ALGORITHM CAPABILITY AND APPLICATIONS IN ARTIFICIAL INTELLIGENCE

by

KATRINA RAY

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School at the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2008

University of Oregon Graduate School

Confirmation of Approval and Acceptance of Dissertation prepared by:

Katrina Ray

Title:

"Algorithm Capability and Applications in Artificial Intelligence"

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer & Information Science by:

Christopher Wilson, Co-Chairperson, Computer & Information Science
Matthew Ginsberg, Co-Chairperson, Computational Intll Res Lab
Eugene Luks, Member, Computer & Information Science
David Etherington, Member, Computational Intll Res Lab
Jonathan Brundan, Outside Member, Mathematics

and Richard Linton, Vice President for Research and Graduate Studies/Dean of the Graduate School for the University of Oregon.

December 13, 2008

Original approval signatures are on file with the Graduate School and the University of Oregon Libraries.

An Abstract of the Dissertation of

Katrina Ray for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken December 2008

Title: ALGORITHM CAPABILITY AND APPLICATIONS IN ARTIFICIAL INTELLIGENCE

Approved:

Dr. Matthew Ginsberg

Dr. Christopher Wilson

Many algorithms are known to work well in practice on a variety of different problem instances. Reusing existing algorithms for problems besides the one that they were designed to solve is often quite valuable. This is accomplished by transforming an instance of the new problem into an input for the algorithm and transforming the output of the algorithm into the correct answer for the new problem. To capitalize on the efficiency of the algorithm, it is essential that these transformations are efficient. Clearly not all problems will have efficient transformations to a particular algorithm so there are limitations on the scope of an algorithm. There is no previous study of which I am aware

on determining the capability of an algorithm in terms of the complexity of problems that it can be used to solve.

Two examples of this concept will be presented in proving the exact capability of the most well known algorithms for solving Satisfiability (SAT) and for solving Quantified Boolean Formula (QBF). The most well known algorithm for solving SAT is called DPLL. It has been well studied and is continuously being optimized in an effort to develop faster SAT solvers. The amount of work being done on optimizing DPLL makes it a good candidate for solving other problems.

The notion of algorithm capability proved useful in applying DPLL to two areas of AI: Planning and Nonmonotonic Reasoning. Planning is PSPACE Complete in general, but NP Complete when restricted to problems that have polynomial length plans. Trying to optimize the plan length or introducing preferences increases the complexity of the problem. Despite the fact that these problems are harder than SAT, they are with in the scope of what DPLL can handle.

Most problems in nonmonotonic reasoning are also harder than SAT. Despite this fact, DPLL is a candidate solution for nonmonotonic logics. The complexity of nonmonotonic reasoning in general is beyond the scope of what DPLL can handle. By knowing the capability of DPLL, one can analyze subsets of nonmonotonic reasoning that it can be used to solve. For example, DPLL is capable of solving the problem of model checking in normal default logic. Again, this problem is harder than SAT, but can still be solved with a single call to a SAT solver. The idea of algorithm capability led to the fascinating discovery that SAT solvers can solve problems that are harder than SAT.

CURRICULUM VITAE

NAME OF AUTHOR: Katrina Ray

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
New Mexico State University, Las Cruces, NM
Harvey Mudd College, Claremont, CA

DEGREES AWARDED:

Doctor of Philosophy in Computer Science, 2008, University of Oregon
Master of Science in Computer Science, 2004, New Mexico State University
Bachelor of Science in Computer Science / Math, 2002, Harvey Mudd College

AREAS OF SPECIAL INTEREST:

Complexity Theory and Theory of Computation
Artificial Intelligence
Algorithms

RESEARCH EXPERIENCE:

Algorithm Capability and Applications to Nonmonotonic Reasoning, University of Oregon, 2004-2008

Automating Ancestral Reconstruction, University of Oregon, 2006

Games on Graphs: Complexity and Optimal Strategies, New Mexico State University, 2003-2004

Lexical Analysis for the Unicon Programming Language, New Mexico State University, 2002-2003

Designing a Portable Work-Stealing Scheduler, Harvey Mudd College, 2002

Determining the Noise in the Output Signal of Satellites as a Function of Ambient Temperature, Harvey Mudd College in coordination with Northrop Grumman, 2001-2002

Optimal Virtual Topologies for Wavelength Division Multiplexing, Harvey Mudd College, 2001

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science, University of Oregon, Eugene, OR, 2004-2008

Research / Teaching Assistant, Department of Computer Science, New Mexico State University, Las Cruces, NM, 2002-2004

Research / Teaching Assistant, Department of Computer Science, Harvey Mudd College, Claremont, CA, 1999-2002

Summer Intern, Alcatel Internetworking Division, Calabasas, CA, 2000

PUBLICATIONS:

Ray, K. and M. Ginsberg. 2008. The Complexity of Optimal Planning and a More Efficient Method for Finding Solutions. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*: 280-287.

Hartline, J.R., R. Libeskind-Hadas, K. Dresner, E. Drucker, and K. Ray. 2003. Optimal Virtual Topologies for One-To-Many Communication in WDM Paths and Rings. *IEEE/ACM Transactions on Networking* 12(2): 375-383.

Libeskind-Hadas, R., J.R. Hartline, K. Dresner, E. Drucker, and K. Ray. 2002. Multicast Virtual Topologies in WDM Paths and Rings with Splitting Loss. *Proceedings of the Eleventh International Conference on Computer Communications and Networking*: 318-321.

ACKNOWLEDGEMENTS

I wish to thank my advisor and the other members of my committee who have been a tremendous help throughout this entire process. I would especially like to thank Matt Ginsberg for the amount of time and effort that he put into helping me reach my goals and training me for what lies beyond graduation. I am grateful to David Etherington and Chris Wilson for taking time out of their schedules to meet with me, discuss research, read my papers, and offer me advice.

I would also like to thank the rest of the employees at CIRL / OTS for their time, suggestions, contributions, and support. In particular I'd like to thank Heidi Dixon, Tristan Smith, and Eric Merchant, who have taken the time to critique my papers, offer me suggestions, and help me hash out details.

Finally I would like to thank my family and friends for their ongoing support and encouragement. I have been blessed with wonderful parents, great siblings, a close extended family, and many fantastic friendships. For all of you and to all of you, I am truly grateful.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

There are typically many different ways of solving a problem. The best algorithm for a real world problem is often the one that most capably exploits the natural structure of the problem and uses the best optimization techniques. It is not uncommon for there to be strong similarities between the best methods used for exploiting the underlying structure for two different problems. This is because the type of structure present in one problem may resemble the type of structure in another regardless of how different the problems may be in real life. Dissimilar problems may also benefit from using similar optimization techniques such as good data structures, learning mechanisms, and caching. Because problems that appear to be very different may benefit from similar methods, an algorithm that works well in practice for one problem may well be a good candidate for solving another problem.

Using an algorithm for solving a problem besides the one it was designed to solve involves transforming each instance of the new problem into an input for the algorithm and transforming each output of the algorithm into the correct solution for the new problem. It is important that the transformations be efficient because if the transformations dominate the runtime then we are unlikely to gain any benefit by using a good algorithm in between. Clearly not all problems will have efficient transformations to a particular algorithm so there are limitations on the scope of algorithms. In order to

capture this notion of algorithm scope, we propose defining the capability of an algorithm in terms of the complexity of problems that it can solve. We demonstrate how this concept is useful by proving the capability of two well known algorithms and showing how this knowledge can be used in applying one of those algorithms to practical problems.

1.1. Motivation

One of the primary goals in computer science is to identify the most efficient algorithms for solving problems. This usually means that one wants to minimize the total execution time, although it can sometimes also mean that one wants to minimize the amount of memory required by the computation. The overall execution time of a program is determined by the number of instructions required and the amount of time per instruction. Developing better algorithms reduces the number of required instructions, which in turn plays a fundamental role in lowering the total runtime.

Researchers have spent a lot of time developing better algorithms and refining existing ones. Some popular algorithms have been well studied and fine tuned through a variety of different optimization techniques. These techniques include use of good data structures, caching, learning, book-keeping, parameter tuning, etc. For example, the Boolean Satisfiability problem (SAT) is to determine for a given Boolean formula whether or not there is some assignment of the variables that makes the formula evaluate to true. The algorithm most commonly used for solving this problem, DPLL (Davis and Putnam 1960, Davis et al. 1962), receives enough attention that there is a regularly

scheduled competition to see who can develop the fastest SAT solver (SAT Competitions).

There is a saying that to the man with only a hammer, everything looks like a nail. While some problems are more specific and require specialized tools, there can be a lot of value in reusing existing tools to solve new problems. Algorithms that are known to work well in practice on a variety of different instances are often valuable tools for solving other problems. Solving a new problem with an existing algorithm is accomplished by converting from each instance of the problem to an input for algorithm, running the algorithm on the converted input, and converting the output of the algorithm into the correct answer for the problem. The algorithm is treated like a black box where only the input and output are manipulated. Applying popular algorithms to additional problems enables us to take advantage of the research conducted on optimizing those algorithms.

To capitalize on the efficiency of an algorithm for additional problems, the transformations to that algorithm must also be efficient. This imposes limitations on the scope of an algorithm because there are cases where we would like to use an algorithm for a new problem, but no efficient transformations exist. This may be because the new problem is significantly harder to solve than the original problem.

We mentioned that one of the primary goals of computer science is to identify the most efficient algorithms for solving problems. Another is to determine how hard problems are to solve in terms of the amount of time or memory required by the computation. Problems are grouped into equivalence classes, called complexity classes, based on their resource requirements. For example, the class P is the set of problems

solvable in polynomial time on deterministic Turing Machine (which is equivalent to a standard computer) (Garey and Johnson 1979).

The primary relationship between algorithms and complexity theory is that problems are grouped into complexity classes based on the most efficient algorithms for solving them. We introduce a new relationship between these fields by defining the capability of an algorithm in terms of the complexity of problems that it can solve. We show that an algorithm can sometimes be applied to problems of higher complexity than the problem it was designed for, but it is still limited in the complexity of problems it is capable of solving.

Informally, if C_1 is a complexity class and C_2 is a class of functions, we say that an algorithm is C_1,C_2-capable if it can be used to solve all problems in complexity class C_1 using C_2-computable transformations of the input and output. We abbreviate C_1,P-capable to C_1-capable, thus we will assume polynomial transformations of the input and output unless otherwise specified.

Some algorithms are nondeterministic and may have several execution paths running in parallel. Computers are inherently deterministic so they simulate nondeterministic algorithms either in depth-first fashion by running execution paths one after another or in breadth-first fashion by taking one step along each path until a solution can be found. When the definition of capability is formalized it will require a transformation from nondeterministic algorithms to deterministic ones. This transformation implies that algorithm is capable of solving all problems in complexity class C on a standard computer.

By showing that an algorithm is C-capable, we are not suggesting that the algorithm will be equally well suited for all problems in C or that the algorithm will be the best method of solving any particular problem. We are just establishing bounds on the complexity of problems that an algorithm can solve.

There are subtle differences between translating into instances of an algorithm and translating into instances of the problem that the algorithm is designed to solve. For instance, DPLL is the most commonly used algorithm for solving SAT. The complement of SAT, UNSAT, involves determining if a formula is false for all possible assignments of the variables. DPLL can solve UNSAT using a straightforward transformation of the output by returning true when DPLL returns false and vice versa. UNSAT is known to be coNP-complete, hence DPLL can solve problems in NP and in coNP, but this does not imply that NP = coNP. Furthermore, all known algorithms for NP problems require exponential time so they may be applicable to problems beyond NP. We will further explain the differences between capability and complexity in a later chapter.

The notion of algorithm capability is useful for at least three important reasons. The first is that it gives a method of determining if a given algorithm will work for a problem without resorting to trial and error. When a new problem arises, we can prove whether it is solvable with a particular algorithm by knowing the capability of the algorithm and the complexity of the problem. Or if an algorithm cannot solve a particular problem, we can characterize what subset of the problem the algorithm can solve.

The second reason this concept is useful is that it gives us insight into the complexity of problems. If an algorithm A is known to be C capable and the algorithm

can solve problem B, then problem B is in C. Or if A cannot solve B then B is not in C. It gives us a new way of looking at the complexity of problems.

The third reason that capability is useful is that it helps to determine if the algorithm being used is more powerful than necessary for solving a given problem. Algorithms that can solve every computable problem are unlikely to be ideal candidates for problems in NP. As techniques are added to make an algorithm run faster, we typically end up decreasing the capability of the algorithm until the capability is as close as possible to the complexity of the problem that it is designed to solve. There are some exceptions to this which will be noted later.

More concrete examples of how this notion is useful will be given, including showing that DPLL is exactly Δ_2-capable. DPLL is currently used for an assortment of different applications because it is known to work well on a variety of different instances. Having proven the capability, we know exactly what applications it may be used for. In cases where DPLL cannot solve a problem, we can analyze exactly what subset of the problem it can solve.

Quantified Boolean Formulae (QBF) is an extension of SAT, and involves determining whether a given Boolean formula is true where each instance is of the form $\forall x_1 \exists x_2 f(x_1, x_2, \ldots)$. QBF belongs to a complexity class called PSPACE. Modified versions of DPLL have been used for problems of higher complexity such as QBF (Cadoli et al. 1998). When the algorithm branches on a variable that is quantified with $\exists$, one of the branches must return true for the formula to be true. On the other hand, if it

branches on a variable quantified with $\forall$, both branches need to return true. In the QBF implementation of the DPLL algorithm, one version of the algorithm handles existential quantifiers and one handles universal quantifiers. These two versions are called in alternation to provide an answer to the quantified boolean formula. The version for handling existential quantifiers is nearly identical to DPLL except for differences in the unit propagation procedure. The version for handling universal quantifiers is similar to DPLL except that it checks both branches instead of just one. These algorithms will be covered in the background section in more detail.

This application of the DPLL algorithm to QBF does not imply that DPLL is PSPACE-capable since it modifies the original algorithm, not just the input and output. The definition of capability treats the algorithm as a black box, manipulating the input and output as necessary. However, we will show later that the QBF algorithm is exactly PSPACE-capable.

Knowing the capability of DPLL is more interesting if we can demonstrate practical applications. One particular problem that we would like to use DPLL to solve is finding optimal solutions to planning problems. There are some problems that have exponential length plans so determining whether or not there is a plan is PSPACE complete. It is often useful to consider restrictions on planning problems that bound the plan length to be a polynomial. In such cases, planning is in NP. We prove the complexity of finding an optimal plan and conclude that, since it falls within the capability of DPLL, we can solve optimal planning with DPLL. We will describe current

methods of finding optimal plans and how to solve optimal planning more efficiently with DPLL in more detail in a later chapter.

We will also discuss the capability of DPLL in the context of planning problems with preferences, which are typically Σ_2-complete when there is a polynomial bound on the length of the plan. We describe a method for using DPLL in solving planning problems with preferences. In addition, we will discuss the subset of planning with preferences that is solvable with DPLL by describing some subsets of planning with preferences that are in Δ_2.

Another problem that we would like to use DPLL for is called Nonmonotonic Reasoning (NMR), which involves being able to retract inferences that are no longer valid when they contradict new information that is added to the knowledge base. Most of the interesting problems in NMR are Σ_2-complete, thus, we cannot use the standard DPLL algorithm for many of the problems in NMR. We can, however, characterize the subset of nonmonotonic reasoning problems that it can solve by describing the subset of NMR that is in Δ_2. We also provide a method of solving specific nonmonotonic reasoning problems that are in Δ_2.

1.2. Outline

Chapter two contains the background information necessary for understanding the ideas presented in this thesis. This includes defining the Boolean satisfiability problem and the quantified Boolean formula problem and providing details on the most popular algorithms for solving them. The necessary background material also includes an

introduction to computational complexity theory. Additionally, we describe two AI problems that will be used later as applications of the theoretical ideas presented. Specifically, we will describe planning problems and nonmonotonic reasoning, present the formal representations, and give an overview on the complexity of these problems.

In chapter three, we give a formal definition of algorithm capability. As mentioned earlier, the definition of capability requires us to convert from nondeterministic algorithms to deterministic ones. We will also discuss how translating into inputs to an algorithm differs from translating into instances of the problem that the algorithm is designed to solve, and how algorithm capability differs from problem complexity.

Chapter four provides examples of capability by proving that DPLL is exactly Δ_2-capable and that the most commonly used algorithm for solving QBF is exactly PSPACE-capable. The first result can be shown by demonstrating that the Odd Maximum Satisfiability (OMS) problem is Δ_2-complete and that DPLL can be used for OMS. This shows that DPLL can solve all problems in Δ_2. We then show that this is also an upper bound on the capability of DPLL by proving that if a problem can be solved with DPLL that it must be in Δ_2.

DPLL has been modified many times to improve its execution time. One example is using intelligent branch heuristics to select the optimal branching order. While the basic algorithm is Δ_2-capable, some of the later additions are not. The most well known branching heuristics used by modern solvers are not known to be Δ_2-capable. An analysis is given in Chapter four of some of the later additions and how they affect the capability

of DPLL. We also analyze how removing the additions that are not known to be Δ_2-capable affects the runtime.

The final section in chapter four is the proof that the modified version of DPLL that solves QBF is PSPACE-capable.

Knowing that DPLL is Δ_2-capable is far more interesting if we can demonstrate some practical application. In chapter five, we show such practical applications by describing how to use DPLL to solve certain problems in Artificial Intelligence. Specifically we look at the problem of finding optimal plans for planning problems. This problem is beyond NP, but within the scope of what DPLL can solve. We also consider planning problems with preferences. Though DPLL cannot solve all planning problems with preferences, we can talk about the types of preferences that it can handle. Knowing the capability of DPLL is applicable to nonmonotonic reasoning. We show how to use DPLL for solving nonmonotonic SAT problems. We characterize the subset of NMR that can be solved with DPLL since most problems in NMR have a higher complexity than DPLL can solve.

Chapter six presents conclusions and a discussion on possible avenues of future research.

CHAPTER II

BACKGROUND

Some background material is necessary to understand the ideas presented in this dissertation. This chapter is divided into four sections, each one devoted to a specific topic. In the first section, we familiarize the reader with the Satisfiability problem, the Quantified Boolean Formula problem, and the most commonly used algorithms for solving them. This information is helpful because we give examples of algorithm capability by proving the capability of these algorithms.

In the second section, we will present some of the key concepts in complexity theory and computability theory. Understanding these concepts is crucial in understanding both the idea of algorithm capability as well as the specific examples because capability is defined in terms of the complexity of problems that an algorithm can solve. Once we have proven the capability of a particular algorithm, we need to know the complexity of a problem in order to show whether or not the algorithm can solve that problem.

The last two sections contain descriptions of certain problems in planning and in nonmonotonic reasoning, including their formal representations and a brief overview of their computational complexities. In later chapters, we show that knowing the capability of the algorithm for solving SAT is valuable by demonstrating that the algorithm can solve these problems.

2.1. Satisfiability and DPLL

After sorting, Satisfiability is perhaps the most famous and well-studied problem in computer science. The problem is to determine if a given Boolean formula in conjunctive normal form (CNF) is satisfiable, meaning that there is some assignment of the variables that makes it evaluate to true. A formula is in CNF if it consists of a set of clauses joined by $\wedge$ (logical and) where each clause is a set of literals joined by $\vee$ (logical or). A literal is a variable or its negation. We use the symbol $\neg$ to denote negation. For example, $(a \vee b) \wedge (\neg a \vee c)$ is a Boolean formula in CNF with two clauses. The first clause is $(a \vee b)$ and the second is $(\neg a \vee c)$.

DPLL is a recursive algorithm for solving Satisfiability first introduced by Davis and Putnam (Davis and Putnam 1960) and later modified by Davis, Logemann, and Loveland (Davis et al. 1962). Starting with an initially empty partial assignment of variables, we construct a solution by first performing unit propagation. Unit propagation involves finding all "unit clauses" which contain one unvalued literal and only other literals that are set to false. Clearly the unvalued literal must be set to *true* in order to satisfy the formula. At this step in the procedure, we set all variables that correspond to unit clauses. Sometimes setting a variable during unit propagation creates new unit clauses. The algorithm will continue to process unit clauses until no more are available or being created.

Next we return the value of the formula if it has already been determined by the current partial assignment. Finally we select an unassigned literal, l, and make one recursive call with l set to *true* and one with l set to *false*.

The algorithm returns a satisfying assignment if one is found and returns UNSAT if a contradiction is reached. Pseudo-code is shown in Figure 2.1.1. Some versions of DPLL just return SAT or UNSAT instead of returning the satisfying assignment.

```
1: DPLL (Clauses, Assignment)
2:   Assignment ← UNIT-PROPAGATE (Assignment)
3:   if Clauses is empty
4:     then return Assignment
5:   if  c ∈Clauses and c is empty
6:     then return UNSATISFIABLE
7:   l ← GET-NEXT-VARIABLE ()
8:   Solution ← DPLL (Clauses, Assignment∪{l})
9:   if  Solution ≠ UNSATISFIABLE
10:    then return Solution
11: return DPLL (Clauses, Assignment ∪{¬l})
```

Figure 2.1.1 Pseudo-code for DPLL.

There are many implementations of the DPLL algorithm. Some of the more popular ones include zChaff (Moskewicz et al. 2001), RSAT (Pipatsrisawat and Darwiche 2007), SATzilla (Nudelman et al. 2004), and Minisat (Eén and Sörensson 2004). These solvers have all performed quite well in recent SAT competitions.

Several modifications have been made to the original algorithm in order to make it more efficient. These modifications include using intelligent branch heuristics, learning new clauses, optimizing data structures, parameter tuning, and many others. These modifications and others will be discussed in more detail in chapter IV.

Another problem similar to the satisfiability problem is determining whether a given formula is a tautology, in other words whether it is true for all assignments to the variables. Thus, a satisfiability problem is of the form $\exists x_1, \ldots, x_n f(x_1, \ldots, x_n)$ and a tautology problem is of the form $\forall x_1, \ldots, x_n f(x_1, \ldots, x_n)$.

This can be generalized by allowing alternation of quantifiers instead of applying the same quantifier to all variables. This problem is called Quantified Boolean Formula (QBF), and a QBF formula is of the form $Q_1 x_1\ Q_2 x_2 \ldots f(x_1, x_2 \ldots)$. Each Q_i represents a quantifier and may either be $\forall$ or $\exists$.

The most well known algorithm for solving QBF is similar to DPLL. It involves writing one procedure to handle the existential quantifiers and one to handle the universal quantifiers. For each quantified variable, the algorithm calls the appropriate version. Pseudo-code for solving QBF is given in figure 2.1.2. The unit propagation procedure is a little different from the standard one in DPLL. If an existentially quantified variable appears in a clause where all other literals are false, then the literal containing that variable must be set to *true*. If a universally quantified variable appears in a clause where all other literals are false, then we can return UNSATISFIABLE or FALSE.

```
1: SOLVE (Clauses)
2:  if outermost variable quantified by ∃
3:     then return SOLVE-Σ (Clauses, {})
4:     else return SOLVE-Π (Clauses, {})


1: SOLVE-Σ (Clauses, Assignment)
2:  Assignment ← UNIT-PROPAGATE (Assignment)
3:  if Clauses is empty
4:     then return Assignment
5:  if  c ∈Clauses and c is empty
6:     then return UNSATISFIABLE
7:  l ← GET-NEXT-VARIABLE ()
8:  Solution ← DPLL (Clauses, Assignment∪{l})
9:  if  Solution ≠ UNSATISFIABLE
10:    then return Solution
11: return DPLL (Clauses, Assignment ∪{¬l})


1: SOLVE-Π (Clauses, Assignment)
2:  Assignment ← UNIT-PROPAGATE (Assignment)
3:  if Clauses is empty
4:     then return Assignment
5:  if  c ∈Clauses and c is empty
6:     then return UNSATISFIABLE
7:  l ← GET-NEXT-VARIABLE ()
8:  Solution ← DPLL (Clauses, Assignment∪{l})
9:  if  Solution = UNSATISFIABLE
10:    then return UNSATISFIABLE
11: return DPLL (Clauses, Assignment ∪{¬l})
```

Figure 2.1.2 Pseudo-code for QBF algorithm.

Cadoli, Giovanardi and Schaerf proved several theorems about when a QBF is trivially true or false (Cadoli et al. 1998). The QBF solver that they use is similar to the above algorithm, but they also check if the formula is trivially true or false by seeing if it satisfies the conditions specified in the theorems that they proved. This makes the algorithm run more efficiently, though the underlying algorithm is essentially the same.

2.2. Computational Complexity and Computability

One branch of computer science, known as computational complexity theory, studies how hard problems are to solve in terms of the resource requirements (usually the amount of time or memory) for computing the solution. Problems are grouped into equivalence classes called complexity classes based on their resource requirements. Most often a complexity class is defined as a set of problems requiring no more than X amount of resource Y given a specific model of computation. The most common resources are time and memory. Often we want to restrict to a polynomial amount or a logarithmic amount of one of these resources.

We say that a problem is in a complexity class if it can be solved by some algorithm that satisfies the resource restrictions of that class. Typically, we are most interested in the lower bound on complexity which is equivalent to the resource requirements of the best algorithm for solving the problem. The best algorithm for solving a problem is the most accurate reflection of how hard the problem actually is to solve.

In order to develop a formal notion of complexity theory, we are best served by starting with the formal definitions of problems and algorithms. Generally speaking, a problem is a question that we wish to answer given some input value. Each possible input string corresponds to an instance of the problem and there is one output or one set of possible outputs for each input.

Often in complexity theory, we consider ***decision problems***, where the answer is either *yes* or *no*. Decision problems can be mathematically formulated as languages which are a set of strings. Under this representation, a ***problem*** is the set of all strings that represent *yes* instances. For instance, SAT is the set of all strings that represent satisfiable Boolean formulae.

It is not much of a hindrance to restrict our attention to decision problems. For most problems there is a natural decision version of the problem. For example if we want to find the largest cycle in a graph, the natural decision version is to ask if there is a cycle of some fixed length k. We can often find the solution to the original problem by asking a small number of decision problems. In our example if there is a cycle of length k then we decrease and ask again; if there is not a cycle of length k then we increase and ask again. We repeat this process until we find the actual value of k that answers the original problem.

An ***algorithm*** is a step by step procedure for solving a problem. We analyze algorithms in terms of the worst case run time, where the run time is expressed as a function of the size of the input. When we express the runtime of a problem, it is common to only write the most significant term and to ignore constants. So if the runtime

of a problem is T $(n) = 3n^2 + 4n + 1$ then we would write that the runtime is $O(n^2)$. In some cases we may analyze algorithms in terms of a different resource such as the amount of memory required by the computation.

An algorithm can be expressed mathematically as a ***Turing machine***. A Turing machine (TM) is a model of computation first proposed by Alan Turing in 1936 (Sipser 1997). Each Turing machine consists of a set of states, an infinite tape (which serves as memory), a tape head or pointer to the current tape square, and an internal transition function which determines how the machine operates. Initially the tape contains the input string and is blank on all other squares. The machine can store additional information by writing to the tape squares. The tape pointer initially points to the beginning of the string and the machine starts in a special start state. The machine will execute until it enters a separate *accept* state or *reject* state. If it never enters one of these two states then it continues computing forever.

The internal ***transition function*** tells us how to get from one step in the computation to the next. The transition function has two inputs: the current state of the machine and the contents of the tape square where the tape head is currently pointing. There are three outputs of the transition function: the state that the machine transitions into, the symbol that we write to the tape square replacing the input symbol, and the direction that the tape head moves (left or right). An example of a simple TM is given in figure 2.2.1. The transition function is shown using directional arcs that are labeled with three elements: the symbol being read from the tape (1, 0, B for blank), the symbol being written to the tape (1, 0, or B for blank), and the direction that the tape head pointer

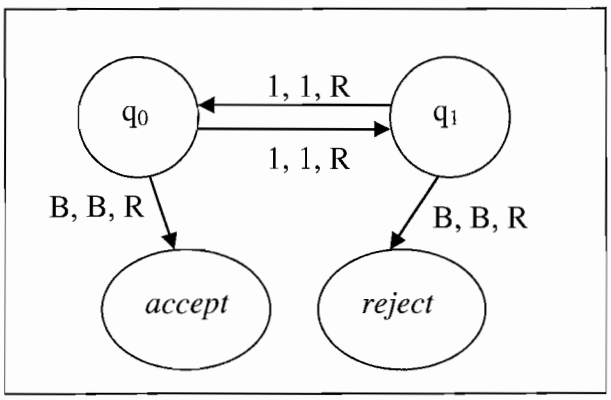

Figure 2.2.1 Example of a simple Turing Machine. The label "B" is used to represent blank. As the input value it means that the square is blank. As the output value it means that we erase the value that is there or leave it blank if it is already blank.

moves (L for left or R for right). For example if we read a 1, write a 1 back to the tape and move right, then the arc is labeled with 1, 1, R. The TM shown in the figure represents all binary strings that have an even number of ones. An ***execution path*** is a valid sequence of states for a given input that can legally occur according to the transition function.

The machine is ***deterministic*** if the transition function behaves like a true function in that there is exactly one output triple for every possible input pair. A ***nondeterministic*** machine is one in which the transition "function" is more of a relation. Each input pair corresponds to a possibly empty set of possible outputs. In other words, from any given point in the computation, there may be no transitions or there may be multiple possible transitions leading to potentially different outcomes. This leads to there being multiple execution paths through the machine. It may be that some of the execution paths end in an accept state, some end in a reject state, some terminate when there are no possible

transitions from the current state, and others may not halt. The machine accepts the input string if there is at least one execution path from the start state to the accept state.

A deterministic TM is capable of computing everything that a standard computer can. Essentially each deterministic TM represents a computer that solves exactly one problem. It is a mathematical expression of a particular algorithm for solving the problem. Algorithms are generally specified by pseudo-code representations that describe how to implement them, but the TM representation is a more mathematical approach. Algorithms could equivalently be expressed using lambda calculus notation.

Now that we have defined what we mean by the terms ***problem*** and ***algorithm***, we are ready to delve into computational complexity. We group problems into equivalence classes based on how hard they are for a computer to solve. Many complexity classes are defined as the set of problems requiring no more than X amount of resource Y given a specific model of computation. For example, the two most well known classes in complexity theory are P and NP. P and NP are defined as the set of problems solvable in polynomial time on a deterministic or nondeterministic Turing machines, respectively. The main advantage of nondeterministic Turing machines is that there may be several execution paths running at the same time, which allows us to test multiple possibilities at once. Another way of characterizing NP is the set of problems whose solutions can be verified in polynomial time. In other words, if we are given a problem and a small proof that the answer is *yes*, we can check in polynomial time whether the proof is correct. (By small, we mean polynomial in size). Consider SAT for

example. A satisfying assignment would suffice as a proof that an instance is a satisfiable formula, and is clearly polynomial in the size of the input. Thus SAT is in NP.

Clearly $P \subseteq NP$ because the set of deterministic Turing machines forms a subset of the set of nondeterministic Turing machines. Another way of looking at it is that if we can solve the problem in polynomial time then certainly we could verify a solution in polynomial time. One of the most important open questions in computer science is whether this containment is proper. In other words, it is still unknown whether there are problems that can be solved in polynomial time on a nondeterministic machine that cannot be solved in polynomial time on a deterministic machine. Most people believe that $P \neq NP$. We know that SAT is in NP, but it is believed that SAT is not in P. If this belief is correct then there is no polynomial time deterministic algorithm that will solve SAT.

The complement of a decision problem is the problem that reverses the *yes* and *no* answers. When we consider a problem as a language or set of strings that correspond to the yes instances, the complement of a problem is the set complement of the language. For example, determining whether a number is prime is the complement of determining whether a number is composite. Determining whether a Boolean formula is unsatisfiable is the complement of SAT.

The set of problems whose complement is in NP is known as coNP. In general, the set of problems whose complement is in complexity class C is expressed as coC. Note that coNP is not the complement of NP, but rather a decision problem is in coNP if its complement is in NP. It is still unknown whether NP = coNP, but P is known to be a subset of both. Since NP is the set of problems where *yes* answers can be verified in

polynomial time, coNP is the set of problems whose *no* instances can be verified in polynomial time.

Sometimes, there are other resources that are equally important or more important than the amount of time required. Another well known class is PSPACE which is the set of problems solvable with a polynomial amount of memory. P ⊆ PSPACE because if we are only given a polynomial amount of time steps, we cannot possible touch more than a polynomial number of memory squares in our computation.

Nondeterministic polynomial space (NPSPACE) has been shown to be equivalent to deterministic polynomial space. Thus NP ⊆ PSPACE since NP ⊆ NPSPACE for the same reason P ⊆ PSPACE. We could also argue that NP ⊆ PSPACE by the fact that each execution path requires polynomial time (thus polynomial space), and if we execute the paths in sequence then we could use the same section of memory for each of the execution paths. Though NP and coNP are both known to be contained in PSPACE, it is unknown whether the containment is proper. See figure 2.2.2 for a diagram of how these classes are related.

SAT is in NP and QBF is in PSPACE. We will later prove that the algorithm we described earlier for solving QBF is capable of solving problems in PSPACE and no more.

There are a number of classes that fall in between NP and PSPACE. Some of the most noteworthy ones can be defined in terms of oracles. An oracle for a problem is an external device that can determine the answer to the problem in a single time step. For

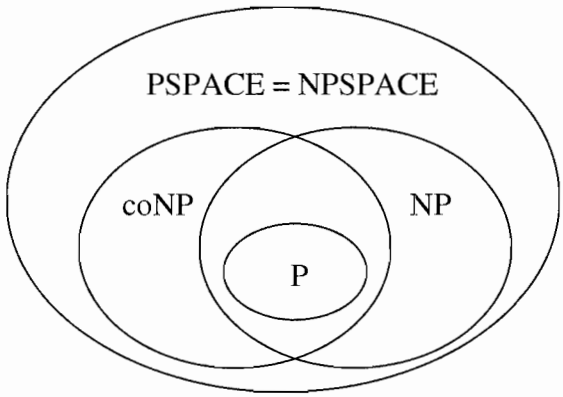

Figure 2.2.2. Diagram showing the relationship between complexity classes.

instance, if we have an oracle for SAT then it could tell us instantaneously whether a given formula is satisfiable. Oracles are not real devices (that I know of), but they are of theoretical value and help us to differentiate the relative complexity of various problems.

In terms of Turing machines, an oracle machine is one in which we have a separate oracle tape, an oracle state $q_?$, and special states q_{yes} and q_{no}. These components correspond to an oracle for a specific problem. The oracle tape may be written to at any point during the computation. When we enter the oracle state, the machine transitions to q_{yes} if the string on the oracle tape encodes a *yes* instance of the problem that the oracle answers and q_{no} if the string encodes a *no* instance. See figure 2.2.3 for a diagram of an oracle machine. The figure represents a generic oracle TM and only shows a start state, accept and reject states, and the oracle. It does not show any additional internal states or the transition function which are dependent upon the specific problem being considered.

If A and B are complexity classes, A^B denotes the set of problems that are in A given an oracle for a problem in B. For instance P^{NP} is the set of classes that are in P given an oracle for SAT or some other problem in NP. Stockmeyer (1976) identified

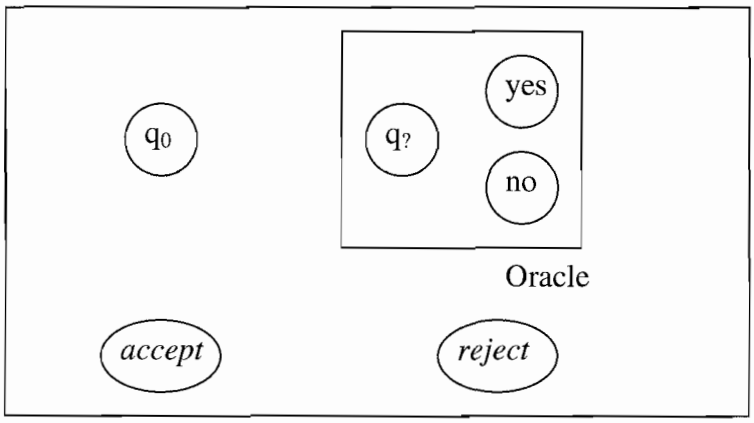

Figure 2.2.3. Diagram of an oracle machine.

many classes in between NP and PSPACE when he defined the polynomial hierarchy. First we let $\Delta_0^p = \Sigma_0^p = \Pi_0^p = \mathrm{P}$. Now for $i \geq 0$ define:

$$\Delta_{i+1}^p = \mathrm{P}^{\Sigma_i}$$
$$\Sigma_{i+1}^p = \mathrm{NP}^{\Sigma_i}$$
$$\Pi_{i+1}^p = \mathrm{coNP}^{\Sigma_i}$$

The p is used to indicate that we are referring to the polynomial hierarchy (as opposed to the arithmetic hierarchy). However, we will not be using the arithmetic hierarchy so we have chosen to omit p and write Δ_i, Σ_i, and Π_i.

Each level of the hierarchy is defined as the set of problems that can be solved with an oracle for a problem in the level below it. In other words, suppose that we have an oracle for solving a problem in Σi. Δ_{i+1} is the set of problems in P given access to the oracle, Σ_{i+1} is the set of problems in NP given access to the oracle, and Π_{i+1} is the set of problems in coNP given access to the oracle. Π_i is equivalent to coΣ_i. See figure 2.2.4 for a diagram of the relationship between classes in the polynomial hierarchy. Based on the

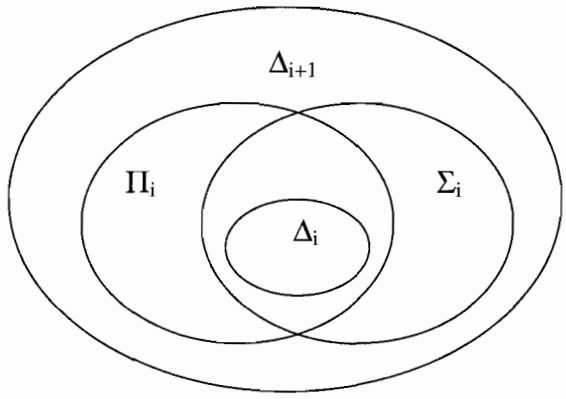

Figure 2.2.4. Relationship between classes in the polynomial hierarchy.

definition, Δ_1 = P, Σ_1 = NP, and Π_1 = coNP. Further note that Δ_2 is P with an NP oracle, Σ_2 is NP with an NP oracle, and Π_2 is coNP with an NP oracle. The polynomial hierarchy is also expressed as PH = $\cup\Sigma_i$.

These definitions are relevant to our work in characterizing the capability of an algorithm in terms of the complexity of problems that it can be used to solve. When we give more concrete examples of capability, we will show that DPLL is exactly Δ_2 capable, whereas some of the problems that we would like to use it for are higher up in the polynomial hierarchy. Many problems in nonmonotonic reasoning, for instance, fall into Σ_2 or Π_2. Understanding the polynomial hierarchy is essential for understanding what variations of nonmonotonic reasoning can be solved with DPLL. Variations on planning problems also have varying degrees of complexity. Understanding the complexity of these problems helps us to determine whether they can be solved with DPLL.

It is an open question whether PH = PSPACE, but if so, then the hierarchy collapses to some finite level and PH = Σ_i for some *i*. Similarly if level *i* and level *i*+1 are equal, then they are also equal to every level above including PSPACE. This means that if $\Sigma_i = \Sigma_{i+1}$ or $\Pi_i = \Pi_{i+1}$ then the hierarchy collapses to level *i*. The reason for this is that if $\Sigma_i = \Sigma_{i+1}$ then a Σ_i oracle is equivalent to a Σ_{i+1} oracle. Thus $\Sigma_{i+1} = \Sigma_{i+2}$. This argument can be extended to every level above Σ_i. Contrarily, if containment is proper at any level, then it is proper at every level below it.

Stockmeyer (1976) also showed that a problem is in Σ_i if and only if it can be expressed as $\exists x_1\ \forall x_2 \ldots Q_i x_i\ R\ (x_1 \ldots x_i)$ where R is polynomially computable and *i* is the number of quantifier alternations. Analogously, a problem is in Π_i if and only if it can be expressed with *i* quantifier alternations where the first quantifier is $\forall$. Each level of the hierarchy can be described in terms of the number of quantifier alternations. QBF is not in any level of the hierarchy because the number of quantifier alternations is not bounded by a constant.

A more intuitive understanding can be gained by looking at specific examples. Primality testing (determining if a number is prime) is in P since it can be solved in polynomial time (Agrawal et al. 2004). This result is fairly recent and was unknown for a long time until Agrawal *et al.* showed a polynomial algorithm for solving the problem. We mentioned earlier that SAT is in NP and UNSAT and TAUT are both in coNP. Another problem in NP is the problem of deciding whether two boolean formula are equivalent. To look beyond NP, consider the set of boolean formula that have no smaller

equivalent formula. This problem is called MINIMAL. The complement of this problem is the set of boolean formula that are not minimal (*i.e.*, formulas that have a smaller equivalent). NONMINIMAL can be solved in NP with an oracle for the equivalence problem (which is in NP). Therefore, NONMINIMAL is an example of a problem in Σ_2 which implies that MINIMAL is in Π_2 (Meyer and Stockmeyer 1972).

There are also many classes in between the different levels of the polynomial hierarchy. Recall that the class Δ_{i+1} is the class or problems solvable with a polynomial number of queries to an oracle for a problem in Σ_i. Another interesting set of complexity classes are those where we are allowed only a logarithmic number of queries to a Σ_i oracle. This is represented as Θ_{i+1}.

Another way to define the difference between Δ_2 and Θ_2 is in terms of the types of queries that are allowed. Queries to an oracle can be either adaptive (also called serial) or nonadaptive (also called parallel). So far we have only considered adaptive queries where we are allowed to determine the next query based on the current state of the computation including answers to prior queries. Nonadaptive queries are ones that must be predetermined from the start. For example, suppose that I am trying to guess a randomly generated number. If I have to list all of my guesses in order up front then these would be nonadaptive queries. If I can ask whether the number is higher than my guess, then I can base my next guess on the answers to previous queries. These are adaptive queries and it generally requires fewer of them in order to guess the random number. It has been shown that having a polynomial number of parallel queries is equivalent to having a logarithmic number of serial queries. Θ_2 is the class of problems solvable with a logarithmic number

of serial queries to an NP oracle or a polynomial number of parallel queries to an NP oracle.

These classes are applicable to our research because Θ_2 resides in Δ_2 but above NP. Since we show that DPLL is Δ_2-capable, this means that problems in Θ_2 are solvable with DPLL even though Θ_2 is higher than the complexity of SAT. There are interesting problems in default logic that have been proven Θ_2-complete. There are also variations on planning problems that can be solved in polynomial time with a logarithmic number of queries to an NP oracle.

Another example of a class between levels of the hierarchy that will be used later is the class D^P. The class D^P contains the decision problems whose yes instances are characterized by the conjunction of an NP property and an independent coNP property. We could clearly answer these type of questions in polynomial time with an NP oracle so $D^P \subseteq \Delta_2$. Some of the problems we look at later are exactly D^P (Complexity Zoo), which is a subset of Δ_2, so we can solve these problems with DPLL. See figure 2.2.5 for a diagram that relates NP, D_P, Δ_2, and Θ_2.

Note that NP$\cup$coNP is the set of problems solvable with a single call to an NP oracle; Θ_2 is the set of problems solvable with a logarithmic number of calls to an NP oracle; and Δ_2 is the set of problems solvable with a polynomial number of calls to an NP oracle.

When we talk about complexity classes, we are describing a set of problems that are roughly just as hard to solve as one another. In order to group problems that have

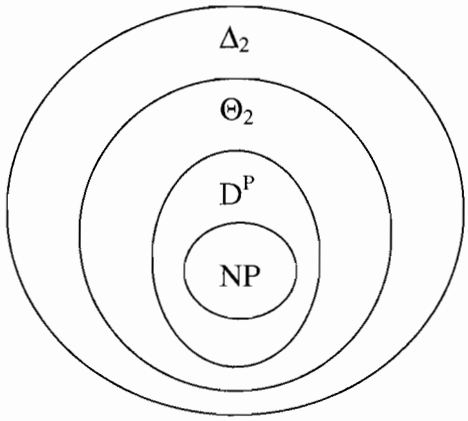

Figure 2.2.5. Diagram showing the relationship between NP, D^P, Θ_2, and Δ_2.

similar requirements, we would like to be able to reduce from one problem to the other to show that the two problems are equally difficult. It is therefore important to define the notion of ***reducibility***.

When we represent a problem as a language, it is a set of strings over some finite alphabet Σ. The set of all possible strings made from the letters in Σ is denoted Σ*. A language *A* is ***mapping reducible*** or ***many-to-one reducible***[1] to another language *B* if there is a computable function f: $\Sigma^* \rightarrow \Sigma^*$ such that $w \in A$ iff $f(w) \in B$. This type of reduction is a reduction between decision problems. This is written $A \leq_m B$. The function f is called a reduction from *A* to *B*. If the function f is in P then it is called a ***polynomial-time reduction***.

Having defined reducibility, we are now prepared to define one of the most important concepts in complexity theory. Given a complexity class C, a problem is C-

[1] There are many other types of reductions, but this is the one that we will be using to describe reductions between languages. Whenever we use the word reduction in regards to languages, we are referring to a many-to-one reduction.

hard if all other problems in C can be reduced to it. If a problem is in C and it is C hard then it is C-***complete***. Intuitively, if a problem is C-complete, then C is both an upper and a lower bound on the complexity (Stockmeyer 1987). Most complexity classes are defined in terms of polynomial-time reductions, but there are some classes that are defined by other reductions such as a logarithmic-time reduction or logarithmic-space reduction. Unless otherwise specified, we are referring to polynomial-time reductions.

We hinted at the notion of completeness earlier in saying that the complexity of SAT is exactly NP. What we mean by this is that SAT is NP complete. It is solvable by a nondeterministic machine in polynomial time and all other problems in NP are reducible to it. Thus if we can develop an algorithm for SAT that proves SAT is in P, then all other problems in NP must also be in P. This would mean that P = NP.

Note, if we know that $A, B \in$ C and A is C-complete, B is proven to be C-complete if we can show that there is a reduction from A to B. Reductions are transitive; since all of C reduces to A and A reduces to B, then all of C reduces to B. It is also important to note that if a complete problem is shown to be polynomial-time solvable then all other problems in the class must also be polynomial-time solvable.

We know that SAT is NP-complete and that QBF is PSPACE-complete. When we show that DPLL is Δ_2-capable, it implies that it cannot be used to solve QBF because QBF is PSPACE-hard. It does not suffice to know that QBF is in PSPACE because that only shows PSPACE is an upper bound on the complexity. Knowing that QBF is PSPACE-complete means that PSPACE is both an upper and lower bound on the complexity so it is not in Δ_2 unless PSPACE = Δ_2.

So far we have discussed classes of languages. Most real world problems are not decision problems but rather they are of the form where we produce some output for each possible input. Though these problems are not always functions[1], we refer to these problems as functional problems. There are complexity classes that correspond to this more traditional view of problems. These are called ***functional complexity classes*** even though the problems are not always functions.

The decision version of SAT is the question of whether or not a Boolean formula is satisfiable. The functional version of this problem is usually to ask for a satisfying assignment of a Boolean formula if one exists and UNSAT otherwise. This problem can still be solved using an NDTM. Instead of either accepting or rejecting the input, the machine could return the string on the tape when it enters an accepting state and reject the input otherwise. These machines are called ***function-computing Turing machines***. The set of problems solvable in polynomial time by function-computing NDTMs is known as FNP.

There is a direct relationship between a NP and FNP. As we mentioned earlier, one way of characterizing NP is the set of problems that have a polynomial sized witness of yes instances. For any language $L \in NP$, we can create a relation $R(x, y)$ between the yes instances and the witnesses. This relation has the property that for all instances x, there exists a y such that $R(x, y)$ if and only if $x \in L$. There may be many different

[1] A function is a mathematical object that returns a specific output for each input. By definition functions have exactly one output for every input. The problem of finding a satisfying assignment for a boolean formula, for example, is not a function because there is more than one possible output for some inputs. It would probably be more appropriate to view these classes as a class of relations or a class of procedures.

relations for each language since there may be many ways of providing witnesses. Each of these relations gives rise to one functional problem of the form: Given x, find a string y such that R (x, y) or return *no* if no such string exists.

We can also define FP^{NP} (also written $F\Delta_2$) and $FP^{NP[\log n]}$ (also written $F\Theta_2$). $F\Delta_2$ is the class of problems that can be computed in polynomial time by a function-computing deterministic Turing machine with an NP oracle. $F\Theta_2$ is the set of function problems that can be computed in polynomial time with access to an NP oracle where we are only allowed to make O $(\log n)$ queries to the oracle.

Krentel defined another functional complexity class as the set of optimization problems called OptP (Krentel 1988). This is the set of problems that can be solved by a nondeterministic machine in polynomial time that writes some value on each accepting path. The machine will return the optimal value out of all of the accepting paths. The value written on each path must have a polynomial number of bits. Krentel showed a number of interesting problems to be complete for OptP, such as finding the lexicographically maximum satisfying assignment. He also demonstrated a relationship between OptP and Δ_2 by showing that every problem in Δ_2 decomposes into two steps where the first is an OptP problem and the second is a polynomial-time computable predicate. This is because the number of bits needed to express the value of an OptP function roughly corresponds to the number of queries to an NP oracle needed to determine the answer. Furthermore he shows that $L \in \Delta_2$ if and only if L can be written as $\{x \mid \mathrm{R}\ (x, g(x))\}$ where $g \in$ OptP and R is a polynomial-time computable predicate.

The reduction between languages will not work for reducing between problems in OptP or other functional complexity classes. In order to talk about classes of functions (or function-like problems), we need a form of reduction that will allow us to reduce these problems to one another. Though these problems are not necessarily functions, a reduction between functions will suffice for these problems also. A ***metric reduction*** from f to g is a pair of P-computable predicates T_1 and T_2 such that $\forall x \in \Sigma^* f(x) = T_2 (x, g(T_1 (x)))$. T_1 transforms the input into a valid input for g and T_2 transforms the output so that it correctly corresponds to the output of the function f. Note that all of the reducibilities described are transitive relations. If A is reducible to B and B is reducible to C then A is reducible to C. A problem is complete for OptP if it is in OptP and all other problems in OptP are metrically reducible to it.

Krentel used the relationship between OptP and Δ_2 to establish the canonical Δ_2-complete problem. We use the completeness result in demonstrating that DPLL is exactly Δ_2-capable. Also, since Krentel showed that there are P-computable transformations between problems in OptP and problems in Δ_2, DPLL is OptP-capable.

Complexity theory is parallel to another topic in the theory of computation called ***computability theory***. In complexity theory, we attempt to classify how hard problems are for a computer to solve. However, there are problems that cannot be solved by a computer regardless of how much time or memory we are allowed (assuming time and memory are both finite). The study of what can and cannot be computed is known as computability theory.

One problem that computers are incapable of solving is called the Halting problem. When writing programs, sometimes an error in the code causes an infinite loop. It can often be difficult to detect whether a program is in an infinite loop. The halting problem is the problem of determining for a given program and input whether the program halts on the specified input.

One central result in computer science is that the halting problem is not computable. There is no algorithm that will decide for every program-input pair whether or not the program halts on the given input. More specifically, the halting problem is semi-decidable meaning that we are guaranteed to return the correct answer for all *yes* instances, but we might run forever if the answer is *no*. We can write a program that will eventually return *yes* for all *yes* instances of the halting problem by simply running the specified program on the given input until it finishes. It will run forever whenever the answer is *no*. A problem is non semi-decidable if we cannot even correctly compute all *yes* instances.

The argument that the halting problem is semi-decidable is simple and straightforward. It is a bit trickier to prove that the halting problem is not computable, which involves proving that we cannot write an algorithm that will return the correct answer for all *no* instances. Basically we assume that there is some program h that will solve the halting problem. Now consider the program $f(i, x)$ which returns 1 if $h(i, x) = 0$ and goes into an infinite loop otherwise. We can write this program because we have assumed h is computable. It has been proven that we can enumerate all programs, so we can assign a unique numeric value to each program. Let u be the value of program f. Now

consider what happens when we look at $h(u, x)$. If $h(u, x) = 1$ then it means that the program f halts on x. For f to halt on x it means that $h(u, x) = 0$. Thus $h(u, x) = 1$ implies $h(u, x) = 0$. Contrarily if $h(u, x) = 0$ then $f(u, x)$ will halt and return 1, but this would mean that $h(u, x) = 1$ since f halts on x. Thus $h(u, x) = 1$ implies $h(u, x) = 0$. We have reached a contradiction so there must not be a program that will compute the halting problem.

One of the most common methods of showing that a problem is not computable is to show that it is equivalent to the halting problem or that the halting problem can be reduced to it.

We have described computability because there are versions of the problems that we consider that are not decidable. This will be explained in more detail later.

2.3. Planning Problems

Though planning often seems like a simple and direct task, the problem is computationally intractable. There are planning problems, such as the Towers of Hanoi (Lucas 1883), whose shortest plan is exponential in length. In general, planning is PSPACE-complete, but often people consider restrictions on planning that make the problem slightly more tractable. The most common restriction is to consider problems where there is some polynomial bound on the length of the plan. Under this restriction, the question of whether or not there exists a plan is in NP.

There are many variations on planning problems, how they are expressed, and how people solve them. This section is divided into subsections to discuss the different

variations and solutions. The first subsection describes STRIPS (Fikes and Nilsson 1971), the most popular representation for generic planning problems. We then describe a few approaches to solving problems expressed in STRIPS notation. Following that, we will define what an optimal solution is and approaches to finding optimal solutions. The final subsection contains a description of an alternate representation.

2.3.1. STRIPS

One of the most popular ways of representing planning problems is to use propositional STRIPS (Fikes and Nilsson 1971) notation. In STRIPS notation, a planning problem consists of four elements. The first is a set of Boolean variables that represent various conditions about the world. For example, suppose we are developing a plan for how to get from home to work. This may depend on the weather or if we are running late so we may choose to include Boolean variables to represent various aspects of the environment such as *raining* to indicate whether or not it is raining or *late* to indicate whether or not we are running late. The set of variables can also be a set of predicates which map from some domain to $\{true, false\}$.

The second element of a planning problem is a set of actions. An action is typically represented as $preconditions \rightarrow postconditions$. The preconditions and postconditions are both sets of literals (a variable or its negation) where the preconditions must be true before the action can be executed and the postconditions are true after the action has been performed. For example, in the case where we are developing a plan to get to work, we may have an action that says $raining \rightarrow drive \vee takebus$.

The last two elements of a planning problem are the initial state and the goal state. The initial state is the set of conditions that are true at the beginning of the problem. The goal state is the set of conditions that we wish to satisfy. A solution to a planning problem (a plan) is a set of actions that take us from the initial state to the goal state.

For example, suppose we wish to move a set of packages from their original cities to their destinations by air. One way to represent this problem is to start by introducing a set of constants that will form the domains for our variables. The constants are planes p_1, …, p_k, packages (or objects) o_1, … o_m, and cities c_1, … c_n. The variables that we will need include the predicates *inPlane* (*plane*, *object*) to indicate that the given object is on a specific plane, *atCity* (*plane*, *city*) to indicate that the given plane is at a particular city, and *locatedAt* (*object, city*) to indicate that the given object is not on a plane but located at the specified city.

The primary actions that we need for the planning problem are fly, load, and unload, as well as a no-op action which does nothing. The first action is *fly* (*plane*, *city1*, *city2*), meaning that we are flying the specified plane from *city1* to *city2*. In order to perform this action, the only precondition is that the plane must be located at *city1*. After the action, the two postconditions that must hold are that *plane* must be at *city2* and it must no longer be at *city1*. So the action can be represented as *atCity* (*plane*, *city1*) → *atCity* (*plane*, *city2*) ∧ ¬*atCity* (*plane*, *city2*). We will need additional actions for loading and unloading. We won't provide full details here, but the remaining actions are fairly straightforward to express.

Finally, the start state encodes the initial location of all packages and planes, and the goal state contains the final location indicating where each package needs to end up.

2.3.2. Planning as Satisfiability

Originally planning problems were solved using logical deduction. Kautz and Selman (Kautz and Selman 1992) proposed an alternative approach that involves representing planning problems as satisfiability problems and using a SAT solver.

In SAT problems, each variable is set to a distinct value and does not change. In planning problems, the value of each variable may change throughout the duration of the problem. Thus, we need to mark each of our variables with a timestamp. Unfortunately this requires an upper bound on the plan length which may not be known in advance. We can start with some initial upper bound and increase the upper bound if no plan is found. If there is no solution to a problem, then this procedure will run forever.

Let S represent the set of all initial conditions and let G be the goal conditions. The initial conditions can be specified by including one clause for each literal in S. We also need one clause for the negation of each positive literal not in S. The clause variables used to specify the initial state are all marked with timestamp one. The goal state can be specified by taking adding one clause for each literal in G. The timestamp on these variables will be the upper bound on the plan length.

Next, we need to encode the actions. Each action a_i is of the form $\{p_1, \ldots, p_n\} \rightarrow \{q_1, \ldots, q_m\}$ where $p_1, \ldots, p_n$ are the preconditions and $q_1, \ldots, q_m$ are the postconditions. Let $a_i(k)$ mean that action a_i is executed at time k. Similarly for each p_j or q_l, let $p_j(k)$ or

$q_l(k)$ mean that the variable is true at time k. We must ensure that if an action occurs at time k then its preconditions are true at time k and its postconditions are true at time k+1. For every time step k we introduce the following clauses:

$$\forall i,j\ (\neg a_i(k) \vee p_j(k))$$
$$\forall i,l\ (\neg a_i(k) \vee q_l(k+1))$$

While these clauses are all true, they are insufficient to encode a planning problem. If we only encode the initial state, the goal state, and the actions then we will admit many anomalous models. Specifically, in some models the world may change even if no action has occurred. To prevent this, frame axioms are used to encode the implicit idea that a literal remains unchanged unless it is an effect of the current action. Let A_l be the disjunction of all actions that contain literal l as a postcondition. When we are looking at a specific time k we use $A_l(k)$. For each time step, frame axioms are encoded using the following clauses:

$$\forall l\ (l(k) \vee \neg l(k+1) \vee A_l(k))$$

In addition to frame axioms, the authors of this work introduced exclusion axioms to ensure that only one action occurs at any time step. Modern solvers often allow for nonconflicting actions to occur simultaneously, but the original planning as satisfiability approach did not. These axioms are simple and straightforward. For each time step k, the following clauses are included:

$$\forall i \neq j\ (\neg a_i(k) \vee \neg a_j(k))$$

The planning as satisfiability approach worked well in spite of the fact that this requires searching an exponentially large search space of models of a SAT formula rather than the logical deduction method of searching the space of resolution proofs.

2.3.3. Graphplan

Another way of developing a solution to a planning problem is by generating and analyzing a *plan graph* (Blum and Langford 1997). A plan graph is a visual representation of the planning problem that can be constructed as follows: create a set of nodes to represent the initial conditions with one node for each condition. Next create a set of nodes to represent all possible actions that may be taken including no-ops. We connect the preconditions of an action to the node representing that action. Then we create a set of nodes representing all of the possible postconditions and we connect each action to its corresponding postconditions. We repeat this process either up to some fixed length k (to bound the plan length to k) or until two successive sets of conditions are identical. The layers of a plan graph are the conditions true at time one, the actions possible at time one, the conditions possible at time two, the actions possible at time two, etc.

An example of the first four layers of a plan graph is shown in figure 2.3.1. This example is based on the problem that was described earlier where we have a set of packages, a set of cities, and a set of planes. The goal is to fly all of the packages from their current location to their destination. In this particular example we have one package

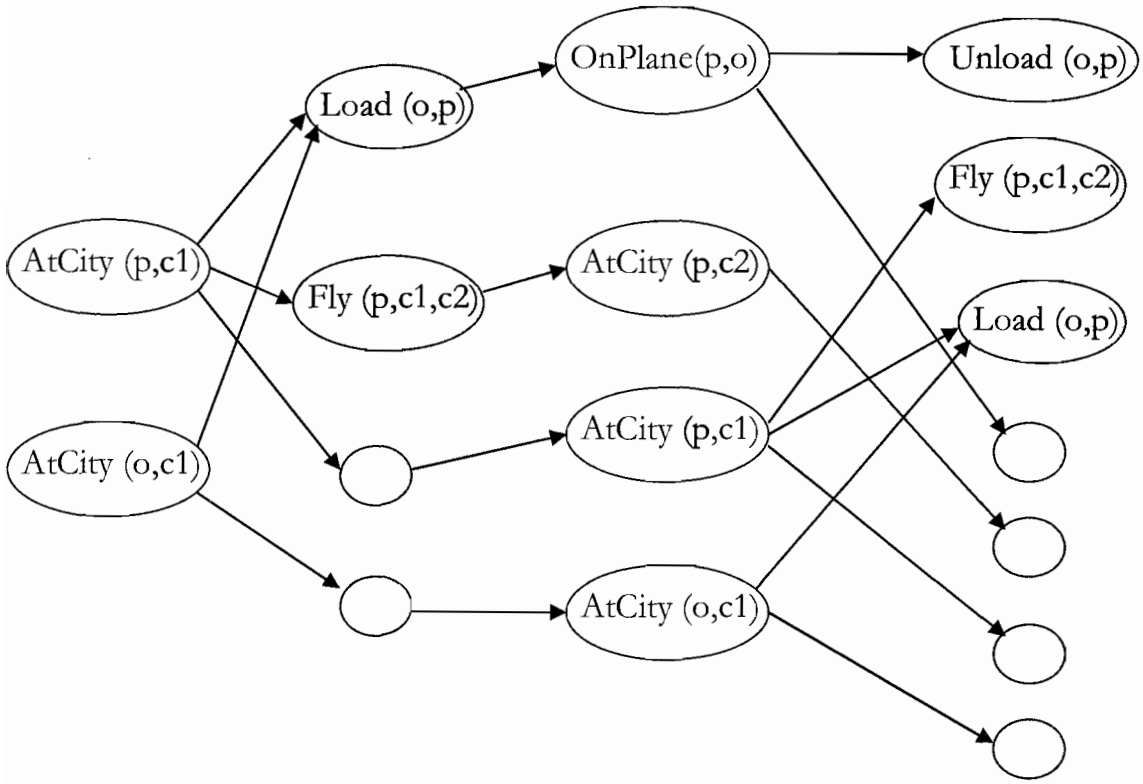

Figure 2.3.1. Example of a plan graph. Action nodes are tinted gray and nodes representing conditions are in white. For simplicity, only positive literals are shown in the figure, but a real plan graph will also include negative literals.

(o), one plane (p), and two cities (c_1 and c_2). The plane and package are both located at c_1 and the goal is to get the package to c_2.

After the plan graph has been constructed, the next step is to identify mutual exclusion relationships among the nodes at each layer. Graphplan goes through the layers in order marking nodes as being mutually exclusive using a few simple rules. The rules do not capture all mutual exclusions, but many of them. Two actions are marked as exclusive if either 1) one deletes a precondition or postcondition of the other action or 2) the preconditions of the actions are marked as exclusive in the previous layer. To

determine if two variables are mutually exclusive, we only look at the actions in the previous layer. Two variables are marked as exclusive if all of the actions that generate one are exclusive of the actions that generate the other.

To start with the algorithm only generates the first layer of the plan graph. At each iteration, the first step is to extend the plan graph by adding all action nodes whose preconditions are in the previous layer and the preconditions are not marked as mutually exclusive. It then extends the plan graph one layer further by adding all of the postconditions of the action nodes that were just added. It marks two nodes in this new layer as mutually exclusive if all of the ways of generating one are mutually exclusive of all of the ways of generating the other.

After the plan graph has been extended, the algorithm searches for a valid plan using a backwards-chaining technique. If the goal conditions do not appear in the last layer of the plan graph it moves on to the next iteration. Given a set of goals to be achieved at time k it attempts to find a set of actions at time k-1 having the goals as postconditions. The preconditions of these actions form a set of subgoals to be satisfied at time k-1. If the set of subgoals at time k-1 cannot be satisfied then the algorithm will try to find another set of actions at time k-1 that have the goals as postconditions. It keeps trying different sets of actions either until it finds a plan or it determines that none can be found.

When the next layer of the plan graph is generated, if it is identical to the previous layer, we can stop and say that no plan can be found. This procedure always halts and

will return a plan if one can be found and return false otherwise. Graphplan performed very well at the time it was introduced in comparison to other complete methods.

2.3.4. Satplan

One of the more well known methods of solving planning problems today is called Satplan (Kautz et al. 2006) which combines the idea of representing planning as SAT with the idea of using a plan graph. The first step is to generate the plan graph up to length k where k is initially one. Next, Satplan converts the plan graph into a SAT formula and calls a SAT solver. If the formula is satisfiable then the algorithm returns the corresponding plan. Otherwise, the system increments k and tries again until k has reached the maximum limit. Satplan uses the mutual exclusion rules from Graphplan in order to restrict the search space.

Satplan won first place in the 4th International Planning Competition and tied for first in the 5th International Planning Competition. Many planners are based on the Satplan algorithm.

2.3.5. Optimal Planning

A variation of planning problems is to look for an optimal plan. This typically means a plan of minimum length. This increases the complexity of planning from NP to $F\Theta_2$. In other words, it is $F\Theta_2$ complete to determine the length of the minimum plan or to find a minimum length plan. We will prove these results later. A consequence of this is that all of the methods mentioned thus far are actually solving a harder problem than

straight-forward planning, since they automatically find a plan of minimum length by checking all possible plan lengths in increasing order.

One of the main problems with solving optimal planning using Satplan is that incrementing k at each iteration causes slow progress through the search space and results in a large number of SAT calls. Another problem is that making multiple calls to a SAT solver discards learned information from the prior iterations. Learning is one technique that was added to DPLL. Each time that we reach a dead end in searching for a SAT assignment, we can use resolution to derive a new clause that helps to identify why we reached that dead end. Adding this new clause to the formula keeps us from unnecessarily exploring certain regions of the search space.

Learning is one of the major reasons for the improved efficiency of modern SAT solvers. Real world problems have structure that learning techniques are able to exploit. Throwing away learned clauses is particularly detrimental in the planning domain because there is so much similarity between the individual calls that Satplan is making. The information gained in searching for a plan of length k will also be useful in searching for a plan of length k+1.

There has been prior work done on reducing the number of SAT calls by increasing k multiplicatively rather than incrementally (Xing et al. 2006). However, once a plan is found, it is not guaranteed to be optimal so the algorithm needs to search smaller values of k to find the optimal plan. There has also been prior work done on retaining learned clauses (Nabeshima et al. 2006). The authors modified the algorithm to analyze

the learned clauses to determine which ones were most likely to be useful and keep them around for the next iteration.

We will later demonstrate a better method of solving optimal planning that arises from showing that DPLL is Δ_2 capable. This method is based on the idea that we can use a single call to a SAT solver, which automatically eliminates the problem of throwing away learned information.

2.3.6. Planning with Preferences

Another variation of planning problems is planning with preferences where we can express preferences for some models over others. Pontelli and Tran (2004) presented a language for expressing preferences in planning problems. In their work, they define a plan as a trajectory $s_0a_1s_1 \ldots a_ms_m$ where s_m entails the goal. They define three types of preferences: basic desires, atomic preferences, and general preferences.

The simplest type of preference is a basic desire. In order to define what these are, we must first define a few other concepts. A ***state desire*** φ means we prefer state s such that $s \vDash \varphi$. A state desire can also be used to express a preference for using a certain action. This is denoted ***occ*** (a) meaning we prefer to use action a. The temporal connective **next** (φ) means that φ is entailed by the action a_i or the state s_i. They use **always** (φ) to mean that φ is entailed by every action and every state. The connective **eventually** (φ) means that there is some action or state that entails φ. They define **until**

(φ_1, φ_2) to mean that there is a state or an action that entails φ_2 and that every state and action up to that point entails φ_1.

We are now ready for the definition of ***basic desires***. State desires and goal preferences are both basic desires. If φ_1 and φ_2 represent basic desires then a basic desire can also be of the following forms: $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$, next (φ_1), until (φ_1, φ_2), always (φ_1), and eventually (φ_1). If α and β are trajectories (plans) then α is preferred over β with respect to preference φ if α satisfies φ and β does not. This is denoted $\alpha \prec_\varphi \beta$. If α is indistinguishable from β with respect to φ we write $\alpha \approx_\varphi \beta$.

The next type of preferences, which are a level up from basic desires, are ***atomic preferences***. If we have a set of basic desires $\varphi_1, \varphi_2, \ldots, \varphi_n$ then an atomic preference is expressed as $\varphi_1 \lhd \varphi_2 \lhd \ldots \lhd \varphi_n$ and indicates an ordering on the basic desires. This allows us to prefer some basic desires over others. Let α and β be trajectories and $\Psi = \varphi_1 \lhd \varphi_2 \lhd \ldots \lhd \varphi_n$ be an atomic preference formula. We say that α is equivalent to β with respect to Ψ (denoted $\alpha \approx_\Psi \beta$) if $\forall i\ (1 \leq i \leq n)\ \alpha \approx_{\varphi_i} \beta$. In words, this says that the two trajectories are equivalent with respect to the atomic preference if they are equivalent with respect to the basic preferences. We say that trajectory α is preferred over β (denoted $\alpha \prec_\Psi \beta$) if $\exists i\ (1 \leq i \leq n)$ such that both of the following conditions hold: 1) $\forall j$ $(1 \leq j \leq i)\ \alpha \approx_{\varphi_j} \beta$ and 2) $\alpha \prec_{\varphi_i} \beta$. In words, α is preferred over β if there is some basic

preference φ in the formula such that α satisfies φ and β does not, and on all preferences that are more important than φ, α and β must be indistinguishable.

The last type of preference that the authors define is general preferences. General preferences can be defined by the following set of rules:

- An atomic preference is a general preference.
- If Ψ_1 and Ψ_2 are general preferences then so are Ψ_1 & Ψ_2, Ψ_1 | Ψ_2, and !Ψ_1.
- Given a collection of general preferences Ψ_1, Ψ_2, …, Ψn then $\Psi = \Psi_1 \lhd \Psi_2 \lhd \ldots \lhd \Psi n$ is a general preference.

The operators &, |, and ! essentially represent and, or, and not but there is a subtle distinction between $\varphi \vee \phi$ and $\varphi \mid \phi$. The first is a single preference or criteria whereas the second represents two criteria with no preference between them.

Now they define an ordering on trajectories based on general preferences by considering each type of general preference. Given trajectories α and β and a general preference Ψ, $\alpha \prec_\Psi \beta$ if:

- Ψ is atomic and $\alpha \prec_\Psi \beta$.
- $\Psi = \Psi_1$ & Ψ_2 and $\alpha \prec_{\Psi_1} \beta$ and $\alpha \prec_{\Psi_2} \beta$.
- $\Psi = \Psi_1$ | Ψ_2 and one of the following three conditions holds: 1) $\alpha \prec_{\Psi_1} \beta$ and $\alpha \approx_{\Psi_2} \beta$, 2) $\alpha \prec_{\Psi_2} \beta$ and $\alpha \approx_{\Psi_1} \beta$, or 3) $\alpha \prec_{\Psi_1} \beta$ and $\alpha \prec_{\Psi_2} \beta$. This says that either α is preferred with respect to both preferences or α is preferred on one preference and they are equivalent on the other.

- $\Psi = !\Psi_1$ and either $\beta \prec_{\Psi_1} \alpha$ or $\alpha \approx_{\Psi_1} \beta$.
- $\Psi = \Psi_1 \lhd \Psi_2 \lhd \ldots \lhd \Psi n$ and $\exists i \; \forall j \; \alpha \prec_{\Psi_i} \beta$ and $\alpha \approx_{\Psi_j} \beta$. This says that there is some point i such that the two trajectories are equivalent on the first i-1 preferences but that α is preferred on the i^{th} preference.

This formal representation of preferences will be used later in discussing the variations of planning with preferences that can be solved with DPLL and those that cannot. Specifically, we can characterize the type of preferences that DPLL is capable of solving. We do this by demonstrating various complexity results for planning with preferences using the notation and types of preferences given by Pontelli and Tran (2004).

2.4. Nonmonotonic Reasoning

In order to teach a computer to "think" more intelligently, it must have mathematically precise ways of representing the information that it knows and ways of reasoning about that information in order to derive new conclusions. Typically a program will have a knowledge base of information and a set of facts that may be derived from the knowledge base.

In classical logic, if a statement can be entailed from a set of facts A, then it can also be entailed from any larger set of facts A $\cup$ B. The set of inferences that may be drawn grows monotonically with the size of the knowledge base. In many real-world applications the addition of new facts may negate previously drawn conclusions, making

it necessary to retract those conclusions. In what is called nonmonotonic reasoning (NMR), adding new information to the knowledge base may invalidate some previously derivable inferences.

Nonmonotonic reasoning is inherently more complex than its classical counterpart. In first-order logic, entailment is only semi-decidable, which means it is equivalent to solving the halting problem. In monotonic reasoning, we can systematically derive inferences from our knowledge base in such a way that given an unlimited amount of time we will enumerate all possible inferences. If a statement is entailed then we will eventually halt and say *yes*, but we may continue running until the end of time if the answer is *no*. NMR is not even semi-decidable because determining if a statement can be inferred involves checking the consistency of the statement with the knowledge base. In order to check the consistency of a statement we must show that the negation is not entailed. If the negation can be derived, we can halt and say that the statement is not consistent, but yes instances of consistency checking may run forever.

In order to make the problem computable, it is typical to restrict the problem to propositional logic, which only allows a finite number of predicates and finite domains for predicates. These predicate logic problems, such as satisfiability, are generally NP-complete. The addition of consistency checking increases the complexity of the problem. Usually if a problem is in the polynomial hierarchy, adding nonmonotonicity increases the complexity by one level within the polynomial hierarchy. So if the original problem is in Σ_1 then adding nonmonotonicity will make the problem in Σ_2.

There are many different formalizations of nonmonotonic reasoning. Here, we will only describe the most popular and well known ones. These include default logic, circumscription, modal nonmonotonic logic, answer-set programming, and preferred-model semantics. The remainder of this section is divided into subsections to cover each of these formal methods.

2.4.1. Default Logic

One of the first types of nonmonotonic reasoning was introduced in a paper by Raymond Reiter, describing what he called *default logic* (1980). Reiter modified classical logic by adding a nonmonotonic construct shown in formula 2.4.1. This is logically equivalent to "if α is believed to be true and if β is consistent with what we believe is true then assume that *w* is true." This formula expresses what is known as a *default rule*. *M* is an operator that should be interpreted as "it is consistent that."

$$\frac{\alpha : M\beta}{w} \qquad (2.4.1)$$

A theory <D, W> in default logic would consist of a set of default rules D and a set of known sentences W. Default logic dictates that if we can apply a default rule we must. In other words if α is in the knowledge base and β is consistent, then we are *required* to add *w* to the knowledge base. By continuing to add sentences until we cannot

apply any more default rules we create an ***extension*** of the theory. An extension is a set of beliefs about the world that is deductively closed.

There will not always be a consistent extension. If we have a default of the form "if β is consistent then conclude $\neg\beta$" there will be no extensions. It would be silly to write a default of this form, but we can imagine more complicated scenarios which lead to the same result. If W is inconsistent in the default theory <D, W> then we may have an inconsistent extension. We can also have inconsistent extensions that arise from not checking the consistency of a statement before it is assumed.

Extensions are not always unique. It is possible to write default rules that conflict such as "assume a person lives in the same town where they work" and "assume a person lives in the same town where their spouse lives." In many cases these two defaults will lead to the same result. However, there are cases where a persons spouse may live in a different city than where he/she works. There are two different extensions here: one in which the person lives in the same city as his/her spouse and one in which the person lives in the same city that he/she works. In instances such as these default theories can have multiple extensions.

Adding a set of defaults adds nonmonotonicity because there are cases where one default theory is a subset of another default theory, but their extensions may not coincide. We may have a default theory <D, W> with extension E, a set of defaults D' and a set of sentences W' such that $<D \cup D', W \cup W'>$ has no extension E' where $E \subseteq E'$. For example, suppose that we have a theory with the following two defaults:

$$d_1 = \frac{bird(x) : M fly(x)}{fly(x)} \qquad\qquad d_2 = \frac{penguin(x)}{\neg fly(x)}$$

The first states that if x is a bird and it is possible that x flies then we conclude $fly(x)$. The second states that if x is a penguin then x does not fly. Consider the world in which we only know $bird(Tweety)$. Given our set of defaults and our known information about the world, we conclude that *Tweety* flies. The only extension of this theory is E = $\{fly(Tweety), bird(Tweety)\}$. Suppose instead we have a world in which we know $\{bird(Tweety), penguin(Tweety)\}$. This original default theory is a proper subset of this one. However, there is no extension of this theory in which Tweety flies, so there is no extension E' such that E $\subseteq$ E'.

Reiter established a method for finding a consistent extension (if one exists) for what he calls a ***normal default theory***. A default is called a ***normal default*** if it is of the form shown in figure 2.4.1, but with $\beta = w$. In a normal default theory, all of the default rules are normal defaults. Reiter showed that a normal default theory always has at least one consistent extension and he described a method for finding the extensions of a normal default theory on closed ***well-formed formulas*** (***wffs***). A wff is a symbol or string of symbols that is generated by the formal grammar of a formal language. The method for finding an extension operates on a set of wffs, *W*, and a set of normal defaults, *D*. The extension *E* is initially equal to *W*. The method then proceeds by selecting some default in *D* that can be applied and adding the consequence *w* to E. When none of the defaults in *D* can be applied then the procedure is complete. This method will always terminate and

produce a valid extension for any normal default theory. While Reiter's work is very mathematically elegant, other formalisms are more often used in practice.

2.4.2. Stable Models or Answer Set Programming

Gelfond and Lifschitz (1988) proposed another formalism called *stable model semantics* which is one of the most popular formalizations today. For additional information on this method see two later papers written by Lifschitz (2002, 2005). There are many implementations of it, but the foundation for implementing stable model semantics is called *answer set programming* (Marek and Truszczynski 1999, Niemala 1999).

The main idea is to have a set of rules where each rule is of the form shown in formula 2.4.2.

$$A_0 \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \qquad (2.4.2)$$

The left side of the rule is the head and the right side is the body. We want to find a *stable model* or *answer set* that satisfies a list of rules. The idea is similar to default logic in that if $A_1, \ldots, A_m$ are in the answer set and $A_{m+1}, \ldots, A_n$ are not in the answer set then we must add A_0 to the answer set. Each of $A_0, \ldots, A_n$ is a single atom. A comma in the rule represents $\wedge$ and a semicolon represents $\vee$. As with default logic, there may be zero, one, or multiple answer sets for any given set of rules.

To make the formulation more compact, Gelfond and Lifschitz also added *choice* rules and cardinality constraints. Choice rules are of the form p; not p which means that we can choose whether or not p is in the answer set. Note that an answer set may be only a partial assignment of variables so the answer set need not contain either p or $\neg p$. Cardinality constraints state that at least l and at most u of $F_1, \ldots, F_n$ must be true. These rules are specified as shown in formula 2.4.3.

$$l \leq \{F_1, \ldots, F_n\} \leq u \tag{2.4.3}$$

Stable model semantics are similar to default logic in that we have a knowledge base and a set of rules (or defaults) for deriving new information. The rules in answer set programming and the defaults in default logic are both of the form where we have conditions for applying them and consequences which result from the application. Also in both cases we are required to apply a rule or default whenever it is possible to do so.

Though the stable models method bears some resemblance to default logic there are some very important differences. The most significant is that a stable model is a set of atoms whereas an extension in default logic is a set of sentences. Similarly, defaults are made up of sentences whereas rules in a program are made up of atoms. There is nothing similar to choice rules or cardinality constraints in default logic. Answer set programming is nonmonotonic for the same reason that default logic is. There may be a stable model such that if we add new rules to the program or new atoms to the set of

atoms that are believed then we may have to retract inferences that we were previously able to make.

2.4.3. A Semantical Approach

Yoav Shoham (1987) proposed an approach to nonmonotonic reasoning that involves finding a most preferred model of a theory given a partial order on the interpretations. His approach involves defining a preference order on the interpretations of a theory so that some models are preferred over others. He uses the notation $B \sqsubset A$ to mean that A is preferred over B or equivalently B is less preferred than A.

A different notion of satisfiability comes into play when a partial order on interpretations is introduced. Shoham defines *preferential satisfiability* as a model M preferentially satisfies A (written $M \models_{\sqsubset} A$) if $M \models A$, and $\nexists M'$ such that $M' \models A$ and $M \sqsubset M'$. In other words, M preferentially satisfies A if it satisfies A and it is equal to or preferred over other models that also satisfy A. The models that preferentially satisfy a theory are called preferred models. He defines *preferential entailment* as A preferentially entails B (written $A \models_{\sqsubset} B$) if all of the preferred models of A are also models (preferred or otherwise) of B.

In classical logic if $A \models C$ then $A \cup B \models C$. The addition of a partial order on interpretations introduces nonmonotonicity because $A \models_{\sqsubset} C$ need not imply $A \cup B \models_{\sqsubset} C$.

2.4.4. Additional Frameworks

There are other formalizations of nonmonotonic reasoning including circumscription (McCarthy 1980), modal nonmonotonic logic (McDermott and Doyle 1980), and autoepistemic logic (Moore 1985). We will not include further details on these formalizations here since they will not be talked about later in this thesis, but the interested reader can read the corresponding papers on these topics for further information.

The formal methods that we have mentioned are the most well known formalisms of nonmonotonic reasoning. Others have been developed, but the complexity of nonmonotonic reasoning has led several people to opt for ad-hoc techniques added on top of monotonic frameworks wherever nonmonotonic constructs are useful. However, research is still being done on improving the algorithms for the more formal theories of nonmonotonic reasoning.

2.4.5. Computational Complexity and Implementations

The complexity of default logic and the complexity of answer set programming are higher than that of satisfiability. Satisfiability is well known to be NP complete. (Cook 1971) It is D^P to determine whether a given logic program has an answer set (Eiter et al. 2004). Recall that D^P is the set of problems that can be solved in polynomial time where we are allowed a single call to an NP oracle and a single call to a coNP oracle. It is Σ_2 complete to determine if a given logic program has an answer set, or to determine if a set of ground terms belongs to an answer set (*brave reasoning*) (Eiter et al. 2004). It is Π_2

complete to determine if a ground term belongs to all answer sets (*cautious reasoning*) (Eiter et al. 2004). Brave reasoning and testing for extensions of a default theory are Σ_2 complete while cautious reasoning is Π_2 complete for default logic (Gottlob 1992).

Despite the increased complexity, many attempts at developing answer set solvers have used the DPLL algorithm for solving SAT. One such solver is called Smodels (Simons 2000). The format of the algorithm for solving Smodels is similar to the DPLL algorithm except that the unit-propagation procedure is modified to find the smallest deductively closed set of a program given the current set of atoms, and the recursive step checks to see if there is an answer set both with the chosen literal included and with it not included in the answer set. GnT (Janhunen and Niemala 2004) is similar to Smodels but was rewritten to include disjunctions in the rule heads.

Another SAT-based solver for answer-set programming is Cmodels (Giunchiglia et al. 2004). Cmodels' first step is to produce the programs completion. Next, it uses a SAT solver to find a model. Finally it checks if the model found is an answer set. If it is, then the model is returned, otherwise the SAT solver is used to find the next model.

ASSAT (Lin and Zhao 2004) is another solver similar to Cmodels. It checks for loops in the completion of a logic program, since loops are what prevent models from being answer sets. When a loop is found, clauses can be added that eliminate the corresponding models. Unfortunately, there may be an exponential number of loops, so preprocessing and running a single execution of a SAT solver may require exponential time and space. Instead they run a SAT solver to find a model. If there are no models then it returns failure. If a model is returned and it is an answer set then it is returned. If the

model is not an answer set, then it finds a loop in polynomial time that causes the model to be invalid, adds the necessary clauses, and restarts the SAT solver.

DLV is another SAT-based solver that generates a model using a SAT solver and tests if it is an answer set (Leone et al. 2006). If it is then the model is returned, otherwise it searches for the next model.

These methods rely on multiple calls to a SAT solver. Most of the problems are beyond the scope of what DPLL can solve, so it is natural to have to make multiple calls. We will later analyze, based on complexity, what versions of nonmonotonic reasoning can be solved with DPLL.

CHAPTER III

ALGORITHM CAPABILITY

One of the main contributions of this thesis is the notion of algorithm capability. Good algorithms are often capable of solving a wide variety of problems, but there are limitations on the scope of what an algorithm can solve. To use an algorithm to solve a new problem we must have some sort of reduction from the new problem to the algorithm. If we do not place any limitations on the complexity of these reductions then we can use an algorithm to solve any computable problem by solving the problem and then hand selecting the algorithm input accordingly. For capability to have any meaning, we must place some restrictions on the complexity of the reductions. Informally, we say that an algorithm is capable of solving a problem if the reduction can be done efficiently (typically in polynomial time). In this chapter we will formally define the capability of an algorithm in terms of the complexity of problems that it can solve using efficient transformations of the input and output.

Recall that algorithms can be mathematically expressed as Turing Machines. A Turing Machine has two possible ways of producing output. One of which is to accept or reject the string on the tape. The other is to return the string that is left on the work tape (or on a separate output tape) when the machine enters an accepting state. Both of these methods are acceptable and will not affect the definition of capability.

Let $A(w)$ denote the output of executing algorithm A on input w. For example, DPLL(w) is a satisfying assignment of the boolean formula represented by w if such an assignment exists, and UNSAT otherwise. To reduce from a problem to an algorithm we want transformations t_1 and t_2 where t_1 maps all possible strings into an input for A and t_2 maps all possible outputs of A into the correct solution for the problem.

We can say informally that an algorithm is C,F-capable where C is a complexity class and F is a class of functions (eg. polynomial-time or log-space), if it can solve all problems in C using F-Computable transformations of the input and output. If F is unspecified, then we assume polynomial-time transformations, so C-capable is shorthand for C,P-Capable. For deterministic algorithms we can easily formalize this notion as follows:

Definition 3.1. A deterministic algorithm A is C,F-capable if $\forall\ L \in C$, $\exists\ t_1: \Sigma^* \rightarrow \Sigma^*$, $t_2: \Sigma^* \times \Sigma^* \rightarrow \{0, 1\}$ where t_1 and t_2 are F-computable such that $w \in L$ iff $t_2(w, A(t_1(w)))$ is true.

The definition that we use for reducing between problems and algorithms is very similar to the definition for metric reductions. Essentially this definition says that an algorithm is capable of solving a problem if there are efficient transformations t_1 and t_2 where t_1 transforms an instance of the problem into an input for the algorithm, and t_2 transforms the output of the algorithm into a correct solution for the problem. An

algorithm is capable of solving all problems in a complexity class if there exist such transformations for all problems in that class.

It is a bit trickier to define capability for nondeterministic algorithms which may have several execution paths running in parallel, some of which may not halt. However, we can definitively answer *yes* or *no* for every computable problem. This implies that every computable problem can be solved by an algorithm that ends in either an accepting or a rejecting state for all execution paths. Our definition of capability will only apply to such algorithms. While this does impose some limit on the applicability of our definition, it is not overly restrictive because it is rare in the real world to want to reuse algorithms for incomputable problems or to solve computable problems with an algorithm that does not halt on all inputs. Essentially we are assuming that an algorithm for solving a problem also solves the complementary problem.

The output of a nondeterministic algorithm is also less clear than in a deterministic one. Algorithms can return the string on the tape when they enter an accepting state. A nondeterministic algorithm accepts its input if any path leads to an accepting state. When a nondeterministic algorithm returns a string, it returns the string on the tape when *any* execution path enters an accepting state. There may be multiple valid outputs because there may be multiple execution paths that end in accepting states. So $A(w)$ may represent many different values.

Computers are deterministic and have different ways of simulating nondeterministic algorithms. One of which is by running the execution paths in sequence

rather than in parallel. We may wish to choose the order in which a computer explores these execution paths because it may impact which solution is returned.

Without knowing what the execution paths are up front or even knowing how many there might be, it seems an impossible task to order them. However, we can impose an order on the execution paths by ordering the nondeterministic branches of the NDTM, which makes the machine deterministic. Thus, to define capability for nondeterministic algorithms we can convert the nondeterministic algorithm into a deterministic one and then use our definition of capability for deterministic algorithms. This way, if an algorithm is C-capable then it is capable of solving all problems in C on a standard computer (since computers are deterministic).

In order to convert the algorithm into a deterministic one, we introduce one extra transformation in addition to t_1 and t_2. The new transformation, t_3, will take an algorithm input and return an ordering on the nondeterministic branches of the algorithm. We will use another algorithm, that we call *det*, that takes an algorithm and the branch order produced by t_3 and creates a deterministic version of the algorithm. Next we describe t_3 and *det* in more detail.

Recall that an NDTM is mathematically expressed as a 7-tuple $\{Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\}$. The transition relation δ relates elements of $Q\times\Gamma$ to elements of $Q\times\Gamma\times\{L, R\}$. (q_i, a) is related to (q_j, b, d) if when the machine is in q_i and the current symbol pointed at by the tape head is a, then the machine can legally transition into q_j, write the symbol b in place of a, and move the tape head in the direction indicated by d. To order the execution paths we want to place a partial order on the elements of this

relation. For instance, suppose that from our initial state on the symbol 0 we could either transition into state q_2, write a 1, and move the tape head to the right or transition into state q_5, write a 0 and move the tape head to the right. We could order them so that $((q_0, 0), (q_2, 1, R)) < ((q_0, 0), (q_5, 0, R))$. It is important that this order be transitive to make any sense. The function t_3 will take an algorithm input and produce sorted lists of transitions.

Essentially what the order means is that whenever there is more than one possible transition, we may have a preference on which one we select. We say that a path satisfies a preference when it follows the preferred transition and that it breaks the preference otherwise. We will probably be required to break some preferences in order to find an accepting path. Given an ordering on the possible transitions we wish to find the output of the most preferred accepting path.

When two paths break different sets of preferences, the path that breaks an earlier preference is less preferred than the path that does not. In other words, suppose that we have two paths P_1 and P_2 that break different preferences. In order to determine which path is preferred, we look at the time step in which these paths diverge. Since the paths diverge, they must be taking different transitions. Whichever path takes the more preferred transition at this step is chosen over the other path. The most preferred path is an accepting path such that no other accepting path is more preferred.

To compute the output of the most preferred path, we use *det* to create a deterministic algorithm A' from our nondeterministic algorithm A and the sorted lists of

elements in δ. Since A may have multiple valid outputs, the purpose of A' is to compute one specific output of A.

The algorithm *det* can most clearly be described in terms of its output, which is shown in figure 3.1. For the sake of clarity, we describe A' using informal pseudo-code rather than as a deterministic TM. A' is going to be a recursive algorithm that takes three inputs: the current state, the contents of the tape, and the tape head position. It either returns an output or REJECT. The initial input to A' is $(q_0, w, 0)$. This process depends on the partial order on branch decisions that will be created. The ordering will be represented by sorted lists of transitions for each (state, symbol) pair.

```
1:  A' (q_current, tape, tapehead)
2:    sorted_transitions = transitions [q_current, tape[tapehead]]
3:    newtape = copy (tape)
4:    result = REJECT
5:    while result = REJECT && sorted_transitions is not empty do
6:      (newstate, newsym, dir) = pop (sorted_transitions)
7:      newtape[tapehead] = newsym
8:      if newstate = q_accept then return newtape
9:      if newstate = q_reject then result = REJECT
10:     else result = A' (newstate, newtape, tapehead+dir)
11:   return result
```

Figure 3.1 Pseudo-code for A'.

The idea is that we first try making a recursive call where we use the most preferred transition. If this results in REJECT then we try the next most preferred

transition and so on until we either find an accepting path or run out of transitions. If we find an accepting path then we return its output.

Now that A' is deterministic we can use the same definition of capability that was given earlier. So to define capability for nondeterministic algorithms, we just incorporate the creation of A' into the definition of capability. The definition becomes:

Definition 3.2. A nondeterministic algorithm A is C,F-Capable if $\forall\ L \in C$,

$\exists\ t_1: \Sigma^* \rightarrow \Sigma^*, t_2: \Sigma^* \times \Sigma^* \rightarrow \{0, 1\}, t_3: \Sigma^* \rightarrow \Sigma^*$ which are F-Computable such that

$w \in L$ iff $t_2(w, det\ (A, t_3(t_1(w)))\ (t_1(w)))$ is true.

Thus t_1 and t_2 transform the input and output as before, but now we have an additional transformation t_3 which takes the input and creates an order on nodeterministic branches. We also have a separate function, *det*, which takes nondeterministic algorithm and the order on nondeterministic branches and creates a deterministic algorithm A'.

DPLL is trivially NP Capable. What is less intuitive is that NP is not an upper bound on the capability of DPLL. This is important because it highlights the difference between capability and complexity.

One difference is that an algorithm for a language can always be used to solve the complement of that language. In other words, C-capable is equivalent to coC-capable. The language of unsatisfiable formulae and the language of satisfiable formulae can be recognized by DPLL even though one is coNP-complete and the other is NP-

complete. This does not mean that NP = co-NP, but a polynomial transformation from unsatisfiable formulae to satisfiable ones would.

Another difference is that a transformation to instances of a problem implies a transformation to inputs of an algorithm solving the problem, but the reverse is not necessarily true. The algorithm must solve all possible inputs of the problem that it was designed for, but there may be more algorithm inputs than there are instances of the original problem.

Note that since DPLL requires exponential time, there is no obvious reason why it cannot be applied to problems of a much higher complexity. However, the algorithm only uses polynomial space so we cannot use it for problems beyond PSPACE. It is not immediately apparent why we cannot use DPLL to solve every problem in PSPACE, but we show in the next section that DPLL can only be used for problems in Δ_2.

In subsequent chapters we will show how the notion of capability is useful by showing the capability of two well known algorithms and how this information can be used to apply one of them to additional problems.

CHAPTER IV

EXAMPLES OF CAPABILITY

In this chapter we present two practical examples of the idea of algorithm capability. One is establishing the capability of the well known DPLL algorithm for solving Boolean satisfiability. In the first section, we prove that DPLL is Δ_2 capable. Though the original algorithm is the one that most modern solvers are based on, it has been significantly modified over the years to include a number of optimization techniques. In section two, we will discuss how these additional techniques affect the capability of DPLL. In the final section, we give an additional example, showing that the algorithm most commonly used for solving QBF is PSPACE capable.

4.1. Proving Δ_2-Capability of DPLL

Satisfiability is one of the most important and widely studied problems in computer science. Many different algorithms have been developed, but DPLL has so far proven to be the most effective. Section 2.1 gives more details on the satisfiability problem and the DPLL algorithm for solving it.

Researchers have applied a variety of optimization techniques in order to make the algorithm more efficient. The optimizations that have been used range from good coding and data structures to branching heuristics, learning methods, and parameter

tuning. There are many implementations of the DPLL algorithm. Some of the more popular ones include zChaff (Moskewicz et al. 2001), RSAT (Pipatsrisawat and Darwiche 2007), SATzilla (Nudelman et al. 2004), and Minisat (Eén and Sörensson 2004).

DPLL is known to work well for an assortment of different applications. Given its popularity and effectiveness, it is useful to characterize the set of problems that it can be used to solve. This way, when a new problem arises, we know whether or not we can apply DPLL based on the complexity of the problem.

In order to show that DPLL is at least Δ_2-capable, it is sufficient to show that it can be used to solve a Δ_2-complete problem. We can reduce all other problems in Δ_2 to any Δ_2-complete problem in polynomial time. So if we can reduce from any problem in Δ_2 to a Δ_2-complete problem and reduce from a Δ_2-complete problem into inputs for DPLL, then by combining the reductions we can solve every problem in Δ_2 with DPLL. The first step is to show that some problem is Δ_2-complete.

Many variations of satisfiability exist, some of which are not known to be in NP. One such problem is determining whether the lexicographically maximum satisfying assignment is odd. Let $\psi(x)$ be a Boolean formula using the variables $x_1, \ldots, x_n$. The Odd Maximum Satisfiability (OMS) problem can be more formally expressed as:

$$\text{OMS} = \{\psi(x) \mid x_n = 1 \text{ in lex max sat assignment of } \psi\}$$

Krentel (1988) demonstrated that OMS is complete for Δ_2. Remember from section 2.2 that Δ_2 is the class of problems that can be solved in polynomial time by a deterministic oracle Turing machine (TM) that makes a polynomial number of queries to an NP oracle. Krentel begins by defining optimization Turing machines and a class called OptP based on them. These are described in chapter II, but basically an optimization TM is a nondeterministic machine in which we write a value on each accepting computation and the machine "magically" returns the maximum or minimum of these values. An OptP problem is one that is solvable in a polynomial amount of time on an optimization TM. Krentel proved a relationship between OptP and Δ_2 which allows us to characterize problems of one class in terms of complete problems of the other. Specifically, every problem in Δ_2 can be expressed in terms of an OptP problem followed by a polynomial computable predicate.

Krentel's proof is significant for at least two important reasons (besides the result). One is that he defined the notion of a metric reduction that has been used in subsequent research and literature. Krentel also was able to establish a relationship between functional complexity classes and traditional complexity classes of decision problems. However, it is still valuable to have a more direct proof based on traditional complexity classes.

The Δ_2-completeness result can be established by using a Cook-style reduction (Cook 1971) from an oracle Turing machine into an instance of OMS. There are a number of advantages to using traditional proof techniques along with conventional complexity classes and models of computation. By using well-known methods and

elements, the proof is often easier to understand and easier to modify in order to prove related results and extend the work that has been done. Additionally, extending Cook's result assists us in better understanding the relationship between NP and Δ_2.

Cook (1971) showed that SAT is NP-complete by demonstrating how to construct a satisfiability instance that simulates the execution of a nondeterministic Turing machine (NDTM). We will provide an outline of Cook's reduction and refer the reader to other sources for more complete details.

Every language in NP is solvable by some NDTM that operates in a polynomial amount of time. Cook created a set of variables Q[i, k] to represent that at time i, the machine is in state k. He also created a set of variables to represent the contents of the tape, S[i, j, k] meaning that at time i, the j^{th} tape square contains the symbol k. The last set of variables that he creates are H[i, j] to represent that the position of the tape head is j at time i. The execution time of the machine is bounded by some polynomial, p(n), thus the number of tape squares and the number of variables created is also bounded by a polynomial.

Using these variables, Cook created a set of clauses that ensure each of the following:

1. M' is in exactly one state
2. The tape head is in exactly one position
3. Each tape square has exactly one symbol at each time
4. At time 0 the machine is in the initial configuration
5. The machine enters the accept state by time p(n)
6. Execution follows according to the transition function

To enforce the first restriction, Cook introduced two sets of clauses. The first are of the form {Q[i,0], Q[i,1], …, Q[i,r]} for every i, meaning that at time i the machine is in at least one state. Next we need to represent that Q[i,j] ⇒ ¬Q[i,j'] for all combinations of i, j, and j' such that $j \neq j'$. This ensures that at time i we are in at most one state, because if we are in state j at time i then we cannot be in state j' for any $j' \neq j$. This is not a clausal representation since it uses an implication, so the second group of clauses are of the form {¬Q[i,j], ¬Q[i,j']} for all combinations of i, j, and j' such that $j \neq j'$.

The clauses for the second and third restrictions are nearly identical to the first. Cook encodes that the tape head is in at least one position at each time and at most one position at a time. He imposes the third restriction by encoding that each tape square contains at least one symbol and at most one symbol at any given time.

For the fourth restriction, Cook introduces the clauses {Q[0,0]} and {H[0,1]} to represent that the machine is in the start state with the tape head at the first position on the tape. He then uses a set of clauses to encode that the contents of the first |*w*| tape squares contain the input string *w*. Finally he includes clauses to represent that the rest of the tape squares are initially blank.

The fifth restriction can be expressed using a single clause {Q[p(*n*), 1]} where the states have been specially numbered such that state 1 is the accepting state. This clause states that we are in the accepting state at time p(*n*).

Cook breaks the last restriction into two parts. The first guarantees that if the tape head is not at position j at time i, then the symbol at position i cannot change from time i to time i+1. The implication is (¬H[i,j] ∧ S[i,j,l]) ⇒ S[i+1,j,l] meaning that at time i, if

the tape head is not at j and the symbol at position j is l, then the symbol at position j must be l at time i+1. The clause to represent this is {H[i,j], ¬S[i,j,l], S[i+1,j,l]} for each possible combination of i, j, and l. Intuitively if the tape head is at position j at time i then the symbol in that tape square is allowed to change. Otherwise either the j^{th} tape square does not contain symbol l at time i, or it does contain symbol l at time i+1.

The second part of the last restriction is that the change in state and any changes to the tape follow from the transition function. First we encode that the tape head moves according to the transition function. At time i, if we are in state k, the tape head is at position j, and the symbol on the tape at that position is l, then this determines whether the transition function will tell us to move left or right (+1 or -1 in our encoding). Let Δ represent the output of the transition function. Then $(H[i,j] \wedge Q[i,k] \wedge S[i,j,l]) \Rightarrow H[i+1,j+\Delta]$. In clausal representation, this translates to $\{\neg H[i,j], \neg Q[i,k], \neg S[i,j,l], H[i+1,j+\Delta]\}$.

The final sets of clauses are designed to require that the state change follows from the transition function and that the symbol written to the tape follows from the transition function. These are constructed similar to the clauses to encode the requirement that the tape head behaves as it should. So we write $\{\neg H[i,j], \neg Q[i,k], \neg S[i,j,l], Q[i+1,k']\}$ and $\{\neg H[i,j], \neg Q[i,k], \neg S[i,j,l], S[i+1,j,l']\}$ where the state k' and the symbol l' are determined by the transition function. If the machine is in the accepting state at time i, then $k' = k$, $\Delta = 0$, and $l' = l$. The set of clauses is satisfiable if and only if there is an accepting computation on the NDTM. For more complete details on Cook's reduction see his paper (Cook 1971). A similar reduction can be used to show the following:

Theorem 4.1.1 OMS is Δ_2-complete.

Proof: Clearly OMS is in Δ_2 because to find the lexicographically maximum assignment we can go through each variable in order asking the SAT oracle if the formula is satisfiable with that variable set to one. This requires a linear number of queries, after which we need only check if the last variable is true or false.

We must now show that all languages in Δ_2 are reducible to OMS. First, let M be a machine that solves an arbitrary language in Δ_2. We then construct a machine M' by replacing the oracle with a machine for SAT. Next, we use a Cook-Style reduction to convert M' into a SAT formula. M' is not equivalent to M because an NDTM for a language is not equivalent to an oracle for the same language, but the Boolean formula that we create will have the property that the lexicographically maximum satisfying assignment is odd precisely when M would have accepted.

Let L be a language in Δ_2, thus there is a deterministic oracle TM that solves L. Without loss of generality we can assume that the oracle is a satisfiability oracle. Let the machine for L be $M = \{Q_M, q_{0\_M}, q_{accept\_M}, q_{reject\_M}, \Sigma_M, \delta_M, q_?, q_Y, q_N\}$ where Q_M is the set of states, q_{0_M} is the start state, q_{accept_M} and q_{reject_M} are the accepting and rejecting states, Σ_M is the alphabet, and δ_M is the transition function. M has two tapes: a work tape and an oracle tape. When the machine enters the oracle state $q_?$, it transitions to q_Y if the string on the oracle tape is satisfiable and q_N otherwise. A diagram of M is shown in figure 4.1.1, with a square drawn around the oracle section to highlight it.

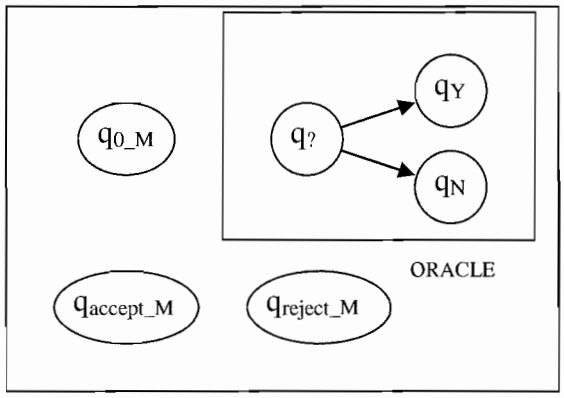

Figure 4.1.1 Diagram of a deterministic oracle TM.

Let MSAT be a polynomial-time NDTM that solves SAT and let MSAT = $\{Q_{MSAT}, q_{0\_MSAT}, q_{accept\_MSAT}, q_{reject\_MSAT}, \Sigma_{MSAT}, \delta_{MSAT}\}$. As before, Q_{MSAT} is the set of states, q_{0_MSAT} is the start state, q_{accept_MSAT} and q_{reject_MSAT} are the accepting and rejecting states, Σ_{MSAT} is the alphabet, and δ_{MSAT} is the transition function.

We now construct another machine M' by replacing the SAT oracle in M with the machine MSAT. Clearly M' is not equivalent to M because an NDTM for a language is not as powerful as an oracle for the same language. M' still has two tapes. The work tape from M contains the input and is still a work tape in M'. The oracle tape from M can still be written to at any time during execution and is used as a work tape in the MSAT section of the machine. The final change that we make is to create a new accepting state with ε-transitions (i.e., transition on no input where the tape head does not change) from the original accept state and reject state. Logically we want M' to always accept and set a bit that indicates whether M would have accepted or rejected. A diagram of M' is shown in figure 4.1.2.

We can assume that there are no transitions back into the start state of MSAT during the computation of the SAT query. If there were then we could easily create a new start state with an ε-transition into the original start state.

M' is an NDTM so we can use a Cook style reduction to a SAT instance. Though the reductions are similar, there are some important distinctions between our reduction and the one that Cook constructed. We begin by relabelling the states $q_0, \ldots, q_v$ where $v = |Q_{M'}| - 1$. Let $q_0 = q_{0\_M'}$, $q_1 = q_{accept\_M'}$, $q_2 = q_{accept\_M}$, $q_3 = q_{reject\_M}$, $q_4 = q_{0\_MSAT}$, $q_5 = q_{accept\_MSAT}$, and $q_6 = q_{reject\_MSAT}$. The other states can be assigned to the remaining labels in any order. This relabeling is shown in figure 4.1.3.

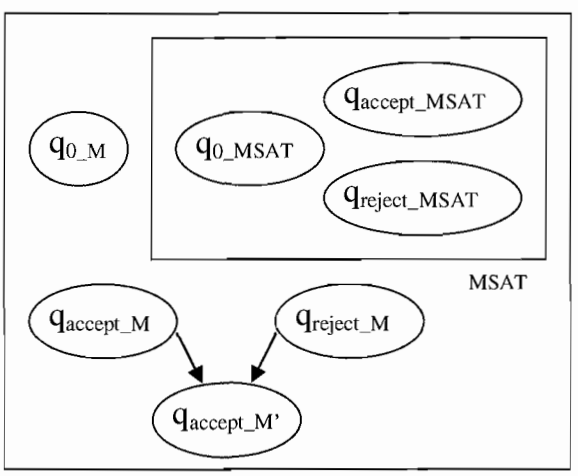

Figure 4.1.2. Diagram of machine M' obtained by modifying oracle TM.

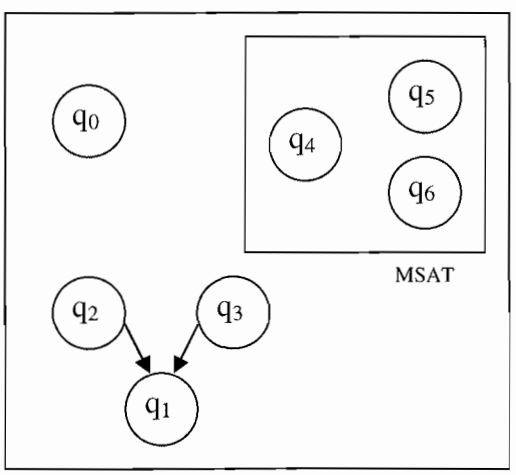

Figure 4.1.3. Diagram of M' with states relabeled.

Let us also label the elements of the tape alphabet Σ as $s_0, \ldots, s_z$ where $z = |\Sigma| - 1$. Let s_0 represent the special blank symbol and assign the remaining labels to the rest of the symbols in any order. As in Cook's reduction, let $p(n)$ bound the number of time steps and assume without loss of generality that $p(n) \geq n \; \forall n \in \mathbb{Z}^+$. During this time we

cannot use tape squares outside the range -p(n) to p(n)+1 on either tape. Moreover, we are guaranteed that we will not enter the oracle state (q_4) more than a polynomial number of times so let q(n) bound the number of oracle queries.

Table 4.1.1 shows the variables used in the construction. Q[i, k] is the same as in Cook's reduction. Cook used H[i, j] to represent the tape head position and S[i, j, k] to represent the tape contents. We need two sets of these variables since there are two tapes. Additional variables are needed to handle the oracle computation. Based on the variable ranges, there are only a polynomial number of variables used in the construction.

Table 4.1.1. Variables used in satisfiability formula to simulate oracle machine.

Variable	Range	Meaning
Q[i, k]	$0 \le i \le p(n)$ $0 \le k \le v$	At time i M' is in state q_k
H1[i, j]	$0 \le i \le p(n)$ $-p(n) \le j \le p(n)+1$	At time i work tape head is at position j
H2[i, j]	$0 \le i \le p(n)$ $-p(n) \le j \le p(n)+1$	At time i oracle tape head is at position j
S1[i, j, k]	$0 \le i \le p(n)$ $-p(n) \le j \le p(n)+1$ $0 \le k \le z$	The symbol k is in position j of the work tape at time i
S2[i, j, k]	$0 \le i \le p(n)$ $-p(n) \le j \le p(n)+1$ $0 \le k \le z$	The symbol k is in position j of the oracle tape at time i
N[i, j]	$0 \le i \le p(n)$ $0 \le j \le q(n)$	At time i we have entered q_4 exactly j times
O[j]	$0 \le j \le q(n)$	The answer to the j^{th} SAT query
ANSWER	1	Indicates whether or not we would have accepted the input on M

Next we create a set of clauses using the variables in the table that simulate the execution of the machine. As mentioned earlier, Cook created a set of clauses to represent the following list of restrictions:

1. M' is in exactly one state
2. The tape head is in exactly one position
3. Each tape square has exactly one symbol at each time
4. At time 0 the machine is in the initial configuration
5. The machine enters the accept state by time p(n)
6. Execution follows according to the transition function

We use the same clauses to represent the above restrictions that Cook used. For the second and third restrictions we create the appropriate set of clauses for both the work tape and the oracle tape. We also ensure that the oracle tape is blank in the initial configuration and that the oracle tape obeys that transition function. We will not write out the clauses here because the extension from Cook's work is obvious. The clauses to represent these restrictions are polynomial in size and number.

We also create clauses to impose the following restrictions in addition to those that Cook uses:

7. N[i,j] correctly represents the number of SAT queries at time i
8. O[j] represents the answer to the j^{th} query
9. ANSWER is 1 if we transition into the original accept state

Table 4.1.2 shows the clauses that are needed to represent the additional restrictions. We left the equalities in the formulas even though it is not proper CNF format because it is easier to understand. The expressions could easily be converted to

Table 4.1.2 Clauses in this construction that were not in Cook's original proof.

Restriction	Clauses	Range
7	$\{N[0, 0]\}$ $\{\neg N[0, j]\}\ \forall j \neq 0$ $\{Q[i, 4] \vee (N[i, j] = N[i-1, j])\}\ \forall i \neq 0$ $\{\neg Q[i, 4] \vee (N[i, j] = N[i-1, j-1])\}\ \forall i \neq 0$	$0 \leq i \leq p(n)$ $0 \leq j \leq q(n)$
8	$\{O[j] = \exists\ t_1, t_2\ (Q[t_1, 4] \wedge N[t_1, j] \wedge Q[t_2, 5] \wedge N[t_2, j])\}$	$0 \leq t_1, t_2 \leq p(n)$ $0 \leq j \leq q(n)$
9	$\{ANSWER = \exists i\ Q[i, 2]\}$	$0 \leq i \leq p(n)$

sets of clauses. The clauses for the 7th restriction essentially state that either we are in state q_4 and the number of oracle queries is incremented or we are in some other state and the number of oracle queries does not change. We also encode that the original number of oracle queries is zero and that any other number of queries at time zero is false. The number and size of clauses to encode the 7th restriction are polynomial when converted to CNF.

The clauses for the 8th restriction state that O[j] is 1 precisely when we enter state q_4 at time t_1, making it the j^{th} query, and that we enter state q_5 at a later time t_2 while we are still on the j^{th} query. In order to show that the clauses representing this restriction are polynomial, let us represent the second half of the expression as $A[t_1, t_2]$. Given the ranges on t_1 and t_2 this introduces $p^2(n)$ new variables.

We add the clauses $\{A[t_1, t_2] = (Q[t_1, 4] \wedge N[t_1, j] \wedge Q[t_2, 5] \wedge N[t_2, j])\}$ for each t_1, t_2. To eliminate the existential quantifier, we write $\{O[j] = \bigvee_{t1, t2} A[t_1, t_2]\}$ for each j. Given the ranges on t_1, t_2, and j there are a polynomial number of these equalities, each one of which has a polynomial conversion to CNF.

The final restriction encodes that ANSWER is true exactly when we enter the original accepting state at some time step. This can be rewritten as {ANSWER = $\vee_i$ Q[i, 2]} which is clearly non-exponential given the range on i. Thus the construction is polynomial.

This encodes M' as a SAT instance, ψ(M'). As mentioned before, M' is not equivalent to M because an oracle for a language is likely to be more powerful than an NDTM for the same language. Specifically the biggest difference is that in an NDTM there are multiple paths through the machine. If the answer to a query is *no*, we are guaranteed that every computational path will end in the rejecting state, but when the answer to a query is *yes*, there may also be some paths that end in the rejecting state.

By taking the maximum satisfying assignment we can ensure that we say *yes* to the oracle queries whenever it is possible to do so. For this we need to define a lexicographic ordering on the variables that we created. Let the oracle queries be first and in order so that O[i] < O[i+1] for all i. The remaining variables can be arranged in any order as long as ANSWER is last. This way the oracle answers are correct and the input is accepted by M if and only if the last bit is 1. Therefore OMS(ψ (M', w)) is equivalent to the question of whether w $\in$ L_M. ■

Using the above result we can prove the exact capability of the DPLL algorithm. The following propositions prove the exact capability of the version of DPLL that returns a satisfying assignment if one exists and returns UNSATISFIABLE otherwise.

Proposition 4.1.2 DPLL is at most Δ_2-capable unless there is a collapse in the polynomial hierarchy.

Proof: We prove this proposition by showing that if DPLL is capable of solving a language, then the language must be in Δ_2. Let L be a language such that there exists polynomial computable functions t_1, t_2, and t_3, where $t_2(w, det(\text{DPLL}, t_3(t_1(w)))\ (t_1(w)))$ is true iff $w \in L$. If we have access to a SAT oracle, we could solve L in polynomial time by first using the function t_1 to transform the instance into an input for DPLL, then running a procedure similar to DPLL except that we use the oracle to eliminate those calls that would result in UNSAT, and using the function t_2 to convert the output into the correct answer. Each of these steps runs in polynomial time. We can ignore t_3 and *det* because we are using an oracle to determine the nondeterministic branches instead of turning the machine into a deterministic one.

Another way of looking at it is that we first produce an input for DPLL where if we ran the algorithm, it would generate a search tree. Instead of creating the entire search tree, we could use the oracle to tell us at each step whether we should branch on true or false. This turns the search tree into a direct path to a solution. The overall process requires polynomial time with a satisfiability oracle including the transformations t_1 and t_2, making L in Δ_2. ■

One of the most important things to note is that we can still use unit propagation, which returns all consequences of the current partial assignment. Intuitively, if the current

partial assignment is part of the maximum model, then any logical consequences of that assignment must still be part of the maximum model. In other words, unit propagation does not rearrange the search space. It only prunes sections of the search space that do not contain any models. More formally:

Proposition 4.1.3 If a particular branching order arranges the leaves of the search tree from maximal to minimal, performing unit propagation will not cause us to incorrectly return a non-maximal model.

Proof: Unit propagation may rearrange the leaves of the search tree and possibly in such a way that the order on interpretations is no longer respected, but we argue that the output is not adversely affected. Suppose that there are two interpretations I_1 and I_2 with I_1 being preferred over I_2. If neither interpretation is a model of the theory then these two leaves can be swapped because neither will be returned. Similarly, if one is a model and the other is not, then we can interchange them because we will not return the non-model.

The only interesting case occurs when both interpretations are models of the theory. In order for unit propagation to force us to incorrectly return I_2, both interpretations must reside in the subtree rooted at some unit propagation. Furthermore, I_1 must be in the left subtree and I_2 in the right because unit propagation does not rearrange the leaves within either subtree. It simply prunes the right subtree. See figure 4.1.4 for a diagram. This leads to a contradiction since we assumed that both interpretations are models of the theory. The leaves in the right subtree will be pruned nodes because they

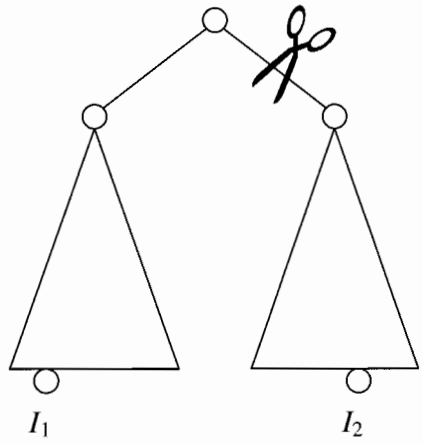

Figure 4.1.4. Search tree with interpretations I_1 and I_2 in separate subtrees.

are not models of the theory. Thus, unit propagation does not cause us to incorrectly return a non-maximal model. ■

Now that we have proven an upper bound on the capability of DPLL, we will prove a lower bound on the capability by showing that it is capable of solving a Δ_2 complete problem. This will give us the exact capability of DPLL.

Proposition 4.1.4 DPLL is at least Δ_2 -capable.

Proof: All problems in Δ_2 can be efficiently reduced to any Δ_2-complete problem. A reduction to a problem implies that we can use any algorithm that solves the problem. Thus, demonstrating that DPLL can solve a Δ_2-complete problem is sufficient to show that DPLL is Δ_2-capable. We show that DPLL can be used to solve Odd Maximum Satisfiability (OMS) which is known to be Δ_2-complete.

$$OMS = \{\psi(x) \mid x_n = 1 \text{ in the lexicographic max sat assignment of } \psi\}.$$

The first transformation, t_1, does not need to do anything because the instances are already SAT formulas. The only nondeterministic components of DPLL are the branching order and whether to try setting variables to true or to false first. The transformation t_3 is used to create a fixed branching order on these variables to make DPLL deterministic. To determine the lexicographic maximum satisfying assignment, we branch on the variables in order and always attempt to set a variable to true first. We can create our deterministic DPLL' in polynomial time by making these simple modifications. Proposition 4.1.3 tells us that we can still apply unit propagation without any negative consequences even though it rearranges the branching order.

After running the DPLL algorithm, let the second transformation t_2 take the satisfying assignment produced and return true if $x_n = 1$. The order on the variables ensures that the satisfying assignment returned is the lexicographically maximum one. The functions t_1, t_2 and t_3 are all polynomial-time computable, thus DPLL is capable of solving OMS and all of Δ_2. ■

Proposition 4.1.4 can also be shown in another way. Giunchiglia and Maratea have shown that the basic DPLL algorithm is capable of solving an OptP complete problem using a fixed branching order. Since it solves an OptP complete problem it can solve every problem in OptP (Giunchiglia and Maratea 2006). Krentel established a relationship between Δ_2 and OptP where every problem in Δ_2 can be represented by an

OptP problem followed by a polynomially computable predicate (Krentel 1988). We could instead have chosen to use Krentel's work converting Δ_2 problems into OptP problems and rely on the results of Giunchiglia and Maratea that DPLL is capable of solving problems in OptP. However, the proof that we provide demonstrates the result more directly. Also, the previous work did not specifically mention or show that unit propagation is still valid and can still be used with the fixed branching order.

The following is a direct consequence of the previous two propositions:

Proposition 4.1.5. DPLL is exactly Δ_2-capable. ■

4.2. How Optimizations Affect the Capability of DPLL

Notice that the proof requires that DPLL return the satisfying assignment if one exists. The version that just returns SAT or UNSAT may not be Δ_2-capable. In this section, we discuss variations of the original DPLL algorithm and which versions may not be Δ_2-capable.

Many modifications have been made to the basic DPLL algorithm to make it more efficient. Some of these modifications can still be used in solving Δ_2 problems, if they do not change the order in which models are discovered. Any modification which reorders the search tree is unusable unless there is another method that can be devised to ensure that the first model found is maximal. The purpose of this section is to analyze

which modifications can still be used in solving Δ_2 problems. For those that cannot be used, we also analyze how this affects the overall performance.

The first change to DPLL that we consider is the use of branching heuristics. Many optimizations of DPLL involve using intelligent branching heuristics that select the next branching variable in an attempt to minimize run time. One of the more popular branching techniques is VSIDS, which will assign an initial value to each variable based on the number of clauses that it appears in. When new clauses are learned, the score is incremented. Periodically all of the scores are divided by a constant so that the score reflects the number of occurrences, with a higher weight on more recently added clauses.

To show that we cannot use branching heuristics let us consider a simple example. Suppose that we use a really simple heuristic of branching on the literal that appears in the most clauses that have not been satisfied by the current partial assignment. If we are given the formula $(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$ then the model that will be returned is $\{\neg x_1, x_2, x_3\}$. While this model is a satisfying assignment of the formula, it is not the maximum satisfying assignment so it does not solve the OMS problem.

The proof that DPLL is Δ_2-capable requires a specific branching order to solve OMS. Unless there is another proof that does not require a fixed branching order or we can add clauses that force the branching heuristic to branch on the variables we want, then we cannot use the intelligent branching heuristics that were later added to the algorithm.

In order to see how much this affects performance, we analyze how much a fixed branching order (or partial order) affects the runtime of two of the most popular SAT

solvers. For our experiments, we modified the RSAT solver to use random branching up to a certain depth. It then relies on the branching heuristic used by RSAT for the rest of the decisions. Our analysis is based on the assumption that a fixed branching order will, in general, be no worse than a random one. We chose RSAT because it was one of the top SAT solvers in the last SAT competition.

The modified solver was run on a series of forty problems taken from SATLIB benchmarks (SATLIB), exactly half of which are satisfiable. These benchmarks include random instances, graph coloring problems, planning problems, Latin squares, as well as problems from the DIMACS benchmark set. Each data point represents the average runtime on the forty problems for a fixed number of random branches. In order to prevent larger problems from dominating the averages, we normalized the data by dividing the runtime by the time it takes on the original RSAT implementation.

As we increase the depth of random branching, we expect that the runtimes will grow no worse than 2^n where n is the depth of randomly selected branches. In the worst case, we are adding a new root to the search tree that does not provide any additional information. In this case, we are essentially making two copies of the original search tree as the left and right children of the root node. Thus the runtime should be no worse than twice the original, and in general should be better. This hypothesis is validated by the data, as can be seen in figure 4.2.1. Using gnuplot (gnuplot homepage), the closest exponential curve fitting the data is $f(x) = 0.82\ (1.67^x)$, which is less than 2^x.

For good measure we also tested the effect on ZChaff which is not as good as RSAT, but certainly a competitive SAT solver. The results were very similar and the best exponential curve fitting the ZChaff data was $f(x) = 3.84\ (1.76^x)$.

In the ZChaff data there was a substantial difference between the runtime for satisfiable versus unsatisfiable instances. As mentioned, in the worse case we are making two copies of the original search tree and looking through both. For many satisfiable instances we do not actually need to search through both copies of the original search tree and thus we should only see marginal degradation in performance. This difference was not seen in the RSAT data. It would be interesting to do further exploration into the reason behind this difference, but it was not investigated in this work.

Using a fixed branching order up to a certain depth should make a much more significant difference in unsatisfiable instances. This expectation was confirmed as can be seen in figure 4.2.2. The curve fitting the unsatisfiable instances is $f(x) = 6.91\ (1.77^x)$ which is still less than 2^x. The runtime growth for the satisfiable instances does not appear to be exponential at all.

The next change to DPLL that we consider is the use of watched literals (Moskewicz et al. 2001). We argued previously that we can still use unit propagation in DPLL. One of the more efficient implementations of unit propagation involves watched literals. For each clause, we pick two unvalued literals to watch. If either of those literals is set to false while the other is unvalued, we select a new literal to watch if possible. If this is not possible, then either the clause is already true (in which case we do nothing) or the clause is a unit clause (in which case we perform unit propagation). If at any point, a

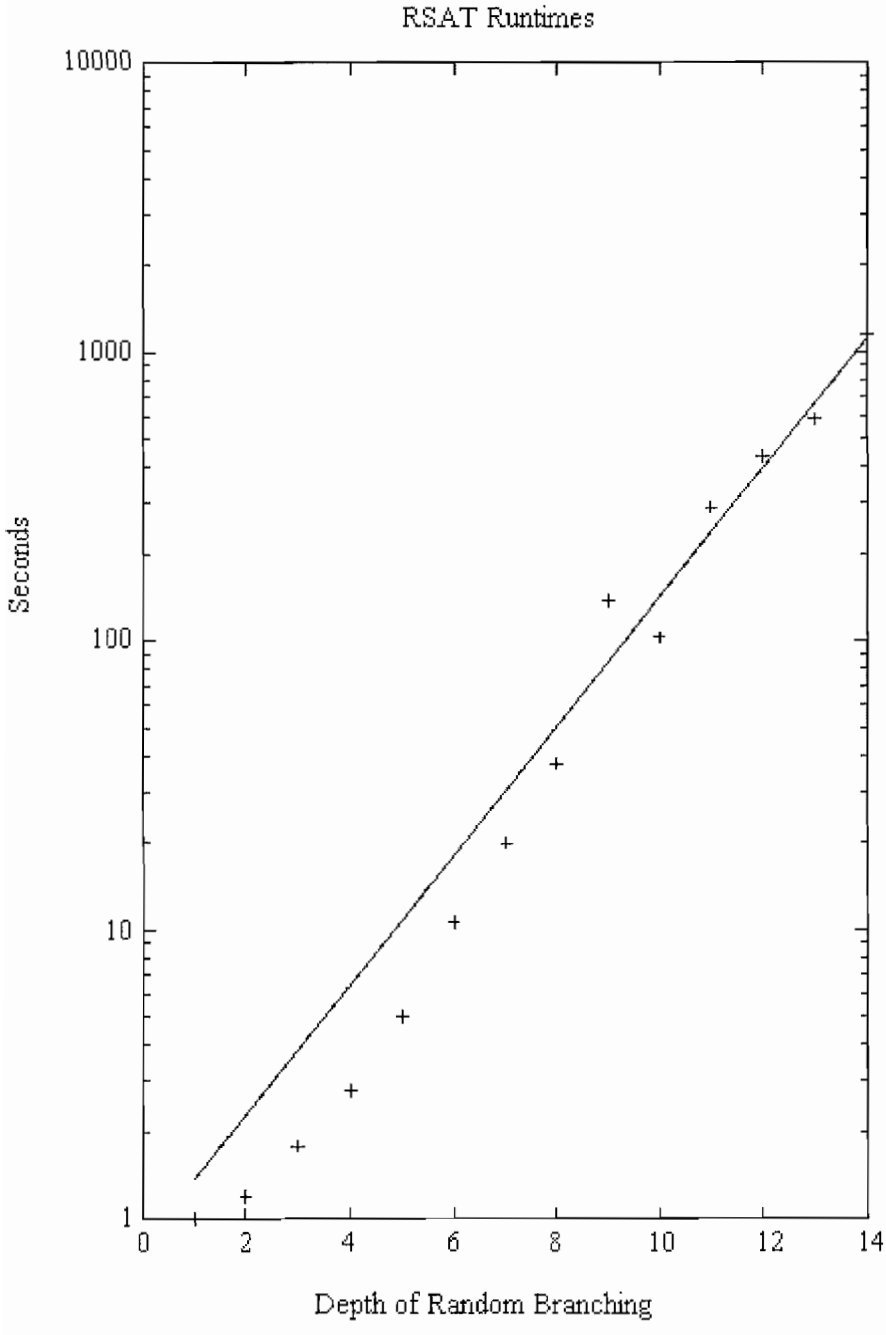

Figure 4.2.1. Runtime vs depth of random branch decisions.

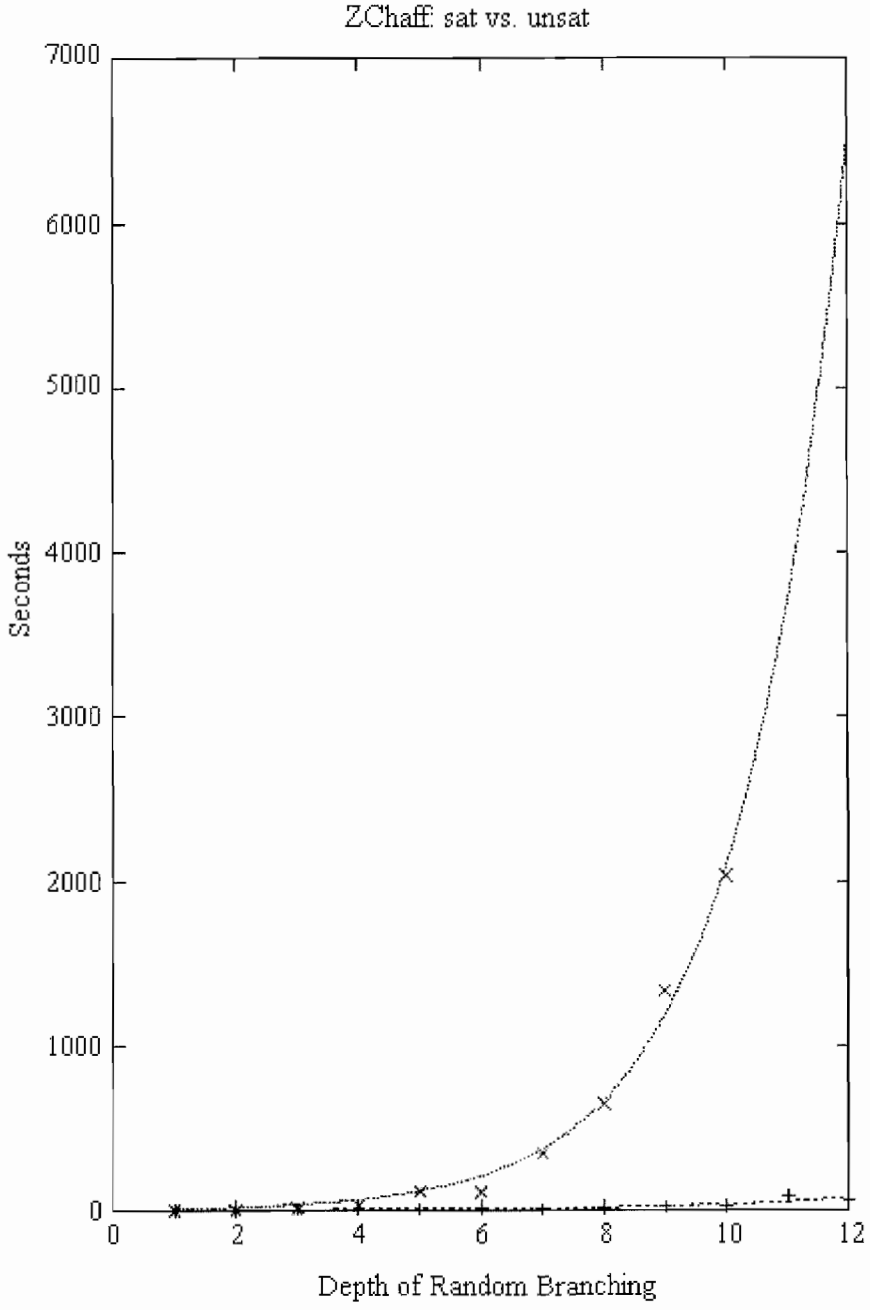

Figure 4.2.2. Difference between sat and unsat instances on ZChaff.

watched literal is set to false and all other literals are already false, we return UNSATISFIABLE. The watched literal invariant that must be maintained is that for every clause either one of the literals is true or both are unvalued. We will now argue that DPLL with watched literals is still Δ_2-capable.

Proposition 4.2.1. Adding watched literals to DPLL does not affect its capability.

Proof: Watched literals are primarily a book-keeping device enabling the algorithm to determine more quickly whether or not there are any unit clauses. It might affect the order in which we find unit clauses, but using watched literals does not alter the branch order and thus does not alter the order in which the interpretations are searched. Since we search the interpretations in order, the first model found will still be a maximum model. We conclude that we can still use this technique for solving problems in Δ_2. ■

Although we can still use watched literals, the watched-literal invariant is easier to maintain when we use nonchronological backtracking, which does rearrange the search space. So next we consider whether or not we can use this technique.

Nonchronological backtracking is used in coordination with learning new clauses. When we reach a dead end in searching for a model, we can use resolution to derive a new clause that represents the reason that we reached a dead end. By adding this new clause to the formula, we may be able to prune sections of the search space that do not contain models. When we learn a new clause, we backtrack to the point where the clause

would have been unit had it been around since the beginning and insert new assignments into the assignment stack to satisfy this clause, which may rearrange the search space.

Proposition 4.2.2. The addition of learning and nonchronological backtracking do not affect the capability of DPLL.

Proof: The most important thing to note about learned clauses is that the new formula is equivalent to the original.

The new search tree that is generated when using learned clauses is clearly not the same as the original search tree without learned clauses. By adding additional clauses, we may introduce new unit propagations that would not have occurred in the original search tree. The new search tree is also not identical to the search tree that would have been generated if we had started with the new formula. If we learn a new clause halfway through the computation we may skip over some of the earlier unit propagations because those unit clauses did not exist at the time.

Earlier we argued that we can still use unit propagation without reordering the models. It is also the case that we can skip unit propagations without rearranging the models. The proof that we can still use unit propagation does not require us to use unit propagation whenever it is possible to do so. The search tree generated with the use of learned clauses and nonchronological backtracking only differs from the search tree of the new formula in that we may skip some unit propagations that appear in the search tree for the new formula. Thus, these two search trees produce the same maximal model. We

argued above that the search tree for the new formula also produces the same maximal model as the search tree for the original formula, thus the model that we find when using learning and nonchronological backtracking is indeed the one that we want.

We conclude that we are still able to use learning and nonchronological backtracking. ■

Another extension to DPLL is called *pure literal propagation*. If a literal occurs in a formula and its negation does not, this variable can be set accordingly. We cannot use this technique to solve the odd max sat problem because the maximum satisfying assignment may set some pure literals to false. This is not a significant setback because people tend to agree that pure literal propagation requires more time than it saves. It is costly to perform searches for pure literals and few instances contain enough pure literals to achieve a performance advantage (Zhang and Malik 2002). However, if there is a specific problem that would benefit from pure literal propagation, then it could still be used after all variables in the preset branching order had been fixed or we could add trivial clauses to eliminate certain pure literals that adversely affect the branch order.

Another extension to DPLL is called symmetry breaking (Crawford et al. 1996). Two variables are symmetric if the formula does not change when the two variables are interchanged. In other words, if there are two models of a theory that are identical except that one has $(x, \neg y)$ and the other contains $(\neg x, y)$ then x and y are symmetric. This technique is currently not used in the ZChaff implementation or the RSAT implementation.

The original version of DPLL ignores symmetries, but exploiting them has the potential to make the algorithm perform faster. One way to make use of the symmetries is to add symmetry-breaking predicates to the theory. Symmetry-breaking predicates are chosen to be true of exactly one element in each equivalence class of assignments generated by the symmetry. In the example used in the previous paragraph, adding (y, x) would break the symmetry.

It is possible to detect some symmetries without knowing the models of a formula. However, if we detect that two variables are symmetric, we do not want to assume that one is true and the other is false because the maximum satisfying assignment requires that both be set to true if possible. Symmetry-breaking predicates create a new formula such that the models of the new formula are a nonempty subset of the models of the original (assuming that there are models of the original). The new formula is not equivalent to the original and we may end up cutting the most preferred model of the theory. Therefore, adding symmetry-breaking may destroy the Δ_2-capability for DPLL.

In the next chapter, we provide a brief discussion of how the capability of DPLL is applicable to real problems, but first we will discuss the capability of another well known algorithm.

4.3. PSPACE-Capability of QBF Algorithm

Quantified Boolean Formula is the canonical PSPACE-complete problem. The most commonly used method of solving the problem is an extension of the DPLL

algorithm. The algorithm essentially has one DPLL-like procedure to handle existential quantifiers and one to handle universal quantifiers. The two versions are called in alternation to solve the problem. For further details see section 2.3.

Since the QBF algorithm solves a PSPACE-complete problem it is clearly at least PSPACE-capable. It only remains for us to show that the algorithm is no more than PSPACE-capable.

Proposition 4.3.1 The QBF algorithm is no more than PSPACE-capable.

Proof: Suppose that a problem Q can be solved using the QBF algorithm. This means that there are polynomial-time transformations of the input and output such that we can use the QBF algorithm as a black box for solving Q. Since the transformations take polynomial time, they cannot take more than polynomial space. We can write a PSPACE algorithm for Q which consists of the input transformation, followed by the QBF algorithm, and ending with the output transformation. Since the individual components operate in polynomial space, the overall algorithm does also, and $Q \in$ PSPACE. ■

CHAPTER V

APPLICATIONS TO PLANNING PROBLEMS AND NMR

Now that we know that the DPLL algorithm is Δ_2-capable, we can examine specific problems that it can and cannot solve. For the problems that it cannot solve, we can analyze whether there are interesting subsets of the problem that can be solved. The two problem domains that we consider are planning problems and nonmonotonic reasoning.

5.1. Optimal Planning

Satplan (Kautz et al. 2006) is one of the most efficient solvers for finding solutions to planning problems. It is based on the Planning as Satisfiability approach introduced by Kautz and Selman (1992). Satplan took first place the International Planning Competition in 2004 and tied for first in 2006.

Satplan's first step involves generating a plan graph for a planning problem. Recall from chapter II that a plan graph is a visual representation of the planning problem that can be constructed as follows: create a set of nodes to represent the initial conditions, with one node for each condition. Next create a set of nodes to represent all possible

actions that may be taken, including no-ops. Then connect the preconditions of an action to the node representing that action. Following that, create a set of nodes representing all of the possible postconditions and connect each action to its corresponding postconditions. Repeat this process either up to some fixed length k (to bound the plan length to k) or until two successive sets of conditions are identical.

The basic algorithm of Satplan is:

- Generate the plan graph up to length *k* (initially 1)
- If the goals are unreachable in the plan graph increment *k* and start over
- Convert the plan graph into SAT formula
- Call a satisfiability solver
    - If UNSAT increment *k* and try again
    - If SAT return solution

Each time a plan graph is generated and converted to cnf, the satisfiability instance represents the question, "can we find a plan of length *k*?" Satplan automatically finds an optimal solution (a plan of minimum length) because it searches all possible plan lengths in increasing order.

There are drawbacks to using Satplan for optimal planning. Probably the biggest concern is that learned information is discarded between successive calls to the SAT solver. Learning techniques have significantly improved the performance of modern SAT solvers. Throwing away learned information between individual calls is particularly detrimental when using satisfiability for planning because of the strong similarities

between successive SAT instances. Information learned in searching for a plan of length k is valuable in searching for a plan of length k+1.

Prior work has been done on retaining learned clauses between successive calls (Nabeshima et al. 2006). They use a sequence of calls to a SAT solver, but keep learned information to avoid redundant work. Their solution outperforms Satplan, demonstrating that retaining learned clauses is useful.

As an alternative to their approach, we propose an approach that uses a single call to a SAT solver. Our approach is based on our notion of capability. We have shown that DPLL is Δ_2-capable and we will prove that Optimal Planning is $F\Theta_2$-complete and hence DPLL is capable of solving Optimal Planning.

To use a single SAT call, we first generate the entire plan graph which involves continuing to generate layers until a layer is identical to the one before it. We then determine a partial order on branch decisions that will ensure optimality. Finally we call the SAT solver, which uses our fixed partial order before relying on its own branching heuristic. Using a single SAT call automatically eliminates the problem of discarding learned information, but may also yield additional benefits by relying on the optimization techniques built in to the SAT solver.

Unfortunately, this approach is not feasible in practice, because generating the entire plan graph requires too much memory. To handle this issue, we place an upper bound on the plan length. For instance, we could solve blocks-world problems by moving all blocks to the table and building up the goal state. The number of steps required to do this is a naïve upper bound on the optimal plan length. We modified Satplan so that it

takes an extra argument which represents an upper bound on the plan length and uses only one SAT call to find an optimal solution. Our experiments indicate that these changes result in a dramatic runtime improvement.

For problems such as blocks-world it is easy to generate an upper bound on the length of an optimal solution. For many other problems it can be more difficult. Furthermore, it is desirable to have a self contained planner that does not rely on additional input. We modified our first version by automating the process of finding an upper bound. Again we outperformed Satplan by a significant margin. Now that we have described our approach in general terms, we will turn the discussion to a more detailed description and then show the results of our experiments.

As mentioned earlier, one problem that we address with using Satplan for optimal planning is the loss of learned information between calls to the SAT solver. We could instead make a single call by generating the entire plan graph up front and using a predefined branching order to guarantee optimality. When converting to a satisfiability instance, we would introduce new variables, G_i, indicating that the goal state has been reached at time i. We add the necessary clauses to the formula to ensure that these variables have the intended meaning.

When we call the SAT solver, we branch on the variables G_1, G_2, … in order, so that the first plan that we find is guaranteed to be optimal. We are effectively doing the same as Satplan by first searching for a plan of length 1, then of length 2, and so on until we find one. However, this method automatically retains learned information because we are pushing the search into one SAT call.

As pointed out earlier, generating the entire plan graph requires too much memory so this approach is not practical. If we have a reasonable upper bound on the plan length, we could solve the problem in a single call by generating the plan graph up to the upper bound, translating to a SAT problem, and solving. The fixed branch order ensures that we find an optimal solution regardless of how large the upper bound may be. This approach compares favorably to Satplan on a set of problems taken from the IPC-5 benchmarks.

Note that Satplan allows us to specify which SAT solver we would like to use. The SAT solvers are independent from Satplan, not built in to it. For our experiments, we used both Tinisat (Huang 2007) and RSAT (Pipatsrisawat and Darwiche 2007).

Now that we have described the idea behind our solver, we will outline the actual modifications that were made to Satplan. The differences are:

- Introduce one new variable for each time step and add clauses that ensure the variable is true iff all of the goals have been satisfied at that time step.
- Modify Satplan to write one extra line to the cnf file which specifies the branch order.
- Modify Tinisat and RSAT to accept a partial order on branching and to use these first before relying on their heuristics for branch decisions.
- Modify Satplan to generate the plan graph up to some fixed upper bound on the plan length.

This is still not ideal because it assumes that we always have an upper bound on the plan length and it requires additional input. Rather than input an upper bound on the plan length, we can automate the process by starting with an initial guess, generating the

plan graph up to that guess, and converting it to a SAT instance. If no solution is found then we multiply our previous guess by some constant (larger than one) and repeat. If a solution is found, it will be optimal because of the partial branching order. We use fixed parameters for the initial guess on the plan length and the multiplicative constant. These parameters are independent of the specific domain or problem being considered.

It is not a new idea to try to guess the optimal plan length and adjust accordingly. However, in previous attempts, if a solution was found from the initial guess, there was no guarantee that the solution was optimal. The planner still had to try again with a smaller value of k to see if a smaller plan could be found. Our method is guaranteed to be optimal because of the partial order on branch decisions. Also, in previous versions k was incremented or decremented. We are multiplying it by some constant which allows us to converge on a solution more quickly. Instead of generating the plan graph up to some fixed upper bound, we make the following additional change:

- Satplan will first generate the plan graph up to some guess k. If a solution is found it returns it as optimal. If no solution is found, we multiply k by some constant and try again.

For small values of k, we will not generate the cnf instance because we can determine from the plan graph that the goals are unreachable. There is a range of values where k is large enough that the goals can be reached in the plan graph (and hence we generate a SAT instance), but small enough that there is no plan of length k or less. There are no calls to a SAT solver before we enter this range, and the final call to a

satisfiability engine will be when we have increased *k* beyond this range. Only when we are within this range do we make additional, unnecessary calls to the SAT solver. Depending on the size of the range and the multiplicative constant, we may skip over this range entirely and still end up making only one call. In most other cases, we end up making only a few calls, because increasing *k* by some multiple each time quickly puts us beyond the range of values where we make additional calls. The overall result is that we make very few calls to a satisfiability engine while still maintaining optimality.

For our experiments, we compared our modified version of Satplan to the original, unmodified version. Satplan uses a command line option to specify which SAT solver is being used. We compared our planner to Satplan using Tinisat for both as well as using RSAT for both. We used the fully automated version that will start from a fixed upper bound value of *k* and multiply *k* by some constant each time that a plan cannot be found.

We named out planner "Cricket" because it takes large jumps through the search space instead of stepping through each possible plan length until finding a plan. We selected a random set of problems from each of the domains used in the 2006 planning competition. We also used a set of blocks-world problems. The results are shown in table 5.1.1, comparing our planner to the original Satplan. We outperformed Satplan on most of the instances, some by a considerable margin. It is also worth noting that in the one domain where Satplan does better, Cricket solved additional instances where Satplan timed out. These instances are not included in the results shown in table 5.1.1.

Table 5.1.1. Cricket vs. Satplan

Planner / Domain	Satplan – RSAT	Cricket - RSAT	Satplan – Tinisat	Cricket – Tinisat
Blocksworld	3025.31	1417.31	3113.54	1740.06
Pathways	164.38	189.64	354.29	195.36
Pipesworld	845.36	506.91	869.21	637.91
Rovers	458.27	253.08	447.92	256.4
Storage	702.24	473.29	2510.96	282.31
TPP	872.2	718.21	799.41	700.5
Trucks	302.19	140	605.25	279.95

We also want to note that the results on RSAT are incomparable to the results on Tinisat because we used different sets of randomly selected instances from the domains. In addition to showing these results numerically in the table, we found it enlightening to show them in a graph as well. The graphs are shown in figures 5.1.1 and 5.1.2.

We will now describe how these results were borne out of our definition of algorithm capability. Plan existence is PSPACE complete in general and NP complete when there is a polynomial bound on plan length. We prove that when there is a polynomial bound on plan length that optimal planning is $F\Theta_2$ complete, where Θ_2 is the set of problems solvable in deterministic polynomial time given a logarithmic number of queries to an NP oracle.

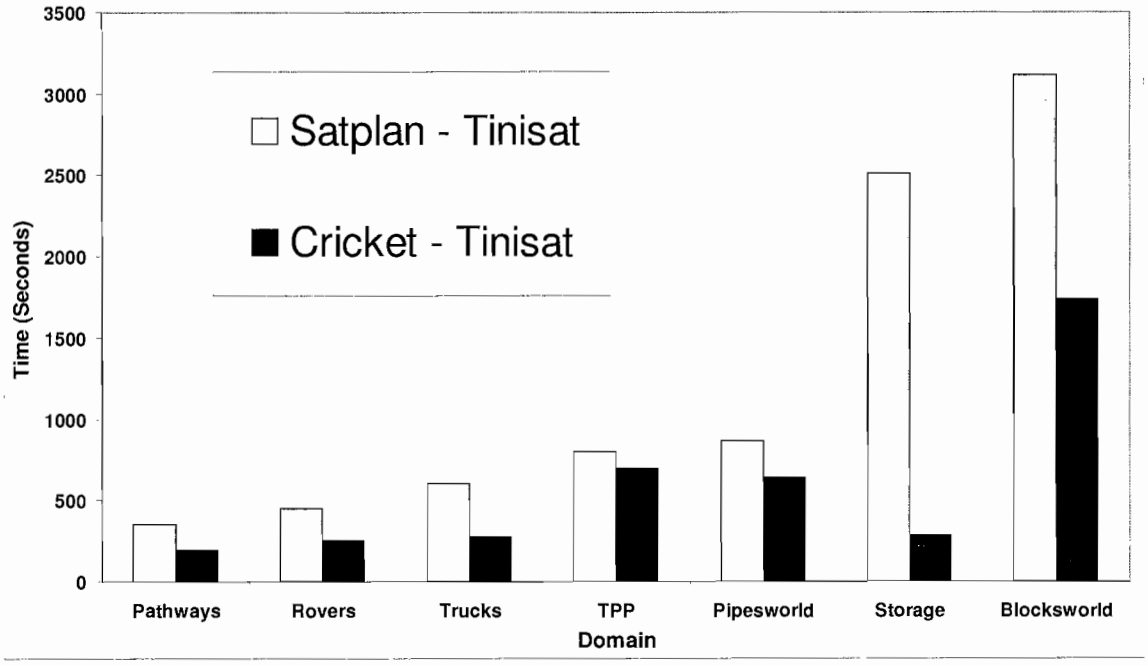

Figure 5.1.1 Cricket vs. Satplan using the solver Tinisat.

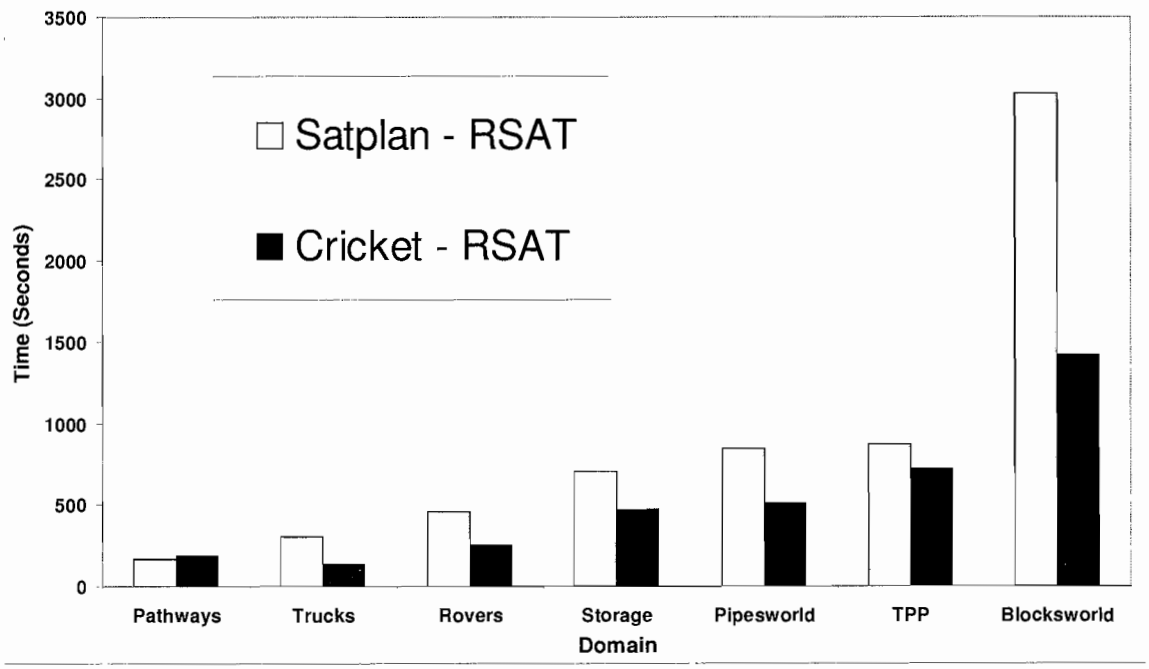

Figure 5.1.2. Cricket vs. Satplan using the solver RSAT.

Planning problems can be arbitrarily complex since some problems require plans of exponential length, thus planning is EXPTime. While it is beyond PSPACE to find a plan for an arbitrary problem, determining plan existence is known to be PSPACE complete (Bylander 1991). There are a number of restrictions, such as a polynomial bound on plan length, that reduce the complexity of planning from PSPACE to NP. If we know that the plan length is bounded by a polynomial, then the plan serves as a polynomial sized witness that the planning problem is solvable. Hence polynomial bounded planning is in NP.

In Satplan, we can generate satisfiability instances in polynomial time if the plan length is polynomially bounded. In general, Satplan may generate exponentially sized SAT instances. The satisfiability instances are only guaranteed to be polynomial when the planning problem is in NP.

Whenever it is NP-complete to determine whether or not there is a plan, it is also NP-complete to ask if there is a plan of length k for some fixed k. However, it is harder than NP to ask for the length of the optimal plan. Similarly the complexity of finding an optimal plan is higher than the complexity of finding any plan. The following proof characterizes the complexity of optimal planning:

Proposition 5.1.1 Optimal planning is $F\Theta_2$-complete when the plan length is polynomially bounded.

Proof: It is in NP to ask whether there is a plan of length k for some fixed k. Since the plan length is bounded by some polynomial $p(n)$, we can use a divide and conquer technique to recursively divide the search space in half at with each oracle query. We initially ask if there is a plan of length ½ $p(n)$. The answer to this question eliminates half of the search space. Thus, the overall number of queries is $O(\log p(n)) = O(\log n)$.

Next we show that optimal planning is $F\Theta_2$-hard by reducing from a know $F\Theta_2$-complete problem. The decision version of clique asks, for a given graph G and integer k, whether there is a clique of size k. Determining the size of the largest clique is $F\Theta_2$-complete (Krentel 1988).

In order to reduce the problem of finding the size of the largest clique to optimal planning, we need to construct a set of variables, a set of actions, a start state, and a goal state. Informally, the actions that we will use are removing vertices, the start state is the initial graph, and the goal state is a complete graph on k vertices.

More formally, let the variables be:

- $edge(v_i, v_j)$ meaning there is an edge between v_i and v_j.
- $removed(v_i)$ meaning that the vertex v_i has been removed from the graph.

The only action that we use in the construction is $remove(v_i)$, which has no preconditions and whose postconditions are $removed(v_i) \wedge \forall j \neg edge(v_i, v_j)$. The start state is that $removed(v_i)$ is false for all i, and $edge(v_i, v_j)$ is true if there is an edge between v_i and v_j in the initial graph. The goal state is that we are left with a complete

graph: $\forall i,j\ removed(v_i) \vee removed(v_j) \vee edge(v_i, v_j)$. In other words for every pair of vertices, either one of them has been removed or there is an edge between them. The plan with the fewest actions removes the fewest vertices resulting in the largest clique. To obtain the size of the largest clique, we subtract the length of the optimal plan.■

Even though optimal planning is harder than SAT we can still use a single call to DPLL to solve it because DPLL is capable of solving all problems in Δ_2 and $\Theta_2 \subseteq \Delta_2$. Figure 2.2.5 gives a complexity diagram illustrating the relationship between the various classes and problems.

The complexity of deterministic planning in the general case is beyond Δ_2. We can still use our method to solve optimal planning in the general case, but the transformations are not guaranteed to be polynomial.

5.2. Planning with Preferences

Sometimes in planning it is useful to consider preferences for some plans over others. In the most general sense, planning with preferences is at least as hard as default logic because we are essentially adding a set of defaults on top of a satisfiability problem. It may be harder, because there may be some partial orders on the models that we are unable to capture with a polynomially sized set of defaults. It is well known that brave reasoning and testing if there is a model of a default theory are Σ_2-complete, and that cautious reasoning is Π_2-complete (Gottlob 1992). Given that DPLL is only Δ_2-capable it

clearly cannot solve all instances of default logic, and thus cannot solve all instances of planning with preferences unless we exponentially increase the problem size. All partial orders can be expressed as a set of preferences, but the number of preferences or the conversion to CNF may be exponential in the original size of the problem. The subset of SAT planning with preferences that can be solved with DPLL is the subset where the partial order on models can be expressed as a predetermined branching order, which implies that this is the subset that is in Δ_2.

Since planning in general is PSPACE-complete, people often consider restrictions that place a polynomial bound on the length of a plan. Each action is specified by a set of preconditions and postconditions. One common way to restrict planning problems is to place restrictions on the type of preconditions and/or postconditions that are allowed. Determining whether or not there is a plan for a general planning problem is still PSPACE complete even if we are restricted to two positive preconditions and two postconditions. The problem becomes NP complete when there are no restrictions on the type of preconditions, but that all postconditions must be positive.

Another way to restrict the problem is to define the type of preferences that are allowed. Earlier we described three types of preferences that were defined by Pontelli and Tran (Pontelli and Tran 2004): basic preferences, atomic preferences, and general preferences. Recall that state desires, goal preferences, and action preferences are all basic desires. Also, if φ_1 and φ_2 are basic preferences then so are $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$, always (φ_1), until (φ_1, φ_2), eventually (φ_1), and next (φ_1).

Proposition 5.2.1. Determining whether there is a plan that satisfies some basic preference φ is NP-complete.

Proof: Clearly we cannot reduce the complexity by adding preferences. Planning without preferences is NP-complete and is directly reducible to planning with basic preferences by using an empty or trivial preference. Thus, planning with basic preferences is NP-hard.

One way to demonstrate that a problem is in NP is by giving a polynomial checkable proof of yes instances. A trajectory suffices as a witness because we can check in polynomial time both whether the trajectory is valid and whether it satisfies the basic preference. Since planning with basic preferences is in NP and is NP-hard, it is NP-complete. ■

This proof demonstrates only that it is NP-complete to determine whether or not there is a plan that satisfies the preference. This does not answer the question of how difficult it is to actually find the best trajectory. With regard to a basic preference, the most preferred trajectory is any trajectory that satisfies the preference. If there are none, then all trajectories are equal which also means that all trajectories are "most preferred."

Proposition 5.2.2. Given a basic preference φ, we can determine a most preferred trajectory using a polynomial number of NP oracle queries.

Proof: First we ask the oracle if there is a trajectory that satisfies the preference. If not we ask if there is any trajectory at all. If there is no trajectory then we return false.

Following that, we pick some action *a* and execute it. This results in some new state *s*. If we know that the basic preference can be satisfied, then we ask the oracle if there is a trajectory from this new state *s* to the goal state that satisfies the preference. If we know that the preference cannot be satisfied, we ask whether there is any trajectory from this new state *s* to the goal state. If there is no trajectory from *s* to the goal state, we try a different action. If there is, we fix this as our first action and move on to trying possibilities for the second. We repeat the process until we find a valid plan.

It is important to note that before moving on to the next step in the trajectory, we have fixed all prior actions. We will not need to backtrack because the oracle has verified that there is a plan with the all prior actions fixed. We use the oracle to avoid testing all possible combinations of actions.

There are only a polynomial number of possible actions at any given time step. There are also only a polynomial number of time steps. At worst, we test all possible actions at each possible time step, which still results in only a polynomial number of queries. ■

Recall that atomic preferences are constructed by placing an ordering on a set of basic preferences. The most-preferred trajectory is not the one that satisfies more of the basic preferences, but rather the one that satisfies the higher-ranked preferences. We now

show that atomic preferences can also be solved with a polynomial number of queries to an NP oracle.

Proposition 5.2.3. Given an atomic preference φ, we can determine a most preferred trajectory with a polynomial number of NP oracle queries.

Proof: The atomic preference φ actually represents $\varphi 1 \lhd \varphi 2 \lhd \ldots \lhd \varphi_n$ where $\varphi_1, \ldots, \varphi_n$ are all basic preferences. We can determine which of the basic preferences will be satisfied by a most preferred trajectory as follows: we first ask if there is a plan that satisfies φ_1. If not, we ask if there is a plan that satisfies φ_2. If there is a plan satisfying φ_1, we ask if there is a plan that satisfies $\varphi_1 \wedge \varphi_2$. The conjunction of two basic preferences is a basic preference so it is in NP to ask if there is a trajectory that satisfies $\varphi_1 \wedge \varphi_2$. We continue to evaluate each basic preference in order from φ_1 to φ_n. If there are n preferences, then we need n queries to determine which basic preferences are satisfied by the optimal trajectory.

Once we know which basic preferences are satisfied, we can create a new basic preference by taking the conjunction of these preferences. We used a polynomial number of queries to construct this basic preference. We can then determine a trajectory for our new basic preference using a polynomial number of queries (proposition 5.2.2). Thus we only use a polynomial number of queries overall. ■

The final type of preferences that are allowed are called general preferences. An atomic preference is the most basic type of general preference. If Ψ_1 and Ψ_2 are general preferences then so are Ψ_1 & Ψ_2, Ψ_1 | Ψ_2, and !Ψ_1. An ordered set of general preferences is also a general preference.

The complexity of planning with general preferences is still unknown, thus we have not yet determined whether we can solve planning with general preferences with DPLL.

Our results are particularly interesting because prior to our work, it was already known that an NP algorithm could solve any NP problem. Finding a trajectory for a planning problem with basic preferences or atomic preferences is a more complex problem than SAT, but can be solved with a SAT algorithm.

5.3. Nonmonotonic Reasoning

In this section, we will describe how to use DPLL for one version of nonmonotonic reasoning, and discuss how this helps in analyzing the complexity of that version. We will then talk about the complexity of other versions of nonmonotonic reasoning and which can be solved with DPLL.

Recall that Shoham added nonmonotonicity within various logical frameworks by defining a preference order on the interpretations of a theory so that some are preferred over others. He uses the notation $B \sqsubset A$ to mean that A is preferred over B or equivalently B is less preferred than A.

Adding nonmonotonicity to satisfiability problems increases the complexity for first order logic from NP to NP^{NP} (Cadoli and Schaerf 1993). For further references on the complexity, Cadoli and Schaerf also cite a paper by Stillman (1992) and another by Gottlob and Fermüller (1971).

DPLL works well for SAT problems so it is a natural candidate for solving Nonmonotonic SAT (NSAT) problems. Since DPLL is only Δ_2-capable, whereas NSAT problems are Σ_2-complete, we will not be able to solve all NSAT problems with DPLL. We show that certain NMR problems are solvable by DPLL by demonstrating subsets of NMR that are in Δ_2. Next, we show how to use DPLL for NSAT problems and then we discuss the subset of nonmonotonic reasoning problems that *are* solvable with DPLL.

NSAT problems have a partial order on the interpretations so that some models of a theory are preferred over other models. The first step in developing a solution to NSAT problems is to represent the partial order on interpretations. In mathematics, partial orders are generally represented using lattices. The most straightforward way to represent a lattice as a data structure is to specify which element is preferred for every pair of elements. If neither element is preferred then that pair may be omitted. However, when the elements are all possible interpretations of a SAT problem, this is not a reasonable encoding.

For most real world problems, the partial order on interpretations will have structure that leads to a more compact representation. By exploiting the structure of problems, we can avoid using an exponential encoding for many partial orders.

As an alternative, let a partial order be represented as a set of rules of the form

$L \Rightarrow R$. L and R each represent either true, false, or a CNF formula. The implication is logically equivalent to stating that if L is true then we prefer that R is also true. For rules of the form $true \Rightarrow R$ we can abbreviate by simply writing R.

Let each rule be assigned a priority level so that if there is no model satisfying all of the rules then we prefer to satisfy the most important rules first. The priority level of each rule is a natural number where a lower number corresponds to a higher priority. For instance, we could write one partial order using the following set of rules:

$$0.\ dead\ (Tweety) \Rightarrow \neg fly\ (Tweety)$$

$$1.\ bird\ (Tweety) \Rightarrow fly\ (Tweety)$$

This means we prefer that if *Tweety* is dead then *Tweety* does not fly, and that if *Tweety* is a bird then *Tweety* does fly. However, if *bird (Tweety)* and *dead (Tweety)*, we cannot satisfy both rules. In this case we would rather satisfy the rule of priority 0 and conclude that since *Tweety* is dead, *Tweety* does not fly.

Given that two models break different sets of rules with mixed priorities, it is not always obvious which model is preferred. Let $Broken_i(A)$ denote the set of rules A breaks at priority level i. For two models A and B, let us define that $B \sqsubset A$ iff:

$$\exists i\ \ Broken_i(A) \subset Broken_i(B) \text{ and}$$
$$\forall\ j<i\ Broken_j(A) = Broken_j(B)$$

In other words, we start by comparing two models at priority level 0. If A breaks a subset of the rules that B breaks, then $B \sqsubset A$. If they break different sets of rules at priority 0 then the two models are incomparable. If the two models break the same rules then we move on to the next priority level and repeat the process.

Based on our definition, if two models are incomparable at priority level 0 then they are incomparable regardless of which rules they break of lower priority. Note also that the number of rules that a model breaks is inconsequential. For example, if A breaks two rules of priority 0 and B breaks a different rule of priority 0 then the two models are incomparable even though B breaks fewer rules.

A set of rules corresponds to a unique partial order, but there may be multiple sets of rules that correspond to the same partial order. Since all partial orders can be represented as a set of CNF formulas, this format may be used to represent any partial order.

Now that we have a notation for representing partial orders, we are ready to move on to the next step. The standard DPLL procedure creates a search tree where the leaves of the tree correspond to a total assignment of variables. The internal nodes are partial assignments. If the standard procedure were used on NSAT problems, then a model of the theory would be returned if one exists, but there is no guarantee that it will be a preferred model.

It requires exponential time to expand the entire search tree to find a preferred model of the theory. Instead, we would like to rearrange the branch points so that the leaves are ordered from most preferred to least preferred. That way, the first model we

find would be guaranteed to be a preferred model. Determining the branch points a priori from the partial order corresponds to creating a static variable ordering.

One important observation is that if we branch on x_1 being true, but x_1 is false in all preferred models, then we have already made a mistake. The left side of the tree is expanded first, so we will find a less preferred model before the preferred models can be expanded. In order for the branch points to correctly order the interpretations, each one must divide the models into two sets where all of the models in one set are preferred over or incomparable to all of the models in the second set.

Proposition 5.3.1 For some sets of rules, there is no static variable ordering such that a preferred model is always chosen.

Proof: Suppose that the partial order consists of the following simple set of rules both of priority 0:

$$0.\ a \Rightarrow b$$
$$0.\ \mathrm{c} \Rightarrow \mathrm{d}$$

Using the previous observation, we can show that there is no way to make the branch decisions such that a preferred model is always chosen. We have four choices for which variable to branch on first. Suppose we branch on b and consider the formula $(\neg b \vee c) \wedge (\neg b \vee \neg d)$. Models in which b is true break the second rule, so the models that satisfy both rules are in the right side of the search tree. Suppose instead that we

branch on $\neg b$ and consider the formula $(a \vee b)$. There are models in which b is false, but the preferred models are in the right side of the tree. In either case, we find a non-preferred model before the preferred models can be expanded. A similar argument can be made for whichever variable we try to branch on first. ■

Even for some simple cases, we are unable to create a static ordering of the variables that will produce a search tree that always finds a preferred model. Now consider the same partial order used in the proof, but allow the introduction of new variables. Let $Rule1 = \neg a \vee b$. The rules become:

$$0.\ Rule1$$

$$0.\ c \Rightarrow d$$

We can branch on setting *Rule1* to true and try to find an assignment that will not break either rule. See figure 5.3.1 for a partial search tree.

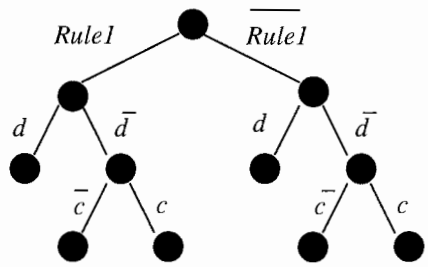

Figure 5.3.1 Partial search tree for NSAT problem where we branch on *Rule1* first.

To ensure that the substitution is meaningful, we add $Rule1 \equiv \neg a \vee b$ to the theory. In general, we can introduce new variables where each variable represents either a conjunction or disjunction of literals. Suppose we introduce a variable w which represents a disjunction $x_1 \vee \ldots \vee x_n$. The clauses we introduce are: $(\neg w \vee x_1 \vee \ldots \vee x_n) \wedge (w \vee \neg x_1) \wedge \ldots \wedge (w \vee \neg x_n)$. There is one clause of length $n+1$ and n clauses of length two. Suppose instead that we introduce w which represents a conjunction $x_1 \wedge \ldots \wedge x_n$. The clauses we add are: $(\neg w \vee x_1) \wedge \ldots \wedge (\neg w \vee x_n) \wedge (w \vee \neg x_1 \vee \ldots \vee \neg x_n)$. Thus if every substitution that we introduce represents either a conjunction or a disjunction of literals then the expression can easily be converted to CNF with a polynomial-time transformation.

To process the set of rules such that all substitutions are either a conjunction or a disjunction of literals, we first introduce one new variable per clause. Since every clause is a disjunction of literals the substitutions are the required format. Following this step, each rule is of the form $LClause_1 \wedge \ldots \wedge LClause_n \Rightarrow RClause_1 \wedge \ldots \wedge RClause_m$. Next, for each side of the rule we introduce one substitution, representing a conjunction of literals. During this step, each rule is transformed into the form $Left \Rightarrow Right$. Finally, we reduce each rule to a single variable by introducing the substitution $Rule = \neg Left \vee Right$, which again matches the desired format.

For example, consider the rules:

0. $dead(Tweety) \vee in\_concrete(Tweety) \Rightarrow \neg fly(Tweety)$

1. $bird(Tweety) \Rightarrow fly(Tweety)$

For simplicity, we do not need variables to represent clauses that have only one literal. In the first step the only substitution we need to introduce is $Incapacitated(Tweety) = dead(Tweety) \vee in\_concrete(Tweety)$. This substitution represents the conditions that would cause *Tweety* to be incapable of flying, such as being dead or cemented in concrete.

We do not need a variable to represent a side that has only one clause so we can skip the second step. For the final step, we introduce the substitutions $Rule1 = \neg Incapacitated(Tweety) \vee \neg fly(Tweety)$ and $Rule2 = \neg bird(Tweety) \vee fly(Tweety)$. Note that if either rule only had one side we would not have needed a variable to represent it.

The preprocessing step that we propose involves introducing substitutions, adding the corresponding information to the theory, and converting the information to CNF.

Proposition 5.3.2 If the representation of the partial order is polynomial in size, then the preprocessing step runs in polynomial time.

Proof: Suppose that the representation of the partial order is polynomial in size. For each rule we introduce at most one variable per clause, one for each side, and one for the rule itself. Since the rules are polynomial in number and in length, the substitutions must be also.

The expressions we add to the theory are the substitutions that we introduce. Thus the information added to the theory is polynomial and can be constructed in polynomial time. Each substitution represents either a disjunction or a conjunction of literals, so we can efficiently convert each substitution to CNF. Therefore, the preprocessing step only requires polynomial time. ■

In the DPLL procedure, we first want to branch on the variables that we created to represent the rules in order of priority. Afterwards we can use other well known heuristics to select the remaining order dynamically. Some of the best heuristics known can be found in the work on ZChaff (Zhang and Malik 2002).

Note that in creating the variable ordering we have imposed a total order on the rules used to express the partial order. Since the total order on the rules respects the original partial order, this will not cause us to return a less preferred model. It may cause two previously incomparable interpretations to no longer be incomparable, but it will not alter any current preferences. In other words, it may eliminate some preferred interpretations, but this is not an issue because the algorithm will still return a preferred model. If there are multiple rules within a priority level, then there is more than one valid static variable ordering.

A variable order can be converted to a set of rules by writing one rule per variable, where some of the variables may represent substitutions. Again, we require that all substitutions are either conjunctions or disjunctions of literals. If the substitutions in

the ordering are polynomial in number and in size then we say that the static variable ordering is polynomial.

Proposition 5.3.3 A polynomial, static variable ordering for a partial order can be converted to a polynomially sized representation of the partial order in our format.

Proof: We showed in the proof for Proposition 5.3.2 that the information being added to the theory is polynomial in size for each substitution. So only a polynomial amount of information is added to the theory for the substitutions. For each variable in the ordering, we write a rule indicating our preference for the value of that variable. The modified theory and the set of rules are polynomial in size. ■

Proposition 5.3.3 implies that if we can create a polynomial static variable ordering for an NSAT problem, then our format is a sufficient representation for the partial order on interpretations.

The only difference between the algorithm for SAT problems and NSAT problems is the procedure for determining the next branch variable. In SAT solvers, any branching heuristic may be used. To solve NSAT problems we must branch on the next rule variable available. If there are no rule variables left then we may use any branching heuristic to complete the branching order.

This provides insight into the complexity of NSAT problems because although we know that the general case is Σ_2-complete, there were no previous characterizations of

any subsets of NSAT problems with lower complexity. Our work demonstrates that any NSAT problem that can be expressed in the format that we described must be in Δ_2 since those problems are solvable with DPLL.

The NMR formalism that was proposed by Shoham and the work that we have done is quite similar to recent work done on SAT planning problems with preferences (Giunchiglia and Maratea 2007). Adding preferences introduces nonmonotonicity into planning problems as it is analogous to adding a set of defaults. Giunchiglia and Maratea essentially encode a partial order on the models and the goal is to return a maximal model. Shoham's work was more theoretical and did not mention a way of encoding the partial order on models. Giunchiglia and Maratea developed a practical approach to solving problems of this type.

The method that Giunchiglia and Maratea use to solve SAT planning problems with preferences begins with a list of simple preferences and creates one new variable $v(p)$ for every preference p. They add the necessary clauses to the theory to enforce that $v(p) = p$. In order to indicate that we prefer plans in which the preferences are true, we branch on the variables $v(p)$ in order of the priority of the preferences. If two preferences are equally preferred then we can branch on them in either order. The model that is produced will be optimal in the sense that if it fails to satisfy a preference, then every other model is either equally preferred or fails to satisfy a different preference of higher priority. The solver that they developed works quite well and they show addition of preferences has a relatively small effect on the overall runtime (though in general they are using a small number of simple preferences). These results match those that we showed

in chapter IV, that introducing a fixed branching order did not have a substantial impact on runtime for satisfiable instances. Their method is nearly identical to the one that was concurrently developed by us, but we have chosen to include both here.

Giunchiglia and Maratea recognized that not all partial orders on models can be encoded using a set of simple preferences that is polynomial in the original problem size (Giunchiglia and Maratea 2006). They show how to solve SAT planning problems with simple preferences. Our work demonstrates why DPLL cannot solve all cases with more complex preferences and helps us to determine which can be solved with DPLL.

We have shown one use of DPLL for nonmonotonic reasoning. Next we will discuss the complexity of various problems in NMR and how it relates to the capability of DPLL. There have been a number of different studies on the computational complexity of nonmonotonic reasoning. Certain cases are easy to solve while others are not even computable. Many of the cases that can be computed are beyond NP. Since DPLL is only Δ_2-capable, it can only solve the subset of NMR that falls in Δ_2. This rest of this section is a survey of the complexity results known for nonmonotonic logics, indicating which versions can be solved with DPLL.

There are many complexity results known for nonmonotonic reasoning. Eiter et al. have written a paper outlining complexity results for answer set programming (2004). They present some new results as well as a survey of existing results. Brave reasoning in general is known to be Σ_2 complete and cautious reasoning is Π_2 complete. In both cases the complexity becomes Δ_3 complete when we add ***weak constraints***. A weak constraint is one that is satisfied if possible, but does not have to be true. This is equivalent to

preferences. Given the complexities of these problems, DPLL is not capable of solving the general cases of these problems. It is also beyond the scope of DPLL to solve answer set programs where disjunction is allowed in the body of a rule. Programs that contain no atoms of the form "not x" have a lower complexity and can be solved with DPLL.

One particular subset of answer set programming that is known to be exactly Δ_2 complete is both brave and cautious reasoning in stratified normal logic programs with or without weak constraints (Eiter et al. 2004). A program is stratified if we can assign a value $s(l)$ to each literal l such that if l is an element of the body and l' is an element of the head, then $s(l) \leq s(l')$ if l is positive and $s(l) < s(l')$ if l is negative. Also if l and l' are both elements of the head then $s(l) = s(l')$. The logic program is normal in the sense that the body is disjunction-free. This subset of nonmonotonic reasoning is exactly the type of problem that we would expect DPLL to perform well on because it is at the outer limits of the scope of DPLL, but it is not one that is often considered.

Default logic is also generally of a higher complexity than SAT. The problem of finding an extension of a default theory and the problem of brave reasoning are Σ_2-complete (Gottlob 1992, Stillman 1992). Cautious reasoning is Π_2-complete (Gottlob 1992, Stillman 1992). The results on brave and cautious reasoning hold even when restricted to normal default theories that are prerequisite free. Recall that a normal default theory is one in which all defaults are normal and a normal default is one in which the justification is the same as the consequence. A normal default is of the form:

$$\frac{\alpha : \mathrm{M}\, w}{w}$$

Model checking involves deciding whether a propositional interpretation is a model of any extension of the theory. Model checking for normal default theories was shown to be Θ_2-complete, and is coNP-complete when the defaults are also prerequisite free (Baumgartner and Gottlob 1999). Since this is within Δ_2, model checking in normal default logic can be solved with DPLL.

Many real-world problems in default logic can be expressed using only normal defaults. Since DPLL is capable of solving normal default logic, it is a good candidate for solving a number of real-world problems involving default reasoning.

Model checking in normal default logic is within the scope of what DPLL can handle, so we now describe a method for solving this problem using a single call to DPLL. In the model checking problem we are given a default theory $\Delta = <D, W>$ and an interpretation M. We want to know if M is a model of any extension of the theory. First we check if M satisfies W and return false if not. This can be done easily in polynomial time. Then let B be the set of defaults whose consequences conflict with M and G be the remaining defaults. Thus, G is our set of "good" defaults and B is our set of "bad" defaults. We want to know if there is an extension that only uses defaults in G.

For each default, we add a variable *a* to represent the prerequisite and a variable *w* to represent the consequence (and necessary clauses so that these variables have the intended meaning). We will also add the variable *consistent*(*w*) to indicate whether or not the variable is consistent. We do not need to add additional clauses for these variables; the reason for this will be explained later.

Baumgartner and Gottlob (1999) proved that for a normal default theory there is at most one extension E such that M $\models$ E, or equivalently there is at most one subset of G that forms a valid extension. Roughly speaking, this is because if the defaults in G do not conflict with M then it is impossible for them to conflict with one another. This makes it easy to break the problem into two parts: in the first phase find the subset of G such that it creates a valid extension of <W,G>, and in the second phase we check if we are required to add any defaults from B to find an extension of <W,D>.

To compute the first phase we are going to start with the SAT formula which represents the conjunction of all formulas in W converted to CNF notation. Next, we add $(\neg a_i \vee \neg consistent(w_i) \vee w_i)$ for each default $d_i \in$ G. This will ensure that if a prerequisite is true and w_i is consistent then we will apply the default. We do not need to add extra clauses to ensure that our consistency variables have meaning because if the prerequisite is true then the consequence must be consistent in order to set it to true. If a prerequisite must be true and the consequence cannot be true, then we mark the consequence as inconsistent. If the prerequisite is not inferred then it does not matter whether we mark the consequence as consistent or inconsistent. The reason that the consistency variables are necessary is that adding them ensures that we will not automatically return false when a prerequisite is forced to be true and the consequence is inconsistent.

The final step in the process is to create the branch order. For each default $d_i \in$ G we branch on $\neg a_i$ so that we will not apply that default unless we have to, and then on

consistent(w_i) so that we will apply the default if it is possible to do so. It does not matter which order we consider the defaults in because the defaults do not conflict with one another.

For the second phase, we want to subscript our variables with phase2 so that we do not get unintended interactions between computing the first part and the second part. For the second phase we want to start with the SAT formula corresponding to W (where variables are subscripted with phase2) and add the variables for all of the defaults (also subscripted with phase2). Next we add clauses that correspond to ($w_i = w_{i,\text{phase2}}$) so that any consequences that we added in the first phase have been appropriately added in the second phase. Similar to the first phase, we add ($\neg a_i \vee \neg$*consistent*(w_i) $\vee w_i$) for each default $d_i \in$ B. Also as before, for each default in B we try branching on $\neg a_i$ so that we will not apply that default unless we have to, and then on *consistent*(w_i) so that we will apply the default if it is possible to do so. If we are able to apply any defaults in B then we return false.

This algorithm works except for in one special case. During either phase we may have two defaults such that the prerequisite of one is α and the prerequisite of the other is $\neg\alpha$. If neither α nor $\neg\alpha$ can be inferred from W, then we cannot apply either default. However, when we branch on $\neg\alpha$ for the first default, we will then conclude that $\neg\alpha$ can be inferred and we will apply the other default. In order to prevent this we can subscript the variables in W and for each individual default to ensure that there are no interactions except for the ones that we want. We enforce the interactions that we do want by creating

additional clauses that say that if a consequence was added by a previous default then it applies in all steps of the procedure. This is fairly straightforward to accomplish by adding clauses of the form $(w_{i,j} = w_{i,k})\ \forall\ i, j, k$. The branch order for each phase does not change (except that now the variables are subscripted).

In order to solve this with DPLL we create a single SAT formula by taking the conjunction of the formulas from the two phases, use our branch order from the first phase, followed by all remaining variables from phase one, then our branch order from the second phase. Phase one is computed first because we branch on those variables first. We may perform some unit propagations, but unit propagations only prune sections of the search space that do not contain valid models, and thus will not affect the outcome. If w_i is true for any default in B we return false and we return true otherwise.

Proposition 5.3.4. The above procedure will correctly determine whether or not there is an extension that is consistent with the interpretation M.

Proof: There are two claims that must be proven in order to prove this proposition. The first is that the method generates a valid extension. The second is that M is consistent with the extension.

An extension is a deductively closed set of beliefs about the world. This means that everything in W is true and we apply a default whenever it is possible to do so. It also means that we do not apply defaults unless we are required to do so. If a prerequisite is true and the consequence is consistent then we are required to set the consequence to

true to satisfy the clause ($\neg a \vee \neg consistent\ (w) \vee w$), because we set our consistency variables to true initially and only set them to false when we have a reason to believe otherwise. We also try branching on prerequisites being false, thus we do not use defaults unless we are required to do so. Therefore we are generating a valid extension.

We checked that the original set of formulas is consistent with the interpretation before we started. We return false if we use any of the defaults that conflict with M. Thus if we return true then we were able to find a set of defaults that generate an extension and do not conflict with the interpretation M. ■

Now that we have discussed normal defaults, we will move on to another popular form of default logic. A default is called *semi-normal* if it is of the form:

$$\frac{\alpha : M\ \beta, w}{w}$$

Even though model checking becomes solvable with DPLL when restricted to normal default theories, it is still Σ_2-complete if all of the defaults are semi-normal (Gottlob 1992). The complexity of credulous reasoning and brave reasoning is still beyond Δ_2 even if all of the defaults are normal and prerequisite free (Cadoli and Schaerf 1993). The prerequisites are specified by α so a default that is normal and prerequisite free simply says "if w is possible, then assume w." These restrictions are all still beyond what DPLL is capable of solving.

CHAPTER VI

CONCLUSION

In this thesis, we have defined the notion of algorithm capability and shown the capability of two well known algorithms. Specifically we showed that DPLL is exactly Δ_2 capable and that the algorithm most commonly used for solving QBF is exactly PSPACE capable. In both cases, we proved the results by showing both upper and lower bounds on the capability. We can show that a complexity class is a lower bound on the capability of an algorithm by showing that the algorithm can solve a complete problem for that complexity class. This works because every problem in the class is efficiently reducible to the complete problem and the complete problem is efficiently reducible to the algorithm. We can prove the upper bound on the capability of an algorithm by showing that if a problem can be solved by the algorithm, then it must be in the complexity class.

In order to prove the capability of DPLL it was necessary to know a Δ_2 complete problem. Krentel proved indirectly that the Odd Maximum Satisfiability problem is Δ_2 complete, so we provided a more direct proof involving a Cook-style reduction.

We provided two main applications in Artificial Intelligence to demonstrate how this notion is useful. Specifically we showed how the capability of DPLL applies to planning problems and to nonmonotonic reasoning. Our theoretical framework was useful in developing a faster solution for optimal planning that runs up to an order of

magnitude faster than Satplan on a variety of problems taken from the IPC-5 benchmarks. Satplan makes multiple calls to a SAT solver, discarding learned information with each iteration. We prove that the complexity of optimal planning is Θ_2 so it can be solved with a single call to a SAT solver. By using a single call we automatically retain learned information.

Knowing the capability of DPLL is also useful in nonmonotonic reasoning. It enabled us to characterize several subsets of NMR that are solvable with DPLL with fixed branching. In addition we showed how to use DPLL in order to solve nonmonotonic satisfiability problems and propositional model checking in normal default logic.

There are several possible ways in which this work can be extended. We discussed several optimizations that were added to DPLL and proved that certain ones do not affect the capability of the algorithm. For others, we speculated that when they are added in to DPLL that the resulting algorithm is no longer Δ_2-Capable, but we have not yet proven that there is any DPLL-based SAT solver that is not Δ_2-Capable. The most likely to be incapable of solving problems in Δ_2 would be the version of the algorithm that returns SAT or UNSAT instead of the satisfying assignment, but this has not been proven.

Another interesting direction would be to implement an answer-set solver for the subset of ASP that is in Δ_2 and compare the performance to other popular solvers. It would also be potentially useful and interesting to develop a program based on DPLL for determining whether there is an extension of a normal default theory.

Other ways to extend the ideas presented here would be to investigate more thoroughly the set of problems that fall into Δ_2 and can thus be solved with a SAT solver. It would also be interesting to determine the capability of other useful algorithms and use the results to test how those algorithms perform on additional problems.

BIBLIOGRAPHY

Agrawal, M., N. Kayal, and N. Saxena. 2004. PRIMES is in P. *Annals of Mathematics* 160:781-93.

Baumgartner, R. and G. Gottlob. 1999. On the Complexity of Model Checking for Propositional Default Logic: New Results and Tractable Cases. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*: 64-9.

Blum, A. and J. Langford. 1997. Fast Planning Through Plan Graph Analysis. *Artificial Intelligence* 90:281- 300.

Bylander, T. 1991. Complexity Results for Planning. *Proceedings of the International Joint Conference on Artificial Intelligence* 1:274-9.

Cadoli, M., A. Giovanardi, and M. Schaerf. 1998. An Algorithm to Evaluate Quantified Boolean Formulae. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*: 262-7.

Cadoli, M. and M. Schaerf. 1993. A Survey on Complexity Results for Non-monotonic Logics. *Journal of Logic Programming*: 17:127-60.

Complexity Zoo. http://www.complexityzoo.com (accessed November 18, 2008).

Cook, S. A. 1971. The Complexity of Theorem Proving Procedures. *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*: 151-8.

Crawford, J., M. Ginsberg, E. Luks, and A. Roy. 1996. Symmetry-Breaking Predicates for Search Problems. *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*: 148-59.

Davis, M., G. Logemann, and D. Loveland. 1962. A Machine Program for Theorem Proving. *Communications of the ACM* 5, 7:394-7.

Davis, M. and H. Putnam. 1960. A Computing Procedure for Quantification Theory. *Journal of the ACM* 7, 1:201-15.

Dixon, H. 2004. Automating Pseudo-Boolean Inference Within a DPLL Framework. PhD Dissertation. University of Oregon.

Eén, N. and N. Sörensson. 2004. An Extensible SAT Solver. *Lecture Notes in Computer Science*: 333-6.

Eiter, T., W. Faber, M. Fink, G. Pfeifer, and S. Woltran. 2004 Complexity of Answer Set Checking and Bounded Predicate Arities for Nonground Answer Set Programming. *9th International Conference on the Principles of Knowledge Representation and Reasoning*: 377-87.

Eiter, T. and G. Gottlob. 2000. Complexity Results for Some Eigenvector Problems. *International Journal of Computer Mathematics* 17(1-2): 59-74.

Eiter, T. and T. Lukasiewicz. 2000. Default Reasoning from Conditional Knowledge Bases: Complexity and Tractable Cases. *Artificial Intelligence* 124(2): 169-241. 2000.

Fikes, R. and N. Nilsson. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2:189-208.

Garey, M. and D. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company. New York.

Gelfond, M. and V. Lifschitz. 1988. The Stable Model Semantics for Logic Programming. *Proceedings of the Fifth International Logic Programming Conference and Symposium*: 1070-80.

Ginsberg, M. 1987. *Readings in Nonmonotonic Reasoning*. California: Morgan Kaufmann Publishers, Inc.

Giunchiglia, E., Y. Lierler, and M. Maratea. 2004. Cmodels2: SAT-Based Answer Set Programming. *Proceedings of the Nineteenth International Conference on Artificial Intelligence*: 61-6.

Giunchiglia, E. and M. Maratea. 2006. Solving Optimization Problems with DLL. *Proceedings of the 17th European Conference on Artificial Intelligence*: 377-81.

Giunchiglia, E. and M. Maratea. 2007. Planning as Satisfiability with Preferences. *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI)*: 987-92.

Goldszmidt, M. and J. Pearl. 1996. Qualitative Probabilities for Default Reasoning, Belief Revision, and Casual Modeling. *Artificial Intelligence* 84(1-2): 57-112.

Gottlob, G. 1992. Complexity Results for Nonmonotonic Logics. *Journal of Logic and Computation* 2(3):397-425.

Gottlob G. and C.G. Fermüller. 1993. Removing Redundancy from a Clause. *Artificial Intelligence Volume 61 Issue 2*: 263-89.

Gnuplot homepage. www.gnuplot.info (accessed April 20, 2008).

Grosse, A., J. Rothe, and G. Wechsung. 2001. Relating Partial and Complete Solutions and the Complexity of Computing Smallest Solutions. *Proceedings of the Seventh Italian Conference on Theoretical Computer Science*: 339-56.

Huang, J. 2007. A Case for Simple SAT Solvers. *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming*: 839-46.

Janhunen, T. and I. Niemala. 2004. GnT – A Solver for Disjunctive Logic Programs. *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning*: 331-5.

Kautz, H. and B. Selman. 1992.Planning as Satisfiability. *Proceedings of the Tenth European Conference on Artificial Intelligence*: 359-63.

Kautz, H., B. Selman, and J. Hoffman. 2006. Satplan: Planning as Satisfiability. *Abstracts of the Fifth International Planning Competition.*

Kautz, H., D. McAllester, and B. Selman. 1996. Encoding Plans in Propositional Logic. *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*: 374-84.

Krentel, M. 1988. The Complexity of Optimization Problems. *Journal of Computer and System Sciences*, 36:490-509.

Krentel, M. 1992. Generalizations of OptP to the Polynomial Hierarchy. *Theoretical Computer Science,* 97:183-98.

Leone, N., G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499-562.

Liberatore, P. and M. Schaerf. 1998. The Complexity of Model Checking for Propositional Default Logics. *Proceedings of the Thirteenth European Conference on Artificial Intelligence*: 18-22.

Lifschitz, V. 2002. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138:39-54.

Lifschitz, V. Unpublished Draft. Introduction to Answer Set Programming. 2005.

Lin, F. and Y. Zhao. 2004. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artificial Intelligence*, 157(1-2): 115-37.

Lucas, É. 1883. *The Tower of Hanoï.* Paris: GAUTHER-VILLARS, printer of the Académie des Sciences and the Ecole Polytechnique Quai des Augustins, 55.

Marek, V. and M. Truszczynski. 1999. Stable Models and an Alternative Logic Programming Paradigm. *The Logic Programming Paradigm: a 25 Year Perspective*: 375-98.

McCarthy, J. 1980. Circumscription – A Form of Nonmonotonic Reasoning. *Artificial Intelligence*, 13:27-39.

McDermott, D and J. Doyle. 1980. Non-monotonic Logic I. *Artificial Intelligence*, 13:41-72.

Meyer, A. and L. Stockmeyer. 1972. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. *Proceedings of the 13th IEEE Symposium on Switching and Automata Theory*: 125–9.

Moore, R. C. 1985. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence* 25:75-94.

Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: Engineering an Efficient SAT Solver. *Proceedings of the Design Automation Conference*: 530-5.

Nabeshima, H., T. Soh, K. Inoue, and K. Iwanuma. 2006. Lemma Reusing for SAT based Planning and Scheduling. *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*: 103-13.

Niemela, I. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence* 25:241-73.

Nudelman, E., A. Devkar, Y. Shoham, K. Leyton-Brown, and H. Hoos. 2004. SATzilla: An Algorithm Portfolio for SAT. SAT Competition 2004 – Solver Description.

Papadimitriou, C. 1994. *Computational Complexity*. Boston: Addison Wesley Publishing Company.

Pipatsrisawat, K. and A. Darwiche. Technical Report D153, Automated Reasoning Group, Computer Science Department, University of California, Los Angeles. RSat 2.0: SAT Solver Description. 2007.

Pontelli, E.and S. Tran. 2004. Planning with Preferences Using Logic Programming. *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning*: 247-60.

Reiter, R. 1980. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81-132.

SAT Competitions. www.satcompetition.org (accessed July 27, 2008).

SATLIB. http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html (accessed September 10, 2006).

Shoham, Y. 1987. A Semantical Approach to Nonmonotonic Logics. *Readings in Nonmonotonic Reasoning*: 227-50.

Simons, P. 2000. Extending and Implementing the Stable Model Semantics. PhD Dissertation, Helsinki University of Technology Laboratory for Theoretical Computer Science.

Sipser, M. 1997. *Introduction to the Theory of Computation*. Massachusetts: PWS Publishing Company.

Stillman, J. 1992. The Complexity of Propositional Default Logics. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*: 794-9.

Stockmeyer, L. 1976. The polynomial hierarchy. *Theoretical Computer Science*, 3:1-22.

Stockmeyer, L. 1987. Classifying the Computational Complexity of Problems. *Journal of Symbolic Logic*, 52,1:1-43.

Xing, Z., Y. Chen, and W. Zhang. 2006. MaxPlan: Optimal Planning by Decomposed Satisfiability and Backward Reduction. *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*: 53-6.

Zhang, L. and S. Malik. 2002. The Quest for Efficient Boolean Satisfiability Solvers. *Lecture Notes in Computer Science*: 313-31.