METHODS FOR SATISFYING HARD BOOLEAN FORMULAS

APPROVED: ______________________
Supervising Professor

____

____

RECOMMENDED FOR ACCEPTANCE: ______________________
Graduate Advisor

ACCEPTED: ______________________
Dean

METHODS FOR SATISFYING HARD BOOLEAN FORMULAS

by

DANIEL ANGEL JIMÉNEZ, B.S.

THESIS

Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
May, 1994

Acknowledgements

I would like to thank my thesis advisor, Hugh B. Maynard, for making me pay attention to details. I would also like to thank Neal R. Wagner for getting me interested in the Boolean formula satisfiability problem.

DANIEL ANGEL JIMÉNEZ

The University of Texas at San Antonio
May 1994

Contents

List of Figures

Chapter 1

Introduction

The satisfiability problem for formulas in the propositional calculus (SAT) is well known to be NP-complete [4], so an efficient general-purpose algorithm is unlikely. There are, however, many procedures that seem to do well in practice, at least on "random" formulas.

The research presented here explores their applicability, in terms of the median time required, to "hard" (in a well-defined sense) random formulas and formulas representing formulations of hard problems. Some of the problems chosen are conjectured not to be in P.

The methods discussed here are

1. The Davis-Putnam procedure, or DP [3]. DP was one of the first good methods for solving SAT, and is the standard to which new methods are often compared.

2. An enhancement of the Davis-Putnam procedure called BS, described in [1].

3. The GSAT procedure [9], a simple but efficient method that tries to build a satisfying assignment from a random one.

4. Several modifications to the Davis-Putnam procedure.

All of the methods will be applied to solving the restricted instance of SAT where the formulas are in conjunctive normal form. A restricted instance of the satisfiability problem in which clauses may have at most three variables is explored by generating random formulas of this kind. This restricted problem, which is also NP-complete, is called 3-CNF-SAT [4].

Each of the methods is discussed, and implementation details are given where appropriate.

Chapter 2

The Boolean Formula Satisfiability Problem (SAT)

A *Boolean variable* can take on one of two possible values, say, 0 and 1. A *Boolean formula* consists of Boolean variables connected by logical operations such as $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT, also denoted as an overline, e.g. $\overline{a}$), with the following semantics:

a	$\overline{a}$
0	1
1	0

a	b	$a \wedge b$	$a \vee b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Other operations can be derived from these. Parentheses can be used to group subexpressions where appropriate. For example, consider the following Boolean formula:

$$(x_1 \vee x_2) \wedge x_3$$

This formula is *satisfiable*, that is, we can find an assignment to the variables x_i such that the formula evaluates to 1. One such assignment is $(x_1, x_2, x_3) = (0, 1, 1)$.

Not every Boolean formula has a satisfying assignment; such an *unsatisfiable* formula is called a *contradiction*. For example,

$$(x_1 \wedge \neg x_1) \vee (x_2 \wedge x_3 \wedge \neg x_2)$$

is a contradiction.

The *Boolean formula satisfiability problem* or SAT is the problem of determining whether a given formula is satisfiable. This problem was the first problem to be shown "NP-complete" [4]. NP-complete problems are conjectured to have no polynomial time solutions; however, there are many NP-complete problems that come up in computer science, such as graph-coloring in register allocation and the travelling salesman problem. The next chapter discusses in more detail the properties and implications of this concept of hardness.

We can restrict SAT instances to those where the formula is in *conjunctive normal form* or CNF, i.e., it is an and of ors of literals (a *literal* is either a variable or its complement, so a CNF would look like e.g. $(a \vee b \vee c) \wedge (\neg a \vee c \vee d)$). There is an efficient (i.e., polynomial-time) algorithm by which we can obtain from any Boolean formula f a CNF f' such that f' is satisfiable iff f is satisfiable, and the size of f' is linear in the size of f [2]. (Note: It follows immediately that SAT for CNF formulas, or CNF-SAT, is also NP-complete.)

We can further restrict the problem to instances where each disjunction (or *clause*) has three literals. (It happens that the above mentioned algorithm yields clauses of this form.) This restricted version of SAT is called 3-CNF-SAT [2] or just 3SAT [4]. The random SAT instances used for this thesis are in 3-CNF-SAT.

Chapter 3

The Theory of NP-completeness

The Boolean formula satisfiability problem is NP-complete. It is important to understand exactly what this means, and why we would like to find efficient algorithms for SAT and why we must settle for algorithms with worst-case exponential running times.

3.1 Decision Problems

A *decision problem* can be thought of as a problem where we are asking a question about something and expect a "yes" or "no" answer. More formally, a decision problem $\Pi = (D_\Pi, Y_\Pi)$ is a set D_Π of *instances* and a subset $Y_\Pi \subseteq D_\Pi$ of instances for which the answer is "yes." We can think of this Y_Π as a language, and the problem is, "is a specific instance $I \in D_\Pi$ in Y_Π" [4]? Usually, decision problems are talked about even more formally in terms of formal languages, but this can be avoided in a general discussion by understanding that, by a set of instances, we really mean a set of computer encodings of instances.

3.2 Polynomial Time

The concepts of a polynomial time algorithm and polynomial time proof system are essential to the theory of NP-completeness. Indeed, the "P" in NP-complete stands for "polynomial."

3.2.1 Polynomial-Time Algorithms

The *time complexity function* $f_A(n)$ of an algorithm A gives the largest time requirement for the algorithm to work on any instance of size n, for all n (we can think of the time requirement as being the number of steps thru an algorithm where a single step is a fundamental operation like addition of two fixed-sized numbers). An algorithm A is said to be a *polynomial time algorithm* if it has a time complexity function in $O(p(n))$ for some polynomial p. If the time complexity function of an algorithm isn't bounded by any polynomial, then it is said to be an *exponential time algorithm*[4]. Time complexity is a statement about the worst case behavior of algorithms, since it is only a tight upper bound on the amount of time taken.

3.2.2 P

The set **P** is defined as the set of decision problems $\Pi = (D_\Pi, Y_\Pi)$ such that there exists a polynomial-time algorithm A_Π that decides if an an element of D_Π is in Y_Π.

Many decision problems are in **P**, such as the problem of deciding, for a given CNF formula f over a set of n variables $x_1 ... x_n$ and an assignment $t : \{x_1 ... x_n\} \to \{0, 1\}$, whether the assignment satisfies the formula. **P** is thought of as corresponding to our notion of tractable problems, problems we know we can solve efficiently (although $O(n^{100})$ isn't exactly efficient, most problems that we know are in **P** and we care about have small exponents).

3.2.3 NP

The set **NP** is the set of decision problems $\Pi = (D_\Pi, Y_\Pi)$ such that there exists a polynomial time *proof system* for verifying that an element of D_Π is in Y_Π.

A proof system for Y_Π is an ordered triple (f, A, C) defined as follows: C is a set such that:

1. f is a function : $D_\Pi \times C \to \{0, 1\}$ such that
   (a) $\forall_{I \in D_\Pi, c \in C}, (f(I, c) = 1) \Rightarrow I \in Y_\Pi$, and
   (b) $\forall_{I \in D_\Pi}, \exists_{c \in C}$ such that $f(I, c) = 1$
2. A is an algorithm that computes f.

An element $c \in C$ is called a "certificate" for $I \in D_\Pi$ if $f(I, c) = 1$.

A proof system (f, A, C) for a decision problem Π is called a polynomial time proof system if A has a polynomial worst-case time complexity and the size[1] of a certificate is always polynomial in the size of the corresponding element of Y_Π (i.e., for some fixed polynomial) [4].

The set **NP** gets its name from "nondeterministic polynomial time;" basically, a problem in **NP** can be solved in polynomial time by a "nondeterministic" computer. A nondeterministic computer is allowed to randomly guess a certificate that shows a particular instance $I \in D_\Pi$ of a decision Π problem is in Y_Π. It then checks the certificate with

[1]Size is usually discussed in terms of the length of an "efficient" encoding as a string from an alphabet with finitely many symbols

the polynomial time proof system. Such a a computer can *decide* instances of decision problems, instead of just proving a certificate is valid for an instance. Unfortunately, nondeterministic computers don't (seem to) exist. Note: $\mathbf{P} \subseteq \mathbf{NP}$.

3.3 Polynomial Transformations

A *polynomial transformation* for a problem Π_1 to a problem Π_2 is a function $f : D_{\Pi_1} \to D_{\Pi_2}$ such that

1. There is a polynomial time algorithm computing f.

2. For all $I \in D_{\Pi_1}, I \in Y_{\Pi_1}$ iff $f(I) \in Y_{\Pi_2}$.

If there is a polynomial transformation from Π_1 to Π_2, we write this as $\Pi_1 \propto \Pi_2$ and read this as "Π_1 transforms to Π_2."

It is immediate that if $\Pi_1 \propto \Pi_2$ then $\Pi_2 \in P$ implies $\Pi_1 \in P$ (and hence $\Pi_1 \notin P$ implies $\Pi_2 \notin P$) [4].

This means we can decide Π_1 in polynomial time if we can decide Π_2 in polynomial time.

So we can say Π_2 is at least as hard as Π_1 if $\Pi_1 \propto \Pi_2$.

We say two problems are *polynomially equivalent* if $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$ (note: this is an equivalence relation).

A problem Π is NP-complete if:

1. $\Pi \in NP$ and

2. For all other $\Pi' \in NP$, $\Pi' \propto \Pi$.

(The second condition above is called "NP-hardness" of Π.) So if Π is NP-complete, then it is, in a sense, "harder" than all the problems in **NP**. The problem is "complete" for **NP**; one can use it to solve any **NP** problem. A polynomial time algorithm for such a Π would lead to a polynomial time algorithm (via a polynomial time transformation) for any other problem in **NP**, and would show $\mathbf{P} = \mathbf{NP}$. On the other hand, a proof that an NP-complete problem has no polynomial time solution would show that all NP-complete problems have no polynomial time solutions, and that $\mathbf{P} \neq \mathbf{NP}$.

No polynomial time algorithms have ever been found for any NP-complete problems, although in practice some good heuristics and algorithms with polynomial *average* case running times (i.e., it "usually" works in polynomial time in some well-defined sense) have been found for NP-complete problems.

In 1971, S.A. Cook showed that the Boolean formula satisfiability problem is NP-complete. This thesis deals with some of the algorithms that have been used to deal with this seemingly intractable problem.

Chapter 4

Instances of CNF-SAT Used

The programs were timed on two classes of CNF-SAT formulas: random 3-CNF formulas and CNF-SAT formulations of other hard problems, such as n-queens and the clique problem. Since the GSAT procedure is not complete, i.e., it can't tell anything if a formula is unsatisfiable and may not find a satisfying assignment even for instances of SAT, we timed the algorithms only on satisfiable formulas.

4.1 Difficulty of Random Formulas

All of the random formulas used in timing the programs were instances of 3-CNF-SAT. In the past, researchers have not been very careful when choosing the distributions of SAT when testing new algorithms. The authors of [6] cite some instances where researchers have given algorithms which seem to solve SAT in time $O(n^2)$ (where n is the size of the instance). However, as the authors point out, these instances are so satisfiable that an algorithm that guessed random truth assignments was just as effective as the algorithm under consideration [6]. They suggest that special attention should be paid to the number of variables versus the number of clauses in a random formula, and that the "hardest" instances of 3-CNF-SAT (for a given number of variables) occur at the point where 50% of formulas are satisfiable. Their results concern only the performance of the Davis-Putnam algorithm, but is a more thorough study of the topic than the typical hand-waving one finds elsewhere. (The work in [1] does use the same criteria for "hardness".)

4.1.1 An Easy Distribution of Random Formulas

Let us consider an example of a distribution of SAT instances which may at first seem like a very reasonable space from which to choose instances in testing a new algorithm.

We'll increase the difficulty of the instances by increasing the number of variables. That is, we'll test a new algorithm on instances randomly chosen from the set of 3-CNF formulas with n variables, as n increases. This certainly seems like a hard test for an algorithm; the naive algorithm for finding satisfying assignment is to try every combination of assignments to variables until a satisfying assignment is found. The number of combinations of assignments for n variables is 2^n, which grows very large as n increases. We'll construct the clauses randomly, so that any literal has an equal probability of appearing in a clause. To make sure that the algorithm has to work hard, we'll have 100 clauses in each formula.

Now, one might expect the performance of the algorithm to be good at first, where there are fewer variables, and then become worse as n increases. Actually, the opposite happens (for most algorithms, including random guessing). This is because the percentage of satisfying assignments to formulas with n variables and m clauses increases as n increases. For example, if we allow only 100 clauses but, say, 400 variables, at least 100 of the variables can never be included in any clause since there are only 300 literals in the formula. So these 100 extra variables can be assigned any values without affecting the chance of finding satisfying assignments for the rest of the variables.

What prevents an assignment from satisfying a formula is that it assigns to at least three literals l_1, l_2, and l_3 zero values, where $l_1 \vee l_2 \vee l_3$ is a clause of the formula. As n increases the probability that the unlucky combination of these three appears in the formula decreases, since there is only a fixed number of clauses.

Similarly, choosing a distribution where the n is fixed and m grows leads to a large number of unsatisfiable formulas. We can regard these as "very" unsatisfiable because certain algorithms (like Davis-Putnam, which we shall see later) can easily detect contradictions when there are so many.

The point is, as a random formula with a fixed number of variables grows in the number of clauses, the number of satisfying assignments (and indeed, the probability that it will be satisfiable) decreases. As the number of variables increases, the opposite happens. So we need to take *both* of these parameters into consideration when constructing random formulas.

4.1.2 Hard Distributions of Random Formulas

The authors of [6] did extensive testing of the Davis-Putnam algorithm on many distributions of SAT, and came to some conclusions about how to choose difficult distributions for testing new SAT algorithms.

For 3-SAT formulas, they concluded that the hardest area for SAT is where 50% of the formulas are satisfiable. They found experimentally that this point occurs when the ratio of clauses to variables is between 4 and 5.

4.2 Random Formulas Used

The random formulas used were generated by a program called `random` that accepts two arguments: the number of clauses m and the number of variables n. The output is a formula in 3-CNF in n variables and m clauses where each of the $2n$ literals is equally likely to occur in any of the clauses.

For our instances, we have chosen the more common definition of 3-CNF as the language of (encodings of) Boolean conjunctive normal form formulas with *exactly* three literals, as opposed to *up to* three literals. Both problems are NP-complete; using the former definition just makes it harder for consensus-based algorithms (like BS and DP) to seem to have an initial advantage because of unit propagation [6].

It is important to note that, although formulas in 3-CNF are used, our results concern only formulas in 3-CNF-SAT, that is, we are interested only in the time it takes to find a satisfying assignment, not to determine that a formula is unsatisfiable.

4.3 CNF Formulations of Hard and NP-Hard Problems

One fascinating aspect of the theory of NP-completeness is the completeness part; there exists a polynomial time transformation from any problem in NP to any NP-complete problem. That is, if we can solve an NP-complete problem efficiently, then we can use this solution to solve any problem in NP efficiently.

We have chosen four "hard" problems to use the SAT algorithms on:

4.3.1 n-queens

The n-queens problem is the problem of placing n queens on an $n \times n$ chessboard so that no queen threatens any other queen. We represent the chessboard by a matrix C of Boolean variables. If C_{ij} is true, then there is a queen in the i^{th} column on the j^{th} row. A matrix is a solution to the n-queens problem if there are zero or one queens on each row, column, and diagonal. We can represent this in conjunctive normal form by having a rule for each pair of queens that the queens may not oppose each other. If two queens are on the same row or column, i.e., if $\exists_{i,j,k,j\neq k} : C_{ij} \wedge C_{ik}$ is true, or $\exists_{i,j,l,i\neq l} : C_{ij} \wedge C_{lj}$ is true, then the matrix C is not a solution. Similarly, if two queens are on the same diagonal, C is not a solution (two queens C_{ij} and C_{kl} are on the same diagonal if $|i-k| = |j-l|$). We must also state that there have to be n queens on the chessboard. It is sufficient to state that there is at least one queen on each row (or column). So the following CNF formula is satisfiable only by an n-queens solution:

$$\bigwedge_{i,j,k\leq n,k\neq i} (\overline{C_{ij}} \vee \overline{C_{kj}})$$

$$\bigwedge_{i,j,l\leq n,l\neq j} (\overline{C_{ij}} \vee \overline{C_{il}})$$

$$\bigwedge_{i,j,k,l\leq n,k\neq i,l\neq j,|i-k|=|j-l|} \left(\overline{C_{ij}} \vee \overline{C_{kl}}\right)$$

$$\bigwedge_{i\leq n} \left(\bigvee_{j\leq n} C_{ij}\right)$$

Clearly, one can stretch this into a formula in conjunctive normal form for a given n.

This formulation was used to test the satisfiability algorithms for values of n from 5 to 11; note, however, that the size of the formula grows as $O(n^3)$ and there exist linear time algorithms for finding n-queens solutions, so this test shows more the ability of the algorithms to deal with large numbers of clauses than their ability to find n-queens solutions. CNF-SAT in general has proven to be harder as the ratio of clauses to variables increases [6]. The n-queens problem is a good test for the algorithms, since the number of clauses is $O(n)$ times the number of variables.

4.3.2 Subgraph-Isomorphism

The subgraph-isomorphism problem asks, for two graphs G and H, whether G contains a subgraph isomorphic to H. From [4]:

SUBGRAPH ISOMORPHISM

INSTANCE: Two graphs, $G = (V_1, E_1)$ and $H = (V_2, E_2)$.

QUESTION: Does G contain a subgraph *isomorphic* to H, that is, does there exist a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|, |E| = |E_2|$, and there exists a 1-1 function $f : V_2 \to V$ satisfying $\{u, v\} \in E_2$ iff $\{f(u), f(v)\} \in E$?

Garey and Johnson go on to prove the problem NP-complete. A SAT formulation for this problem appears in [11] in a language under development for programming in propositional logic (H. Stamm-Wilbrandt, posted an article describing this formulation to the Usenet `comp.theory` newsgroup almost the same day this author was looking for such formulation). Given two graphs (as Boolean adjacency matrices), we can use this formulation and techniques for converting from this language into pure propositional logic [10], i.e., an instance of SAT. We can do some more algebraic manipulations to get the formula into conjunctive normal form and use our CNF-SAT algorithms to solve the problem.

Before the formula, some explanation of Stamm-Wilbrandt's (also the author of [10]) language is in order. He uses the notation *at_most_one* and *at_least_one* to mean that at most (or at least, respectively) one of the supplied Boolean variables is true. For example, *at_most_one(*a, b, c*)* is true iff zero or one of a, b, and c is true. The notation *exactly_one* means *at_most_one* $\wedge$ *at_least_one*. He gives the the following "naive" versions of these before going into detail about special cases and efficient implementation:

$$at\_most\_one\{l_1, \ldots, l_x\} := \bigwedge_{1 \leq i < j \leq x} (\overline{l_i} \vee \overline{l_j})$$

$$at\_least\_one\{l_1, \ldots, l_x\} := (l_1 \vee l_2 \vee \ldots \vee l_x)$$

Here is Stamm-Wilbrandt's formulation in his language `pipl` of the subgraph-isomorphism problem:

```
// [GT48] Subgraph Isomorphism (subgraph_isomorphism.pipl)
//
// Instance:  Graphs G=(V_G,E_G), H=(V_H,E_H)
// Question:  Does G contain a subgraph isomorphic to H, i.e.,
//            a subset V of V_G and a subset E of E_G such that
//            |V|=|V_H| and |E|=|E_H|, and there exists a
//            one-to-one function f:V_H-->V satisfying {u,v}
//            in E_H if and only if {f(u),f(v)} in E ?
//
input

ugraph G;
ugraph H;

conditions
```

```
|V(G)| >= |V(H)|
|E(G)| >= |E(H)|

variables

{ (v,x) | v in V(G), x in V(H) }

output

{ (v,x) | v in V(G), x in V(H), #(v,x) }

formula

{ exactly_one{ (v,x) | v in V(G) } | x in V(H) }
{ at_most_one{ (v,x) | x in V(H) } | v in V(G) }

{ at_most_one{ (v,x), (w,y) } |
   {v,w} in E(G), {x,y} in ((V(H)*V(H))-E(H)) }
{ at_most_one{ (v,x), (w,y) } |
   {v,w} in ((V(G)*V(G))-E(G)), {x,y} in E(H) }
```

Here is a "compilation" of the program into conjunctive normal form (using Garey and Johnson's slightly different notation for the graphs):

$$\bigwedge_{x \in V_2} \mathit{exactly\_one}\{(v,x) \mid v \in V_1\}$$

$$\bigwedge_{v \in V_1} \mathit{at\_most\_one}\{(v,x) \mid x \in V_2\}$$

$$\bigwedge_{\{v,w\}\in E_1,\{x,y\}\in \overline{E_2}} at\_most\_one\{(v,x) \mid x \in V_2\}$$

$$\bigwedge_{\{v,w\}\in \overline{E_1},\{x,y\}\in E_2} at\_most\_one\{(v,x) \mid x \in V_2\}$$

which yields

$$\bigwedge_{x\in V_2} \left[\left(\bigvee_{v\in V_1} (v,x) \right) \wedge \left(\bigwedge_{v_i,v_j\in V_1} \left(\overline{(v_i,x)} \vee \overline{(v_j,x)} \right) \right) \right]$$

$$\bigwedge_{v\in V_1} \left[\bigwedge_{x_i,x_j\in V_2} \left(\overline{(v,x_i)} \vee \overline{(v,x_j)} \right) \right]$$

$$\bigwedge_{\{v,w\}\in E_1,\{x,y\}\in \overline{E_2}} \left[\overline{(v,x)} \vee \overline{(w,y)} \right]$$

$$\bigwedge_{\{v,w\}\in \overline{E_1},\{x,y\}\in E_2} \left[\overline{(v,x)} \vee \overline{(w,y)} \right]$$

which finally yields

$$\bigwedge_{x\in V_2} \bigvee_{v\in V_1} (v,x)$$

$$\bigwedge_{x\in V_2} \left[\bigwedge_{v_i,v_j\in V_1} \left(\overline{(v_i,x)} \vee \overline{(v_j,x)} \right) \right]$$

$$\bigwedge_{v\in V_1} \left[\bigwedge_{x_i,x_j\in V_2} \left(\overline{(v,x_i)} \vee \overline{(v,x_j)} \right) \right]$$

$$\bigwedge_{\{v,w\}\in E_1,\{x,y\}\in \overline{E_2}} \left[\overline{(v,x)} \vee \overline{(w,y)} \right]$$

$$\bigwedge_{\{v,w\}\in \overline{E_1},\{x,y\}\in E_2} \left[\overline{(v,x)} \vee \overline{(w,y)} \right]$$

The resulting formula has a size in $O(|E_1||E_2| + |V_1||V_2|^2)$.

To generate random instances of subgraph-isomorphism, we wrote a program called `SUBGRAPH-ISOMORPHISM`. This program accepts four parameters: the numbers of vertices G_n and H_n and numbers of edges G_e and H_e in both graphs. It prints an adjacency list representation for two random graphs with the given number of edges. The output is piped through another program called `subgraph` that generates the CNF formula that is satisfiable only by an isomorphism function between a subgraph of G and H. The `SUBGRAPH-ISOMORPHISM` program makes sure that there is a subgraph-isomorphism by choosing the second graph to be a random re-labelling of the first H_n vertices and edges in the first graph. This in no way diminishes the hardness of actually finding the subgraph-isomorphism, since the satisfiability programs don't have access to the random permutation.

4.3.3 Clique

The clique problem is another NP-complete problem from [4]. The problem asks, for a given graph G and integer $J > 0$, whether the complete J-vertex graph is a subgraph of G. From Garey and Johnson:

CLIQUE
INSTANCE: A graph $G = (V, E)$ and a positive integer $J \leq |V|$.
QUESTION: Does G contain a *clique* of size J or more, that is, a subset $V' \subseteq V$ such that $|V'| \geq J$ and every two vertices in V' are joined by an edge in E?

This problem is also NP-complete; Garey and Johnson prove this with a polynomial transformation from the vertex-cover problem; another proof using CNF-SAT is given in [2].

The formulation of the clique problem in CNF-SAT is clear; we simply use the formula for subgraph-isomorphism on G and the complete graph with J vertices. The completeness of the second graph simplifies the formula, so we get:

$$\bigwedge_{x \le J} \bigvee_{v \in V} (v, x)$$

$$\bigwedge_{x \le J} \left[\bigwedge_{v_i, v_j \in V} \left(\overline{(v_i, x)} \vee \overline{(v_j, x)} \right) \right]$$

$$\bigwedge_{v \in V} \left[\bigwedge_{x_i, x_j \le J} \left(\overline{(v, x_i)} \vee \overline{(v, x_j)} \right) \right]$$

$$\bigwedge_{\{v,w\} \in \overline{E}, x, y \le J} \left[\overline{(v, x)} \vee \overline{(w, y)} \right]$$

We use a program called `CLIQUE` that accepts three parameters: the number of vertices and edges in the graph and J. `CLIQUE` generates an edge list for a corresponding random graph, and an edge list for the J-vertex complete graph. The output of `CLIQUE` is fed into the `subgraph` program to get an instance of CNF-SAT satisfiable only by an isomorphism function mapping a subgraph of V onto the complete J-vertex graph (the simplification of the formula is done automatically).

4.3.4 Graph Isomorphism

The graph isomorphism problem asks, for graphs G and G', whether they are isomorphic in the sense of the subgraph-isomorphism problem. Graph-isomorphism is conjectured to be in NPI, the "NP-incomplete" languages in NP that are neither NP-complete nor in P [4] (note: the existence of any element of NPI is still a conjecture). From Garey and Johnson:

GRAPH ISOMORPHISM

INSTANCE: Graphs $G = (V, E)$, $G' = (V, E')$.

QUESTION: Are G and G' *isomorphic*, that is, is there a 1-1 function $f : V \to V$ such that $\{u, v\} \in E$ iff $\{f(u), f(v)\} \in E'$?

Clearly, graph isomorphism is a special case of subgraph isomorphism where $|V_1| = |V_2|$. The subgraph isomorphism formulation for this problem was used. This problem is nicer than the more general subgraph-isomorphism problem because it reduces the dimensions of the problem; the only parameters are the graph size and number of edges. The graphs, as before, are guaranteed to have an isomorphism function; it is up to the algorithms to find it.

Chapter 5

BS: A Consensus Method

The BS method solves SAT by a combination of replacing the formula with an equivalent but smaller formula and an implicit search for a satisfying assignment. Along the way, a contradiction may be found that immediately implies the formula is not satisfiable. We present the algorithm given in [1].

5.1 Some Definitions and Conventions

It is important to note that the problem of Boolean formula satisfiability has two incarnations, the more common is that of trying to determine the satisfiability of an equation $f(x) = 1$ where f is given in CNF. The work in [1] uses the dual definition of SAT as the problem of determining the satisfiability of an equation $f(x) = 0$ where f is given in *disjunctive normal form* (DNF), i.e., as a disjunction of conjunctions of literals.

It's easy to see that the two are polynomially equivalent; given an equation $f(x) = 1$ with f in CNF, we can transform the problem by applying DeMorgan's laws (that is, $\overline{p \wedge q} \Leftrightarrow \overline{p} \vee \overline{q}$ and $\overline{p \vee q} \Leftrightarrow \overline{p} \wedge \overline{q}$) and then look at the problem as an equation $f'(x) = 0$

with f' being the new formula in DNF. This type of SAT problem is the problem of determining whether a formula in DNF is a *tautology* (i.e., equal to the constant 1), instead of a contradiction.

Here are some definitions we need for the algorithm:
First, we define the *consensus* of two clauses: Let $\mu = \alpha\beta y_i$ and $\nu = \gamma\beta\overline{y_i}$ be two clauses in the Boolean formula f such that only one variable x_i of f can be substituted for either y_i or $\overline{y_i}$, i.e., y_i has the unique property that it is complemented in μ but not in ν. Note: not all pairs of clauses of f may have this property. α denotes a set of literals that are in μ but not in ν, γ is the set of literals in ν but not μ, and β is the set of literals common to both. We define the consensus of μ and ν as $\alpha\beta\gamma$, and write this consensus operation as $\mu * \nu$. Note: $x_i * \overline{x_i}$ is the empty clause: a tautology.

Let us also define the partial order $\leq$ on Boolean functions. Given two Boolean functions $f(x)$ and $g(x)$, $f(x) \leq g(x)$ iff $\forall_{x \in \{0,1\}^n}, f(x) = 1 \Rightarrow g(x) = 1$. So $f(x)$ is less than $g(x)$ if the set of satisfying assignments to f is a subset of the satisfying assignments to g go to 1. Let's look at some properties of this partial order:

1. If $f \leq g$ and $f = 1$ then $g = 1$.

2. If $f \leq g$ then $f \vee g = g$.

3. For all f, $f \leq 1$.

4. We can have two functions f and g such that $f \not\leq g$ and $g \not\leq f$.

5. If α and β are clauses in f and $\{\alpha\} \leq \{\beta\}$, then f is equivalent to $f - \{\alpha\}$, i.e., α is redundant and can be thrown out.

A clause μ is called *prime* for a function f if $\mu \leq f$ and there is no $\nu \leq f$ such that $\mu \leq v$. So a prime clause of f is in a sense a maximal clause, corresponding to a largest group of 1's in a Karnaugh map (a device for deriving minimal Boolean DNF's from truth tables[5]). A set of clauses is a *basis* for f if their logical OR equal to f; such a set is a *prime basis* if it contains only prime clauses. Note that f is unsatisfiable (in the sense of [1], i.e., a tautology) iff $\{1\}$ is the prime basis for f.

5.2 The BS Algorithm

The algorithm in [1] called BS strikes a balance between computing the entire prime basis of f (an inefficient approach) and a pure implicit binary tree search of the formula for satisfying assignments. This technique is used in the Davis-Putnam algorithm, but DP uses only unit consensus, that is, consensus in which μ is a literal. The BS algorithm uses a different criterion for choosing which consensuses to take, as we shall see:

procedure BS

Input: a Boolean function $f(x)$ as a DNF formula.

Output: "unsatisfiable" or an assignment x such that $f(x) = 0$.

begin

  call algorithm_one;

  if 1 is a monomial of f then return "unsatisfiable".

  call algorithm_two;

  if 1 is a monomial of f then return "unsatisfiable".

  if $f = y_1 \vee y_2 \vee ... \vee y_n$ is a disjunction of literals

  then return an assignment setting each literal to whatever value will cause it to evaluate to 0.

  choose a variable x_i (according to a method described later)

  call BS recursively on $f \vee \{x_i\}$

and $f \vee \{\overline{x_i}\}$.

if either returns a satisfying assignment, **return** it with the appropriate value of x_i else return "unsatisfiable."

end

Algorithm_one

Input: f as a DNF formula.

Output: f as another DNF formula.

begin

while there exist two clauses μ and ν of f

such that $\alpha = \mu * \nu$ is not less than a

clause of f and the degree (number of literals) of

α is less than $d(\mu, \nu) =$

max(degree of μ, degree of ν),

do

supress from f all the clauses less than α;

let $f := f \vee \alpha$.

end while

end

Algorithm_two

Input: f as a DNF formula.

Output: f as another DNF formula.

begin

while 1 is not a clause in f and there exists a

literal that is not (by itself) a clause of f,

do choose the literal y_i that is not a clause of f

and that appears in the fewest terms of f;

$f := f \vee \overline{y_i}$;

apply algorithm_one to f, fixing $d(\mu, \nu) = 2$

for all μ, ν.

end while

end [1]

The recursive branching done in the BS procedure is made on a variable x_k such that

$$M \cdot \min(v_k, w_k) + v_k + w_k$$

$$= \max_{i=1}^{n} \left[M \cdot \min(v_i, w_i) + v_i + w_i\right]$$

where v_i (resp. w_i) is the number of three-literal clauses of f that contain x_i (resp. $\overline{x_i}$) and M is a large constant that can be adjusted.

5.3 The Implementation

The C implementation of the BS algorithm is given in Appendix A. The main issues involved were choosing compact and easily accessed representations for clauses and formulas. We chose clauses to be linked lists of literals, ordered by index. Formulas are represented as arrays of clauses. The function `exist_mono()`, implementing the "while there exists a monomial..." part of algorithm one, uses a special data structure for indexing formulas by literals, so it only considers possible consensuses (the authors of [1] mention a similar technique used for their implementation, but with very few details). For the instances of SAT considered, the implementation of BS is efficient.

It is important to note that the implementation differs slightly from the algorithms given above. Specifically, the authors of [1] fix the d function to 2 for all but the first iteration of algorithm one. Their reason is that many three variable monomials are produced in the later iterations that don't contribute to the algorithm. Note also that their implementation of the BS algorithm uses a more elaborate data structure than the one given here: they use linked lists for the cubic monomials and arrays for the quadratic and linear terms, while the implementation here uses linked lists for everything. We chose to do it this way so that there would be no artificial improvement of BS over DP (or vice-versa); indeed, they both use exactly the same C code modulo a few `#ifdef`'s.

Chapter 6

The Davis-Putnam procedure

Long before the theory of NP-completeness was introduced, people still had a need to find satisfying assignments to formulas in propositional logic. The standard algorithm against which most other methods are judged is the Davis-Putnam procedure, introduced in [3] in 1960. Davis-Putnam, or DP, is a general-purpose method for solving SAT[1]. If a formula is satisfiable, DP will find a satisfying assignment, and if a formula is not satisfiable, DP will report the formula is a contradiction. DP does well on many classes of clauses, but can show exponential behavior in terms of both time and storage on the harder instances of SAT.

We will use the version of DP given in [1]. Their algorithm is an "enhancement" to DP, but can be modified to only do the Davis-Putnam procedure. This fact led to an easy job coding up both the algorithm from [1] and DP in the same C program using `#ifdef` statements.

[1]For this chapter, we will continue using the DNF version of SAT, since the DP implementation comes from the BS implementation

Basically, DP consists of four rules [6]:

- If the formula f is empty, return "satisfiable."
- If the formula f contains an empty clause, return "unsatisfiable."
- (Unit-Clause rule) If f contains a unit clause, i.e., a clause that is a single literal, assign to the corresponding variable the value that will satisfy that literal.
- (Splitting Rule) Select a variable v that hasn't been assignned a truth value. Give it a value and call DP on the resulting formula. If this call returns "satisfiable," then return "satisfiable" else set the variable to the other value and return the result of calling DP on the resulting formula.

We will present more explicitly the DP algorithm using the same terminology as the BS algorithm (e.g., consensus, less than, etc). The DP algorithm follows:

procedure DP

Input: a Boolean function $f(x)$ as a DNF formula.

Output: "unsatisfiable" or an assignment x such that $f(x) = 0$.

begin

  while there exists a unit clause μ and a clause ν in f

    such that $\alpha = \mu * \nu$ is not less than a

    clause of f and the degree (number of literals) of

    α is less than the degree of ν,

  do

    supress from f all the clauses less than α;

    let $f := f \vee \alpha$.

  end while

if 1 is a monomial of f **then return** "unsatisfiable".

if $f = y_1 \vee y_2 \vee ... \vee y_n$ is a disjunction of literals

then return an assignment setting each literal to whatever value will

cause it to evaluate to 0.

choose a variable x_i;

call DP recursively on $f \vee \{x_i\}$

and $f \vee \{\overline{x_i}\}$.

if either returns a satisfying assignment, **return** it with the

appropriate value of x_i else return "unsatisfiable."

end

6.1 The Implementation

As noted, the DP algorithm is very similar to the BS algorithm; the same C program with `#ifdef` statements was used for both. The main difference is that consensus operations are restricted to considering monomials μ of degree one.

Chapter 7

The Davis-Putnam Variants

Three original enhancements to the Davis-Putnam algorithm were also considered and coded. They and their implementations are discussed in this chapter.

7.1 D1: Davis-Putnam Variant One

The first and second variants deal with the splitting rule of Davis-Putnam. When the DP algorithm splits, it assigns 0 to a variable, calls itself recursively on the resulting formula, then, if unsuccessful, does the same thing for 1. The D1 and D2 algorithms try to make an "educated guess" whether to assign 0 or 1 first. Clearly, if we could ask an NP oracle which way to go, we could find a satisfying assignment or contradiction in linear time.

The D1 variant uses a heuristic method to choose the first value to assign variable v in an n-clause formula f before invoking D1 recursively. It chooses whichever value of v will maximize the number of satisfied clauses in f. For example, if the clause $x_1 \wedge x_2 \wedge x_3$ is in f and we need to decide what value to assign x_1 first, we will notice that assigning 0 to x_1 satisfies this clause (satisfiaction still in the sense of [1]). If it satisfies more clauses than assigning 1 to x_1, we choose it; otherwise we choose 1.

7.2 D2: Davis-Putnam Variant Two

The D2 variant tries to "learn" what works best with a particular formula f. The search tree generated by splitting is often very wide, and the decision whether to choose 0 or 1 is usually made many times for the same variable. D2 keeps two arrays of integers (initially zeroes) `right[`n`]` and `wrong[`n`]`, where n is the number of variables in the formula. When the choosing zero is "right" for the variable v, i.e., choosing it leads to finding a satisfying assignment to some sub-formula, we increment `right[`v`]`. If it leads to finding a contradiction (something we don't want in general), we increment `wrong[`v`]`.

When it is time to choose a value for a variable v, we look at the arrays. If `wrong[`v`]` $>$ `right[`v`]`, then we choose 1, since in the past 0 has led to more "wrong" subformulas than 1. Otherwise, we choose 0.

7.3 D3: Davis-Putnam Variant Three

This is the version of Davis-Putnam that is found in [8]. The D3 variant tries to simplify subformulas before they are considered by the rest of the Davis-Putnam algorithm. It uses the "pure-literal" rule mentioned also in [7] on page 433 as an exercise. A literal l in a formula f is said to be *pure* if it is either a *positive* or *negative* literal. A positive literal is one that appears only uncomplemented in the formula, e.g., x_1. A negative literal is one that appears only complemented in the formula, e.g., $\neg x_1$.

It can be shown that pure literals can be immediately replaced by the values that satisfy them, i.e., 0 for negative and 1 for positive literals.

Algorithm D3 inserts all pure literals of a formula as unit clauses before each recursive call of Davis-Putnam (unless the literal was already there as a unit clause). This immediately procduces (hopefully) many new unit literals that the rest of the Davis-Putnam procedure can use for consensus and unit propagation.

Chapter 8

The GSAT Method

8.1 The Method

The GSAT procedure [9] is a randomized method for finding satisfying assignments to Boolean CNFs. It uses a greedy strategy to satisfy as many as possible of the clauses in the formula. If all are satisfied, then the formula is satisfied and the resulting assignment is returned. The GSAT procedure is not complete, because it may fail to find a satisfying assignment to a satisfiable formula. It will also give no information in the case that the formula is not satisfiable. However, for the situations in which testing for satisfiability is the main problem, we are often presented two formulas representing two different possible models in a larger problem. We can use GSAT on both, knowing that exactly one of them is satisfiable. Only if GSAT fails on both do we need to resort to other, slower methods.

The GSAT procedure is quite simple:

procedure GSAT

Input: a set of clauses α, MAX-FLIPS, and MAX-TRIES
Output: a satisfying truth assignment of α, if found
begin
  for $i := 1$ **to** MAX-TRIES
    $T :=$ a randomly generated truth assignment
    for $j := 1$ **to** MAX-FLIPS
      if T satisfies α **then return** T
      $p :=$ a propositional variable such that a change
        in its truth assignment gives the largest
        increase in the total number of clauses
        of α that are satisfied by T
      $T := T$ with the truth assignment of p reversed
    end for
  end for
  return "no satisfying assignment found"
end [9]

8.2 The Implementation

The author's C implementation of GSAT can be found in Appendix B. The implementation is straigtforward; an array of C integers is used to hold T. Note that the same library is used to do standard clause operations as in the other programs. The GSAT program in Appendix B can be compiled using a `#define` statement to implement a completely random strategy for guessing satisfying assignments; this program was used along with the other three methods to make sure the algorithms were an improvement over completely random guessing. This may seem extreme, but it is important to make sure the instances of SAT chosen are “really” hard, that they can’t be satisfied simply by guessing.

Chapter 9

Computational Results

The algorithms were timed on several instances of CNF-SAT using a Sun-4 running Solaris 2.2 with two SuperSPARC 50 MHz processors. The computer had 256MB of main memory. Several scripts and programs were used to facilitate the timing of the programs. The Unix `clock(2)` system call was initially used; however, under Solaris 2.2 the system call is not reliable after 36 minutes because of an overflow in the 32-bit arithmetic. When the programs started exceeding this bound, the scripts were switched to use the `time(1)` command.

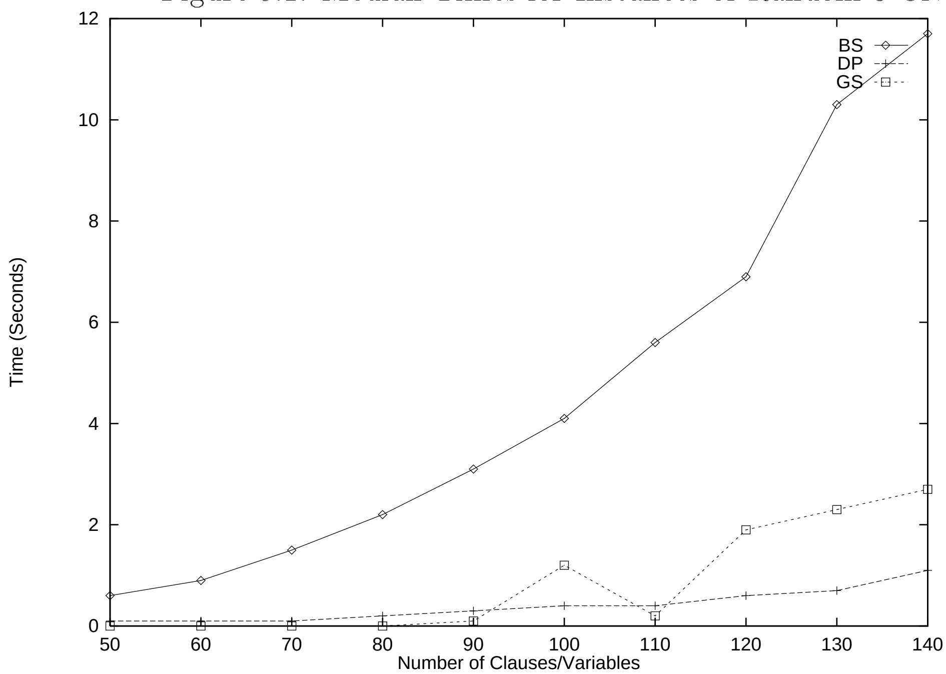
In many cases, multiple instances of the same size were tried to get more information about the performance of the algorithms. Since GSAT is a randomized algorithm, the performance may vary from one run to another even for the same instance. Also, all of the randomly generated instances (the graph and random 3-CNF-SAT problems) were tried several times. In each case, the median sample is used to measure the performance in this chapter. Appendix D contains a summary of all the information gathered in the experiments; for each problem and algorithm, the performance of the implementation is given in terms of the median, mean, and mode of the time taken for each set of instances considered.

Note: Originally, the BS algorithm was coded in a way that made it seem to perform horribly. When we saw how the authors of [1] suggested they coded theirs, a new version was written. However, we had to discard many of the data that were gathered before then in order to test BS on the same (random) instances of problems as for DP and GSAT. This recoding also affected the coding of DP, but the performance was neither improved or degraded.

9.1 Performance on Random 3-CNF-SAT Instances

Figure 9.1 shows a graph of the performance of DP, GSAT, and the BS algorithm on

Figure 9.1: Median Times for Instances of Random 3-CNF-SAT



instances of 3-CNF-SAT where the number of clauses equals the number of variables.

These results were obtained by taking the median time for each algorithm on sixteen different random formulas for each number of clauses/variables in the range 50 to 140, in increments of 10. All of the algorithms were tested on the same sets of clauses. Note: since GSAT can't determine unsatisfiability, the only times reported are those for which the random formulas were found to be satisfiable by DP or BS. If a formula was found to be unsatisfiable at any point, the script simply deleted it so the other programs wouldn't have to deal with it.

In this range, DP seems to do the best, although the somewhat erratic growth of the GSAT curves suggests that it could go either way. The BS algorithm does poorly in this here.

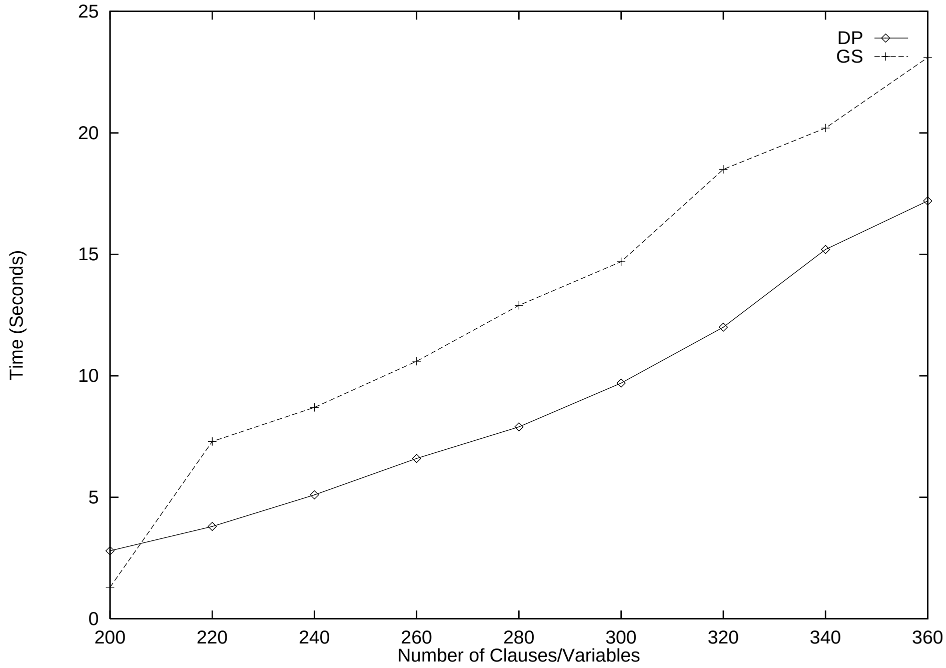
Figure 9.2 shows a graph of just DP versus GSAT for very large instances of random 3-CNF-SAT. Although the performance of DP has a much more predictable performance, DP and GSAT seem to perform comparably in this range, separated by an additive constant we can attribute to differences in implementations.

Performance measures for the DP variants are in Appendix D; a graph reveals only that D3 does poorly here.

9.2 Performance on the n-queens problem

The programs were run on SAT formulations of the n-queens problem for $n = 4$ up to 10. These formulations, if we use the ratio of clauses to variables as a measure of difficulty, are quite formidable. The 10-queens formulation, for instance, has 1480 clauses and only 100 variables. Figure 9.3 shows a plot of the performance of GSAT, DP, and BS on these instances. The DP algorithm far outperforms GSAT and BS. Data where $n = 12$ is not

Figure 9.2: Median Times of GSAT and DP for Large Instances of Random 3-CNF-SAT

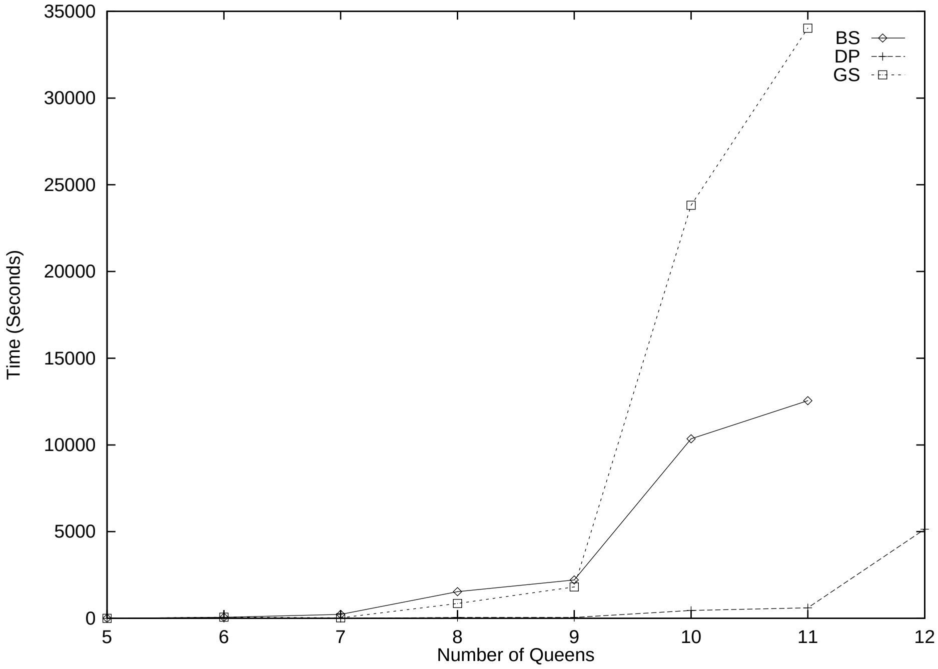


available for GSAT or BS because the programs simply ran too long.

Note: at this point, the reader may notice that the figures in this thesis show a spike at the last few data points; this is the point at which we stopped gathering data, because after this point, the programs began to take far too long. GSAT, remember, is a randomized algorithm and thus requires several runs to get a good median. This spike appears in the other formulations as well; it should be kept in mind that the number of clauses and variables for these problems grow as > 1 degree polynomials. For the n-queens problem, the number of clauses grows as $O(n^3)$, while there are only $O(n^2)$ variables. Remember, satisfiability (in general) has been experimentally shown to be harder as ratio of the number of clauses to the number of variables increases [6].

Figure 9.4 shows the performance of the DP variants on the n-queens problem. In the

Figure 9.3: Median Times for n-queens Formulations

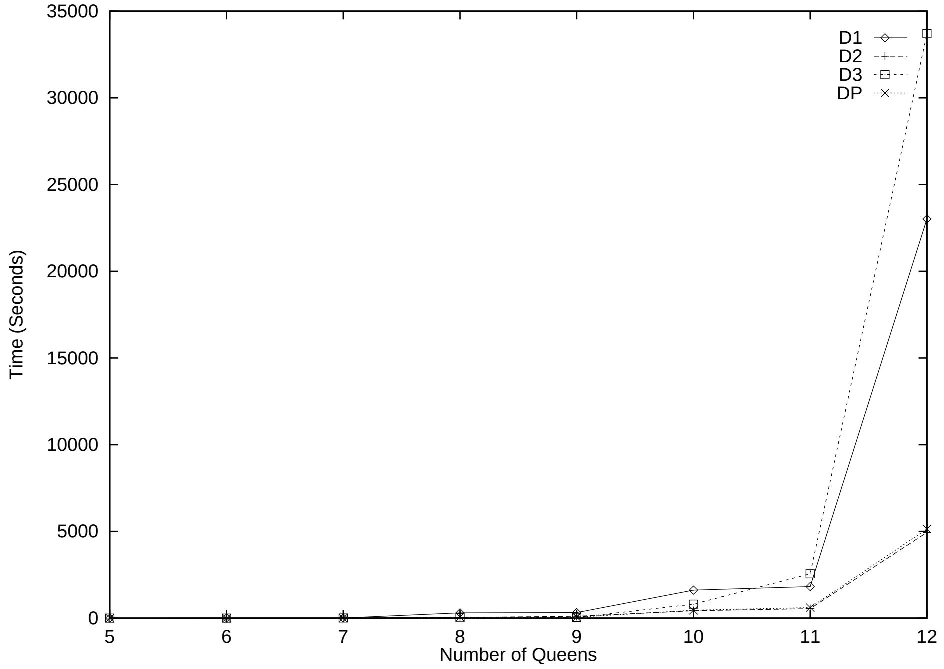


whole graph D1 and DP seem to do equally well while D2 and D3 perform a little worse. At the "spike" point of twelve queens, D2 and D3 do a lot worse, while D1 and DP deal well with the difficulty.

9.3 Performance on the Graph-Isomorphism Formulations

For the graph isomorphism problem, we generated random graphs with equal numbers of vertices and edges. The `GRAPH-ISOMORPHISM` program made sure that there was actually an isomorphism simply by having the second graph be a random relabelling of the first graph. Figure 9.5 shows the performance of the three programs on this formulation, with numbers of vertices in the range 5 to 9. Before 7 vertices, all the algorithms found satisfying isomorphisms very easily; after this points, each algorithm spiked in the time needed.

Figure 9.4: Median Times for n-queens Formulations, DP Variants



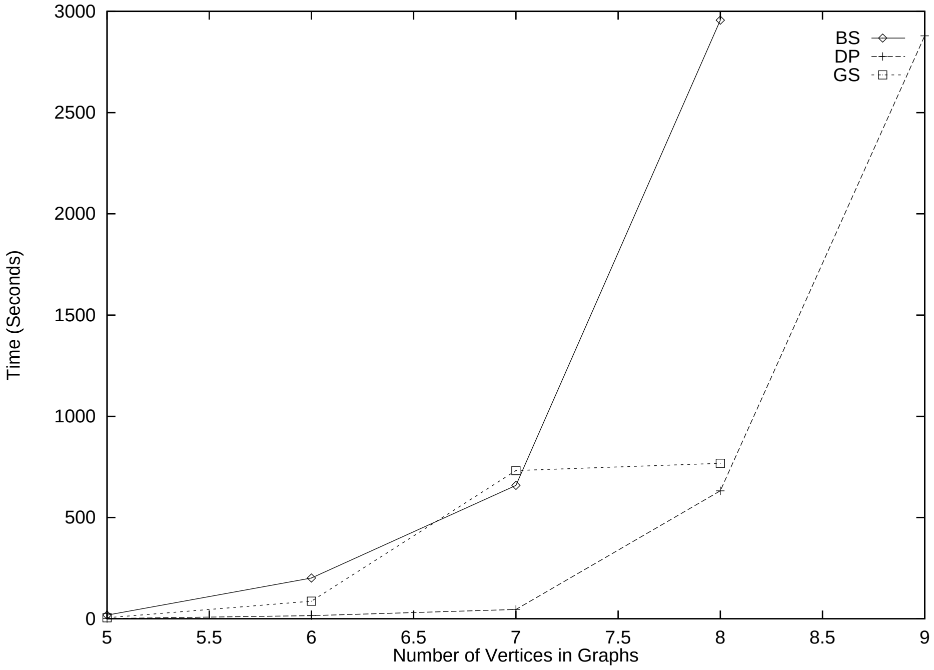
Some data points are missing; this is where the programs could not finish because they were taking too much time. Again, DP performs the best. GSAT seems hopeful, until one sees that we couldn't get data for nine vertices because of the time needed.

Figure 9.6 shows the performance of the DP variants on the graph- isomorphism problem. In this case, algorithm D1 performs the best by far, working much more quickly than the others.

9.4 Performance on the Subgraph-Isomorphism Formulations

The programs were run on subgraph-isomorphism instances where the larger graph had ten vertices and the smaller graph had from two to ten vertices and the same number of

Figure 9.5: Median Times for Graph-Isomorphism Formulations

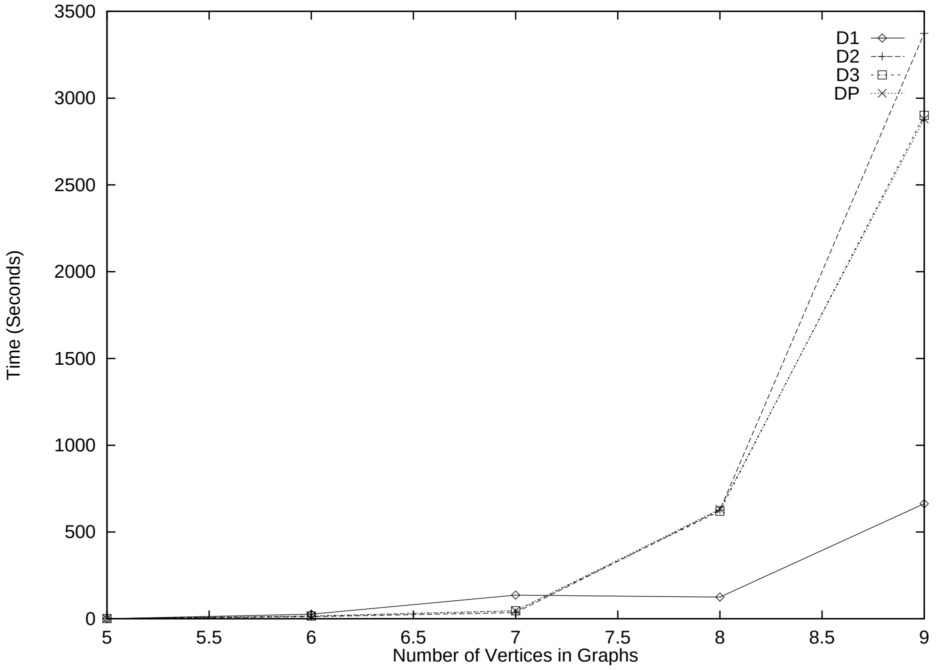


edges. Again, an isomorphism was guaranteed, this time by having the smaller graph be just a relabelling of a subset of the vertices from the larger graph, with the appropriate edges included.

Figure 9.7 shows the performance of GSAT, DP, and BS on these problems. The BS algorithm does the best for this problem, although this conclusion is only strongly substantiated by the last data point. GSAT and DP do about the same. For these data, CPU time was a major consideration, so only a few samples were taken for each point (see Appendix D for the exact numbers).

We did not test the DP variants on this problem due to time constraints, both human and computer.

Figure 9.6: Median Times of DP variants for Graph-Isomorphism Formulations



9.5 Performance on the Clique Formulations

The programs were run on instances of the clique problem where the graphs went from five to twelve vertices. In each case, the formulation specified a clique of a size half the number of vertices. Figure 9.8 shows a graph of the performance of GSAT, DP, and BS on these formulations. Clearly, GSAT performs the best, followed (surprisingly not closely) by DP and BS.

Figure 9.9 shows the performance the the DP variants on the clique formulations. In this case, the D1 variant performs much better, while the others do about the same.

Figure 9.7: Median Times for Subgraph-Isomorphism Formulations

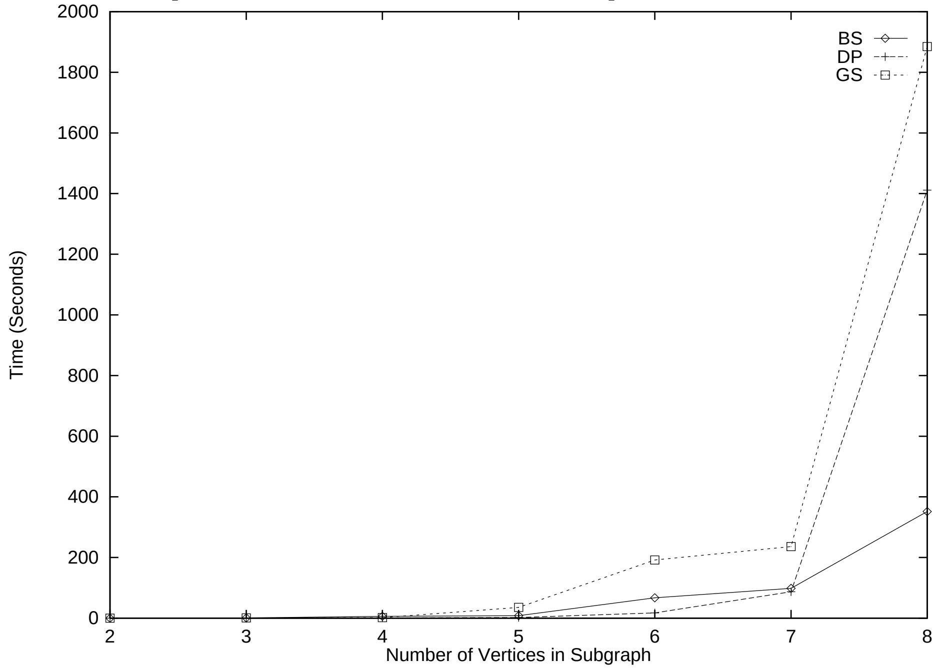


Figure 9.8: Median Times for Clique Formulations

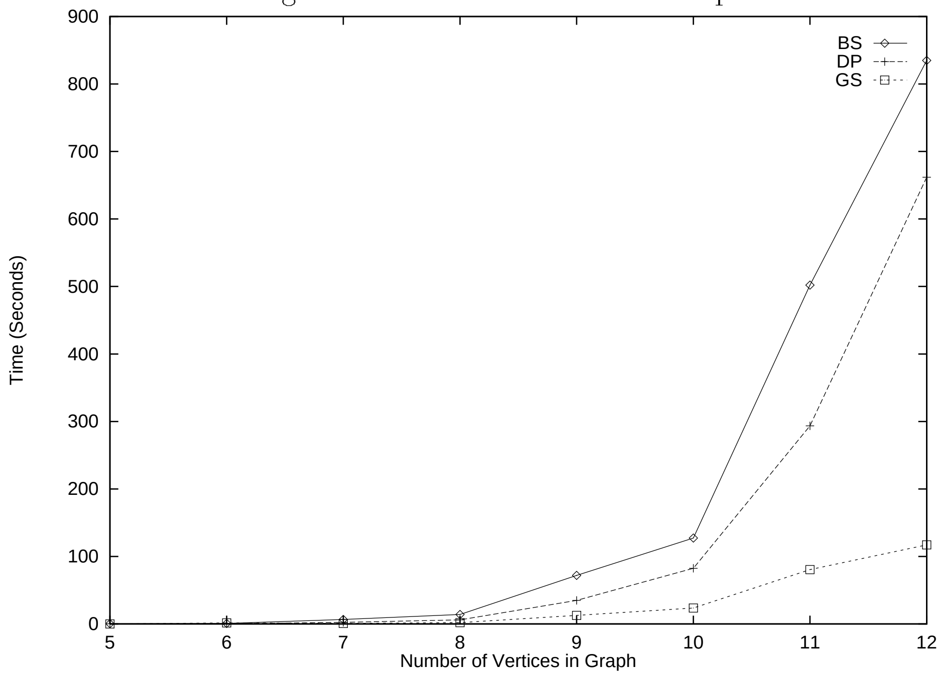
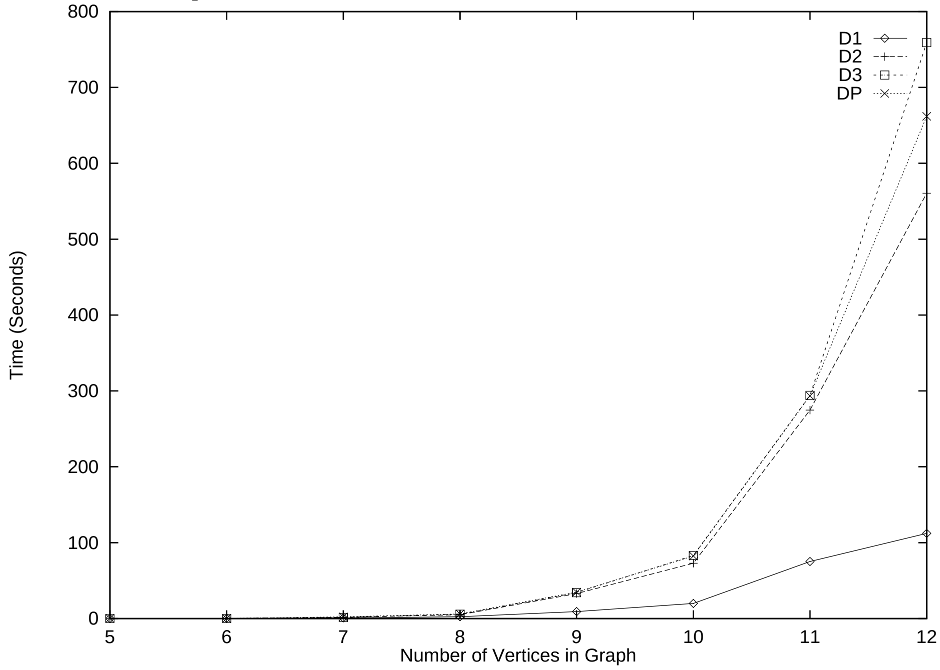


Figure 9.9: Median Times for Clique Formulations, DP Variants



Chapter 10

Conclusions

From the computational results, the D1 variant of DP seems to perform the best, followed by DP (and sometimes the other variants), then GSAT and BS. Although only limited instances of SAT were chosen, we feel that the D1 variant is the best choice for most satisfiability problems except perhaps in special cases where the others might fare better (for example, if one knew that a problem has an optimal greedy algorithm, one might suspect that GSAT would be a good candidate for solving a SAT formulation of this problem; in this case, one is probably better off using the optimal algorithm). The GSAT algorithm did perform very well on the clique problem, while BS worked well on the subgraph-isomorphism problem.

From the code in the appendices, we can see that the implementation of GSAT is efficient and does not rely too heavily on the clause library routines (which performs linear searches, for example). One might expect BS to outperform DP since the program for both is constructed in the context of the BS algorithm (from the first description of DP, it's clear that it can be implemented in many different ways). However, this is clearly not the case. It is this author's opinion that BS spends too much time finding consensuses between clauses that don't really contribute to finding a satisfying assignment. DP only performs the consensuses (unit consensus) where it is profitable and is thus a more efficient algorithm.

Although the D2 and D3 variant usually did about the same and in some cases worse than the DP algorithm the D1 variant did very well on the graph-isomorphism and clique problems, and worked equally as well as DP (as opposed to the other two) for the n-queens problem. We feel D1 is a significant improvement to the Davis-Putnam procedure, and deserves further research.

Appendix A

The BS and DP program

This is the program that implements the BS and DP algorithms. It was written by the author of the thesis.

bsdp.c:

```
/*
 *   bsdp.c
 *
 *   This program is an implementation of the Davis-Putnam (DP)
 *   algorithm first described in Davis. M., and Putnam, H.,
 *   (1960) A Computing Procedure for Quantification Theory,
 *   Journal of the ACM, 1960, Vol 7, 201-215, and the consensus-
 *   based algorithm (an enhancement to DP) given in Billionnet,
 *   A. and Sutter, A. (1992).  An efficient algorithm for the
 *   3-satisfiability problem, Operations Research Letters vol.
 *   12, July  1992, 29-36.
 *
 *   Each algorithm ends up searching the formula as an implicit
 *   binary tree for satisfing assignments.
 *
 *   DP uses the following rules before embarking
 *   on the search:
 *   1.  If the formula is empty, return "satisfiable."
 *   2.  If the formula contains an empty clause, return
 *       "unsatisfiable."
```

```
*   3.  (Unit-Clause rule) If the formula  contains a unit
*       clause, i.e., a clause that is a single literal, assign
*       to the corresponding variable the value that will
*       satisfy that literal.
*
*   The consensus-based algorithm generalizes the Unit-Clause
*   rule to clauses of a certain length that can be combined
*   with other clauses to reduce the number of clauses in the
*   formula and hopefully the size of the search tree.
*
*   If the macro DAVIS_PUTNAM is defined, the program will
*   use DP, otherwise it will use the consensus algorithm.
*
*   Usage: dp [ -q | -ck | -sg | -gi ] [file-name]
*
*   If given a command-line switch, the program will print the
*   satisfying assignment it finds (if any) in a form appropriate
*   to the formulation given:
*   -q for n-queens problem
*   -ck for clique problem
*   -sg for subgraph isomporphism problem
*   -gi for graph isomorphism problem
*   The file-name parameter is the name of the file from which
*   input is
*   coming (ignored if not given).   If the formula is not
*   satisfiable, then the file is removed to prevent any other
*   program from wasting  time on it.
*
*   The program will print "SAT" or "NOT" followed by the CPU
*   time consumed exitting.  "SAT" means the formula was found
*   to be satisfiable; "NOT" means a contradiction was found, so
*   the formula is unsatisfiable.
*
*   The file contains the following functions:
*
*    void demorgan (f, n):
*       f is a formula in an array of clauses; complement each
*       literal in each clause as if applying DeMorgan's law
*       (~(p or q) = ~p or ~q )
*
*    int find_opposite (a, b):
*       a and b are clauses; return the number of a variable
```

```
*       that 's complemented in one clause but not the other
*       (-1 if none is found).
*
*    int less_than (a, b):
*       return 1 iff the clause a is "less than" the clause b.
*       this partial order is defined "a < b iff a(x) => b(x)
*       for all assignments x"
*
*    int d (a, b, default_d):
*       return the maximum of the degrees of a and b if
*       default_d is -1, or just return default_d otherwise.
*
*    int exist_mono (f, n, alpha, default_d):
*       The following is part of the consensus (and DP)
*       algorithm: while there exist two clauses mu and nu of f
*       such that alpha = consensus (mu, nu) is not "less than"
*       a clause of f and the degree of alpha is less than
*       d(mu, nu) ... this function finds (if possible) such
*       an alpha and returns 1.  if it can't find an alpha, it
*       returns 0.
*
*    int do_algorithm_1 (f, n, default_d):
*       do "algorithm one" from Billionett et. al.
*
*    int count_literal (f, n, y, y_comp):
*       return the number of times the literal with variable
*       'y' complemented if y_comp == 1 appears in the n-clause
*       formula f
*
*    int nonlinear (f, n, y, comp):
*       returns 1 if the literal (y, comp) appears in a clause
*       of degree greater than 1 in the n-clause formula f
*
*    int do_algorithm_2 (f, n):
*       do "algorithm two" from Billionett et. al.
*
*    int do_new_sat (f, n, var, value, success, new_f):
*       call sat(), setting the variable 'var' to value.
*
*    int find_nonlinear_var (f, n):
*       return the variable number of a literal in the n-clause
*       formula f that has degree > 1 in some clause of f.
```

```
*
*    int branch_function (f, n, var):
*       this is the function described in Billionett et. al. as
*       the branch criterion; if 'var' maximizes this function,
*       it will be chosen as the variable (in e.g. do_new_sat())
*       to branch on.
*
*    int choose_branch_var (f, n):
*       call branch_function() on all the variables, returning
*       the number of the one that maximizes the
*       branch_function()
*
*    int choose_branch_value (f, n, var):
*       choose the value for variable var in function f that
*       "looks like" it has the best chance for satisfying all
*       the clauses in f. we do this by looking at the increase
*       in the number of satisfied and unit clauses created by
*       putting either a 0 or 1 in var
*
*    int do_algorithm_3 (f, n, success):
*       do the implicit tree search of the formula after
*       tweaking the formula with algorithms one and two
*
*    int sat (clauses, n, success):
*       determine the satisfiabiliy of the formula in the array
*       clauses using DP or the consensus algorithm; calls
*       the different algorithms involved.
*
*    int cmp (a, b):
*       compare two clauses lexically; used as an argument to
*       qsort(3) to aid in eliminating duplicate clauses
*
*    int elim_dups (f, n):
*       after the array of clauses f has been qsort()ed, go
*       through the array eliminating duplicate clauses (which
*       will be bunched together because of the sort - good old
*       O(n log n) algorithms)
*
*    int *get_assignment (f, n):
*       after the formula in f has been determined to be
*       satisfiable, a satisfying assignment is returned in an
*       integer array
```

```
 *
 *    clause *consensus1 (a, b):
 *       do the consensus operation on clauses a and b
 *
 *    clause *consensus (a, b):
 *       wrapper for the consensus operation (for debugging)
 *
 *    clause *get_nonlinear_minimum (f, n):
 *       algorithm two contains the following:
 *       while 1 is not a clause in f and there exists a
 *       nonlinear literal of f, choose the literal y_i that is
 *       nonlinear in f and that appears in the fewest terms of
 *       f. get_nonlinear_minimum() finds and returns such a
 *       y_i.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <unistd.h>

#define MAX_VARIABLES 10000
#define MAX_CLAUSE 10000
#define MAX_OPS    20000
#define NO_GC
#define QUIET
#define M 100

int num_clauses,    /* number of clauses and variables;
                       set by */
    num_variables,  /* the clauselib.c functions when formula
                       is read in */
    calls = 0,      /* number of recursive calls made to sat() */
    alg1 = 0,
    cons = 0,
    not_cons = 0,
    ncons = 0,
    suggested_one = 0,
    suggested_zero = 0,
    was_wrong = 0;
int wrong[2][MAX_VARIABLES],
```

```
    right[2][MAX_VARIABLES];

/* get the time reporting and clause libraries */

#include "../lib/report.c"
#include "../lib/clauselib.c"

int find_opposite (a, b)
clause *a, *b;
{
   /* find a variable that's complemented in one but not
      the other */

   clause *p;

   while (a) {
      p = in (a->var, b);
      if (p) if (a->comp != p->comp) return a->var;
      a = a->next;
   }
   return -1;
}

clause *consensus1 (a, b)
clause *a, *b;
{
   int v;
   clause *con, *p, *q;

   v = find_opposite (a, b);
   if (v == -1) return none;
   con = NULL;
   for (p=a; p; p=p->next) {
      if (p->var != v) insert_literal (&con, p->var, p->comp);
   }
   for (p=b; p; p=p->next) {
      if (p->var != v) {
         insert_literal (&con, p->var, p->comp);

         /* if at *this* point con is null, then we *know* that
          * inserting something into it annhilated it, so we can
          * safely say that these two clauses don't have a
```

```
        * consensus.
        */

       if (con == NULL) return none;
     }
  }
  return con;
}

clause *consensus (a, b)
clause *a, *b;
{
  clause *p;

  cons++;
  ncons++;
  p = consensus1 (a, b);
  if (p == none) not_cons++;
  return p;
}

int less_than (a, b)
clause *a, *b;
{

  clause *p, *q;
  int ret;

  p = NULL;
  for (q=a; q; q=q->next)
     insert_literal (&p, q->var, q->comp);
  for (q=b; q; q=q->next) {
     insert_literal (&p, q->var, q->comp);
     if (p == NULL) {
        free_clauses (p);
        return 0;
     }
  }
  if (equal (a, p)) ret = 1; else ret = 0;
  free_clauses (p);
  return ret;
}
```

```
int d (a, b, default_d)
clause *a, *b;
int default_d;
{
   int r, da, db;

   da = degree (a);
   db = degree (b);
   if (da > db) r = da; else r = db;
   if ((default_d == -1) || (default_d > r)) return r;
   return default_d;
}

/* a typedef and some functions to handle quick building of a
 * table
 * of possible consensus operations
 */

typedef struct _op {
  clause *c;
  struct _op *next;
} op;

op ops[MAX_OPS];
int num_op;
op *new_op () {
   if (num_op >= MAX_OPS) {
      fprintf (stderr, "need more ops.\n");
      exit (1);
   }
   return &ops[num_op++];
}

void insert_op (L, c)
op **L;
clause *c;
{
   op *p;

   p = new_op ();
   p->c = c;
```

```
   p->next = *L;
   *L = p;
}

op *cns[MAX_VARIABLES][2];

int exist_mono (f, n, alpha, default_d)
clause *f[];
int n, default_d;
clause **alpha;
{
   int i, j, k, flag, dd, other, max_elim= 0, count, tries = 0;
   clause *mu, *nu, *a, *best_clause;
   op *p;

   if (default_d == -1) default_d = 3;
   num_op = 0;
   for (i=0; i<num_variables; i++) for (j=0; j<2; j++)
      cns[i][j] = NULL;
   best_clause = none;
   for (i=0; i<n; i++) {
      mu = f[i];
      for (a=mu; a; a=a->next) {
         other = !(a->comp);
         /* try to get a consensus between mu and all the
          * current members of the 'other' list
          */
#ifdef DAVIS_PUTNAM
         if (degree (mu) == 1)
#else
         if (mu && (degree (mu) <= default_d))
#endif
         for (p=cns[a->var][other]; p; p=p->next) {
            *alpha = consensus (mu, p->c);
            if (*alpha != none)
               if (degree (*alpha) <= d (mu, p->c, default_d)) {
               flag = 1;
               count = 0;
               for (i=0; flag && (i<n); i++) {
                  if (less_than (*alpha, f[i])) flag = 0; else
                  if (less_than (f[i], *alpha)) count++;
               }
```

```
            if (flag) {
               if (count > max_elim) {
                  max_elim = count;
                  best_clause = *alpha;
               }
               if (best_clause != none) tries++;
               if (tries > 10) {
                  *alpha = best_clause;
                  if (best_clause != none) return 1;
               }
            }
         }
      }

      /* no good consensus?  oh well, let's insert this
         clause */

      if (degree (mu) <= default_d)
         insert_op (&(cns[a->var][a->comp]), mu);
   }
}
if (best_clause != none) {
   *alpha = best_clause;
   return 1;
}
return 0;
}

/* some functions to help sort a formula by the size of clauses;
 * helpful in both DP and BS since unit consensuses will be found
 * first; these are usually the most profitable, since they cause
 * immediate promotion of other clauses to units. this is done
 * really for the sake of the BS algorithm, which performs
 * even more abysmally if we don't do this.
 */

int size_cmp (a, b)
clause **a, **b;
{
   int c, d;

   c = degree (*a);
```

```
   d = degree (*b);
   if (c < d) return -1;
   if (c > d) return 1;
   return 0;
}

int sort_by_size (f, n)
clause *f[];
int n;
{
   static clause *one[MAX_CLAUSE],
                 *two[MAX_CLAUSE],
                 *other[MAX_CLAUSE];
   int i, d, i1=0, i2=0, io=0;

   for (i=0; i<n; i++) {
      d = degree (f[i]);
      switch (d) {
         case 1: one[i1++] = f[i]; break;
         case 2: two[i2++] = f[i]; break;
         default: other[io++] = f[i];
      }
   }
   d = 0;
   for (i=0; i<i1; i++) f[d++] = one[i];
   for (i=0; i<i2; i++) f[d++] = two[i];
   for (i=0; i<io; i++) f[d++] = other[i];
   if (d != n) printf ("Hmm!\n"), exit (1);
   return i1;
}

int do_algorithm_1 (f, n, default_d)
clause *f[];
int n, default_d;
{
   clause *alpha;
   int i, i1, j, flag, count = 0;

   i1 = sort_by_size (f, n);
   if (i1) default_d = 1;
   flag = exist_mono (f, n, &alpha, default_d);
   while (flag) {
```

```
      if (alpha == NULL) {
         f[0] = NULL;
         return 1;
      }
      j = 0;
      for (i=0; i<n; i++)
         if (!less_than (f[i], alpha))
            insert_clause (f, &j, f[i]);
      insert_clause (f, &j, alpha);
      n = j;
      i1 = sort_by_size (f, n);
      if (i1) default_d = 1;
      flag = exist_mono (f, n, &alpha, default_d);
      count++;
   }
   return n;
}

int count_literal (f, n, y, y_comp)
clause *f[];
int n, y, y_comp;
{
   int i, count = 0;
   clause *p;

   for (i=0; i<n; i++) {
      p = in (y, f[i]);
      if (p) if (p->comp == y_comp) count++;
   }
   return count;
}

int nonlinear (f, n, y, comp)
clause *f[];
int n, y, comp;
{
   int i, linear;
   clause *p;

   for (i=0; i<n; i++) {
      p = f[i];
      if (degree (p) == 1) if (p->var == y) return 0;
```

```
   }
   return 1;
}

clause *get_nonlinear_minimum (f, n)
clause *f[];
int n;
{
   int i, y, comp, count, min, least_y, least_y_comp;
   static clause foo;

   for (i=0; i<n; i++) if (f[i] == NULL) return NULL;
   min = MAX_CLAUSE + MAX_VARIABLES + 1;
   least_y = -1;
   for (y=0; y<num_variables; y++) {
      for (comp=0; comp<2; comp++) {
         if (nonlinear (f, n, y, comp)) {
            count = count_literal (f, n, y, comp);
            if (count) if (count <= min) {
               min = count;
               least_y = y;
               least_y_comp = comp;
            }
         }
      }
   }
   if (least_y == -1) return none;
   foo.var = least_y;
   foo.comp = least_y_comp;
   return &foo;
}

/*
 *       do "algorithm two" from Billionett et. al.
 */

int do_algorithm_2 (f, n)
clause *f[];
int n;
{
   clause *y, *z, *new_f[MAX_CLAUSE];
   int i, old_n;
```

```
   old_n = n;
   for (i=0; i<n; i++) new_f[i] = f[i];
   y = get_nonlinear_minimum (new_f, n);
   while (y && (y != none)) {
      z = new_clause (y->var, !y->comp);
      insert_clause (new_f, &n, z);
      n = do_algorithm_1 (new_f, n, 2);
      y = get_nonlinear_minimum (new_f, n);
   }
   if (y == NULL) {
      f[0] = NULL;
      return 1;
   }

   /* at this point, if all terms are linear, we have a
      solution! */

   for (i=0; i<n; i++) {
      if (degree (new_f[i]) > 1) return old_n;
   }
   for (i=0; i<n; i++) f[i] = new_f[i];
   return n;
}

int sat ();

/*
 *       call sat(), setting the variable 'var' to value.
 */

int do_new_sat (f, n, var, value, success, new_f)
clause *f[],     /* the old function */
       *new_f[]; /* the new function after the assignment
                    of the value */
int n, var, value,
    *success;    /* set to 1 if we satisfied f */
{
   clause *p, *q, *r;
   int i, j, new_n;

   j = 0;
```

```
for (i=0; i<n; i++) {

   /* if the variable is in this clause... */

   if (r = in (var, f[i])) {

      /* ...and if the literal evaluates to one,
       * then delete it from the clause
       */

      /* (here, we're saying "if the variable is not
       * complemented and is equal to 1, or if the variable
       * is complemented and is equal to 0, then it evaluates
       * to 1.")
       */
      if ( (value && !(r->comp)) || (!value && (r->comp)) ) {
         p = NULL;
         for (q=f[i]; q; q=q->next)
            if (q->var != var)
               insert_literal (&p, q->var, q->comp);
         insert_clause (new_f, &j, p);
      } /* else we don't include this clause;
         * it is annhilated by the zero
         */
   } else {
      insert_clause (new_f, &j, f[i]);

      /* make sure we don't delete this one in the future */

      f[i]->copies++;
   }
}
new_n = sat (new_f, j, success);
if (*success) {
   /* yea! */
   for (i=0; i<new_n; i++) f[i] = new_f[i];

   /* put in the single literal the way it will help
    * satisfy the formula
    */
   insert_clause (f, &new_n, new_clause (var, value));
   return new_n;
```

```
   }
   return n;
}

int find_nonlinear_var (f, n)
clause *f[];
int n;
{
   int i;
   clause *p;

   for (i=0; i<n; i++)
      if (degree (f[i]) > 1) return f[i]->var;
   return -1;
}

int branch_function (f, n, var)
clause *f[];
int n, var;
{
   int i, v, w, sum;
   clause *p;

   v = 0, w = 0;
   for (i=0; i<n; i++) {
      if (degree (f[i]) == 3) {
         p = in (var, f[i]);
         if (p) {
            if (p->comp) w++; else v++;
         }
      }
   }

   /* this is stupid ... */

   if (v < w) sum = v; else sum = w;
   sum *= M;
   sum += v; sum += w;
   return sum;
}

int choose_branch_var (f, n)
```

```
clause *f[];
int n;
{
   int i, k, b, max, save;

   save = find_nonlinear_var (f, n);
   if (save == -1) return -1;
   k = -1;
   max = 0;
   for (i=0; i<n; i++) {
      b = branch_function (f, n, i);
      if (b > max) max = b, k = i;
   }
   if (k == -1) return save; else return k;
}

int choose_branch_value (f, n, var)
clause *f[];
int n, var;
{
#ifndef DAN_BRANCH1
   return 0;
#else
   int i, j, d, suggestion,
      satisfied[2]; /* number of clauses satisfied by letting
                       var = index */
   clause *c;

   satisfied[0] = 0;
   satisfied[1] = 0;
   for (i=0; i<n; i++) {
      /* look for the variable in this clause */
      c = in (var, f[i]);
      if (c) {
         d = degree (f[i]);
         for (j=0; j<2; j++) {
            /* if we let var = j, then do we get a satisfied
               (0) clause? */
            if (j == c->comp) satisfied[j]++;
         }
      }
   }
```

```
   if ( (satisfied[0]) > (satisfied[1]) )
      suggestion = 1;
   else
      suggestion = 0;
   if (suggestion) suggested_one++; else suggested_zero++;
   return suggestion;
#endif
}

int do_algorithm_3 (f, n, success)
clause *f[];
int n, *success;
{
   clause *new_f[MAX_CLAUSE];
   int hmm, i, var, m, value, w0, w1, r0, r1,
       vote_for_1, vote_for_0;

   var = choose_branch_var (f, n);
   if (var == -1) {
      *success = 1;
      return n;
   }

   value = choose_branch_value (f, n, var);

#ifdef DAN_BRANCH2
   vote_for_1 = 0;
   vote_for_0 = 0;
   w0 = wrong[0][var];
   w1 = wrong[1][var];
   r0 = right[0][var];
   r1 = right[1][var];
   /* this rule gets us 10 seconds on 8-queens */
   if (w0 > r0) vote_for_1++;
   if (vote_for_1 > vote_for_0) value = 1;
#endif
   /* try the formula with what looks like a good value */

   m = do_new_sat (f, n, var, value, success, new_f);
   if (*success) {
#ifdef DAN_BRANCH2
      /* we were right about this value. */
```

```
      right[value][var]++;
#endif
      return m;
   }
   was_wrong++;
   /* we were wrong about this value */
#ifdef DAN_BRANCH2
   wrong[value][var]++;
#endif

   /* didn't work?  try it with the other value, then */

   m = do_new_sat (f, n, var, !value, success, new_f);
   if (*success) {
      /* ok, we were right about *this* value! */
#ifdef DAN_BRANCH2
      right[!value][var]++;
#endif
      return m;
   }

   /* still didn't work?  ok, f is not satisfiable.
    * make it obvious by putting in a NULL.
    */
   f[0] = NULL;
#ifdef DAN_BRANCH2
   wrong[!value][var]++;
#endif
   return 1;
}

int do_pure_literals (f, n)
clause *f[];
int n;
{
   int i, new_n;
   static int pure[MAX_VARIABLES];
   clause *c;

   new_n = n;
   for (i=0; i<num_variables; i++) pure[i] = -1;
   for (i=0; i<n; i++) {
```

```
    /* if this is already a literal of the formula, don't
     * insert it at all; it's already there.  if it isn't
     * in 2-CNF, insert the variables.
     */
    c = f[i];
    if (degree (c) != 2)
       for (; c; c=c->next) pure[c->var] = -2;
    else for (; c; c=c->next) {

       /* if this is the first time we've seen this variable,
        * put whether or not it's complicated in the 'pure'
        * array
        */
       if (pure[c->var] == -1) pure[c->var] = c->comp; else

       /* maybe we've seen this before; if so and it had a
        * different attitude (i.e., was or was not complemented
        * where now it's the opposite) put a -2 in the 'pure'
        * array
        */
       if (c->comp != pure[c->var]) pure[i] = -2;
    }
  }

  /* we could be linear in the number of variables, but this
   * seems faster?
   */
  for (i=0; i<n; i++) {
    for (c = f[i]; c; c=c->next) {
       if (pure[c->var] >= 0)
          insert_clause (f, &new_n,
             new_clause (c->var, pure[c->var]));
       pure[c->var] = -2;
    }
  }
  return new_n;
}

int sat (clauses, n, success)
clause *clauses[];
int n, *success;
```

```
{
   int i;

   calls++;
#ifdef DAN_LITERAL
   n = do_pure_literals (clauses, n);
#endif
   if (calls == 1) n = do_algorithm_1 (clauses, n, 3);
   else n = do_algorithm_1 (clauses, n, 2);
   /* at this point, check to see if 1 is a monomial of clauses.
    * if so, we can't satisfy it so return 0.
    * otherwise, we don't know anything so go on to algorithm 2
    */
   for (i=0; i<n; i++) if (clauses[i] == NULL) {
      clauses[0] = NULL;
      return 1;
   }
#ifndef QUIET
   printf ("oh, well.\n");
#endif
#ifndef DAVIS_PUTNAM
   if (calls == 1) n = do_algorithm_2 (clauses, n);
   /* at this point, either clauses is 1 (not satisfiable) or
    * is a disjunction of *literals* which describe a satisfying
    * assignment to the variables
    */
#endif
   n = do_algorithm_3 (clauses, n, success);
   return n;
}

int cmp (a, b)
clause **a, **b;
{
   if ((*a)->var < (*b)->var) return -1;
   if ((*a)->var > (*b)->var) return 1;
   return 0;
}

int elim_dups (f, n)
clause *f[];
int n;
```

```
{
   int i, j;
   clause *new_f[MAX_CLAUSE];

   j = 0;
   new_f[j++] = f[0];
   for (i=1; i<n; i++) {
      if (f[i-1]->var != f[i]->var) new_f[j++] = f[i];
   }
   for (i=0; i<j; i++) f[i] = new_f[i];
   return j;
}

int *get_assignment (f, n)
clause *f[];
int n;
{
   int i, T[MAX_VARIABLES];

   /* find the assignment that will satisfy f(x) = 0 */

   for (i=0; i<MAX_VARIABLES; i++) T[i] = 0;
   for (i=0; i<n; i++)
      if (f[i]->comp) T[f[i]->var] = 1;
   return T;
}

void demorgan (f, n)
clause *f[];
int n;
{
   int i;
   clause *p;

   for (i=0; i<n; i++)
      for (p=f[i]; p; p=p->next) p->comp = !p->comp;
}

main (argc, argv)
int argc;
char *argv[];
{
```

```
char s[1000];
int i, *t, success, queens = 0, subgraphs = 0, graph_iso = 0,
    clique = 0, graph_n, graph_m, delete = 0;
clause *p;
clause *clauses[MAX_CLAUSE];
FILE *f;

if (argc >= 2) {
   if (strcmp (argv[1], "-q") == 0) queens = 1; else
   if (strcmp (argv[1], "-sg") == 0) subgraphs = 1; else
   if (strcmp (argv[1], "-gi") == 0) graph_iso = 1; else
   if (strcmp (argv[1], "-cl") == 0) clique = 1; else
   if (strcmp (argv[1], "-d") == 0) delete = 1;
}
initialize_time ();
num_clauses = 0;
num_variables = 0;
for (i=0; i<MAX_VARIABLES; i++) {
   wrong[0][i] = 0;
   wrong[1][i] = 0;
   right[0][i] = 0;
   right[1][i] = 0;
}
none = new_clause (1, 1);
none->next = NULL;
srand (time (NULL));
while (!feof (stdin)) {
   gets (s);
   if (!feof (stdin)) if (s[0] != '#') {
      p = make_clause (s);
      if (p) clauses[num_clauses++] = p;
   }
   if (s[0] == '#') {
      switch (s[1]) {
         case '1': graph_n = atoi (&s[3]); break;
         case '2': graph_m = atoi (&s[3]); break;
         default: break;
      }
   }
}
num_variables++;
```

```
   for (i=0; i<num_clauses; i++) if (clauses[i] == NULL) {
      num_clauses = i;
      break;
   }

   /* convert this CNF formula into DNF because this algorithm
    * tries to do the dual thing of CND-SAT, i.e., DNF-TAUT
    */

   demorgan (clauses, num_clauses);

#ifndef QUIET
   printf ("%d clauses in %d variables\n",
      num_clauses, num_variables);
# ifdef ECHO_FORMULA
   for (i=0; i<num_clauses; i++) print_clause_ln (clauses[i]);
# endif
#endif
   success = 0;

   num_clauses = sat (clauses, num_clauses, &success);

   if (clauses[0] == NULL) {
      if (delete) unlink (argv[2]);
      report_time ("NOT");
   } else {
      report_time ("SAT");
      if (queens) {
         t = get_assignment (clauses, num_clauses);
         print_queens (t, num_variables);
      } else if (subgraphs && graph_n) {
         t = get_assignment (clauses, num_clauses);
         print_subgraph (t, graph_n, graph_m, 1);
      } else if (graph_iso && graph_n) {
         t = get_assignment (clauses, num_clauses);
         print_subgraph (t, graph_n, graph_m, 2);
      } else if (clique && graph_n) {
         t = get_assignment (clauses, num_clauses);
         print_subgraph (t, graph_n, graph_m, 3);
      }
   }
```

```
   qsort ((void *) clauses, (size_t) num_clauses,
      (size_t) sizeof (clause *),
      (int (*) (const void *, const void *)) cmp);
   num_clauses = elim_dups (clauses, num_clauses);
#ifdef DAVIS_PUTNAM
# if defined(DAN_BRANCH1) || defined(DAN_BRANCH2)
   f = fopen ("./dandp.stats", "a");
   fprintf (f, "dandp: ");
# else
   f = fopen ("./dp.stats", "a");
   fprintf (f, "dp: ");
# endif
#else
   f = fopen ("./consensus.stats", "a");
   fprintf (f, "consensus: ");
#endif
   fprintf (f, "%d calls to sat()\n", calls);
#if defined(DAN_BRANCH1) || defined(DAN_BRANCH2)
   fprintf (f, "suggested 0 %d times\n", suggested_zero);
   fprintf (f, "suggested 1 %d times\n", suggested_one);
#endif
   fprintf (f, "took the wrong branch %d times\n", was_wrong);
   fprintf (f, "alg1 = %d\n", alg1);
   fprintf (f, "cons = %d\n", cons);
   fprintf (f, "not_cons = %d\n", not_cons);
   fclose (f);
#ifndef QUIET
   printf ("output from sat:\n");
   for (i=0; i<num_clauses; i++) print_clause_ln (clauses[i]);
#endif
}
```

end of bsdp.c

Appendix B

The GSAT program

This is the program that implements the GSAT procedure. It was written by the author of the thesis.

gsat.c:

```
/*
 *   gsat.c
 *
 *   This program is an implementation of the algorithm GSAT i
 *   described in Selman, B., and Levesque, H.J., and Mitchell, D.
 *   G. (1992), A New Method for Solving Hard Satisfiability
 *   Problems. Proceedings of the Tenth National Conference on
 *   Artificial Intelligence (AAAI-92), San Jose, CA, July 1992,
 *   440-446.
 *
 *   The algorithm uses a greedy strategy to satisfy as many
 *   clauses as possible, hopefully satisfying them all.
 *
 *   If the macro GUESSING is #define'd, the program uses a
 *   completely random strategy (i.e., independent guessing),
 *   otherwise GSAT is used.
 *
 *   Usage: gsat [ -q | -ck | -sg | -gi]
```

```
*   If given a command-line switch, gsat will print the
*   satisfying assignment it finds (if any) in a form
*   appropriate to the formulation given:
*   -q for n-queens problem
*   -ck for clique problem
*   -sg for subgraph isomporphism problem
*   -gi for graph isomorphism problem
*
*   gsat will print "SAT" or "???" followed by the CPU time
*   consumed on exitting.  "SAT" means the formula was found
*   to be satisfiable; "???" means GSAT couldn't find a
*   satisfying assignment.
*
*   The file contains the following functions:
*
*   int get_max_increase (t, c, n):
*      return the variable that, if toggled, will produce the
*      largest increased in the number of clauses satisfied.
*      note: even if no variable increases the clauses, we will
*      still return something since we have to change something
*      for any hope of satisfying the formula.
*
*   int *guessing (clauses, n, max_tries):
*      try to guess a satisfying assignment to the formula in
*      the clause array clauses max_tries times, then give up.
*      returns an array of int's that is a satisfying
*      assignment, or NULL if none is found.
*
*   int *gsat (clauses, n, max_flips, max_tries):
*      do the GSAT procedure on the n-clause formula in
*      clauses.  max_flips is the number of times to ``flip,''
*      or toggle a variable.  max_tries is the number of times
*      to try max_flips flipps before giving up.
*
*   void get_random (t):
*      put a random sequence of bits in the int array t.
*      we put num_variables (a global set when reading in the
*      clauses) random bits in t.
*
*   main(argc, argv):
*      the main program; handles the command line and reads in
*      the formula.
```

```
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <unistd.h>

#define MAX_VARIABLES 10000
#define MAX_CLAUSE 10000
#define MAX_FLIPS 100
#ifdef GUESSING
# define MAX_TRIES 50000
#else
# define MAX_TRIES 5000
#endif

/* supress debug output */
#define QUIET

int num_clauses,    /* number of clauses and variables; */
    num_variables;  /* set by clauselib.c */


/* include the libraries needed (I know, I know) */

#include "../lib/report.c"
#include "../lib/clauselib.c"


/* the set of clauses we will read in */

clause *clauses[MAX_CLAUSE];

void get_random (t)
int *t;
{
   int i;

   for (i=0; i<num_variables; i++) t[i] = rand () % 2;
}
```

```
int get_max_increase (t, c, n)
int *t;
clause *c[];
int n;
{
   int i, k, max, maxi;

   max = 0;
   maxi = 0;
   for (i=0; i<n; i++) {
      t[i] = !t[i];
      k = satisfies (t, c, n);
      t[i] = !t[i];
      if (k > max) max = k, maxi = i;
   }
   return maxi;
}

int *guessing (clauses, n, max_tries)
clause *clauses[];
int n, max_tries;
{
   /* just guess randomly */

   static int T[MAX_VARIABLES];
   FILE *f;
   int i;

   for (i=0; i<max_tries; i++) {
      get_random (T);
      if (satisfies (T, clauses, n) == n) {

        /* print out some statistics before we have to leave...*/

         f = fopen ("./guessing.stats", "a");
         fprintf (f, "guessing: %d tries before success\n",
            i + 1);
         fclose (f);
         return T;
      }
   }
```

```
   /* couldn't figure it out; print out how hard we tried */

   f = fopen ("./guessing.stats", "a");
   fprintf (f, "guessing: %d tries before failure\n", max_tries);
   fclose (f);
   return NULL;
}

int *gsat (clauses, n, max_flips, max_tries)
clause *clauses[];
int n, max_flips, max_tries;
{
   int i, j, p;
   static int T[MAX_VARIABLES];
   FILE *f;

   for (i=0; i<max_tries; i++) {
#ifndef QUIET
      printf (".");fflush(stdout);
#endif
      get_random (T);
      for (j=0; j<max_flips; j++) {
#ifndef QUIET
         printf ("-");fflush (stdout);
#endif
         if (satisfies (T, clauses, n) == n) {

            /* print out some statistics about how long it took
             * to find an assignment; this is useful for tuning
             * MAX_FLIPS and MAX_TRIES.
             */
            f = fopen ("./gsat.stats", "a");
            fprintf (f,
               "gsat: %d flips, %d tries before success\n",
               j+1, i+1);
            fclose (f);
            return T;
         }
         p = get_max_increase (T, clauses, n);
         T[p] = !T[p];
      }
   }
```

```
   f = fopen ("./gsat.stats", "a");
   fprintf (f, "gsat: %d flips, %d tries before failure\n",
      max_flips, max_tries);
   fclose (f);
   return NULL;
}

main (argc, argv)
int argc;
char *argv[];
{
   char s[10000];
   int i, *t,
       queens = 0,     /* set to 1 if -q option */
       subgraphs = 0,  /* set to 1 if -sg option */
       graph_iso = 0,  /* set to 1 if -gi option */
       clique = 0,     /* set to 1 if -ck option */
       graph_n,        /* number of nodes in the first graph */
       graph_m;        /* number of nodes in the second graph */
   clause *p;

   if (argc == 2) {
      if (strcmp (argv[1], "-q") == 0) queens = 1; else
      if (strcmp (argv[1], "-sg") == 0) subgraphs = 1;
      if (strcmp (argv[1], "-gi") == 0) graph_iso = 1; else
      if (strcmp (argv[1], "-cl") == 0) clique = 1;
   }
   initialize_time ();
   num_clauses = 0;
   num_variables = 0;
   srand (time (NULL) + getpid ());


   /* read the formula from stding */

   while (!feof (stdin)) {
      gets (s);
      if (!feof (stdin)) if (s[0] != '#') {
         p = make_clause (s);
         if (p) clauses[num_clauses++] = p;
      }
```

```
      /* comments (preceded by '#') in the input will tell us,
       * for a graph problem, the sizes of the graphs
        */
      if (s[0] == '#') {
         switch (s[1]) {
            case '1': graph_n = atoi (&s[3]); break;
            case '2': graph_m = atoi (&s[3]); break;
            default: break;
         }
      }
   }
   num_variables++;

   /* get rid of NULL clauses that pop up occasionally */

   for (i=0; i<num_clauses; i++) if (clauses[i] == NULL) {
      num_clauses = i;
      break;
   }
#ifndef QUIET
   for (i=0; i<num_clauses; i++) print_clause_ln (clauses[i]);
#endif
#ifdef GUESSING
   t = guessing (clauses, num_clauses, MAX_TRIES);
#else
   t = gsat (clauses, num_clauses, MAX_FLIPS, MAX_TRIES);
#endif

   /* if t isn't NULL, gsat (or guessing) returned with a
    * satisfying assignment to the formula; report a SAT
    */
   if (t) {
#ifdef QUIET
   report_time ("SAT");
#else
      printf ("satisfying assignment:\n");
      for (i=0; i<num_variables; i++)
         printf ("%d %d\n", i, t[i]);
#endif

      /* of one of the formulations was specified, print out the
       * satisfying assignment in the appropriate format
```

```
     */
    if (queens) {
       print_queens (t, num_variables);
    } else if (subgraphs && graph_n) {
       print_subgraph (t, graph_n, graph_m, 1);
    } else if (graph_iso && graph_n) {
       print_subgraph (t, graph_n, graph_m, 2);
    } else if (clique && graph_n) {
       print_subgraph (t, graph_n, graph_m, 3);
    }
  } else
#ifdef QUIET

  /* we didn't find a satisfying assignment; report ??? */

  report_time ("???");
#else
  printf ("no satisfying assignment found.\n");
#endif
}
```

end of gsat.c

Appendix C

The Clause and Timing libraries

These are libraries of functions important in the two other programs. The first one, `clauselib.c`, contains code for manipulating clauses and formulas, and also for printing satisfying assignments as n-queens chessboards and graph isomorphism functions. The second has a two functions that facilitate and standardize gathering timing information on the two programs.

`clauselib.c`:

```
/*
 *   clauselib.c
 *
 *   This file contains C code to handle ``clauses'' of boolean
 *   variables.
 *
 *   A clause is a set of literals, which are complemented or
 *   uncomplemented boolean variables.
 *
 *   The file also contains two routines for printing output
 *   related to the n-queens and graph SAT formulations.
 *
 *   The following functions are in this file:
 *
 *   void free_clauses (p):
 *      free memory malloc'ed for a clause p
 *
 *   void insert_clause (f, n, p):
```

```
*      insert a clause p into an array f of n clauses
*
*   void insert_literal (p, var, comp):
*      insert a literal variable var into clause p,
*      complemented iff comp
*
*   void print_clause (p):
*      print a clause in a human-readable form (for debugging)
*
*   void insert_unique_clause (f, n, p):
*      insert a clause p into an array f, eliminating duplicates
*
*   void print_ugly_clause (p):
*      print a clause p in a form readable by the SAT algorithms
*
*   void print_queens (t, n):
*      print the variable assignment t as an n-queens chessboard
*
*   void print_subgraph (t, n, m, hmm):
*      print the assignment t as a graph isomorphism
*      (hmm is a flag telling which kind of isomorphism:
*      clique, graph, or subgraph
*
*   int satisfies (t, c, n):
*      returns the number of clauses of the formula c (an array
*      of n clauses) satisfied by the variable assignment in t.
*      note: t satisfies c iff satisfies() returns n.
*
*   int equal (a, b):
*      returns 1 iff the clause a is equal to the clause b
*
*   int degree (a):
*      returns the number of literals in the clause a
*
*   int isqrt (n):
*      returns integer square root of n iff n is a perfect
*      square < 50^2
*
*   clause *new_clause (var, comp):
*      returns a pointer to a new clause with variable var,
*      complemented if comp == 1
*
```

```
 *   clause *make_clause (s):
 *      convert the machine-readable clause format (output by
 *      the formulation programs or print_ugly_clause()) into a
 *      linked list clause, and return the new clause
 *
 *   clause *in (v, p)
 *      returns a pointer to the element of a clause p
 *      containing a variable v, or NULL if no such element
 */

/* types and variables */
/* a clause is a triple with the variable number and a boolean
 * 0 if not comp, 1 if comp
 */
typedef struct _clause {
   int var,     /* the variable number */
       comp,    /* 1 iff the literal is complemented */
       copies;  /* number of aliases to this pointer
                 * (for garbage collection)
                 */
   struct _clause *next; /* next literal in this linked list */
} clause;

clause *none; /* a special value for ``not a clause,'' returned
               * by the consensus operation when there is no
               * consensus
               */
clause *new_clause (var, comp)
int var, comp;
{
   clause *p;

   p = (clause *) malloc (sizeof (clause));
   if (!p) {
      fprintf (stderr, "malloc: returned NULL :-(\n");
      exit (1);
   }
   p->next = NULL;
   p->var = var;
   p->comp = comp;
   p->copies = 0;
   return p;
```

```
}

void free_clauses (p)
clause *p;
{
   clause *q;

   if (p == NULL) return;
   q = p->next;
   free (p);
   free_clauses (q);
}

void insert_clause (f, n, p)
clause *f[];
int *n;
clause *p;
{

   /* make sure the size of the clause doesn't exceed the maximum
    * number in an array of clauses.  this can happen for
    * exponential space runs of the DP algorithm; the fix is to
    * recompile with a larger size since we don't want to give up
    * the speed afforded by an array rather than a list
    * implementation of formulas
    */
   if (*n >= MAX_CLAUSE) {
      fprintf (stderr, "MAX_CLAUSE exceeded.\n");
      report_time ("ERR");
      exit (0);
   }
   if (p == NULL) {
      f[0] = NULL;
      *n = 1;
   } else f[(*n)++] = p;
}

void insert_literal (p, var, comp)
clause **p;
int var, comp;
{
   clause *q, *r, **P;
```

```
   P = p;
loop:
   if (*p == NULL) {

      /* base case: inserting into an empty clause */

      *p = new_clause (var, comp);
      return;
   }

   /* case 1: we're trying to insert a duplicate.  ignore it.
    * (note: if it's the same literal, but not the same
    * complement, we have a contradiction so we can blow away
    * the whole set of clauses.)
    */

   if ((*p)->var == var) {
      if ((*p)->comp == comp) return;
      else {
         free_clauses (*P);
         *P = NULL;
         return;
      }
   }

   /* case 2: the current var is too big.
    * insert our var before it and link pointers
    */

   if ((*p)->var > var) {
      q = *p;
      *p = new_clause (var, comp);
      (*p)->next = q;
      return;
   }

   /* case 3: do the whole procedure on the next literal in the
    * clause since we didn't insert the new literal yet.
    * this was originally recursive, hence the goto (gcc
    * probably would have noticed that, but we're just making
    * sure here).
```

```
    */

   p = &((*p)->next);
   goto loop;
}

clause *make_clause (s)
char *s;
{
  int i, c;
  clause *p, *q;
  char *t;

#define max(c) if (c > num_variables) num_variables = c;

  p = NULL;
  s[strlen(s)+1] = 0;
  t = s;

  /* while there are more literals in the string...*/

  while (*t) {

     /* get rid of the space at the end of the digits */

     t[6] = 0;

     /* insert the value and complement into the new clause */

     insert_literal (&p, atoi (&t[1]), t[0] == '~');

     /* move 't' to the next literal */

     t += 7;
  }
  q = p;

  /* quick check to set the global 'num_variables' to the maximum
   * variable number we encounter
   */
  while (q) { max (q->var); q=q->next; }
  return p;
```

```
}

void print_clause (p)
clause *p;
{
   if (p == none) {
      printf ("( none )");
      return;
   }
   printf ("(");
   while (p) {
      printf ("%s%d ", p->comp ? " ~": " ", p->var);
      p = p->next;
   }
   printf (")");
}

#define print_clause_ln(p) (print_clause((p)),printf("\n"))

int satisfies (t, c, n)
int *t;
clause *c[];
int n;
{
   int i, o, count = 0;
   clause *p;

   for (i=0; i<n; i++) {

     /* we will find the `sum' of the values of the literals;
      * start out at 0.
      */
     o = 0;
     for (p=c[i]; p && !o; p=p->next) {
        if (p->comp) {
           /* if the literal is complemented and the variable
            * is 0, the clause is satisfied
            */
           if (t[p->var] == 0) o = 1;
        }

        /* otherwise if the literal isn't complemented and the
```

```
         * variable is assigned 1, the clause is satisfied
         */
        else if (t[p->var] == 1) o = 1;

        /* otherwise nothing; we're still at 0 */
     }

     /* increment count if this clause was satisfied */

     count += o;
   }
   return count;
}

clause *in (v, p)
int v;
clause *p;
{

   while (p) {
      if (p->var == v) return p;
      p = p->next;
   }
   return NULL;
}

int equal (a, b)
clause *a, *b;
{
   /* compare a and b for equality */

   while (a && b) {
      if (a->comp != b->comp) return 0;
      if (a->var != b->var) return 0;
      a = a->next;
      b = b->next;
   }
   if (a == b) return 1;
   return 0;
}

int degree (a)
```

```
clause *a;
{
   int c = 0;

   while (a) c++, a=a->next;
   return c;
}

int isqrt (n)
int n;
{
   int i;

   for (i=0; i<50; i++) if (i*i == n) return i;
   fprintf (stderr, "isqrt: n (%d) must be a square\n", n);
}

void insert_unique_clause (f, n, p)
clause *f[];
int *n;
clause *p;
{
   int i;

   for (i=0; i<*n; i++) if (equal (p, f[i])) return;
   if (*n > MAX_CLAUSE) {
      fprintf (stderr, "too many clauses\n");
      exit (1);
   }
   f[(*n)++] = p;
}

void print_ugly_clause (p)
clause *p;
{
   while (p) {
      printf ("%c%0.5d", p->comp ? '~' : '_', p->var);
      if (p->next) printf (" "); else printf ("\n");
      p = p->next;
   }
}
```

```
void print_queens (t, n)
int *t;
int n;
{
   int i, j, s;

   s = isqrt (n);
   for (i=0; i<s; i++) {
      for (j=0; j<s; j++) printf ("%d ", t[s*i+j]);
      printf ("\n");
   }
}

void print_subgraph (t, n, m, hmm)
int *t;
int n, m, hmm;
{
   int i, j, N;

   if (hmm == 1) printf ("Ok, here's a subgraph isomorphism:\n");
   if (hmm == 2) printf ("Ok, here's a graph isomorphism:\n");
   if (hmm == 3) printf ("Ok, here's a clique:\n{ ");
   N = n * m;
   for (i=0; i<N; i++)
      if (t[i]) {
         if (hmm == 3) printf ("%d ", i / m); else
         printf ("%d <--> %d\n", i / m, i % m);
      }
   if (hmm == 3) printf ("}\n");
}
```

end of clauselib.c

report.c:

```
/*
 *   report.c
 *
 *   This file contains C code to print the amount of CPU time
 *   used by a SAT procedure; the SAT program calls
 *   report_time() with a string (usually SAT or NOT) describing
 *   what it found.
 */

/* solaris needs this */

#ifndef CLOCKS_PER_SEC
# define CLOCKS_PER_SEC 1000000.0
#endif

/* solaris also needs us to call clock() once at the beginning
 * of the program (?)
 */
void initialize_time ()
{
#ifdef sun
   (void) clock ();
#endif
}

/* print out the value of clock() with the string to stderr */
void report_time (s)
char *s;
{
   fprintf (stderr, "%s\n", s);
   fprintf (stderr, "%0.3lf\n",
      (double) clock () / (double) CLOCKS_PER_SEC);
   fflush (stderr);
   fclose (stderr);
}
```

end of report.c

Appendix D

Tables of Computational Results

Here, we present the results of the experiments in a tabular form. For each problem, the instance size is given, followed by the number of instances used (n), the median time, the mean time (μ), and the standard deviation (σ) of the times.

D.1 Times for Random Formulas

BS				
Size	n	Median	μ	σ
50	7	0.6s	0.61s	0.08s
60	7	0.9s	1.01s	0.28s
70	7	1.5s	1.47s	0.16s
80	7	2.2s	2.17s	0.10s
90	7	3.1s	3.13s	0.12s
100	7	4.1s	14.21s	24.65s
110	7	5.6s	5.57s	0.14s
120	5	6.9s	6.96s	0.24s
130	6	10.3s	41.03s	69.63s
140	5	11.7s	11.74s	0.41s

DP Variant 1				
Size	n	Median	μ	σ
50	3	0.0s	0.03s	0.05s
60	3	0.1s	0.73s	0.90s
70	3	0.3s	0.93s	1.04s
80	3	0.3s	0.30s	0.08s
90	2	0.6s	0.45s	0.15s
100	2	1.9s	1.15s	0.75s
110	1	0.6s	0.60s	-
120	1	0.8s	0.80s	-
130	1	1.1s	1.10s	-
140	1	1.5s	1.50s	-

DP Variant 2				
Size	n	Median	μ	σ
50	3	0.0s	0.07s	0.09s
60	3	0.0s	0.03s	0.05s
70	3	0.1s	0.10s	0.00s
80	3	0.2s	0.20s	0.00s
90	3	0.2s	0.23s	0.05s
100	3	0.4s	0.40s	0.00s
110	2	0.5s	0.45s	0.05s
120	2	0.7s	0.65s	0.05s
130	2	0.9s	0.90s	-
140	2	4.6s	2.85s	1.75s

DP Variant 3				
Size	n	Median	μ	σ
50	3	2.3s	2.17s	1.64s
60	3	0.4s	4.77s	6.32s
70	3	0.2s	0.40s	0.28s
80	2	21.5s	10.85s	10.65s
90	3	0.5s	5.27s	6.88s
110	2	50.6s	38.30s	12.30s
120	1	1.0s	1.00s	-
130	2	13.5s	9.35s	4.15s
140	1	3.6s	3.60s	-

DP				
Size	n	Median	μ	σ
50	13	0.1s	0.16s	0.24s
60	14	0.1s	0.09s	0.03s
70	14	0.1s	0.14s	0.05s
80	14	0.2s	0.24s	0.10s
90	14	0.3s	0.26s	0.05s
100	14	0.4s	6.10s	20.44s
110	13	0.4s	0.45s	0.08s
120	11	0.6s	0.62s	0.08s
130	12	0.7s	0.85s	0.42s
140	12	1.1s	53.47s	109.92s

GSAT				
Size	n	Median	μ	σ
50	3	0.0s	0.17s	0.24s
60	3	0.0s	0.00s	-
70	3	0.0s	0.20s	0.28s
80	3	0.0s	0.27s	0.38s
90	3	0.1s	0.07s	0.05s
100	2	1.2s	0.65s	0.55s
110	2	0.2s	0.15s	0.05s
120	1	1.9s	1.90s	-
130	2	2.3s	1.25s	1.05s
140	2	2.7s	2.60s	0.10s

D.2 Times for Large Random Formulas (DP and BS only)

DP				
Size	n	Median	μ	σ
200	12	2.8s	28.09s	83.28s
220	12	3.8s	1098.22s	3612.62s
240	11	5.1s	5.31s	1.07s
260	12	6.6s	6.48s	0.38s
280	6	7.9s	7.98s	0.43s
300	9	9.7s	1609.00s	4497.72s
320	7	12.0s	422.81s	1004.49s
340	8	15.2s	2683.53s	7059.76s
360	7	17.2s	224.80s	507.49s

GSAT				
Size	n	Median	μ	σ
200	12	1.3s	7.58s	9.36s
220	12	7.3s	7.78s	8.01s
240	11	8.7s	7.54s	6.69s
260	12	10.6s	14.31s	16.79s
280	6	12.9s	15.83s	15.69s
300	9	14.7s	12.38s	7.33s
320	7	18.5s	26.09s	17.93s
340	8	20.2s	22.94s	19.13s
360	7	23.1s	48.10s	44.44s

D.3 Times for the n-queens Problem

BS				
Size	n	Median	μ	σ
5	1	8.4s	8.40s	-
6	1	58.2s	58.20s	-
7	1	226.9s	226.90s	-
8	1	1535.2s	1535.20s	-
9	1	2216.5s	2216.50s	-
10	1	10354.3s	10354.30s	-
11	1	12548.2s	12548.20s	-

DP Variant 1				
Size	n	Median	μ	σ
5	1	0.3s	0.30s	-
6	1	6.6s	6.60s	-
7	1	6.7s	6.70s	-
8	1	304.5s	304.50s	-
9	1	320.6s	320.60s	-
10	1	1611.2s	1611.20s	-
11	1	1822.4s	1822.40s	-
12	1	23019.7s	23019.70s	-

DP Variant 2				
Size	n	Median	μ	σ
5	1	0.2s	0.20s	-
6	1	5.1s	5.10s	-
7	1	2.8s	2.80s	-
8	1	41.3s	41.30s	-
9	1	103.8s	103.80s	-
10	1	426.1s	426.10s	-
11	1	542.2s	542.20s	-
12	1	4943.8s	4943.80s	-

DP Variant 3				
Size	n	Median	μ	σ
5	1	0.2s	0.20s	-
6	1	3.2s	3.20s	-
7	1	5.0s	5.00s	-
8	1	31.0s	31.00s	-
9	1	40.0s	40.00s	-
10	1	803.1s	803.10s	-
11	1	2554.2s	2554.20s	-
12	1	33701.2s	33701.20s	-

DP				
Size	n	Median	μ	σ
5	1	0.1s	0.10s	-
6	1	2.6s	2.60s	-
7	1	2.8s	2.80s	-
8	1	50.4s	50.40s	-
9	1	56.6s	56.60s	-
10	1	457.2s	457.20s	-
11	1	606.1s	606.10s	-
12	1	5141.3s	5141.30s	-

GSAT				
Size	n	Median	μ	σ
5	2	3.3s	1.75s	1.55s
6	2	68.5s	39.60s	28.90s
7	2	33.2s	19.90s	13.30s
8	2	854.7s	537.35s	317.35s
9	2	1812.2s	932.05s	880.15s
10	2	23823.8s	13020.70s	10803.10s
11	1	34030.3s	34030.30s	-

D.4 Times for the Subgraph-Isomorphism Problem

BS				
Size	n	Median	μ	σ
2	5	0.2s	28.68s	35.63s
3	3	0.7s	1.57s	1.23s
4	3	6.2s	5.63s	2.36s
5	3	8.2s	9.57s	2.22s
6	3	67.0s	62.27s	6.69s
7	3	98.5s	454.37s	546.46s
8	3	351.6s	422.80s	216.51s

DP				
Size	n	Median	μ	σ
2	5	0.1s	0.08s	0.04s
3	3	0.2s	0.27s	0.09s
4	3	0.7s	1.03s	0.47s
5	3	2.1s	2.47s	0.52s
6	3	17.4s	29.33s	22.24s
7	3	86.9s	119.33s	51.19s
8	3	1411.5s	2401.40s	1594.88s

GSAT				
Size	n	Median	μ	σ
2	5	0.1s	0.12s	0.04s
3	3	1.0s	0.80s	0.28s
4	3	1.9s	21.57s	27.81s
5	3	35.3s	42.30s	9.90s
6	3	192.0s	142.40s	84.68s
7	3	236.1s	366.63s	190.14s
8	3	1884.8s	4229.90s	3384.94s

D.5 Times for the Clique Problem

BS				
Size	n	Median	μ	σ
5	7	0.3s	0.30s	0.09s
6	7	0.7s	0.76s	0.18s
7	7	6.6s	6.56s	0.73s
8	7	14.0s	14.43s	1.35s
9	7	72.0s	70.17s	8.94s
10	7	127.3s	130.90s	8.67s
11	7	502.1s	524.31s	55.95s
12	7	835.0s	821.96s	46.27s

DP Variant 1				
Size	n	Median	μ	σ
5	7	0.0s	0.03s	0.05s
6	7	0.1s	0.11s	0.06s
7	7	0.9s	0.77s	0.32s
8	7	2.4s	2.27s	0.24s
9	7	9.2s	10.04s	1.23s
10	7	19.9s	19.69s	1.66s
11	7	75.1s	64.80s	26.58s
12	7	112.3s	97.74s	40.06s

DP Variant 2				
Size	n	Median	μ	σ
5	7	0.1s	0.06s	0.05s
6	7	0.1s	0.14s	0.12s
7	6	1.8s	1.33s	0.90s
8	5	5.0s	5.16s	0.81s
9	5	33.0s	36.24s	9.60s
10	6	72.9s	78.48s	23.87s
11	5	274.8s	237.50s	126.81s
12	7	560.6s	589.50s	340.63s

DP Variant 3				
Size	n	Median	μ	σ
5	7	0.1s	0.09s	0.03s
6	7	0.1s	0.16s	0.10s
7	7	1.3s	1.53s	1.03s
8	7	5.8s	5.94s	0.86s
9	7	34.2s	39.54s	10.76s
10	7	83.1s	89.39s	20.68s
11	7	294.2s	277.53s	139.42s
12	7	759.2s	761.51s	390.95s

DP				
Size	n	Median	μ	σ
5	7	0.1s	0.07s	0.05s
6	7	0.1s	0.14s	0.12s
7	6	2.3s	1.57s	1.11s
8	5	6.0s	6.06s	0.83s
9	5	34.8s	41.96s	12.15s
10	5	82.4s	91.44s	24.58s
11	5	293.5s	284.70s	165.85s
12	5	661.9s	691.28s	430.42s

GSAT				
Size	n	Median	μ	σ
5	7	0.0s	1.13s	2.28s
6	6	1.6s	2.77s	3.36s
7	5	0.7s	0.68s	0.13s
8	5	2.0s	8.72s	8.59s
9	5	12.7s	12.50s	1.57s
10	5	23.5s	42.14s	39.36s
11	5	80.4s	66.58s	33.44s
12	5	117.2s	342.76s	511.88s

D.6 Times for the Graph-Isomorphism Problem

BS				
Size	n	Median	μ	σ
5	2	18.1s	16.60s	1.50s
6	2	201.5s	191.15s	10.35s
7	2	658.6s	540.20s	118.40s
8	2	2956.7s	1753.30s	1203.40s

DP Variant 1				
Size	n	Median	μ	σ
5	2	1.0s	0.85s	0.15s
6	2	26.3s	15.05s	11.25s
7	2	136.1s	98.90s	37.20s
8	2	124.8s	84.25s	40.55s
9	1	662.8s	662.80s	-

DP Variant 2				
Size	n	Median	μ	σ
5	2	1.4s	0.80s	0.60s
6	2	12.5s	12.05s	0.45s
7	2	35.8s	19.80s	16.00s
8	2	627.3s	326.20s	301.10s
9	1	3373.6s	3373.60s	-

DP Variant 3				
Size	n	Median	μ	σ
5	2	1.3s	0.80s	0.50s
6	2	15.7s	14.50s	1.20s
7	2	45.9s	24.95s	20.95s
8	2	620.1s	323.20s	296.90s
9	1	2900.4s	2900.40s	-

DP				
Size	n	Median	μ	σ
5	2	1.4s	0.80s	0.60s
6	2	15.7s	15.60s	0.10s
7	2	46.2s	25.10s	21.10s
8	2	631.9s	329.15s	302.75s
9	1	2879.4s	2879.40s	-

GSAT				
Size	n	Median	μ	σ
5	2	5.1s	4.85s	0.25s
6	2	86.8s	62.95s	23.85s
7	2	732.1s	680.85s	51.25s
8	2	767.7s	607.85s	159.85s

Bibliography

[1] Billionnet, A. and Sutter, A. (1992). An efficient algorithm for the 3-satisfiability problem, Operations Research Letters vol. 12, July 1992, 29-36.

[2] Cormen, T. C., Leiserson, C.E., and Rivest, R.L., (1990) *Introduction to Algorithms*, The MIT Press, 1990.

[3] Davis. M., and Putnam, H., (1960) A Computing Procedure for Quantification Theory, *Journal of the ACM*, 1960, Vol 7, 201-215.

[4] Garey, M.R. and Johnson (1979), D.S., *Computers and Intractibility: A Guide to the Theory of NP-completeness*, W.H. Freeman and Company, 1979.

[5] Grimwald, R.P. (1989) *Discrete and Combinatorial Mathematics*, Addison-Wesley Publishing Company, 1989.

[6] Mitchell, D., Selman, B., and Levesque, H.J. (1992).
Hard and easy distributions of SAT problems. Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA, July 1992, 459-465.

[7] Purdom, P.W. and Brown, C.A. (1985).
The Analysis of Algorithms, Holt, Rinehart, and Winston, 1985.

[8] Sahni, S. (1985), *Concepts in Discrete Mathematics*, The Camelot Publishing Company, 1985.

[9] Selman, B., and Levesque, H.J., and Mitchell, D.G. (1992).
A New Method for Solving Hard Satisfiability Problems. Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA, July 1992, 440-446.

[10] Stamm-Wilbrandt, H. (1993), Programming in Propositional Logic or Reductions: Back to the Roots (Satisfiability), technical reports of Department of Computer Science III, University of Bonn, Germany, March 1993.

[11] Stamm-Wilbrandt, H. (1994), Usenet article in comp.theory, with `Subject: Re: Reduction from Subgraph Isomorphism to Satisfiability`,
`Date: 24 Jan 1994 11:28:28 GMT`,
`Message-ID: <2i0bcs$4rl@olymp.informatik.uni-bonn.de>`.

Vita

Daniel Angel Jiménez was born in Fort Hood, Texas to Dr. and Mrs. Angel R. Jiménez on September 1, 1969. After attending Tom C. Clark High School in San Antonio, Texas, he entered the University of Texas at San Antonio. He received his Bachelor of Science in Computer Science in August of 1992. He entered UTSA's graduate program in Computer Science that September.

Permanent address: 3600 Falls Creek
San Antonio, TX 78230

This thesis was typeset by Daniel A. Jiménez