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 $\LaTeX\,2_\varepsilon$.

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 $t : V \rightarrow \{T, F\}$ 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 () ∈ F then
2:    return False
3:  else if F = () then
4:    return True
5:  end if
6:  a ← ChooseAssumption(F, A)
7:  UnitPropagate(a, F, A)
8:  if DPLL(F, A) = False then
9:    UndoPropagate(a, F, A)
10:   UnitPropagate(ā, F, A)
11:   if DPLL(F, A) = False then
12:     UndoPropagate(ā, 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.

Chapter 2

A Theory of Choice

A *branching scheme* or choice function determines the order in which search is executed. Its importance for many search methods cannot be understated. When compared to a fixed or even random order an advanced branching rule is extremely effective. The point at which proof search branches is generally the splitting rule in many mechanised proof systems. It is from this rule that we derive the potential exponential size of a proof (or search space) in systems like DPLL or Tableaux methods. It is thus desirable that any choice made reduces the potential number of subsequent choices made later. That such a scheme is possible is intuitive, and numerous empirical observations have verified that utilising some intelligent ordering scheme can reduce the search space by orders of magnitude [56].

There are numerous approaches to branching schemes and they are typically specialised for the kind of search problem for which they are used. Within the domain of propositional satisfiability there are many examples of branching schemes. Currently the most successful schemes combine multiple branching rules with some other technique such as a look-ahead [71] in order to refine the selection. What is most common about the approaches in deriving branching rules is that they are initially motivated by intuitive notions about what a branching rule should do, and these kinds of hypotheses are investigated and improved by empirical observation alone. What is most surprising is that very little solid theory about branching mechanisms exists. While performance observations of various schemes can be readily observed, there is very little theory about why a successful branching rule should be successful, other than the intuitive reasons or empirical methods from which it was derived.

This chapter presents a method for theoretically deriving a branching scheme for the propositional satisfiability problem. It illustrates some relationships between this scheme and some other well known branching schemes. It is demonstrated that the branching scheme derived is both computationally efficient and effective in reducing search space. We also show the relationship of this method to the Kappa search space model [43] and argue that our method can be used to derive efficient and effective branching schemes for many search algorithms that perform some kind of splitting rule. It is also shown that when the scheme is refined with some simple empirical analysis one can achieve the same sort of performance exhibited by the best known empirically derived branching schemes.

2.1 A History of Choice Strategies

In general a branching rule is a function of statistical data gathered from the problem at a point in the search. The rule specifies some ordering of the potential choices to identify the best candidate. For example one might count the number of occurrences of each variable, and then select the variable whose occurrences are greatest. Furthermore, a branching rule can specify the value of the variable to be branched upon first (i.e. the negative or positive literal). We define a branch point as having an *assumption branch* (the first branch value taken) and a *deduction branch* (the branch value deduced via refutation of the assumption). Branching rules are numerous in the literature and we refer the reader to Hooker and Vinay [56] and Freeman [36] for a more complete review of how many of these rules are actually computed. The review in this chapter concentrates on the rules that seem most important or most influential in the history of branching rules. The focus is on the associated theory of a scheme along with its performance. This opposes the more common approach of just comparing individual performance gains of a set of branching schemes. We review the successful ideas which have motivated branching schemes and have been shown to be useful in reducing search cost.

2.1.1 The Simplification Hypothesis

Hooker and Vinay performed a comprehensive empirical study using several branching rules in order to refine understanding and improve performance [56]. They consider several DPLL propositional search systems which differ by branching rules only. Their most important conclusion is the empirical support they find for the *Simplification Hypothesis*:

> Other things being equal, a branching rule works better when it creates simpler sub-problems.

This advances an earlier perspective they identify as the *Satisfaction Hypothesis*:

> Other things being equal, a branching rule performs best when it creates sub-problems that are more likely to be satisfiable.

which is motivated by the observation of some of the rules they investigate. The effectiveness of one hypothesis over the other is illustrated by their use and subsequent modification of the Jeroslow-Wang rule [58]. This rule, like most, can be described with a *weighting scheme* and a *priority function*. A weighting scheme defines a function which computes a score associated with a literal. This usually incorporates some weighting of occurrence counts in clause sizes. The priority function defines the ranking of variables. Some priority functions are quite simplistic and may just select the maximum score. In general the priority function can be considered as a function of both the positive and negative literal weighting scheme scores for a variable. The Jeroslow-Wang rule is a weighting scheme using the variable occurrence counts where weighting is based on the size of the clauses in which the variable occurs. A literal l is given a score

$$W_{JW}(l) = \sum_{i, l \epsilon C_i} 2^{-n_i}$$

where n_i is the number of literals in the clause C_i. Most weighting schemes use this kind of function — the dot product of the weighting values for clauses and occurrence counts for clauses. In this case the weighting value of 2^{-n_i} is motivated by the fraction of assignments that the clause eliminates (ignoring dependence). The score for each remaining literal in the problem is computed and the priority function simply maximises $W_{JW}(l)$. Hooker and Vinay's improved rule is named the Two-Sided Jeroslow-Wang rule. It uses the same weighting scheme, but the priority function maximises

$$2SJW(l) = W_{JW}(l) + W_{JW}(\bar{l})$$

and chooses to branch first on *max*$(W_{JW}(l), W_{JW}(\bar{l}))$. This second version is far more successful than the original. It becomes apparent that, when evaluating a potential candidate literal as the next branching variable, we must take into account the possibility that we will return to the branch point and explore the tree under the negation of the assumption. Priority functions can be classified as being *one-sided* if they do not consider the effects of both branches at a point, and *two-sided* if they do.

In terms of refutation proof generation this concept must hold. We make an assumption l, then prove this invalid and follow by asserting $\bar{l}$. For the case of unsatisfiability we must always investigate both sides of the branch point. The Two-Sided Jeroslow-Wang branching rule was found to be one of the most successful rules tested by Hooker and Vinay, and in general those rules which comply to the Simplification Hypothesis are more successful than those which do not account for both possibilities leading from a branch point.

2.1.2 Further Advances

Many variants of weighting schemes and priority functions have been proposed and experimented with. See [16, 36, 26, 31, 108, 115] for a wide range of examples. These schemes are justified by some notional reasons and empirical evaluation. One of the most prevalent techniques recurrent in the literature is the use of a very simple occurrence count. It is called the MOMS (Most Occurrences in Minimum clause Size) [88], and ranks candidates by choosing the literal that most commonly occurs, but only counts occurrences within the smallest clause size within the problem. Many variations of MOMS exist, mainly with different priority functions. The effectiveness of this approach is commonly attributed to its computational simplicity. It is used in state of the art systems in combination with other branching schemes.

It seems the two most prevalent and successful ideas in recent advancements for branching schemes were

- the use of a multiplicative priority function to achieve better balancing in the search tree, and
- the use of a look-ahead procedure to gain more accurate information about the use of a variable.

We discuss each with some detail in the next sections.

2.1.2.1 Multiplication

One of the most effective modifications to this approach was the empirical observation that a multiplicative priority function was far more effective than the additive versions. It seems to have first been used by Van Gelder and Tsuji [108]. While the authors recognise its power they provide no significant theory as to why it may work. Multiplying the two weighting scores results in better balancing of the branch point. For example, if the scores for a variable v are $(W(v), W(\overline{v}))$ then $(3,3)$ will be chosen over $(7,1)$ yielding a far more balanced looking branch point.

In his PhD thesis Freeman [36] includes a fairly extensive review of various branching schemes. He concludes, based on empirical experience, that the best priority function, based on some weighting scheme $W(x)$ is

$$H(x) = W(x)W(\overline{x})2^c + W(x) + W(\overline{x})$$

where c is some fixed constant (usually 10). The variable selected is the one which maximises $H(x)$. The first branch chosen to be explored is decided by the function *max*$(W(x), W(\overline{x}))$. Note that the term 2^c acts to use the sum of the two weighting scores as a discriminator. In the event of a tie the function prefers unbalanced branches. Freeman's weighting scheme used with this function in `posit` only counts the smallest clauses (i.e. a purely MOMS approach). This function is used by other recent DPLL implementations and is generally accepted to be extremely useful, especially when combined with look-ahead procedures, which we discuss below.

2.1.2.2 Further Comments on Multiplication

The branching scheme theory we present in this chapter does not use a multiplicative or additive function, and other than the obvious balancing act that the multiplicative functions do, we have not been able to find a solid theoretical foundation for their success. However, we made the following observations in the process of this work which are of some interest:

i) A multiplicative priority function does a reasonable job of balancing a tree. Consider the class of recurrence relations

$$F_{a,b}(n) = F_{a,b}(n-a) + F_{a,b}(n-b)$$

Given appropriate initial values we can model the size of a search tree of a problem with n variables using this function when we eliminate a variables whenever we branch left and b variables when we branch right. This is a fairly unlikely assumption but the model is simple and it is easy to compute $F_{a,b}(n)$ using dynamic programming techniques. We evaluated the tree sizes for some small integer pairs (a, b) using a large but realistic value of n. To smooth the results and eliminate ties we used the sum of all tree sizes up to n. The pairs were ordered by *ascending* tree size. When we compared the *descending* order for the priority function $P(a, b) = ab$ we found a good correspondence. Most of the pairs were in the same order with a few exceptions. For example $F_{8,1} < F_{3,3}$ (and $F_{7,1} > F_{3,3}$) but $P(8, 1) < P(3, 3)$. Obviously $P(9, 1) = P(3, 3)$ but Freeman's function would break the tie by choosing the pair $(9, 1)$. This agrees with the smallest tree ordering given by $F_{a,b}(n)$, noting $F_{a,b}(n) > F_{a+1,b}(n)$.

ii) For some additive priority function score $P(x) = W(x) + W(\overline{x})$ the product of every possible pair $(W(x), W(\overline{x}))$ is maximal when $W(x) = W(\overline{x})$. Conversely the sum of every possible pair from a given multiplicative priority function score is maximal when the difference between the two weighted scores is greatest. The two approaches seem complementary, and perhaps their success in combination is related to this.

In hindsight the multiplicative function is just a better total balancing mechanism than the additive version. That it approximates a better balanced search tree both in practice and model suggests that, whatever a "best" priority function is, the multiplicative functions are just a better approximation of it than the earlier additive ones.

2.1.2.3 Look-Ahead

A look-ahead procedure executes searches on multiple possible branching points in order to evaluate the outcome. The important difference between earlier branching schemes and several recent schemes, including `posit`'s, is that they use a combination of branching rules combined with a look-ahead procedure in order to refine the choice the branching algorithm finally makes. For a look-ahead mechanism to be successful it must be computationally inexpensive. Common methods cap the number of look-ahead operations performed at any branch point. They also tend to use a combination of branching schemes in order to rank the choices. One branching scheme may select candidates for look-ahead, and the second may choose from those candidates. One of the most successful implementations of this method is `satz` which is mostly based on the performance of its look-ahead scheme and priority functions [71].

The most likely reason for such significant gains is that look-ahead can capture the effect of unit propagation. The branching scheme and the look-ahead functions in `satz` seem to be geared for this effect. Recently Slaney has observed that unit propagations occur in large groups, or clusters, down the heuristic branch [102]. It is common at branch points to have few or no units generated. At some branch points however several units are generated, and this is termed a *unit cascade*. This means that at some branch points there is a lot of information to be gained, and if this can be detected earlier, then the search space can be significantly simplified. The other effect is failed literal detection: a literal x may be instantiated (to True) immediately if it is known that $\overline{x}$ leads to a contradiction. We performed some basic experiments which suggest that about half the branch points in a search tree have this property. Note that half the branch points in a search tree are at the very bottom, and this is where contradiction is detected.

If an efficient look-ahead method can be devised for a branching scheme it can provide huge gains. The look-ahead process acts as an error correction mechanism. The statistical estimates made from a branching scheme can be revised and so a better discrimination of candidates is achieved. The expense of this method often leads to very basic branching schemes being used to select from the look-ahead candidates. We do not deeply investigate look-ahead extensions to branching schemes in this chapter, but hypothesise that, in the real terms of search space reduction and best choice, the function of look ahead is to correct the error in estimates. The central difficulty is identifying how to limit the cost of this method.

2.1.3 Newer Schemes

So far our review has covered the motivations and techniques used in most branching heuristics. However there are two recent branching schemes that are derived using a stronger theoretical basis and both merit attention. The first is a scheme based on the Kappa model for search space. This model and the branching scheme is discussed in Section 2.1.3.1. The second branching scheme is motivated by issues in complexity analysis and is briefly discussed in Section 2.1.3.2.

2.1.3.1 A Kappa Based Heuristic

The Kappa model captures the notion of the "constrainedness" of a problem [43]. When a search problem is underconstrained it is likely to have many solutions, and when it is overconstrained it is likely to have none. In satisfiability the constraints are the clauses in the given formula. The model defines the constrainedness for an *ensemble of problems*, that is the model is representative for a class of search problem. Within the entire search space there is an expected number $\langle Sol \rangle$ of possible solutions – the solution space for the ensemble. Any point in the search space may be represented by some unique N-bit binary number, corresponding to the assignments of variables, and N is the logarithm, base 2, of the size of the entire search space. The Kappa value for constrainedness is defined as

$$\kappa = 1 - \frac{log_2(\langle Sol \rangle)}{N}$$

When $\kappa = 0$ all the points in the search space are solutions, and the problem is underconstrained. As κ tends to infinity the problem is overconstrained. At $\kappa = 1$ the ensemble of problems are said to be critically constrained, and this corresponds to the point at which "hard" problems are observed in phase transition phenomena.

The Kappa branching scheme was developed by Walsh [110]. This method is motivated by observations of the behaviour of Kappa down the heuristic branch in random 3-SAT problems. The term used for $\langle Sol \rangle$ in the expression for calculating Kappa is defined by the following: If there are l_i clauses of length i, the number of possible truth assignments from the total search space 2^n not ruled out by each clause is $(1 - \frac{1}{2^i})$, thus

$$\langle Sol \rangle \approx 2^n . \prod_i (1 - \frac{1}{2^i})^{l_i}$$

Walsh shows that when the problem is over-constrained, i.e. the clause to variable ratio is above the phase transition value, Kappa increases as the search progresses. This is due to the likelihood that the problem is unsatisfiable, and so the solution space is decreasing. When the problem is under-constrained, i.e. on the other side of the phase transition boundary, Kappa decreases as the search progresses. At the point of phase transition, where the clause to variable ratio is around 4.3 (in the "hard" region), the change in Kappa is much less noticeable and these problems are deemed to be on the "knife-edge". These are the problems that are just as likely to be satisfiable as unsatisfiable.

It was suggested by Gent *et al.* [43] that for soluble problems a search algorithm should try to get off the knife-edge and select a branch which provides the most under-constrained problem. Walsh's heuristic follows this idea and his branching scheme selects the literal which minimises Kappa. This means that a one-sided priority function is used. For soluble problems the method seems to perform better than a simple MOMS based approach in terms of the number of branches taken. CPU time cost is quite different. Calculating Kappa for each literal is tantamount to computing a look-ahead scheme for every literal in the formula. This is a very expensive task. An alternative approach that maximises Kappa is also tested for unsatisfiable problems but is not quite so successful. The "knife-edge" phenomena and associated approaches to branching strategies are discussed in further detail in Section 2.2.3.

The branching scheme presented in this chapter turns out to be a Kappa-style model. We derive the properties of the weighting scheme and priority function motivated by the concept of the *potential search space* beneath a point in the search. In essence our method calculates an approximation for the value of $\langle Sol \rangle$ in the Kappa model. We use this value for scoring the candidate literals. We do not follow the knife-edge theory, but use a conservative two-sided priority function that aims for a reduction in the theoretical potential search space. Further details of the relationships between our scheme and the Kappa model will be discussed in more detail in Section 2.2.3.

2.1.3.2 A Complexity Based Heuristic

This newer branching scheme is due to Kullman [65]. Kullmann's branching scheme is implemented in the publicly available system `OKSolver`. The scheme is very complex. It seems motivated by issues of complexity theory for satisfiability [66], and the scheme attempts to minimise the "decision complexity" of a branch. The system also performs a full look-ahead procedure, but has been shown to be competitive with other state of the art systems in some problem domains [65]. It still however uses an empirically derived weighting scheme. Kullman derives a set of "magic numbers" which define the weight values for the weighting scheme used. We shall revisit these values later in this chapter.

2.1.4 Other Work

Recently Marques-Silva has shown that, when combined with an intelligent backtracking approach, the difference in performance of some branching schemes is not very significant [75]. The work compares several schemes ranging from the two sided Jeroslow-Wang scheme to random or input ordered schemes. These experiments show that the ability of an intelligent backtracking system to correct bad choices is very powerful. The cost in performing the corrections appears to be quite inexpensive. We note that the comparisons were performed using different branching techniques within the system `GRASP` [76] which employs a variety of search techniques, including an intelligent backtracking scheme which prunes unnecessary search space. Comparisons were made through performance evaluations for the DIMACS benchmark problems (see [60]). Although a variety of branching schemes were used, the only multiplicative priority function employed was Freeman's combined with a MOMS scoring (unweighted) scheme. The most advanced weighting scheme appears to be from Jeroslow-Wang. No schemes that are known to be superior to these methods are included – for example a weighted multiplicative scheme [108], or a look-ahead scheme [36, 71]. Experiments performed using the system `satz` on the same benchmarks show that an efficient look ahead scheme alone is capable of being competitive with state of the art systems that employ intelligent backtracking techniques, including `GRASP` [72]. Furthermore Chen and van Beek illustrate that a "perfect" variable ordering cancels the effect of the intelligent backtracking technique Conflict Directed Backjumping [19], which is complementary to Marques-Silva's work.

It is apparent from these studies, and other performance evaluations by satisfiability system authors, that the ideal approach would be a combination of both look-ahead and "look back" techniques. There is no clear boundary between the kind of search behaviour that each technique exhibits. The advantage of an intelligent backtracking system is that, for a fixed cost, pruning can be performed and bad decisions can be eradicated no matter how good the branching scheme is. There is no evidence that these enhancements can ever be superseded by a branching scheme alone, nor is it likely. Intelligent backtracking schemes are the subject of the next chapter.

2.2 A Theoretical Approach to Choice

We now describe a theoretically motivated method of deriving a branching scheme. Starting from basic premises we derive a weighting scheme and the corresponding priority function.

2.2.1 The Weighting Scheme

The motivating force behind a good branching scheme is that it reduces the amount of potential search space that needs to be examined. Consider the simple worst case search space of 2^n for an n variable propositional satisfiability problem: we know that, in a propositional logic, knowing the value of a single variable reduces the *potential search space* by half. In general we can define a reduction factor of the given search space whose value is effected by the state of the information we have about that search space. Note that the model of the "potential search space" is based on the potential of a worst case scenario, and this can be interpreted as allowing the possibility of a problem instance being unsatisiable to dominate the search space model.

So, given a search space S, the addition of some knowledge (for example some known variable assignments) reduces the initial search space S, giving resulting search space

$$S' = S\frac{1}{2^r}$$

where r is an exponential term that determines the actual reduction of S. The greater the value of r, the smaller S' will be. Thus to reduce the potential search space as much as possible we want the reduction factor to be as small as possible. For example if we know the values of 5 variables then our reduction factor is $1/2^5$. In terms of assigning a value to r it is easy to see that the value of a single variable (or a unit clause) is worth 1, but how much are clauses of greater sizes worth? The introduction of a binary constraint reduces the search space by $3/4$, that is, it only allows $3/4$ of the possible assignments in the search space since it eliminates the choice of both variables being false. In terms of an exponent of two, which is how we model the reduction factor, we calculate the reduction by the introduction of a binary clause to be

$$\frac{3}{4} = \frac{1}{2^c} \qquad \text{thus} \qquad c = log_2(\frac{4}{3})$$

and in general, for a clause of length i, the equivalent exponential term for our reduction factor is

$$c_i = log_2(\frac{2^i}{2^i - 1})$$

The values of the first 5 exponential terms for clause reduction factors are given in Table 2.1.

For a collection of p clauses of size i, the reduction factor for that set of clauses is simply

$$\frac{1}{2^{pc_i}}$$

What we wish to know is the total reduction factor resulting from the assumption of

c_1	c_2	c_3	c_4	c_5
1.0	0.415037	0.192645	0.093109	0.045803

Table 2.1: Search space reduction factors, in terms of an exponent of two for a single clause, sizes one to five. Note that for simplicity we formulate the reduction factor as the inverse function $1/2^c$ rather than having negative constants.

a particular literal. By using the exponential constants we need only keep track of the value of the exponential term to calculate the search space reduction.

At a branch point we assume the value of one variable. Propagating the implications of this value changes the relative distributions of clauses in the formula. We define $\sigma_{a,i}$ to denote the number of occurrences of the literal a in clauses of length i in a given formula. At some branch point, when we assume $\overline{a}$, we generate $\sigma_{a,2}$ new unit clauses which force variable assignments. However, we also eliminate the binary clauses which are subsumed by the new units. We also lose a further $\sigma_{\overline{a},2}$ binary clauses due to subsumption by the assumption. Here we are following the theory of "constrainedness" – the current formula contains information about how the search space is constrained, and by making an assumption we affect that constrainedness. We gain information by making the assumption, but we also lose the information that becomes subsumed.

Since we model the reduction of search space by $1/2^r$, we can capture the reduction made by an assumption in the single parameter r. Our weighting function calculates what the exponent r is for the assumption. Note that when we make an assumption we assume that the smallest clause size is 2, since we always perform unit subsumption. We define our weighting function as the value of r. It is a function of a literal x and is defined as

$$W(x) = \sum_i c_{i-1}\sigma_{x,i} - c_i\sigma_{x,i} - c_i\sigma_{\overline{x},i}$$

for all i such that the variable x occurs in a clause of length i. It is clear that larger values of $W(x)$ correspond to a greater reduction in search space under the assumption of the literal x. We are however ignoring the fact that there are dependencies between the variables in the clauses we look at. In particular a score from this function uses clauses where there is at least one variable in common (the assumption). We will look at this more closely in Section 2.2.4.

Note that unlike many other weighting schemes $W(x)$ can produce scores which are negative, but when this happens $W(\overline{x})$ will be positive. This may be important for experimenters depending on how the the priority function is formulated. It certainly highlights the difference in the theoretical approach to empirically constructed weighting schemes.

2.2.1.1 Correspondence with Empirical Schemes

The weighting scheme proposed in this chapter is quite different from other existing methods, mainly because we include the effects of information loss. A weight score calculated for x includes information about $\overline{x}$. We questioned whether there was any correspondence between our scheme and those that had been derived empirically. The Jeroslow-Wang scheme has theoretical weight values motivated by the idea that weighting for clauses sizes should be different, but is not quite powerful enough to be extended to values in terms of search space. `satz` uses a weighting scheme of fifths [71], which is inspired by the empirically optimal value for the highest weighting values. This scheme's weights can be simply described as the value $(1/5)^{n-1}$ where the variable occurs in a clause of length n. Another contemporary system `OKSolver` uses weights that are empirically derived through random SAT experiments [65]. These systems use different strategies and have different ways of computing priority, but both recognise that weighting based on clause size is required. The weight values for both systems are very similar. The derivation of the weights is entirely empirical and, to the best of our knowledge, there appears to be no approach for how they might be accurately explained.

These weighting values are essentially normalised by their creators. The final score of a weighted computation is just the dot product of the weight values and a variable's occurrence counts. Scaling of the weight values will not affect the ranking of scores. When comparing the weight values for `OKSolver` and `satz` we chose an appropriate normalisation so that the weight for a variable occurring in a binary clause was 1.

In order to compare our weight values to the empirical values, we have to make an assumption about the distribution of positive and negative occurrences of the literal. If we assume that the number of occurrences of x and $\overline{x}$ in a given clause size are equal, then we can derive a weighting rule which eliminates the need for both sets of occurrence counts. In this case the scheme would be much simplified and defined as

$$W'(x) = \sum_i (c_{i-1} - 2c_i)\sigma_{x,i}$$

for all i such that the literal x occurs in a clause of length i.

The leading factor in the new weighting scheme now defines a set of weight values which correspond to a calculation that only uses the literal occurrence counts, under the assumption of equal distribution of its negation. This weighting scheme can also be scaled without effecting the final rankings of candidates tested, so we normalise the weighting value for clauses of length 2 to match the scale of the empirical weight values. These constants, the constants from the rule of fifths, and the constants identified by Kullman [65] are shown in Table 2.2 for c_2 to c_6. Here c_i corresponds to the weight for an occurrence in a clause of length i.

Weight Scheme	c_2	c_3	c_4	c_5	c_6
Fifths	1.0	0.2	0.04	0.008	0.0016
Kullman	1.0	0.2	0.05	0.01	0.003
W' (normalised)	1.0	0.175	0.038	0.009	0.0021

Table 2.2: Weight values for two different empirical approaches and an entirely theoretical derivation.

All three weighting schemes are surprisingly similar. The theoretical weighting scheme is very close to the values derived from empirical methods. For random 3-SAT problems the theoretical constant $c_3 = 0.175$ is still a very reasonable value to use in combination with a choice function like Freeman's $H(x)$, but the optimum value is definitely 0.2.

We note that for the empirically derived weighting schemes the values are derived primarily from experimentation with random SAT problems. The results show that the proposed weighting scheme is on the right track. They also show that this theoretical weighting scheme offers some explanation of the success of existing empirically derived weighting schemes.

2.2.1.2 Related Work

Some of the ideas in this section are not so new. The use of truth table style reduction factors have occurred in other weighting schemes such as the Jeroslow-Wang rule and its variants. Certainly the motivations behind using such values are related, but the formulation and application seems to be critical for success. The concept has also been used to show that the probability of a random 3-SAT problem being unsatisfiable approaches unity (as n approaches infinity), when the clause to variable ratio is 5.19 (several authors, see [22]). This bound that has been improved using other methods [32, 64].

The idea behind accounting for the loss of information under an assumption also occurs in the literature. The relationship of this method with the Kappa model and Kappa based heuristics is strong. We will discuss these relationships in detail in Section 2.2.3. That the loss of information may be important was also observed by Hooker and Vinay in their First and Second Order Probability branching rules [56]. Their results show that these methods were not entirely successful, and we argue that the formulations of gain and loss for the branching rules was just not accurate.

2.2.2 The Priority Function

The priority function of a branching scheme determines how balanced the search tree will be. In general it is the weighting scheme that performs the ranking, and the priority function that combines the scores for a literal and its negation in order to judge best a balanced outcome. Some schemes may not be concerned with balancing and may follow a "greedy" approach. These would follow Hooker and Vinay's satisfaction hypothesis, which was shown to be a less effective approach when compared to the simplification hypothesis. We follow the simplification hypothesis in order to derive a theoretical priority function.

A theoretical model for search tree branching structure was suggested in Section 2.1.2.2. This recurrence relation would be expensive to compute for integer values for (a, b), a complexity of $O(n^2)$. For real valued pairs, which our weighting scheme produces, the expense is far greater and an approximation scheme would be necessary. Although there may be ways of computing approximations to this function efficiently, we define a very simple $O(n)$ function that models the partitioning of search space when a problem is split into two sub-problems at a branch point. Since the sub-problem under the assumption x effectively reduces the search space by $1/2^{W(x)}$ our priority function accounts for the other sub-problem where $\overline{x}$ is true. The function is thus defined as

$$T(x) = \frac{1}{2^{W(x)}} + \frac{1}{2^{W(\overline{x})}}$$

The drawback of this real valued function is that it is more costly to compute, and this is revisited in Section 2.3. It represents the reduced search space achieved when the variable x is eliminated and the two corresponding sub-problems have to be separately solved. To rank the scores for variables we seek to minimise $T(x)$, i.e. choose the variable whose resultant search space appears smallest. Note that $T(x)$ remains positive with negative weight scores. For our experiments shown below we chose to select the branch that resulted in the smallest search space first. This is discussed further in Section 2.3.

This function models the potential search space *at that branch point*. It does not capture the interactions of successive reductions at this and subsequent branch points. It dismally fails the example from Section 2.1.2.2, where for all successive branch points, an $(8, 1)$ tree is smaller than a $(3, 3)$ tree. However, in its defence, it is a relatively simple function which has a theoretical derivation corresponding to the proposed theoretical weighting scheme. That it can be derived illustrates a general approach to branching schemes for search problems. In the absence of a good approximation scheme the $O(n)$ complexity makes it a far more attractive choice than computing potential search tree sizes. The practical use of this function is investigated in Section 2.3.

2.2.3 Relationships with the Kappa Model

It is reasonably obvious that the expression for our weighting scheme is similar to Walsh's Kappa based heuristic [110]. The differences in our approach are that we calculate the *change in solution space size* by accounting for the new clauses and the clauses eliminated. The leading factor of 2^n in Walsh's term for $\langle Sol \rangle$ is eliminated – it is the same for all branch candidates, as is the rest of the expression for Kappa. We don't capture the true value of Walsh's $\langle Sol \rangle$ as we don't perform look-ahead. Our weighting scheme is essentially an approximation for $\Delta\langle Sol \rangle$, and is significantly less expensive to compute.

In terms of the Kappa model we calculate the change in solution space, and branch to a variable that reduces the space for both branches of a candidate. As discussed above, our priority function is a simple model that essentially assumes a worst case scenario. The function attempts to capture the change in solution space for both branches, as opposed to only accounting for the change in solution space for one side of the branch as performed by Walsh's heuristic.

2.2.3.1 Knife Edge Phenomena

Walsh's one-sided priority function is motivated by his observation of the "knife edge" phenomena occurring at the phase transition boundary of random 3-SAT problems. Those problems that are over constrained become more constrained as the search depth increases. Those problems which are under constrained become less constrained with increasing search depth. Walsh's experiments show that in between these two extremes there exists a "knife edge" where the apparent constrainedness of the problem stays the same as the search deepens, and that this point corresponds to the critical value of the phase transition phenomena associated with 3-SAT problem difficulty.

It is posed that for soluble problems a scheme minimising Kappa should be successful as it *increases* the space for potential solutions, i.e. it attempts to maximise $\langle Sol \rangle$. Our priority function attempts to *reduce* the "solution space" because we see this as really being "search space". If a solution does exist, then it should be found faster by searching through less total search space. If there is no solution, then we want to traverse through the least amount of search space to verify this. For an ensemble of problems the concept of solution space may be useful in modelling a search problem domain. For an individual problem instance we cannot so readily assume where and when solutions may be found. We do suggest that the one-sided heuristic may be useful for incomplete methods that incorporate some "local search". This may be an interesting avenue for experimentation.

Slaney has shown that the observations of the knife-edge phenomena are less a function of the criticality of a problem and more a function of the algorithm used to solve it [101]. He shows that a DPLL algorithm without Unit Propagation gives a very different profile of the behaviour of Kappa as search progresses. This casts possible doubt on the motivations for Walsh's heuristic [110] as the knife edge phenomena is not fixed for every algorithm. However, the phenomena is not so much affected by the algorithm itself as much as the way in which Kappa is calculated. Slaney's work highlights the importance for an accurate calculation of $\langle Sol \rangle$, or the potential search space. The Kappa model should require that some logical consistency amongst the collection of the constraints be derived, that is one must observe that the state of the formula implied by the assumption is the true state of constrainedness. For example, if there are a number of identical clauses in the formula, then the estimate is tainted. This situation also occurs when there are strong dependencies between sub-formulae, such as a unit clause and several clauses containing that variable (i.e. when unit propagation is not performed). An estimate of Kappa, or the potential search space, is thus far more inaccurate in such situations. A branching scheme using these kinds of search space models must take into account the accuracies of their estimates of search or solution space. The degree to which an algorithm maintains some kind of "inferential" logical consistency (e.g. eliminating dependence between unit clauses and the rest of the formula via Unit Propagation) will affect the estimates in potential search space.

It should be reasonably obvious that any search problem modelled with Kappa can follow our proposed scheme to derive a suitable and effective branching strategy. The weighting scheme is derived directly from the model for $\langle Sol \rangle$. The priority function can be derived in a similar fashion to ours, but must account for multiple branching and greater domain size. This means that the priority function may be expensive to compute, but approximate alternatives may be derivable. We comment further on this particular issue in the discussion of our experimental results below.

2.2.4 Accounting for Dependence

One problem with the weighting scheme shown in this chapter is that it assumes that no dependencies occur between variables. This is a possible source of inaccuracy. A full calculation accounting for all dependencies is tantamount to solving the problem. However, there is one case of dependence that is immediately obvious. The weighting scheme calculates values using sets of clauses with common occurrences of a single literal. We can derive an expression which more accurately depicts the search space reduction from a set of clauses which all contain the same fixed literal. We ignore the possibility that the other literals may occur more than once in this set of clauses. The expression can be found by simply observing the distribution rule[1] for the set of clauses e.g.

$$a \wedge ((b \vee c) \wedge (d \vee e) \wedge \ldots)$$

which extracts the common literal leaving an "independent" set of clauses. Thus the expression for the search space reduction for the assumption a when it occurs in n clauses of length i is

$$\frac{1 + (\frac{2^{c-1}-1}{2^{c-1}})^n}{2}$$

This expression shows that the search space reduction factor for a particular clause size is dependent on the number of clauses of that size. Unfortunately the expression does not allow the reduction to be easily calculated in exponents of 2, as in the proposed weighting scheme. We were interested in how significant the addition of this dependency was on the theoretical branching scheme. Not only is the calculation of a weighted score more complex, but it is also susceptible to computational errors such as round-off errors. To completely eliminate this problem we implemented this weighting scheme within the proposed theoretical branching scheme using the `gnump` multiple precision libraries[2]. Each reduction factor was represented as a rational number. The computational cost is very high for such approaches, but we were interested in whether the branching scheme detected better candidates and thus reduced the search tree. We compared the search cost, in terms of branch points, with the original implementation of the branching scheme. We found no significant difference in performance when solving a small range of random 3-SAT problems. It is possible that for these problems the number of clauses in occurrence counts, especially binary counts, may not have always been large enough to create a substantial change in the estimate. While it is interesting to observe that the weighting values are not independent of the occurrence counts, we are led to believe that this is far less important than other dependencies related to score estimates. This is made more apparent in the next section.

[1] Thanks to John Slaney for pointing this out.

[2] `gnump` is a freely available library allowing arbitrary precision for integers, rationals and reals. It is available from `http://www.swox.com/gnump`

2.3 Experimental Analysis

The actual performance of the theoretical scheme was investigated by comparing several different approaches to branching schemes on a range of random 3-SAT problems. The empirically motivated schemes are generally derived from their performance on these problems, and this means that the comparisons will be made against the best known branching schemes for this domain.

Other well known methods are included for comparisons. Some obvious modifications to the basic theoretical approach were also performed to illustrate their effect. Each method used in the experiments is discussed in detail below.

2.3.1 Method

Random 3-SAT problems were generated with a clause to variable ratio of 4.3, which is in the "hard" region for these problems. This also produces a reasonably even distribution between satisfiable and unsatisfiable instances. For all instances we tested, the fraction that were found to be satisfiable was about 44 percent. We look at problem sizes ranging from 120 variables to 220 variables in steps of 5. For each point 500 problems were used. The same set of problems was used at each point for every branching scheme tested. All the experiments were carried out using a Linux operating system with an AMD K-6 500MHz CPU with 128Mb of memory.

Each implementation of a branching scheme uses exactly the same C++ code, bar the function which selects the next variable in the DPLL algorithm. This means that all the schemes are on a even footing. We didn't optimise any scheme, but we attempted to implement each choice function as best we could. Each choice function is based on the same skeleton code where a loop identifies all unassigned variables and accumulates the appropriate occurrence counts for that scheme. It calculates the score for each candidate, and keeps track of the best found so far. The rest of the code implements a basic DPLL algorithm using Unit Propagation and Pure Literal detection. An indexing scheme recording the occurrences of literals in clauses is used to enhance traversal of the formula during unit propagation. It also aids in giving the DPLL routine control of the state of the formula, rather than leaving that to the call-stack frame. There is much scope for other optimisations, the effects of which would be linear.

The number of branches recorded is the number of internal branch points in the search tree. In this case we just increment the branch count when the assumption of a variable leads to contradiction. At the leaf nodes of the search tree contradiction is detected by Unit Propagation. The branch count is not incremented when a solution is found under the assumption at that branch point i.e. the first branch taken from that point. The method essentially counts each node in the search tree that has two children. This branch counting method does not significantly differ from other methods commonly used. The time taken to solve a problem is the CPU time for the DPLL algorithm to terminate.

2.3.2 Branching Schemes

The branching schemes that were used in this experiment are now described in detail. For simplicity each method is assigned a simple descriptor which is used in the performance graphs and the following discussion.

MOMS This is the basic MOMS approach as discussed in Section 2.1.2. To rank the candidates we used the priority function $P(x) = W(x) + W(\overline{x})$ where $W(x)$ is the number of occurrences of the literal x in the minimum clause size in the formula. For random 3-SAT problems this size is almost always 2. This method is essentially the first priority function used in the `tableaux` system [26].

2SJW This is the two sided Jeroslow-Wang branching scheme that was found to be most effective by Hooker and Vinay [56]. The details are given in Section 2.1.1.

H(x) This is the combination of Freeman's priority function defined in Section 2.1.2.1, with a weighting scheme that uses the empirically derived weights from Kullman [65] and Li and Anbulagan [71] (since the experiments use 3-SAT problems the weighting values are the same). To the best of our knowledge this is the best known approach for a branching scheme that does not use any other additional techniques (such as look-ahead or relaxation methods). Freeman also claims that his priority function betters all its predecessors [36], and we know of no newer priority functions which don't use adjunct methods such as look-ahead. This combination of priority function and weights is one of the branching schemes from `satz`.

T(x) This branching scheme is the theoretically derived scheme we have developed in this chapter. We choose to select the branch giving the smallest sub-problem first, which corresponds to the branch with the highest weight score. If the problem is unsatisfiable it does not matter. If it is satisfiable and we have chosen an unsatisfiable branch, then at least (we hope) that the search space beneath it can be traversed faster than its counterpart. A counter-argument is that, since solutions are more likely to exist in a less constrained problem we should choose the problem that results in a greater potential search space first. Some preliminary experimental results for 3-SAT problems show that the counter argument is better, but the actual order makes very little difference, probably due to the extremely conservative nature of the balancing performed by the function. For more advanced priority functions or different search algorithms we suggest that experimental analysis may be required (see Section 2.2.3).

T′(x) This is a modified version of the theoretical scheme $T(x)$. We realise that there is a potential for inaccuracy in the theoretical weighting scheme. One example is the dependence between variable occurrences. A more likely problem is the fact that the weighting scheme cannot account for the gain and loss of clauses that arise from Unit Propagation.

Our approach here is to modify the weight values by empirical analysis. We wish to demonstrate that the theoretical weights cannot completely account for the state of the search space after a choice is made. Furthermore we wish to illustrate that a fairly rough and ready approach to tuning can result in performance gains. This is particularly significant for those who are interested in deriving a branching scheme using the Kappa model. As was the case for experimenters before us, our method may suffer from over-fitting. This may not matter for those who wish to identify branching schemes for specialised search problems.

To find better weights we sampled some points, keeping c_1 as unity and decreasing the second weight as it is most significant, but also decreasing the third weight in proportion to the second. We looked for some minimum performance value on small sets of random 3-SAT problems with 140 variables. We found many local minima, but found that there was a general minimum somewhere around the constants that we use for this experiment. We then took a few samples from problems of size 160 and 180 variables to roughly confirm that minima were consistent with a change in problem size. The empirically adjusted weights are $c_2 = 0.14$ and $c_3 = 0.03$. It seems that changes in these values of around 10 percent do not change the performance significantly. We also note that there is no minimum at $c_3 \neq 0$. The weights are significantly different from the theoretically derived ones. We shall revisit this issue in the discussion of performance results.

We stress here that a comparison of the fitted weights, as in Section 2.2.1.1, would be inconsistent as the function we fitted accounts for both loss and gain of clausal information. The priority function is provided to balance a pair of scores, and it is the weighting scheme which is the predominant factor in the empirical derivation of the new weight values. If we used the weighting scheme $W'(x)$ from Section 2.2.1.1 then the weight values should approach the empirically derived values shown in that section.

LAT(x) One obvious modification is the addition of look-ahead. As discussed in Section 2.1.2.3, we believe that a look-ahead procedure acts as a correction mechanism for a branching rule. To illustrate the effects of this correction we added a simple look-ahead procedure to the theoretical branching scheme $T(x)$. The scheme is not a very good one, but serves the purpose required. This scheme performs the branch choice calculations as in $T(x)$, and keeps all candidate scores. It then performs the branching step on the best 5 percent of the candidates. After a branch has been taken we re-calculate $T(x)$ using the actual gain and loss of information, as it is saved as "undo" information in our backtracking procedure.

Our implementation is very expensive – we did not write specific code to perform the look-ahead, but treated each look-ahead candidate as an assumption. The look-ahead techniques that we know of use special routines for the technique, for example they don't perform subsumption, and they use inexpensive secondary scoring methods. Furthermore our scheme does not use a constant value to cap the number of look-ahead candidates, so the cost just grows with the number of variables.

Although our implementation is naive, it does provide us with a comparison that will allow us to see how accurate the branching schemes are when they do not predict the effects of unit propagation. In the experiments for this approach we did not count a failed literal that was detected by the look-ahead scheme as a branch point. This case is considered to be an implication of the previous branch.

It is finally noted that preliminary experiments were performed with an assortment of branching schemes, including some with one-sided priority functions. An approximate version of Walsh's Kappa heuristic [110] was created by using the theoretical weighting scheme combined with a priority function that sought to maximise the potential "solution" or search space. For all cases of one-sided priority function branching schemes the experimental results agreed with Hooker and Vinay's observations [56] that they were significantly worse when compared to branching schemes using two-sided priority functions. One-sided branching schemes were not included in the experiments presented below.

2.3.3 Search Cost Results

The median search cost in terms of the number of branches taken for all branching schemes described in the previous section are shown in Figure 2.1. The three most expensive schemes (without look-ahead) are MOMS, 2SJW and T(x). The search cost

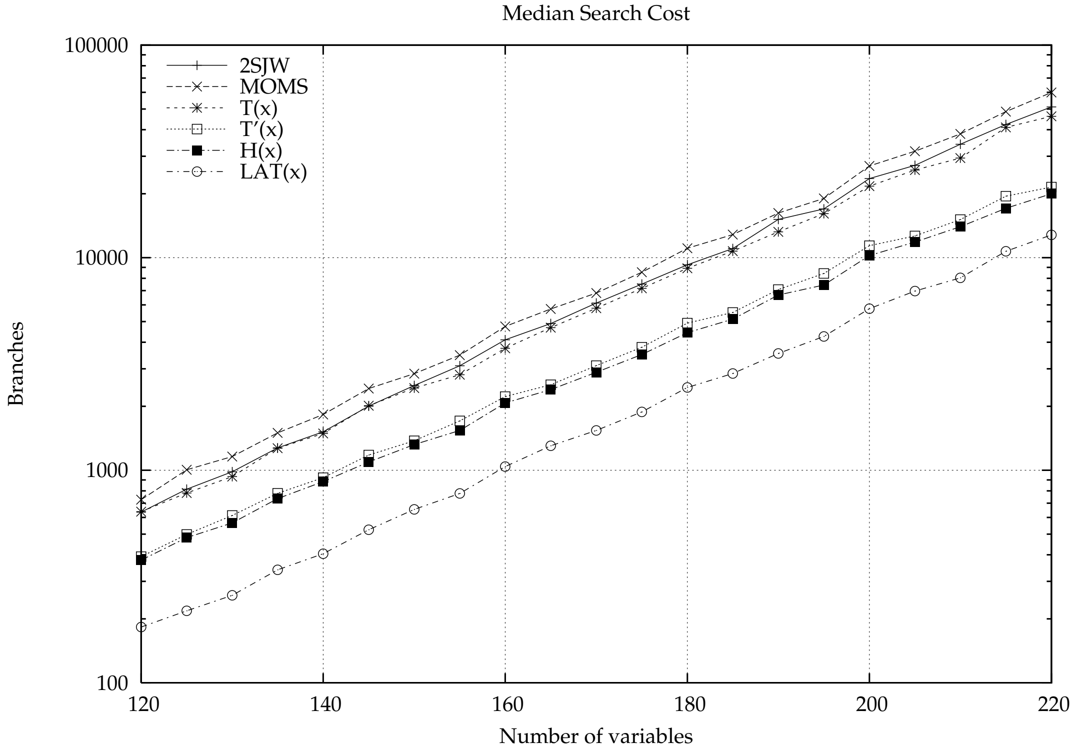


Figure 2.1: Median search cost for hard random 3-SAT problems measured by the number of branch points in the search.

is very close, though it is apparent that T(x) performs somewhat better than the other two. Our theoretically derived scheme is able to compete with well accepted schemes without any experimental interaction.

The two next best schemes are H(x) and T′(x). It is apparent that the fully empirically derived method is slightly better. We do not know whether an exhaustive and methodical search for best parameterisation of the T′(x) weight values will be able to perform significantly better than H(x). However, in our experience, we don't believe it likely. Given the amount of research and time spent on deriving a system like H(x), it is likely that the results are the optimum obtainable from a branching scheme that only uses the information available to it at that branch point. Including polynomial time inferences would refine the information (i.e. reduce the formula to a simpler state), but ultimately branching will not be deferred.

We think that, after whatever refinements are made, the only way to significantly do better branch selection is to predict better the effects of a particular choice. This is supported by the performance results of LAT(x). It is clear that the effect of Unit Propagation must be very significant in calculating the best candidate. The LAT(x) scheme uses candidates selected by the T(x) scheme, and then differentiates between them by computing the actual value of the search space reduction. We suspect that the difference in weight values between T(x) and T′(x) is explained by this effect. The lower weight values in T′(x) acts to compensate the effect of Unit Propagation. Recall that the first weight value c_1 is the dominant weight, and that it weights the occurrences of the candidate literal in binary clauses. A larger difference between c_1 and c_2 means that the scheme will prefer to gain inferred units over the gain (and loss) of binary clauses. In the greater scheme of things this compensation can be interpreted as representing the likelihood for greater search space reduction through the use of Unit Propagation.

Note that since the first round of candidate selection must still estimate the potential effect of Unit Propagation, the scheme T′(x) is more likely to produce a set of look-ahead candidates that will achieve greater search space reduction. The look-ahead process can then discriminate between candidates from a potentially better set. Thus an LAT′(x) branching scheme is likely to be better than the LAT(x) scheme.

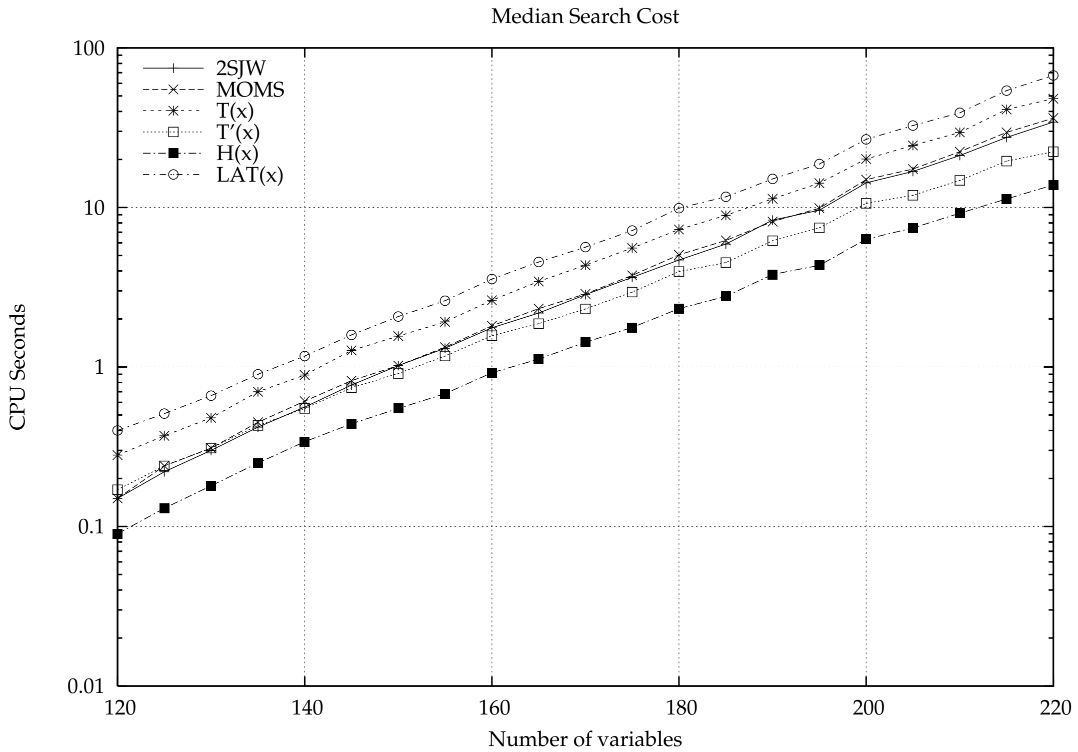


Figure 2.2: Median search cost for hard random 3-SAT problems measured by CPU seconds taken to execute the search.

Branching schemes have an associated execution cost that arises from their complexity. Typically we use choice routines that are bounded by $O(n)$. In general higher complexity routines cannot perform effectively within that tractable window of search problems. Even in low complexity choice routines the relative cost can be apparent, though these effects can disappear as the problems become larger. The success of techniques such as MOMS derives from this property. Unless we are interested in finding the shortest proof of unsatisfiability, as in verification problems, then time is also a useful measure of search cost performance. We note though that time comparisons are not as useful when the actual implementations of search algorithms differ significantly e.g different authors or computer languages. Figure 2.2 shows the median search cost in terms of CPU seconds. As stated previously our implementation for LAT(x) is highly inefficient, and this system is the worst performer in terms of CPU time. In terms of ranking, both T(x) and T′(x) have lost ground when compared to their performance in branches searched. This is entirely due to the cost of calling the `c` language standard floating point mathematics library `pow()` function (for computing 2 to the real valued power $W(x)$ for some literal x) when computing the priority function. There is no other major difference between their computations and the other schemes. In order to remedy this one would need to use an approximation that was easier to calculate (e.g. rounding to integers or a numerical approximation scheme for real powers of 2). Alternatively one could identify a computationally cheaper priority function, such as the function used in H(x). This choice, however, would not be so clear if one was building a branching scheme for a search problem via a Kappa model of that problem domain. The other schemes used in these experiments retain the same ranking seen in search cost measured by branch counts.

Our results show that the theoretically derived branching scheme can compete with other effectively well known branching schemes. We have not compared the scheme to less powerful branching schemes such as input ordering or random ordering. These schemes yield a significantly worse performance when used without other useful methods such as intelligent backtracking. The theoretically derived scheme and modified versions can be improved with optimisation, though the empirically derived scheme is probably close to the optimum performance obtainable through branching schemes alone. A better priority function that models future search space (as in Section 2.1.2.2) is likely to be more effective at reducing search space, but not CPU time. The results also indicate that the function of look-ahead acts as an error correction to the branching scheme, and this is the most likely place where performance gains will be found. An effective look-ahead system must have an effective bounding mechanism on the extra computation, which the LAT(x) system lacks. A theoretical model that relates error correction to cost may be useful, but empirical analysis of CPU time loss versus search space reduction will probably be needed to refine the model.

2.3.4 SAT vs. UNSAT Search Cost Results

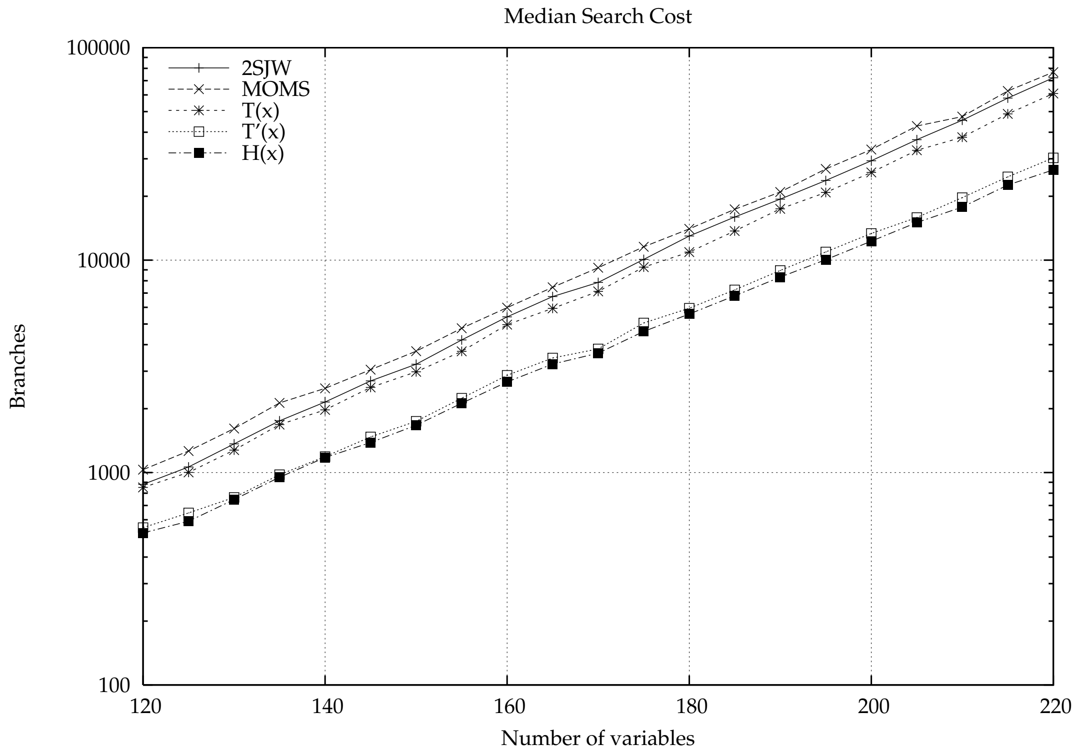


Figure 2.3: Median search cost for the unsatisfiable hard random 3-SAT problems measured by the number of branch points in the search.

A possible criticism of the theoretical priority function proposed in this chapter is that it may be biased to solve unsatisfiable instances. We have argued (and seen in Section 2.1.1) that some level of balancing will always be necessary when solving a search problem using a complete method. The proposed priority function generally fits the theory associated with the weighting scheme derivation. However, it does assume that both branches are always part of the sub-problem. This section compares the performance of the branching schemes used in the experiment, but separates the results for the satisfiable and unsatisfiable problems. The fraction of satisfiable problems from the entire set used was about 44 percent with little variance, so the distribution of 500 problems between the two sets yields a reasonable number of samples for both cases.

Figure 2.3 shows the search cost performance in terms of the number of branches for the unsatisfiable problems instances used in these experiments. We compare the results to the total set by observing the original performance rankings of the branch-

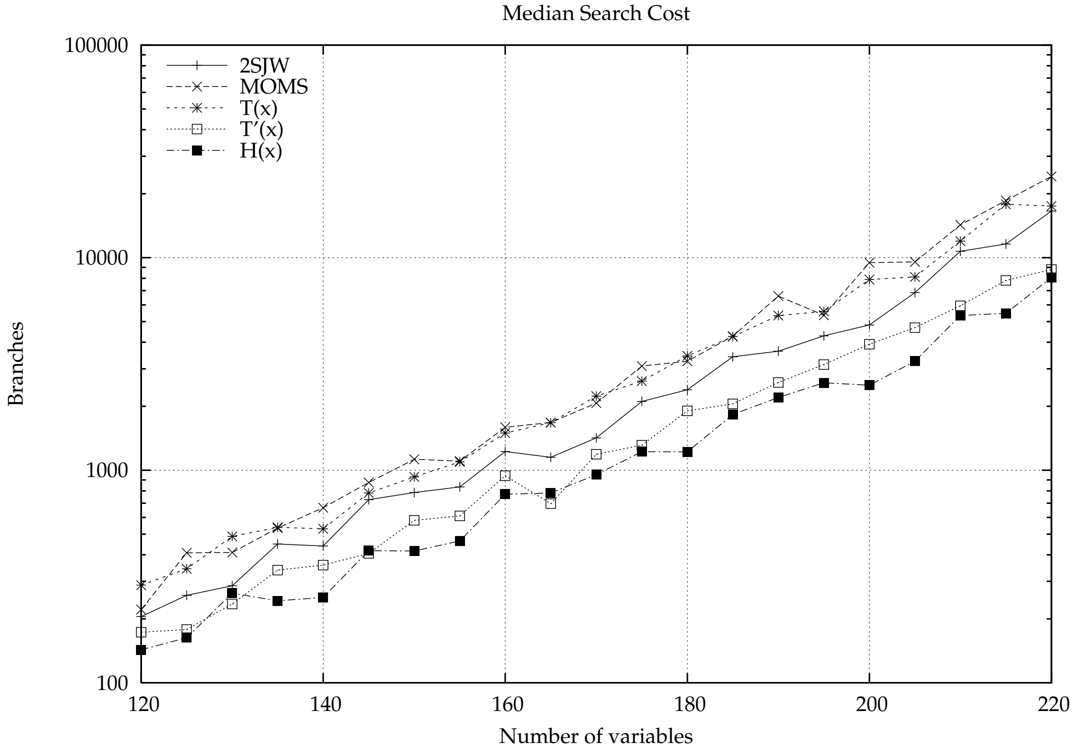


Figure 2.4: Median search cost for the satisfiable hard random 3-SAT problems measured by the number of branch points in the search.

ing schemes. There is no significant difference between the ranking of the branching schemes in these results and the results for all problems. Figure 2.4 compares the search cost for the satisfiable instances. Here the results are more erratic. This is expected, since the search cost measures the amount of space searched before a solution was found, and the variance in this cost is far greater when the instance is satisfiable. It is apparent that the theoretical function and its modified counterpart perform somewhat worse. The effect is more apparent as the 2SJW scheme seems to be doing consistently better than T(x).

There is no significant difference in the ranking of performance for the theoretical priority function when we look at the satisfiable and unsatisfiable instances individually. The CPU time taken is proportional to the number of branches, so the cost in CPU time is indicated by the graphs in the previous section.

2.4 Conclusions

In this chapter we have considered the branching strategy of a search algorithm to be a mechanism which reduces the potential search space. This concept yields a theoretical derivation for a weighting scheme for variable occurrences in SAT formulae. This theory can also explain the values that have been derived using empirical methods for related weighting schemes used for solving satisfiability problems.

The proposed branching strategy includes a theoretical model for a priority function which matches the theory associated with the weighting scheme. The branching strategy attempts to reduce the potential amount of search space after a given assumption, and chooses to branch on variables where this estimate is greatest. This approach is strongly related to the Kappa concept of modelling the constrainedness of search space. By observing the approaches used to find a Kappa model of a search problem one can use the proposed branching strategy to generate an effective choice mechanism for a variety of search problems. This could be particularly useful for problem domains where no useful choice heuristics are known. The approach generated in this chapter produces an effective branching strategy which is far more efficient to compute than previous Kappa motivated branching strategies.

The branching strategy was compared experimentally with other well known approaches on random 3-SAT problems. This problem class has been used to find empirically optimal weighting schemes, and we compare the performance of the theoretical approach to these empirically derived methods. It is apparent that the proposed scheme is able to outperform popular basic branching strategies. It performs reasonably well when used as a general branching strategy, but performance is improved when some empirical tuning is performed. The performance of the tuned branching strategy is almost as good as those which have been completely derived with empirical methods.

The calculation of potential search space is inaccurate due to dependencies in the formula. This can be seen through the dependencies between clauses with common literals, however the dominant influence on inaccuracy appears to come from the new unit clauses generated by a branch assumption. Some basic experiments with look-ahead routines support this notion. Look-ahead can thus be viewed as an error correction for the estimate in potential search space under some assumption. The expense of this procedure can outweigh the savings in search space traversed, but an effective balance between error correction cost and total search cost is elusive.

The proposed branching scheme provides a unique interpretation on the generation and function of branching strategies. It provides an understanding of the branching mechanism which contrasts with previous approaches where very little theory was associated with the branching strategy. The approach used to generate this scheme is general enough that it may be applied in other search problem domains. The theory behind the approach is justified by experimentation and appears to be an effective and promising approach.

Chapter 3

Relevance and Backtracking

One of the key techniques for increased performance of search algorithms is to perform a limited rearrangement of the backtrack order so that unnecessary parts of the search tree can be eliminated. For the purposes of this chapter we shall classify all of these kinds of approaches as *intelligent backtracking* techniques. The essential motivation behind intelligent backtracking techniques is to avoid unnecessary computation which is often referred to as *thrashing* [73]. Although there is no formal definition for thrashing, it can be identified as when a search algorithm performs the same search computation more than once, or includes unnecessary, redundant or irrelevant computation. Intelligent backtracking techniques include redundancy detection, backtrack reordering and saving information in order to *prune* the search tree. The success of intelligent backtracking systems has been demonstrated in many search problem solvers. In particular, the implementations of `GRASP` [76] and `relsat` [10] have been particularly successful in solving a wide variety of real world problems using combinations of intelligent backtracking techniques.

The work in this chapter will discuss some of the problems with classical propositional logic (CPL) which give rise to the inefficiencies in traditional backtracking methods. Previous approaches in intelligent backtracking techniques will be reviewed in this light, along with their motivations and relative abilities to avoid thrashing. The kinds of problems that motivate intelligent backtracking as opposed to traditional backtracking methods are not unique to search algorithms. Approaches in some non-classical logics attempt to eradicate philosophical paradoxes of implication by including the notion of relevance within the logical system, and some formulations of Natural Deduction in classical logics account for relevancy in order to construct concise proofs (see [33, 69]). Some of the problems with classical propositional logic which correspond to the problems of redundancy in proof search are dealt with in *relevant logics*. These logics have been intensely studied and their formalisms and properties are well known. A brief introduction to the practical mechanics of relevant logics, along with a relevant fragment system analogous to clausal propositional satisfiability, will be described later. The fragment system shows the interesting result that while the resultant backtracking algorithm is superior to the classical one due to thrashing avoidance, the formal logical system it is based upon is strictly less powerful than propositional classical logic.

In this chapter it is shown that by observing techniques borrowed from relevant logic, and less restricting logical formulations like Natural Deduction, a simple and effective intelligent backtracking algorithm can be defined. By basing the algorithm on a formal system of deduction, the resulting algorithm and the mechanics of its reasoning become simpler. The work presented is also influenced by the interests of refutation proof generation, as opposed to searching for satisfiability (or a counter-model to the desired proof), and in this sense a backtracking algorithm should aim to reduce the resultant proof size. The approach described in this chapter yields a framework for integrating other known search techniques without any interferences. For example, this includes integration of intelligent backtracking learning techniques and relaxation methods for clausal propositional satisfiability. The framework may also be applied to fuller or richer logics. The algorithm presented is as capable as other general intelligent backtracking mechanisms. Furthermore, the approach also yields a simple and effective method of backward movement in the search space; that is, it allows for proof rearrangement and backtrack point selection with linear space costs.

3.1 Preliminaries

This section serves to introduce both the logical foundations for intelligent backtracking techniques, and the perspective that the work in this chapter takes on tackling the problem of designing intelligent backtracking algorithms. It also reviews and describes previous approaches for intelligent backtracking algorithms.

3.1.1 Search as Proof Construction

The search process can be perceived as a process of *proof construction*. While the aim of intelligent backtracking is to reduce both thrashing and search space, it is also presented here as the equivalent process of proof minimisation, which has useful applications, for example, in the area of stored proof verification [84]. The approach of proof construction focuses the investigation of intelligent backtracking at a purely logical level. This may then be translated back into a familiar Davis Putnam style algorithm with a further level of insight into the problem of relevant proof construction and intelligent backtracking.

The process of solving a satisfiability problem using a complete method can be viewed as attempting a proof by refutation, where the satisfiability problem, in CNF, is assumed to be obtained by negating some assertion to be proved. A satisfiability solver either finds a model which satisfies the CNF formula, or refutes it and thus shows that the original assertion is actually a theorem. The search method is complete as it is able to find a satisfying assignment if one exists. Otherwise it provides a proof that that there is no such assignment. The mechanised system that is the focus of this chapter is the DPLL style approach of proof tree generation which has had great success in solving satisfiability problems. These systems can be seen as equivalent (with appropriate modifications) to similar mechanised (refutation) proof systems such as Tableaux. More powerful systems (e.g. Hilbert, Frege and Natural Deduction) are capable of simulating these mechanised systems.

The aim of generating a small proof should also help to find a model (or satisfying assignment) more efficiently. If less search space is explored then the time it takes to reach or deduce the model will also be reduced. While it is often said that intelligent backtracking systems "correct mistakes" made in the choice ordering, they can also be seen to be pruning the search space, or in the case of an unsatisfiable problem they optimise the resultant proof size. This an important process if one wants to generate a succinct proof that can be verified with a reliable secondary system (e.g. using a `HOL` based system, see [49]). This could also be very important in proof carrying code.

3.1.2 (One of) The problem(s) with classical logic

One of the well recognised dangers of a classical logical system is that from a contradiction one may deduce anything. The mechanised process of a proof by refutation is a process of generating contradictions and then deducing something is inconsistent, thus eliminating (or replacing) one of the assumptions. In a finite domain search problem (e.g. satisfiability of a formula in propositional logic) one may derive "anything", but only from the finite domain. This reduces the aforementioned danger somewhat, but does not guarantee that the use of *reductio ad absurdum* (RAA), a deduction via contradiction, is a useful one. Yet it is this rule that lies at the heart of many mechanised proof by refutation systems. In a simple example, the assumptions C, A, B and the formula $(B \to \overline{C})$ yield $(C \wedge \overline{C})$ and thus we can legally conclude $\overline{A}$, but this particular conclusion may not necessarily be useful in constructing a proof. More complex formulae yield more complex situations. Cancelling assumptions in the order they were made (chronological backtracking) becomes prone to situations like these. Consider a situation where the ordered assumptions are $A_0, A_1, \ldots, A_{107}$ and the formula contains the implications

$$A_0 \to \overline{A}_{109}, \quad A_{107} \to A_{109}, \quad \overline{A}_{107} \to \overline{A}_{108}, \quad \overline{A}_{108} \to A_{109}$$

At the bottom of the search tree $\overline{A}_{107}$ will be deduced using the first two implications (by using the *assumptions* A_0 and A_{107} to produce the *deductions* $\overline{A}_{109}$ and A_{109}, thus inferring contradiction). The last two implications yield another contradiction, but the backtrack order of eliminating assumptions demands $\overline{A}_{106}$ rather than the more expedient $\overline{A}_0$. With this ordering of assumptions the basic DPLL algorithm with Unit Propagation (Algorithm 1.1) will discover this sub-proof 2^{105} times before the search returns to the top of the tree and finally deduces $\overline{A}_0$.

While it is possible that a better branching scheme would avoid such situations, examples can be easily contrived where a given branching scheme fails. The situation can be avoided in all cases by merely remembering those assumptions which led to the contradiction, and only using those as candidates for elimination. Information about dependence for each deduction is needed so that the set of assumptions that derive a contradiction can be identified. There are many names for such a concept: dependencies, conflict sets, uses etc. In this chapter we shall refer to the *dependencies* of a derivation. Previous intelligent backtracking techniques have treated a single dependency as being any instance where a variable takes on a value i.e. both assumptions and intermediate derivations. In this chapter a single dependency can only be an assumption. Each deduction, in the course of a proof, has an associated *dependency set* – the set of assumptions that were used to derive it (though for reasons discussed in Section 3.1.4.3 a mechanised proof system may use a multi-set). For example, if we have assumptions A, B, C and a formula that contains $(B \to C)$ then the deduction C depends upon B only and this shall be expressed by $C : \{B\}$. Note that assumptions actually depend on themselves. The elimination of an assumption via RAA (i.e. the point at which the second path is taken at a backtrack point) is dependent upon those remaining variables that were used to deduce contradiction. For example, suppose we deduce the following with its associated dependency set

$$(X \wedge \overline{X}) : \{A, B, C, E\}$$

then we may choose to negate one of the assumptions, say C, so that

$$(\overline{C}) : \{A, B, E\}$$

Note that the labelling system encodes the possible deductions of implicational formula i.e. implication ($\rightarrow$) introduction as defined by the following rule

$$(\rightarrow \text{Intr.})\ \frac{A : \Gamma \cup \{X\}}{(X \rightarrow A) : \Gamma}\ (X \notin \Gamma)$$

so that, following from the example above, we could deduce the implicational formula

$$(B \rightarrow \overline{C}) : \{A, E\}$$

Indeed we could iterate to obtain $(A \rightarrow E \rightarrow B \rightarrow \overline{C})$: etc. This kind of reasoning is not necessary for Davis-Putnam style search algorithms, but may be useful for extending the capabilities of a relevant proof search algorithm.

We will be primarily concerned with deductions made by methods such as Unit Propagation. Here all we need to define are the rules for operations for making deductions with dependency sets. The Modus Ponens (MPP) rule and the classically equivalent rule for Disjunctive Syllogism (DS) are defined as follows:

$$(\text{MPP})\ \frac{A : \Gamma_1, (A \rightarrow B) : \Gamma_2}{B : \Gamma_1 \cup \Gamma_2} \qquad (\text{DS})\ \frac{\overline{A} : \Gamma_1, (A \vee B) : \Gamma_2}{B : \Gamma_1 \cup \Gamma_2}$$

We can also define a relevant style formulation for RAA which stems from the *Relevant Deduction Theorem* from relevant logic (see below and [33]) and the notation defined above.

$$(\text{RAA})\ \frac{\bot : \Gamma \cup \{A\}}{\overline{A} : \Gamma}\ (A \notin \Gamma)$$

For the purposes of satisfiability search, falsehood or $\bot$ is represented by the empty clause, $()$.

Although these rules are sufficient for the work in this chapter, care must be taken when defining rules for other logical operations. Suppose we define a rule for Conjunction Introduction where the resulting conjunction must obviously rely on the assumptions which derived each of its conjuncts. How do we then deal with a rule for Conjunction Elimination where it is now possible that the result will inherit possibly irrelevant assumptions? Quite possibly we shouldn't have made this irrelevant deduction in the first place. This situation also allows for a construction of RAA which may not be a relevant formulation. Possible discourse concerning such situations is in the realm of relevant logics and is beyond the scope of this work. The concept and mechanics of the deductions sets have been taken from Hilbert or Natural Deduction formulations of these logics. The next section briefly introduces aspects of relevant logics relevant to this chapter.

3.1.3 Practical Implications of Relevant Logic

Relevant logics are motivated by the problems of the paradoxes of material implication within classical reasoning systems. It is beyond the scope of this work to review the enormous body of work on these logics. This section only serves to highlight the mechanics of the formulations of relevant logics, and thus the inspiration of the work in this chapter. Of central importance to relevant reasoning is the Relevant Deduction Theorem which states that any deduction made is always relevant with respect to the hypotheses, that is, every hypothesis is actually used to make that deduction. The reader is referred to work by Anderson *et al.* [3] and Dunn [33] for a more comprehensive overview of relevant logic.

The practicalities of a relevant logic require some kind of formulation for reasoning relevantly (e.g. Hilbert, Natural Deduction). As previously stated the work in this chapter draws from formulations of relevant logics in using the concept of the dependency set to ensure relevant deductions. This idea is not restricted to relevant logics. It appears, in various forms, in intelligent backtracking schemes. It is also a common method to keep track of assumptions or hypotheses in (classical) logic textbooks (e.g. Lemmon [69]), though the original idea appears to date back as far as work by Gentzen in 1934 (see [33]). The work in this chapter is also influenced by the freer nature of the systems that are used to express relevant logics (i.e. Natural Deduction). More mechanised tableaux systems have been formulated for relevant logics, though they are more complex than their classical tableaux counterparts [79, 80]. Alternative algorithmic approaches for non-classical logics have also been investigated [85]. It is finally noted that mechanised implementations of relevant logics exist [106, 14, 95], but such approaches would be impractical for the purposes of propositional satisfiability. The aims of a relevant theorem prover are far different to the highly tuned nature of solvers for clausal propositional satisfiability.

3.1.3.1 A Mapping from Classical CNF to RI

As an aside to the purposes of developing better intelligent backtracking schemes, this section demonstrates how a fragment of the relevant logic R is sufficient to compute propositional satisfiability, requiring only an additional linear space cost. The effect is that an intelligent backtracking scheme is ensured purely due to the nature of the logic. This is interesting because the fragment is a weaker form of classical propositional logic, but using a mechanised relevant system corresponds to solving the problem with an intelligent backtracking scheme. A weaker sub-structural system simulating an advanced search method seems to be contradictory, but in fact highlights the lack of sophistication in current mechanised methods for reasoning about propositional satisfiability.

The following technical discussion is restricted to the well known system of $R_{\neg,\rightarrow}$ also known as RI, the implication-negation fragment of R. For the purposes of this section we keep the description as simple as possible and provide a Hilbert-style formulation. The implicational fragment alone can be easily formulated with Modus Ponens (from A and $A \rightarrow B$ deduce B) and the following set of axioms:

Self-Implication	$A \rightarrow A$
Prefixing	$(A \rightarrow B) \rightarrow [(C \rightarrow A) \rightarrow (C \rightarrow B)]$
Contraction	$[A \rightarrow (A \rightarrow B)] \rightarrow (A \rightarrow B)$
Permutation	$[A \rightarrow (B \rightarrow C)] \rightarrow [B \rightarrow (A \rightarrow C)]$

The implicational fragment of this sub-structural logic corresponds to the weak implicational calculus of Church [20]. To include the negational part of the fragment we need only include an axiom for double negation, a definition of negation itself, and the propositional constant $\perp$ whose value is false.

Double Negation	$\overline{\overline{A}} \rightarrow A$
Negation	$\overline{A} \stackrel{\mathrm{df}}{=} (A \rightarrow \perp)$

Note that a suitable rule for reductio is derivable from this axiomatisation of RI.

It is possible to show that the system RI is sufficient to perform a mechanised refutation proof search with a Davis-Putnam style algorithm for classical propositional CNF formulae.

It is sufficient to translate a given classical propositional CNF formula to an equivalent implicational form for RI and then perform a Davis Putnam style search (or other mechanised refutation system) in order to determine satisfiability. This is demonstrated in the remainder of this section. The system RI guarantees that "relevance" is maintained in the proof, thereby avoiding some of the possible redundancies. The requirements of relevant deduction immediately result in a smaller proof when compared to the same algorithm within the system of classical logic.

To use RI a suitable and efficient (polynomial-time) transformation $\Phi(F)$ must be applied to a given classical propositional CNF formula F. The trick to ensuring consistency between the non-classical and classical system is embedded in this translation. While the translation from a disjunctive to an implicational formula is simple, there are two further considerations: i) avoid duplicate variables inside disjunctions (implications) and ii) allow classical access (via permutation) to the consequent of any implicational sub-formula. Point i) ensures that the complexity of proof search remains the same as the classical system – the relevant system will be concerned with which instance of some duplicated variable was used during proof. Point ii) simply means that the classical conversion of a disjunction $A \vee B$ to an implication should not commit to one particular consequent $\overline{A} \rightarrow B$ but should create an implication for each possible consequent, $\overline{A} \rightarrow B$ and $\overline{B} \rightarrow A$. The following algorithm suffices:

> For a formula consisting of a conjunction of clauses C containing the disjunction of a subset of literals from the set L
>
> 1. Remove all tautological clauses (i.e. containing $A \vee \overline{A}$).
> 2. For each clause, remove all duplicate entries of literals.
> 3. For every clause $c \in C$ where c contains $l_1, l_2, \ldots, l_k$ generate k implicational clauses where, for each l_i ($1 \leq l_i \leq k$), l_i is the final consequent and the remaining literals are the antecedents (not necessarily in order) i.e. one could generate the clauses
>
> $$(l_1 \rightarrow \ldots \rightarrow l_{j-1} \rightarrow l_{j+1} \rightarrow \ldots \rightarrow l_j)$$
>
> for all l_j ($1 < j < k$) and the clauses
>
> $$(l_2 \rightarrow \ldots \rightarrow l_k \rightarrow l_1), (l_1 \rightarrow \ldots \rightarrow l_{k-1} \rightarrow l_k)$$

Steps 1. and 2. remove the parts of classical logic that are incompatible with the fragment of RI generated. They also guarantee that the complexity of the proof search is equivalent to classical propositional logic by eliminating the possibility of combinatorial constructions of possible proofs. In other words, the absence of a refutation proof for *any* search ordering is sufficient to show there is no such proof in CPL whereas in RI this is not the case. Furthermore it eliminates the requirement that each duplicate literal must be discharged separately.

Step 3. converts the CNF into implicational ($\rightarrow$) formulae. Generating implicational clauses for each literal encodes the possible use of the disjunctive syllogism (or, in implicational terms, classical Permutation followed by Modus Ponens) that is required to perform the search. This is normally disallowed in RI, but the extra clauses generated contain the instances where the consequent has already been permuted, thus predefining classical permutation access. The resulting system is a fragment of RI, tailored for relevant satisfiability search on a given classical propositional formula (i.e. the system RI plus the translated formula).

Search for satisfiability could be performed in the usual manner of a (relevant) Davis-Putnam style algorithm. As mentioned above it can be seen as a proof by refutation where the satisfiability problem is assumed to be the negation of the hypothesis to be proved. Any refutation proof method may easily be used, for example a tableaux system for RI [79, 80]. The only difference in this relevant logic version of the Davis-Putnam algorithm is how the Unit Propagation rule is executed. For a given assumption a (or any unit for that matter) the Axiom of Permutation is applied to all implicational clauses that contain a as an antecedent until it is the first antecedent of that clause. For example,

$$(b \to c \to d \to a \to e)$$

becomes

$$(a \to b \to c \to d \to e)$$

after several applications of Permutation. Modus Ponens may then be applied to the clause and the "unit" a to obtain

$$(b \to c \to d \to e)$$

This operation is equivalent to the unit resolution that is normally performed. All clauses containing a as the consequent or $\overline{a}$ as an antecedent are satisfied. All clauses containing $\overline{a}$ as the consequent may have this consequent replaced by the propositional constant $\bot$, and thus contradiction is found when the clause $(\bot)$ is derived – this being equivalent to the classical notion of the empty clause. Housekeeping matters for clause usage remain the same. The system of relevant deduction still requires that any deduction remain relevant, and thus redundancy due to irrelevant assumptions is eradicated, but since the same relevant reasoning is retained, $\Phi(F)$ is satisfiable under RI if and only if F is satisfiable under CPL.

This method illustrates a correspondence between classical propositional search for satisfiability and a sub-structural logic. However, the resulting sub-structural fragment will generate smaller refutation proofs and can prune redundant search space where its classical counterpart will not (although this is still implementation dependent). This is primarily of observational interest only as in practice direct application of the method is inefficient. Most interesting is the fact that the relevant version of this search for satisfiability yields a smaller proof (or pruned search space), due purely to the underlying logic. However, the logical system RI is weaker than CPL. This highlights the fact that the refutation proof search method considered to be the most generally efficient (i.e. DPLL) is only using a fragment of CPL.

3.1.4 Intelligent Backtracking Approaches

Intelligent backtracking techniques arise from the observations of thrashing in traditional backtracking techniques. In terms of refutation proof construction, an arbitrary mechanical order of assumptions is not necessarily the most efficient way to construct a proof.

Several methods appear in the literature, many of which are variants of a few central methods. This section discusses the core approaches for intelligent backtracking schemes. In introducing some of these algorithms the concept of dependency sets and relevant deduction is used. Where necessary the appropriate distinctions between this and the original approach will be made.

The reader is referred to the work of Baker in his PhD thesis [7] which chronologically reviews both fundamental intelligent backtracking techniques and the kinds of thrashing that they eliminate. Baker's work highlights the function of backtracking and associated computational redundancy of different techniques. The work provides an excellent categorisation of the kinds of thrashing that have been observed and controlled through intelligent backtracking techniques.

3.1.4.1 Backjumping

Backjumping is the original approach for eliminating redundant conclusions devised by Gaschnig [40, 41]. This approach tested for redundancy at the leaves of the search tree. The method was improved by Prosser [89] in the system he named *Conflict Directed Backjumping* (CBJ). CBJ performs the redundancy checks at each backtrack point. This method will be described in terms of relevant deductions below. Both Gaschnig and Prosser recognised that a variable may be irrelevant in reaching a contradictory conclusion. The difference between relevant deduction and their techniques is that there is no distinction between the deduction of a variable and an assumption. This is the central difference between the approach presented in this work and all other intelligent backtracking systems. It should be noted that the earlier algorithms were devised for constraint satisfaction problems and that no other logical interactions (e.g. operations like Unit Propagation) were considered as potential additions to the algorithms. This particular issue will be discussed further below. One further difference with the algorithm presented in this chapter is that dynamic modification of a single copy of the formula is performed, rather than making copies or copying via the call stack. This is also discussed in detail below. The *backjumping* algorithm presented below is, in all other ways, equivalent to the original formulation (CBJ) by Prosser.

The backjumping search in the tree is executed in the same way as the DPLL algorithm presented in Algorithm 1.1. The difference is that, since each deduction has an associated set of assumptions used to derive it, the backtrack action can check to see if a backtrack point in the search is relevant, and thus can eradicate it by "jumping" over it. The DPLL procedure is essentially a proof by refutation and the empty clause, $()$, denotes contradiction (thus RAA can be applied). The dependency set for that clause contains all the assumptions that were used to derive it.

3.1.4.2 (Relevant) Backjumping

Algorithm 3.1 Using dependency sets with a backjumping algorithm

$DependencySet$ **BJDP**$(Formula\ F,\ AssignmentState\ A)$
if exists $c : D \in F$ *such that* $Empty(c)$ **then**
  return D
else if $Satisfied(F)$ **then**
  return $SATISFIED$
end if
$a \leftarrow ChooseAssumption(F)$ {Select an assumption $'a'$}
$UnitPropagate(a, \{a\}, F, A)$
$D \leftarrow BJDP(F, A)$
if $D \neq SATISFIED$ **then**
  $UnPropagate(a, \{a\}, F, A)$
  if $a \in D$ **then**
    $D_{\overline{a}} \leftarrow D - \{a\}$
    $UnitPropagate(\overline{a}, D_{\overline{a}}, F, A)$
    $D \leftarrow BJDP(F, A)$
    if $D \neq SATISFIED$ **then**
      $UnPropagate(\overline{a}, D_{\overline{a}}, F, A)$
    end if
  end if
end if
return D

The backjumping algorithm operating with dependency sets of assumptions is shown in Algorithm 3.1 and is named $BJDP()$. The algorithm is designed for presentation and it is noted that several aspects would not be so explicit in an efficient machine representation. This is generally the case for algorithms presented in this chapter. The logical notation used previously is extended in the pseudo code to allow the colon,:, to denote a type which is a pair of types. This is used to provide a simple notation for pairing clauses or literals with dependency sets.

This version of the backjumping algorithm receives two parameters. The first is the formula to be tested which consists of a set of clauses. Each clause has an associated dependency set and this is represented as $C : D$, where C is the set of literals in the clause and D is the set of assumptions used to deduce the state of that clause. The formula in the pseudo code represents a set of these pairs. In this algorithm the deductions are performed in the $UnitPropagation()$ routine which is discussed later. The second parameter to $BJDP()$ is a set of assignments made which serves to record and perhaps report a satisfying assignment.

In the algorithm presented the parameters are passed by reference. This means that there is a single copy of the formula and assignment state, and it is up to the routine to preserve consistency between calls to the routine. Alternative implementations may use the call-stack or make separate copies in order to return the formula to its original state [11], i.e. call-by-value, which is far simpler. The method presented here emulates the call stack and maintains consistency by "undoing" changes made to the formula and uses the $UnPropagate$ routine to execute it. While this method is somewhat more complex, it can be far more efficient, and proves extremely useful for creating the relevant style backtracking system described in Section 3.2.

The routine returns a dependency set upon completion. The idea here is that it reports the assumptions used to refute the search tree created by that call to $BJDP$. A special return value $SATISFIED$ signifies that the call identified a satisfying assignment which is recorded in A. If the given formula is unsatisfiable it will finally return an empty set.

Lines 1–5 perform a check on the initial formula. If the formula contains an empty clause then the assumptions used to derive it are returned. If the formula is satisfied then the special value $SATISFIED$ is returned. For the purposes of the chapter the routine $Satisfied(F)$ is true when all clauses in formula F are subsumed by the current assignment state, and the routine $Empty(c)$ is true when there are no literals in the clause c that can be assigned to make the clause satisfied. These can be performed using whatever mechanism is appropriate to the implementation.

Lines 6–8 select an assumption and propagate its effect on the formula (and assignment state). The $UnitPropagate()$ routine receives the assumption chosen and the corresponding dependency set (i.e. $a \rightarrow a$). This assumption branch of the search tree is executed with a recursive call to $BJDP()$. The assumptions required to form a refutation proof from this point are returned to the dependency set D.

Lines 9–20 check the relevance of the assumption made and perform the clean up when necessary. If the formula has not been satisfied then the effects of the assumption a are removed. Here it is deemed unnecessary to perform any extra work when the formula is satisfied since all that is required is to report this fact and the satisfying assignment (or counter model in the case of proof search).

The relevance is checked by determining whether a was used in the refutation proof after it was assumed. If a was not used then it is not necessary to explore the branch $\overline{a}$, since it can be closed with the same proof found under a. If a was used (was relevant) then the deductive branch $\overline{a}$ must be explored. This deduction is dependent on those assumptions required to close the proof beneath assumption a, hence $\overline{a}$: $D - \{a\}$ so the effects of the deduction are made and the deduction branch is executed (Lines 12-14). This second recursive call to $BJDP()$ returns the dependency set for the branching point. Note that the dependencies for closure under both the assumption and deduction branch are returned as long the dependencies for $\overline{a}$ are actually used in the deduction branch.

3.1.4.3 (Relevant) Unit Propagation

Unit propagation is central to the efficiency of DPLL style algorithms. The "relevant" version differs only from the original by keeping track of the effects of logical operations on the dependency sets. The corresponding unit propagation procedure is listed in Algorithm 3.2 ($UnitPropagate()$), and this shows how the operations on the dependency sets occur. Detail of the basic Unit Propagation process is given in Section 1.3.1. For brevity we only discuss the differences between this version of unit propagation and the basic form. As in $BJDP()$ the parameters of the formula and assignment state are passed by reference.

Algorithm 3.2 Unit propagation for the backjumping algorithm

UnitPropagate$(Literal\ a,\ DependencySet\ D,\ Formula\ F,\ AssignmentState\ A)$

$A \leftarrow \{a\} \cup A$
for all $c : E \in F$ *such that* $a \in c$ **do**
  $MarkSubsumed(c : E)$
end for
for all $c : E \in F$ *such that* $\overline{a} \in c$ **and** $\overline{Subsumed(c)}$ **do**
  $MarkInactive(\overline{a},\ c : E)$
  $E \leftarrow E \cup D$
end for
for all $c : E \in F$ *such that* $\overline{a} \in c$ **and** $\overline{Subsumed(c)}$ **and** $Active(c : E) = \{x\} : E$ **do**
  $MarkAsStandingDerivation(x : E)$
  $UnitPropagate(x, E, F, A)$
end for

At a technical level the issue of dynamically modifying the parameters F and A within the search routine by $UnitPropagation()$ and $UnPropagate()$ must be addressed. Here a simple method is described. Indexing techniques, used in the experiments of the previous chapter, can make things easier. An occurrence list is preconstructed to record, for each literal, a list of references to clauses. This yields an efficient fixed order traversal when applying unit subsumption or unit resolution. Subsumption is performed with $MarkSubsumed(Clause\ c)$ by incrementing a subsumption count that is associated with each clause. When the count is non-zero the clause is subsumed. A formula is satisfied when all the clauses it contains have a non-zero subsumption count. Unit resolution is performed by marking a literal as inactive in a clause. We consider a clause c (disregarding dependency sets for the moment) to be partitioned into two distinct subsets: i)$Active(c)$ contains the remaining literals of the clause and ii)$Inactive(c)$ contains those literals eliminated by Unit Resolution. The routine $MarkInactive(Literal\ l,\ Clause\ c)$ moves the active literal l from the active set to the inactive set for that clause c. A clause is empty when the active part is empty. The corresponding "undo" operations for these routines take the same parameters and just reverse the effects of the "marking". When a new unit is found it is marked as "the standing derivation". The details and reasons for this are discussed below.

The unit propagation scheme discussed here is more than is required for a simple backjumping algorithm, though the essentials for the simpler version should be apparent. It uses a single copy of the formula which may then be efficiently traversed. The extra cost of having to repair the formula later should not be too great, as one should expect (on average) that the changes made are only a fraction of the formula itself. Dynamic modification of the formula also avoids the reliance on the actual depth of the search tree (call-stack) for the state of the formula. This issue is revisited later when a general relevant backtracking algorithm is devised.

Lines 2–4 perform unit subsumption by marking as opposed to deletion.

Lines 5–8 perform unit resolution using the marking technique discussed above. Syntactically, for CNF formulae, the relevant Modus Ponens is applied in the form of a disjunctive syllogism. Those assumptions used to derive a (i.e. D) are now included in the assumptions that yielded c (i.e. E).

Lines 9–12 ensure that new units created as a result of unit resolution are also propagated. The dependencies of a new assignment are those assumptions which were used to derive that unit clause. At this point units can only have been created by the prior step of unit resolution, hence we only consider those clauses which contain $\overline{a}$ (which can be indexed). A unit occurs when the active part of a clause, $Active(c)$, is of size 1. The derivation of this unit, composed of the unit and its dependency set, is marked as the *standing derivation* of that unit value. This is used to keep the dependency sets consistent (see below).

The discussion above shows a method for the dynamic control of the formula's clauses, but excludes using dependency sets. Each time a unit-resolution is made the resultant clause depends on the union of the dependency sets of its parent clauses. Undoing this operation is not quite so simple. Consider the problem where the formula contains

$$(\overline{a} \vee b) : \{x, p\},\ (\overline{b} \vee c) : \{x\}$$

If a is assumed the result is

$$(b) : \{x, p, a\},\ \text{therefore}\ (c) : \{x, p, a\}$$

but subtraction of the dependency sets when removing the assumption a yields

$$(\overline{a} \vee b) : \{x, p, a\} - \{a\}$$

but since the derivation of b depended on $\{x, p, a\}$

$$(\overline{b} \vee c) : \{x, p, a\} - \{x, p, a\}$$

which is wrong as we lose dependency information. The solution is to use a *multi-set* so that after assuming a we get

$$(b) : \{x, p, a\},\ (c) : \{x, x, p, a\}$$

and using simple multi-set difference returns the dependency (multi)set to its original value $\{x\}$. In an implementation the multi-set need only store a count of occurrences for each member. This is a linear cost to the number of variables in the formula. Note here that the use of a multi-set representation is only necessary at the level of reasoning (i.e. unit propagation), and that at the backtrack algorithm level we may, for simplicity, convert any derived dependency multi-sets to simple sets before they are used.

A unit can be considered to be a derivation. For example, if $(x) : \{a, b, c\}$ then $(a \,\&\, b \,\&\, c) \rightarrow x$. It is possible that a unit has more than one derivation. It is also possible that the undo process discovers an alternate derivation before it discovers the derivation used by the unit propagation procedure. Undoing the effects of a derivation must correspond to the original derivation used. The unit propagation procedure selects the first derivation found. Once propagated, this unit subsumes all other possible derivations, and therefore a unit is never derived twice. The first derivation found during unit propagation is marked as the *standing derivation* by marking that clause (Algorithm 3.2 uses $MarkAsStandingDerivation(Clause\ c)$). Once a unit is marked it remains the only derivation considered for that unit. The undoing process only executes a recursive $UnPropagate()$ (Algorithm 3.3) call for a unit that was the standing derivation. This ensures that the correct dependencies are removed from all clauses affected by the instantiation of that unit.

The procedure for undoing unit propagation, $UnPropagate()$, is shown in Algorithm 3.3. It reverses the effects of unit propagation. If the assumption being removed was used in a standing derivation then that derivation will be found in the traversal of the implications that resulted from that assumption i.e. it is found during the process of undoing all unit resolutions caused by the assumption. However, it does not matter if it is traversed in a different order, as long as the correct dependencies are

Algorithm 3.3 Undoing unit propagation for the backjumping algorithm

UnPropagate$(Literal\ a,\ DependencySet\ D,\ Formula\ F,\ AssignmentState\ A)$

for all $c : E \in F$ *such that* $\overline{a} \in c$ **and** $Active(c : E) = \{x\} : E$ **and** $(x \neq \overline{a})$ **and** $IsStandingDerivation(c : E)$ **do**
    $UnMarkAsStandingDerivation(x : E)$
    $UnPropagate(x, E, F, A)$
end for
$A \leftarrow A - \{a\}$
for all $c : E \in F$ *such that* $a \in c$ **do**
    $UnmarkSubsumed(c : E)$
end for
for all $c : E \in F$ *such that* $\overline{a} \in c$ **and** $\overline{Subsumed(c)}$ **do**
    $MarkActive(\overline{a},\ c : E)$
    $E \leftarrow E - D$
end for

used when a unit is withdrawn.

3.1.4.4 Dependency Directed Backtracking

Dependency Directed Backtracking is an intelligent backtracking method which addresses the problem where, in a standard DPLL style search, a particular contradiction may be discovered over and over again. This method, due to Stallman and Sussman [104], records information about discovered contradictions as an extra clause which is called a *no-good*. The set of no-goods is considered to be part of the formula for the remainder of the search. Again this method makes no distinction between assumptions and deduced information. For example, using the notation described in this chapter, suppose a contradiction is derived

$$() : \{a, b, c\}$$

then by adding a clause

$$(\overline{a}, \overline{b}, \overline{c})$$

it is guaranteed that the search will not repeat exploring the search space required to justify that clause.

The drawback to the method is that it may use an exponential amount of space in recording the no-goods. The search algorithm is required to check more clauses each time a no-good is added.

The notion of bounding the amount of space used by no-good clauses is defined in the approach *k-order learning*, due to Dechter [29]. This method limits the size of a no-good to some fixed k, and is bounded by a polynomial. Other methods include the deletion of no-goods which are irrelevant [7, 11]. Bayardo and Schrag use *relevance-bounded learning* [9] which, for some fixed k, limits the size of a no-good to k, but also requires that assignments to variables relevant to that no-good have changed [11]. This means that the no-good is discarded once the search leaves the space relevant to that no-good.

Successful implementations of satisfiability solvers that use Dependency Directed Backtracking generally use a learning method to control its behaviour [116, 76, 11]. It is generally used as an adjunct technique, i.e. in combination with choice heuristic or other search strategies. In this chapter the method of recording clauses or no-goods is considered to be a useful, but adjunct, technique of intelligent backtracking. The idea of remembering sub-proofs is not excluded by the framework discussed in this chapter, as the use of the Implication Introduction rule suffices. The real challenge is to identify a good learning technique to keep the best no-goods, or subproofs, and discard less useful ones. There are other successful approaches that use unbounded Dependency Directed Backtracking and these are discussed in Section 3.1.4.6.

3.1.4.5 Dynamic Backtracking

Dynamic Backtracking (DB) is another intelligent backtracking method motivated by the potential problem of losing work when backtracking over it [45]. Consider the case where backjumping discovers a contradiction deep in the search tree but can jump back over many assumptions and their related parts of the search space. When they are jumped over they, and their implied information, is lost. Yet this work may be repeated in order to complete the search. Dynamic Backtracking addresses these issues.

The procedural method of Dynamic Backtracking is iterative, compared to the usual recursive formulations of backtracking procedures. It utilises a two-dimensional array of binary no-goods or *culprits* to guide a systematic search through a novel construction of the search space. The culprits array represents the cross product of all values each variable may take. Thus conflicts may be recorded in a pairwise fashion. This method has been shown to be useful on some domains of constraint satisfaction problems (CSPs) like crossword solving [45]. The method also allows some freedom of backtrack point choice.

Like the other intelligent backtracking methods discussed previously DB makes no distinction between assumptions made and implied information. When recanting an assumption a possible effect is that any implied information is retained (e.g. new assignments by Unit Propagation). This kind of behaviour means that the implied information does not have to be recomputed. This may not always be useful as the search will treat the remaining implied assignments as assumptions, thereby increasing the search space. Due to this reason, a translation of this algorithm to the framework of dependency and refutation proof sets is difficult, and so is omitted. Baker observed the problem of retaining implied information when experimenting with propositional satisfiability problems using DB [6, 7]. To solve the problem Baker implements an adjunct erasing routine to clean up implied units gained via unit propagation. This yielded somewhat better results for DB when compared to backjumping on hard random 3-SAT problems [7].

While the general form of DB may be better suited to certain domains of CSPs, it is an important development as it re-addresses the problem of search space construction and recognises that seemingly irrelevant but expensive work may not be irrelevant later in the search. It has also led to some other interesting methods in the realm of search strategies. These are discussed in the next section.

3.1.4.6 Other Related Work

The successes of incomplete or local search methods have inspired some research into the hybridisation of non-systematic random exploration search algorithms with the systematic schemes of backtrack procedures. With a greater degree of freedom backtracking can be "restarted" in a different region of search space. Incomplete methods exist (e.g. [63]), but a further challenge has been to devise a complete method.

Ginsberg and McAllester suggest a hybrid algorithm using Dynamic Backtracking that retains completeness while also proving a polynomial bound on the amount of information recorded during the search [46]. This approach stems from work based on Partial Order Dynamic Backtracking (PDB) [78] which is a method that follows observations by Bruynooghe of a "partial order" of variables when executing an intelligent backtracking search [15] – this is akin to keeping track of dependence or partitioning related inferences. It turns out that the work presented in this chapter also takes advantage of the partial order in variable dependence, though in quite a different way to Bruynooghe or McAllester. Due to the complexities of these approaches a discussion and comparison is delayed until Section 3.2.6.1.

Ginsberg and McAllester also develop an approach which has greater freedom of movement, but may require exponential space [46]. Richards developed a system which combines the ideas behind randomised and local search methods with the ideas of no-good recording. The space required may be exponential but the search is complete [90, 91].

The idea behind these methods is this: retain the randomised technique of traversing the search space, but ensure completeness by recording which areas have been searched. A no-good can record where search has been done, and processing of the no-good set (e.g. subsumption) can help to control its size. These kinds of systems have been shown to be competitive with other complete backtracking methods [90] by taking advantage of the abilities of random search. Recording clauses to map the traversal of search space has been shown to be incredibly successful within the implementation of the satisfiability system Chaff for a wide variety of practical benchmark problems [82]. Such systems still use traditional search tree construction methods, but can restart the search and guarantee termination by recording the space already searched as new clauses.

These approaches are hybrids inspired by non-systematic approaches to search. While this chapter centres around systematic methods of the actual proof tree construction, these hybrid methods are mentioned because of their ability to move about in the search space. The question of how a systematic method may achieve similar freedom is addressed later in this chapter. Hybridisation with clause recording systems is relegated to further work (see Section 3.2.8).

3.2 Relevance for Backtracking

The usefulness of intelligent style backtracking has been shown by several previous approaches, many of which address different redundancy problems in proof tree (search space) construction. The methods often differ in nomenclature and description, but these ultimately serve similar purposes. With a somewhat longer history, the practical use of relevant logics has required that the mechanisms that monitor relevance be captured for use within the formal logical systems, i.e. Hilbert style or Natural Deduction systems. This section describes a "Davis-Putnam style" algorithm that is motivated by relevant deduction systems, and yields a relevant backtracking algorithm for CPL. Using a relevant logical system yields a framework that can be used for many search algorithms. We present a basic DP based search algorithm which has similar capabilities to its predecessors, and is described with a straightforward logical system, reminiscent of its relevant logical roots and corresponding formulations. The system presented has the advantage of simplicity, while its malleability yields an ability to both answer and ask further questions about the capabilities of intelligent backtracking algorithms. Initially the system, with formulation, is described and its capabilities noted. A mechanised strategy is demonstrated, and a search algorithm is then shown.

3.2.1 A System to Work With

In a system of Natural Deduction there is considerably more freedom in constructing a proof. This freedom may be desirable for a mechanised proof by refutation system. In this section we will introduce some basic concepts and mechanics of freer reasoning.

3.2.1.1 A Basic Formulation

A simple pseudo-relevant system, for refutation proof or satisfiability checking of CNF formula, and corresponding to DPLL, can be defined by supplying the following rules:

RAA reductio ad absurdum, the mechanism of refutation.

UR unit resolution, which is the disjunctive syllogism form of MPP.

US unit subsumption, used in order to monitor the state of the formula with respect to the assignment state of the variables, thus providing a simple mechanism to detect a counter model or satisfying assignment.

We further define the function

UP(a) which performs all possible applications of UR and US using the (unit) value a. Furthermore it is recursively applied to any unit values resulting from those operations. This function corresponds to the Unit Propagation procedure of a DPLL algorithm.

All of these rules are based on relevant formulations as discussed in Section 3.1.2. Note that in the following discussion, the concept of refutation proof construction is used, and in this case the " given formula" will be the negation of the hypothesis. In terms of satisfiability checking, the process corresponds to searching for or constructing a proof that the given formula is unsatisfiable, and thus the refutation system attempts to show that the negation of the given formula (the hypothesis) is a theorem. The *assignment state* records the values assigned to variables during the search process. It also serves as a counter model or satisfying assignment when the refutation proof fails.

Atomic assumptions can be made, and this will change the assignment state. Both UR and US are operations on atomic objects (i.e. units) and therefore can only be invoked when the assignment state changes (e.g. when an assumption is made or some unit is implied by rule application). Within this system the following restriction is made:

> **Restriction 1:** On changing the assignment state (e.g. making an assumption) the state of the given formula must be completely consistent with that change with regard to UR and US. That is, all possible UR and US operations resulting from the introduction of the assumption (or deduction) must be made. Enforcing this restriction for an assumption a corresponds to performing UP(a).

After enforcing this restriction the formula may contain (multiple) contradictions, i.e. empty clauses with associated dependencies. The usage of RAA is considered later.

3.2.1.2 Erasing and Consistency

In the course of creating a refutation proof with this system, assumptions are recorded in a list. The list is ordered from first to last assumption. For example,

$$T = [a, b, c, d, e]$$

indicates that 5 assumptions were made, the last being e.

The given formula will be consistent with all the assumptions made so far, according to Restriction 1. The "list" actually represents proof progress and is directly analogous to a search tree. However, this will be examined further on.

For simplicity, at this stage in the discussion, only logically consistent formulae are considered, i.e. cases where the formula does not contain any empty clauses. Now suppose that we decide that one of the assumptions (say b from the example) was not necessary and we would prefer to remove it altogether. How can this be achieved without "backtracking", in reverse chronological order, to b and starting again? Normally we would have to perform this as the clauses subsumed by b may be affected by the assumptions made afterwards, both by US and UR, and Restriction 1 must be enforced. The way to solve this problem efficiently is to return to the ideas in Section 3.1.4.3 so that the assumption can be immediately undone, yet the formula remains consistent with respect to Restriction 1.

Consistence with respect to US: First we treat subsumption by marking a clause as subsumed for every assumption made, regardless of whether it is already subsumed. Algorithm 3.2 achieves this by using a subsumption count for each clause. Thus any clause subsumed by an assumption, or its unit implicants, that is later subsumed by another assignment, remains subsumed when the subsumption count is decremented.

Consistence with respect to UR: The second problem is to ensure that clauses that will no longer be subsumed remain consistent with the state of the formula. This can be achieved by applying unit resolution on clauses that have already been subsumed, thereby keeping them up to date. If we remove the test $\overline{Subsumed(c)}$ on line 5 in Algorithm 3.2 the active and inactive partitions of the clause reflect the state of the formula, disregarding subsumption. The corresponding undo operation must also be modified in the same way (line 9 of Algorithm 3.3). We will call these new algorithms $UnitPropagate'()$ and $UnPropagate'()$ which include the performance of unit resolution on subsumed clauses. Note that for a subsumed clause the subsuming variable will always remain in the active partition of the clause, and will never be resolved away. Thus the clause will never qualify as a candidate for unit propagation, as it is subsumed. In terms of cost the revised operations may be slightly more expensive as they will always perform a write (the UR operation) rather than a read (subsumption check) followed by the possible write to memory.

Undoing unit resolution operations only eliminates derivations, and their associated dependencies, which relied upon the assumption being removed. It essentially follows a syntactic trail. It also removes the effects of all the standing derivations that relied upon it, i.e. all implied units that were found directly after the assumption, as well as those that were later derived via other assumptions (see discussion in Section 3.1.4.3). As discussed earlier, it is possible that an independent derivation of the assumption or any of its implicants exists, though this unit clause and its dependencies will have been subsumed by the first derivation. However, this independent derivation may not rely on the assumption being removed. Indeed it is possible that an independent derivation made by other assumptions later in the list may exist. This is dealt with when regarding consistence with respect to UP.

Consistence with respect to UP: On undoing the effects of the assumption, the clauses that were affected by UR and US will be consistent with the assignment state. However, the possibility of unit clauses from the reinstantiated subsumed set means that Restriction 1 is broken, since had these unit clauses not been subsumed they would have been propagated when they were derived. The solution to this problem is to perform a post check for unit clauses that are not subsumed, and propagate them (as the Standing derivation) along with their dependency set. Note that there may be more than one derivation for a subsumed unit, and one may be more optimal than another. Algorithm 3.4 ($UndoAssignment()$), presented below, simply chooses the first derivation found. This becomes the standing derivation for that unit's value. The algorithm uses the $UnPropagate()$ routine as defined in Algorithm 3.3. Modification of the proof progress list is left until later.

Algorithm 3.4 Dynamically undoing an assignment consistently

UndoAssignment($Literal : DependencySet\ a : D,$
$Formula\ F,\ AssignmentState\ A$)

$UnPropagate'(a, D, F, A)$
while *there exists* $c : E \in F$ *such that* $\overline{Subsumed(c)}$ **and** $Active(c : E) = \{x\} : E$ **do**
  $MarkAsStandingDerivation(x : E)$
  $UnitPropagate'(x, E, F, A)$
end while

3.2.1.3 Using Reductio

Along with assumptions, assertions of an assignment via refutation (i.e. uses of RAA) are stored in the list, along with their respective dependency sets. Storing of dependencies for assumptions is trivial (they depend on themselves) and is omitted in the examples. We now make a second simple restriction on this refutation system:

> **Restriction 2:** If the given formula is consistent with the assignment state, and it contains an empty clause, then RAA must be applied. There is one exception: if the formula contains an empty clause which has no dependencies (i.e. the case where RAA cannot be applied to refute an atomic proposition), then this acts as the refutation for the given formula, i.e. the formula has implied contradiction, independent of anything else.

This restriction maintains logical consistency. Note the following: If an empty clause is derived then it must be a consequent of the most recent assumption, or a consequent of the most recent application of RAA. Therefore it will only require at most one change in the assignment state for that empty clause to return to logical consistency. For the purposes of the notation we allow the empty clause (falsehood) to appear temporarily in the list, but given Restriction 2 it must be acted upon.

An *assumption frame* is defined as any sequence of contiguous assumptions in the list, and should formally be considered as a set. Sequences of assumptions are broken by an instance of RAA in the list (though in the examples below we break them with instances of contradiction, i.e. pre-RAA, to aid explanation). Suppose we have the proof progress list $T = [a, b, c, d, e]$ and that a further assumption of f yields the empty clause $() : \{a, c, e, f\}$. This may be represented (temporarily and for the purposes of explanation) in the proof progress list as

$$T = [a, b, c, d, e, f, () : \{a, c, e, f\}]$$

but Restriction 2 requires some action to be taken. The entire list represents an assumption frame, ending with the empty clause, or falsehood. Using the results in the last section we know that we can erase any of the assumptions we choose and still get a formula consistent with the other assumptions without backtracking. Therefore we may apply RAA and recant any assumption required to derive the empty clause. If we choose e, we erase the assumption and assert $\overline{e}$. A third restriction regarding the placement is made:

> **Restriction 3:** Any instance of a derivation via RAA in the proof progress list must appear after any assumption used to derive it.

The important point here is that swapping elements in the list only needs to comply with Restriction 3. A hierarchy is beginning to emerge, but this will be detailed later. Further note that while swapping strategies may make the list more "readable", or more efficiently manipulated by a machine (in light of some set of operations), a second version of some list with assumptions swapped around (complying with Restriction 3) will still be equivalent, in terms of proof search state, to the original list.

The example list is now

$$T = [a, b, c, d, f, \overline{e} : \{a, c, f\}]$$

and this illustrates a simple example of choosing a backtrack point. This ability is extended later.

Given the notation of the list, clearly the aim of the refutation proof is to get an empty clause with no dependencies, thus using the notation $T = [\ldots, () : \{\}, \ldots]$ (noting that this contradiction can be moved anywhere in the list). This can be interpreted as $F \to ()$ (the given formula F implies falsehood). The definition of a complete system is delayed until a few more issues are addressed. Before that, another example is illustrated: Suppose we have a different situation and the list is

$$T = [a, b, c, d, e, f, () : \{a, c, f\}]$$

The assumptions e and f are interchangeable in the list. Changing their position does not change the state of the formula, and if one is undone then the state of the formula is consistent with the remaining assumptions. So, we could have assumed f before e anyway, and changing them now won't make any difference in the greater scheme of things. Suppose that f is chosen as the candidate for RAA. The ability to swap within the assumption frame allows us to place the instance of RAA before e, but after f

(according to Restriction 3). So the list becomes

$$T = [a, b, c, d, \overline{f} : \{a, c\}, e]$$

but we might as well have swapped some of the other irrelevant assumptions around as well and produced

$$T = [a, c, \overline{f} : \{a, c\}, b, d, e]$$

since we could have generated the refutation of f by only assuming a and c, and then made the other (irrelevant) assumptions afterwards. This example shows that the refutation derivation can be "pushed" over the irrelevant assumptions. This directly corresponds to the backjumping mechanism, where the backtrack refutation action happens above irrelevant assumptions by "jumping" over them. The example also shows that, in contrast to backjumping, the irrelevant assumptions are kept. This partially corresponds to the effects of Dynamic Backtracking which keeps all work not related to the "conflict" variables. The relevant reasoning method differs in that it does not keep any information that was implied by the assumption chosen for refutation, as it would be inconsistent with the notion of relevance used in this system. If f implied some interesting information, then we can assume those implicants, in their own right, next. The remaining irrelevant information may be drawn into the proof later via dependencies.

When an assumption is recanted, that is, it is chosen as a candidate for RAA, the fact that it may have been used in a derivation of some other refutation must be accounted for. In this case all instances of RAA that depend on the assumption being recanted must be removed during the process of undoing the effects of that assumption. Erasing an instance of a refutation assignment is simple – it is erased like any assignment. The removal procedure simply follows the syntactic trail of the literal in question, and its corresponding dependencies are taken with it. So during refutation we scan the list, from end to beginning, and remove any refutation assignments that depend on the assumption about to be removed, and also undo the assignment itself. This simple procedure, excluding any other possible proof rearrangements, is outlined in Algorithm 3.5 ($Recant()$). Note that for simplicity the refutation is inserted into the proof progress list at some position determined by the routine $Insert()$. For the purposes of consistency this routine need only comply with Restriction 3, however in Sections 3.2.3 and 3.2.4 this is revisited in far more detail.

Within the proof progress framework alone, assumptions and indeed refutations, can be rearranged in the list as long as Restriction 3 is adhered to. For example,

$$T = [a, b, c, \overline{d} : \{a, b, c\}, e, f, \overline{g} : \{b\}]$$

Algorithm 3.5 Recanting an assumption

Recant$(Literal\ x,\ DependencySet\ D,\ ProgressList\ T,$
$Formula\ F,\ AssignmentState\ A)$

for all $c : E \in T$ *such that* $x \in E$ **do**
$UndoAssignment(c, E, F, A)$
end for
$Insert(\overline{x} : D - \{x\}, T)$
$UnitPropagate'(\overline{x}, D - \{x\}, F, A)$

may be rearranged into

$$T = [b, \overline{g} : \{b\}, a, c, \overline{d} : \{a, b, c\}, e, f]$$

Although the ordering of assumptions is important, it is only important with respect to the derivations which relied upon those assumptions, i.e. use of RAA. Note that the use of rules such as MPP encode the required dependencies so that, if the rule was used in the course of deriving an instance of RAA, the assumptions required are passed on.

Rearranging the list can be seen as a consistent method of rearranging the (partial) proof. Moving instances of RAA to earlier positions in the list is analogous to "lifting a sub-proof". Using such a feature should aim to reduce the resultant proof size, though it may have other applications such as stochastic proof rearrangement in local search.

3.2.2 The T Tree

The logical system and formulation so far described contains operations that directly correspond to a DPLL style search (e.g. UP), but the mechanics of the formulation have some extended capabilities such as proof rearrangement and backtrack point choice. The proof progress list is by no means an obscure abstraction, and the more traditional representation of a search tree can easily be extracted from it. The assumption entries represent the left going, or assumption branches, and the instances of refutation represent the right going, or deduction branches.

An example is shown to illustrate the analogy as well as the concept of rearrangement in tree representation. Figure 3.2.2 shows a rearrangement example that corresponds to a possible rearrangement strategy that prefers to push smaller refutation sub-proofs up. A contradiction is identified at the bottom of the left hand tree, and b is chosen as the assumption to recant. The result is shown in the right hand tree. The effect of the rearrangement strategy is to keep relevant assumptions together (in assumption frames) and push the current point in the proof tree further into search space, a concept dealt with in the next Section.

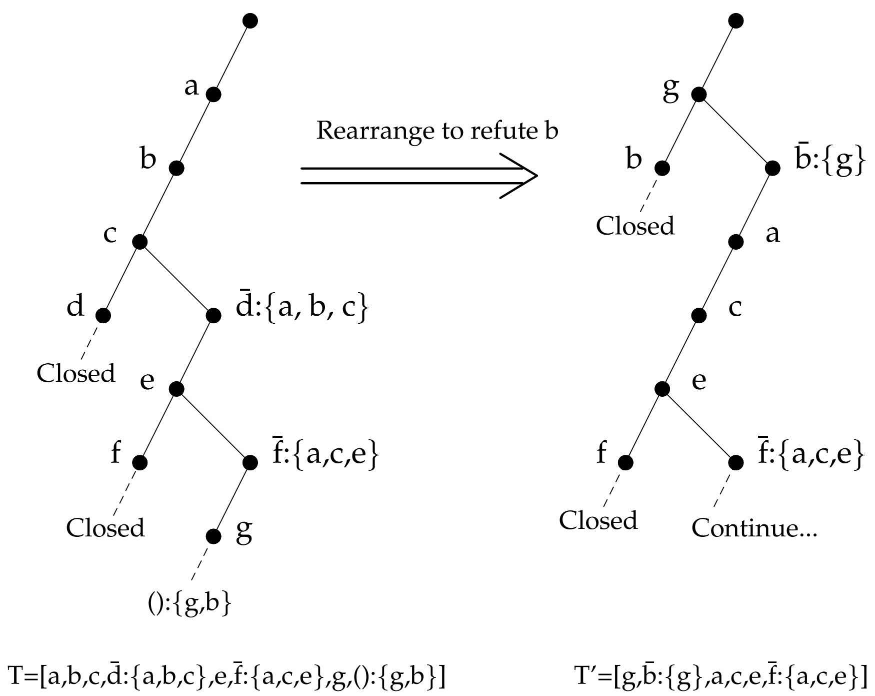


Figure 3.1: The diagrams illustrate the analogy of the search tree to the proof progress list. At the bottom of the left hand tree a contradiction has been found. The right hand tree is a rearrangement where the assumption b is refuted by that contradiction.

3.2.3 Termination and Choice

The sections above have described a system for consistent refutation style reasoning for CNF formulae. There may be proof strategies like moving smaller proofs up higher, but technically this is in the realm of heuristics for backtrack point choice (logical consistency remains the same wherever a refutation appears in the list, noting Restriction 2). A complete method of proof construction requires a strategy or mechanism which guarantees termination. In this section a basic strategy is shown. The strategy calls for a further restriction on rearrangement of the proof progress list (or search tree) to guarantee that progress in the search space is actually being made.

Consider a function $Position(T)$ of the proof progress list which returns a binary number (an integer), or string of 0s and 1s. The total number of digits in this value is fixed to n, the number of variables in the given formula. The list T is read from left to right. Each assumption entry produces a 0, and each refutation entry produces a 1. If the length of T is x, then the remaining $n - x$ digits are padded with 0s. The function $Position(T)$ corresponds to calculating the integer valued position in the search space represented by the analogous search tree of T (as highlighted in the previous section).

When the search begins $Position(T)$'s value contains only 0s. This includes the case where assumptions were made which did not lead to a refutation, i.e. got us nowhere. In the worst (and highly unlikely) case, the search must terminate when $Position(T)$'s value contains only 1s. In this case i) every single assignment has been made via refutation, ii) there are no assumptions so every dependence set for the refutations is empty, and finally iii) the assignment state satisfies the formula if and only if the current state of the formula does not contain an empty clause.

So, progress in constructing a proof could be measured by comparing $Position(T)$ to $Position(T')$ for a rearrangement represented by T'. Therefore the following restriction is made

> **Termination Strategy** In the course of a proof the proof progress list T may only be rearranged to T' if $Position(T') > Position(T)$

What does this mean for our system? It limits the choice of backtrack position. Some examples show the effects. Note that both lexicographic or integer comparisons can be made when determining the validity of a rearrangement. The following rearrangement,

$$[a, b, c, \overline{f} : \{a, b, c\}, d, e, () : \{a, c, e\}] \quad \text{to} \quad [c, e, \overline{a} : \{c, e\}, b, d]$$

(i.e. $0001\ldots$ to $001\ldots$) is allowed, but

$$[a, b, \overline{f} : \{a, b\}, c, d, e, () : \{a, c, d, e\}] \quad \text{to} \quad [c, d, e, \overline{a} : \{c, d, e\}, b]$$

(i.e. $001\ldots$ to $0001\ldots$) and

$$[a, b, c, \overline{f} : \{a, b, c\}, d, e, () : \{a, b, c, e\}] \quad \text{to} \quad [b, c, e, \overline{a} : \{b, c, e\}, d]$$

(i.e. $000100\ldots$ to $00010\ldots$) are not. If we define a routine to make a heuristic selection of the backtrack point (i.e. the assumption to be recanted), it must return a point which will comply with the Termination Strategy.

A simple approach may be to take a copy of the proof progress list, and perform a list rearrangement. Nothing is modified except the copy of the progress list. To test whether refuting a yields a valid backtrack point we perform the rearrangement and then evaluate $Position(T_a) > Position(T)$.

Note there is an alternate method that does not require copying the progress list. This is obtained by observing the pattern of assumptions and refutations in the proof progress list. Treating the contiguous assumptions as assumption frames (see Section 3.2.1.3) it is possible to show that a new assumption frame (and the corresponding new refutation) cannot be inserted above an assumption frame which is smaller (in the number of assumptions it contains). The possibility that an existing assumption frame may be disrupted if its refutation depends on the literal about to be recanted must also be taken into account. Suppose we wish to recant a which will rely on the assumptions in the dependency set D. If we wish to insert the new assumption frame $A_{\overline{a}}$, corresponding to the refutation for a, before the assumption frame A_i, then $A_{\overline{a}}$ will contain those assumptions that do not occur in any assumption frame preceding A_i (recalling Restriction 3). Furthermore assume that the refutation corresponding to A_i (the refutation entry occurring directly after it in the progress list) does not depend on a. Then the proposed insertion may be performed if $|A_{\overline{a}}| < |A_i|$, since this implies that $Position(T_{\overline{a}}) < Position(T)$, where $T_{\overline{a}}$ is the new arrangement. In other words, a better position in the search space is obtained when a smaller assumption frame is inserted prior to a larger one.

3.2.4 Possible Rearrangement Strategies

The termination strategy defined above limits possible backtrack choices but also makes it clear how to evaluate which ones are legal. So, a backward choice heuristic needs to choose a legal point, but how does it discriminate between these? One possible heuristic would be to consider any literal x that contributes to a contradiction and recant the literal which gives the greatest $Position(T_{\overline{x}})$, where $T_{\overline{x}}$ is the rearrangement after refuting x. Note there will be at least one of these, since the minimum case is where the most recent assumption is refuted, thereby changing the corresponding progress $Position()$ value of 0 to 1. This heuristic will push smaller proofs up. This method could further stipulate that, when any refutation is removed because it depends on an assumption about to be recanted, all assumptions immediately above it (the corresponding assumption frame) are pushed down the progress list as far as possible (according to Restriction 3). The effect of this is to retain relevancy for assumption frames and their corresponding refutation, as well as minimising the size of assumption frames. Note that some of this rearrangement may be carried out during the insertion of the new refutation in the procedure $Recant()$. Overall this rearrangement and backtrack choice strategy cooperates with the termination strategy defined above to gain as much ground, in terms of the proof tree, as possible. This greedy and simple heuristic is used later in Section 3.2.7.

The methods discussed in Section 3.1.4.6 are motivated by the ability of randomised incomplete algorithms to move about in search space. Stochastic methods for forward choice in backtracking algorithms have been shown to be successful in combination with "restarts" (Gomes, Selman, Kautz) [48], so it is not unreasonable to suggest that they may also be effective in backward choice. In this light a relevant backtracking rearrangement may be interpreted as a restricted partial restart. Prestwich produced a hybrid incomplete algorithm combining ideas from Dynamic Backtracking and local search or stochastic methods, which yields promising results by allowing an arbitrary backtrack point to be selected randomly [87], though the choice of backtrack point was completely unrestricted. It appears that previous investigations of stochastic approaches suggest that investigating a randomised and complete backtracking may be fruitful.

Another possibility would be to use the theory introduced in the previous chapter. We can evaluate the potential search space, and attempt to approximate the change in this for various backtrack point choices. It would also be possible to use a lookahead strategy for backtracking, i.e. several backtrack choices could be evaluated so that a better estimate of the change in potential search space was found. A simple approach would be to execute the backtrack lookahead on copies of the formula, assignment state, and progress list, though a dynamic version may also be constructed. A truly effective technique would interact with a suitable forward choice mechanism, defining a true proof strategy to "concentrate" on generating smaller proofs. This relatively complex situation is relegated to further work (see Section 3.2.8).

Algorithmic details for a backward choice function, say $ChooseLegalRefutation()$, would depend on the heuristic chosen. For the purposes of this chapter it is enough to show that such a routine can choose to make a refutation which guarantees that the search progresses forward and thus will eventually terminate. The termination and rearrangement strategies presented have placed restrictions on the backtrack choice mechanism, and we finally note that different strategies may result in different restrictions.

3.2.5 A Relevant Backtracking Algorithm

A pseudo relevant logical system has been defined for refutation style proof, consisting primarily of a unit propagation routine accompanied by the rule RAA. The relevance was obtained by embedding relevance in the logical system. It was shown that it is possible to mechanise the unit propagation routine efficiently so that there was considerably more freedom in proof construction, yet consistency was maintained in accordance with Restriction 1. This was achieved by inventing techniques that enabled the mechanisation to operate on just a single copy of the formula. The proof progress list serves as a mechanism to record assumptions and deductions via refutation, but also mimics the traditional search tree representation of refutation proofs. By extracting the tree structure into an abstract object the proof process no longer relies on a fixed call-stack based method of proof tree construction and thus may be controlled by the system. A mechanisation of refutation was devised in order that relevant style backtracking could occur, along with the possibility of proof rearrangement. This also leads to a guarantee of termination, where only rearrangements that move forward in the search space are allowed. This yields a system of mechanisation where the proof tree, i.e. the object traditionally defined by a backtracking mechanism, is abstracted and defined by the way in which we wish to reason, which is relevantly and as freely as possible.

The mechanised system offers two basic operations, assume and recant, which are derived from the familiar DPLL style unit propagation and backtracking method respectively. Keeping in mind the Restrictions defined for the system we now define an abstracted relevant backtracking routine ($ARBT()$) in Algorithm 3.6. The algorithm is quite simple, relying on the embedded reasoning and proof construction process provided by previously defined routines.

Algorithm 3.6 An abstracted relevant style backtracking algorithm

$AssignmentState$ **ARBT**$(Formula\ F)$
1: $AssignmentState\ A$
2: $ProgressList\ T$
3: **while** $\overline{Satisfied(F)}$ **and** $() : \{\} \notin T$ **do**
4:   **if** $() : D \in F$ **then** {Recant}
5:     $b : D \leftarrow ChooseLegalRefutation(T, F, A)$
6:     $Recant(b, D, T, F, A)$
7:   **else** {Assume}
8:     $a \leftarrow ChooseAssumption(F)$
9:     $Append(a, T)$
10:     $UnitPropagate'(a, \{a\}, F, A)$
11:   **end if**
12: **end while**
13: **if** $Satisfiable(F)$ **then**
14:   **return** A
15: **else**
16:   **return** $\{\}$
17: **end if**

Lines 3–12 perform the simple task of either applying a refutation (Restriction 2) or making an assumption and propagating it (Restriction 1). The loop continues until a satisfying assignment is found, or until the formula is proved unsatisfiable.

Lines 5–6 enact upon a contradiction by selecting a legal backtrack point and perform that refutation. This includes making any rearrangements that may be performed. Note that the choice of backtrack point is aware of how a refutation rearrangement is performed in order to determine whether it is legal according to some termination strategy.

Lines 8–10 perform the familiar process of making an assumption or choice.

Lines 13–17 are responsible for returning the result of the search.

Abstraction of the underlying logic, and mechanised proof construction yields a very simple system. The places where the algorithm departs from these abstractions and ventures into the realm of heuristics are made obvious. At this level adjunct methods, such as learning techniques, may also be incorporated.

3.2.6 Soundness and Completeness

The relevant reasoning system is used to capture a smaller proof or prune the search space of the classical domain. As outlined in Sections 3.1.2 and 3.1.3 the proposed system provides a reasoning framework suitable for satisfiability search for CNF formula in classical propositional logic using a DP style algorithm. The issue of soundness and completeness for the relevant backtracking algorithm presented in this chapter can be shown to rest upon the soundness and completeness of DP and intelligent backtracking improvements. The details are outlined below. The argument is that prior algorithms can achieve the same search flexibility by simply restarting the search with a new forward choice ordering in order to reach the desired rearranged state. Without proof rearrangement the relevant framework yields a perfect simulation of Conflict Directed Backjumping - it uses the dependency set information from a conflict to determine the the most recent assumption that caused that conflict. By distinguishing assumptions from deductions it can determine the correct backtrack track point instantly, rather than having to determine at what point a variable involved in a conflict was derived. While there are a number of possibilities for "book keeping", the dependency set approach is tailored for an overall integration of the underlying logic (e.g. as presented in Section 3.1.2).

Suppose we use a CBJ algorithm with very basic clause recording facilities. It allows up to n recorded clauses for a given formula with n variables – given the variables $v_1, \ldots, v_n$, a clause is recorded in the i'th recorded clause position when a backtracking to the variable v_i. Note that dependency sets are used to keep track of the conflicts in terms of assumptions i.e. the direct source of the conflict is recorded. As a separate computation, but alongside this algorithm we use the relevant reasoning book keeping process comprising of any further dependency set calculations and proof/path list maintenance. This resulting algorithm, a combination of the intelligent backtracking search and relevant reasoning proof/path list computations, will be referred to as RS. In terms of soundness and completeness, RS is a standard DP style algorithm with a fixed forward choice ordering, intelligent backjumping and a limited clause recording technique, for all of which soundness and completeness is assumed.

Allow the algorithm execution RS_1 to start a search on a given formula F. Every time RS_1 backtracks (uses RAA) without rearrangement it records this as a clause. At some stage in the search it is advantageous to perform a rearrangement from the path list P_1 which represents the point in search space RS_1 is in, to some path list P_2. Now a second search execution, RS_2, is started. It inherits any recorded knowledge from RS_1 as they are just logical consequences of the given formula F. The difference between RS_1 and RS_2 is that the order of choice in the search tree is determined by P_2. Some branches are closed early (refuted) due to the knowledge about the problem that the execution RS_1 recorded. In terms of formula and assignment state RS_2 can be seen to "catch up" to RS_1. The number of steps to do this is just the length of P_B since refutation computation is "saved" in the recorded clauses. The combination of RS_1 and RS_2 (and any further RS_i's used for other rearrangements) comprise a full simulation of relevant backtracking by restarting a known sound and complete algorithm. The termination argument of Section 3.2.3 still holds as well – a restart guarantees a progression in search space. In the final execution, RS_e, a solution is discovered or the formula is found to be unsatisfiable in the usual manner. The simulation is thus sound and complete, so the original in-situ approach described above is also sound and complete as long as it maintains the same logical consistency between assignment and formula as the simulation (see Section 3.2.1).

3.2.6.1 Related Work – Partial Order Backtracking Systems

In Section 3.1.4.6 the issue of systems taking advantage of a "partial order" for variable conflict analysis was mentioned. The discussion was relegated to this section for reasons of complexity and comparison to the work in this chapter. Bruynooghe's method of intelligent backtracking [15] uses a *cause-list* to keep track of the reasons why a conflict was generated for a particular assignment. At a basic level the cause starts out as a clause which has identified a conflict in the assignment, and this is similar to the way Dynamic Backtracking behaves. The partial order arises from the ordering of the assignments of variables involved in a cause-list. The partial order determines the order of backtracking search and also assures the completeness (and termination) of the search. McAllester's Partial-Order Dynamic Backtracking (PDB) exploits the idea of the partial order and the abilities of the Dynamic Backtracking algorithm to enable some rearrangement in the order of past assignments [78]. To achieve this rearrangement McAllester uses a set of "safety conditions". Here some condition $x < y$ essentially denotes that the assignment of x must precede before y. The safety conditions are initially determined from conflict clauses. Manipulation of variable ordering requires a legal topological sort of the safety conditions to be found i.e. some complete ordering complying with the safety conditions. It should also be noted that PDB operates on a *total assignment* of the variables and that soundness, completeness and termination rest on the properties of Dynamic Backtracking.

While the work in this chapter is quite a different approach, an analogue can be seen in the way the proof progress list is maintained – there are rules and restrictions about how it can be operated on, thus implying some kind of order. This ordering is not only used to operate the intelligent backtracking mechanism, but also to derive desirable properties such as termination. The earlier and cursive work on intelligent backtracking using ideas of partial ordering was not known to us when relevant backtracking was first devised, and while it shares some of the same motivations and base concepts, the resultant incarnation is significantly different enough to make it difficult to clearly demarcate the differences and similarities in behaviour as it is for example, with CBJ. However, the rest of this section will attempt to do this.

Relevant backtracking takes advantage of a partial order concept – it uses it to simplify the intelligent backtracking mechanism and to allow past variable reordering. The definitions provided for the proof progress list carry out the the equivalent function of the safety conditions in PDB, but are also used to guarantee termination (similar to Bruynooghe's approach). The proof progress list provides a given order (loosely, a sequence of partial orderings or sets), and the restrictions on rearrangement are minimal (see Section 3.2.1.3). This seems more succinct and generalised, and perhaps results in the ability to combine the issues of the backtracking mechanism and other algorithmic properties. Furthermore it provides a direct analogy to the traditional search tree interpretation, and in the sense that PDB is based on Dynamic Backtracking, relevant backtracking is based on the Davis Putnam algorithm, if not Natural Deduction.

The fundamental differences between the aforementioned prior work and relevant backtracking is that relevant backtracking reasons from a given set of logical rules that incorporate relevancy (in this case limited to those required to implement Davis Putnam). This means that the rules corresponding to techniques such as Unit Propagation are easily incorporated, whereas it is not immediately obvious how PDB would include additional rules for reasoning (though Baker adjusted Dynamic Backtracking to include Unit Propagation [7], see also Sections 3.1.4.5 and 3.2.7). While previous approaches are still "logically correct", relevant backtracking distinguishes between what it considers assumptions and derivations – the algorithm shown uses assumptions, intermediate deductions and refutation derivations to efficiently and relevantly *construct* a solution or refutation proof. This contrasts to constraint reasoning systems that do not specifically "reason" about the causality or relevance of a particular assignment. For example, for a relevant system the initial derivation of a contradiction relies on whatever assumptions were used to derive it, not just the contents of the violated constraint. Overall, the relevant system could be considered to be a top down approach. In contrast PDB works from the bottom up – it takes a total assignment and attempts to resolve conflicts in clauses by changing the assignment state and later generates inferential knowledge via conflict resolution. Thus the search space traversal enacted by the two algorithms is very different.

From a practical point of view the relevant backtracking algorithm, RB, enforces relevance under Unit Propagation as well as RAA (backtrack conflict resolution) to directly identify the source of conflict in terms of assumptions. The recognition of cause and effect within reasoning means that conflict resolution should be far more efficient. The behaviour is such that RB prefers to stay in a logically consistent state while it attempts to "construct" a proof, whereas PDB tends to resolve conflicts from within an inconsistent state. PDB does not distinguish between a stated "cause" or assumption as it may (effectively) identify, after some number of backtracking steps, an alternative derivation for some intermediate value used to derive a conflict – indeed it could even (effectively) consider the derived value as an assumption. This may be considered as a freedom that RB does not have. On the other hand RB has a recognition of cause and effect (e.g. Unit Propagation or Modes Ponens) and the logical framework allows further logical operations to be captured in the same way. While the two algorithms appear to have similar features, they will can behave considerably differently during search.

The logical basis of the relevant approach not only gives rise to the flexibility of the algorithm derived, but also illustrates a general flexibility in the approach. These issues are highlighted in Section 3.2.8. It is also worth noting that the relevant backtracking framework is based on a syntactic notion of refutation proof search. One could construe an equivalent semantic interpretation suitable for use in a Davis Putnam style algorithm and this would be closer to the way Dynamic Backtracking (ergo PDB) operates (see also Section 3.2.8).

3.2.7 Experimental Analysis

The result of using a relevant reasoning scheme was investigated experimentally by comparing traditional backtracking methods with a possible approach for relevant backtracking. Intelligent backtracking schemes are best known for their ability to prune search space and to recover from bad choices made during search. The systems are compared by adding basic choice mechanisms and testing on random 3-SAT problems. This provides a relatively simple unstructured scenario for evaluating the usefulness of a backtracking mechanism's ability to prune and recover. The experimental method and the backtracking schemes implemented are detailed below.

3.2.7.1 Method

Random 3-SAT problems were generated with a clause to variable ratio of 4.3, which is in the "hard" region for these problems. For each point 200 problems were used. The same set of problems was used at each point for every backtracking scheme tested. All the experiments were carried out using a Linux operating system with an AMD K-6 500MHz CPU with 256Mb of memory.

Each implementation of a backtracking scheme is implemented in C++ and is based on the same basic framework. Linear code optimisation for any given system was not a priority. To evaluate performance the median value of the number of assumptions (or choices) made during search from a set of random 3-SAT problems is used. This measurement is applicable for all systems implemented, and by analogy corresponds to the number of internal nodes in a search tree or proof. Finally it is noted that all of the intelligent backtracking schemes implemented have some overhead in maintaining dependency information, though in all these cases it is polynomially bounded and relatively inexpensive to manipulate.

The most effective method to solve random 3-SAT problems is an advanced choice heuristic. The additional presence of intelligent backtracking makes very little difference to performance, and in the case of systems that perform proof manipulation such as Dynamic Backtracking and possible relevant reasoning schemes, it can detract from the performance as the manipulation may oppose the forward choice strategy. This is discussed later in Section 3.2.8. However, in other problem domains this is not the case. In order to simulate "unknown territory" the a priori knowledge of the effectiveness of branching techniques for random 3-SAT problems was eliminated. A simple fixed branching scheme (a predetermined fixed order of choice) was used initially, and a randomised branching scheme was used in a second experiment to introduce further perturbation. Further details are discussed in Section 3.2.7.3. The use of simple branching schemes accentuates the effect of using intelligent backtracking schemes and simulates unknown problem domains where choice mechanisms are not so dramatically effective.

3.2.7.2 Back Tracking Schemes

The backtracking schemes that were used in these experiments are now described in detail. For simplicity each method is assigned a simple descriptor which is used in the discussions and tables of results.

BT The traditional backtracking approach – no intelligent backtracking mechanism is used. This provides a baseline comparison for the effectiveness of pruning and recovery with the other schemes.

BJ Backjumping or Conflict Directed Backjumping – as described in this chapter.

UD Dynamic Backtracking with unit propagation enabled – the method as described by Baker [7] is followed in order to include unit propagation with Dynamic Backtracking. Without unit propagation Dynamic Backtracking performs very badly [7]. It is noted that Baker embeds some cases of reasoning within his choice mechanism to enforce and optimise unit propagation. The final case, where an assumption or choice is actually made is very basic. In our implementation the choice mechanism is replaced by the one used by all other backtracking mechanisms for an experiment.

Rp Relevant proof search – this is a possible relevant search strategy combining relevant reasoning with proof rearrangement. It cooperates with the termination strategy to select the refutation (backtrack point) that gives the furthest progress in completing the search tree. This approach was outlined in Section 3.2.4. Although the strategy is basic it should concentrate on generating smaller proofs.

It is further noted that the relevant reasoning framework allows perfect simulation of Backjumping. Such a system only needs to use the Proof Progress structure to represent the call stack, and when a contradiction is found it erases all elements from the end to the backjump point. This simulated backjumping algorithm was implemented and was in total agreement with the backjumping implementation used in these experiments. A pseudo-dynamic backtracking algorithm can also be implemented by performing simulated backjumping but not erasing assumptions. When implemented this system showed a reasonable approximation of Dynamic Backtracking, but is not strictly as powerful as Dynamic Backtracking and fares somewhat worse.

3.2.7.3 Search Cost Results

Table 3.1 shows the median number of assumptions required to solve random 3-SAT problems with a clause to variable ratio of 4.3 for a range of problem sizes. The branching scheme is fixed, and due to the nature of the problems is essentially random but will always select the next assumption to be made in a predetermined order. It is clear that, while some irrelevant assumptions are being made, there is no hugely significant difference between standard backtracking (BT) and backjumping (BJ). The nature of the random 3-SAT problems excludes the kinds of complex problem structures that backjumping can effectively manoeuvre in. On the other hand, the Dynamic Backtracking with unit propagation (UD) and the relevant proof search strategy (Rp) have the same capabilities as BJ but perform non-traditional traversal of the search space and retain information about irrelevant assumptions. This yields far better results. Rp is superior to UD for two reasons: i) it uses relevant reasoning which highlights the effects of the assumption on the search tree, and ii) it has a much better ability to rearrange the search order.

Number of Variables	BT	BJ	UD	Rp
60	535	491	277	288
70	1,701	1,585	786	753
80	3,846	3,643	1,666	1,454
90	9,979	9,115	3,982	2,756
100	24,033	22,216	9,422	7,638
110	71,019	65,242	25,841	16,268
120	178,976	158,631	54,477	39,169
130	372,344	347,102	109,094	69,239
140	856,955	767,362	272,774	160,625

Table 3.1: Median number of assumptions made during search with a fixed branching scheme from 200 random 3-SAT problems with a clause to variable ratio of 4.3.

The actual depth of a search (that is the number of assumptions made at any one time) is not great when compared to the number of variables. By using fixed branching order the search is limited to seeing only a small fraction of the total number of possible assumptions. A random branching order will randomly select from the entire set of unassigned variables. Using such a scheme introduces far more perturbation in the search process by allowing far more variety in the assumptions made. While this approach is simple, it is also effective, and search performance has been shown to be comparable to branching heuristics when used in combination with intelligent backtracking systems [75]. This second experiment is identical to the first, but uses a random branching scheme. The results are shown in Table 3.2.

The results are quite different to the first experiment. For BT, BJ and UD the results are significantly worse. For a fixed branching scheme the sample of assumptions likely to be made is fixed, and is small since the search depth will usually be much smaller than the total number of possible assumptions. For a given single search execution this small set of assumptions may contain a catastrophic choice, but over several problems the number of catastrophic choices is not huge (i.e. catastrophic assumptions are fairly rare). The random scheme is likely to see a much larger set of possible assumptions, and hence a given single search execution is more likely to

Number of Variables	BT	BJ	UD	Rp
60	687	631	342	285
70	2,376	1,821	1,124	710
80	6,204	5,323	3,519	1,349
90	14,378	13,401	8,682	3,222
100	43,755	40,324	28,036	6,472
110	116,985	98,485	83,305	12,402
120	287,638	270,622	213,674	25,254
130	735,298	645,599	654,617	46,831
140	1,930,238	1,686,780	1,672,798	86,004

Table 3.2: Median number of assumptions made during search with a randomised branching scheme from 200 random 3-SAT problems with a clause to variable ratio of 4.3.

commit to a catastrophic assumption.

If we compare UD to BJ the results are much worse than when a fixed assumption order was used. Although UD is better than BJ it is not as significant as it appears to be in the first experiment. Like BJ and BT, UD is susceptible to making catastrophic choices, but appears to have further difficulties. Although UD uses a novel search space traversal, it is still reliant on the assumption order. The "culprit" information it generates and retains relies on the fact that some given assumption was made, but may be recorded in terms of deductions not assumptions. In other words UD cannot distinguish between the effects of the assumptions that were made at some previous point in the search, and the effects of assumptions made later. What it lacks is a notion of relevancy. Retaining extra information that is not directly relevant to the construction of the proof in terms of assumptions appears to confuse the search space traversal.

We finally note that UD includes a lookahead calculation to test early for contradiction. This appears to cause the total number of unit propagation calls to be much higher than expected, though this is not represented in the number of assumptions made. The total number of assignment state changes correspond to the number of times that a single call of unit propagation or undo propagation was called, and thus represents how much time is spent performing or undoing the fundamental unit resolution and unit subsumption operations. In fact with a random branching scheme UD does worse than BJ when the total number of changes to the assignment state are compared. A comparison of the median number of assignment state changes made for the random branching scheme experiment is shown in Table 3.3. It may be possible to re-engineer UD to avoid this problem by reordering the way in which it tests for contradiction, though a relevant reasoning approach appears to be more useful in the

Number of Variables	BT	BJ	UD	Rp
60	28,651	23,905	26,403	10,670
70	114,130	83,192	98,267	32,142
80	340,437	272,689	338,542	69,263
90	890,995	771,270	911,872	192,493
100	2,990,584	2,561,763	3,172,613	425,585
110	8,764,323	6,935,415	10,318,457	925,393
120	23,558,846	20,595,409	28,293,207	1,962,900
130	65,778,851	52,901,704	90,117,272	4,003,213
140	183,909,992	148,943,512	248,541,765	7,771,369

Table 3.3: Median number of assignment state changes made during search with a randomised branching scheme from 200 random 3-SAT problems with a clause to variable ratio of 4.3.

long run. While Partial Order Dynamic Backtracking was not implemented for the experiments, the relationship with Dynamic Backtracking may have yielded similar problems, though another issue would have been how to properly incorporate Unit Propagation.

Rp is the clear winner of all the backtracking systems and when combined with the random branching order exhibits a highly significant difference. More interestingly it performs better in this situation that with a fixed branching order. Using relevant reasoning means that knowledge of cause and effect is acquired. The larger sample set of assumptions can be used to advantage when selecting a backtrack point – catastrophic assumptions can be identified and ignored and useful collections of assumptions can be combined to create a smaller refutation proof.

3.2.7.4 Search Cost Results in Time

Initially we performed a set experiments where execution times were not recorded, as done by Baker [7]. The code was designed to be platform independent and testing and initial experiments were run in parallel on different machines. It was later decided that execution times be included in the experiments, so the experiments were repeated on a single machine: a 2.66 GHz Pentium 4 with 1Gb of memory running a Linux operating system. All data presented in this chapter is from the final set of experiments. There are two main caveats in the comparisons of the execution time measurements: i) the systems used, meaning the implementation and to a lesser extent the hardware used, and ii) the benchmark problems chosen for the measurements. This section discusses first the caveats and then the findings of the execution time comparisons.

The comparison of execution times is certainly useful, but it is difficult to guarantee a level playing field. The backtracking algorithms were designed with simplicity as the primary guiding factor, as building several complex backtracking algorithms correctly within a short time frame is non-trivial. The resulting implementations are quite generic in the sense that they attempt (re)use simple common base objects. The top-level code is far more like pseudo-code than a specific implementation for a given algorithm might be. The common usage of base level objects may appear fair, but in fact this may not be so – a given data structure and algorithm combination can be more effective in different scenarios. For example it is well known that a set of sorting algorithms with the same worst case complexity can behave quite differently on different kinds of input. The key point made here is that a search algorithms effectiveness in terms of execution time will certainly be affected by the data structures and corresponding sub-algorithms they use, and the effects can be quite significant.

Each implementation of a search algorithm is implemented as fairly as possible. Where the object framework yielded an obvious redundancy for a particular algorithm the object was specialised to eliminate the problem. On the other hand there was no real attempt to optimise any particular algorithm, though some generic optimisations are used across the board (e.g. literal indexing). A linear optimisation is one designed to reduce the "at node" cost of execution. There are a number of linear optimisations that may be more or less applicable to a given algorithm, and there may be better ways to implement a given algorithm for a given language or machine architecture. Any one the algorithms we implemented could probably be linearly optimised to attain much greater performance in terms of CPU time. Furthermore, a specific implementation for any of the algorithms may be engineered to be more efficient than the generic design.

The "more advanced" algorithms take much more time per node to perform pruning related computation. This "at node" processing is not optimised, and there is certainly room for improvement. Optimising the data structures and associated operations used would yield significant speedups, for example dependency set is simply based on a list and is not the most efficient representation, but is very simple. In particular the Dynamic Backtracking algorithm due to Ginsberg and Baker uses a "fix" to ensure unit propagation works with the original Dynamic Backtracking algorithm and this is implemented as described and thus remains a "fix" requiring additional "at node" cost. In reality the algorithm could be re-engineered for far greater efficiency (see previous section).

What distinguishes the intelligent backtracking algorithms from the naive backtracking method is their ability to prune search space and thus visit less nodes. This comes at the cost of extra computation. The extra cost must be weighed against the overall effectiveness of the technique. For random 3 SAT problems, as used in our experiments, it is generally well known that intelligent backtracking algorithms do not offer a good solution. The experiments presented in this chapter take advantage of the proliferation of random 3 SAT problems and use them as a vehicle of comparison – the naive backtracking algorithm provides a baseline with which the performance of the other algorithms are compared. The comparison of node counts to the base line value gives us an idea of how effective a given technique prunes search space in a difficult scenario.

Unsurprisingly the execution times for the algorithms using a fixed branching scheme, in Table 3.4, show that the overall cost of intelligent backtracking is too expensive when compared to the naive backtracking method. Though the pruning may win out as the problem size increases, the crossover point may well be past the point of intractability.

Number of Variables	BT	BJ	UD	Rp
60	30	60	100	80
70	110	190	330	250
80	280	510	880	610
90	810	1,530	2,540	1,650
100	2,240	4,350	7,420	5,140
110	7,260	14,670	23,070	13,870
120	19,690	41,330	58,570	43,180
130	45,810	104,250	128,030	92,950
140	112,960	269,150	374,000	280,360

Table 3.4: Median execution time in milliseconds of search with a fixed branching scheme from 200 random 3-SAT problems with a clause to variable ratio of 4.3.

With the addition of perturbation from the random branching scheme the results are different. They are perhaps more indicative of how an effective intelligent backtracking scheme can improve execution time performance. Table 3.5 shows that relevant backtracking is effective enough to win out over the efficient but naive backtracking method when the problem size reaches around 100 variables – this corresponds to the point where BT searches nearly 7 times as many nodes as Rp.

The naive backtracking algorithm is generally more efficient when compared to the intelligent backtracking implementations on random 3 SAT problems. Using a randomised branching order allows the relevant backtracking algorithm to catch up. In this scenario, for problems sets with around 100 variables or more, the extra cost of processing performed by Rp, even using this implementation, is far more advantageous in terms of execution time.

Number of Variables	BT	BJ	UD	Rp
60	40	70	110	70
70	160	230	470	230
80	470	790	1,800	600
90	1,220	2,340	5,370	1,700
100	4,140	8,160	20,250	4,030
110	12,070	23,410	71,510	10,620
120	32,610	73,040	213,210	25,050
130	92,930	201,330	747,540	57,750
140	257,700	628,120	2,114,270	128,030

Table 3.5: Median execution time in milliseconds of search with a randomised branching scheme from 200 random 3-SAT problems with a clause to variable ratio of 4.3.

3.2.8 Extensions – Further Work

It is apparent that the relevant logical framework and its applications have much promise. There are several areas that would be interesting and beneficial to investigate:

Alternative rearrangement strategies and planning heuristics Although the backtrack choice and rearrangement strategy used in experiments was highly successful there may be other alternatives. Randomisation may also be promising, but the results of Chapter 2 suggest that there is a method which can determine which is the better backtrack point from a given set of choices. That is, we should be able to estimate which rearrangement of the proof tree is most beneficial according to a model of potential search space or proof size. A more difficult challenge would be to design a strategy where the choice heuristic and the backtrack strategy cooperated, as alone they may behave in opposition i.e. the forward mechanism will try to plan ahead, but the backtracking mechanism is unaware of this. Ideally there should be an overall strategy (both in planning and refutation construction) for the construction of a proof. There are several further paths of investigation of alternatives to the mechanised proof rearrangement strategies used in this chapter.

Alternative Termination Strategies The relevant reasoning framework is quite malleable. Alternative proof or search space construction strategies may be defined. The method used in this chapter cooperates with the termination strategy, and while there may be better rearrangement strategies, there may also be better termination strategies. One possibility is to combine a learning technique (clause recording) to enhance the termination strategy and possibly provide even more freedom in backtrack point choice (see Sections 3.1.4.6 and 3.1.4.4). This kind of termination guarantee is not new and has been shown to be successful in state of the art systems [82].

Further Techniques and Alternatives Techniques such as learning generally improve search performance. Devising a relevant clause recording technique would not be difficult, and such a scheme would correspond more closely to actual proof construction. Local search methods may also benefit from relevant reasoning as it could make search space traversal more efficient. In both cases it may be possible to integrate a relevant reasoning core into an existing system.

Extension of Logic The logical basis for relevant reasoning, in scope of Davis-Putnam style search, is relatively simple. It would be interesting to investigate extensions for richer languages, not only for arbitrary propositional formulae, but also for richer logics in general. Furthermore it would be interesting to identify at which point the relevant foundations fail for the classical environment.

Semantics Related Approach The approach outlined in this chapter is based on a syntactic method of proof search. As alluded to in Section 3.2.6.1 it would be possible to create a semantic approach insofar as it interprets or reasons with models of assumptions rather than embedding the assumptions as a syntactic addition to the logical operations (i.e. the use of dependency sets as described in Section 3.1.2). Care would have to be taken as to how much of a given syntactic approach could be beneficially captured. It is possible to interpret the approach of the Dynamic Backtracking algorithm and derivatives as semantic style approaches. Ultimately a semantic approach would effectively implement a lazily evaluated lookup table for dependency information. Though there may be limitations on the extent of logical operations, it could well yield faster implementations of the search algorithm.

3.3 Conclusions

This chapter has considered the motivations and logical basis for intelligent backtracking schemes. The approaches used to avoid thrashing behaviour have been investigated through an analogy of relevant proof construction. By borrowing concepts from deduction systems of relevant logics we reconstructed known techniques and constructed new techniques for intelligent backtracking. The concept of labelling deductions with dependency sets of assumptions is derived from the formulations of relevant logic, but remains roughly equivalent to previous approaches to intelligent backtracking. However, identifying assumptions as the causes of conflict, and defining relevant rules for reasoning, led to logically consistent and efficient constructions of other relevant reasoning mechanisms, such as unit propagation. The extraction of the concept of dependencies, by their incorporation into the underlying logic, produces a solid foundation for investigation and design of intelligent backtracking methods and refutation proof construction. The concept of proof tree construction was also abstracted from its usual representation, and this enabled a mechanisation where the tree could be rearranged. The advantage of the approach outlined in this chapter is in its simplicity, and the influence of the freer systems such as Natural Deduction has yielded a simple yet more advanced intelligent backtracking algorithm. Experimental analysis showed that these advantages are obtainable and effective. The performance differences are highly significant when compared to other backtracking methods.

The non-classical formalisation of reasoning with dependencies and elimination of irrelevancy, although far more cumbersome than classical approaches, is precisely the kind of system that intelligent backtracking mechanisms aspire to. Although the underlying philosophy of relevant logics differs from the realm of the classical propositional world of the satisfiability problem, this work demonstrates that the formalised mechanics of relevant logics easily translate to solving satisfiability problems "intelligently". That a fragment of a non-classical system is able to do this highlights the fact that current methods in solving satisfiability problems are only using a weak fragment of classical propositional logic.

A labelled deductive system inspired by less mechanised systems has enabled us to generate a straightforward, but abstracted, relevant backtracking algorithm for the satisfiability problem. It avoids the cases of thrashing handled by previous approaches in search tree construction and has other advantages. The system has an underlying logic capturing relevancy, an efficient mechanisation of "in place" modifications yielding more freedom of movement, and an abstraction of the search tree which enables proof rearrangement and thus backtrack point choice. Separation of these concerns has allowed each problem to be solved individually, yet by modelling the process on both approaches from relevant logic and DPLL style proof construction the solutions can be integrated easily. The implementation overheads remain linear in space and time with respect to the size of the input. We have left clause recording techniques and other heuristic methods aside, demonstrating that these can be used as adjunct mechanisms to backtracking. We believe that further research in relevant proof construction and the design of heuristics for such a system would be fruitful. The concept of proof rearrangement, or non-chronological algorithms, also yields many more possibilities for DPLL style or systematic approaches.

Chapter 4

Structure in Problems

We now change our focus from the search algorithm to the search problem. Identifying a generally efficient algorithm is beneficial to solving "real world" problems, but given that these problems are ultimately the focus of search algorithms we ask: Is there something intrinsic about this domain that could help us understand the more practical side of search? In reality this is a very difficult question to answer as we don't even understand very much about the generic properties of problems in the real world problem domain. In this chapter we investigate the use of real world problem models in order to discover useful information regarding the practical difficulties in search and satisfiability checking.

With the rising interest in the use of satisfiability algorithms to tackle real world problems there still remains a disparity in the understanding and availability of such problems when compared to the oft studied and well understood random 3-SAT problems. Empirically we observe that certain techniques such as intelligent backtracking or equivalence testing are extremely useful for solving real world problems, but this provides little theory about why and how any backtracking search routines behave in the real world domain. The question remains as to whether we can predict and exploit this behaviour to better understand search, and improve current search techniques. One way to begin to answer this question is to provide a parameterised model of a set of real world problems with which we can experiment.

In this chapter we look at ways of attempting to model real world problems. We justify and investigate the use of a random 3-SAT model which attempts real world similarity by incorporating a basic structure of connected clusters, or components, in the problem.

4.1 Modular Structure in the Real World

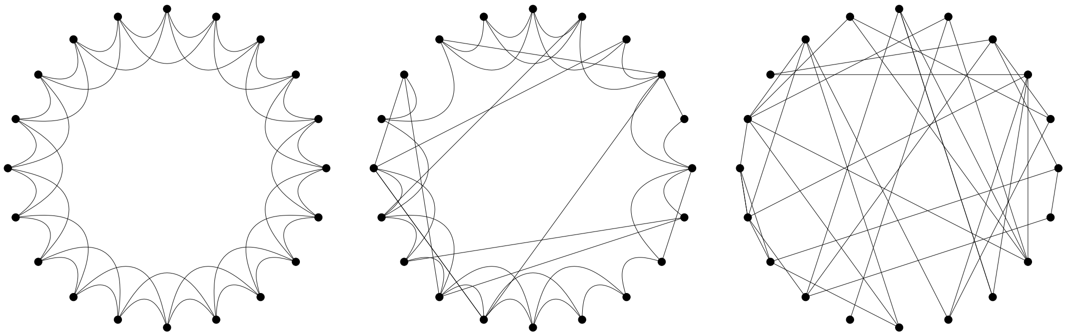
It is evident that, on the whole, real world problems do have exploitable properties which are independent of any particular classification of problem that we want to solve. This is shown by the performance of particular advanced search techniques that are successful on a wide variety of real world benchmark sets. What is seemingly apparent is that real world problems all have some kind of structure; that is, they most differ from random problems in that they have non-uniform distributions and that certain parts of the problem (or inferences from that part) are integral units to the problem as a whole. The latter point is evidenced by the fact that techniques such as dependency directed backtracking, where "sub-proofs" are recorded according to their apparent usefulness, are successful on real world problems [11]. Smaller parts of a problem and a corresponding sub-proof are re-used to solve other parts of the entire proof.

Intuitively we should expect there to be structure in real world problems. Real world problems model various interactions with real world objects. These interactions occur within a structured environment. Consider the problem of verification of hardware or software, where we would at least expect a modular structure in terms of subroutines or logical integrated circuit components. Indeed it is not difficult to envisage that some modular partitioning of many real world problem environments exists. The modularity of such problems will be observed by the clustering of certain groups of variables. In essence each cluster defines a sub-problem that interacts with the rest of the problem using fewer variables than it actually contains.

A theoretical viewpoint to the observation of structure in problems has been investigated by Walsh [111], motivated by the observations of "small world" phenomena observed by Watts and Strogatz [112]. The "small world" model describes a relational system which is somewhere between a completely structured object and a completely random object. Several small groups are highly related, but relations between some members of different groups also exist. These systems can be illustrated and analysed using the network topology of a connected graph. The work in this chapter is inspired by some of the practical implications of that work.

4.1.1 The Small World Topology

In order to capture the notion of the small world topology, Watts and Strogatz combine quantified structural properties of a *connected graph*. A graph topology may be characterised by computing the *characteristic path length* and *the clustering coefficient*. The characteristic path length L specifies the average shortest distance between any two vertices in the graph. The clustering coefficient C is defined to be the average fraction of the fully connected graph that any given vertex may be a member of. More formally, C may be computed as the average C_v for all vertices v, where C_v is defined as follows. A vertex v with k_v neighbours can have have at most $k_v(k_v - 1)/2$ edges between them (the fully connected graph K_{k_v}). C_v is the ratio of the actual number of edges between any of v or its neighbours to the total number of possible edges. These parameters are used to look at a range of graphs from the completely regular, or structured, to the completely random.



(a) Ring Lattice $p = 0$ (b) Small World Graph $0 < p < 1$ (c) Completely Rewired $p = 1$

Figure 4.1: Examples of "connectedness" with a graph of 20 vertices: (a) shows a ring lattice where each vertex is connected to the four closest neighbours, (b) shows a partially rewired ring lattice where each edge is rewired with probability p, (c) shows the graph where every edge has been rewired.

A graph defining a ring lattice, where each vertex is joined to its k nearest neighbours has a high clustering coefficient. We construct a random graph of the same size by randomly assigning the edge connections. Random graphs tend to have a smaller characteristic path length and much smaller clustering coefficients relative to their regular counterparts. With appropriate bounds guaranteeing graph connectivity these two extreme topologies are used to find the small world phenomena.

A small world graph will have a relatively high clustering coefficient as well as a small characteristic path length. Watts and Strogatz use a "rewiring" concept, whereby an edge is rewired to a new destination with probability p. The values of C and L for a ring lattice and a random graph may be interpolated between. This is done by varying the parameter p. At $p = 0$ the graph is completely regular, and at $p = 1$ the graph is completely random. Watts and Strogatz find small world graphs do exist in the interval $[0, 1]$ for p. Their analysis shows a small world network is one where $L \gtrsim L_{rand}$ and $C \gg C_{rand}$, where C_{rand} and L_{rand} are the characteristic path length and clustering coefficient from a random graph with the same number of vertices and edges. Figure 4.1 shows an example of the ring lattice, the partially rewired small world graph, and the random graph for visual comparison.

The details of the rewiring process are less important – similar constructions from

	L	L_{rand}	C	C_{rand}	μ
Film Actors	3.65	2.99	0.79	0.00027	2396
Power Grid	18.7	12.4	0.08	0.005	10.61
C. elegans	2.25	2.25	0.28	0.05	4.755

Table 4.1: Analysis of real networks. The values shown are the characteristic path length, the characteristic path length for a random graph of the same size, the clustering coefficient, the clustering coefficient for a random graph of the same size, and the proximity ratio μ (see below, Section 4.1.2). Note that $L \gtrsim L_{rand}$ and $C \gg C_{rand}$ which also corresponds to the small world characterisation from the Watts and Strogatz rewiring experiments. From [Walsh, 1999].

alternative randomisation models can be made [44]. The point that we make here is that a theory quantifying graph topologies attempts to capture the notion of the small world phenomenon. This can be seen through the experimental use of a theoretical model that combines randomness and ordering. More strikingly it can also be observed in real world relationships. Watts and Strogatz apply the small world topological analysis to some examples of networked relationships and find that the small world network characterisation according to their model is apparent. Table 4.1 shows values for their results on i) Film Actors – from a database of collaborations between actors in feature films, ii) Power Grid – the electrical power grid for the western United States, and iii) *C. elegans* – the neural network from a nematode worm. The data is reproduced from Walsh [111] following Watts and Strogatz [112] and includes the values for the *proximity ratio* which is discussed in the next section.

4.1.2 The Small World Model for Search Problems

Walsh extends the small world analysis by Watts and Strogatz by including the *proximity ratio*. It is defined as a normalised relationship between the characteristic path length and the clustering coefficient of a graph:

$$\mu = \frac{C}{L}\frac{L_{rand}}{C_{rand}}$$

where C_{rand} and L_{rand} are the characteristic path length and clustering coefficient from a random graph with the same number of vertices and edges. This measure allows the "small world" characterisation of one graph to be compared against another. Note that a random graph will have a proximity ratio of 1 and lattices will have small proximity ratios. For small world graphs $\mu \gg 1$. Table 4.1 includes the proximity ratio values from Walsh [111] for the real world networks studied by Watts and Strogatz [112].

Using this measurement Walsh demonstrates that several existing benchmarks for graph colouring have high proximity ratios. Furthermore he shows that high proximity ratios can be calculated for timetabling benchmarks and quasigroup problems. Similar studies showing high proximity ratios for an interpretation of satisfiability benchmarks appears in [44]. In a search cost study for graph colouring Walsh also shows that graphs with a small world topology can be extremely difficult to solve. This effect is explained by the fact that a local colouring choice in a small cluster can quickly propagate globally to many other clusters. The work's conclusion includes comments about the desirability for better problem generators, and that topological features in search problems can have a large impact on search cost.

4.1.3 Real Modularity

We have so far argued that real world search problems are likely to contain some kind of modularity. The observations of small world networks show that this kind of structure does occur, and can be quantified. Walsh has demonstrated that this kind of quantification can be applied to real world search problems, and found that modularity not only exists, but can also make search problems more difficult. This has led to a class of randomly generated problems that are challenging for state of the art graph colouring algorithms.

Network topology analysis for graphs is directly translatable to graph colouring problems. In the case of satisfiability, other notions defining the relations between variables, such as co-occurrence in clauses, have been used to determine the proximity ratio [44]. This relationship is not quite so strong when the clause sizes are larger than 2. Essentially the constraining effects of information is not accounted for – an n-ary clause is converted into $(n^2 - n)/2$ binary relationships which are each considered to be as relevant as the single relationship generated from a binary clause. In terms of the actual propositional reasoning this is hardly the case. Translating a satisfiability problem into a graph colouring problem is not the answer – the structure that we wish to detect (in particular the clustering coefficient) will be hidden by the structure of the translation. (See [38] for details of such a translation.) To accurately capture structure in terms of the logical relationships in satisfiability problems, we need a far better model.

The concept of the clustering coefficient requires that the modularity be exposed at an atomic level. This means that a sub-problem or module is only identifiable when the clustering coefficient in a sub-problem is high, however in practice we should not expect this. It is possible to conceive of techniques which partition a problem into sub-problems in a different fashion but this is not the focus of this chapter. Our review of research on small world phenomena is presented as a strong argument that modular structure may be captured through modelling, and more importantly that it exists and appears to be a strong feature of real world problems.

Walsh also comments that a method used by Hogg [55] to distinguish problems with modular characteristics does not correlate with the measure of the proximity ratio in his experiments. The method by Hogg considers the "approximate entropy" which utilises comparisons of sub-graph isomorphism for a given graph problem topology. This approach also relies on the structure being exposed at an atomic level, but is perhaps not quite so severe. Unfortunately sub-graph isomorphism is a difficult problem even for a single comparison [39] and yet comparisons with large graphs may be required to fully capture structure. For practical purposes this method may well be very difficult (for example see [24]). Consider again the example of a random 3-SAT problem mapped into a graph colouring problem: Structure on a very small level is evident because of the translation. The occurrence of sub-graph isomorphism is likely for small (and tractable) sub-graph comparisons because the expected variable distributions are uniform, but the results of this approximate measurement may not identify the overall randomness of the problem.

4.2 Parameterising reality

Sets of random satisfiability problems are defined by their parameterisations. We know variable distributions and problem sizes. If we want to know how a parameter affects the process of search then we can observe the effects by performing a series of experiments on sets of problems with different parameterisations. Real world problems on the other hand do not generally have simple parameterisations, and those that do have parameterisations which are based specifically on that type of problem, for example a pigeon-hole proof problem or a factorisation problem. What we would like is a basic parameterisation modelling a property that we expect to exist in some degree in real world problems in general. In this section we review and discuss approaches that attempt to model some kind of structure in order to emulate real world satisfiability problems.

There are perhaps two possible approaches to modelling real world structure in problems. The first is to extend the description of a real world problem classification in order to gain further generalisation, but hopefully retain the structural properties of that problem type. The second is to attempt to extend some general problem description, such as a random SAT problem, by introducing some structural constraints in the problem definition. This acts to capture some essence of structure. In this section we review previous work and introduce our own work. To do this we identify two categories: i) problem generators which are extrapolated from a given set of real world problems and ii) problem generators which are extrapolated from a purely random model.

4.2.1 Real World Generators

Our investigation focuses on identifying wide-ranging generic properties of real world problems. We are motivated by the desire for a better understanding of search and search problems in the real world. We are also motivated by the lack of variety in real world benchmarks available to the research community. One successful approach in alleviating the second problem is to find real world domains for which a number of problems can be generated. We provide a brief review of the techniques developed in this area.

A basic example for problem generation is the traditional pigeon hole problem, for which many instances can be generated for different sizes. Quasigroup problems are a real world domain and have a solid mathematical basis which allows easy and reasonably plentiful problem generation. Quasigroup completion problems encoded as constraint satisfaction were proposed by Gomes and Selman [47], where a partial definition of a quasigroup is provided, and must be completed to be solved. This method is akin to correctly filling in some blank entries to complete a Latin Square. Generating satisfiable problem instances within the quasigroup domain is further investigated in [1]. Other approaches include problem generation by encoding the parity problem [25] and the domain of cryptography [77].

A more general approach called *morphing* was devised by Gent *et al.* [44]. Their work categorises some approaches to "morphing", each of which defines a translation from given problem definitions to a new problem via the introduction of noise. More precisely the "noise", or randomisation, defines the degree of interspersion between two different problem definitions. (A simple case is where the second problem is purely random.) Each morph (or translation) approach is parameterised by some probability value p which is used during the transformation process. For example the rewiring of a ring lattice may be performed by combining the fraction $(1 - p)$ of the edges of the lattice and a fraction p of the edges from a random graph of the same size. The morphing technique can be a very powerful tool for generating a variety of similar problem instances. Given a real world problem (or even a small set), we identify the appropriate morph technique, and can generate a larger set of problems. Obviously the technique and amount of noise will effect the properties of the resulting set, but comparative empirical analysis of search costs leads to a better understanding of that particular problem domain (as performed in [44]), and provides a wider variety of benchmark problems. For implementers with a realistic outlook this is a far better alternative for search algorithm tuning than using random 3-SAT problems.

4.2.2 Structural Generators

The second approach is to define a model that incorporates some structural content in a problem and allows the problems to be randomly generated based upon that model. It is necessary that the structure is a scalable property of the model. We could easily introduce a simple structure by, for example, mapping some other random NP-complete problem to a satisfiability instance. In such cases the structure is fixed and will generally be constant and independent of the size of the original problem. We only capture the structure of the transformation process. The actual mapping of the elements of the original problem will still be mapped in the original random fashion, albeit on top of some interpretive structure that defines the mapping algorithm. For any random based model we need to ensure that the concept of structure is embedded in the model and is in some way proportional to the size of the problem, but is still proportional to reality. More importantly we need to justify that the induced structure corresponds to the kind of structure that will be encountered in real world domains.

In this chapter we have argued that modularity is a good generic structural property of many real world domains. We combine instances of random 3-SAT problems as modules or *clusters*. The model will be discussed in detail in Section 4.3. There are similar approaches which we briefly review here.

Two methods for random generation of problems with possible real world similarity were proposed by Rish and Dechter [92]. The first model is similar to our model in that its modular components are random 3-SAT problems, which they call subtheories. The main difference is that they are connected in a chain by joining neighbouring clusters with a single binary clause. The small world notion is not captured since the characteristic path length (or some meta-level equivalent notion) is high for long chains. That the induced structure corresponds to some real world domain is justified by the observation that the clique-chain structure occurs in temporal domains "that possess the Markov property (the future is independent of the past given the present)" [92]. We discuss their results concerning this model in more detail in Section 4.6. Their second model, the $(k, m) - tree$, generates a randomly conglomerated set of "cliques" for which a number of clauses is defined. Each clique has $(k + m)$ vertices where k is the size of the intersection between two adjacent cliques. While we can see the graph theory analogy for variable interaction, it is unclear to us exactly how the clauses for each problem are generated and what the properties of an individual component are. The correspondence to real world structure for this model is justified through the work in [34], where it is shown that circuit diagnosis benchmarks can be embedded in a tree of cliques. Both problem types proposed have bounded *induced width*, a parameter used to further model performance and complexity. The authors define induced width as "a graph parameter that describes the size of the largest clique created in the problem's interaction graph during inference". Here the interaction graph is a graph whose edges are defined by co-occurrence of variables within the clauses. Further relationships are captured with the resolvents – the "induced" information.

There seem to be other problem generators based around pattern construction and repetition for other search problem domains (e.g. graph colouring). Walsh [111] notes the work in [54, 50] for more realistic graph generation. Studies modelling real world networks (e.g. the World Wide Web) with graph toplogy, have produced accurate interpretations of the behavioural characteristics of real dynamic network scenarios [8]. See the introduction of [112] for further commentary on the approaches in realistic graph topology generation.

4.3 A Problem Model for Clustering

The observation of small world phenomena in search problems infers that at least some real world problems will have some modular structure. Our intuitive notions about the sorts of structure in problems reinforces this concept. In order to capture modularity in a reusable and parameterisable problem set we propose a problem model based on the random generation of fixed size modules or clusters. The model uses a set of individual random 3-SAT problems which are "connected" by a small set of extra clauses.

A given instance of this problem can be parameterised in the typical random 3-SAT model by the number of variables and the clause to variable ratio. We choose this as a strategy since the random 3-SAT problems are well understood. For a given cluster the level of hardness and probability of being satisfied is easily predicted and familiar. The overall problem is essentially a rearranged random 3-SAT problem, and this can be useful for comparisons to instances of the traditional one cluster random 3-SAT problem.

Formally we define the random clustered problem model as follows. A clustered problem instance has n variables, m clauses, c clusters and p percent links. We may use the clause to variable ratio r, a common parameterisation in random 3-SAT experiments to indicate the 'hardness' of each cluster. In this case the cluster, on its own, would have a ratio $r = (1 - p/100)m/n$. As p increases, the structure of the individual clusters decay, and eventually the problem will become a regular random 3-SAT problem. We wish to investigate the effects of structural changes such as this. Furthermore we want to define some level of difficulty within the individual clusters so that they appear to be a separate sub-problem to the search algorithm. We also want the link clauses to make the problem appear in some sense as a whole.

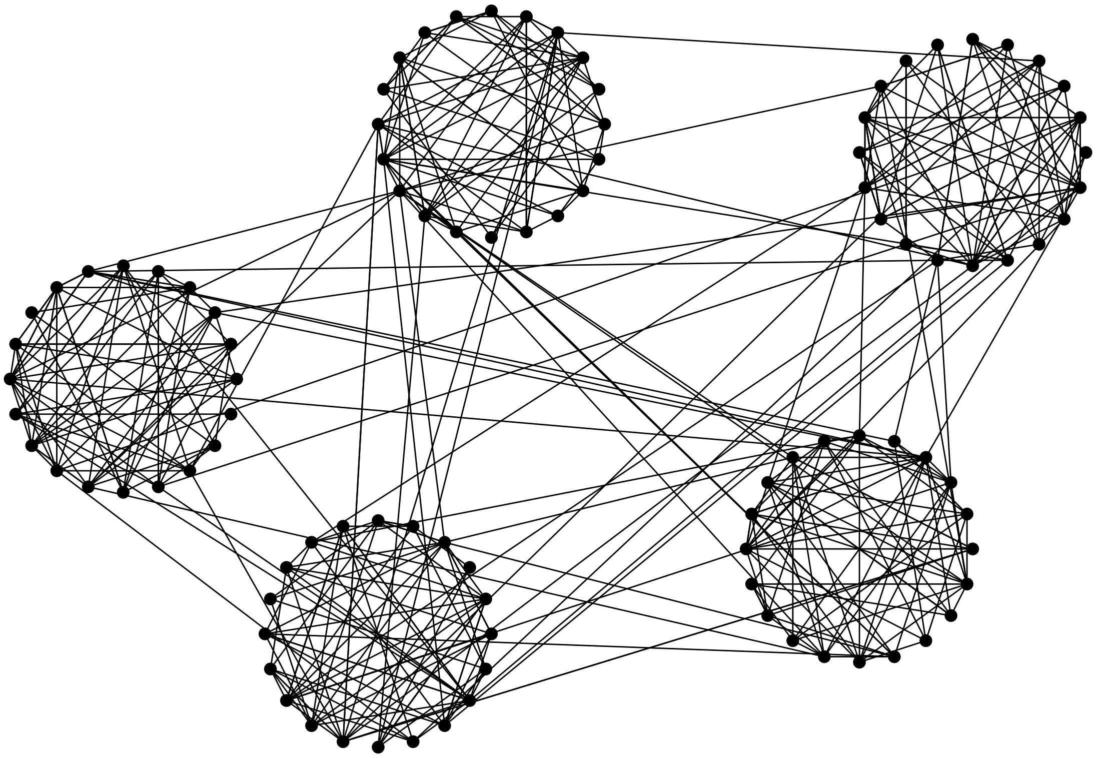


Figure 4.2: Co-occurrence graph for a random cluster problem with 5 clusters of 20 variables each. The clause to variable ratio is 4.25, but 5% of the clauses act as links. The density in a single cluster represents a local clause to variable ratio of 4.04.

The aim of this problem model is to capture structure through some arbitrary modularity. It does not necessarily have the quantifiable properties of a "small world" problem, since the size of each individual cluster can be varied. A random cluster problem will probably only have a small world characterisation when the cluster sizes are small since the measurement of the proximity ratio requires relationships at the atomic level. For small clusters, the clustering coefficient will be larger, and overall the characteristic path length should be shorter between most variables. On a meta-level, however, we emulate the small world situation by modelling the local interactions by clusters, and the global interactions by the links. This concept is somewhat more independent of the cluster size. Figure 4.2 shows the co-occurrence graph for a problem with five clusters of 20 variables. The depiction makes it obvious that the problem has some central components, but that the interdependencies, or links, make the problem a whole.

We shouldn't expect the problems to be particularly difficult if the cluster problems are not difficult. The problems should measure the ability of a search algorithm to "concentrate" on a particular problem, or at least identify which parts of the search space are relevant to the current search state. Consider why some large real world problems are not as difficult as a hard random 3-SAT problem. A good search algorithm will be able to identify, through some means, an appropriate ordering of the search so that each cluster is solved individually – a divide and conquer style approach. The model does not explicitly capture relationships between clusters that force particular solutions in neighbouring clusters through dependencies on key variables. We cannot see a way to include this property within the model without losing some of the more generic properties. For our model the situation must arise randomly. However the situation is more likely to arise if we extend the model proposed by Rish and Dechter [92], by using numerous binary relationships (clauses) rather than ternary relationships (clauses). These will more readily "force" assignments in a neighbouring cluster. Overall the proposed model can provide large sets of generic problems that contain modularity.

4.4 Experimenting with the Model

Experiments were performed to measure the changes in search cost for various parameterisations of the proposed random cluster model. The experimentation reveals that the theory behind the model can predict search algorithm behaviour. There were some interesting exceptions in the initial experiments which we were able to explain after seeing the results. This first set of experiments points out possible dangers in mixing search techniques. It also highlights the problems which can occur when using methods which can lead to an explosion of calculations. These results are discussed in Section 4.4.2. Subsequent experiments yielded a more predictable set of results and these are presented in Section 4.4.3.

The satisfiability search system `satz` [71] was chosen for performing preliminary experiments. It is an efficient and reliable implementation, and furthermore is very capable in solving large random 3-SAT problems. This allows testing on larger problem sizes and the use of large sample sets. It is posited that the use of a state of the art system is a reasonable approach in testing the proposed benchmark problems. Additionally, the use of large cluster sizes allows greater potential for "isomorphic richness" in individual clusters i.e. the number of isomorphism classes for a given size of sub-problem (under some sensible topological interpretation) is much greater. The author believes that the use of very small random 3-SAT clusters is not representative, a point which will be discussed below. All our experiments were carried out on a Sun Enterprise workstation with four 248MHz UltraSPARC II processors and 1Gb of memory. Each individual search problem was solved using only 1 processor.

A final round of experiments was performed to compare the relative abilities of a selection of state of the art satisfiability search systems. A relative performance ranking of these systems for a set of random 3-SAT problems was compared with a performance ranking for a set of random clustered problems. These results are shown later in Section 4.5.

4.4.1 Generating Clustered Instances

To generate the random clustered problems, a problem generator that takes the problem set parameters and produces the set of problems was implemented. Clauses which contain a duplicate variable are not included, and problems that are unconnected are not produced; i.e. if the set of links does not join each cluster through co-occurrence then the links are regenerated. For particular sets of parameters this may be difficult to achieve (e.g. when the percentage of clauses as links is very small) so users or implementers of this method should be wary of such situations. Similarly, avoiding duplicate variables in clauses is more difficult for very small cluster sizes.

4.4.2 Initial Experiments

Experimenting began with the clustered problems by generating a set of problems with a number of clusters in the range 1 to 9 (with a 1 cluster problem being equivalent to the standard random 3-SAT problem). Each problem had 200 variables and 840 clauses. Furthermore several problem sets were generated, each with a different percentage of links. Surprisingly, at around 5 clusters the problems became significantly harder to solve. For low values of percentage links, a number of the problems could not be solved within the 5 minute CPU time limit set for each problem. This contrasts severely with the CPU time taken to solve a single cluster problem (the regular random 3-SAT problem) of the same size. For a set of single cluster problems a median time of 0.56 seconds was measured, where 69 percent of the instances were satisfiable.

Figure 4.3 shows the results computed for the cluster problems using 5 percent of the clauses as links. The points that do not appear on the graph were all over the 300 second time limit. The drop in satisfiability is consistent with predictions, but the performances are not. As the percentage of links increases the level of difficulty drops. The most difficult problems are the ones with 5, 6 or 7 clusters, that is, they are in the centre of the range we investigated. These problems are significantly difficult when compared to the single cluster problem. Figure 4.4 shows the same experiments

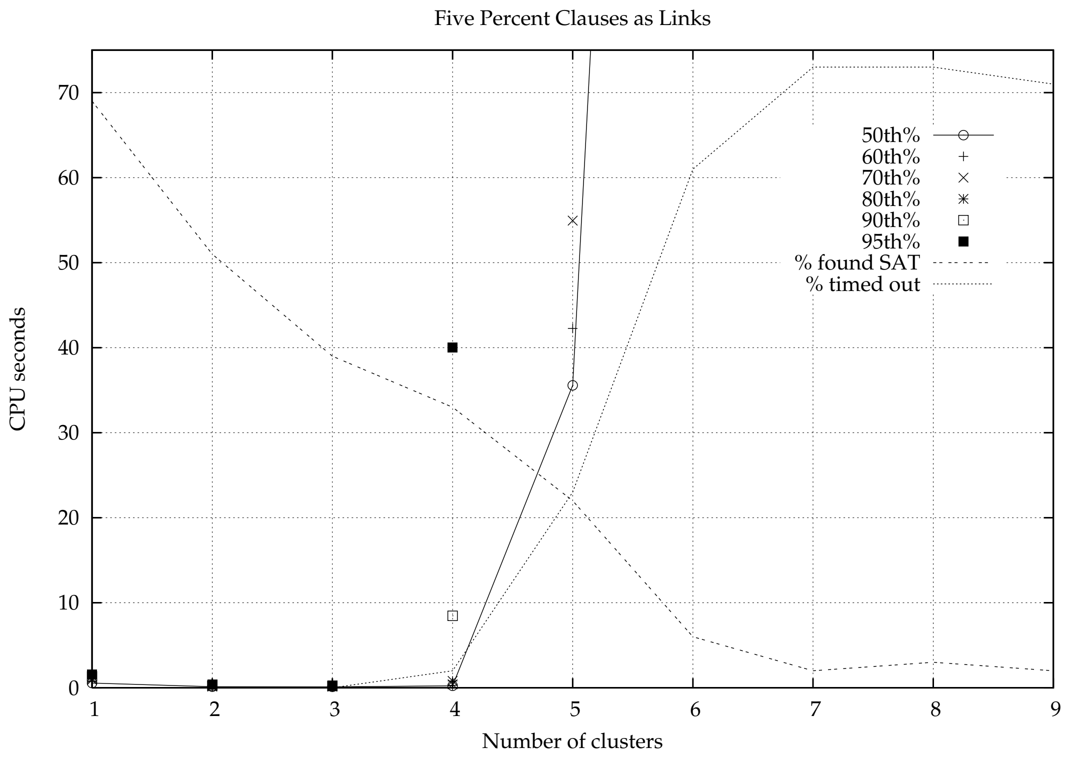


Figure 4.3: Time taken for clustered problems of 200 variables and 840 clauses. The number of clusters ranges from 1 to 9. The number of clauses that are used as links is 5 percent of the total number of clauses. 1000 problems were tested for each cluster size. The median values are shown by the solid curve, and various percentile points are plotted. The percentage of problems found satisfiable as well as the percentage of problems that timed out are plotted as dashed curves. Data points that occur above the visible area exceeded the CPU time limit of 5 minutes.

carried out using fifteen percent of the number of clauses as links. The difficulty curve is more apparent here. Viewing several different percentage links values reveals a trend of the performance curve to drop, and it will eventually completely subside as the entire problem consists of random links.

It appears that in the initial experiments the difficulty is a function of the percentage of links as well as the number of clusters. The source of the difficulty is not within the **DPLL** search part of `satz`, but in a preprocessing technique it uses. It is claimed by the authors of `satz` that this technique reduces the search cost by about 10 percent for hard random 3-SAT, and a variety of speed-ups is seen for benchmark problem classes [72]. The authors also note that the technique renders some benchmark problems unsolvable. The technique essentially performs a limited resolution process from the given clauses in the formula. It creates resolvent clauses where the

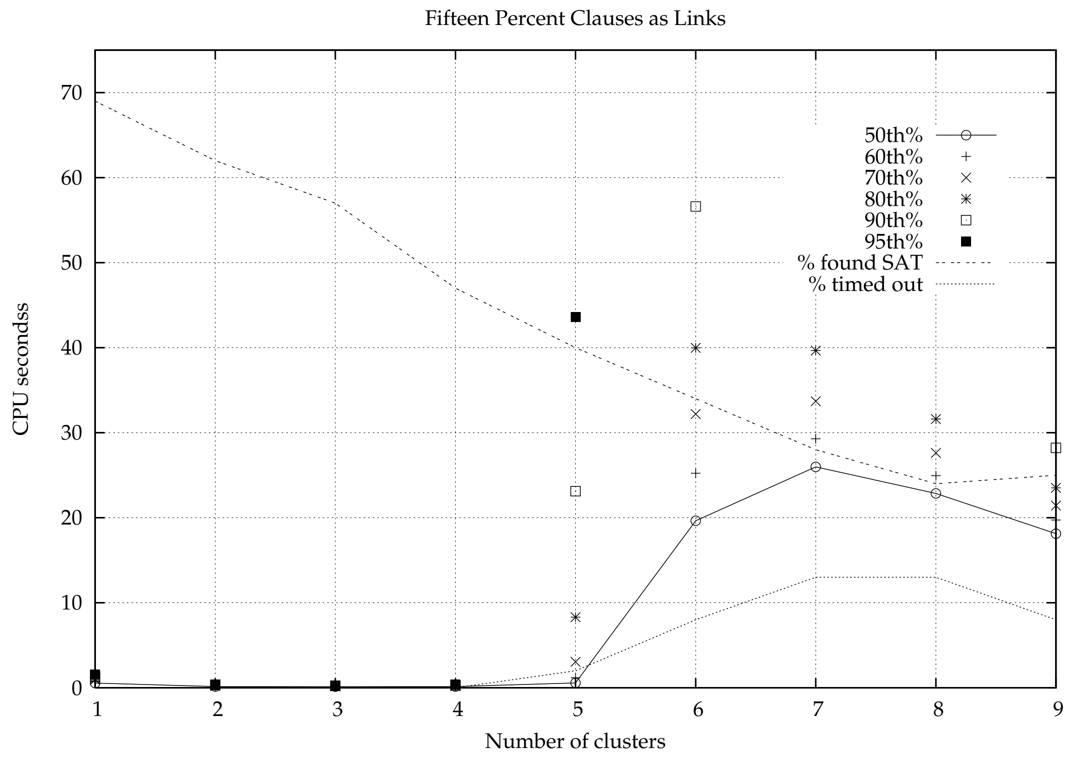


Figure 4.4: This graph is the same as for Figure 4.3, but the number of clauses that are used as links is 15 percent of the total number of clauses. With a further 10 percent of the total clauses acting as links many more problems are solvable within the CPU time limit, and in general they are all easier to solve.

size of the resolvent is no greater than the resolving clauses. Further stipulations are made about interactions of clauses of a particular size (various rules for binary and ternary interactions are given). Secondarily the process combines the look-ahead feature of the program with this notion of resolvents to refine the scoring of a particular choice. The fine details of this procedure are not important - the key issue is that an unbounded resolution process occurs before any **DPLL** style search is performed. The rules limiting resolvent sizes do not bound the process strongly enough to cope with the situations presented in some of the random cluster problems.

For the apparently difficult random cluster problems generated, `satz` created far too many resolvents for the preprocessing technique to cope with. While it is common for state of the art satisfiability solvers to be somewhat slower than basic implementations for small problems, it is surprising to see that a small number (e.g. 5 clusters) of these problems causes catastrophic behaviour when they have only slight relationships defined by the links clauses. The small world qualities of these cluster problems is likely to be higher merely due to their size, and it is the cliqueishness of each cluster that causes the breakdown of the preprocessing technique. The probability of co-occurrence of variables is high for small random problems, thus the potential number of small resolvents in each cluster will also be high. The effect of chaining several small random problems (i.e. the clusters), together with their resolvents, creates an intractable problem for this simplified version of the resolution process. It is highly likely that a fuller version of resolution could actually solve the individual cluster problems as in [92], which shall be discussed in further detail in section 4.6.

The initial experiments show a caveat for preprocessing or intermediate techniques that have the potential to create unnecessarily large amounts of computation. While it is clear that resolution inspired techniques can be useful, their integration must be carefully designed. The difficulty that the `satz` preprocessor experiences is caused by interactions at the atomic level of small clusters in the random cluster problem model. It is not unimaginable that similar kinds of structure may be found in real world problems. In order to alleviate these situations such approaches need to embed some intelligence which gauges the usefulness of the application of preprocessing so that unnecessary intractability is avoided.

4.4.3 Subsequent Experiments

Fortunately the preprocessing that `satz` performs can be disabled. The next set of experiments was performed on problems with 250 variables and 1061 clauses, using from 1 to 10 clusters, for a variety of percentage links values. For these experiments 1000 problems were tested for each combination of parameters used. Figure 4.5 shows the median search cost in CPU time for a variety of percentage links, against the number of clusters in the problems. As expected the cost decreases as the number of clusters increase since the individual sub-problems are easier to solve. For higher values of percentage links the problems are more like the single cluster, or basic random 3-SAT, and are harder to solve than the problems with well defined clustering. This may be because there is less information about where the search algorithm should concentrate as more links will direct the branching heuristics to a different sub-problem. We note that the median cost for a single cluster problem is 4.2 seconds, which is greater than any other values that appear in this figure.

The 95th percentile costs are shown in Figure 4.6 and show that for the most difficult problems the hardness is not extreme. For comparison the 95th percentile cost for the single cluster problem was 9.4 seconds. The relative level of difficulty between the median case and the hardest problems seems somewhat greater for the clustered problems than for the basic random 3-SAT model, but this difference may not be sig-

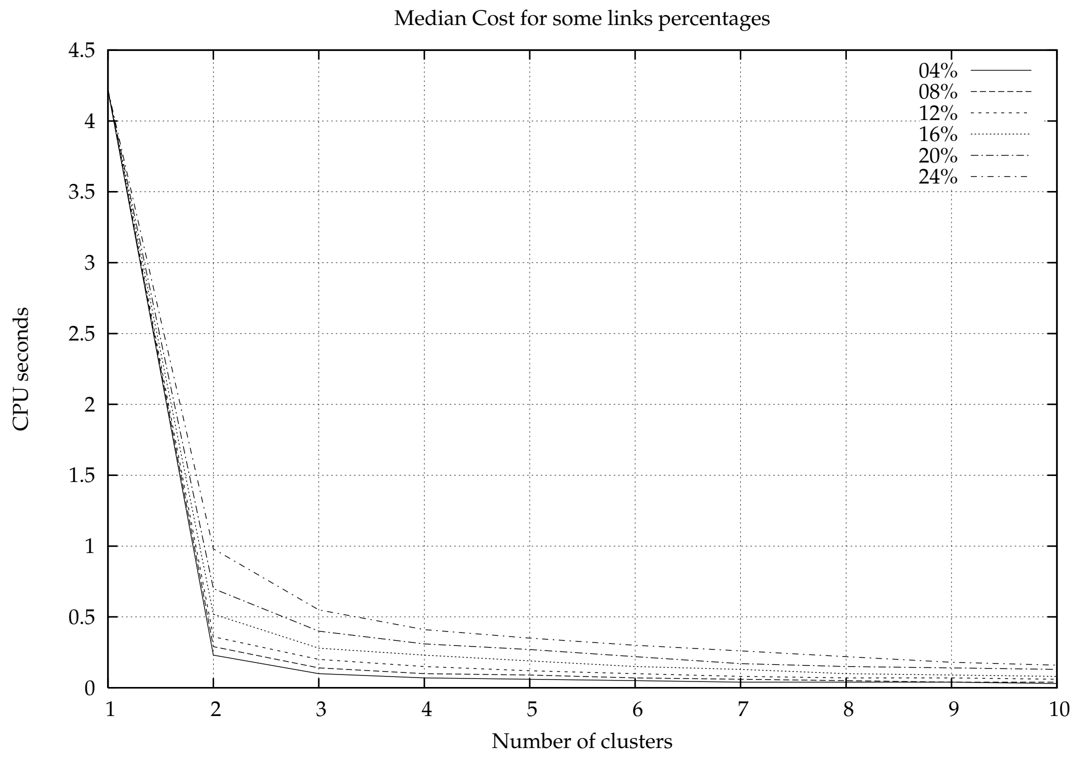


Figure 4.5: Median time taken for clustered problems of 250 variables and 1061 clauses (a global clause to variable ratio of about 4.24) for a variety of percentage links. The key indicates the percentage of the clauses that are used as links for particular curves. 1000 problems were used for each cluster size.

nificant. The trend in the costs as the percentage of links increases is the same for the median costs. The susceptibility to outliers for high percentiles is apparent, as the cost curves do not descend as smoothly as the median case.

Figure 4.7 shows the change in the number of problems that are satisfiable as the number of clusters increase for a selection of percentages of links. As predicted the percentage of satisfiable problems decreases with cluster size. This is due to the compounding of the individual probability of any cluster being satisfiable. Using this property a given problem set can be generated such that a desired distribution can be obtained. The interaction of the percentage of clauses that act as links lifts the probability of satisfiability toward the single cluster value. For larger values of percentage links this effect can contribute greatly to the final distribution, since the clause to variable ratio of the individual clusters is decreased and hence more likely to be satisfiable. In 1000 single cluster problems of the same size 53 percent were satisfiable.

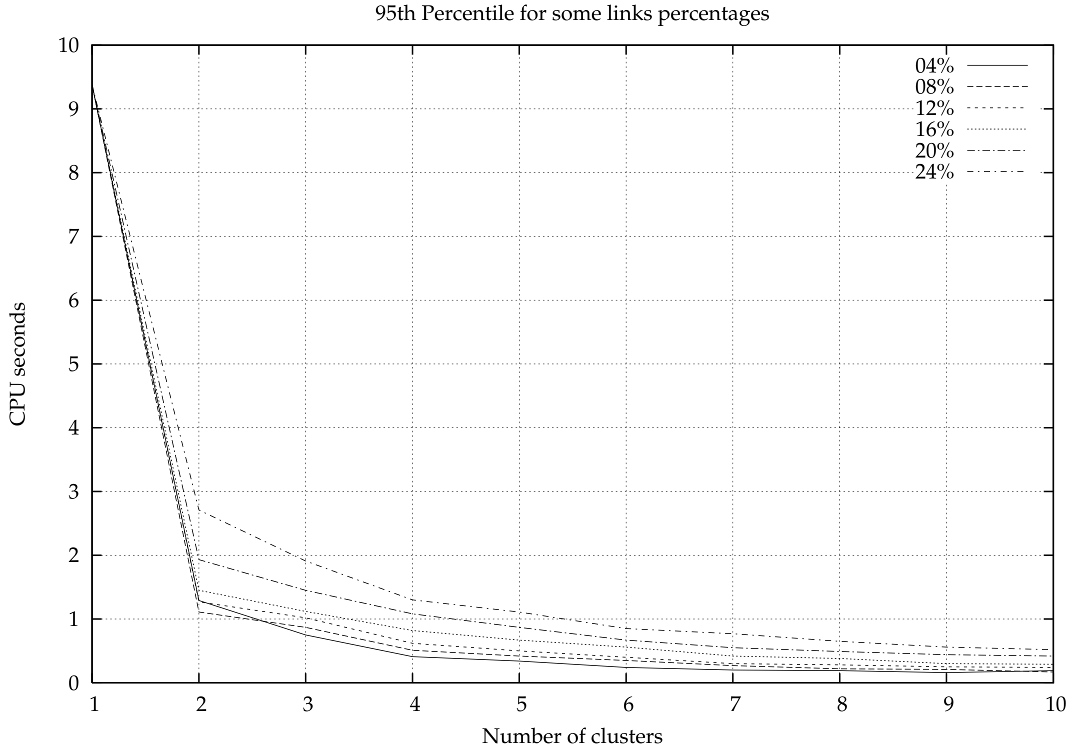


Figure 4.6: As for Figure 4.5 but showing the costs at the 95th percentile.

4.5 Measuring A System's "Concentration"

Earlier it was hypothesised that the random cluster model could be used to measure how well a satisfiability search system could "concentrate" on a problem. By this we mean how well it can search and solve a sub-problem before moving onto some other part of the problem. An ability to concentrate may be just that it makes assumption choices in an order which corresponds to the variables in a sub-problem first. This is what is happening in the case of `satz`. Other systems use a variety of different combined techniques. A variety of state of the art systems were performance tested on the random cluster model. In order to get an idea of how well they coped their ability was compared to the basic random 3-SAT problems by ranking the performance results for both problem types. Those that rank highly in the random cluster problems should have a good ability to "concentrate". By comparing the rankings to those for the random 3-SAT problems we should see the relative difference when modularity is introduced. The systems used were:

`satz` [71] The initial system used for our experimentation which relies solely on its ability to make the best choices in the search tree. This system is renowned for

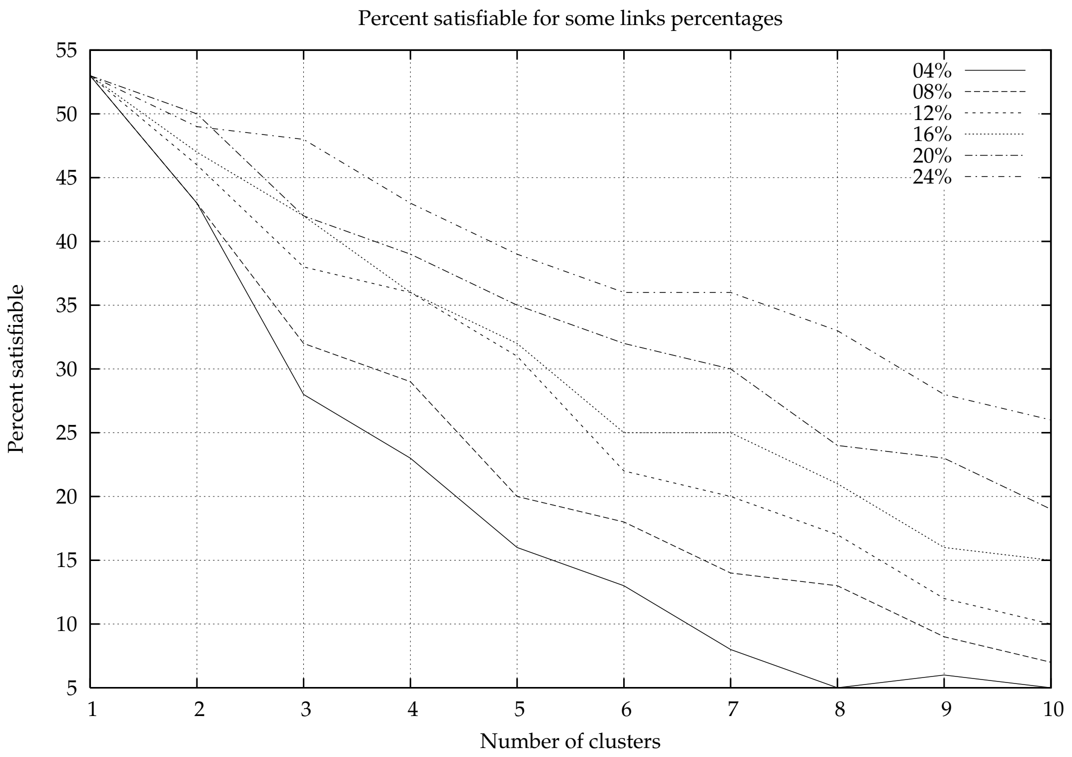


Figure 4.7: As for Figure 4.5 but showing the percentage problems that were satisfiable.

its performance on random 3-SAT problems, so the individual cluster problems should not be difficult for it.

`posit` [36] A somewhat earlier system which uses a combination of methods to select good variable choices and prune search space. Freeman's thesis demonstrates that it is successful in solving a wide variety of benchmark problems as well as random SAT problems [36].

`relsat` [11] An advanced system that utilises a version of dynamic backtracking and learning techniques. Its ability to reason about relevancy in search space should make it an ideal candidate for modular problems, and it is generally very successful on real world benchmark problems.

`OKSolver` [65] A system which uses a variety of pruning techniques and branch ordering heuristics which seem to be motivated by complexity analysis. It should give consistent but not best results.

`sato` [116] This system utilises an alternative data structure for search reasoning and has been very successful on quasi-group problems, having solved open problems in this domain.

System	Average	Median	Maximum	Minimum	Std Dev.
`posit`	0.345	0.34	1.29	0.03	0.198
`satz`	0.355	0.36	0.99	0.04	0.195
`OKSolver`	1.123	1.13	4.54	0.05	0.819
`relsat`	1.792	1.69	7.72	0.07	1.360
`sato`	3.891	3.02	28.85	0.00	3.745

a) Ranked Results for One Cluster Problems

System	Average	Median	Maximum	Minimum	Std Dev.
`satz`	10.806	5.68	151.57	0.19	15.140
`relsat`	13.051	8.56	137.30	0.29	14.452
`OKSolver`	28.754	19.58	277.38	0.18	31.722
`sato`	167.571	58.30	3852.26	0.08	333.333
`posit`	151.737	65.69	3392.30	0.19	266.545

b) Ranked Results for Nine Clusters Problems

Table 4.2: These tables show an ordering of run-time statistics for a selection of state-of-the-art systems. Each table is ordered by increasing median value. Note that the ordering of the other statistics is similar but does not always agree. Table a) shows results for 1000 single cluster, or basic random 3-SAT, problems of 200 variables with a clause to variable ratio of 4.25. Table b) shows results for another 1000 problems of 450 variables also with a clause to variable ratio of 4.25. These problems contain 9 clusters where 20 percent of the clauses act as links. The difference in the ordering of the two tables indicates how the different methods cope with the random cluster model.

The results are ranked in order of smallest median score. Both sets of problems contained 1000 samples. The random 3-SAT problems had 200 variables and a clause to variable ratio of 4.25, and the random cluster problems had 450 variables and a clause to variable ratio of 4.25 with 20 percent of the clauses acting as links. We stress that the two sets of results are produced to gain a ranking only, as we cannot directly compare the performances on these potentially very different problem sets. Table 4.2 a) shows statistical data gathered from the random 3-SAT experiment. Both `satz` and `posit` outperform the other systems, and the other median scores are several times greater than the two top systems. Both `satz` and `posit` are really on home ground here since both were extensively tested and tuned for this kind of problem. The results shown in Table 4.2 b) show quite a different trend. As predicted `relsat` copes far better with the random cluster problems than the other systems, except for `satz` with a fairly close first. The bottom three systems again have median costs several times greater than the best, with `sato` and `posit` being slower by more than an order of magnitude. `OKSolver` performs reasonably well on both problem sets. `sato` does not gain a high ranking on either problem set, but this may be explained by the fact that it was primarily designed to cope with a different kind of problem (quasigroups). The most interesting relative rankings are for `posit` and `relsat`. An intelligent backtracking system is able to cope with sub-problems as it can reason about which deductions are relevant to its current position in the search space. `posit` however does not seem to be able to take advantage of the existence of small separate problems even though it incorporates an advanced choice heuristic.

The rankings of a selection of different systems show that the search method is important for solving this kind of modular structure. Since the individual problems are random 3-SAT problems we would normally expect that a system that solves these well will cope with random clusters. This was not the case with `posit` whose search technique was not powerful enough to detect the sub-problems as well as the best ranked systems. `relsat`, known for an ability to deal with real world domain problems fared much better in the rankings for the random cluster model problems than with the random 3-SAT problem. This is evidence that the modular structure is apparent to its search method, and furthermore suggests that the random cluster model is a far better model of the real world domain than the random 3-SAT model.

4.6 Related Work

We previously noted the work of Rish and Dechter [92] who independently created a random cluster problem model they call random 3-CNF chains. In a series of experiments that appears to show Directional Resolution (DR) is more capable than DPLL algorithms, they find anomalously hard problems for their implementation of DPLL. The difference in search times is of orders of magnitude. They do use an advanced search heuristic but this is obviously not powerful enough to match the performance of DR. We found no anomalous hardness spikes in any of our experiments using DPLL style algorithms, bar the resolution based preprocessing problems in the initial experiments. The difference in observations is most likely to be due to the fact that our clusters are not chained – the path length from any cluster to another is very short. Rish and Dechter pose that when a sub-problem at the end of the chain is unsatisfiable then the entire problem must be re-solved. This would infer that their algorithm chose a particularly unfortunate ordering, and it seems that such a situation would be reasonably rare. In our experiments we sampled tens of thousands of large problems with different parameterisations. It is conceivable that the same unfortunate ordering could occur when the last cluster to be solved was unsatisfiable. However we did not notice any problems that were as extremely hard as those observed in [92]. By adding back-jumping to their DPLL implementation Rish and Dechter are able to avoid most difficult problems, but they do suggest a possible phase transition phenomenon in chain problems. This seems unlikely in view of the other experiments.

Although their model is quite different to ours in terms of connectivity between sub-problems, it will still measure an algorithm's ability to concentrate. It will be generally more efficient to solve sub-problems one at a time. We suspect that a combination of the heuristic used and the scale of the model used by Rish and Dechter is also responsible for large search costs. In fact, for the number of variables in the problems, some of the worst search costs suggest that the supposedly advanced choice ordering has failed badly. Since this ordering is based on `tableaux` [26], we suspect that the empirically derived choice mechanism is over-fitted to random 3-SAT problems. In fact it is possible that the branching scheme prefers to choose a variable, perhaps in a binary clause, somewhere else, rather than concentrating on solving the current sub-problem. We note that the `tableaux` system follows much of Freeman's work on branch heuristics for `posit`. Our comparison experiments in section 4.5 show that although `posit` is effective in solving random 3-SAT problems, this ability did not translate to random cluster problems. We also believe that the size of the sub-problems Rish and Dechter use is a significant factor. For most of the experiments with their chain model, each chain has only 4 or 5 variables. Hardness is observed as a function of number of clauses. For example, one figure shows their DPLL algorithm having difficulty with clusters that appear to have 5 variables and about 14 clauses. The isomorphic richness in these chain sub-problems is very poor, and the probability of co-occurrence for any pair of variables is extremely high. A resolution based approach is bound for success due to the high probability of co-occurrence for variables, i.e. there will be many small and therefore more useful resolvents. Larger chain components will eliminate this advantage.

The 3-CNF chain model is a step away from completely random problem generators. It is argued that since they have a relationship to real world problems and induced width parameterisation can be performed, they are a good benchmark candidate. They may indeed provide a useful testing environment. However, graph relationship calculations based on co-occurrences for satisfiability problems are not quite as strong as those for graph colouring, and in this situation the induced width concept may be misleading if it is associated with search difficulty. We gave further detail when commenting on similar difficulties for small world analysis for satisfiability in Section 4.1.3. That a small chain-like formation may occur in reality is definitely possible, but we don't believe that it represents generic real world structure, nor do we think that a good implementation of DPLL will find arbitrary parameterisations of this model difficult. We do think that it is possible that there will be differences in performance between DPLL and DR for particular parameterisations, as observed by Rish and Dechter. However, large performance discrepancies include possible cases where the chains are not so susceptible to resolution approaches and the DPLL algorithm will outperform DR. The work on the 3-CNF chain model highlights the advantages of the use of a good resolution technique where clusters are tightly related (have a large clustering coefficient). We suspect that their techniques for DR would be excellent for these situations. A preprocessing or intermediate processing technique for DPLL based on DR would be far more useful than the `satz` preprocessor as the level of reasoning is far more advanced.

4.7 Conclusions

In the course of investigating the concept of structure in real world problems modularity has been identified as an important candidate concept based on intuitive reasoning and the work of Walsh [111]. We have discussed the need for a flexible problem model in order to experiment with search algorithms and problems that have a structural nature. The proposed problem model based on clusters of random 3-SAT problems gives a simple framework to generate modular problems that can be manipulated via the parameter set that defines them. This problem model generally behaves as the theory predicts when used in experimentation. However, our experimentation revealed possible problems with search methods that integrate particular resolution style techniques. This is because the nature of small connected sub-problems can yield an exponential explosion in the number of resolvents. A resolution based algorithm, even just as a preprocessor, must be advanced enough to avoid these situations. A simple bound on calculation, or intelligent selection of resolvents when they are too numerous, may suffice for the preprocessor approach. The success of Rish and Dechter's DR algorithm indicates that more advanced implementations of resolution algorithms to perform the search are far less likely to succumb to the problems we observed. Our experiments reveal that a reasonable (DPLL) search algorithm will be able to solve the random cluster problems relatively easily. Our observations suggest further explanations for the performance results observed in [92] for a related problem model and highlight the fact that choice heuristics derived from 3-SAT experimentation may not translate to other problem domains.

A reasonable search algorithm is one which is able to "concentrate" reasonably well on a problem. It will not become side-tracked into the search space of another sub-problem while it is trying to solve another. We demonstrated the differences between some state of the art systems in coping with this situation as created by the random cluster model. These results indicate that the proposed model has much better real world similarity than basic random models. It is apparent that several real world problem domains will have some modularity, and that their relative difficulty can be explained not only by the likelihood that they contain many intricately constraining clauses, but also that the difficulty is likely to be only as hard as the sub-problems they contain. Recent work published on the success of using detection of modularity within reasoning systems [2] reinforces these arguments.

Our model has the advantage of being reasonably predictable. It does not guarantee to capture "bottleneck" situations. We believe this is likely to occur in real world scenarios where a problem is constrained by a few critical variables. An extension of the approach in [92] may be more likely to capture this notion. We think that an extension of this model where clusters are linked through variable equivalencies (rather than the simpler co-occurrence) has good potential for capturing critical variable scenarios. The model could also be further extended in clause sizes and variability of cluster and clause size. We believe that this may make the problems significantly more interesting as they will less resemble the basic random 3-SAT model which may have had over-fitting effects on many systems.

Chapter 5

Conclusions

This chapter summarises the conclusions from investigations performed in this work. This dissertation has explored some fundamental issues concerning satisfiability and search. The approach taken was to investigate practical issues via the construction of a solid theoretical basis. While the research focussed on the satisfiability problem, many of the results and insights are applicable in the wider field of search and automated reasoning. The work questioned how effective search mechanisms actually work and whether the practical problem domains they are applied to can help us understand this. Two facets of the backtracking search methods were analysed: The theoretical foundations of the forward moving mechanism of choice heuristics, and the logical systems involved for the backward moving mechanisms in intelligent backtracking. The third issue addressed was the pursuit of a realistic problem model that could be beneficial for testing satisfiability search methods.

Much research has been performed on the analysis of branching strategies or forward construction in search. In satisfiability the prevalent methods were largely inspired by empirical results. In Chapter 2 it was questioned whether there was a reasonable theoretical explanation for the construction of a weighting scheme and priority function used in a branching scheme. The work derives such a scheme by considering the potential search space created by a given choice. The weights calculated show a significant similarity to those that have been empirically derived in other work. The theoretical branching scheme was shown to be competitive with other popular methods in performance experiments. When the proposed scheme was empirically tuned the performance approached the other schemes. The theory behind the branching scheme also incorporates an explanation of the success of look-ahead routines. The relationships of the approach to the Kappa search space model was discussed, and this demonstrated the flexibility of the approach for other search domains, as well as highlighting potential problems with the use of other Kappa based heuristics for choice.

Intelligent backtracking systems are designed to reduce redundancy in search. Certain techniques accomplish this by controlling the way the search space, or proof, is constructed. In Chapter 3 this was recognised as the equivalent aim of reasoning relevantly, as done in the class of non-classical relevant logics. It was demonstrated that a logical system, based on a fragment of the non-classical relevant logic, yielded an intelligent backtracking mechanism. A formal approach defining rules for a labelled classical relevant reasoning system intended for backtracking was used to describe an efficient algorithm for the method of conflict directed backjumping. Inspired by the technique of dynamic backtracking, a reasoning system was defined that emulated backjumping without erasing intermediate work. However, the traditional approach of proof free construction was retained, allowing the integration of other well known techniques. It was shown that the proof construction could be manipulated allowing proof rearrangement. A termination strategy was outlined and this defined some limitations on the rearrangement or backward choice but guaranteed a complete method. Experimental analysis revealed that a simple relevant search algorithm performed significantly better than other backtracking methods. While freedom in search movement has been inspired by results in randomised algorithms it was suggested that the results in Chapter 2 could provide reasons for making a particular backtrack point selection. While the proposed algorithm performs traditional search tree exploration, the logical foundations yield simplicity regardless of its capabilities.

In Chapter 4 the question of whether randomly generated problems could emulate characteristics observed in real world instances was investigated. The concept of modularity was introduced and justified, inspired by previous work. A random 3SAT problem model with the modular properties seen in real world scenarios was designed. It was then questioned whether such a model was useful for experimentation and for the emulation of possible real world scenarios. Experimentation revealed that certain parameterisations of the model were unexpectedly difficult for some resolution based techniques though, in general, performance was as expected. The anomalous behaviour was explained and this also suggested an explanation for similar phenomena observed in other work. A performance comparison based on characterisations on satisfiability solvers indicated that the model appeared to have some real world qualities. The malleability and characteristics of the problem model appear to be useful for investigating the interactions between satisfiability solving methods and the problems they solve. At the time of writing, a collection of instances was included in the SAT2002 workshop solver competition.

This work has focussed on developing theoretical foundations for aspects in satisfiability and search with the aim to improve understanding of the way in which problems are solved and investigate aspects that would be of practical use to satisfiability and related search domains. This body of work shows these aims have been met. There remain many possibilities for the extension of this work, some of which are outlined in the central chapters.

5.1 Limitations and Further Work

This section summarises those areas which we think further work would provide an interesting avenue of investigation. It also outlines some of the limitations perceived by the author of the work presented in this thesis. It is presented on a "by chapter" basis.

5.1.1 Chapter 2 – A Theory of Choice

A choice mechanism derived from some generic problem space or search space model will be limited by how accurate that model *can* (in the limit) actually be. The process of refining a largely empirically tuned system would hopefully identify the limits of the model, as well as indicating how the model could be improved.

5.1.1.1 Limitations

The choice framework and mechanism presented yields a useful and powerful but non-specific method of identifying how to make forward choices. A resulting weakness in terms of effectiveness is that this framework is quite broad i.e. it does not attempt to capture a particular problem domain. This exposes a reliance on the assumption of an expected distribution from the underlying search space model being based on "an ensemble of problems". For a given specific problem (or even a problem class) there may be some extra knowledge about the structure which infers information about making a good choice. This kind of limitation could only be lifted with the use of some adjunct technique. The model itself may be dominated by the assumption of a worst case scenario – this is exemplified by the priority function modelling a maximum potential value for search space down both branches. The priority function itself is expensive to compute, and it turns out that using an approximation of the function is more beneficial. Further investigation into dynamic search space modelling is required in order to address these limitations.

5.1.1.2 Further Work

While there may be ways to extend or improve the model of search space given, the actual utilisation and application of the choice mechanism seems an obvious candidate for improvement. Cost/benefit investigations into the use of a lookahead routine to improve prediction would be beneficial. A promising way to control the use of lookahead, as well as the "tuning of parameters", would be to create a dynamic approach that can adapt or learn.

The model could incorporate dynamic evaluations for clausal information, and in particular the "value" of learned information recorded as clauses. What value, in terms of search space, does a learned clause have? The new clause is the product of inferences using existing clauses and thus the value is interdependent. More importantly however a learned clause represents the closing of a fixed amount of previously traversed search space, i.e. it represents an amount of work actually done and also the potential savings in future steps. This concept could be highly useful, especially when the method is combined with an intelligent backtracking scheme as discussed in Chapter 3.

5.1.2 Chapter 3 – Relevant Backtracking

The relevant backtracking framework is a novel combination of more natural approaches to proof in philosophical logic and more traditional techniques for machine reasoning. As an alternative method for automated classical proof construction it is still fairly unsophisticated in the sense that it only uses a fragment of the available logic and like many other mechanised proof methods it lacks an overall strategy. However, it is this simplicity that has aided its own construction and presents a panoply of prospects.

5.1.2.1 Limitations

As with any intelligent backtracking process there is an "at node" computation overhead. This is relatively expensive even when compared with other linear space intelligent backtracking algorithms. In certain domains the approach will not be useful as the cost will outweigh the gains, or there will be a more efficient specific technique for that domain.

From a theoretical viewpoint it is still unclear how much of classical logic can be effectively enveloped in the "relevant framework". The current position presents difficulties for general proof construction for arbitrarily formed formula (see Section 3.1.2. Extensions to the approach for classical or non-classical logics may have to consider further layers of complexity in order to account for the behaviour of relevance under different operators.

5.1.2.2 Further Work

There are many possible paths of further investigation, some of which are detailed in Section 3.2.8. From a practical point of view the optimisation of procedures and the use of refined data structures would yield great performance improvements. Using a lookup table to calculate dependencies is analogous to a semantic interpretation and similar to the method Dynamic Backtracking employs, and would reduce at node processing of dependencies. Relevant backtracking style search would surely be enhanced with the addition of further search techniques. The use of a forward and backward choice mechanism that cooperated in some strategy to complete a proof or prune search space seems a promising path of investigation.

The theoretical aspects of maintaining relevancy with a syntactic definition of the logic present many opportunities for further investigation. One critical question is what kinds of operations and strategies would make proof and satisfiability search more effective. Possible solutions for relevant versions of these approaches would then have to be constructed. Such investigations are not limited to classical logic – the application of relevant style backtracking techniques for non-classical logics yields promising investigations.

5.1.3 Chapter 4 – Structure in Problems

The aim of problem generation is to provide a wider variety of benchmarks for testing and comparison. Realistic modelling attempts to capture some aspect of those problems derived from real world situations. In order to do this some precise knowledge of "real world" structure must be known. This can range from a very specific problem class structure to something more generic like modularity.

5.1.3.1 Limitations

It is quite clear that the proposed model cannot claim to actually be real. It represents an aspect of real problems based on theoretical and experimental observations. In many ways it resembles a regular random 3SAT problem, but it incorporates structure with modularity to provide a different kind of problem. Although additional extensions could eliminate the random 3SAT like appearance, the question remains as to what kind of "structure" or problem model would be chosen for this to provide an interesting test case or benchmark.

5.1.3.2 Further Work

Empirical evaluation of extended models would yield an understanding of how hard the extensions would be to solve, and this may lead to further useful benchmarks. The challenging problem in generating pseudo real problems is to identify structural characteristics which are common in real world problem domains so that they can be used as a template for generation. The parallel investigation to this asks what makes real world problems harder or easier to solve. In essence we look for commonalities in real problem domains that can be modelled but also exploited by a search algorithm.

Bibliography

1. D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 2000.
2. Eyal Amir and Sheila McIlraith. Partition-based logical reasoning. In *7th International Conference on Principles of Knowledge Representations and Reasoning*, 2000.
3. A.E. Anderson, N.D. Belnap, and J.M. Dunn. *Entailment: the Logic of Relevance and Necessity Vol 2*. Princeton University Press, 1992.
4. Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979. See also erratum [5].
5. Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. Erratum: "A linear-time algorithm for testing the truth of certain quantified Boolean formulas". *Information Processing Letters*, 14(4):195–195, June 1982. See [4].
6. Andrew Baker. The hazards of fancy backtracking. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 288–293, 1994.
7. Andrew Baker. *Intelligent Backtracking On Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, University of Oregon, 1995. CIS-TR-95-08.
8. A. Barabási and R. Albert. Emergance of scaling in random networks. *Science*, 286:509–512, October 1999.
9. Roberto J. Bayardo, Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 298–304, 1996.
10. Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, volume 1118 of *LNCS*, pages 46–60. Springer, 1996.

11. Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, 1997.

12. P. Beame, R.M. Karp, T. Pitassi, and M.E. Saks. On the complexity of unsatisfiability proofs for random k-CNF formulas. In *ACM Symposium on Theory of Computing*, pages 561–571, 1998.

13. P. Beame and T. Pitassi. Propostional proof complexity: Past, present, and future. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 65, 1998.

14. A. W. Bollen. Relevant logic programming. *Journal of Automated Reasoning*, 7(4):563–585, 1991.

15. M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, February 1981.

16. M. Buro and H. Kleine Büning. Report on a SAT competition. Technical Report FB17, Mathematik/Informatik Universitat Paderborn, November 1992.

17. V. Chandru and J. N. Hooker. Extended Horn sets in propositional logic. *Journal of the ACM*, 38(1):205–221, January 1991.

18. Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–340, 1991.

19. Xinguang Chen and Peter van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.

20. Alonzo Church. The weak theory of implication, 1951. In A. Menne, A. Wilhelmy, and H. Angsil, editors, *Kontroliertes Denken: Untersuchungen zum Logikkalk ul und zur Logik der Einzelwissenschaften*, pages 22–37. Kommissions-Verlag Karl Alber, Munich, 1951.

21. S.A. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 1971.

22. S.A. Cook and D.G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 35. American Mathematical Society, 1997.

23. S.A. Cook and R. Reckkhow. On the lengths of proofs in the propositional calculus. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, pages 135–148, 1974.

24. D.G. Corneil and D.G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM Journal of Computing*, 9(2):281–297, 1980.

25. J. Crawford and M. Kearns. Dimacs parity problem benchmark, 1993. See `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/crawford/README` and [60].

26. J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81:31–57, 1996.

27. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

28. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

29. Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

30. W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–84, 1984.

31. O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: the Second DIMACS Implementation Challenge*. American Mathematical Society, 1993. Published 1996.

32. O. Dubois and Y. Boufkhad. A general upper bound for the satisfiability threshold of random r-sat formulae. *Journal of Algorithms*, 24(2):395–420, 1997.

33. J.M. Dunn. Relevance Logic and Entailment. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic Volume 3: Alternatives To Classical Logic*, pages 117–224. D. Reidel Pub., 1986.

34. Y. El Fattah and R. Dechter. An evaluation of structural parameters for probabilistic reasoning: results on benchmark circuits. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995.

35. S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

36. Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

37. Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

38. M. R. Garey and D. S. Johnson. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976.

39. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

40. John Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, page 457, 1977.

41. John Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, 1979.

42. Ian Gent and Toby Wash. The search for satisfaction. Technical report, Dept. of Computer Science, University of Strathclyde, 1999.

43. Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The constrainedness of search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.

44. I.P Gent, H.H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, 1999.

45. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

46. Matthew L. Ginsberg and David A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 226–237, 1994.

47. Carla Gomes and Bart Selman. Problem structure in presence of perturbations. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 221–226, 1997.

48. Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 431–437, 1998.

49. Mike Gordon. *HolSatLib*. Computer Laboratory, University of Cambridge, 1.0b edition, 2001.

50. S. Grant and B.M. Smith. Where the exceptionally hard problems are. In *Proceedings of the CP-95 workshop on Really Hard Problems*, 1995.

51. Jan Friso Groote and Joost P. Warners. The propositional formula checker heeerhugo. *Journal of Automated Reasoning*, 24(1-2):101–125, 2000.

52. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.

53. John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, 1996. Springer Verlag.

54. Tadd Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81, 1996.

55. Tadd Hogg. Which search problems are random? In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 438–443, 1998.

56. J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

57. Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the year 2000*, pages 283–292. Kluwer Academic, 2000. `http://www.satlib.org`.

58. R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

59. Allen Van Gelder John V. Franco. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Applied Mathematics*, pages 177–214, 2003. Obtained preprint from 1998.

60. David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26. American Mathematical Society, 1993.

61. Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384. Morgan Kaufmann, San Francisco, California, 1996.

62. Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-96)*, 1992.

63. Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201, 1996.

64. Lefteris M. Kirousis, Evangelos Kranakis, and Danny Krizanc. Approximating the unsatisfiability threshold of random formulas. *Random Structures and Algorithms*, 1998.

65. Oliver Kullman. Heuristics for sat algorithms: Searching for some foundations. Technical report, Dept. Computer Science, University of Toronto, 1999.

66. Oliver Kullman. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223:1–72, 1999.

67. Tracy Larrabee and Yumi Tsuji. Evidence for a satisfiability threshold for random 3CNF formulas. Technical Report UCSC-CRL-92-42, University of California, Santa Cruz, Jack Baskin School of Engineering, November 1992.

68. Daniel Le Berre. satlive. `http://www.satlive.org`, 2000.

69. E.J. Lemmon. *Beginning Logic*. Chapman & Hall, 1991.

70. Chu-Min Li. Integrating Equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 2000.

71. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, 1997.

72. Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference on Principles and Practice of*

Constraint Programming (CP-97), volume 1330 of *LNCS*, pages 341–355. Springer-Verlag, 1997.

73. A.K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. John Wiley & Sons, 1992. Volume 1, second edition.

74. Joao P. Marques-Silva and Karem A. Sakallah. Boolean Satisfiability in Electronic Design Automation. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, 2000.

75. J.P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.

76. J.P. Marques-Silva and K.A. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 1996.

77. Fabio Massacci. Using walk-sat and rel-sat for cryptographic key search. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 290–295, 1999.

78. David A. McAllester. Partial order backtracking, 1993.

79. M.A. McRobbie, A.E. Anderson, N.D. Belnap, and J.M. Dunn. *Relevant analytic tableaux*, chapter X:60. In [3], 1992.

80. Michael A. McRobbie and Nuel D. Belnap. Relevant analytic tableaux. *Studia Logica*, 38:187–200, 1979.

81. David Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.

82. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

83. B.A. Murphy and R.P. Brent. On quadratic polynomials for the number field sieve. *Australian Computer Science Communications*, 20(3):199–213, 1998.

84. G. Necula and P. Lee. Proof carrying code. Technical Report CMU-CS-96-165, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, September 1996.

85. Nicola Olivetti. *Algorithmic Proof Theory for non-classical and modal logics*. PhD thesis, Dipartimento di Informatica, Università di Torino, 1995.

86. V.R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

87. Steven Prestwich. A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In *The Sixth International Conference on Principles and Practice of Constraint Programming*, volume 1894, pages 337–352. Springer-Verlag, 2000.

88. D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: the Second DIMACS Implementation Challenge*. American Mathematical Society, 1993. Published 1996.

89. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

90. E. Thomas Richards. *Non-systematic search and No-good learning*. PhD thesis, University of London, IC Parc, 1998.

91. E. Thomas Richards and Barry Richards. Nonsystematic search and no-good learning. *Journal of Automated Reasoning*, 24(4):483–533, 2000.

92. Irina Rish and Rina Dechter. Resolution versus Search: Two strategies for SAT. *Journal of Automated Reasoning*, 25(1-2):225–275, 2000.

93. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

94. R. Rodosek. A new approach on solving 3-satisfiability. In *Proceedings of the 3rd International Conference on Artificial Intelligence and Symbolic Mathematical Computation*, volume 1138 of *LNCS*, pages 197–212, 1996.

95. Hajime Sawamura and Daisaku Asanuma. Mechanizing relevant logics with hol. In Jim Grundy and Malcolm C. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 443–460. Springer, 1998.

96. Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226, 1978.

97. John S. Schlipf, Fred S. Annextein, John V. Franco, and R.P. Swaminathan. On finding solutions for extended Horn formulas. *Information Processing Letters*, 54:133–137, 1995.

98. Bart Selman, Henry Kautz, and David McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1997.

99. Bart Selman, Hector J. Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, 1992.

100. Laurent Simon and Philippe Chatalic. Satex: a web-based framework for sat experimentation. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, 2001. `http://www.lri.fr/~simon/satex/satex.php3`.

101. John Slaney. Is there a constrainedness knife-edge? In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, 2000.

102. John Slaney. Unpublished work. Results provided by author, 2000.

103. R. Smullyan. *First order Logic*. Springer-Verlag, 1968.

104. R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

105. G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Technical report, European Patent N 0403 454 (1995), US Patent N 5 276 897, Swedish Patent N 467 076 (1989), 1989.

106. P.B. Thistlewaite, M.A. McRobbie, and R.K. Meyer. *Automated Theorem-Proving in Non-Classical Logics*. Pitman Pub., 1988.

107. Alasdair Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

108. Allen Van Gelder and Yumi K. Tsuji. Efficiency and stability of hypergraph SAT algorithms. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring,*

and Satisfiability: the Second DIMACS Implementation Challenge. American Mathematical Society, 1993. Published 1996.

109. M.N. Velev and R.E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *38th Design Automation Conference (DAC '01)*, pages 226–231, June 2001.

110. Toby Walsh. The constrainedness knife-edge. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 1998.

111. Toby Walsh. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.

112. D.J. Watts and S.H. Strogatz. Collective dynamics of 'small world' networks. *Nature*, 393:440–442, 1998.

113. Larry Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

114. Larry Wos and Gail W. Pieper. *A Fascinating Country in the World of Computing*. World Scientific Pub. Co., 2000.

115. R. Zabih and D.A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-96)*, 1988.

116. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *LNAI*, pages 272–275. Springer-Verlag, 1997.

117. H. Zhang. Specifying latin squares in propositional logic. In R. Veroff, editor, *Automated Reasoning and Its Applications, Essays in honor of Larry Wos*, chapter 6. MIT Press, 1997.