Relevant Backtracking: Improved Intelligent Backtracking Using Relevance

Andrew Slater

Canberra Research Laboratory, National ICT Australia, Research School of Information Science and Engineering, Australian National University, Acton, 0200

Abstract. This work investigates and develops a backtracking algorithm with a novel approach to enumerating and traversing search space. A basic logical framework inspired by relevant logics is presented, highlighting relationships between search and refutation proof construction. Mechanisation of a relevance aware Davis Putnam Logemann Loveland procedure is investigated, and this yields an intelligent backtracking algorithm with abilities similar to other mechanisms including extended freedom in manipulating search space or rearranging refutation proof construction. Simplicity is achieved by a separation of concerns of the underlying logic and the construction of a sound and complete algorithm. The key advantage in the method is that it captures the notion of proof construction and relevant causality, and empirical analysis shows that this is an effective approach.

1 Introduction

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 paper 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* [20]. 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 various arenas, for example the propositional satisfiability (SAT) problem solvers `zchaff` [26], `GRASP` [22] and `relsat` [5] successfully use combinations of intelligent backtracking techniques.

This paper 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 [11, 19]). 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 for decades and their formalisms and properties are well known. A brief introduction to the practical mechanics of relevant logics will be described later.

In this paper 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 constructed. By basing the algorithm on a formal system of deduction, deriving the algorithm and associated mechanics of its reasoning becomes 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 also yields a framework for integrating other adjunct search techniques. For example, this includes integration of intelligent backtracking learning techniques and relaxation methods for clausal propositional satisfiability. The algorithm presented is as capable as other general intelligent backtracking mechanisms. This includes 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. The key difference between this and previous investigations is the constructive method focusing on relevance and causality, yielding an efficient mechanisation that incorporates more complex reasoning in order to more efficiently navigate through search space and intelligently construct refutation proofs.

2 Preliminaries

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

2.1 Search as Proof Construction

A satisfiability search process can also 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 [27]. The approach of proof construction focuses the investigation of intelligent backtracking at a purely logical level.

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 *conjunctive normal form* (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 paper is the Davis-Putnam-Logemann-Loveland algorithm (DPLL)[9] style approach of proof tree generation which has had great success in solving propositional 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 "forward looking" choice ordering procedures, 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, extending work in [17]). This could also be very useful in proof carrying code.

2.2 One of the problems 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 $(B \rightarrow \overline{C})$ 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

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

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 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 paper 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 paper 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 2.3 a mechanised proof system may use a multi-set). For example, if we have an assumption A and a formula that contains $(A \to B)\&(B \to C)$ then the deduction C depends upon A only and this shall be expressed by $C : \{A\}$. 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 dependency set labelling system encodes the possible deductions of implicational formula i.e. implication ($\to$) introduction as defined by the following rule

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

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:

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

We can also define a relevant style formulation for RAA which stems from the *Relevant Deduction Theorem* from relevant logic (see below and [11]) and the notation defined above.

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

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

Although these rules are sufficient for the work in this paper, care must be taken when defining constructive rules for other logical operations, for example maintaining the correct dependency sets when using Conjunction Introduction and then Conjunction Elimination. Retaining dependency set information for each conjunct is a possible solution, however further discourse concerning such situations is in the realm of sequent calculi and relevant logics and is beyond the scope of this work.

The Relevance of Relevant Logic. Relevant logics are motivated by the problems of the paradoxes of material implication found in classical reasoning. Notably, these non-classical logics are sub-structural as they lack certain structural rules present in classical logic and are thus considered weaker than classical logic. 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 paper. 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.* [1] and Dunn [11] 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). 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 [19]), though the original idea appears to date back as far as work by Gentzen in 1934 (see [11]). The work in this paper 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 [24, 25], largely due to the sub-structural nature of the logics. Alternative algorithmic approaches for non-classical logics have also been investigated [28]. It is finally noted that mechanised implementations of relevant logics exist [38, 7, 34], 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.

While the concept and mechanics of deductions sets have been taken from Hilbert or Natural Deduction formulations of relevant logics, exposing the relationships of relevant logics to this work are the subject of a separate work. It can be demonstrated that a fragment of the relevant logic RI 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. Further details can be found in [36].

2.3 An Overview of Intelligent Backtracking Techniques

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 also referred to Baker's thesis [3] which chronologically reviews fundamental intelligent backtracking techniques and the kinds of thrashing that they eliminate.

Backjumping Backjumping is the original approach for eliminating redundant conclusions devised by Gaschnig [12, 13]. This approach tested for redundancy at the leaves of the search tree. The method was improved by Prosser [30] 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 technical difference with the algorithm presented in this paper 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.

Backjumping search in the tree is executed in the same way as the DPLL algorithm. 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.

Relevant Formulated Backjumping The backjumping algorithm operating with dependency sets of assumptions is shown in Algorithm 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 paper. 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 or descriptions may use the call-stack or make separate copies in order to return the formula to its original state [6], 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.

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 this paper 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.

Algorithm 1 Using dependency sets with a backjumping algorithm

```
    $DependencySet$ **BJDP**($Formula\ F,\ AssignmentState\ A$)
 1: **if exists** $c : D \in F$ $such\ that$ $Empty(c)$ **then**
 2:
 3:     **return** $D$
 4: **else if** $Satisfied(F)$ **then**
 5:
 6:     **return** $SATISFIED$
 7: **end if**
 8: $a \leftarrow ChooseAssumption(F)$ {Select an assumption '$a$'}
 9: $UnitPropagate(a, \{a\}, F, A)$
10: $D \leftarrow BJDP(F, A)$
11: **if** $D \neq SATISFIED$ **then**
12:     $UnPropagate(a, \{a\}, F, A)$
13:     **if** $a \in D$ **then**
14:         $D_{\overline{a}} \leftarrow D - \{a\}$
15:         $UnitPropagate(\overline{a}, D_{\overline{a}}, F, A)$
16:         $D \leftarrow BJDP(F, A)$
17:         **if** $D \neq SATISFIED$ **then**
18:             $UnPropagate(\overline{a}, D_{\overline{a}}, F, A)$
19:         **end if**
20:     **end if**
21: **end if**
22:
23: **return** $D$
```

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.

Relevant Formulated 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, $UnitPropagate()$, is listed in Algorithm 2, and this shows how the operations on the dependency sets occur. 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.

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 using literal indexing techniques. 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.

Algorithm 2 Unit propagation for the backjumping algorithm

```
    UnitPropagate($Literal\ a,\ DependencySet\ D,$
        $Formula\ F,\ AssignmentState\ A$)
1:  $A \leftarrow \{a\} \cup A$
2:  for all $c : E \in F$ such that $a \in c$ do
3:      $MarkSubsumed(c : E)$
4:  end for
5:  for all $c : E \in F$ such that $\overline{a} \in c$ and $\overline{Subsumed(c)}$ do
6:      $MarkInactive(\overline{a},\ c : E)$
7:      $E \leftarrow E \cup D$
8:  end for
9:  for all $c : E \in F$ such that $\overline{a} \in c$ and $\overline{Subsumed(c)}$ and $Active(c : E) = \{x\} : E$
    do
10:     $MarkAsStandingDerivation(x : E)$
11:     $UnitPropagate(x, E, F, A)$
12: end for
```

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 details of 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 $\{x, p, a\}$

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

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 2 uses $MarkAsStandingDerivation()$). Once a unit is marked it remains the only derivation considered for that unit. The undoing process only executes a recursive $UnPropagate()$ (Algorithm 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. 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 used when a unit is withdrawn.

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 [37], 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 paper,

Algorithm 3 Undoing unit propagation for the backjumping algorithm

```
    UnPropagate($Literal\ a,\ DependencySet\ D,$
        $Formula\ F,\ AssignmentState\ A$)
 1: 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
 2:   $UnMarkAsStandingDerivation(x : E)$
 3:   $UnPropagate(x, E, F, A)$
 4: end for
 5: $A \leftarrow A - \{a\}$
 6: for all $c : E \in F$ such that $a \in c$ do
 7:   $UnmarkSubsumed(c : E)$
 8: end for
 9: for all $c : E \in F$ such that $\overline{a} \in c$ and $\overline{Subsumed(c)}$ do
10:   $MarkActive(\overline{a},\ c : E)$
11:   $E \leftarrow E - D$
12: end for
```

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 [10]. 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 [3, 6]. Bayardo and Schrag use *relevance-bounded learning* [4] 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 [6]. 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 [40, 22, 6]. It is generally used as an adjunct technique, i.e. in combination with choice heuristic or other search strategies. In this paper 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 paper, 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 2.3.

Dynamic Backtracking Dynamic Backtracking (DB) is another intelligent backtracking method motivated by the potential problem of losing work when backtracking over it [14]. 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 are 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. The algorithm essentially emulates the call stack of a traditional backtracking function storing the variable order, then the corresponding dependence information can then be extracted from the culprits array. This permits intermediate computation to be retained rather than erased upon backtracking. This method has been shown to be useful on some domains of constraint satisfaction problems (CSPs) like crossword solving [14]. 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 [2, 3]. 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 [3].

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, as outlined in the next section.

Partial Order Backtracking Systems Partial Order Backtracking (POB) systems observe and maintain a "partial order" of branching variables when executing an intelligent backtracking search, giving it the ability to select a backtrack point from those variables involved in creating a conflict, limited by the partial order information. This concept appears to have started with work by Bruynooghe et. al. [8, 33] who develops a traditional backtracking scheme utilising partial ordering. This is akin to keeping track of dependence, or partitioning related conflict information. McAllester developed the Partial Order Dynamic Backtracking (PODB) algorithm [23] which extends DB, an approach which retains completeness and a polynomial bound on the amount of information recorded during the search and gives significantly more freedom of movement during search. The work presented in this paper also takes advantage of the partial order in variable dependencies, though in a distinctly relevant fashion. Due to the complexities of these approaches a discussion and comparison is delayed until Section 3.7.

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. [18]), but a further challenge has been to devise a complete method.

Ginsberg and McAllester [15] investigate and compare the search behaviours of both Dynamic Backtracking and Partial Order Dynamic Backtracking with the stochastic but incomplete search behaviour of the GSAT algorithm [35]. They also develop an approach that extends the abilities of Partial Order Dynamic Backtracking which has greater freedom of movement, but requires exponential space [15].

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 [31, 32]. The core 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 [31] 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 `zchaff` for a wide variety of practical benchmark problems [26]. 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 paper 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. Hybridisation of such techniques with the relevant reasoning framework is a topic of further work.

3 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.

3.1 Mechanised Relevant Reasoning

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.

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 2.2. 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.

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 2.3 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 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 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). 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 2.3). 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 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. Modification of the proof progress list is left until later.

Algorithm 4 Dynamically undoing an assignment consistently

```
   UndoAssignment($Literal : DependencySet\ a : D$,
       $Formula\ F,\ AssignmentState\ A$)
1: $UnPropagate'(a, D, F, A)$
2: while $there\ exists\ c : E \in F\ such\ that\ \overline{Subsumed(c)}$ and $Active(c : E) = \{x\} : E$ do
3:     $MarkAsStandingDerivation(x : E)$
4:     $UnitPropagate'(x, E, F, A)$
5: end while
```

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 $F \rightarrow ()$ (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 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 3.3 and 3.4 this is revisited in far more detail.

Algorithm 5 Recanting an assumption

```
   Recant($Literal\ x$, $DependencySet\ D$, $ProgressList\ T$,
        $Formula\ F$, $AssignmentState\ A$)
1: for all $c : E \in T$ such that $x \in E$ do
2:    $UndoAssignment(c, E, F, A)$
3: end for
4: $Insert(\overline{x} : D - \{x\}, T)$
5: $UnitPropagate'(\overline{x}, D - \{x\}, F, A)$
```

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,

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

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 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 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.

3.3 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.

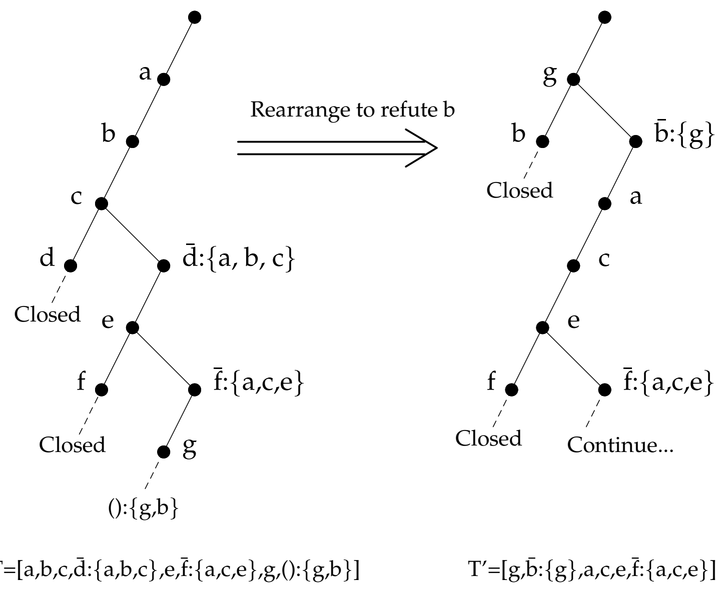

Fig. 1. The diagrams illustrate the analogy of the search tree to the proof progress list. At the bottom of the left hand tree a contradiction has been found. The right hand tree is a rearrangement where the assumption b is refuted by that contradiction.

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... to 001...) 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... to 0001...) 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... to 00010...) 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 3.1) 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.

3.4 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 3.8.

The methods discussed in Section 2.3 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) [16], 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 [29], 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 model the search space and attempt to approximate the change in this for various backtrack point choices. This technique has been used successfully to derive forward variable choice heuristics [39, 36]. 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 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.

Algorithmic details for a backward choice function, say *ChooseLegalRefutation()*, would depend on the heuristic chosen. For the purposes of this work 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.

3.5 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 ($ARB()$) in Algorithm 6. The algorithm is quite simple, relying on the embedded reasoning and proof construction process provided by previously defined routines.

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.

3.6 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 2.2 and 3.1 the proposed system provides a reasoning framework suitable for satisfiability search

Algorithm 6 An abstracted relevant style backtracking algorithm

```
    $AssignmentState$ **ARB**($Formula\ F$)
 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:
15:   **return** $A$
16: **else**
17:
18:   **return** $\{\}$
19: **end if**
```

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 work 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 2.2).

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 $DP*$. In terms of soundness and completeness, $DP*$ is a standard DPLL style algorithm with a fixed forward choice ordering, conflict directed intelligent backjumping and a limited clause recording technique, for all of which soundness and completeness is assumed.

Allow an algorithm execution $DP*_1$ to start a search on a given formula F. Every time $DP*_1$ backtracks (uses RAA) without rearrangement it records a clause for the variable it backtracks to. 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 $DP*_1$ is in, to some path list P_2. Now a second search execution, $DP*_2$, is started. It inherits any recorded knowledge from $DP*_1$ as they are just logical consequences of the given formula F. The difference between $DP*_1$ and $DP*_2$ 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 $DP*_1$ recorded. In terms of formula and assignment state $DP*_2$ can be seen to "catch up" to $DP*_1$. 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 $DP*_1$ and $DP*_2$ (and any further $DP*_i$'s used for other rearrangements) comprise a simulation of relevant backtracking by restarting a known sound and complete algorithm. The termination argument of Section 3.3 still holds as well – the restart enforces a progression in search space. In the final execution, $DP*_{end}$, 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 3.1).

3.7 Related Work – Partial Order Backtracking Systems

A discussion and comparison of intelligent backtracking systems taking advantage of a "partial order" for variable conflict analysis, briefly introduced in Section 2.3, will now be made. The observation of a partial order in backtracking search allows a limited rearrangement of earlier choices in order to best choose a variable to backtrack to. This is achieved by maintaining partial order information derived from conflict resolution. There are situations in during search where some variable ordering would have produced the exactly the same outcome as another, however committing to the ordering early forces the ordering of backtracking. The partial ordering information provides a way around this. Existing POB systems do not reason "irrelevantly", but they do not distinguish relevant assumptions, they discover them. This may detract from overall proof construction strategies.

Bruynooghe's method of partial order intelligent backtracking [8] 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. 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 (PODB) 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, whilst retaining intermediate information irrelevant to a backtrack conflict [23]. 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. A distinct difference between McAllester's approach and other methods is that the algorithm operates on a "total assignment" of the variables, and maintains a selection of "no-goods" or identified conflicts which are consistent with that assignment. The search progresses via the selection of new conflicts, in contrast with other typical backtracking approaches which construct a partial assignment which is consistent with the set of given clauses. This yields quite a different perspective on choices made during search, both forward and backward.

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 proof progress list provides the equivalent of a relevantly derived partial order. The elements defining this partial order consist only of assumptions or the original causes. This results in providing a more informed collection of variables when choosing a backtrack point and should represent the overall strategy taken to construct the proof. Recording assumptions in the progress list results in succinct reasoning and is a useful way of focusing the process of proof construction.

What previous partial order backtracking systems lack is the notion of relevant causality. The relevant backtracking mechanism commits to an "assumption" and carries that information through to each conflict. A relevantly derived conflict records the assumptions used to derive the conflict, not just the variables in the clause that has become empty. A POB system may, through successive steps of resolution and backtracking, discover the ultimate cause, but given there may be many possible paths of inference. For a simplistic example, suppose we have deduced a from assumptions $\mathit{\Gamma}$ and x from $\mathit{\Sigma}$ and the sub-formula contains

$$(a \rightarrow b)(x \rightarrow c)(c \rightarrow f)(b \rightarrow \overline{f})$$

On deriving a and x the relevant system immediately discovers the cause lies in $\mathit{\Gamma} \cup \mathit{\Sigma}$. A partial order backtracking system, assuming unit propagation is implemented, finds the cause is from b or c. It then must choose which to refute: If it chooses b it can resolve back through to a and further on to some element of $\mathit{\Gamma}$. The choice of c leads to an element of $\mathit{\Sigma}$. Thus the early choice of conflict resolution commits the search to a constrained set of ultimate backtrack points, and there is no way of determining the full set without evaluating all possible backward resolution paths. The same argument holds for PODB: When it finds a conflict it recursively resolves the collection of no-goods, but must make a (refutation) variable selection at each step. We note here that it should be possible to include a mechanism to evaluate all backward resolution paths with polynomial overhead, but this would require some forethought as to how to construct resulting mechanised proofs, making any given algorithm increasingly complex. The complete recognition of cause and effect within relevant style reasoning means that conflict resolution is more efficient since it avoids resolution search back to some cause, but perhaps more importantly it remains focused on the variables it has selected as assumptions, and the selection of backtrack points is far more informed.

One important further advantage of using a basis of relevance style logic means that the reasoning rules corresponding to techniques, such as Unit Propagation, are easily incorporated, whereas it is not immediately obvious how other POB systems would implement additional rules for reasoning. The addition of such techniques are clearly crucial to constructing an efficient advanced satisfiability solver.

3.8 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 – a difficult domain for systems based purely on intelligent backtracking. This provides a relatively simple and unstructured scenario for comparison and evaluation of a backtracking mechanism's ability to prune and recover. The experimental method and the backtracking schemes implemented are detailed below.

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. The experiments were carried out a 2.66 GHz Pentium 4 with 1Gb of memory running a Linux operating system.

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.

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 it is generally well known that intelligent backtracking algorithms do not offer a good solution. The experiments presented in this work 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.

The most effective method to solve random 3-SAT problems is an advanced choice heuristic, and the additional presence of intelligent backtracking makes very little difference to performance. In order to simulate "unknown territory" a simple fixed branching scheme using a predetermined fixed order of choice, and a randomised branching scheme, introducing further perturbation to the search, were used. 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.

Back Tracking Schemes The backtracking schemes that were used in these experiments are now described. 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 work.

UD Dynamic Backtracking with unit propagation enabled – the method as described by Baker [3] is followed in order to include unit propagation with Dynamic Backtracking. Without unit propagation Dynamic Backtracking performs very badly [3]. 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.

RB Relevant backtracking proof search – this is just one 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 3.4. Although the strategy is basic it should concentrate on generating smaller proofs.

An implementation of a standard partial order backtracking scheme was not included due to complexity of the techniques and the lack of any concrete methods to efficiently implement unit propagation and a cooperating conflict clause selection mechanism. It is clear that unit propagation is an element crucial to the efficiency in satisfiability solving, as exampled by Baker's work with Dynamic Backtracking. Without it any backtracking scheme fails to perform.

Search Cost Results Table 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 backtracking greedy proof search strategy (RB) 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. RB 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. The actual depth of

Number of Variables	BT	BJ	UD	RB
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 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.

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 [21]. This second experiment is identical to the first, but uses a random branching scheme. The results are shown in Table 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

Number of Variables	BT	BJ	UD	RB
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 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.

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 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. 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 long run.

Number of Variables	BT	BJ	UD	RB
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. 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.

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.

RB 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. The relevant approach takes advantage of the ongoing search tree structure in terms of the assumptions made during search and can modify or correct the structure in order to produce a more succinct proof and reduce overall search cost.

Search Cost Results in Time Time comparisons can rely on the underlying implementation and even the operating environment. The backtracking algorithms were designed with simplicity as the primary guiding factor, and the resulting implementations are quite generic. Each implementation of a search algorithm was implemented as fairly as possible. Where the basic 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). 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. These optimisation cases are particularly true of the algorithms that spend more time processing information at each node of the search tree.

Unsurprisingly the execution times for the algorithms using a fixed branching scheme, in Table 4, show that the overall cost of intelligent backtracking is too expensive when compared to the naive backtracking method when solving random 3-SAT problems. 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	RB
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 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 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 RB.

Number of Variables	BT	BJ	UD	RB
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 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.

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 RB, even using this implementation, is far more advantageous in terms of execution time.

4 Conclusions

This work 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. Identifying assumptions as the ultimate 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, due to 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 with respect to the relevant assumptions made during search. The simplicity of the approach allows for efficient search space traversal, and a framework for developing strategies for search and proof construction. Experimental analysis showed that these advantages are obtainable and effective. The performance differences are highly significant when compared to other backtracking methods.

Further Work It is apparent that the relevant logical framework has much promise, and is amenable to the addition of further reasoning techniques. There are several areas that would be interesting and beneficial to investigate, some of which have been mentioned in this work. Further work includes investigations of i) alternative rearrangement strategies and planning heuristics, including a complementary forward and backward choice mechanism, ii) extensions integrating other reasoning techniques such as clause recording, leading to sub-proof representation and planning proof structure, and iii) relevant syntactic representations for mechanising richer logics.

Acknowledgements The author wishes to thank John Slaney, Rajeev Goré, Bob Meyer and reviewers for helpful comments and feedback on this work.

References

1. A.E. Anderson, N.D. Belnap, and J.M. Dunn. *Entailment: the Logic of Relevance and Necessity Vol 2.* Princeton University Press, 1992.

2. Andrew Baker. The hazards of fancy backtracking. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 288–293, 1994.
3. Andrew Baker. *Intelligent Backtracking On Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, University of Oregon, 1995. CIS-TR-95-08.
4. 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.
5. 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.
6. 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.
7. A. W. Bollen. Relevant logic programming. *Journal of Automated Reasoning*, 7(4):563–585, 1991.
8. M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, February 1981.
9. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.
10. Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
11. 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.
12. 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.
13. John Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, 1979.
14. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
15. 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.
16. 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.
17. Mike Gordon. *HolSatLib.* Computer Laboratory, University of Cambridge, 1.0b edition, 2001.
18. 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.
19. E.J. Lemmon. *Beginning Logic.* Chapman & Hall, 1991.
20. 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.

21. 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.
22. 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.
23. David A. McAllester. Partial order backtracking. Technical report, MIT Artificial Intelligence Laboratory, http://www.ai.mit.edu/people/dam/, 1993.
24. M.A. McRobbie, A.E. Anderson, N.D. Belnap, and J.M. Dunn. *Relevant analytic tableaux*, chapter X:60. In [1], 1992.
25. Michael A. McRobbie and Nuel D. Belnap. Relevant analytic tableaux. *Studia Logica*, 38:187–200, 1979.
26. 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.
27. 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.
28. Nicola Olivetti. *Algorithmic Proof Theory for non-classical and modal logics*. PhD thesis, Dipartimento di Informatica, Università di Torino, 1995.
29. 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.
30. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
31. E. Thomas Richards. *Non-systematic search and No-good learning*. PhD thesis, University of London, IC Parc, 1998.
32. E. Thomas Richards and Barry Richards. Nonsystematic search and no-good learning. *Journal of Automated Reasoning*, 24(4):483–533, 2000.
33. W. Rosiers and M. Bruynooghe. Empirical study of some constraint satisfaction algorithms. In *Artificial Intelligence II, Methodology, Systems, Applications, Proc. AIMSA'86*, 1986.
34. 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.
35. 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.
36. Andrew Slater. *Investigations into Satisfiability Search*. PhD thesis, Computer Sciences Laboratory, The Australian National University, Canberra, 2003.
37. 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.
38. P.B. Thistlewaite, M.A. McRobbie, and R.K. Meyer. *Automated Theorem-Proving in Non-Classical Logics*. Pitman Pub., 1988.
39. Toby Walsh. The constrainedness knife-edge. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 1998.

40. 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.