A Rearrangement Search Strategy for Determining Propositional Satisfiability

Ramin Zabih
Computer Science Department
Stanford University

David McAllester
Computer Science Department
Cornell University

Abstract

We present a simple algorithm for determining the satisfiability of propositional formulas in Conjunctive Normal Form. As the procedure searches for a satisfying truth assignment it dynamically rearranges the order in which variables are considered. The choice of which variable to assign a truth value next is guided by an upper bound on the size of the search remaining; the procedure makes the choice which yields the smallest upper bound on the size of the remaining search. We describe several upper bound functions and discuss the tradeoff between accurate upper bound functions and the overhead required to compute the upper bounds. Experimental data shows that for one easily computed upper bound the reduction in the size of the search space more than compensates for the overhead involved in selecting the next variable.

1 Introduction

Determining the satisfiability of propositional formulas in Conjunctive Normal Form has been an important problem for Computer Science. It was the first problem to be proven $\mathcal{NP}$-complete [Cook, 1971]. This problem is important because any inference system for propositional logic that is complete must effectively determine propositional satisfiability. In addition, many search problems have natural encodings as CNF formulas. We are particularly interested in constraint satisfaction problems, a class which includes such problems of interest to the Artificial Intelligence community as the n-queens problem. These problems have simple encodings as CNF formulas, so an algorithm for determining satisfiability provides a way of solving them.

It is well known that the size of the search tree for a given problem can depend heavily on the order in which choices are made. We have designed an algorithm for determining propositional satisfiability that dynamically rearranges the order in which propositional variables are assigned truth values. This selection is guided by an upper bound on the size of the remaining search problem; we arrange the search ordering to decrease this upper bound as quickly as possible.

We present two different upper bounds on the size of the search remaining. The first bound is related to 2-level dynamic search rearrangement, a strategy described by Purdom, Brown and Robertson in [1981]. This bound provides a good estimate of the size of the remaining search but the overhead of computing it appears to be too high for practical application. Our second upper bound is weaker but can be calculated in constant time at any node in the search tree. The rearrangement search procedure based on this upper bound appears to be a slight improvement for the n-queens problem over the best of the 10 algorithms surveyed in [Haralick and Elliot, 1980], and shows promise on certain graph-coloring problems.

This paper begins with some notes on propositional satisfiability. We also present some simple algorithms for determining satisfiability, culminating in a version of the Davis-Putnam procedure. Section 3 describes the upper bound functions and the rearrangement search procedures based on them. Section 4 compares our algorithms with other work, focusing on 2-level dynamic search rearrangement. Section 5 provides some empirical results about the performance of our algorithms.

2 The Satisfiability Problem

In order to discuss the problem of satisfiability, we first need some definitions and some notation.

Definition: A *literal* l is either a proposition (symbol) φ or the negation $\neg\varphi$ of a proposition φ. A *clause* C is a disjunction of literals $l_1 \vee l_2 \vee \ldots \vee l_r$. A *Conjunctive Normal Form (CNF) formula* Ψ is a conjunction of clauses $C_1 \wedge C_2 \wedge \ldots \wedge C_s$. A *literal occurrence* is a pair (l, C) such that $l \in C$. $|\Psi|$ is the number of distinct literal occurrences in Ψ, i.e., the length of Ψ written as a formula. $\|\Psi\|$ is the number of distinct proposition symbols which appear in Ψ ($\|\Psi\|$ is no larger than $|\Psi|$, and usually much smaller).

Definition: A *labeling* ρ is a mapping from propositions to truth values, i.e. the set $\{\mathcal{T}, \mathcal{F}\}$. If a labeling is defined on all the propositions that appear in a formula, then we will say that labeling is *complete*; otherwise, the labeling is *partial*. Let $\|\rho\|$ denote the number of different propositions that ρ assigns a truth value. If $\rho(\varphi)$ is the truth value v then we define $\rho(\neg\varphi)$ to be the opposite of v. A labeling ρ thus gives a truth values to literals as well as propositions. For any labeling ρ, literal l, and truth value v, we define $\rho[l \leftarrow v]$ as the labeling which is identical to ρ except that it assigns the literal l the value v.

Definition: A clause $C \in \Psi$ is *violated* by a labeling ρ if ρ labels all of C's literals with $\mathcal{F}$. A clause C is *satisfied* by ρ if ρ labels any literal in C with $\mathcal{T}$. If C is neither violated nor satisfied by ρ, then we say C is *open*. If C is open and ρ labels all but one of C's literals with $\mathcal{F}$, then C is a *unit open clause*.

Definition: A formula Ψ is violated by a labeling ρ if there exists some clause $C \in \Psi$ that is violated by ρ. Similarly, Ψ is satisfied by ρ if every clause $C \in \Psi$ is satisfied by ρ.

We are interested in determining the satisfiability of an arbitrary CNF formula Ψ. Ψ is satisfiable just in case there exists a labeling of the propositions in Ψ that satisfies Ψ; otherwise, Ψ is unsatisfiable. The set of satisfiable CNF formulas is known as SAT.

CNF formulas arise in a variety of applications. We are particularly interested in constraint satisfaction problems [Montanari, 1974]. Constraint satisfaction problems require finding a consistent value assignment to a set of variables subject to constraints. Many well-known problems from artificial intelligence or combinatorics, such as the n-queens problem and finding a coloring for a graph, are constraint satisfaction problems. Any constraint satisfaction problem can be straightforwardly (and efficiently) compiled into an equivalent CNF formula. Solutions to the original problem will naturally correspond to truth labelings on the formula, and vice versa.

We can always determine the satisfiability of a CNF formula Ψ by enumerating all complete labelings of Ψ's propositions. This produces a search of size $O(2^{\|\Psi\|})$. Because SAT is $\mathcal{NP}$-complete, we cannot expect to find a polynomial time algorithm for computing satisfiability. We are interested in correct algorithms that perform well on problems of practical interest. We will present algorithms that attempt to search the labelings as efficiently as possible.

2.1 A Simple Algorithm

Our first algorithm is slightly more clever than enumerating all the $2^{\|\Psi\|}$ labelings of Ψ. We look at a clause at a time, construct a labeling that satisfies that clause, and move on to the next clause to be satisfied. We choose clauses rather than propositions for reasons that will become clear in section 3.

ALGORITHM **Clause-Search**(Ψ, ρ):

1. [Termination check] If ρ violates a clause in Ψ, then return. If there are no remaining open clauses, then print out ρ and halt.
2. [Clause selection] Select a clause C from Ψ which is open under ρ.
3. [Recursion] For each unlabeled literal $l \in C$, call **Clause-Search**$(\Psi, \rho[l \leftarrow \mathcal{T}])$.

We can now determine the satisfiability of Ψ by calling **Clause-Search**$(\Psi, \emptyset)$, where $\emptyset$ is the empty labeling. If Ψ is satisfiable, this will produce a labeling that satisfies Ψ; otherwise, Ψ is unsatisfiable. (Strictly speaking, we might not produce a complete labeling satisfying Ψ. However, a partial labeling that satisfies Ψ and leaves m propositions unlabeled can be viewed as standing for 2^m complete labelings that satisfy Ψ.)

This algorithm can be improved in a fairly straightforward way. Suppose that $C = l_1 \vee l_2$ is the open clause we choose in step 2, and that ρ doesn't label either l_1 or l_2. The first recursive call in step 3 will find any solution for Ψ that is an extension (superset) of $\rho[l_1 \leftarrow \mathcal{T}]$. The second recursive call will find any solution that extends $\rho[l_2 \leftarrow \mathcal{T}]$. In the second recursive call we can assume without loss of generality that l_1 is labeled $\mathcal{F}$, since we already checked for solutions with l_1 labeled $\mathcal{T}$ in the first recursive call. Thus we can replace step 3 with:

3′. [Recursion] Repeat the following while C is open under ρ. Choose an unlabeled literal $l \in C$, call **Clause-Search**$(\Psi, \rho[l \leftarrow \mathcal{T}])$, and set ρ equal to $\rho[l \leftarrow \mathcal{F}]$.

Using step 3′ the first literal in the chosen clause will be set to $\mathcal{F}$ before the other literals are considered. The procedure iteratively chooses a literal and first searches for solutions where the literal is true, then searches for solutions where the literal is false. In this way it explores a binary search tree where each node in the tree is associated with a particular proposition. The number of nodes in such a search tree is $O(2^{\|\Psi\|})$.

2.2 Boolean Constraint Propagation

There is another easy improvement we can make in our search procedure. The algorithm **Clause-Search** deals badly with unit open clauses. Suppose that as a result of the extension of ρ we incrementally construct in step 3′, some clause C in Ψ is now a unit open clause with unlabeled literal l. Then we can extend ρ to $\rho[l \leftarrow \mathcal{T}]$ without loss of generality, because any extension of ρ which labels l with $\mathcal{F}$ will violate C. Furthermore, replacing ρ with $\rho[l \leftarrow \mathcal{T}]$ can cause other clauses of Ψ to become unit open clauses, as $\neg l$ can appear in other clauses, and so the process can repeat. Boolean Constraint Propagation extends a truth labeling by closing all unit open clauses.

ALGORITHM **BCP**(Ψ, ρ):

1. [Clause propagation] If Ψ contains any unit open clauses under ρ, then select such a clause C, find the unlabeled literal $l \in C$ and return **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$.
2. [Termination] Otherwise return the labeling ρ.

For a CNF formula Ψ and partial labeling ρ, every extension of ρ that satisfies Ψ is also an extension of the partial labeling **BCP**(Ψ, ρ). Unless the labeling returned in step 2 violates Ψ, the order in which unit open clauses are chosen has no effect on the result.

BCP is sufficient to determine the satisfiability of a reasonable class of CNF formulas (including, but not limited to, Horn clauses). With appropriate preprocessing and use of efficient data structures, **BCP** can be made to run in time proportional to $|\Psi|$, the number of literal occurrences in Ψ. More precisely, one can implement a version of **BCP** that does no more than a constant amount of work for every literal occurrence in the input formula. These points are discussed in more detail in [McAllester, 1987].

2.3 Using Constraint Propagation

We can now improve our clause search algorithm by running Boolean Constraint Propagation to extend ρ.

ALGORITHM **BCP-Search**(Ψ, ρ):

0. [Propagation] Set $\rho = $ **BCP**(Ψ, ρ).

1. [Termination check] If ρ violates a clause in Ψ, then return. If there are no remaining open clauses, then print out ρ and halt.
2. [Clause selection] Select a clause C from Ψ which is open under ρ.

3′. [Recursion] Repeat the following while C is open under ρ. Choose an unlabeled literal $l \in C$, call **BCP-Search**$(\Psi, \rho[l \leftarrow \mathcal{T}])$, and set ρ to **BCP**$(\Psi, \rho[l \leftarrow \mathcal{F}])$.

BCP-Search is essentially the propositional component of the procedure that Davis and Putnam described in [1960]. Davis and Putnam, however, did not provide any heuristics for selecting clauses in step 2 of the procedure. The size of the search space can depend strongly on the decision made in step 2. Our focus is on novel heuristics for clause selection.

3 Search Rearrangement Heuristics

Our basic observation is that at every point in the search tree there is a natural upper bound on the size of the remaining search. Suppose that we are looking for extensions of a partial labeling ρ which leaves unlabeled n propositions that appear in open clauses. Then there are only 2^n extensions of ρ that can satisfy Ψ. Formally, we have the following definition and simple lemma.

> **Definition:** A *future proposition* for Ψ and ρ is a proposition that is not labeled by ρ which appears in an open clause of Ψ under ρ. We define $\kappa(\Psi, \rho)$ to be the number of future propositions for Ψ and ρ.
>
> **Lemma:** The number of terminal nodes in the search tree generated by **BCP-Search**(Ψ, ρ) is no larger than $2^{\kappa(\Psi, \rho)}$.

Now consider the recursive calls to **BCP-Search** in step 3 of the procedure. In each recursive call some unlabeled literal l of the chosen clause C will be assigned the the value $\mathcal{T}$. A recursive call with $l \leftarrow \mathcal{T}$ will produce a search tree with at most

$$O(2^{\kappa(\Psi, \rho[l \leftarrow \mathcal{T}])})$$

nodes. If we sum this quantity over the unlabeled literals of C, we have a bound on the size of the search remaining if we pick the clause C.

Our idea is to pick the clause which minimizes such a bound on the remaining search space. We generate a much better bound than this, however. Notice that at step 0 we replace ρ by **BCP**(Ψ, ρ). This fact, together with the lemma, produces the following corollary.

> **Corollary:** The number of terminal nodes in the search tree generated by **BCP-Search**(Ψ, ρ) is no larger than $2^{\kappa(\Psi, \mathbf{BCP}(\Psi, \rho))}$.

Since **BCP** can considerably reduce the number of future propositions, this is a much better upper bound. We will use this to select among clauses.

Suppose that at step 2 we select a clause C which has unlabeled literals $S = \{l_1, l_2, \ldots, l_r\}$. If we define

Clause-BCP-Bound(Ψ, C, ρ)

$$\stackrel{\text{def}}{=} \sum_{l \in S} 2^{\kappa(\Psi, \mathbf{BCP}(\Psi, \rho[l \leftarrow \mathcal{T}]))}$$

then we have the following result.

> **Corollary:** If step 2 of **BCP-search**(Ψ, ρ) selects clause C, then the number of terminal nodes in the search tree generated by step 3 is no larger than **Clause-BCP-Bound**(Ψ, C, ρ).

We can now simply pick the clause to work on which minimizes the above upper bound on the remaining search. This produces our new reordering search algorithm.

ALGORITHM **BCP-Reorder**(Ψ, ρ):

0. [Propagation] Set $\rho = $ **BCP**(Ψ, ρ).
1. [Termination check] If ρ violates a clause in Ψ, then return. If there are no remaining open clauses, then print out ρ and halt.

2′. [Clause selection] Select the clause $C \in \Psi$ that is open under ρ and that minimizes the value of **Clause-BCP-Bound**(Ψ, C, ρ).

3. [Recursion] Repeat the following while C is open under ρ. Choose an unlabeled literal $l \in C$, call **BCP-Reorder**$(\Psi, \rho[l \leftarrow \mathcal{T}])$, and then set ρ equal to **BCP**$(\Psi, \rho[l \leftarrow \mathcal{F}])$.

BCP-Reorder can be characterized as a "greedy" algorithm, because it attempts to decrease an upper bound on the remaining search space as quickly as possible.

3.1 Using Stored Labelings

In order to select the correct clause at step 2, we must calculate **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$ for every unlabeled literal l that appears in some open clause. The overhead involved can be greatly reduced by incrementally maintaining additional labelings. More specifically, for each literal l_i we explicitly store a distinct labeling ρ_i. The labelings ρ_i are related to the base labeling ρ by the invariant

$$\rho_i \quad = \quad \mathbf{BCP}(\Psi, \rho[l_i \leftarrow \mathcal{T}]).$$

Whenever ρ is updated in the above procedure, each of the stored labelings ρ_i must also be updated to maintain this invariant. There are at most $2 \cdot \|\Psi\|$ literals l which appear in Ψ, so explicitly storing the labelings ρ_i requires $O(\|\Psi\|^2)$ space.

The total time required to incrementally update all the stored labelings down a single path in the search tree is $O(\|\Psi\| \cdot |\Psi|)$ (there are $2 \cdot \|\Psi\|$ labelings, each of which can require at most $|\Psi|$ total updating time). By spreading the cost of updating the stored labelings over all the search nodes in a given search path, we can reduce the overhead significantly.

There are also important improvements which can be made by taking advantage of the incrementally stored labelings ρ_i. Recall that the procedure must maintain the invariant that ρ_i equals **BCP**$(\Psi, \rho[l_i \leftarrow \mathcal{T}])$. This implies that if ρ_i violates Ψ then any extension of ρ which satisfies Ψ must assign l_i the value $\mathcal{F}$. In this case we can set the base labeling ρ to be $\rho[l_i \leftarrow \mathcal{F}]$ and correspondingly update all the other labelings ρ_i. We will call this *assignment by refutation*; we assign $l \leftarrow \mathcal{F}$ because **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$ violates Ψ, thus refuting $l \leftarrow \mathcal{T}$.

We can take advantage of storing the labelings ρ_i to provide a somewhat stronger propagation algorithm than **BCP**.

ALGORITHM **BCP2**(Ψ, ρ):

1. [Propagation] Set ρ equal to **BCP**(Ψ, ρ).
2. [Recursion] If there exists a literal l which is not labeled by ρ, which appears in some clause of Ψ which is open under ρ, and which has the property that **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$ violates Ψ, then return **BCP2**$(\Psi, \rho[l \leftarrow \mathcal{F}])$. Otherwise, return ρ.

This algorithm extends the labeling ρ so that there are no unit open clauses left, and is at least as strong as **BCP**. **BCP2**(Ψ, ρ) may produce an extension of **BCP**(Ψ, ρ), so **BCP2** is a stronger propagator than **BCP**.

As the notation **BCP2** suggests, one can define a series of ever more powerful constraint propagation function **BCP3**, **BCP4**, etc. However, the constraint propagators stronger than **BCP2** have a very large overhead which probably renders them useless in practice. If we are maintaining the labelings ρ_i for other reasons, as in the above search procedure, then there is no additional overhead in replacing the search procedure's explicit calls to **BCP** with calls to the more powerful **BCP2**.

3.2 An Easily Computed Upper Bound

The need to explicitly calculate the $O(\|\Psi\|)$ different partial labelings of the form **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$ results in considerable overhead. It turns out that useful, but weaker, upper bounds on the search size can be computed much more efficiently. In order to define the clause selection process based on this weaker upper bound some new terminology is needed.

Definition: An *open binary clause* under a labeling ρ is a clause with two literals, neither of which is labeled by ρ. Let **Open-Binaries**(Ψ, ρ, l) denote the number of open binary clauses in Ψ which contain the literal l. Let $\overline{\kappa}(\Psi, \rho, l)$ denote

$$\kappa(\Psi, \rho) - \textbf{Open-Binaries}(\Psi, \rho, \neg l).$$

This is the number of future propositions minus the number of open binary clauses containing the opposite literal of l.

It is not immediately obvious that this is a useful quantity to compute. However, it can provide a bound on the size of the search remaining.

Lemma: The number of terminal nodes in the search tree that **BCP-Search**$(\Psi, \rho[l \leftarrow \mathcal{T}])$ generates is no larger than $2^{\overline{\kappa}(\Psi, \rho, l)}$.

Proof: Let ρ' be **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$. If ρ' violates some clause in Ψ then there are no search nodes under the node with the labeling ρ', so the relation holds. Now assume ρ' does not violate any clause in Ψ. In this case we can prove

$$\|\rho'\| \geq \|\rho\| + \textbf{Open-Binaries}(\Psi, \rho, \neg l).$$

To see this note that when we set l equal to $\mathcal{T}$ every open binary clause which contains the opposite of l will be come an open unit clause and thus lead to propagation. All of the open binary clauses which contain the opposite of l are distinct (assuming Ψ contains no duplicate clauses), so each clause will lead to a truth assignment to a different literal. If **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$ does not violate any clause in Ψ then all these literals must involve distinct propositions and the above relation holds. There are no more than $2^{\|\Psi\| - \|\rho'\|}$ distinct extensions of ρ' that can satisfy Ψ, and every terminal node of the search tree is a distinct extension, so the lemma follows. $\square$

This lemma states that $\overline{\kappa}(\Psi, \rho, l)$ yields an upper bound on the search remaining when we label $l \leftarrow \mathcal{T}$. Furthermore the number $\overline{\kappa}(\Psi, \rho, l)$ can be computed without knowing **BCP**$(\Psi, \rho[l \leftarrow \mathcal{T}])$. In fact, it can be calculated with constant overhead. It is also easy to verify

$$\forall\, \Psi, \rho, l \qquad \kappa(\Psi, \textbf{BCP}(\Psi, \rho[l \leftarrow \mathcal{T}])) \leq \overline{\kappa}(\Psi, \rho, l),$$

which shows that $\overline{\kappa}(\Psi, \rho, l)$ provides a weaker upper bound than $\kappa(\Psi, \textbf{BCP}(\Psi, \rho[l \leftarrow \mathcal{T}]))$.

If we choose to work next on an open clause C with unlabeled literals $S = \{l_1, l_2, \ldots l_r\}$ then the remaining search will be no larger than

$$\textbf{Clause-Occur-Bound}(C, \rho) \quad \stackrel{\text{def}}{=} \quad \sum_{l \in S} 2^{\overline{\kappa}(\Psi, \rho, l)}.$$

We can now simply pick the clause to work on which minimizes this upper bound on the remaining search. This produces a variant of the previous reordering search algorithm where the clause selection at step $2'$ uses **Clause-Occur-Bound** rather than **Clause-BCP-Bound**. We call the resulting procedure **Occur-Reorder**.

One objection to **Occur-Reorder** might be that it relies on the existence of binary clauses in the original CNF formula Ψ. While this may be a problem in general, it is not a problem for formulas which represent constraint satisfaction problems with only binary constraints, such as the n-queens problem or graph coloring.[1] The natural translation of a binary constraint satisfaction problem uses binary clauses to represent the fact that two values of mutually constrained variables are mutually inconsistent. These binary clauses play a central role when using **Occur-Reorder** to find a satisfying assignment to the CNF encoding of a binary constraint satisfaction problem.

4 Related Work

The algorithm **BCP-Reorder** is closely related to dynamic 2-level search rearrangement, as described by Purdom, Brown and Robertson in [1981]. Purdom, Brown and Robertson use a simple backtrack procedure which, like our procedure, takes a given partial assignment ρ and searches for an extension of ρ which satisfies the given CNF formula Ψ. If the given partial labeling ρ does not already satisfy every clause in Ψ, and if ρ does not violate any clause in Ψ, then the Purdom, Brown and Robertson procedure selects some proposition Φ and recursively searches for extensions of $\rho[\Phi \leftarrow \mathcal{T}]$ and $\rho[\Phi \leftarrow \mathcal{F}]$. The efficiency of the search is sensitive to exactly which proposition is selected for assignment in the recursive calls. Different selection techniques correspond to different search algorithms.

[1] Since any non-binary constraint satisfaction problem can be converted into a binary one in polynomial time, it is possible that this algorithm could be effective even on constraint satisfaction problemswith non-binary constraints.

Let us call a proposition φ *forced* for a partial assignment ρ and a CNF formula Ψ if one of the assignments $\rho[\varphi \leftarrow \mathcal{T}]$ or $\rho[\varphi \leftarrow \mathcal{F}]$ violates some clause in Ψ. The Purdom, Brown and Robertson procedure always selects forced propositions before non-forced propositions. For each forced proposition one of the two recursive calls to the search procedure immediately fails. The process of selecting a series of forced propositions corresponds precisely to running **BCP**.

If there are no forced propositions then the procedure must select among the unforced propositions. Purdom, Brown and Robertson's procedure involves a parameter β; they suggest setting β equal to the average branching factor in the search tree. If β is set equal to 2 then the 2-level choice heuristic described by Purdom, Brown and Robertson selects the proposition φ which minimizes the sum

$$\kappa(\Psi, \mathbf{BCP}(\Psi, \rho[\varphi \leftarrow \mathcal{T}])) + \kappa(\Psi, \mathbf{BCP}(\Psi, \rho[\varphi \leftarrow \mathcal{F}])).$$

This sum is an upper bound on the number of leaf nodes in the search tree generated when Φ is selected as the next proposition. This upper bound is simpler than our clause-based upper bound.

To compare our clause-based bound and the above proposition-based bound, some general observations about upper bounds are needed. Different choices of proposition order result in different search trees. For each proposition order one can examine a root fragment of the search tree and compute an upper bound on the number of leaf nodes in the total tree. More specifically, for each node n let $k(n)$ be the number of future propositions at search node n. Given a fragment of the search tree below a node n one can compute the sum of $2^{k(m)}$ for all nodes m at the fringe of the fragment tree. This sum is an upper bound on the number of leaf nodes in the entire search tree below n. In summary, given a proposition order one can compute a root fragment of the remaining search tree and from that fragment one can compute an upper bound on the size of the remaining search. One can then choose the proposition order which minimizes this computed upper bound.

The Purdom, Brown and Robertson 2-level procedure performs a certain lookahead into the search tree for each possible proposition which might be selected. Our selection procedure can also be viewed as computing an upper bound by looking ahead into the search. By selecting clauses rather than propositions, however, our procedure makes a commitment to a certain order of propositions down one branch of the search tree. Given this commitment, we can compute an upper bound which is based on a larger root fragment of the search tree. Thus our procedure gets a tighter upper bound by effectively examining a larger fragment of the search tree.

5 Experimental Results

We have implemented the algorithms described in this paper, and used them to find all the solutions to several problems. The problem we have examined most intensively is the n-queens problem, which is to place 1 queen in each column of an n-by-n chessboard so that no 2 queens attack.

Reordering Strategy	Search size	Assignments	Time
Occur-Reorder	642	5,619	8.5
MIN	704	6,362	9.7
BCP-Reorder	316	311,272	1420

Figure 1: Performance of various algorithms on the 8-queens problem. Running time in seconds on a Symbolics 3650.

5.1 Methodology

We measure performance with three metrics. The size of the search tree, the total number of labelings considered, is our first metric. Our second metric is the number of truth assignments, which is the number of times that a labeling ρ is extended to $\rho[l \leftarrow \mathcal{T}]$ for some literal l. The first metric tells how good a search tree a given algorithm produces, ignoring any extra overhead that it introduces. The second metric provides an measure of the total amount of work that an algorithm does, taking overhead into account. It can be thought of as the running time of an ideal implementation. Our final metric is the actual running time of our implementation.

Ties provide the major source of statistical fluctuation in our data. When an several appear equally good, we choose one at random. This effects both the size of the search tree and the number of truth assignments. Another source of randomness is the precise order in which **BCP** examines unit open clauses when it discovers a contradiction. This will not effect the size of the search tree, but does effect the number of truth assignments and the actual running time.

The algorithm that we use as a standard of comparison is what Haralick and Elliot [1980] call "optimal order forward checking", which is the best algorithm of the 10 they survey. We follow Stone's terminology [1986] and refer to this algorithm as **MIN**. This algorithm operates on constraint satisfaction problems by selecting the most constrained variable to examine at each point in the search.[2] Since we are interested in solving constraint satisfaction problems by compiling them to SAT, we have implemented **MIN** as a SAT algorithm.

Our implementations of **MIN** and **Occur-Reorder** share all of their code except for the function that chooses the next clause to examine. This makes it plausible to compare their running times. Our implementation of 2-level dynamic search rearrangement makes use of stored labelings in much the same way as our implementation of **BCP-Reorder**; these two algorithms also share almost all of their code.

5.2 Preliminary Data

We have compared **MIN** with our two new algorithms. The performance of these algorithms on the 8-queens problem is representative of their behavior on n-queens.

Figure 5.2 suggests that **BCP-Reorder** is impractical for the n-queens problem. If we were only concerned with the size of the search trees **BCP-Reorder** would be very

[2] The original source for this algorithm is a paper by Bitner and Reingold [1975].

Reordering Strategy	Search size	Assignments
Occur-Reorder	1,259 ± 1,000	32,141 ± 20,000
MIN	2,583 ± 1,000	63,789 ± 40,000

Figure 2: Performance on 5 randomly generated graph problems, each with 50 nodes, edge probability .15, and no solutions. Numbers shown are averages and standard deviations.

impressive. However, the additional truth assignments that the lookahead introduces cost far too much.

The constant-overhead version of our algorithm, however, is a practical approach to this problem. We produce better search trees and fewer truth assignments (although the improvement is slight for this problem). **Occur-Reorder** provides an improvement on the n-queens problem over **MIN** of about 2%–10% for n between 4 and 12.

We have also compared **MIN** and **Occur-Reorder** on a few randomly generated constraint satisfaction problems. We have looked at 4-coloring a random graph with 50 vertices, with a uniform probability p that there will be an edge between any pair of vertices. We have done some experiments with for $p = .15$, shown in Figure 5.2.

The improvement in average performance that **Occur-Reorder** provides seems promising, but we need to do more measurements to determine if the difference is significant.

6 Conclusions

The algorithms we have presented are based on re-ordering the choice of clauses to work on to take advantage of bounds on the size of the search remaining. One upper bound adds overhead per node that ranges from quadratic to linear. Our implementation of this bound is clearly too expensive to be practical, although the search tree that it produces is promisingly small. The other upper bound adds constant overhead, and produces an algorithm that performs well enough to be practical. We intend to continue to investigate the behavior of these algorithms, either by empirical investigations or by mathematical analysis.

6.1 Acknowledgements

We are grateful to Igor Rivin for many useful and stimulating discussions. Alan Bawden, David Chapman, Pang Chen, Johan deKleer, Jeff Siskind and Joe Weening also provided helpful comments. Jeff Shrager provided valuable office space.

Our initial implementation of these algorithms was written during the summer of 1987 at Rockwell International's Palo Alto Laboratory; we thank Michael Buckley for making this possible. Ramin Zabih is supported by a fellowship from the Fannie and John Hertz Foundation.

References

[Bitner and Reingold, 1975] Bitner, J. and Reingold, E., "Backtrack programming techniques", *Communications of the ACM* **18** (1975) 651–656.

[Cook, 1971] Cook, S., "The complexity of theorem proving procedures," *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing* (1971).

[Davis and Putnam, 1960] Davis, M. and Putnam, H., "A computing procedure for quantification theory," *Journal of the ACM* **7** (1960), 201–215.

[Haralick and Elliot, 1980] Haralick, R. and Elliot, G., "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence* **14** (1980), 263–313.

[Knuth, 1980] Knuth, D., "Estimating the efficiency of backtrack programs," *Mathematics of Computation* **29** (1975), 121–136.

[McAllester, 1987] McAllester, D., "Ontic: a representation language for mathematics," MIT AI Lab Technical Report 979, July 1987. To be published by the MIT Press.

[Montanari, 1974] Montanari, U., "Networks of constraints: fundamental properties and applications to picture processing," *Information Sciences* **7** (1974) 95–132.

[Purdom *et al.*, 1981] Purdom, P., Brown, C. and Robertson, E., "Backtracking with multi-level dynamic search rearrangement," *Acta Informatica* **15** (1981) 99–114.

[Stone and Stone, 1986] Stone, H., and Stone, J., "Efficient search techniques — an empirical study of the N-queens problem," IBM Research Report RC 12057 (#54343), 1986.