Investigations into Satisfiability Search

Andrew Slater

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

January 2004

Typeset in Palatino by TEX and LATEX 2_ε.

Except where otherwise indicated, this thesis is my own original work.

Andrew Slater
17 August 2002

Acknowledgements

There are many people to whom I am grateful for guidance and support during the course of this work. First and foremost I would like to thank my supervisors Rajeev Goré and John Slaney for their guidance and discussions and for teaching me many interesting things. My sincere gratitude also goes to Toby Walsh, John Lloyd, Bob Meyer, Daniel Le Berre and Matthias Fuchs for their helpful discussions and feedback on my research. I am also very grateful to the numerous members of staff and students of the Research School of Information Sciences and Engineering and the Department of Computer Science who helped me along the way. It has, undoubtedly, been an excellent experience.

My gratitude also goes to the Australian National University for providing a scholarship enabling me to perform this research.

Thanks to my fellow students of reasoning with whom I have had many interesting discussions, Kal, Nic and Paul. I have also appreciated the company of all the other computer and coffee geeks, imbibing coffee whilst discussing theses and all things digital.

Finally I thank all my family and friends for their support during the course of this study. My heartfelt thanks goes to Libby, who let me stay up late, and Alia who coloured in drafts for me. I thank everyone in my family for their continual support and encouragement. Additional thanks to my parents Susan and Wayne for their acronyms and chocolate, amongst many other things, and to my other parents Helen and Brendan for letting me use the cool 122, also amongst many other things. Thanks to my friends who were always supportive and also happily listened to my sometimes obscure discourse.

Thanks be to God.

∞

Abstract

In this dissertation we investigate theoretical aspects of some practical approaches used in solving and understanding search problems. We concentrate on the Satisfiability problem, which is a strong representative from search problem domains. The work develops general theoretical foundations to investigate some practical aspects of satisfiability search. This results in a better understanding of the fundamental mechanics for search algorithm construction and behaviour. A theory of choice or branching heuristics is presented, accompanied by results showing a correspondence of both parameterisations and performance when the method is compared to previous empirically motivated branching techniques. The logical foundations of the backtracking mechanism are explored alongside formulations for reasoning in relevant logics which results in the development of a malleable backtracking mechanism that subsumes other intelligent backtracking proof construction techniques and allows the incorporation of proof rearrangement strategies. Moreover, empirical tests show that relevant backtracking outperforms all other forms of intelligent backtracking search tree construction methods. An investigation into modelling and generating world problem instances justifies a modularised problem model proposal which is used experimentally to highlight the practicability of search algorithms for the proposed model and related domains.

Contents

Chapter 1

Introduction

This thesis is concerned with the theoretical nature of some fundamental but practical aspects of the search and the satisfiability problem. This chapter provides a brief informal overview of search and the satisfiability problem, followed by a formal description of the satisfiability problem and a basic overview of research related to the work presented in this thesis. In depth reviews of related research are given in each chapter.

1.1 An Informal Description

The following discussions introduce the concepts behind search and satisfiability, and illustrate the richness and diversity of the research in these areas.

1.1.1 Searching for an Answer

A *search problem* may be simply classified as one which requires us to actually "search" for the answer. The problem in question may have no obvious method which may be followed to determine a solution, other than to intelligently search through all possible solutions, the *search space*, until one is found. Typically we may have an efficient way of determining whether one of the candidate solutions is actually correct, but no efficient way of determining how to find a correct solution. We may also want to know whether there is a solution at all. There are many such problems, both theoretically and practically motivated, but they all have these difficulties in common.

Consider a simple example where we wish to know whether a number n can be factored into two different numbers: $n = ab$. We are asking whether n is non-prime. It is simple to check a solution by simply multiplying the two factors to see if the product is n, but we may have to test whether n is divisible by a for a great many number of possible values for a. This is a special kind of search problem in itself, and a detailed logical system related to this appears in [86]. In practice however we would exploit properties of the factoring problem itself to find an answer efficiently

[83], but the problem remains as a challenge for the area of satisfiability testing [22]. In general when we consider the checking or verification of a solution to be "efficient" we are saying that the number of steps required by this process is bounded by some polynomial function. In contrast, the size of the search space is more likely to be defined by an exponential function. (For factorisation, the search space is exponential due to the number of bits required to represent the product, and search space of total possible factors may be simply bounded by two to the power of that number.)

More typically we consider problems that may not be so readily exploited by numeric properties. We may encounter "real world" problems that have a rich complexity which makes analysis difficult, but for which we want to find solutions. These problems include theorem-proving, electronic circuit diagnosis, and various planning problems. The real world domain of such search problems is discussed further in Section 1.4, but for the moment we point out that, in general, a search problem has a derivable equivalence to another and that, in some cases, solving the equivalent problem can be more straightforward than solving the original. This is not because the second problem is any easier, but because there is a better understanding of how to solve instances of the second type of problem. The techniques that are used in solving a particular kind of search problem often also occur within the domain of other kinds of search problems. The successes in one field of research are soon translated to similar fields.

While we can view search problems as some kind of generic whole, there are some important differences. In this dissertation we are mainly concerned with those problems for which we can actually determine whether or not there is a solution. These kinds of problems are called *decidable*. There are problems for which this property does not hold, for example theorem proving in first order logic. The addition of a possibly infinite search space means that we cannot always cover the search space required to determine an answer. Although there are similarities in techniques used to solve both decidable and undecidable problems, there are often better techniques that are specialised to deal with the problem domains which manage infinite search spaces, e.g. [113, 37, 114]. Much of the class of decidable search problems can be characterised by their membership in the set NP, or NPC, as defined from the field of complexity theory. These sets characterise complexity classes that classify the potential difficulty of a problem [39]. The class of problems in NPC are NP Complete, and these are at the top of the difficulty hierarchy in the class NP. The Satisfiability problem is a prototypical NP-Complete problem and was in fact the first problem to be proved as NP-Complete [21]. Further discussion of the complexity of satisfiability appears in Section 1.3.3. The satisfiability problem may be used as a vehicle for studying search problems in general and, while other problems may be expressed as instances of satisfiability, one may also extract techniques used to solve satisfiability problems so that they may be applied directly to problems in other domains.

1.1.2 Satisfiability Problems

The general problem of satisfiability is to find some solution to a given set of requirements such that the solution satisfies each and every requirement in at least one way. Consider a simple example where one desires to organise a party. Every friend invited has a list of requirements for the party. They may include things like having loud music or not having loud music, having lemonade or not having lemonade, or allowing window breakages or not allowing window breakages. Each friend will agree to come to the party if at least one item on their list of requirements is met. The question is then: Can one organise a party where every friend will attend? Solving this problem may be straightforward. Perhaps everyone invited has asked for free beer on tap. On the other hand there may be a large guest list, considerably many options, and many disagreements amongst the guests. An efficient and popular approach to finding a solution is to logically construct an answer and test it. We may start with the assumption that fire-breathing be allowed, then after making further assumptions and finding no possible solution, deduce that fire-breathing events cannot be held. We then continue with this new information. At every stage of the solution construction we are performing the method of assumption and test. When an assumption seems valid, we continue assuming until we have satisfied everyone, or we have identified that someone has been left out, and must therefore change one of our assumptions. A methodical ordering of assumptions and corresponding deductions ensures that we check all possibilities in the event we cannot find a solution. The problem of solving satisfiability is discussed in Section 1.1.3.

Apart from organising parties, satisfiability has been shown to have a wide range of practical applications in other areas such as microprocessor design [109], planning [62, 63, 61] and theorem proving [116, 117]. The application of techniques for satisfiability solving to "real world" problem domains is reviewed in Section 1.4. Later in this work we question whether the problems from the "real world" have some common characteristics that would enable us to better understand search problems and more effectively use the techniques for solving them.

1.1.3 Solving Satisfiability

As discussed above, the most efficient general method of mechanically solving satisfiability problems is to search for a solution by successive assumptions and possible deductions until they lead to an obviously recognisable contradiction or satisfying assignment. This method is also referred to as *proof by refutation*. The "splitting" nature in such searches, where both the effect of an assumption that something is true and its corresponding negation might be explored, accounts for the possible exponential cost in finding a solution. In between assumptions we can of course make other smaller deductions which may prove to simplify the problem. The nature of this method of search is very similar to other mechanical means of proof such as Tableaux [103]. If the search discovers a solution then that solution is known as a *satisfying assignment* for the problem. In the case of the method finding no satisfying assignment, then the record of the search constitutes a proof that there is no solution. Note in this case the "answer" may not be efficiently verifiable as it consists of the search performed in a large *search space*.

Another way to solve the satisfiability problem is to try to "guess" the answer. Generating a random assignment and using some strategy to modify that assignment to attempt to make it a satisfying assignment is known as *local search*. This gives no guarantees other than if a solution is found then it is a correct solution. This approach is not the focus of this thesis but is further discussed in section 1.3.2.

The aim of an efficient satisfiability algorithm is to reduce the search space that must be traversed. Making a good choice for an assumption may mean that the number of successive splits is significantly reduced. It is the splitting or "branching" of the search space that produces the exponential nature of the search space. Intermediate reasoning, or refinement before branching, can also significantly simplify the problem. It is also possible to further reduce or "prune" the search space by analysing the relevance of some assumptions. It may turn out that they were not needed at all, in which case the branch points where they occur can be removed altogether. A successful satisfiability checker will often combine these techniques to reduce the search space as much as possible. Understanding the function and effectiveness of individual techniques helps to enhance their effects and understand the nature of the problem. In this work we investigate the theory behind some of these techniques.

1.1.4 Further Information and Resources

There are many different areas ranging from the entirely theoretical, e.g. complexity analysis, to the entirely empirical, e.g. system construction. Excellent general reviews of research in satisfiability exist. An informative and comprehensive overview appears in [22]; a considerably larger categorisation of algorithmic approaches and research on satisfiability appears in [52] and a somewhat more recent review appears in [42].

There are centralised electronic resources for research into satisfiability: `satlib`

[57] is a repository for collections of solvers, benchmarks, and other research resources for satisfiability. The `satlive` [68] web site is a community driven resource for providing information on recent publications and announcements related to research in satisfiability. `Sat-Ex` [100] is a web based experimental platform for the comparison of satisfiability solvers.

1.2 The Satisfiability Problem

The formal definition of Satisfiability, or SAT, derives from an equivalent problem in propositional logic. We can state the problem as asking whether we can find a suitable assignment to the propositional variables in a given formula that makes the formula true. We can restrict the definition to the equivalent question where the formula is in conjunctive normal form (CNF) – a conjunction of disjunctive clauses, in essence a collection of lists of requirements such as the simple party example above. Following [39], we formally define the problem as follows: Let V be a set of boolean variables

$$V = \{v_1, v_2, v_3, \ldots, v_n\}$$

A truth assignment for V is a function $t : V \to \{T, F\}$ mapping a variable to truth values. When $t(v_i) = T$, v_i is true under t, and when $t(v_i) = F$, v_i is false under t. For every $v \in V$, v and $\overline{v}$ are literals over V. For any truth assignment t, the literal v is true if and only if the variable v is true under t, and the literal $\overline{v}$ is true if and only if the variable v is false under t. A *clause* is a set of literals representing the disjunction of those literals, thus a clause can be defined as $C = (l_1 \vee l_2 \vee \ldots \vee l_k)$ where each l_i is a literal. Given a truth assignment t, a clause is *satisfied* if and only if at least one of its members is true under t. Note that an empty clause represents the empty disjunction, which is false under all truth assignments. It therefore represents a contradiction. The given CNF formula F is a conjunction of a set of clauses and thus can be defined as

$$F = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_m$$

A CNF formula is *satisfiable* if and only if there is a truth assignment t which simultaneously satisfies each of its member clauses. This assignment is known as a satisfying assignment. Note that an empty formula (i.e containing zero clauses) is true under all truth assignments. The satisfiability problem is: Given a formula F (constructed over some finite set of variables V), is there a satisfying assignment?

An example is as follows: Let $V = \{a, b, c, d\}$ and

$$F = (a \vee b) \wedge (\overline{a} \vee c) \wedge (\overline{d} \vee \overline{a} \vee \overline{b})$$

then assigning a and c to true, and b to false is sufficient to satisfy the formula. Our simple example of organising a party can be seen in propositional logic when we consider that each friend contributes a clause, and that each variable represents a possible event at the party.

1.3 Solving SAT

The most effective approaches used to solve SAT problems are generally based on the Davis Putnam Logemann Loveland algorithm [27] (DPLL). It is sometimes referred to as the Davis Putnam algorithm (DP), however the original Davis Putnam algorithm [28], was quite different and resembled resolution proof techniques. The revised DPLL algorithm, published soon after, is the basis for algorithms in use today. The DPLL algorithm is further illustrated in Section 1.3.1. While DPLL is the dominating technique, there are other approaches, sometimes used as an adjunct to DPLL. The more common methods will be briefly reviewed in Section 1.3.2.

1.3.1 The Davis Putnam Algorithm

DPLL traverses the search space by constructing a tree whose paths correspond to variable assignments. Additional reasoning allows the approach to avoid the worst case of exploring all possible assignments. The key performance gain that this algorithm has is called Unit Propagation. Making an assignment (or assumption) corresponds to a branch in the search tree. After propagating the effects of this assignment on the given formula, this assignment may infer the values of other variables. A *unit clause* is one that can only be satisfied by one assignment, because other assignments have eliminated the other members of that clause. When the algorithm finds a unit clause, it makes that assignment, and then propagates it across the remaining clauses. These simple deductions, when performed after each assumption, create massive gains in efficiency.

Algorithm 1.1 ($DPLL()$) describes a simple pseudo-code version of a DPLL style routine. The notation () in the code denotes the empty clause or empty set. Note that an empty clause in a CNF formula implies contradiction. An empty formula implies that each clause has been satisfied. The $DPLL()$ procedure takes the formula and an assignment state as parameters which are passed by reference, i.e. alterations of these variables by the procedure are seen by the calling routine. The procedure returns $True$ when the given formula is satisfiable and in this case the assignment state holds a solution.

Lines 1–5 test whether the formula contains a contradiction or is satisfied.

Algorithm 1.1 A Simple Davis Putnam Logemann Loveland algorithm

```
   $boolean$ **DPLL**($Formula\ F, Assignment\ A$)
1: if $() \in F$ then
2:    return $False$
3: else if $F = ()$ then
4:    return $True$
5: end if
6: $a \leftarrow ChooseAssumption(F, A)$
7: $UnitPropagate(a, F, A)$
8: if $DPLL(F, A) = False$ then
9:    $UndoPropagate(a, F, A)$
10:   $UnitPropagate(\overline{a}, F, A)$
11:   if $DPLL(F, A) = False$ then
12:      $UndoPropagate(\overline{a}, F, A)$
13:      return $False$
14:   end if
15: end if
16: return $True$
```

Line 6 selects an assignment to be made from the set of all currently unassigned variables. The heuristic process of choosing the next assignment can greatly affect the performance of the search and is the subject of Chapter 2.

Line 7 propagates the effect of the assumption chosen. The $UnitPropagate()$ routine is discussed below. Note that on lines 9 and 12 the effects of $UnitPropagate()$ are reversed with a call to $UndoPropagate()$. This guarantees that the state of the formula is consistent with the assignment state. It is possible to use other approaches. A much more detailed discussion appears in Chapter 3. For the moment we note that, at the very least, undoing unit propagation may be simply implemented by taking a copy of the formula prior to modification, and that copy may be reinstated if required.

Lines 8–16 perform the recursive construction of the search tree. If after making an assumption a solution is not found, the alternative space, where the negation of the assumption is true, is explored. The negation is "deduced" via the contradiction found after making the original assumption (*reductio ad absurdum*). Intelligent construction of the search tree can yield a far more efficient traversal and exploration of the search space. These issues are the subject of Chapter 3.

Unit propagation consists of some simple additional rules of reasoning, but it is a powerful technique. The unit propagation routine executes *unit resolution* and *unit subsumption* on a given assumption. An assumption is treated as a *unit clause* – a clause containing a single literal which must be true to create a satisfying assignment. Unit resolution is the resolution operation between two clauses where one of the clauses is

Algorithm 1.2 The unit propagation routine

```
    **UnitPropagate**( $Literal\ a, Formula\ F, Assignment\ A$ )
1:  $A \leftarrow \{a\} \cup A$
2:  for all $c \in F$ do
3:      if $a \in c$ then
4:          $F \leftarrow F - \{c\}$
5:      end if
6:  end for
7:  for all $c \in F$ do
8:      if $\overline{a} \in c$ then
9:          $c \leftarrow c - \{\overline{a}\}$
10:     end if
11: end for
12: while exists $c$ such that ($c \in F$ and $c = \{x\}$) do
13:     $UnitPropagate(x, F, A)$
14: end while
```

a unit clause. Unit resolution thus eliminates the part of the other clause that cannot be satisfied. Unit subsumption eliminates any clauses that are subsumed by a given unit clause. This is not strictly a deduction but a simplification performed by eliminating the parts of the formula that have been satisfied. The steps of unit resolution (UR) and unit subsumption (US) with regard to the state of the formula can be described as follows

$$\text{(UR)}\ \frac{(\overline{l_1}), (l_1 \vee l_2 \vee \ldots \vee l_k), \ldots}{(\overline{l_1}), (l_2 \vee \ldots \vee l_k), \ldots} \qquad \text{(US)}\ \frac{(l_1), (l_1 \vee l_2 \vee \ldots \vee l_k), \ldots}{(l_1), \ldots}$$

Unit propagation further "propagates" the effect of an assumption by recursively applying itself to any unit clauses that are created during the process. Algorithm 1.2 shows a basic unit propagation routine. The parameters are again passed by reference.

Line 1 updates the assignment state used to record a possible solution.

Lines 2–6 perform unit subsumption – the value a subsumes any clauses that contains it, thus such clauses are eliminated.

Lines 7–11 perform unit resolution – resolution is performed between the value a and clauses containing its negation, thus occurrences of $\overline{a}$ are eliminated from the formula.

Lines 12–14 apply unit propagation recursively to any newly created units.

1.3.2 Other Methods

Although DPLL style algorithms are generally accepted as the most efficient approach for solving satisfiability problems, there are several other approaches [52, 42], many of which seem to be motivated by methods used for other decision problems. In this section we briefly mention current successful techniques with foundations in propositional reasoning.

Resolution [93] based systems are far more predominant in theorem proving for first order logics, but are able to solve particular classes of satisfiability problems very efficiently. A specialised version called directional resolution has been shown to be particularly successful on specific classes of random problems that have similarities to real world problem domains [92]. Resolution based techniques can also be integrated within DPLL style approaches [71]. Some practicalities of resolution are addressed in Chapter 4.

The use of more complex rules for reasoning (i.e. the application of useful theorems) is also prevalent. Stålmarck's algorithm [105] (see also [53]) has been used to implement a satisfiability solver [51]. Integration of equivalency testing to simplify search has also been shown to be useful [70]. Exploiting properties of polynomially bounded satisfiability decision problems during search is discussed below in Section 1.3.3.

So far we have discussed methods that are considered *complete* – methods that will determine whether there is a solution or not. Other notable approaches are those which employ an *incomplete method*. These algorithms do not attempt to verify that a problem has no solution, but can *locally search* for possible solutions within the space of all possible assignments in a way similar to a random walk. These systems have had remarkable success on a variety of problems and the most popular methods are based around `GSAT` [99]. Further gains are made when incorporating limited amounts of reasoning, dubbed "local search". A class of constraint satisfaction algorithms which combine aspects of incomplete and complete methods is investigated in [46].

1.3.3 Proof and Complexity

The satisfiability problem has roots in complexity theory and the interests of proof in propositional logic [21, 23]. Satisfiability may be extended to all syntactic variations of propositional formulae by observing that any propositional formulae has an equivalent CNF representation. Furthermore using a complete method for solving SAT can be interpreted as constructing a proof by refutation of the negation of the given formula. For example, the original formula to be "proved" may be in disjunctive normal form (DNF) and the CNF formula for SAT algorithm input is obtained simply by negating it. This approach allows us to consider the CoNPC problem TAUT: Given a formula F, is F true under all possible assignments? In other words, is F a theorem? Indeed verification problems may be posed as satisfiability problems: Does the refutation proof produce a counter-model? While the subject of this work is search for satisfiability it is noted that it has relevance to theorem proving and that there is a correspondence of certain search methods to proof by refutation.

The worst case time bounds for solving satisfiability problems are exponential. Bounds are generally defined for the restricted problem of 3-SAT, where each clause contains at least 3 literals. For 3-SAT, a bound of $O(1.5045^n)$ has been shown using a method based in propositional logic [66]. This bound is improved to $O(1.476^n)$ by mapping the satisfiability problem into a specialised constraint satisfaction problem [94]. However, several studies of experimentation with "hard" 3-SAT problems (see Section 1.4) show the average cost is much lower (e.g. [36, 26, 65]). The least average solution cost found, for difficult problems, is 1.0334^n [65]. The complexity of propositional proof is covered in [12, 13, 107].

For practitioners perhaps the most interesting results from complexity analysis are those which show certain classes of SAT problems to be solvable in polynomial time. The incorporation of these methods into general satisfiability algorithms are called *relaxation techniques*. For 2SAT problems a solution can be found in $O(n)$ by computing the transitive closure of the implication graph representing the problem [4, 35]. Despite the polynomial expense of this operation, variations of this technique have been attempted [16, 67]. A set of propositional Horn clauses is also decidable in linear time [30]. Several further methods extend this result by relabelling variables and detecting special cases, yielding computations bounded by low-order polynomials, e.g. see [17, 97]. Schaefer developed a scheme which defined sub-classes of the satisfiability problem and showed that within this infinite class of satisfiability problems any member is either polynomial-time decidable or NP complete [96] . This powerful result is known as the *dichotomy theorem*. More recently Franco and Van Gelder reviewed and investigated polynomial time solvable classes of satisfiability problems and introduce novel classifications of tractable satisfiability problems [59].

1.4 Satisfiability Problems

There is a wide range of examples for practical applications in satisfiability research including hardware design [109, 74, 67], planning [62, 63, 61], theorem proving [116, 117] and encryption [77]. These applications exploit efficient translations from their natural domain in order to utilise the abilities of satisfiability search procedures. While collections of benchmark problems from practical scenarios exist, there is only a limited number of instances. Comparatively there is an almost inexhaustible supply of random satisfiability problems based on simple problem models.

"Hard" random 3-SAT problems arise from the observation of phase transition behaviour discovered in empirical studies of some decision problems [18, 81]. This is known as a phase transition due to its similarity with the thermodynamic behaviour of physical matter as it changes state or phase. For random 3-SAT problems, with number of variables n and number of clauses m, the clause to variable ratio is defined as $r = m/n$. If n is fixed and a sample set of random problems are generated for values of r, then there will be a sudden transition, as r increases, from the problems being mostly satisfiable to the problems being mostly unsatisfiable. This transition corresponds to a peak in the difficulty of solving the problems in the sample set. Randomly generated 3-SAT problems with a clause to variable ratio in the vicinity of 4.25 are particularly difficult when compared to other ratios. These problems are particularly useful for testing satisfiability algorithms as they are simple, difficult, and plentiful. While the phenomenon of "hard" problems has been the subject of much research, the problem of generating random but "realistic" satisfiability instances is yet to be fully explored. Indeed this has been proposed as a challenge to the satisfiability research community [98]. Approaches for realistic modelling of real world scenarios in satisfiability instances are the subject of Chapter 4.

1.5 This Thesis

This thesis investigates some fundamental, but practical, aspects of research in satisfiability. It is motivated by the lack of theory that is capable of both describing and generalising those important and successful aspects in research on satisfiability. Using approaches that capture theoretical foundations yields a better understanding of satisfiability and, if general enough, yields a better understanding of search. Enhancements and insights developed in this thesis can be extended to other search problem and proof generation domains. The aims of this thesis are to explain some fundamental mechanics of successful search methods in satisfiability, and to question whether considerations about "real world" problem domains lead to a better understanding of the practical nature of search methods. Although the work presented develops theoretical foundations, the processes used yield immediate and practical results for the satisfiability problem. When applicable we demonstrate where a particular approach may be used for richer problem domains.

Chapter 2 investigates the mechanism used for choice in constructing a search tree. It develops a theoretically derived branching scheme, which empirical analysis shows to be an effective and efficient strategy. We find that the parameterisations of the scheme are very similar to other schemes that are derived empirically. The foundations of the proposed scheme yield explanations of why choice mechanisms work so well. The method used to derive the scheme for satisfiability can be applied to other search problems.

Chapter 3 investigates the mechanism of backtracking, focussing on intelligent backtracking schemes and the notion of logical relevance from non-classical logics. It demonstrates that the aims of intelligent backtracking search tree construction directly correspond to ideas in a formal system of relevant reasoning. The work develops a formulation and framework that subsumes previous backtrack search tree construction techniques and shows that the new approach yields far more flexibility in proof construction. It is illustrated that the simple use of formal logical foundations yields a malleable framework for developing systems concerned with solving problems in proof construction and search.

Chapter 4 investigates modelling "real world" problem scenarios, and whether such models lead to a better understanding of search methods for practical problems. We propose and justify a model capturing some real world properties. Through experimentation we analyse search behaviour and identify where certain popular search techniques can succeed or fail in the situations that the model captures.