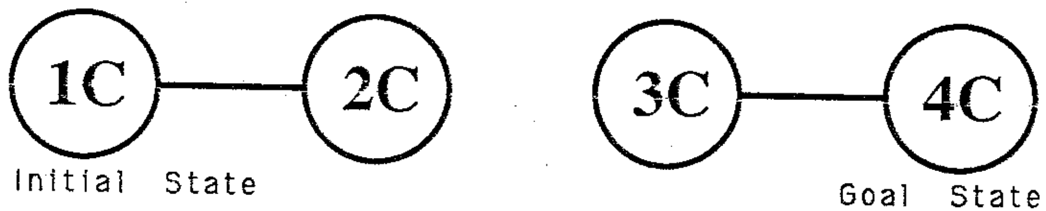
These subgraphs allow us to give a more formal definition to the idea of goal protection introduced in Chapter 2. If we achieve a goal such as closing the door



(a) Problem description



(b) State space



(c) Subgraph with door closed

Figure 3.1: The two rooms problem

and then protect that goal, we confine ourselves to the subgraph that has the goal satisfied. If a problem is to be solved using goal protection, then as each sub-goal is achieved the final goal state must be reachable while remaining within the subgraph defined by the conjunction of all sub-goals achieved so far.

The final goal state can also be defined in terms of these subgraphs. The final goal state is any state in the intersection of the subgraphs for all of the sub-goals. In the two rooms problem, the final goal subgraph is the single state satisfying both sub-goals.

3.3 Intuitive analysis

Before continuing with rigorous analysis based on graphs, it is helpful to gain some intuitive understanding by looking at the state space of the two rooms example. The two rooms problem turns out to be a difficult one for some systems. As we will see, goal ordering is not effective for this problem, so LAWALY and ABSTRIPS cannot solve this problem.

A planning system such as ABSTRIPS might try to solve the two rooms problem as follows. Since the door-closed goal is already solved in the initial state, that goal is not perceived as a difference to be resolved. To solve the remaining goal of being next to the sink, the plan would be to go next to the door (state *2C*), then open the door (state *2O*), then go through the door (state *3O*), then go next to the sink (state *4O*). Unfortunately, this plan undoes the goal of having the door closed.

We can see why goal ordering does not work here by looking at the subgraphs for each of the sub-goals. The door-closed subgraph (Figure 3.1(c)) consists of two separate parts. The initial state is in one of those parts and the final state is in the other, so when we try to protect the door-closed goal that is initially true we cannot reach the part of the subgraph that has the goal state. Similarly, if we solve the next-to-sink goal in the most direct fashion and then protect that goal, we find ourselves in a subgraph that consists of only one state, but not a goal state.

Since neither goal order allows us to solve each goal in the most direct fashion possible, systems based on goal ordering will not solve this problem. The correct solution follows essentially the plan described above, but inserts the step of closing the door after going through it but before going next to the sink. In other words, the door closed goal is actually solved in the *middle* of the process of going next to the sink.

In general, then, we see that if a goal is to be achieved and then protected, it must be achieved in such a way that we are in a part of the goal's subgraph that also contains a final goal state. In the following section we formalize this notion of solving each sub-goal within a particular part of the subgraph.

3.4 Connected components

In graph theory, the "parts" of the subgraph discussed in the previous section are called *connected components.* The subgraph in Figure 3.1(c) has two connected components. The subgraph for the goal of being next to the sink also has two connected components, each an isolated node.

A connected component for a directed graph can be defined in terms of chains. [Berge 73] defines a chain as follows:

> A chain is a sequence $\mu = (\mu_1, \mu_2, \ldots, \mu_q)$ of arcs of [a graph] G such that each arc in the sequence has one endpoint in common with its predecessor in the sequence and its other endpoint in common with its successor in the sequence.

Note that a chain is more general than a path. In a chain the traversal of an arc in the sequence does not have to follow the direction of the arc.

If a graph has a chain connecting every pair of distinct nodes in the graph, then the graph is called a *connected graph.* A *connected component* is a maximal connected subgraph. In other words, two nodes in a graph are part of the same connected component if and only if there is a chain between them. A connected graph has exactly one connected component.

The subgraph for a sub-goal can have multiple connected components, even if every state in the full graph is reachable from every other state. This is the case with our two rooms example.

For simplicity, we will assume that if a world state can be changed to another world state in one step (one operator), then the reverse operation may also be performed in one step. Many planning problems meet this assumption, including the blocks world and most robot worlds. Some examples of worlds where the assumption does not hold are worlds with consumable resources and robot worlds with one-way doors.

Under this assumption we can treat the graph as an undirected graph. More importantly, any state within a connected component is reachable from any other state within that connected component.

We can achieve a sub-goal by entering any of the connected components in its subgraph, but protecting the sub-goal will confine us to that connected component. If the connected component does not contain a final goal state then protecting that sub-goal prevents us from solving the problem. If no connected component in the subgraph contains a final goal state, then the problem cannot be solved.

Achieving a sub-goal in a connected component that contains a final goal state is clearly a sufficient condition for solving the remaining goals without undoing the first goal, since under our assumptions every state within a connected component is reachable from every other state. It is also a necessary condition, since we cannot solve the problem unless the connected component we have selected does contain a final goal state. Note that if we had not assumed reversible operations, selecting a connected component that contains a final goal state would not be sufficient, since the goal state is not necessarily reachable from all states within the connected component.

Achieving each sub-goal within a connected component that contains a final goal state is therefore a necessary and sufficient condition for solving a problem if we are going to protect each sub-goal as it is solved. This result does not tell us how to solve a sub-goal within a connected component that has a final goal state. It does tell us that any planning system that does not solve sub-goals in this manner cannot be a complete system. Furthermore, if we did have a planning algorithm that could solve each sub-goal within a connected component that had a final goal state, we would know that the system was complete.

If sub-goals are not protected, as with MACRO-OPS, then a sub-goal may be achieved and undone any number of times. Even here, though, each sub-goal will be achieved for the *last* time at some point. The world state must then be part of a connected component that contains a final goal state. A planning system that does not always solve sub-goals in this fashion cannot be a complete system.

3.5 Summary

We saw in the two rooms example that a sub-goal may have to be solved in a particular way so that it will not have to be undone in order to achieve the remaining sub-goals. In this particular problem the door must be closed while the robot is in the same room as the sink. This idea of achieving a sub-goal in a particular way to avoid conflicts with other sub-goals is characterized in this chapter in terms of connected components. We showed that finding connected components with final goal states is a necessary and sufficient condition for planning.

The next chapter will use graph theory to prove the complexity of planning. The fact that goal protection requires that a planner solve goals within certain connected components will be shown in Chapter 5 to be the source of failure for some previous systems. In Chapter 6 we will develop a technique for finding the added conditions that guarantee that a sub-goal has been solved according to the connected component requirement.

Chapter 4

The complexity of planning

No problem is so big and so complicated that it can't be run away from.
—LINUS

For every complex problem there is a solution that is simple, neat, and wrong.
—H. L. MENCKEN

He who every morning plans the transaction of the day and follows that plan, carries a thread that will guide him through the maze of the most busy life.
—VICTOR HUGO

This chapter demonstrates that the worst-case computational complexity of planning is proportional to the size of the state space. We then show that the presence or absence of sub-goal interactions does not change the inherent complexity of planning.

All of the complexity results in this chapter refer to computational complexity, not to time complexity. Parallel computation may reduce the actual time required to solve a problem, but this does not reduce the inherent computational complexity of a problem. We will refer to "linear-time" algorithms in the discussion, but we are assuming here that no concurrency of operations is being used.

In investigating the computational complexity of a problem we are not concerned with the fact that many problems in the domain may be substantially easier than the worst-case problems. This discussion is only concerned with the worst-case complexity. We have already noted that many problems were solved in linear time by NOAH and other algorithms. As we will see, these problems are much simpler than the worst-case problems that we describe in this chapter.

As mentioned previously, graph theory is only applicable when the state space is finite. In fact, Chapman showed in [Chapman 85] that planning problems are not decidable if the state space can be infinite. All of our analysis assumes a finite state space.

4.1 A lower bound on complexity

This chapter uses a maze as an example problem to demonstrate a lower-bound on worst-case planning complexity. The problem consists of a robot finding a path between two specified rooms in a maze. The robot has a map (a list of room connections) so it does not have to move through the maze to discover paths. We assume that the robot has not seen the maze previously, and so does not start with any additional information. This problem can be stated in STRIPS-like notation as shown in Figure 4.1.

If a doorway connects room X and room Y, then the initial database has CONN(X,Y) and CONN(Y,X). We assume that all doors are open and that the robot can go through a door in either direction. The goal is INROOM(Z), for some room Z.

We can determine the complexity of solving this problem by observing that it can be mapped directly to a graph. Each room in the maze corresponds to a node and each CONN predicate corresponds to an arc. Since the CONN predicates are chosen so that we do not have any "one-way doors," we can treat the graph as an undirected graph.

A depth-first search will find a path to the goal node with worst-case complexity of $O(max(|E|,|V|))$ operations, where $|E|$ is the number of edges and $|V|$ is the number of vertices [Aho 74]. We can see that an algorithm that always finds a path through an arbitrary graph must be of at least $O(max(|E|,|V|))$ complexity by observing that in the worst case an algorithm may find the goal node only after visiting every other node in the graph. That last node might only be found after trying every arc in the graph. Therefore, the complexity of finding a path in a maze or a graph must be at least $O(max(|E|,|V|))$.

We see from this that, in general, the maze problem cannot be solved in linear time; some search with backtracking (or the equivalent) will be required. Note that our discussion is limited to worst-case problems. We have already seen that some problems can be solved in linear time by algorithms such as NOAH's hierarchical planning. Worst-case problems such as mazes require search with backtracking, and therefore cannot be solved by any linear-time algorithm.

4.2 An upper bound on complexity

The maze problem is a model for searching a state space, since by a change in representation any state space can be treated as a maze. Since a state space search is a combinatoric algorithm, we would expect that maze problems are of the same complexity.

```
Initial world

        INROOM(A)

        CONN(A,B)
        CONN(B,A)
        CONN(A,C)
        CONN(C,A)
        CONN(B,D)
        CONN(D,B)

        ... etc.

goto(x,y) operator

        Preconditions:
                CONN(x,y)
                INROOM(x)

        Delete list:
                INROOM(*)

        Add list:
                INROOM(y)
```

Figure 4.1: STRIPS-like description of maze problem

The fact that any state space can be treated as a graph allows us to transform any problem into a maze problem. Therefore the complexity of the maze problem is an upper bound on the complexity of any planning problem. Planning problems therefore have a worst case complexity of $O(max(|E|, |V|))$.

Although the previous chapter assumed that operators are always reversible, this worst-case complexity is valid for general graphs. Since the maze problem has reversible operators, the same complexity holds under that assumption.

4.3 A note on representations

The previous sections showed, by using the maze example, that some planning problems are of $O(max(|E|, |V|))$ complexity. Next it was shown that no planning problems are worse than $O(max(|E|, |V|))$ complexity, since the state space for any problem can be treated as a maze.

This argument is valid for establishing a worst-case complexity and proving that a complete planning algorithm must do some searching. But even though some problems such as mazes are inherently difficult, many real problems are significantly easier. A maze has no "structure" to take advantage of, while many problems have some structure that can be useful.

A mail delivery robot, for example, may have to find its way from room to room within a building. This problem has some similarity to the maze problem. For most buildings, however, the problem of finding a room can be reduced to the problem of finding the right floor followed by the problem of finding the right room on that floor. The structure of floors and rooms reduces the amount of searching that is required. If the problem were structured with two predicates, FLOOR and ROOM, we might say that the goal interactions between the FLOOR and ROOM goals were helpful in reaching a solution.

If a problem has some inherent structure, then that structure must be preserved under any isomorphic transformation of the problem's representation. The inherent computational complexity is not affected by a change in representation as long as the new representation is isomorphic to the old one. But computational complexity and ease of solution by a human are two separate things. It is possible that an isomorphic change of representation may make a problem significantly easier (or more difficult) for a human, without changing the computational complexity.

As an example consider a game that begins with nine playing cards, ace through nine. Each card is worth its face value in points, with the ace worth one point. The players take turns selecting cards from the table, and the winner is the first player whose hand contains three cards totaling fifteen points.

8	3	4
1	5	9
6	7	2

Figure 4.2: Card arrangement for tic-tac-toe isomorphism

The isomorphism between this game and tic-tac-toe consists of arranging the cards to form a magic square, with each row, column, and diagonal totaling fifteen. The arrangement is shown in Figure 4.2. Each card represents a square on a tic-tac-toe board, and a player taking a card corresponds to playing in the square for that card.

Although we have an isomorphism between the two problems, the problems are not equally easy for a human to solve. The tic-tac-toe problem has a spatial structure corresponding to the mathematical structure of the card game. A child can play tic-tac-toe because the spatial structure involved is easy to understand. The card game seems to require more mental calculation. Yet both games have the same computational complexity.

In arguing about computational complexity, isomorphic transformations are allowable. We do need to be careful when transforming problem representations that we do not unwittingly add or remove structure in the problem, since changing the inherent structure can change the computational complexity.

The next section uses a transformation of representation to show that goal interactions do not, in the worst case, reduce the complexity of planning. This is not denying that some problems, due to the nature of their goal interactions, are not as complex as the worst-case problems. Unfortunately, the results given here do not help characterize the types of goal interactions that are helpful.

4.4 Complexity from goal interactions

The maze example is difficult because the domain for the INROOM predicate is large and sparsely connected. Because we have only one predicate, we cannot have any goal interactions to analyze. Without any additional information about the structure of the maze we cannot do anything but search.

Much of the successful work in planning has been based on the analysis of goal interactions. Furthermore, typical problems do have goal interactions. This raises the question of the effect of goal interactions on the complexity of planning.

We will show that in the worst case goal interactions do not reduce the complexity of planning. We do this by converting the maze problem to an equivalent problem whose complexity is entirely due to goal interactions.

The first step is to assign each room an arbitrary N-digit binary number, where N is large enough to allow a unique number for each room. We then create a predicate for each binary digit, and convert our states and operators as follows.

The INROOM predicate is replaced by N predicates. For example, if we need three binary digits to encode all of the rooms, and room A is assigned the number 011, then our initial state of INROOM(A) becomes the three predicates INROOM_0(0), INROOM_1(1), and INROOM_2(1).

Each CONN predicate is replaced by a CONN predicate with $2N$ arguments, with each of the two original arguments converted to N arguments by taking the N digits of its binary number. For example, if room A is assigned the number 011 and room B is assigned the number 101, then CONN(A,B) is replaced by CONN(0,1,1,1,0,1).

The *goto* operator is transformed accordingly. Our previous STRIPS-like example might be transformed as shown in Figure 4.3.

Each INROOM predicate now only has two possible values, zero and one, so none of the complexity of the problem in its present form can be due to the large domain of a predicate. We instead have interactions between the INROOM predicates, and our goal is now a conjunction of N sub-goals. If it is always true that goal interactions simplify the process of planning, then we should be able to solve the problem in better than $O(max(|E|,|V|))$ time. But since an arbitrary maze is equivalent to an arbitrary graph, no technique can improve on the worst-case complexity of $O(max(|E|,|V|))$.

We saw that complexity in planning problems could be due to a predicate with a large domain. We then showed that even if each predicate has a domain of only two values, the interactions between the operators can give us the same degree of complexity. Furthermore, we showed that problems of one type can be converted to the other type. From this we see that we cannot differentiate between the two types of interactions. In the worst case, analyzing goal interactions is no more effective than enumerating the state space and searching.

```
Initial world

        INROOM_0(0)
        INROOM_1(1)
        INROOM_2(1)

        CONN(0,1,1, 1,0,1)
        CONN(1,0,1, 0,1,1)
        CONN(0,1,1, 0,0,1)
        CONN(0,0,1, 0,1,1)
        CONN(1,0,1, 1,0,0)
        CONN(1,0,0, 1,0,1)

        ... etc.


goto(x,y,z) operator

        Preconditions:
                CONN(a,b,c,x,y,z)
                INROOM_0(a)
                INROOM_1(b)
                INROOM_2(c)

        Delete list:
                INROOM_0(*)
                INROOM_1(*)
                INROOM_2(*)

        Add list:
                INROOM_0(x)
                INROOM_1(y)
                INROOM_2(z)
```

Figure 4.3: Maze problem with goal interactions

4.5 Summary

This worst-case analysis helps us understand the fundamental limits for a "complete" algorithm — one that solves all planning problems within this particular paradigm of planning. Some problems, such as traversing a maze, are inherently unstructured and therefore difficult. Unfortunately, this analysis does not tell us what characteristics of a representation make it a good representation. This is a good area for future research.

Because the complexity of planning is proportional to the number of states in the state space, and state spaces can be arbitrarily large, a complete algorithm is not actually very interesting. A depth-first search algorithm can solve any of these planning problems, and is of optimal complexity for worst-case problems. For many other problems, of course, a depth-first search is far from optimal. A more interesting direction for future research is to find a useful subset of planning problems with a manageable computational complexity.

Chapter 5

Analysis of goal ordering

Divide each problem into as many parts as are feasible, and as are requisite for its better solution.

—RENÉ DESCARTES, *Discours de la méthod*

Direct your thoughts in an orderly way; beginning with the simplest objects, those most apt to be known, and ascending little by little, in steps as it were, to the knowledge of the most complex.

—RENÉ DESCARTES, *Discours de la méthod*

This chapter analyzes the technique of goal ordering in terms of connected components. This analysis applies only to means-ends analysis; it does not apply to other techniques, such as hierarchical planning. We are not analyzing the computational complexity of these problems, just investigating the limits to the effectiveness of means-ends analysis for a given problem and representation.

5.1 Goal ordering

We noted previously that goal ordering is a powerful idea, although it does not help solve all problems. Ideally, goal ordering would allow the possible goal predicates to be given a fixed order, independent of the initial state or goals for a particular problem. In terms of graph theory, this is possible if, for the given order, every connected component of the subgraph for the conjunction of the first N sub-goals contains a final goal state.

A *non-linear* problem is by definition a problem that cannot be solved by simple goal ordering. Non-linear problems are those that, for any sub-goal order, have at least one connected component without a final goal state in the subgraph for the first N sub-goals, for some N.

These definitions are actually too strict. A connected component that does not contain a final goal state is a necessary condition for non-linearity, but whether or not a planning algorithm falls into the trap of solving a sub-goal in that dead-end connected component depends on the algorithm. One algorithm may choose the right connected component, but another algorithm may choose the wrong one. If either of two operators can solve the sub-goal, the difference may simply be the operator selected by each algorithm. We know from our complexity results that no algorithm can always choose the right connected component unless it searches, but such an algorithm is no longer goal ordering.

Each algorithm solves the sub-goal in what it considers to be the most direct way. Unfortunately we cannot give a more precise definition without defining the "most direct way" of achieving a sub-goal. For this analysis, we will assume that the most direct way is the one that requires the fewest number of operations, although no known planning algorithm claims to ensure this.

5.2 Enhancements to goal ordering

Goal ordering is not a general technique. We have already seen that the two rooms problem does not yield to goal ordering algorithms. When goal ordering does work, however, it leads very directly to a solution. Planning can be done in roughly linear time when goal ordering is effective. This alone would be motivation to look for a generalization of the technique.

In looking at the two rooms problem we can easily find ways to allow goal ordering to work. We first observe that the only goal state has the robot in room B. If we add this requirement as a goal, then the goal ordering

1. *inroom(B)*
2. *door(closed)*
3. *nextto(sink)*

is effective. We can look at the effect of solving each of these sub-goals in the state space in Figure 3.1(b). Solving *inroom(B)* leaves us in a subgraph with only one connected component, consisting of the states labeled *3O, 3C, 4O* and *4C*. If we then close the door, we again have a subgraph with only one connected component, states *3C* and *4C*. The final goal state is then reachable in one step.

With another representation, however, the same problem doesn't yield to this approach. The same state space can be represented with only a door predicate and a location predicate, with the goals being *door(closed)* and *atrobot(4)*. This

follows the state diagram in Figure 3.1(b) more closely. Now we cannot add the *inroom(B)* sub-goal because it is not part of the representation.

We can extend the idea of goal ordering and solve this problem by allowing negated sub-goals. For example, goal ordering is effective with the following sequence of sub-goals

1. $\sim atrobot(1)$
2. $\sim atrobot(2)$
3. $door(closed)$
4. $atrobot(4)$

The goal of not being at location 1 is solved first and protected. As each of these sub-goals is achieved, the resulting sub-graph is again a single connected component.

Yet another alternative is to allow a disjunction of sub-goals to act as a single sub-goal. The following goal order also works:

1. $atrobot(3) \vee atrobot(4)$
2. $door(closed)$
3. $atrobot(4)$

This follows naturally since the disjunctive goal here is logically equivalent to the two negated goals in the previous example, for this system. That is, if the robot is either at location 3 or location 4 then the robot is not at location 1 and not at location 2.

These last two examples are easily understood by looking at Figure 3.1(c), the subgraph defined by $door(closed)$. As we have frequently noted, the $door(closed)$ sub-goal must be solved in the right connected component. This connected component is specified by any of the following:

- $inroom(B)$
- $atrobot(3) \vee atrobot(4)$
- $\sim atrobot(1) \wedge \sim atrobot(2)$

Each of these extensions consists of adding a sub-goal or sub-goals that cause a move into the correct connected component.

Many other problems allow goal ordering with the addition of simple sub-goals. The CAB problem given in Figure 2.2, for example, becomes linear with the addition of the goal of putting block C on the table. The goal order then becomes:

1. *on*(C, *table*)
2. *on*(B, C)
3. *on*(A, B)

As with the two rooms example and the *inroom*(B) sub-goal, the added sub-goal is always true in any final goal state.

If it were possible to convert any problem to a linear problem by adding goals, we would have to design a way to discover what the added goals should be. As we see in the following section, however, this result does not lead to a useful generalized goal ordering technique.

5.3 Limitations to goal ordering

In this section we consider a problem that does not yield to any of the extensions mentioned above. Moreover, we will argue that no added sub-goal, simple or compound, allows this problem to be solved by means-ends analysis.

The problem involves three variables, A, B and C, in a Pascal program. The objective is to swap the contents of two variables, A and B. Variable C is available to hold an intermediate value, but its initial and final values must both be zero. Our compound goal has three sub-goals; variable A must have the value originally found in B, variable B must have the value originally found in A, and variable C must have the value zero.

If we copy variable A to variable B immediately, we no longer have the original value from variable B available to copy into variable A. This means the sub-goal has been solved in the wrong connected component. The same is true if we try to solve the sub-goal for variable A immediately. The move required to get us into the correct connected component is to copy either A or B into variable C. But variable C will have the value zero in both the initial state and the final state. Therefore we cannot add a sub-goal that causes the necessary operation of using variable C as a temporary variable. The technique of adding sub-goals to make non-linear problems into linear problems is not a general technique.

5.4 Summary

Goal ordering is a powerful technique when it works. For this reason, it is reasonable to look for a generalization that would allow it to solve more problems. We showed that some non-linear problems can be made into linear problems simply by adding additional sub-goals. For some problems these added sub-goals had to be more complex than just positive atomic wffs. We also saw that some problems cannot made linear by the addition of sub-goals.

The analysis here applies only to means-ends analysis. It is not powerful enough to analyze planning algorithms such as NOAH. NOAH was able to solve more problems in linear time than means-ends analysis could, but was still limited in the types of problems that it could solve.

Chapter 6

An algorithm using logic

"Contrariwise," continued Tweedledee, "if it was so, it might be, and if it were so, it would be; but as it isn't, it ain't. That's logic!"
—LEWIS CARROLL

When you have eliminated the impossible, whatever remains, however improbable, must be the truth.
—SHERLOCK HOLMES

Inconsistent data — I must re-evaluate.
—*NOMAD space probe, Star Trek*

We saw in previous chapters that some problems require that certain conditions be met when a particular sub-goal is achieved. Those conditions were described in terms of connected components. Unfortunately, the graph theory results did not give us a technique for deriving the conditions that must be met for a given problem. In this chapter we develop such a technique using first-order logic.

Note that although we assume a particular order for achieving the sub-goals this technique is not limited only to goal-ordering algorithms. The conditions that must apply as each sub-goal is achieved are independent of the algorithm used to find a plan to achieve the sub-goal. If a particular algorithm allows a sub-goal to be achieved and then undone, it must still achieve each sub-goal for the last time at some point, and the conditions found here must apply at that point.

The technique presented here has several limitations. First, it applies only when the compound goal consists of two sub-goals. Another limitation is that, while it is capable of deriving the conditions that must apply when the first sub-goal is achieved, it does not tell us how to solve the sub-goal to satisfy those conditions. We also do not make any claims about the complexity of this approach. The value in investigating this technique comes from the fact that it opens up some potentially useful formal representations for planning problems.

6.1 Representing connected components

Graph theory can be used to characterize planning in terms of connected components. When we want to find the connected components, however, graph theory is of limited use. Many research planning problems are stated with wffs, so we might consider using first-order logic in our analysis.

One difficulty with using logic for planning is that the world state is modified by the application of operators. But first-order logic does not deal with time or changing worlds; it is monotonic. We can, however, use wffs in first-order logic to represent a single world state.

We will limit our analysis to problems whose goal consists of two sub-goals. In this analysis we want to represent a world state that has one sub-goal already achieved, and derive the conditions that must apply if the remaining sub-goal is to be achieved without undoing the first sub-goal. If these conditions are met, then we have found a connected component with a final goal state.

To do this we introduce the idea of a *capability*. A capability is the potential to use a given operator without violating a sub-goal that has been achieved and protected. This operator does not have to be immediately applicable; the capability of using that operator exists even if other operators must first be used to reach a state that allows the operator to be applied.

Suppose that the current world state has a sub-goal satisfied, and we want to protect that sub-goal. If we disallow all operators that can make the subgoal false by denying those capabilities, then the sub-goal is protected. If no operator can make the sub-goal false, then we are forced to remain in the current connected component of the subgraph for that sub-goal.

But we also want to be sure that the connected component we choose contains a final goal state. Again we can state this in terms of capabilities. If we know that the capability exists of applying an operator that can achieve the remaining un-solved sub-goal, then we know that this connected component contains a final goal state.

So with two sub-goals we can express the conditions that must be true for a connected component with a final goal state by using capabilities. We disallow any operator that could make us leave the connected component for the first sub-goal, and also require that some operator to achieve the second sub-goal is applicable within the connected component. The following section describes a system in predicate logic that uses logical consistency to express each of these requirements.

6.2 Capabilities and consistency

A world state may be stated as a wff. The world state wff is the conjunction of all of the atomic wffs that are true in the world and the negation of all the atomic wffs that are false. For example, if our world consists of block A on block B, then the world state might be represented by the wff:

$$on(A,B) \wedge \sim on(B,A) \wedge clear(A) \wedge \sim clear(b)$$

Other predicates may be necessary, depending on the representation being used. A different representation may also use the wffs $over(A,B)$ and $\sim over(B,A)$.

It is possible to derive formulas in predicate logic that are consistent only with legal world states. GPS, for example, used tests to determine whether a state was legal. In the blocks world we would have the statement

$$\forall x \forall y\,[on(x,y) \rightarrow \sim clear(y)]$$

meaning that a block cannot have a clear top if some other block is on it. We call these wffs ***legality constraints.***

A sub-goal, such as $on(A,B)$, is also a wff. This wff is consistent only with states that have the sub-goal satisfied. If our two sub-goals are logically inconsistent with the legality constraints, then we know that it is not possible to reach a state satisfying both sub-goals. We would like to use logical consistency in a similar fashion to determine whether or not a state with one sub-goal achieved is within a connected component that will allow us to solve the other sub-goal.

To do this we need to find the restrictions that affect capabilities, called ***capability constraints.*** An example of a capability constraint is:

$$\forall x \forall y\,[\sim pickup(x) \wedge on(x,y) \rightarrow \sim pickup(y)]$$

This says that if we cannot pick up one block because it would undo a protected sub-goal, and if that block is on top of a second block, then we cannot pick up the second block either. This is because a block can only be picked up if it is clear, and we cannot clear a block unless we can move all blocks that are on top of it. The capability constraints represent the indirect effects of protecting a sub-goal.

Now suppose we have a set of legality constraints and capability constraints, stated as formulas in predicate logic, so that only legal combinations of states and capabilities are logically consistent with that system. We now assert a wff called a ***sub-goal constraint.*** If our two sub-goals are A and B then we assert

$$A \wedge \sim undo(A) \wedge \sim B \ \wedge cando(B)$$

where *undo*(A) is the conjunction of all capabilities that could undo sub-goal A, and *cando*(B) is the disjunction of all capabilities that can achieve sub-goal B.

A world state is consistent with this sub-goal constraint only if sub-goal A has been achieved in a manner that allows sub-goal B to be achieved without having to undo sub-goal A. The assertion of sub-goal A and $\sim undo(A)$ restricts us to a connected component satisfying sub-goal A. Sub-goal B is not yet satisfied, but by asserting *cando*(B), we know that at least one operator for achieving sub-goal B can be applied. A logical inconsistency in this system of constraint wffs tells us that no world state meets the connected component requirements, and the goals cannot be solved in this order.

We can see here the difficulty in extending this approach to more than two sub-goals. Asserting *cando*(B) is sufficient to guarantee that we can achieve that sub-goal, but the capability of achieving each of several sub-goals independently does not guarantee that we can reach a state that satisfies all of them simultaneously.

6.3 Using the constraints

We can use the constraints described in the previous section to determine whether a given state satisfies the connected component requirements. We want to carry this a step further. Given our sub-goals A and B, we would like to derive the conditions that must be met in order to be in the correct connected component.

Suppose that we assert the sub-goal constraints and find that we have a consistent system of wffs. This tells us that we can solve sub-goal A before sub-goal B, and not have to undo sub-goal A to achieve sub-goal B. But now suppose that some wff X is consistent with this system of wffs, but that $\sim X$ is not consistent. We then know that we can solve sub-goal A before sub-goal B only if condition X is satisfied.

Of these conditions, we do not need to be concerned about any that involve capabilities. We can also ignore any that are direct consequences of legality constraints, since only legal states can be achieved. The remaining wffs, if any, give us the conditions that must be met when sub-goal A is achieved and protected.

Unfortunately, deriving all of the implications and identifying the ones that are important does not appear to be a simple task. The following section describes a technique that is more straightforward.

6.4 Constraints and counterfactuals

Rather than trying to derive all of the conditions that must be met when the first sub-goal is achieved, we can simply solve the sub-goal in the most direct manner, then analyze the resulting world state to find the conditions that prevent the remaining goal from being achieved. The first sub-goal then must be re-planned to remove those conditions. Finding the source of a logical inconsistency is not part of standard first-order logic. A type of logic that does almost exactly what we want is *counterfactual logic*. Counterfactual logic is described in [Rescher 68].

Counterfactual logic consists of taking an inconsistent set of wffs and making the set consistent by throwing out some of those wffs. The wffs are grouped into modal categories that indicate how strongly the wff is believed, with lower numbers having stronger belief. Given a choice of making the system consistent by discarding either of two wffs the counterfactual engine will throw out the wff with the higher modal category.

The counterfactual engine used here is described in [Roach 85b]. The input to the counterfactual engine consists of the constraints discussed earlier in the chapter and a world state. The world state is derived by solving one of the sub-goals without worrying about interactions with other sub-goals. The resulting world state may not be one that is useful in solving the problem, but if this is the case the counterfactual engine will be used to determine why it is not useful. The capabilities for undoing that sub-goal are negated, and capabilities for achieving the remaining sub-goals are asserted.

Figure 6.1 gives the input to the counterfactual engine for the CAB problem described in Figure 2.2. The counterfactual engine and its input are written in a local dialect of Prolog called HC [Roach 85a]. The *mk_facts* rules assert facts such as (*on b c*) for block B on block C. Variables are prefixed by an asterisk. Two modal categories are used. Modal category one assert the fact that no block can be on top of itself. Modal category two asserts facts that are true in the quick-and-dirty world state that we are testing against the connected component requirements.

The *mk_rules* rule assert the constraints that must hold true. The rule:

```
((over *a *b) -> (not (clear *b)))
```

says that if one block is over another, then second block cannot be clear. This is an example of a legality constraint.

The capability used here is *move*, with three parameters. If block A is taken from the top of block B and placed on top of block C, that operation is

```
(move a b c)
```

```
(mk_facts 1
        (not (on a a)) (not (on b b)) (not (on c c))
        (not (over a a)) (not (over b b)) (not (over c c))
)

(mk_facts 2
        (on b c)          (not (on b a))
        (on c a)          (not (on c b))
        (not (on a b))    (not (on a c))

        (clear b) (not (clear a)) (not (clear c))

        (over b c)        (over b a)
        (over c a)        (not (over c b))
        (not (over a b))  (not (over a c))

        (move a *e *f) (move b *g *h) (move c *i *j)
)

(mk_rules
        ((over *a *b) -> (not (clear *b)))
        ((clear *a)(!= *x *a) -> (not (over *x *a)))
        ((on *a *b) -> (over *a *b))
        ((not (over *a *b)) -> (not (on *a *b)))
        ((over *a *b) -> (not (over *b *a)))
        ((over *a *b)(over *b *c) -> (over *a *c))
        ((not (move *a *x *y))(over *a *b)
                -> (not (move *b *z *w)))
)
```

Figure 6.1: Counterfactual program for CAB problem

```
((on b c) (not (move b *0 *1)) (move a *2 b)
    => ((not (move c *0 *1))
        => ((not (over c a))
            => (not (on c a))))
    (not (over b a)))
```

Figure 6.2: Results for CAB problem with counterfactuals

The only capability constraint is:

```
((not (move *a *x *y))(over *a *b) -> (not (move *b *z *w)))
```

indicating that if a block cannot be picked up, then neither can any block that it is over. The rules given here are not complete, but they are sufficient for the solution of this example.

In this example the (*on b c*) goal has been solved by simply putting block B on block C. This approach is wrong, of course, since now block A cannot be put on block B without taking block B off of block C.

If the goal has been solved in a wrong connected component the system is logically inconsistent, and the counterfactual engine identifies the causes of the inconsistency. The counterfactual engine is invoked by the command:

```
(cf ((on b c)(not (move b *x *y))(move a *z b)) 1)
```

The first argument is a list of facts that must be true. These are the sub-goal of having block B on block C, the negation of the capability of moving block B, and the capability of putting block A on block B to achieve the second sub-goal. The second argument requires that consistency be restored by changing only wffs with a modal category higher than one.

The answer returned is shown in Figure 6.2. The first line is the list of facts that must be true. The counterfactual engine then derives the fact that since block B cannot be moved, and block B is on block C, then block C cannot be moved either. But since the system must ensure that block A can be moved, if block C cannot be moved then block C cannot be over block A. If block C cannot be over block A, it cannot be on block A. Finally, by similar reasoning, block B cannot be over block A either.

The key fact that contradicts our initial guess at how to achieve block B on block C is (*not* (*on c a*)). When we put block B on block C, it must be done

in such a way that block C is not over block A. For this problem, adding this new condition as a sub-goal happens to give us a problem that is linear. We saw in chapter 5, though, that the conditions required to specify the right connected component cannot always be added as sub-goals. For this reason, it is unclear how to proceed in general once we have derived the conditions that must be satisfied.

The example shown here executes in three cpu seconds on a lightly loaded Vax 11/780. The counterfactual engine requires about twelve cpu seconds to load the facts and rules. The Prolog that we are using executes at approximately one thousand logical inferences per second (1 K-lips). Currently available Prologs are about two orders of magnitude faster.

If we run the above example with the goal order reversed, trying to put block A on block B before we put block B on block C, the counterfactual engine returns *nil* indicating that it is impossible to solve the goals in that order. The system is inconsistent and cannot be made consistent without throwing out the hypothesis that block A is on block B.

6.5 Summary

A planning algorithm based on the counterfactual technique would need to re-plan to solve the sub-goal according to the constraints returned by the counterfactual engine. The new plan would also have to be tested by the counterfactual engine, possibly turning up new constraints. This is because the engine is grouping all facts into those that are retained and those that are thrown out. Those facts that are retained, however, could be further divided into two groups: those that are constrained to be true, and those that are superfluous. Those facts that are constrained to be true need to be maintained during the re-planning, but the current counterfactual engine does not make this distinction.

This technique does not directly lead to a planning algorithm. The system is capable of determining what makes a state good or bad, but does not tell us how to reach a good state. One possibility for future research is to incorporate this technique into a constraint satisfaction system. It is also not clear how to generalize this technique to more than two sub-goals.

Another problem with this method is that it is not clear how to generate the constraints. A typical formulation of blocks world operators allows the operators to be applied to illegal states, generating new states that may or may not be legal. Because of this, it is impossible to derive the legality constraints by simply analyzing the operators. A legal world state must be taken as a starting point, and from there it seems that enumeration of states may be necessary. The capability constraints also do not seem to be easily derivable.

The technique presented here is currently of theoretical interest only. It gives a system that supports experimentally some of the theory presented in previous chapters. It also represents a formal system that may be useful in future analysis.

Chapter 7

Summary and conclusions

There are two kinds of people in this world: those who divide everything into two groups, and those who don't.
—KENNETH BOULDING

I am a "scruffy" (or hacker or pragmatist) but there seem to be plenty of people in AI who hold that the problems will fall apart if and only if we solve the underlying difficult cases rigorously.
—KENNETH LAWS

This final chapter of the thesis is divided into three parts. The first section discusses how this thesis fits into the broad field of planning and artificial intelligence. The second section reviews the accomplishments of the thesis. The last section gives some possibilities for future research springing from this thesis.

7.1 How this thesis fits into the world of AI

Several authors have divided research in the field of artificial intelligence into two parts, one emphasising theory and one more interested in working systems. The theoretical or "neat" side is accused of being impractical. The practical or "scruffy" side is accused of being just a bunch of hackers.

Artificial intelligence has been described as an attempt to solve intractable problems with tractable algorithms that handle enough cases to be useful. The theoretical side is concerned with analyzing the intractability of a problem. The practical side is concerned with discovering the tractable algorithms.

We have seen examples of both types of work. The STRIPS, ABSTRIPS, LAWALY, HACKER, NOAH and NONLIN systems are examples of the hacker

side. Each could solve a few more problems than could be solved before, but there was no characterization of the types of problems that could or could not be solved. Some authors gave examples of problems that they could not solve; the rest only hinted at the incompleteness of their systems.

The examples of theoretical research in planning are fewer and more recent. GAGS, MACRO-OPS, TWEAK and this thesis are examples of a more formal approach to planning.

I would like to suggest that this theoretical branch of AI research can itself be broken down into two parts. On one side we have the explainers; they devise a planning system, and give a rigorous explanation for its effectiveness. On the other side we have the explorers; they begin with a formal representation, then explore the limits of the problem by analysis with that representation.

MACRO-OPS and TWEAK are good examples of the explanatory side of theoretical AI. Both of these are generalizations of previous work. MACRO-OPS followed several other attempts at using macro operators in planning; TWEAK followed the non-linear style of planning begun by Sacerdoti. Both systems significantly improved on the previous work, and both systems justified their improvements with a formal argument. The formal characterizations explain the success of the new methods.

The exploratory side operates in reverse. Here the formal representation is used to derive a new method. With GAGS, the theory was based on formal languages, and GAGS formed the basis for a planning algorithm. This thesis follows a similar pattern. Beginning with graph theory, some characteristics of planning are investigated. A partial algorithm is described based on this work.

The two groups have different characteristics. The explanatory group tends to have more practical results, since the work ultimately derives from prior "scruffy" research. The ability to prove the effectiveness of these algorithms allows future research to identify potential improvements and to compare various approaches in a more concrete manner.

The theoretical tools developed in exploratory research tend to give better insight into the inherent nature of a problem. While that approach may not lead to practical systems immediately, it should influence the development of future systems.

7.2 What this thesis accomplished

The accomplishments of this thesis can be grouped into two major parts. First, the results on the complexity of planning show us some limitations of a complete algorithm. For worst-case problems, no complete algorithm is better than a brute-force

search. Because of this, complete algorithms are not as interesting as algorithms that take advantage of the characteristics of some useful subset of planning problems.

This work also suggests a number of formal systems that may be useful in further exploratory research. Graph theory has proven useful, as have first-order logic and counterfactual logic. The section on future research gives some reasons that finite state automata and semigroup theory may prove helpful.

7.3 Directions for further research

Most of the worst-case problems we have examined are either unstructured, as with the maze problem, or have state representations that do not reflect the problem's structure. Typical problems, however, often have some useful structure. A good direction for future work would be to quantify the degree of structure in a problem, and identify its effect on the algorithmic complexity. A related idea is to try to change the representation of a problem so that it can be solved more easily. Korf has addressed this problem in [Korf 80].

Another direction for future research is to investigate other formal systems for representing and analyzing planning algorithms. A possibility is finite state automata (FSA). FSA are closely related to both graph theory and formal languages. FSA have many similarities to planning problems: both have states represented by one or more variables, with one initial state and one or more goal states. Some theories of FSA decomposition have been developed using semigroup theory [Stone 73] [Arbib 68].

A state machine may be decomposed into simple interacting state machines if the state variables are independent or partially independent. If the variables are independent, then the state machine decomposes into several smaller machines sharing the same input. The reverse process is the formation of a cross-product machine, where two or more smaller machines that share the same input can be combined into a single machine. With only partial independence, transitions for one machine may depend on the current state of another machine. This interdependence of state variables is reminiscent of goal interactions; a measure of partial interactions might prove useful.

Some simplifying assumptions appear worth looking into. For example, most problems that have been studied have the property that only one operator is available to achieve each sub-goal. Here we are referring to a parameterized operator, corresponding to more than one arc in the problem graph. This assumption appears to create some degree of structure within the problem, but it is not clear how to take advantage of that structure.

Bibliography

[Aho 74] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms,* Wesley Publishing Company, 1974.

[Arbib 68] Arbib, Michael A., *Algebraic Theory of Machines, Languages, and Semigroups,* Academic Press, 1968.

[Berge 73] Berge, Claude, *Graphs and Hypergraphs,* North Holland Publishing Company, 1973.

[Carre 79] Carre, Bernard, *Graphs and Networks,* Clarendon Press, 1979.

[Chapman 85] Chapman, David, "Planning for Conjunctive Goals." Technical Report 802, MIT Artificial Intelligence Laboratory.

[Ernst 69] Ernst, George W., and Allen Newell, *GPS: A Case Study in Generality and Problem Solving,* Academic Press, 1969.

[Fikes 71] Fikes, Richard E., and Nils J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *Artificial Intelligence 2,* 1971, pp. 189–208.

[Hall 68] Hall, P., "Equivalence between AND/OR graphs and context-free grammars." *Comm. of the ACM,* 16 (1973), pp. 444–445.

[Korf 80] Korf, Richard, "Toward a Model of Representation Changes," *Artificial Intelligence 14,* 1980, pp. 41–78.

[Korf 85] Korf, Richard, *Learning to Solve Problems by Searching for Macro-Operators,* Pitman Publishing, 1985.

[Levi 75] Levi, Giorgio and Franco Sirovich, "A Problem Reduction Model for Non-Independent Subproblems." *Proc. of the Fourth International Joint Conference on Artificial Intelligence,* (IJCAI) 1975, pp. 340–344.

[Levi 76] Levi, Giorgio and Franco Sirovich, "Generalized And/Or Graphs." *Artificial Intelligence 7,* 1976, pp. 243–259.

[Rescher 68] Rescher, N., *Hypothetical Reasoning,* North-Holland, 1968.

[Roach 72] Roach, John, "The JANITOR Robot." Internal Report, Computer Science Department, University of Texas at Austin, 1972.

[Roach 85a] Roach, John, and Glenn Fowler, "Virginia Tech Prolog/Lisp: A Dual Interpreter Implementation." Technical Report TR-85-18, Department of Computer Science, Virginia Tech, 1985. Also published in *Proceedings of the Eighteenth Hawaii International Conference on System Sciences,* January 1975.

[Roach 85b] Roach, John, Fred Eichelman, and Doug Whitehead, "A Coherence Logic for Counterfactual Reasoning." Department of Computer Science, Virginia Tech, March 1985.

[Sacerdoti 74] Sacerdoti, E., "Planning in a Hierarchy of Abstraction Spaces." *Artificial Intelligence 5,* 1974, pp. 115–135.

[Sacerdoti 75] Sacerdoti, E., "The Nonlinear Nature of Plans." *Proc. of the Fourth International Joint Conference on Artificial Intelligence,* (IJCAI) 1975, pp. 206–214.

[Sacerdoti 77] Sacerdoti, E., *A Structure for Plans and Behavior,* American Elsevier Publishing Company, 1977.

[Siklóssy 72] Siklossy, L., "Modeled Exploration by Robot." Technical Report TR-1, Department of Computer Science, University of Texas at Austin, April 1972.

[Siklóssy 73] Siklossy, L., J. Dreussi, "An Efficient Robot Planner Which Generates its Own Procedures." *Proc. of the Third International Joint Conference on Artificial Intelligence,* (IJCAI) 1973, pp. 423–430.

[Stone 73] Stone, Harold S., *Discrete Mathematical Structures and Their Applications,* Science Research Associates, Inc., 1973.

[Sussman 75] Sussman, Gerald J., *A Computer Model of Skill Acquisition,* American Elsevier Publishing Company, 1975.

[Tate 77] Tate, Austin, "Generating Project Networks." *Proc. of the Fifth International Joint Conference on Artificial Intelligence,* (IJCAI) 1977, pp. 888–893.