An Implementation of the DPLL Algorithm

Tanbir Ahmed

A thesis

in

the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

July 2009

Your file Votre référence
ISBN: 978-0-494-63140-9
Our file Notre référence
ISBN: 978-0-494-63140-9

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Tanbir Ahmed

Entitled: An Implementation of the DPLL Algorithm

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. Rajagopalan Jayakumar

______________________________ Examiner
Dr. Peter Grogono

______________________________ Examiner
Dr. Clement Lam

______________________________ Supervisor
Dr. Vašek Chvátal

Approved by ______________________________
Chair of Department or Graduate Program Director

__________ ______________________________
Date Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

An Implementation of the DPLL Algorithm

Tanbir Ahmed

The satisfiability problem (or SAT for short) is a central problem in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification. Because of its inherent difficulty and widespread applications, this problem has been intensely being studied by mathematicians and computer scientists for the past few decades. For more than forty years, the Davis-Putnam-Logemann-Loveland (DPLL) backtrack-search algorithm has been immensely popular as a complete (it finds a solution if one exists; otherwise correctly says that no solution exists) and efficient procedure to solve the satisfiability problem. We have implemented an efficient variant of the DPLL algorithm. In this thesis, we discuss the details of our implementation of the DPLL algorithm as well as a mathematical application of our solver.

We have proposed an improved variant of the DPLL algorithm and designed an efficient data structure for it. We have come up with an idea to make the unit-propagation faster than the known SAT solvers and to maintain the stack of changes efficiently. Our implementation performs well on most instances of the DIMACS benchmarks and it performs better than other SAT-solvers on a certain class of instances. We have implemented the solver in the C programming language and we discuss almost every detail of our implementation in the thesis.

An interesting mathematical application of our solver is finding van der Waerden numbers, which are known to be very difficult to compute. Our solver performs the best on the class of instances corresponding to van der Waerden numbers. We have computed thirty of these numbers, which were previously unknown, using our solver.

Acknowledgements

I would like to express my hearty gratitude to my supervisor, Dr. Vašek Chvátal, for his immense help and encouragement. I am grateful to him for generously taking me as his student. This work would not have been possible without his invaluable guidance. He patiently read several drafts of the thesis, graciously helped me to put my thoughts in order, and directed me towards finishing it. He is more of a school than an individual.

I would like to cordially thank my committee members, Dr. Clement Lam, and Dr. Peter Grogono for carefully reading my thesis and giving their precious comments and suggestions. I especially want to thank Ehsan Chiniforooshan for reading part of my thesis and providing useful feedbacks.

I would like to thank Halina Monkiwewicz, Pauline Dubois, Edwina Bowen, Massy Joulani, and Hirut Adugna for their kind help in administrative matters in various times.

Finally, I thank Andalib, my wife, for her endless love, which has sustained me throughout.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Satisfiability problem

The satisfiability problem (or SAT for short) is a central problem in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification. Following is a definition of the satisfiability problem taken from Chvátal and Reed [10]:

> A *truth assignment* is a mapping f that assigns 0 (interpreted as FALSE) or 1 (interpreted as TRUE) to each variable in its domain; we shall enumerate all the variables in the domain as $x_1, \ldots, x_n$. The complement $\overline{x}_i$ of each variable x_i is defined by $f(\overline{x}_i) = 1 - f(x_i)$ for all truth assignments f. Both x_i and $\overline{x}_i$ are called *literals*; if $u = \overline{x}_i$ then $\overline{u} = x_i$. A *clause* is a set of (distinct) literals and a *formula* is a family of (not necessarily distinct) clauses. For example, $\{x_1, \overline{x}_2, x_3\}$ is a clause with three distinct literals and $\{\{x_1, x_2\}, \{x_1, \overline{x}_2\}, \{\overline{x}_1, x_2\}, \{\overline{x}_1, \overline{x}_2\}\}$ is a formula with four clauses over two variables.

A truth assignment *satisfies a clause* if it maps at least one of its literals to 1; the assignment *satisfies a formula* if and only if it satisfies each of its clauses. A formula is called *satisfiable* if it is satisfied by at least one truth assignment; otherwise it is called *unsatisfiable*. The problem of recognizing satisfiable formulas is known as the *satisfiability problem*, or SAT for short.

1.2 The DPLL algorithm

Given a formula F and a literal u in F, we let $F|u$ denote the *residual formula* arising from F when $f(u)$ is set to 1: explicitly, this formula is obtained from F by (i) removing all the clauses that contain u, (ii) deleting $\overline{u}$ from all the clauses that contain $\overline{u}$, (iii) removing both u and $\overline{u}$ from the list of literals.

Example: $F = \{\{x_1, x_2, x_3\}, \{\overline{x}_1, \overline{x}_2, x_4\}, \{x_1, x_3, x_5\}, \{\overline{x}_3, x_6\}\}$

$$F|x_1 = \{\{\overline{x}_2, x_4\}, \{\overline{x}_3, x_6\}\}$$

Algorithm 1.1 ALGORITHM DP_KERNEL(F)

$G = F$
while G includes a clause C such that $|C| \leqslant 1$ **do**
  if $C = \emptyset$ **then return** G
  else if $C = \{v\}$ **then** $G = G|v$
end while
while there is a monotone literal in G **do**
  $v =$ any monotone literal
  $G = G|v$
end while
return G

Trivially F is satisfiable if and only if at least one of $F|u$ and $F|\overline{u}$ is satisfiable. It is customary to refer to the number of literals in a clause as the *length* (rather than size) of the clause. Clauses of length one are called *unit clauses* If a formula F includes a unit clause $\{u\}$, then every truth assignment f that satisfies F must have $f(u) = 1$; hence F is satisfiable if and only if $F|u$ is satisfiable. A literal u in a formula F is called *monotone* if $\overline{u}$ appears in no clause in F. If u is a monotone literal and if F is satisfiable, then F is satisfied by a truth assignment f such that $f(u) = 1$. Hence F is satisfiable if and only if $F|u$ is satisfiable. These observations have been used by Davis and Putnam [19] in an algorithm for solving SAT. Their recursive applications transform any formula F into a formula G such that G is satisfiable if and only if F is satisfiable. G is referred in Ouyang [51] as DAVIS-PUTNAM KERNEL (the term was originally coined by Chvátal) of F. Algorithm 1.1 is the pseudocode for computing G, where the first `while` loop is referred as *unit-clause-propagation* and the second `while` loop is referred as *monotone-literal-fixing*. Monotone-literal-fixing does not create unit clauses; during this process, clauses containing monotone literals are removed entirely, leaving the other clauses unchanged. Davis, Logemann amd Loveland [18] use the the Davis-Putnam Kernel in an algorithm for testing satisfiability, which is called Davis-Putnam-Logemann-Loveland algorithm, or the DPLL algorithm (Algorithm 1.2) for short.

Each recursive call of DPLL may involve a choice of a literal u. Algorithms for making these choices are referred to as *branching rules*. Different branching rules are discussed in detail in section 2.1.

Algorithm 1.2 RECURSIVE ALGORITHM DPLL(F)

function DPLL(F)
    while F includes a clause C such that $|C| \leqslant 1$ **do**
        if $C = \emptyset$ **then return** UNSATISFIABLE
        else if $C = \{v\}$ **then** $F = F|v$
    end while
    if $F = \emptyset$ **then return** SATISFIABLE
    Choose a literal u using a branching rule
    if DPLL($F|u$) = SATISFIABLE **then return** SATISFIABLE
    if DPLL($F|\overline{u}$) = SATISFIABLE **then return** SATISFIABLE
    return UNSATISFIABLE
end function

1.3 Motivation and Scope

SAT is a very interesting problem both theoretically and practically. Cook [12] proved it to be NP-Complete [27]. We know that there is no deterministic algorithm that solves every SAT instance in polynomial time [27] unless P=NP. The speed of an implementation of a SAT algorithm like DPLL depends mostly on the branching rule, the data structure, and the search techniques used. Size of the DPLL-tree (as defined in section 2.1) varies greatly with branching rules and for a given branching rule, the speed of the implementation may significantly vary because of the data structure used to store and manipulate the formula. Again if we have a good implementation that performs well on most known instances, one can always come up with a new and challenging instance. All these demoralising circumstances can hardly stop us from writing another implementation with a new idea either in the branching rule or in the data structure or both. Sometimes, a tough instance motivates us to write our own version.

Many interesting search problems (Integer programming, Travelling Salesman

Problem, Graph Colouring, Subgraph Isomorphism, Subset Sum problem, etc.) that we encounter in our day-to-day lives and in the industry are NP-Complete [27]. Implementation of SAT introduces us to the solution techniques of those problems and enhances our understanding about them as well. In this thesis, we do the following:

(i) Survey various algorithms, known results and implementation techniques,

(ii) Implement DPLL with a fast data structure,

(iii) Compare the performance of our solver with other well-known solvers on popular benchmark instances,

(iv) Use the solver to compute some new van der Waerden numbers (section 4).

1.4 Organization of the thesis

The next four chapters are organized in the following manner:

CHAPTER 2: This chapter contains a detailed survey on the implementation aspects of DPLL. In section 2.1, we discuss the branching rules that can be described using a unified Paradigm (section 2.1.1): Dynamic Largest Individual Sum (DLIS), Dynamic Largest Combined Sum (DLCS), Jeroslow-Wang (JW), 2-Sided Jeroslow-Wang, DSJ rule, MOMS heuristics, Approximate Unit-Propagation Count (AUPC), and CSat rule. In section 2.2, we discuss well-known data structures: adjacency-lists (assigned literal hiding, counter-based approach, counter-based approach with satisfied clause hiding) and lazy data structures (Head-Tail lists and Two Literals

Watch method). In section 2.3, we discuss preprocessing techniques: adding resolvents, subsumption, and binary equivalent literal propagation. In section 2.4, we discuss pruning techniques: literal assignment forced by a pair of binary clauses, clause recording (implication graph, learning conflict clauses, random restarts), and non-chronological backtracking. In section 2.5, we discuss various features (for example, branching rule, data structure) of some modern SAT solvers like GRASP, SATZ, CHAFF, ZCHAFF and MINISAT.

CHAPTER 3: In this chapter, we describe the implementation of the solver in detail. It includes the DPLL pseudocode of our implementation, data-structure (storing variables and clauses, stack of changes, assignment information and other global variables and arrays) and details of functions (reducing the formula, reversing the changes, unit-propagation, branching, backtracking and backjumping). Code listings of different parts of the algorithm are in C.

CHAPTER 4: In this chapter, we investigate an interesting mathematical application of our SAT solver. Using a suitable branching rule, we compute some previously unknown van der Waerden numbers.

CHAPTER 5: In this chapter, we summarize the work we have done and the work we have not done. We also discuss possible future improvements on the solver.

APPENDIX A: Here, we describe some easily verifiable counting conditions under which a formula is satisfiable. In each case, we discuss the condition, an efficient algorithm to find a satisfying assignment, and optimality of the condition.

APPENDIX B: Here, we describe known deterministic k-SAT (in a *k-SAT* problem, every clause is of length k) algorithms other than DPLL.

Chapter 2

Implementation aspects of DPLL

In this chapter, we describe some of the implementation aspects of the DPLL algorithm such as branching rules (defined in section 1.2), data structures, preprocessing and techniques used to dismiss parts of the search space. We also briefly describe features of some of the popular SAT solvers. This survey will introduce the reader to various implementation techniques of the DPLL algorithm and provide necessary background material for Chapters 4 and 5, which describe the main contribution of the thesis.

2.1 Branching rules

Branching rules used for choosing a literal to set to TRUE during the search represent a key aspect of backtrack search SAT algorithms [4, 26, 34, 35, 66]. Several heuristics have been proposed over the years, each striving for a tradeoff between the time it requires and its ability to reduce the amount of search [34]. In this section, we discuss a few well-known branching rules and in section 2.5, we discuss some others.

It is customary to represent each call of DPLL(F) by a node of a binary tree. By branching on a literal u, we mean calling DPLL($F|u$). If this call leads to a contradiction, then we call DPLL($F|\overline{u}$). Every node that is not a leaf has at least one child and may not have both the children. This tree is referred as *DPLL-tree* in the literature. The shape and size of the tree depends not only on the input formula F, but also on the branching rule.

2.1.1 A paradigm for branching rules

Here is a paradigm for branching rules introduced in Ouyang [51], which associates a weight $w(F, u)$ with each literal u and chooses a function Φ of two variables. The paradigm is this:

$\star$ Find a variable x that maximizes $\Phi(w(F, x), w(F, \overline{x}))$;
choose x if $w(F, x) \geqslant w(F, \overline{x})$ and choose $\overline{x}$ otherwise.

If more than one variable maximizes Φ, then ties have to be broken by some rule. Usually, $w(F, u)$ is defined in terms of $d_k(F, u)$, which is the number of clauses of length k in F that contain literal u.

2.1.2 Branching rules that fit the paradigm

2.1.2.1 Dynamic Largest Individual Sum (DLIS)

This branching rule is ($\star$) with

$$\begin{aligned} w(F, u) &= \sum_k d_k(F, u), \\ \Phi(x, y) &= \max\{x, y\}. \end{aligned}$$

This is the default branching rule of GRASP [46, 47]

2.1.2.2 Dynamic Largest Combined Sum (DLCS)

This branching rule is ($\star$) with

$$\begin{aligned} w(F,u) &= \sum_k d_k(F,u), \\ \Phi(x,y) &= x+y. \end{aligned}$$

2.1.2.3 Jeroslow-Wang (JW) rule

This branching rule is ($\star$) with

$$\begin{aligned} w(F,u) &= \sum_k 2^{-k} d_k(F,u), \\ \Phi(x,y) &= \max\{x,y\}. \end{aligned}$$

This rule was proposed by Jeroslow and Wang [35].

2.1.2.4 2-Sided Jeroslow-Wang rule

This branching rule is ($\star$) with

$$\begin{aligned} w(F,u) &= \sum_k 2^{-k} d_k(F,u), \\ \Phi(x,y) &= x+y. \end{aligned}$$

This rule was proposed by Hooker and Vinay [34].

2.1.2.5 DSJ rule

This branching rule is $(\star)$ with

$$\begin{aligned} w(F,u) &= 4d_2(F,u) + 2d_3(F,u) + \sum_{k\geqslant 4} d_k(F,u), \\ \Phi(x,y) &= (x+1)(y+1). \end{aligned}$$

This rule was proposed by van Gelder and Tsuji in [65]

2.1.2.6 MOMS heuristics

This branching rule is $(\star)$ with

$$\begin{aligned} w(F,u) &= d_s(F,u), \\ \Phi(x,y) &= xy2^k + x + y. \end{aligned}$$

where s be the length of the smallest unsatisfied clause in F. MOMS is shorthand for Maximum Occurences on clauses of Minimum Size [26]. The value of k can vary, e. g., MOMS is used in SATZ[44] with $k = 10$.

Van Gelder and Tsuji [65] independently came up with MINLEN which is $(\star)$ with the same $w(F,u)$ as MOMS and $\Phi(x,y) = (x+1)(y+1)$, which is the same $\Phi(x,y)$ as MOMS with $k = 0$.

2.1.2.7 Approximate Unit-Propagation Count (AUPC) rule

This branching rule is $(\star)$ with

$$\begin{aligned} w(F,u) &= d_2(F,\overline{u}), \\ \Phi(x,y) &= xy + x + y. \end{aligned}$$

This rule was used In the solver BERKMIN [29]. Here, $w(F,u)$ counts the number of new unit clauses generated by setting u to TRUE. The reason for the word 'Approximate' in its name is that an actual unit-propagation is not conducted. The function $\Phi(x,y)$ is the same as MOMS with $k=0$.

2.1.2.8 CSat rule

This branching rule is $(\star)$ with

$$\begin{aligned} w_0(F,u) &= \sum_k \ln\left(1 + \frac{1}{4^k - 2^{k+1}}\right) d_k(F,u) \text{ and,} \\ w(F,u) &= w_0(F,u) + \sum_{\{u,v\}\in F} w_0(F,\overline{v}), \\ \Phi(x,y) &= x + y + 1.5 \cdot \min\{x,y\}\,; \end{aligned}$$

This rule was proposed by Dubois, Andre, Boufkhad, and Carlier in [22].

How good are branching rules in DPLL?

The following example [50] demonstrates how dramatically the choice of a branching rule can influence the size of the DPLL-tree. Take a formula with variables

$x_1, x_2, \ldots, x_n$ and clauses

1. $\{x_i, x_{n-1}, x_n\}$, $\{x_i, \overline{x}_{n-1}, x_n\}$, $\{x_i, x_{n-1}, \overline{x}_n\}$, $\{x_i, \overline{x}_{n-1}, \overline{x}_n\}$
   for $i = 1, 2, \ldots, n-2$,

2. $\{\overline{x}_j, \overline{x}_{j+1}, \ldots, \overline{x}_{n-3}, \overline{x}_{n-2}\}$ for $j = 1, 2, \ldots, n-3$.

This formula is unsatisfiable. Here, the size of the DPLL-tree branching on the variable with the smallest subscript is $2^{n-1} - 1$ and the size of the DPLL-tree branching on the variable with the biggest subscript is 7.

2.2 Data structures

The performance of a good implementation of the DPLL algorithm depends not only on the branching rule but also on the data structure. A survey of the data structures can be found in [45]. Here we describe some of the data structures used in some well-known SAT solvers.

2.2.1 Adjacency lists

Most implementations of the DPLL algorithm represent clauses as lists of literals and associate with each variable x a list of clauses that contain a literal in $\{x, \overline{x}\}$. In general, we use the term *adjacency lists* to refer to data structures in which each variable x contains a complete list of clauses that contain a literal in $\{x, \overline{x}\}$.

2.2.1.1 Assigned literal hiding

For each clause, three lists are maintained: unassigned, assigned TRUE and assigned FALSE. A clause is satisfied if one or more of these literals are assigned TRUE, unsatisfied if all its literals are assigned FALSE, and unit (current length equals one) if exactly one literal is unassigned and the remaining literals are assigned FALSE.

2.2.1.2 The counter-based approach

An alternative approach to keep track of unsatisfied, satisfied and unit clauses is to associate literal counters with each clause. For a clause C, let n_t and n_f be the number of literals assigned TRUE and FALSE, respectively. The clause C is unsatisfied if n_f equals $|C|$, satisfied if $n_t \geqslant 1$, and unit if $n_f = |C| - 1$ and $n_t = 0$. When a clause is declared unit, the list of literals is traversed to identify which literal needs to be set to TRUE. An example of a SAT solver that utilizes counter-based adjacency lists is GRASP[46].

2.2.1.3 Counter-based approach with satisfied clause hiding

When a literal is assigned a truth value, a potentially large number of clauses have to be traversed in order to be marked as satisfied. Some of these clauses may have been already satisfied by a previous assignment to some other literal. Hence, each time a clause C becomes satisfied, C is hidden from the list of clauses of all the literals that are contained in C. This technique was used in SCHERZO [14] to solve covering problems.

2.2.1.4 Counter-based approach with satisfied clause and assigned literal hiding

In addition to hiding satisfied clauses as described in section 2.2.1.3, literals that are assigned FALSE are hidden from the list of literals in clauses.

2.2.2 Lazy data structures

Adjacency list-based data structures share a common problem: each literal u keeps references to a potentially large number of clauses. Moreover, it is often the case that most of u's clause references need not be analyzed when u is assigned, since they do not become unit or unsatisfied. *Lazy data structures* are characterized by each literal keeping only a reduced set of clause references.

2.2.2.1 Head-Tail lists

The first lazy data structure proposed for SAT was the *Head-Tail lists* data structure, originally used in the SATO SAT solver [67]. Each clause maintains two references: the *head* and the *tail* references. Initially, in each clause, the first and the last literals are referenced as head and tail, respectively. Each literal u maintains two linked lists:

- list of clauses with literal u as the head reference and
- list of clauses with literal u as the tail reference.

If a literal u is set to `FALSE`, then

1. in each clause C containing u as the head reference, the solver looks for an unassigned literal in the direction of the tail reference such that

   (i). If a literal is found, which is set to `TRUE`, then the clause C is identified as satisfied and the search for an unassigned literal in C is stopped.

   (ii). If such a literal v is found, which is unassigned and is not the tail reference of C, then v becomes the new head reference of C. The corresponding literal references are updated.

   (iii). If such a literal v is found, which is unassigned and is the tail reference of C, then C is identified as a unit clause and the tail reference is identified as unit literal.

   (iv). If no such v is found, the tail reference is reached and the tail reference is assigned `FALSE`, then C is identified as unsatisfied.

2. in each clause C containing u as the tail reference, the solver looks for an unassigned literal in the direction of the head reference; and the above process is repeated.

When backtracking, recovering the previous references is necessary.

2.2.2.2 Two literal watch method

SAT solver CHAFF [49] proposed a new data structure called the *Two literal watch method*. Each clause maintains two references as *watched literals*. Each literal u maintains a list of clauses that contain u as one of the two watched literals.

If a literal u is set to FALSE, then for each clause C that contains u as a watched literal, the solver looks for a literal, which is not set to FALSE:

1. If such a literal v is found and v is assigned TRUE, then C is identified as satisfied.

2. If only such literal is v, which is unassigned and is not the other watched literal, then v becomes the new watched literal.

3. If only such literal is v, which is unassigned and is the other watched literal, then C is identified as unit clause and the other watched literal is identified as unit literal.

4. If no such v is found, then C is identified as unsatisfied.

When backtracking, recovering the references is not necessary.

2.3 Preprocessing the formula

In this section, we describe some preprocessing techniques, which if judiciously applied, significantly simplify the input formula before calling DPLL. Some of the operations may be integrated in the recursive DPLL algorithm as well.

2.3.1 Adding resolvents

Two clauses C_1 and C_2 are said to *clash* if there is exactly one literal u, such that $u \in C_1$ and $\overline{u} \in C_2$. If C_1 and C_2 clash, then their *resolvent* is defined as $C_1 \cup C_2 - \{u, \overline{u}\}$ and is denoted by $C_1 \nabla C_2$. If clauses C_1 and C_2 are satisfied by some truth assignment z, then their resolvent is also satisfied by z. Adding $C_1 \nabla C_2$ does not change the satisfiability status of the formula.

If C_1 and C_2 differ in only one literal u such that $u \in C_1$ and $\overline{u} \in C_2$, then C_1 and C_2 are called *neighbours*. Clauses, which are neighbours clash and their resolvent is a subset of both of them. If C_1 and C_2 are neighbours, then any truth assignment that satisfies $C_1 \nabla C_2$ will also satisfy both C_1 and C_2. So adding $C_1 \nabla C_2$ to the formula and removing C_1 and C_2 from the formula will not change the satisfiability status of the formula.

If an empty clause is obtained as a resolvent in a formula F, then F is unsatisfiable. Given an unsatisfiable formula, we can always generate a sequence of operations of adding resolvents that produces an empty clause. The latter observation is due to Robinson [53].

If we try to add resolvents corresponding to every pair of literals u and $\overline{u}$, then we may end up having too many clauses. At the same time, we will have many long clauses. So we often put restrictions on the maximum length of the resolvent and on the maximum number of resolvents to be added to the formula.

2.3.2 Subsuming clauses

If two nonempty clauses C_1 and C_2 are such that $C_2 \subseteq C_1$, then any truth assignment that satisfies C_2 will satisfy C_1. So C_1 can be removed from the formula without changing its satisfiability status. The operation is called *subsumption*, where C_2 *subsumes* C_1.

We can apply subsumption between every pair of clauses in the formula prior to initiating the search. This produces a very small number of subsumed clauses for most benchmarks. But a combination of adding resolvents and subsumption simplifies some instances. For instance, the clauses C_1 being $\{u_1, u_2, u_3\}$ and C_2 being $\{\overline{u}_1, u_2\}$, cannot subsume each other, but $C_1 \nabla C_2$ adds the clause $\{u_2, u_3\}$, which subsumes C_1.

2.3.3 Binary equivalent literal propagation

Let a formula F contain clauses $\{\overline{u}_1, u_2\}$ and $\{u_1, \overline{u}_2\}$. Since these two clauses are satisfied if and only if either $\{u_1 \mapsto \texttt{TRUE}, u_2 \mapsto \texttt{TRUE}\}$ or $\{u_1 \mapsto \texttt{FALSE}, u_2 \mapsto \texttt{FALSE}\}$, u_1 and u_2 are said to be *equivalent*. Hence all occurences of u_1 (respectively $\overline{u}_1$) can be substituted by u_2 (respectively $\overline{u}_2$), so that F having one variable less, is simplified. If $\{\overline{u}_1, u_3\}$ and $\{u_2, \overline{u}_3\}$ are clauses in F, then the first substitution changes $\{\overline{u}_1, u_3\}$ into $\{\overline{u}_2, u_3\}$ makes u_2 and u_3 equivalent. So, the equivalency relation can be propagated to simplify F.

2.4 Pruning techniques

In this section, we describe techniques that can be applied during search to reduce the size of the DPLL-tree.

2.4.1 Literal assignment forced by a pair of binary clauses

If F contains no unit clause but two binary clauses $\{u_1, u_2\}$ and $\{u_1, \overline{u}_2\}$, setting u_1 to FALSE leads to a conflict. So u_1 is forced and could be used to simplify F by computing $F|u_1$.

2.4.2 Clause recording

Given a set of variable assignments that leads to a conflict, a new clause is created that prevents the same assignments from occurring simultaneously again during the subsequent search. To create such a clause, an implication graph (as defined in the following section) has to be maintained during unit-propagation.

2.4.2.1 Implication graph

An *implication graph* is a directed acyclic graph where each vertex represents a variable assignment. A label $x = b@d$ of a vertex means the variable x is assigned a truth value b in $\{0, 1\}$ at decision level d. The decision level for all forced assignments is the same as that of the corresponding decison assignment in the unit-propagation. Let C contains literals u_i and u_j drawn from the variables x_i and x_j, respectively. If u_i is set to FALSE and u_j is the only unassigned literal when C becomes unit, then a directed edge from x_i to x_j, labelled by C, is added to the implication graph. Here,

C is called the *antecedent clause* of the literal corresponding to the variable x_j. In Figure 2.1 (taken from [47]), a subset of a SAT formula is shown.

Current truth assignment: $\{x_9 = 0@1, x_{12} = 1@2, x_{13} = 1@3, x_{10} = 0@4, x_{11} = 0@5, \cdots\}$
Current branching assignment: $\{x_1 = 1@8\}$

$C_1 = \{\overline{x}_1, x_2\}$
$C_2 = \{\overline{x}_1, x_3, x_9\}$
$C_3 = \{\overline{x}_2, \overline{x}_3, x_4\}$
$C_4 = \{\overline{x}_4, x_5, x_{10}\}$
$C_5 = \{\overline{x}_4, x_6, x_{11}\}$
$C_6 = \{\overline{x}_5, \overline{x}_6\}$
$C_7 = \{x_1, x_7, \overline{x}_{12}\}$
$C_8 = \{x_1, x_8\}$
$C_9 = \{\overline{x}_7, \overline{x}_8, \overline{x}_{13}\}$
....

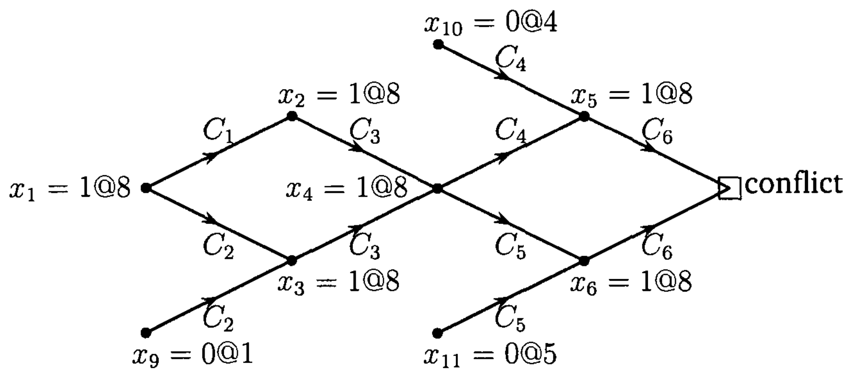

Figure 2.1: A TYPICAL IMPLICATION GRAPH

2.4.2.2 Learning conflict clauses

Conflict analysis relies on the implication graph to determine the reasons for the conflict. The conjunction of the decision assignments in the antecedent clauses in a unit-propagation is the reason for the conflict in that unit-propagation. By negating it, we obtain a clause, which is called *conflict clause.*

The clause learned can be used to prevent the same set of assignments from occuring again during the subsequent search. Inspecting the implication graph in Figure 2.1, we can readily conclude that the sufficient condition for this conflict to be identified is $(x_{10} = 0) \wedge (x_{11} = 0) \wedge (x_9 = 0) \wedge (x_1 = 1)$. In that case, the conflict clause learned is $\{\overline{x}_1, x_9, x_{10}, x_{11}\}$.

2.4.2.3 Random restarts and clause recording

To find a satisfying assignment quickly, some solvers (e. g., CHAFF, MINISAT) utilize a technique called *random restart* [30]. They want to avoid spending a long time searching in some unproductive branch of the DPLL-tree. Random restart periodically undoes all the decisions and restarts the search from the very beginning. Restarting is not a waste of the previous effort as long as the recorded clauses are still present.

2.4.3 Non-chronological backtracking

Clause recording is tightly associated with *non-chronological backtracking*, which is also known as *conflict-directed backjumping* [4].

The *chronological backtracking* search strategy always causes the search to consider the last, yet untoggled, decision assignment. By contrast, *non-chronological backtracking* may backtrack further up to a higher decision level. This technique was originally proposed by Stallman and Sussman [60] and further studied by Gaschnig [28] and others. It attempts to identify the conflict clauses and backtrack directly so that at least one of those variable assignments is modified. This technique was implemented initially by Bayardo and Schrag [4] and Silva and Sakallah [47].

Recorded clauses are used for computing the decision level to backtrack, which is defined as the most recent decision level of all variable assignments of the literals in each newly recorded clause. Figure 2.2 illustrates non-chronological backtracking on the same example as in Figure 2.1, with learned conflict clause C_{10} added.

$C_1 = \{\overline{x}_1, x_2\}$
$C_2 = \{\overline{x}_1, x_3, x_9\}$
$C_3 = \{\overline{x}_2, \overline{x}_3, x_4\}$
$C_4 = \{\overline{x}_4, x_5, x_{10}\}$
$C_5 = \{\overline{x}_4, x_6, x_{11}\}$
$C_6 = \{\overline{x}_5, \overline{x}_6\}$
$C_7 = \{x_1, x_7, \overline{x}_{12}\}$
$C_8 = \{x_1, x_8\}$
$C_9 = \{\overline{x}_7, \overline{x}_8, \overline{x}_{13}\}$
$C_{10} = \{x_9, x_{10}, x_{11}, \overline{x}_1\}$
....

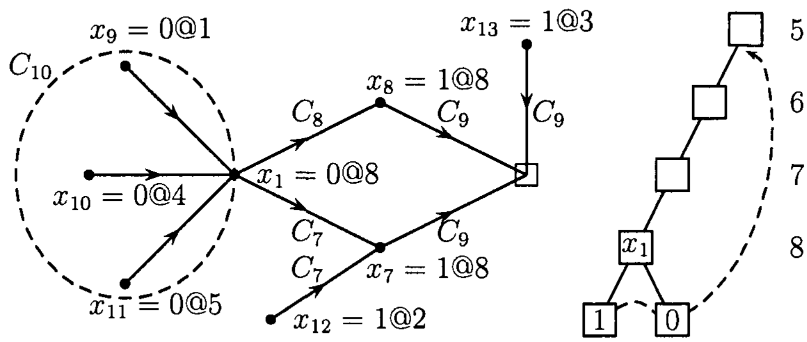

Figure 2.2: COMPUTING THE BACKTRACK LEVEL (SEE FIGURE 2.1)

Here, the new conflict clause is $\{x_9, x_{10}, x_{11}, \overline{x}_{12}, \overline{x}_{13}\}$. So the backtrack level is 5.

2.5 Well-known SAT solvers

In this section, we discuss some well-known SAT solvers with brief descriptions of some of their important features.

2.5.1 Satz

SATZ was developed by Li and Anbulagan [44].

(i) SATZ'S UPLA BRANCHING RULE: In SATZ, unit-propagation is integrated in the branching rule itself. The function UP(F) returns the resulting formula after running a unit-propagation.

Algorithm 2.3 SATZ - BRANCHING RULE

procedure SELECT ▷ Given a formula F
    for each unassigned variable x **do**
        Let F_1 and F_2 be two copies of F
        $F_1 = \text{UP}(F_1 \cup \{x\})$, $F_2 = \text{UP}(F_2 \cup \{\overline{x}\})$
        if $F_1 = \emptyset$ or $F_2 = \emptyset$ **then return** SATISFIABLE
        if both F_1 and F_2 contain an empty clause **then return** UNSATISFIABLE
        else if F_1 contains an empty clause **then** $x =$ FALSE and $F = F_2$
        else if F_2 contains an empty clause **then** $x =$ TRUE and $F = F_1$
        if neither F_1 nor F_2 contains an empty clause **then**
            $w(x) =$ number of binary clauses in F_1 but not in F
            $w(\overline{x}) =$ number of binary clauses in F_2 but not in F
        end if
    end for
    for each unassigned variable x in F **do**
        $\Phi(x) = w(\overline{x}) \times w(x) \times 1024 + w(\overline{x}) + w(x)$
    end for
    branch on the unassigned variable x such that $\Phi(x)$ is the highest
end procedure

Although solvers POSIT[26] and TABLEAU[16] used unit-propagation based branching rules, the real power of unit-propagation is integrated in SATZ[44] on top of MOMS heuristic.

(ii) SATZ'S PREPROCESSOR [43]: SATZ runs a loop with unit-propagation, binary equivalent literal propagation, adding resolvents of length at most 3, and using subsumption until F contains an empty clause or no change occurs in F.

Algorithm 2.4 SATZ - PREPROCESS

procedure PREPROCESS
    repeat
        unit-propagation
        if $\{u_1, \overline{u}_2\} \in F$ and $\{\overline{u}_1, u_2\} \in F$ **then**
            replace all occurences of u_1 (and $\overline{u}_1$) with u_2 (and $\overline{u}_2$ respectively)
            remove $\{u_1, \overline{u}_2\}$ and $\{\overline{u}_1, u_2\}$ from F
        end if
        if there are clause C_1 and C_2 s. t. they clash and $|C_1 \nabla C_2| \leqslant 3$ **then**
            add $C_1 \nabla C_2$ to F
        end if
        if $C_1 \subset C_2$ **then** $F = F - \{C_2\}$
    until F contains an empty clause or no change happens in F
    if F contains an empty clause **then return** UNSATISFIABLE
end procedure

2.5.2 GRASP

GRASP was developed by Silva and Sakallah [46, 47]. The name stands for Generic seaRch Algorithm for the Satisfiability Problem. GRASP views the occurence of a conflict as an opportunity to augment the problem description with conflict clauses. Conflict clauses are used to compute backtrack decision levels and recognize similar conflicts later in the search. The GRASP algorithm calls the recursive function SEARCH(d, β), which returns SUCCESS, i. e., SATISFIABLE if it finds a satisfying assignment, or else returns FAILURE, i. e., UNSATISFIABLE. Here, d, which is an input parameter, is the current decision level and β, which is an output parameter, saves the decision level to backtrack.

The recursive SEARCH function consists of four major operations:

(i) DECIDE(d): If all the clauses are satisfied, then this function returns SUCCESS. Otherwise, it chooses a decision assignment at level d of the search process. It chooses the variable and the assignment that directly satisfies the largest number of clauses.

(ii) DEDUCE(d): This function implements unit-propagation and implicitly maintains the resulting implication graph. It returns with a SUCCESS unless one or more clauses become unsatisfied. In the latter case, the implication graph is updated and a CONFLICT indication is returned.

(iii) DIAGNOSE(d, β): This function identifies the conflict clauses that can be added to the formula, as described in section 2.4, to avoid similar conflicts in future.

(iv) ERASE(): This function deletes the assignments at the current decision level.

The SEARCH function starts by calling DECIDE(d) to choose a variable assignment at decision level d. It then determines the consequences of this elective assignment by calling DEDUCE(d). If this assignment does not cause any clause to be unsatisfied, SEARCH is called recursively at decision level $d+1$. If a conflict arises due to this assignment, DIAGNOSE(d, β) function is called to analyze this conflict and to determine an appropriate decision level for backtracking the search. When SEARCH encounters a conflict, it returns with a CONFLICT indication and causes the elective assignment made on entry to the function to be erased.

Algorithm 2.5 GRASP ALGORITHM - SEARCH

```
function SEARCH(d, β)    ▷ d: current decision level;β: backtrack decision level
    if DECIDE(d) = SUCCESS then return SUCCESS
    while TRUE do
        if DEDUCE(d) ≠ CONFLCIT then
            if SEARCH(d + 1, β) = SUCCESS then return SUCCESS
            else if β ≠ d then ERASE() and return CONFLICT
        end if
        if DIAGNOSE(d, β)=CONFLICT then
            ERASE(), return CONFLICT
        end if
        ERASE()
    end while
end function
```

2.5.3 Chaff, zChaff

Moskewicz et al. [49] developed CHAFF, which efficiently implements DPLL with the following specific features:

(i) OPTIMIZED UNIT-PROPAGATION: In practice, for most SAT problems, approximately 90% of the solver's running time is spent in unit propagation. CHAFF implements an efficient unit-propagation engine. It maintains a counter of the number of unassigned literals for each clause. A clause is visited for the unit-clause-literal only when the counter is one. CHAFF uses the *watch literal* schme that was described in section 2.2.

(ii) BRANCHING RULE: CHAFF introduced the Variable State Independent Decaying Sum (VSIDS) branching rule mentioned in section 2.1. Each literal u is associated with a counter initialized to the number of clauses that contain u in the initial formula. When a clause is added to the formula, the counter associated with each literal in the clause is incremented. An unassigned literal with the highest counter is chosen for branching. Ties are broken arbitrarily. All counters are divided by a constant, say 2, (i. e., a decay of 50%) after every 1000 conflicts. So literals in older clauses drop in values over time, ensuring that recent clauses are satisfied first.

(*iii*) CONFLICT ANALYSIS: CHAFF employs a conflict resolution scheme that is very much similar to GRASP.

(*iv*) CLAUSE DELETION: CHAFF supports the deletion of added conflict clauses to avoid running out of memory. It uses scheduled lazy clause deletion. When each clause is added, it is examined to determine at what point in the future, if any, the clause should be deleted.

(*v*) RESTARTS: CHAFF employs the restart feature which clears all literal assignments but keeps the learned clauses. This policy helps to avoid the conflicts occured in the previous run.

zCHAFF implements the well known CHAFF [49] algorithm. It was the best complete solver in the SAT competition [54] in 2004 in the industrial category. It uses the VSIDS decision heuristic for branching, two-literal watch-lists for unit-propagation and conflict-driven clause learning along with non-chronological back-tracking for restructuring the DPLL-tree.

zCHAFF periodically deletes some learned clauses using usage statistics and clause lengths to estimate the usefulness of a clause.

2.5.4 MiniSat

MINISAT is an optimized CHAFF-like SAT solver written by Eén and Sörensson [58]. It is based on the two-literal watch-list for fast unit-propagation [49] and clause learning by conflict-analysis[47]. It entered the SAT Competition [54] in 2005 and was awarded Gold in three industrial categories and one of the crafted categories. Important features in MINISAT are:

(i) ORDER OF ASSIGNMENT: The decision heuristic of MINISAT is an improved VSIDS order, where variable activities are decayed 5% after each conflict. The original VSIDS decays 50% after each 1000 conflicts. Empirically, this performs better than the original VSIDS. To keep the variables sorted by activity at all times, a heap is used.

(ii) BINARY CLAUSES: Binary clauses are implemented by storing the literal to be propagated directly in the watch list. This scheme outperforms the version storing all binary clauses in separate set of vectors on the side.

(iii) CLAUSE DELETION: MiniSat deletes learned clauses more aggressively than the other solvers like CHAFF on an activity heuristics. The limit on how many learned clauses are allowed is increased after each restart. Keeping the number of clauses low seems to be important for some small but hard instances.

(iv) CONFLICT CLAUSE MINIMIZATION: For each literal u in a newly formed conflict clause C, it checks each antecedent clause C' of $\overline{u}$ to possibly find a neighbour of C. Then $C \nabla C'$ subsumes C and u is removed from the conflict clause.

Chapter 3

Our implementation of the solver

In this chapter, we describe the way we have implemented the DPLL algorithm with details of its data-structure and functions. Code is in C and lines of code are numbered for reference in the description. Before going into the pseudocode of our implementation, we present a revised version (Algorithm 3.6) of Algorithm 1.2, which avoids unnecessary work wherever possible.

3.1 Revised DPLL algorithm

We have the following observations:

1. We get an empty clause in $F|u$ only if $\overline{u}$ is in a clause of length one in F. So during unit-clause-propagation, for every new unit-clause-literal u, we avoid computing $F|u$ and return UNSATISFIABLE when F contains both $\{u\}$ and $\{\overline{u}\}$.

2. We can learn clauses to compute backtrack levels and restructure the DPLL-tree remaining in the original framework of the DPLL algorithm.

In Algorithm 3.6, Global variables `depth` and `backtrack_level` are used to restructure the DPLL-tree. Details of these variables are discussed in section 3.3.4.

Algorithm 3.6 REVISED DPLL ALGORITHM

```
function DPLL(F)
    while TRUE do
        if there are contradictory unit clauses then return UNSATISFIABLE
        else if there is a clause {v} then F = F|v
        else break
    end while
    if F = ∅ then return SATISFIABLE
    choose a literal v (using a branching rule)
    if DPLL(F|v)=SATISFIABLE then return SATISFIABLE
    C1 = conflict clause learned
    if backtrack_level ⩾ depth-1 then
        if DPLL(F|v̄)=SATISFIABLE then return SATISFIABLE
        C2 = conflict clause learned
        update backtrack_level using C1 and C2
    end if
    return UNSATISFIABLE
end function
```

3.2 DPLL pseudocode of our implementation

Algorithm 3.7 is the version of DPLL that we have implemented. The key functions are DPLL() itself, SETVAR(v) (computes $F|v$) and UNSETVAR(v) (recovers F from $F|v$). Ming Ouyang used the last two names in [51]. Our implementation of these functions with corresponding C code listings will be discussed in section 3.4. The function GETBRANCHINGVARIABLE chooses a yet-to-be-assigned variable for branching using a prescribed branching rule. One other important function which is used to add non-chronological backtracking (conflict-directed backjumping), is LEARNCONFLICTCLAUSE. In addition to them, there are basic stack functions to operate on locally and globally declared stacks.

Algorithm 3.7 THE DPLL PSEUDOCODE OF OUR IMPLEMENTATION

```
function DPLL()                                              ▷ runs on formula F
    while TRUE do
        if there are contradictory unit clauses then
            while ISSTACKEMPTY(local_unit_clauses_stack) = FALSE do
                u = STACKPOP(local_unit_clauses_stack)
                UNSETVAR(u)
            end while
            store the antecedent clauses to learn a conflict clause
            return UNSATISFIABLE
        else if there is a clause C = {u} then
            STACKPUSH(local_unit_clauses_stack, u)
            SETVAR(u)
        else
            BREAK
        end if
    end while
    if F = ∅ then return SATISFIABLE
    v = GETBRANCHINGVARIABLE(branching_rule)
    SETVAR(v)
    if DPLL() = SATISFIABLE then return SATISFIABLE
    UNSETVAR(v)
    C1 = LEARNCONFLICTCLAUSE
    if backtrack_level ⩾ depth − 1 then
        SETVAR(−v)
        if DPLL() = SATISFIABLE then return SATISFIABLE
        UNSETVAR(−v)
        C2 = LEARNCONFLICTCLAUSE
        Update backtrack_level using C1 and C2
    end if
    while ISSTACKEMPTY(local_unit_clauses_stack) = FALSE do
        u = STACKPOP(local_unit_clauses_stack)
        UNSETVAR(u)
    end while
    return UNSATISFIABLE
end function
```

3.3 The data structure

It is not obvious how to best represent a formula F such that Algorithm 3.7 runs the fastest. In this section, we describe the data structures used to implement the algorithm in detail with the corresponding codes in C. While designing the data structure, we have identified the following areas of possible improvement:

1. UNIT CLAUSE PROPAGATION: Since we spend most of the time during the search in unit-propagation, it is a good idea to perform all basic operations required for unit-propagation in as little time as possible.

2. RECORDING AND REVERSING CHANGES: Each time we compute $F|v$ from F, we make certain changes to the formula. When we backtrack, we have to recover F from $F|v$ reversing all those changes. It is important to record the changes in such a way that the reversal process remains inexpensive.

We make the following assumptions (preprocessor takes care of them) about the formula:

1. *The formula contains no empty clause.* If the initial formula does not contain an empty clause, then it never generates an empty clause during the search.

2. *The maximum length of a clause is 32* (number of bits in the microprocessor). If there is a clause longer than 32, then the preprocessor replaces them with smaller clauses introducing new variables. This assumption is necessary for faster retrieval of unit-clause-literal when a clause becomes unit.

3.3.1 Storing literals and clauses

Throughout the searching process, we need the list of all literals in each clause and the list of all clauses each literal is in. Structure literal_info maintains information specific to a literal.

```
typedef struct literal_info{
  int is_assigned;
  int n_occur;
  int * lit_in_clauses;
  int * lit_in_clause_locs;
  int is_unit;
  int antecedent_clause;
}literal_info;

literal_info linfo[MAX_VARS][2];
```

The field is_assigned which is either YES or NO helps to maintain the list of free (unassigned) literals during runtime. Fields n_occur, lit_in_clauses and lit_in_clause_locs store respectively the count, list of clauses in the original formula that contain the literal and the list of locations of the literal in the corresponding clauses. Variable linfo is a global array where linfo[j][SATISFIED] stores the information related to literal j: if it is assigned, number of times j occurs in F, list of clauses that contain j and the list of locations of the literal in the corresponding clauses. Similarly, linfo[j][SHRUNK] stores the information related to literal -j. When literal k becomes the only unassigned literal in a clause C, the is_unit field of k is set to YES and C is recorded in the antecedent_clause field.

Structure clause_info maintains information specific to a clause.

```
typedef struct clause_info{
  int * literals;
  int current_length;
  int original_length;
  int is_satisfied;
  int binary_code;
  int current_ucl;
}clause_info;

clause_info * clauses;
int n_clause, r_clauses;
```

The field literals stores the list of all literals in the clause. The original and the current lengths of the clause are stored in fields original_length and current_length respectively. When a literal in the clause is set to FALSE, the current_length decreases by one. The is_satisfied field is NO if the clause is not satisfied (i. e., none of its literals is set to TRUE) and YES otherwise. The field binary_code holds an integer, the binary encoding of which corresponds to the bit-string obtained from the literals setting '1' if UNASSIGNED and '0' if FALSE. The field current_ucl stores the unit-clause-literal if the clause has become unit and stores zero otherwise. Global array clauses stores the clauses and they remain in the memory throughout the search. No clause is physically removed from the formula when satisfied, only the is_satisfied field is set to TRUE. Variables n_clauses and r_clauses hold the original and current size (number of clauses) of the formula respectively.

3.3.2 Stack of Changes

The following structure keeps track of changes made while computing the residual formula and is used when the changes are needed to be reversed.

```
typedef struct changes_info{
  int clause_index;
  int literal_index;
}changes_info;

changes_info changes[MAX_CLAUSES];
unsigned int changes_index;
unsigned int n_changes[MAX_VARS][2];
```

When the `is_satisfied` field of a clause is changed to YES, the clause-index is saved. When a currently unassigned literal in a clause is set to `FALSE`, both the clause-index and the literal-index are saved in `clause_index` and `literal_index` respectively, so that they can be directly accessed when reversal of the changes is needed. The global array `changes` stores all the changes and is indexed by `changes_index`.

Variables `n_changes[depth][SATISFIED]` and `n_changes[depth][SHRUNK]` store respectively the number of clauses satisfied (or resolved) and the number of clauses shrunk at level `depth` in the branching tree while computing residual formula with the corresponding literal at that level. They are used while changes need to be reversed.

3.3.3 Storing assignment information

For each variable we store the current assignment information through the following structure.

```
typedef struct assign_info{
  int type;
  int depth;
  int decision;
}assign_info;

assign_info assign[MAX_VARS];
```

When a literal is assigned a value, `TRUE` or `FALSE`, the value is stored in the field `type` and the depth at which the assignment is made is stored in `depth`. The field `decision` stores `ASSIGN_BRANCHED` or `ASSIGN_IMPLIED` depending on whether the literal was chosen by a branching decision or was forced to have an assignment. By default, the field `decision` stores `ASSIGN_NONE`. In addition to storing assignment information, this structure is used to compute backtracking levels for non-chronological backtracking.

3.3.4 Other global variables and arrays

There are global variables that play crucial roles in the search process, which are discussed below:

(i) Variables `contradictory_unit_clauses` and `conflicting_literal`: when literals x and $\overline{x}$ are the only unassigned literals in two different yet-to-be-satisfied clauses, the variable `contradictory_unit_clauses` is set to YES. This saves an unnecessary unit-propagation, which would end up with an empty clause. If this variable is set to YES, then we return `UNSAT`. One of the conflicting literal is stored in `conflicting_literal`.

(*ii*) Array `gucl_stack` and variable `n_gucl`: when a new unit clause is detected, the unit-clause-literal is stored in global array `gucl_stack` which implements a stack of size `n_gucl`. Element popped from this stack is used for unit-clause-propagation when there are no contradictory unit clauses.

(*iii*) Variables `depth`, `backtrack_level` and `max_depth`: the variable `depth` stores the level of a node in the branching tree. (The `depth` level of the node is at most `backtrack_level`). Variable `backtrack_level` is usually one less than `depth`, but it can be equal to `depth` when `depth` equals to zero and for `depth` greater than one, the difference can be more than one indicating that a conflict-directed backjumping is necessary. Variable `max_depth`, local to `dpll`, is used to track non-chronological backtracking.

(*iv*) Array `impl_clauses` and `ic_cnt`: array `impl_clauses` and variable `ic_cnt` are used to store the antecedent clauses in an unit-propagation that leads to a contradiction.

3.4 Details of functions

In this section, we discuss key functions and procedures we use to implement DPLL. Most of the functions have a pseudocode followed by numbered code-listing in C and description of the code.

3.4.1 SetVar - computing $F|v$

Algorithm 3.8 shows the pseudocode for implementation of the SETVAR procedure.

Algorithm 3.8 DPLL - SETVAR

```
procedure SETVAR(v)
    for each yet-to-be-satisfied clause C such that v ∈ C do
        mark C as satisfied
        update clause count for C
        save changes information for v
    end for
    for each yet-to-be-satisfied clause C such that v̄ ∈ C do
        set v̄ as FALSE and update the length of C
        if |C| = 1 then find the literal ℓ in C that is unassigned
        if ℓ̄ is also a unit-clause-literal then mark contradictory unit clauses
        save changes information for v̄
    end for
    update depth and backtrack level
    remove v and v̄ from the list of unassigned literals
end procedure
```

The following code listing shows the C implementation of SETVAR.

```
void SetVar(int v)
{
  register int i;
  register int p = abs(v), q = (v>0) ? SATISFIED : SHRUNK;
  for(i=0; i<linfo[p][q].n_occur; ++i)
  {
    register int j = linfo[p][q].lit_in_clauses[i];
    if(clauses[j].is_satisfied) continue;
    clauses[j].is_satisfied = YES;
    --r_clauses;
    changes[changes_index++].clause_index = j;
    n_changes[depth][SATISFIED]++;
  }
  q = !q;
```

```
15      for(i=0; i<linfo[p][q].n_occur; ++i)
16      {
17        register int j = linfo[p][q].lit_in_clauses[i];
18        if(clauses[j].is_satisfied) continue;
19        register int k = linfo[p][q].lit_in_clause_locs[i];
20        --clauses[j].current_length;
21        clauses[j].binary_code -= ((1 << k));
22        changes[changes_index].clause_index = j;
23        changes[changes_index++].literal_index = k;
24        n_changes[depth][SHRUNK]++;
25        if(clauses[j].current_length == 1)
26        {
27          register int loc = int(log2(clauses[j].binary_code));
28          register int w = clauses[j].literals[loc];
29          register int s = abs(w), t = (w>0) ? SATISFIED : SHRUNK;
30          linfo[s][t].antecedent_clause = j;
31          if(linfo[s][(!t)].is_unit == YES)
32          {
33            contradictory_unit_clauses = TRUE;
34            conflicting_literal = w;
35          }
36          else if(linfo[s][t].is_unit == NO)
37          {
38            gucl_stack[n_gucl] = clauses[j].current_ucl = w;
39            linfo[s][t].is_unit = YES;
40            ++n_gucl;
41          }
42        }
43      }
44      if(depth && backtrack_level == depth-1)
45        ++backtrack_level;
46      ++depth;
47      linfo[p][SATISFIED].is_assigned = YES;
48      linfo[p][SHRUNK].is_assigned = YES;
49    }
```

Different parts of the function SetVar work as follows:

5-13 This for loop implements lines 2-5 of Algorithm 3.8. It scans through the lit_in_clauses list associated with the literal v. In each iteration, we retrieve a clause, say C, that contains the literal v (in $O(1)$ time). The is_satisfied field of C is set to YES and the size of the formula, r_clauses is decremented by one. All changes are saved in the changes list, and the number of changes made in this phase of action (which is stored in n_changes[depth][SATISFIED]) is incremented by one.

15-43 This for loop implements lines 7-12 of Algorithm 3.8. It scans through the lit_in_clauses list associated with the literal -v. In each iteration, we retrieve a clause, say C, that contains the literal -v (in $O(1)$ time). The binary_code field of the clause C is initially an integer $2^{|C|} - 1$, which is a bitstring of $|C|$ 1's. If -v is the k-th ($k \in \{0, 1, \cdots, |C| - 1\}$) literal in C, then we subtract $(1 \ll k)$ (shift operations are constant-time) from binary_code of C. When $|C| = 1$, we know that binary_code equals 2^t for some non-negative integer t less than 32. We can compute, in time $O(1)$, the integer $\log_2$(binary_code) which is the location of the unit-clause-literal, say w, in C. The clause that becomes unit is saved in the antecedent_clause field of the corresponding entry of the linfo list. If -w is also a unit-clause-literal, then contradictory_unit_clauses is set to YES and w is recorded as the conflicting_literal. Otherwise, w is saved both in gucl_stack and in the current_ucl field of C and w is identified as a unit-clause-literal. All changes are saved in the changes list, and the number of changes made in this phase of action (which is stored in n_changes[depth][SHRUNK]) is incremented by one.

44-48 These lines implement lines 13-14 of Algorithm 3.8. Once the residual formula is obtained, `depth` is incremented by one and the `backtrack_level` is updated. Finally, both the literals `v` and `-v` are identified as assigned. Therefore, we have computed the residual formula `F|v`.

3.4.2 UnSetVar - recovering F from $F|v$

Algorithm 3.9 shows the pseudocode for implementation of the UNSETVAR procedure.

Algorithm 3.9 DPLL - UNSETVAR

procedure UNSETVAR(v)
  update depth and backtrack level
  while the stack-of-changes for $\overline{v}$ is not empty **do**
    retrieve the clause C and the corresponding literal-index
    increment length of C by 1
    if the literal was set as unit **then** undo it
    update binary code of C
  end while
  while the stack-of-changes for v is not empty **do**
    retrieve the clause C
    mark C as not satisfied
    increment the formula size by 1
  end while
  set v and $\overline{v}$ as unassigned
end procedure

Following is the C-code listing of UNSETVAR.

```
void UnSetVar(int v)
{
  register int i;
  register int p = abs(v), q = (v>0) ? SATISFIED : SHRUNK;
  --depth;
  if(depth && backtrack_level == depth)
```

```
7        --backtrack_level;
8      while(n_changes[depth][SHRUNK])
9      {
10       --n_changes[depth][SHRUNK];
11       register int j = changes[--changes_index].clause_index;
12       register int k = changes[changes_index].literal_index;
13       ++clauses[j].current_length;
14       if(clauses[j].current_length == 2)
15       {
16         int s = abs(clauses[j].current_ucl);
17         int t = (clauses[j].current_ucl > 0) ? SATISFIED : SHRUNK;
18         linfo[s][t].is_unit = NO;
19         clauses[j].current_ucl = 0;
20       }
21       clauses[j].binary_code += ((1 << k));
22     }
23     while(n_changes[depth][SATISFIED])
24     {
25       --n_changes[depth][SATISFIED];
26       register int j = changes[--changes_index].clause_index;
27       clauses[j].is_satisfied = NO;
28       ++r_clauses;
29     }
30     linfo[p][SATISFIED].is_assigned = NO;
31     linfo[p][SHRUNK].is_assigned = NO;
32   }
```

Different parts of the function `UnSetVar` work as follows:

5-7 These lines implement line 2 of Algorithm 3.9. We are now reversing all the changed made in level `depth-1`. The value of `depth` is decremented by one and the `backtack_level` is updated.

8-22 This `while` loop implements lines 3-8 of Algorithm 3.9. It runs through the stack of changes `n_changes[depth][SHRUNK]` times. In each iteration, we retrieve the clause C and the literal-index k in that clause from the `changes` list in $O(1)$ time and increment the clause-length by one. If $|C| = 2$, i. e., if C was a unit-clause, then information related to the unit-clause-literal is updated. The binary_code field of C is updated by adding 2^k, i. e., $1 \ll k$, to it.

23-29 This while loop implements lines 9-13 of Algorithm 3.9. It runs through the stack of changes n_changes[depth][SATISFIED] times. In each iteration, we retrieve the clause-number (which was set as satisfied) in $O(1)$ time, turn it back to not satisfied, and increment the formula size by one.

30-31 These lines implement line 14 of Algorithm 3.9. Both v and $\overline{v}$ are taken back to the list of unassigned literals.

3.4.3 The DPLL function

The dpll function has the prototype int dpll(); it does not receive any parameter and returns either SAT or UNSAT. Local array lucl_stack implements a stack of size n_lucl. This stack is a dynamically extendable list (it uses the C realloc function for allocation), which is freed when DPLL returns UNSAT. For convenience in describing the details of this function, we break down the code listing into parts and describe these parts separately.

3.4.3.1 Unit-propagation block

The while loop of lines 5-39 implements lines 2-16 of Algorithm 3.7.

```
1   int dpll()
2   {
```

```
3     int * lucl_stack = NULL;
4     register unsigned int n_lucl = 0;
5     while(1)
6     {
7       if(contradictory_unit_clauses)
8       {
9         icl_cnt = 0;
10        int cl = abs(conflicting_literal);
11        impl_clauses[icl_cnt++] = linfo[cl][SATISFIED].antecedent_clause;
12        impl_clauses[icl_cnt++] = linfo[cl][SHRUNK].antecedent_clause;
13        assign[cl].decision = ASSIGN_NONE;
14        while(n_lucl)
15        {
16          UnSetVar(lucl_stack[--n_lucl]);
17          register int s = abs(lucl_stack[n_lucl]);
18          register int t = lucl_stack[n_lucl]>0 ? TRUE : FALSE;
19          impl_clauses[icl_cnt++] = linfo[s][t].antecedent_clause;
20          assign[s].type = UNASSIGNED;
21          assign[s].decision = ASSIGN_NONE;
22        }
23        contradictory_unit_clauses = FALSE;
24        free(lucl_stack);
25        n_gucl = 0;
26        return UNSAT;
27      }
28      else if (n_gucl)
29      {
30        lucl_stack = (int*)realloc(lucl_stack,(n_lucl+1)*sizeof(int));
31        register int implied_lit = gucl_stack[--n_gucl];
32        lucl_stack[n_lucl++] = implied_lit;
33        assign[abs(implied_lit)].type = implied_lit>0 ? TRUE : FALSE;
34        assign[abs(implied_lit)].decision = ASSIGN_IMPLIED;
35        SetVar(implied_lit);
36        n_units++;
37      }
38      else break;
39    }
```

7-27 This `if` block implements lines 3-9 of Algorithm 3.7. If there is a pair of contradictory unit clauses, then this block is executed. We do not need to make a lookup for contradictory unit clauses. In fact, whenever the `is_unit` field of a literal `u` is already set to `TRUE` and `-u` becomes a unit-clause-literal, the global constant `contradictory_unit_clauses` is set to `TRUE`. Lines 10-12 retrieve the antecedent clauses of the conflicting literals and store them in the `impl_clauses` array. Each iteration of the `while` loop (14-22) pops a literal from `lucl_stack`, reverses the changes made by SETVAR and retrieves (to store in the `impl_clauses` array) the antecedent clause of that literal. Lines 23-26 set the `contradictory_unit_clauses` to `FALSE`, free the `lucl_stack`, set the global unit clauses stack `gucl_stack` as empty and return `UNSAT`.

29-37 This block implements lines 10-12 of Algorithm 3.7. When there is no pair of contradictory unit clauses, we look for unit-clause-literals in the global unit clauses stack `gucl_stack`. Unit-clause-literal popped from `gucl_stack` is pushed into the local stack `lucl_stack`. The fact that the assignment was forced is recorded by marking the unit-clause-literal as `ASSIGN_IMPLIED`. Finally, the residual formula is computed using that literal.

3.4.3.2 Branching

Lines 40-65 implement lines 17-22 of Algorithm 3.7.

```
40      if(!r_clauses) return SAT;
41      register int v = GetLiteral2SJW();
42      assign[abs(v)].type = v > 0 ? TRUE : FALSE;
43      assign[abs(v)].depth = depth;
44      assign[abs(v)].decision = ASSIGN_BRANCHED;
45      SetVar(v);
46      if(dpll()) return SAT;
```

```
47    UnSetVar(v);
48    assign[abs(v)].decision = ASSIGN_NONE;
49    register int max_depth = 0, i, j, k, m, left = FALSE;
50    if(icl_cnt)
51    {
52      while(icl_cnt)
53      {
54        i = impl_clauses[--icl_cnt];
55        k = clauses[i].original_length;
56        for(j=0; j < k; ++j)
57        {
58          m = abs(clauses[i].literals[j]);
59          if(assign[m].decision == ASSIGN_BRANCHED &&
60             assign[m].depth > max_depth)
61            max_depth = assign[m].depth;
62        }
63      }
64      left = TRUE;
65    }
```

40 This line implements line 17 of Algorithm 3.7. If the formula is empty, i. e., `r_clauses` is zero, then we return `SAT`.

41-46 These lines implement lines 18-20 of Algorithm 3.7. At this point, since there exist no contradictory unit clauses and there remain clauses to be satisfied, we choose a literal using the 2-sided-Jeroslow-Wang branching rule (any other branching rule could be accomodated by changing the single line 41) to be assigned as `TRUE`. The fact that the assignment was made by a branching decision is recorded by marking the literal as `ASSIGN_BRANCHED`. After computing the residual formula $F|v$, we proceed to the left child of the `DPLL`-tree by making a recursive call to `dpll`.

47-65 These lines implement lines 21-22 of Algorithm 3.7. If the left child of the

DPLL-tree returns UNSAT, then we recover F from $F|v$ by calling UnSetVar. The while loop (lines 52-63) looks at the literals that were assigned by a branching decision in the impl_clauses (the antecedent clauses of the unit-literals during unit-propagation) and update the variable max_depth with the assignment depth of the most recent branching decision.

3.4.3.3 Backtracking and backjumping

Lines 66-105 implement lines 23-35 of Algorithm 3.7.

```
66  ++n_backtracks;
67  if(backtrack_level >= depth-1)
68  {
69    assign[abs(v)].type = !assign[abs(v)].type;
70    assign[abs(v)].decision = ASSIGN_IMPLIED;
71    SetVar(-v);
72    if(dpll()) return SAT;
73    UnSetVar(-v);
74    assign[abs(v)].type = UNASSIGNED;
75    assign[abs(v)].decision = ASSIGN_NONE;
76    if(left && icl_cnt)
77    {
78      while(icl_cnt)
79      {
80        i = impl_clauses[--icl_cnt];
81        k = clauses[i].original_length;
82        for(j=0; j < k; ++j)
83        {
84          m = abs(clauses[i].literals[j]);
85          if(assign[m].decision == ASSIGN_BRANCHED &&
86             assign[m].depth > max_depth)
87            max_depth = assign[m].depth;
88        }
89      }
90      if(max_depth < backtrack_level)
91        backtrack_level = max_depth;
```

```
92        }
93      }
94      icl_cnt = 0;
95      while(n_lucl)
96      {
97        int z = lucl_stack[--n_lucl];
98        UnSetVar(z);
99        assign[abs(z)].type = UNASSIGNED;
100       assign[abs(z)].decision = ASSIGN_NONE;
101     }
102     free(lucl_stack);
103     contradictory_unit_clauses = FALSE;
104     return UNSAT;
105   }
```

Line 67 is used for diagnostic purposes. Lines 67-93 implement the `if` block (lines 23-29) of Algorithm 3.7.

69-72 These lines implement lines 24-25 of Algorithm 3.7. If backjumping is not suggested by the backtrack level, then we proceed to the right child of the DPLL-tree making a recursive call to `dpll` on the reduced formula $F|\overline{v}$.

The assignment decision of the literal is switched from `ASSIGN_BRANCHED` to `ASSIGN_IMPLIED` as it was forced.

73-89 These lines implement lines 26-27 of Algorithm 3.7. If the right child of the DPLL-tree returns `UNSAT`, then we recover F from $F|\overline{v}$ by calling `UnSetVar`. The `while` loop in lines 78-89 are identical to lines 52-63 of the `dpll` listing.

90-91 These lines implement line 28 of Algorithm 3.7. The backtrack level is updated.

95-104 These lines implement lines 30-34 of Algorithm 3.7. The changes made in the unit propagation immediately preceeding the decision to branch on v are reversed. Finally lucl_stack is freed, and UNSAT is returned.

3.4.4 Monotone literal fixing

After the unit-propagation block, every clause is of length at least two. At this point, we can look for literals that do not appear in their complemented form in the residual formula. Such literals are referred as monotone literals, as described in section 1.2. Setting a monotone literal u to TRUE has the following effects:

1. The clauses that contain u are removed.
2. No clause shrinks.
3. The satisfiability of the instance does not change (F is satisfiable if and only if $F|u$ is satisfiable).

If we want to add this feature to the solver, then we have to insert the following lines and the C code segments in Algorithm 3.7 and in the listing of dpll, respectively.

1 The following two lines should be inserted after line 4 of dpll listing. Here, ml_stack is a local array that implements a stack of size n_ml.

```
int * ml_stack = NULL;
int n_ml = 0;
```

2 The following 4 lines should be inserted after line 17 in Algorithm 3.7.

for each monotone literal u in F **do**
    STACKPUSH(local_monotone_literal_stack, u)
    SETVAR(u)
end for

The corresponding C listing that should be inserted after line 40 of dpll, immediately follows. It scans through all unassigned literals in the residual formula and if there is a monotone literal u, then it computes $F|u$ using SetVar.

```
for(int i=1; i<=n_vars; ++i)
{
  int x, y, u, C;
  x = y = 0;
  if(assign[i].decision == ASSIGN_NONE)
  {
    u = 0;
    for(int j=0; j<linfo[i][SATISFIED].n_occur; ++j)
    {
      C = linfo[i][SATISFIED].lit_in_clauses[j];
      x += 1-clauses[C].is_satisfied;
    }
    for(int j=0; j<linfo[i][SHRUNK].n_occur; ++j)
    {
      C = linfo[i][SHRUNK].lit_in_clauses[j];
      y += 1-clauses[C].is_satisfied;
    }
    if(x && !y) u = i;
    if(y && !x) u = -i;
    if(u)
    {
      ml_stack = (int*) realloc(ml_stack,(n_ml+1)*sizeof(int));
      ml_stack[n_ml++] = u;
      assign[abs(u)].type = u>0 ? TRUE : FALSE;
      assign[abs(u)].depth = depth;
      assign[abs(u)].decision = ASSIGN_IMPLIED;
      SetVar(u);
    }
  }
```

```
}
```

3 The following 4 lines should be inserted after line 29 in Algorithm 3.7. The C implementation of the `while` loop, which should be inserted after line 93 of `dpll` listing, follows immediately. For each literal v popped from `ml_stack`, we recover F from $F|v$.

```
while IsStackEmpty(local_monotone_literal_stack) = False do
    u = StackPop(local_monotone_literal_stack)
    UnSetVar(u)
end while
```

```
while(n_ml)
{
  int u = ml_stack[--n_ml];
  UnSetVar(u);
  assign[abs(u)].type = UNASSIGNED;
  assign[abs(u)].decision = ASSIGN_NONE;
}
```

3.4.5 Input cleanup and preprocessing

Our solver reads SAT instances in DIMACS satisfiability format [20]:

1. Comments may appear before the actual problem specification begins. Each comment line begins with a lower-case character `c`. These lines are ignored by the solver.

2. After the comments, there is a single line that specifies the instance. The line begins with a lower-case letter `p`, followed by the word `cnf` (to indicate that the problem is in Conjunctive Normal Form), the number n of variables (variables are the integers $1, 2, \cdots, n$), the number m of clauses; followed by the encoding of the m clauses. Each of the clauses is encoded by a list of integers followed by a zero (indicating the end of the clause). These integers are chosen from $\{1, 2, \cdots, n, -1, -2, \cdots, -n\}$ as literals and they appear in an arbitrary order. There may be redundant literals in a clause and redundant clauses in a formula. For instance, the file

   ```
   c This is a comment
   p cnf 4 2
   2 -1 4 0
   2 -3 0
   ```

   represents the formula with variables with variables x_1, x_2, x_3, x_4 that consists of the two clauses $\{\overline{x}_1, x_2, x_4\}$ and $\{x_2, \overline{x}_3\}$.

When reading the input file, we do the following:

1. We remove variables that do not occur in the formula from the list of free variables. We do so by storing `ASSIGN_REMOVED` in the `assign[u].decision`

for the variable u. When a satisfying assignment is found, these variables remain as "don't cares".

2. We remove clauses that contain a pair of literals u and $\overline{u}$, which makes the clause satisfied by any truth assignment.

3. We remove duplicate literals from a clause and store the remaining literals in sorted order.

4. We remove duplicate clauses.

5. If some clause C consists of literals $u_0, u_1, \cdots, u_k$ where $k \geqslant 32$, then we replace C by a set of clauses $\{C_0, C_1, \cdots, C_{t-1}\}$ with $t = \lceil (k-1)/30 \rceil$. These clauses involve $t-1$ new variables $y_0, y_1, \cdots, y_{t-2}$ and are defined as follows:

   (i). $C_0 = \{u_0, u_1, \cdots, u_{30}, y_0\}$,

   (ii). $C_i = \left\{\overline{y}_{i-1}, u_{30i+1}, \cdots, u_{30(i+1)}, y_i\right\}$ for $1 \leqslant i \leqslant t-2$, and

   (iii). $C_{t-1} = \left\{\overline{y}_{t-2}, u_{30(t-1)+1}, \cdots, u_k\right\}$.

   If C is not satisfied by a truth assignment z, i. e., the literals $u_0, u_1, \cdots, u_k$ are all set to `FALSE`, then $\{C_0, \cdots, C_{t-1}\}$ is not satisfied by z. Because, C_0 cannot be satisfied by setting y_0 to `FALSE` and if we set y_0 to `TRUE` to satisfy C_0, then C_{t-1} cannot be satisfied. If C is satisfied by z, then a literal u_j, with $0 \leqslant j \leqslant k$, must have been set to `TRUE` by z. If $u_j \in C_i$ for $0 \leqslant i \leqslant t-1$, then we can satisfy $\{C_0, \cdots, C_{t-1}\}$ by setting $y_0, y_1, \cdots, y_{i-1}$ to `TRUE` and $y_i, \cdots, y_{t-2}$ to `FALSE`.

3.4.5.1 Our preprocessor

Our preprocessor is described in Algorithm 3.10. Here, we implement unit-propagation, monotone-literal-fixing, restricted resolution and subsumption. We repeat them in this order until there is no change in F. The unit-propagation is similar to the unit-propagation block in section 3.4.3.1 except that there is no need to store the unit literals and the implication clauses. Fixing monotone literals is also same as described in section 3.4.4 except that there is no need to store the monotone literals.

Of course, other features like equivalency reasoning (see [43]) can be added to solve the DIMACS `pret` and `par` instances easily.

Algorithm 3.10 OUR PREPROCESSOR

```
function PREPROCESS(F)
    initialize threshold on the number of resolvents
    while TRUE do
        unit-propagation
        fix monotone literals
        add resolvents retricted by length and the threshold
        subsume clauses if you can
        if no changes occurs to F then break
    end while
end function
```

Adding resolvents

We compute length restricted-resolvents as in Algorithm 3.12. As defined earlier in section 2.3.1, two clauses C_1 and C_2 are said to *clash* on a variable x, if x is the only variable such that $x \in C_1$ and $\overline{x} \in C_2$ (or, $\overline{x} \in C_1$ and $x \in C_2$). In that case, the resolvent of C_1 and C_2 is defined as $C_1 \cup C_2 - \{x, \overline{x}\}$ and denoted by $C_1 \nabla C_2$.

Algorithm 3.11 COMPUTE RESOLVENTS

```
function COMPUTERESOLVENT(x, j, k, length, length_limit)
    C = {}
    for each i ∈ {j, k} do
        for each literal u in C_i do
            if x = abs(u) or u ∈ C then
                continue
            else if ū ∈ C then
                return false
            else
                C = C ∪ {u}
                if |C| > length_limit then return false
            end if
        end for
    end for
    length = |C|
    return true
end function
```

The following C listing implements Algorithm 3.11.

```
int compute_resolvent(int x, int a, int b, int & len, int limit)
{
  register int j, k;
  int * check = (int *)calloc(n_vars+1, sizeof(int));
  int found = FALSE;
  int res_size = 0;
  int C[2] = {a, b};
  for(j=0; j<2; ++j)
  {
    for(k=0; k<clauses[C[j]].original_length; ++k)
    {
      register int w = abs(clauses[C[j]].literals[k]);
      if(w == x) continue;
      else if(check[w] == clauses[C[j]].literals[k]) continue;
      else if(check[w] == -clauses[C[j]].literals[k])
      {
        free(check); return FALSE;
      }
```

```
19          else if(assign[abs(clauses[C[j]].literals[k])].decision
20                  != ASSIGN_NONE) continue;
21          else if(check[w] == 0)
22          {
23            check[w] = clauses[C[j]].literals[k];
24            resolvent[res_size++] = clauses[C[j]].literals[k];
25            if(res_size > limit)
26            {
27              free(check); return FALSE;
28            }
29          }
30        }
31      }
32      len = res_size;
33      free(check);
34      return TRUE;
35    }
```

Here, The local array `check` is used to detect duplicate and complemented literals while scanning through the two clauses to obtain a resolvent. To store a new resolvent, we use the global array `resolvent`, which is indexed by `res_size`.

8-31 These lines implement lines 3-14 of Algorithm 3.11. The parameter x is a variables such that x belongs to one of the clauses and $\overline{x}$ belongs to the other clause. We look at each literal of each clause C of the two clauses. If the current literal u is x or $\overline{x}$, then we continue. If `check[abs(`u`)]` equals to u, i. e., $u \in C$, then we continue; if `check[abs(`u`)]` equals to $\overline{u}$, i. e., another variable other than x appears in one of the clauses and appears complemented in the other clause, then we return `FALSE`. If `check[abs(`u`)]` equals to zero, then we store u in `check[abs(`u`)]` and `resolvent`. If `res_size` is bigger then `limit`, then we return `FALSE`.

32-34 Otherwise, we return `TRUE`.

Note that unit-propagation and monotone literal fixing make some clauses satisfied; this is the reason why we write "unsatisfied clauses" rather than "clauses" in lines 2 and 3 of Algorithm 3.12.

Here, `resolvents_added` and `n_resolvents_threshold` are global variables storing the number of resolvents added so far and the number of resolvents we are allowed to add, respectively.

Algorithm 3.12 GETTING RESTRICTED RESOLVENTS

```
1:  function GETRESOLVENTS(x, length_limit, F)
2:      for each unsatisfied clause C_j such that x ∈ C_j do
3:          for each unsatisfied clause C_k such that x̄ ∈ C_k do
4:              if COMPUTERESOLVENT(x, j, k, length, length_limit)=true then
5:                  if resolvents_added < n_resolvents_threshold then
6:                      add the resolvent to F
7:                  else
8:                      return false
9:                  end if
10:             end if
11:         end for
12:     end for
13: end function
```

The following C listing implements Algorithm 3.12.

```
int get_restricted_resolvent(int x, int limit)
{
  register int i, j, k, a, b, res_length;
  int found;
  changes_occured = FALSE;
  for(i=0; i<linfo[x][SATISFIED].n_occur; ++i)
  {
    a = linfo[x][SATISFIED].lit_in_clauses[i];
    if(clauses[a].is_satisfied == NO)
    {
      for(j=0; j<linfo[x][SHRUNK].n_occur; ++j)
      {
```

```
13          b = linfo[x][SHRUNK].lit_in_clauses[j];
14          if(clauses[b].is_satisfied == NO)
15          {
16            found = compute_resolvent(x, a, b, res_length, limit);
17            if(found)
18            {
19              if(resolvent_added < n_resolvents_threshold)
20              {
21                resolvent_added +=
22                add_a_clause_to_formula(resolvent, res_length);
23                changes_occured = TRUE;
24              }
25              else return -1;
26            }
27          }
28        }
29      }
30    }
31    return -1;
32  }
```

6-30 The nested `for` loops implement lines 2-12 of the algorithm 3.12. If a resolvent is found, i. e., `compute_resolvent` returns `TRUE`, then the clause stored in `resolvent`, which is of length `res_length`, is our length-restricted resolvent. If the resolvent-count does not exceed the threshold, then the resolvent is added to the formula; otherwise, we stop searching resolvents. The function `add_a_clause_to_formula(int [], int)`, which takes an array and its size as parameters and stores it into the data structure as a clause, is described in section 3.4.5.2.

Removal of subsumed clauses

Algorithm 3.13 takes two clauses C_j and C_k as parameters. It returns TRUE if $C_j \subset C_k$; and returns FALSE otherwise.

Algorithm 3.13 SUBSUMABLE

```
function SUBSUMABLE(j, k)
    for each literal u ∈ C_k do
        store u in check[abs(u)]
    end for
    for each literal u ∈ C_j do
        if u ≠check[abs(u)] then return false
    end for
    return true
end function
```

The C implementation of Algorithm 3.13 is as follows:

```c
int subsumable(int j, int k)
{
  register int i;
  int * check = (int *) calloc((n_vars+1), sizeof(int));
  for(i=0; i<clauses[k].original_length; ++i)
    check[abs(clauses[k].literals[i])] = clauses[k].literals[i];
  for(i=0; i<clauses[j].original_length; ++i)
    if(clauses[j].literals[i] != check[abs(clauses[j].literals[i])])
      { free(check); return NO; }
  free(check);
  return YES;
}
```

Here, we use local array check to mark the literals in C_k. For each literal u in C_k, we store u in check[abs(u)]. For each literal u in C_j, if u is not equal to check[abs(u)], then we return FALSE and else we return TRUE.

We remove subsumed clauses as described in Algorithm 3.14.

Algorithm 3.14 SUBSUMING CLAUSES

```
function SubsumeClauses(F)
    for each unassigned literal u do
        for each unsatisfied clause C_j such that u ∈ C_j do
            for each unsatisfied clause C_k such that u ∈ C_k do
                if j = k then continue
                if |C_j| ⩾ |C_k| then continue
                if Subsumable(j,k)=true then remove C_k from F
            end for
        end for
    end for
end function
```

Lines 1-33 implement Algorithm 3.14.

```
1   int preprocess_subsume()
2   {
3     register int n_subsumed = 0;
4     register int i, j, k, c1, c2, type;
5     changes_occured = FALSE;
6     for(i=1; i<=n_vars; ++i)
7     {
8       if(assign[i].decision != ASSIGN_NONE) continue;
9       for(type=0; type<=1; ++type)
10      {
11        for(j=0; j<linfo[i][type].n_occur; ++j)
12        {
13          for(k=0; k<linfo[i][type].n_occur; ++k)
14          {
15            if(j==k) continue;
16            c1 = linfo[i][type].lit_in_clauses[j];
17            c2 = linfo[i][type].lit_in_clauses[k];
18            if(clauses[c1].is_satisfied ||
19               clauses[c2].is_satisfied) continue;
20            if(clauses[c1].original_length >=
21               clauses[c2].original_length) continue;
22            if(subsumable(c1, c2))
```

```
23              {
24                clauses[c2].is_satisfied = YES;
25                --r_clauses;
26                n_subsumed++;
27                changes_occured = TRUE;
28              }
29            }
30          }
31        }
32      }
33    }
```

1-33 The function preprocess_subsume() scans through the literals and for each unassigned literal u, it picks each pair of distinct yet-to-be-satisfied clauses C_j and C_k that contain u. If $|C_j| < |C_k|$ and C_j subsumes C_k, then C_k is removed from F.

The following listing is the C implementation of Algorithm 3.10.

```
int preprocess()
{
  register int total_changes_occured, n_s = 0;
  if(n_clauses < 500) n_resolvents_threshold = n_clauses * 5;
  else if(n_clauses < 1000) n_resolvents_threshold = n_clauses * 4;
  else if(n_clauses < 1500) n_resolvents_threshold = n_clauses * 3;
  else if(n_clauses < 3000) n_resolvents_threshold = n_clauses * 2;
  else n_resolvents_threshold = n_clauses;
  while(1)
  {
    total_changes_occured = 0;
    if(preprocess_unit_propagation()==UNSAT)
    {
      printf("Resolvents: %d\n", resolvent_added);
      printf("Subsumed: %d\n", n_s);
      return UNSAT;
    }
    total_changes_occured += changes_occured;
```

```
19       preprocess_monotone_literal_fixing();
20       total_changes_occured += changes_occured;
21       if(resolvent_added < n_resolvents_threshold)
22       {
23         for(int i=1; i<=n_vars; ++i)
24           if(assign[i].decision == ASSIGN_NONE)
25             if(get_restricted_resolvent(i, 3)==UNSAT)
26             {
27               printf("Resolvents: %d\n", resolvent_added);
28               printf("Subsumed: %d\n", n_s);
29               return UNSAT;
30             }
31         total_changes_occured += changes_occured;
32       }
33       n_s += preprocess_subsume();
34       total_changes_occured += changes_occured;
35       if(total_changes_occured == 0) break;
36     }
37     printf("Resolvents: %d\n", resolvent_added);
38     printf("Subsumed: %d\n", n_s);
39     return -1;
40   }
```

3-8 These lines initialize the value of `n_resolvents_threshold`.

9-36 This `while` loop executes unit-propagation, monotone-literal-fixing, restricted resolution and subsumption in this order until no change occurs (maintained by the variables `changes_occured` and `total_changes_occured`) in F.

3.4.5.2 Adding a clause to the formula

Algorithm 3.15 describes what additions and updates we make in the data structure when we add a clause.

Algorithm 3.15 ADDING A CLAUSE TO F

function ADDCLAUSE(C, n)
    sort the array C
    if C is already in F **then return** false
    initialize `clauses` and store literals
    for each literal, update `linfo` structure
    if $n = 1$ **then**
        if $\{-C[0]\}$ is also a clause in F **then** set `contradictory_unit_clauses`
        else store $C[0]$ as a unit clause literal and update `gucl_stack`
    end if
end function

Following is the C implementation of Algorithm 3.15:

```
int add_a_clause_to_formula(int C[], int n)
{
  register int i;
  qsort (C, n, sizeof(int), compare);
  if(clause_present(C, n)) return FALSE;
  clauses = (clause_info *)realloc(clauses,
            (n_clauses+1)*sizeof(clause_info));
  clauses[n_clauses].is_satisfied = NO;
  clauses[n_clauses].current_length = n;
  clauses[n_clauses].original_length = n;
  clauses[n_clauses].binary_code = (((1<<(n-1))-1)<<1) + 1;
  clauses[n_clauses].current_ucl = 0;
  clauses[n_clauses].literals =
            (int *) malloc((n + 1) * sizeof(int));
  if(n>max_clause_len) max_clause_len = n;
  for(i=0; i<n; ++i)
  {
    int p = abs(C[i]), q = C[i]>0 ? SATISFIED : SHRUNK;
    linfo[p][q].lit_in_clauses =
```

```
20              (int *) realloc(linfo[p][q].lit_in_clauses,
21              (linfo[p][q].n_occur+1) * sizeof(int));
22        linfo[p][q].lit_in_clause_locs =
23              (int *)realloc(linfo[p][q].lit_in_clause_locs,
24              (linfo[p][q].n_occur+1) * sizeof(int));
25        linfo[p][q].lit_in_clauses[linfo[p][q].n_occur] = n_clauses;
26        linfo[p][q].lit_in_clause_locs[linfo[p][q].n_occur] = i;
27        linfo[p][q].n_occur++;
28        linfo[p][q].is_assigned = NO;
29        clauses[n_clauses].literals[i] = C[i];
30        assign[p].decision = ASSIGN_NONE;
31        assign[p].type = UNASSIGNED;
32      }
33      if(n == 1)
34      {
35        int s = abs(clauses[n_clauses].literals[0]);
36        int t = clauses[n_clauses].literals[0]>0 ? SATISFIED : SHRUNK;
37        linfo[s][t].antecedent_clause = n_clauses;
38        if(linfo[s][(!t)].is_unit == YES)
39        {
40            contradictory_unit_clauses = TRUE;
41            conflicting_literal = clauses[n_clauses].literals[0];
42        }
43        else if(linfo[s][t].is_unit == NO)
44        {
45            gucl_stack[n_gucl] = clauses[n_clauses].literals[0];
46            clauses[n_clauses].current_ucl=clauses[n_clauses].literals[0];
47            linfo[s][t].is_unit = YES;
48            ++n_gucl;
49        }
50      }
51      ++n_clauses;
52      ++r_clauses;
53      return TRUE;
54    }
```

4-15 These lines implement lines 2-4 of Algorithm 3.15.

16-32 The `for` loop line 5 of Algorithm 3.15.

33-50 These lines implement lines 6-9 of Algorithm 3.15.

The function `clause_present(int C[], int n)` is implemented as follows:

```
int clause_present(int C[], int n)
{
  register int i, j, k, p, q;
  p = abs(C[0]); q = C[0] > 0 ? SATISFIED : SHRUNK;
  for(j=0; j<linfo[p][q].n_occur; ++j)
  {
    if(clauses[linfo[p][q].lit_in_clauses[j]].original_length == n)
    {
      int match_count = 0;
      for(k=0; k<n; ++k)
      {
        if(clauses[linfo[p][q].lit_in_clauses[j]].literals[k]==C[k])
          match_count++;
        else break;
      }
      if(match_count == n) return TRUE;
    }
  }
  return FALSE;
}
```

If a clause C is a duplicate of some already existing clause C', then every literal in C must be in C' and the lengths must be equal. We pick the very first literal u in C and scan the clauses that contain u. If the number of matching literals in any of these clauses equals n, then C is duplicate and we return `TRUE`. Otherwise, we return `FALSE`. (Note that both C and C' are sorted.)

3.4.6 Branching rules

Not only is it important to select the right branching rule, but also it is necessary to have a fast implementation of it. Here, we describe a few branching rules that we have implemented in our solver. In section 3.5, we compare their performances with the aim of choosing a branching rule in our solver for further experiments. Throughout section 3.4.6, we let $d_k(F, u)$ be the number of yet-to-be-satisfied clauses of length k in F that contain u. As defined in section 2.1, with each literal u, we associate a weight function $w(F, u)$. We find a variable x that maximizes $\Phi(w(F, x), w(F, \overline{x}))$; and then we choose the literal x if $w(F, x) \geqslant w(F, \overline{x})$ and choose $\overline{x}$ otherwise.

3.4.6.1 Dynamic Largest Combined Sum (DLCS)

Here, $w(F, u)$ is the number of occurences of literal u in the unsatisfied clauses and $\Phi(s, t) = s + t$. Algorithm 3.16 shows the pseudocode for implementation of the GETLITERALDLCS procedure.

Algorithm 3.16 DPLL - GETLITERALDLCS

```
procedure GETLITERALDLCS
    max = 0
    for each unassigned variable x do
        s = Σ_k d_k(F, x)
        t = Σ_k d_k(F, x̄)
        r = s + t
        if r > max then
            max = r
            if s ⩾ t then u = x else u = x̄
        end if
    end for
    return u
end procedure
```

Now we provide the C listing of DLCS branching rule:

```
1   inline int GetLiteralDLCS()
2   {
3     register unsigned int i, j, C;
4     register unsigned int max = 0, r, s, t;
5     register int u;
6     for(i=1; i<=n_vars; ++i)
7     {
8      if(assign[i].decision == ASSIGN_NONE)
9      {
10       s = t = 0;
11       for(j=0; j<linfo[i][SATISFIED].n_occur; ++j)
12       {
13         C = linfo[i][SATISFIED].lit_in_clauses[j];
14         s += 1-clauses[C].is_satisfied;
15       }
16       for(j=0; j<linfo[i][SHRUNK].n_occur; ++j)
17       {
18         C = linfo[i][SHRUNK].lit_in_clauses[j];
19         t += 1-clauses[C].is_satisfied;
20       }
21       r = s + t;
22       if(r > max)
23       {
24         max = r;
25         if(s >= t) u = i;
26         else u = -i;
27       }
28     }
29    }
30    return u;
31  }
```

11-15 These lines implement line 4 of Algorithm 3.16.

16-20 These lines implement line 5 of Algorithm 3.16.

3.4.6.2 MOMS heuristic-based branching rule, MinLen

Here, $w(F, u)$ is the number of occurences of literal u in the smallest unsatisfied clauses and $\Phi(s,t) = (s+1)*(t+1)$. Algorithm 3.17 shows the pseudocode for implementation of the GETLITERALMINLEN procedure.

Algorithm 3.17 DPLL - GETLITERALMINLEN

1: **procedure** GETLITERALMINLEN
2:     k = length of the shortest unsatisfied clause in F
3:     **for** each unassigned variable x **do**
4:         $s = d_k(F, x)$
5:         $t = d_k(F, \overline{x})$
6:         $r = (s+1)*(t+1)$
7:         **if** $r > max$ **then**
8:             $max = r$
9:             **if** $s \geqslant t$ **then** $u = x$ **else** $u = \overline{x}$
10:         **end if**
11:     **end for**
12:     **return** x
13: **end procedure**

The following listing is the C implementation of line 2 in Algorithm 3.17.

```
inline int get_length_of_shortest_clause()
{
  register int i, j, C, type, min = max_clause_len;
  if(min == 2) return min;
  for(i=1; i<=n_vars; ++i)
  {
    if(assign[i].decision == ASSIGN_NONE)
    {
      for(type=0; type<2; ++type)
      {
        for(j=0; j<linfo[i][type].n_occur; ++j)
        {
          C = linfo[i][type].lit_in_clauses[j];
          if(!clauses[C].is_satisfied &&
```

```
              clauses[C].current_length < min)
          {
            min = clauses[C].current_length;
            if(min == 2) return 2;
          }
        }
      }
    }
  }
  return min;
}
```

The following listing is the C implementation of lines 4-5 in Algorithm 3.17. It takes a variable x and the length of the shortest clause k as input parameters and outputs $d_k(F,x)$ and $d_k(F,\overline{x})$ in s and t respectively.

```
void get_MOMS(int x, int k, unsigned int &s, unsigned int &t)
{
  register int j, c;
  s = t = 0;
  for(j=0; j<linfo[x][SATISFIED].n_occur; ++j)
  {
    c = linfo[x][SATISFIED].lit_in_clauses[j];
    if(clauses[c].current_length == k)
      s += 1-clauses[c].is_satisfied;
  }
  for(j=0; j<linfo[x][SHRUNK].n_occur; ++j)
  {
    c = linfo[x][SHRUNK].lit_in_clauses[j];
    if(clauses[c].current_length == k)
      t += 1-clauses[c].is_satisfied;
  }
}
```

Now we provide the C listing of Algorithm 3.17:

```
inline int GetLiteralMinLen()
{
```

```
    register unsigned int i, k;
    register unsigned int max = 0, r, s, t;
    register int u;
    for(i=1; i<=n_vars; ++i)
    {
     if(assign[i].decision == ASSIGN_NONE)
     {
       k = get_length_of_shortest_unsatisfied_clause();
       get_MOMS(i, k, s, t);
       r = (s+1)*(t+1);
       if(r > max)
       {
         max = r;
         if(s >= t) u = i;
         else u = -i;
       }
     }
    }
    return u;
  }
```

Originally Satz's (described in section 2.5) branching rule used MOMS heuristic with $\Phi(s,t) = s + t + s * t * 1024$, which has similar performance as Minlen. Later, unit-propagation based lookahead was integrated to Satz's branching rule to reduce the number of nodes in the DPLL-tree. As a result, the branching rule has become expensive as it makes many calls to SetVar and works best only with Satz and Satz-like solvers, where branching rule is highly integrated to the solver.

3.4.6.3 2-sided-Jeroslow-Wang

Here, $w(F,u)$ is defined as $\sum_k 2^{-k} d_k(F,u)$ and $\Phi(s,t)$ as $s+t$. Algorithm 3.18 shows the pseudocode for implementation of the GETLITERAL2SJW procedure.

Algorithm 3.18 DPLL - GETLITERAL2SJW

procedure GETLITERAL2SJW
    $s = t = max = 0$
    $mlen$ = length of the longest clause in F
    for each unassigned variable x **do**
        $s = \sum_k 2^{mlen-k} d_k(F,x)$
        $t = \sum_k 2^{mlen-k} d_k(F,\overline{x})$
        $r = s + t$
        if $r > max$ **then**
            $max = r$
            if $s \geqslant t$ **then** $u = x$ **else** $u = \overline{x}$
        end if
    end for
    return u
end procedure

Now, we provide the C listing of 2-Sided Jeroslow-Wang:

```
inline int GetLiteral2SJW()
{
 register unsigned int i, j, C;
 register unsigned int max = 0, r, s, t, mlen = max_clause_len;
 register int u;
 for(i=1; i<=n_vars; ++i)
 {
  if(assign[i].decision == ASSIGN_NONE)
  {
    s = t = 0;
    for(j=0; j<linfo[i][SATISFIED].n_occur; ++j)
    {
      C = linfo[i][SATISFIED].lit_in_clauses[j];
      s += ((!clauses[C].is_satisfied)<<(mlen-clauses[C].length));
    }
```

```
16        for(j=0; j<linfo[i][SHRUNK].n_occur; ++j)
17        {
18          C = linfo[i][SHRUNK].lit_in_clauses[j];
19          t += ((!clauses[C].is_satisfied)<<(mlen-clauses[C].length));
20        }
21        r = s + t;
22        if(r > max)
23        {
24          max = r;
25          if(s >= t) u = i;
26          else u = -i;
27        }
28      }
29    }
30    return u;
31  }
```

6-29 This `for` loop implements lines 4-12 of Algorithm 3.18. For each unassigned variable x, two `for` loops (lines 11-20) compute respectively s and t (lines 5-6) of Algorithm 3.18.

3.5 Comparing performance of branching rules

We know that the size of the DPLL-tree depends significantly on the branching rule. Several different branching rules have been proposed over the last couple of decades, but it is not really understood why a particular branching rule is better then the others. Most of the branching rules are based on intuitive ideas but no guarantees or theoretical proofs are given. Due to the inherent difficulty of the satisfiability problem, it seems impossible to design a branching rule that is good for nearly all instances of the satisfiability problem. In choosing a branching rule for our solver, we are no different. We look at the number of calls made to SetVar using different branching rules on selected instances from popular DIMACS benchmarks and see which one makes the least number of calls to `SetVar` on most instances. Here, we are assuming that the instances are simplified by the preprocessor.

3.5.1 DIMACS benchmark instances

DIMACS SAT challenges [20] include the instances: `aim`, `lran`, `jnh`, `dubois`, `gcp`, `parity`, `ii`, `hanoi`, `bf`, `ssa`, `phole`, and `pret`. These instances are widely used by SAT-solvers for testing performances. In this section, we check the the performance of our solver with different branching rules (DLCS, MINLEN, and 2SJW) on the instances `aim`, `pret`, `dubois`, `par`, `ii`, and `jnh`.

3.5.1.1 `aim` instances

Asahiro, Iwama, and Miyano [2] developed techniques to generate random formulas with some prescribed parameters (satisfiablity, literal distribution, clause distribution and number of satisfying truth assignments) in addition to the number of variables. The formulas generated have names started with `aim` for Asahiro, Iwama and Miyano. Each of the satisfiable instances has a unique satisfying assignment. Many of the these instances (satisfiable and unsatisfiable) can be solved by the preprocessor and the solver is invoked only if a satisfaction or a contradiction is not reached during preprocessing. Table 3.1 shows the total number of resolvents added, total number clauses subsumed, number of calls to `SetVar` and the CPU time. The number of resolvents added is restricted by the threshold on the number of resolvents.

Table 3.1: PERFORMANCE ON aim INSTANCES

INSTANCE	RESOLVENTS	SUBSUMED	#SETVARS(BR)	CPU TIME
aim-100-1_6-yes1-1.cnf(S)	249	57	100 (2SJW)	0.00s
aim-100-1_6-yes1-2.cnf(S)	348	149	100 (2SJW)	0.00s
aim-100-1_6-yes1-3.cnf(S)	290	115	100 (2SJW)	0.00s
aim-100-1_6-yes1-4.cnf(S)	800	492	100 (2SJW)	0.00s
aim-100-2_0-yes1-1.cnf(S)	990	454	100 (2SJW)	0.00s
aim-100-2_0-yes1-2.cnf(S)	1000	314	100 (2SJW)	0.00s
aim-100-2_0-yes1-3.cnf(S)	369	119	100 (2SJW)	0.00s
aim-100-2_0-yes1-4.cnf(S)	665	995	100 (2SJW)	0.00s
aim-100-3_4-yes1-1.cnf(S)	1700	682	102 (MINLEN)	0.00s
aim-100-3_4-yes1-2.cnf(S)	1700	584	100 (2SJW)	0.00s
aim-100-3_4-yes1-3.cnf(S)	1695	441	100 (MINLEN)	0.00s
aim-100-3_4-yes1-4.cnf(S)	1695	744	100 (MINLEN)	0.00s
aim-100-6_0-yes1-1.cnf(S)	2396	1008	100 (MINLEN)	0.00s
aim-100-6_0-yes1-2.cnf(S)	2400	972	100 (MINLEN)	0.00s
aim-100-6_0-yes1-3.cnf(S)	2400	1088	100 (2SJW)	0.00s
aim-100-6_0-yes1-4.cnf(S)	2392	899	100 (MINLEN)	0.00s
aim-100-1_6-no-1.cnf(U)	800	384	12 (MINLEN)	0.00s
aim-100-1_6-no-2.cnf(U)	800	349	46 (2SJW)	0.00s
aim-100-1_6-no-3.cnf(U)	785	446	7 (MINLEN)	0.00s
aim-100-1_6-no-4.cnf(U)	800	345	11 (MINLEN)	0.00s
aim-100-2_0-no-1.cnf(U)	814	58	0	0.00s
aim-100-2_0-no-2.cnf(U)	995	792	1 (2SJW)	0.00s
aim-100-2_0-no-3.cnf(U)	990	505	1 (2SJW)	0.00s
aim-100-2_0-no-4.cnf(U)	780	555	0 (2SJW)	0.00s
aim-200-1_6-yes1-1.cnf(S)	1600	1030	200 (2SJW)	0.00s
aim-200-1_6-yes1-2.cnf(S)	766	403	200 (2SJW)	0.00s
aim-200-1_6-yes1-3.cnf(S)	1289	952	200 (2SJW)	0.00s
aim-200-1_6-yes1-4.cnf(S)	1600	922	201 (2SJW)	0.00s
aim-200-2_0-yes1-1.cnf(S)	1985	959	200 (2SJW)	0.00s
aim-200-2_0-yes1-2.cnf(S)	1995	996	200 (2SJW)	0.00s
aim-200-2_0-yes1-3.cnf(S)	1995	1198	200 (2SJW)	0.00s
aim-200-2_0-yes1-4.cnf(S)	1995	1277	200 (2SJW)	0.00s
aim-200-3_4-yes1-1.cnf(S)	2716	758	200 (2SJW)	0.00s
aim-200-3_4-yes1-2.cnf(S)	2716	894	676 (MINLEN)	0.00s
aim-200-3_4-yes1-3.cnf(S)	2716	569	201 (MINLEN)	0.00s
aim-200-3_4-yes1-4.cnf(S)	2708	926	200 (MINLEN)	0.00s
aim-200-6_0-yes1-1.cnf(S)	3525	1042	200 (MINLEN)	0.00s
aim-200-6_0-yes1-2.cnf(S)	3567	700	200 (2SJW)	0.00s
aim-200-6_0-yes1-3.cnf(S)	3549	2251	200 (MINLEN)	0.00s
aim-200-6_0-yes1-4.cnf(S)	1592	706	200 (2SJW)	0.00s
aim-200-1_6-no-1.cnf(U)	1600	1156	21 (MINLEN)	0.00s
aim-200-1_6-no-2.cnf(U)	1585	1212	12 (MINLEN)	0.00s
aim-200-1_6-no-3.cnf(U)	1600	1149	22 (MINLEN)	0.00s
aim-200-1_6-no-4.cnf(U)	1600	1496	6 (2SJW)	0.00s

Continued on Next Page...

Table 3.1: PERFORMANCE ON aim INSTANCES (CONTINUED...)

aim-200-2_0-no-1.cnf(U)	1995	1531	8 (2sJW)	0.00s
aim-200-2_0-no-2.cnf(U)	1995	1397	4 (2sJW)	0.00s
aim-200-2_0-no-3.cnf(U)	1995	1246	7 (2sJW)	0.00s
aim-200-2_0-no-4.cnf(U)	2000	1600	7 (2sJW)	0.00s

We observe that 2sJW performs better on some of the aim instances and MinLen performs better on the others.

3.5.1.2 dubois instances

Oliver Dubois contributed a SAT formula generator, called gensathard.c, to the DIMACS collection. A dubois formula of degree d is an encoding of the parity problem of the multigraph in figure 3.1. The graph has $2d$ vertices and $3d$ edges. The lower leftmost vertex is assigned parity 0, and the other vertices are assigned parity 1. Since the sum of the parities is odd, the formula is unsatisfiable. A dubois formula with degree d has $3d$ variables (a variable labels an edge) and $8d$ clauses (four times the number of vertices). Most of these instances can be solved during preprocessing.

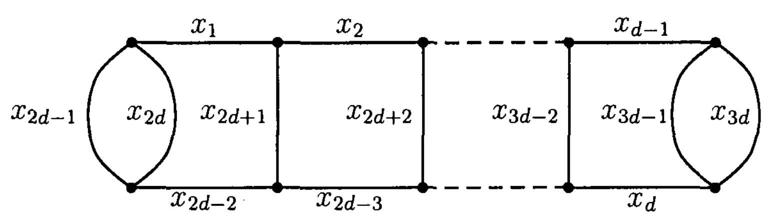

Figure 3.1: THE MULTIGRAPH UNDERLYING THE *dubois* FORMULA OF DEGREE d

Table 3.2: PERFORMANCE ON dubois INSTANCES

INSTANCE	RESOLVENTS	SUBSUMED	#SETVARS(BR)	CPU TIME
dubois20.cnf(U)	720	448	0	0.00s
dubois21.cnf(U)	840	655	0	0.00s
dubois22.cnf(U)	788	796	0	0.00s
dubois23.cnf(U)	920	714	0	0.00s
dubois24.cnf(U)	856	544	0	0.00s
dubois25.cnf(U)	1000	773	0	0.00s
dubois26.cnf(U)	924	592	0	0.00s
dubois27.cnf(U)	1080	831	0	0.00s
dubois28.cnf(U)	992	640	0	0.00s
dubois29.cnf(U)	1160	899	0	0.00s
dubois30.cnf(U)	1060	688	0	0.00s
dubois50.cnf(U)	1740	1168	0	0.00s

For all these instances, contradictory unit clauses are found during preprocessing and hence UNSAT is returned before calling the solver.

3.5.1.3 pret instances

Daniele Pretolani contributed to the DIMACS collection a SAT formula generator trisat.c that generates the PRET instances. Given an integer s greater than 3, the generator first produces a connected 3-regular graph with s vertices. Then it starts with PRET4, which is K_4, the complete graph with four vertices(Figure 3.2(a)). It then keeps expanding the graph as follows: (i) take a vertex v and two of its neighbors v_1 and v_2, (ii) introduce two new vertices v_1' and v_2' and replace the two edges $\{v, v_1\}$ and $\{v, v_2\}$ by the five edges $\{v, v_1'\}$, $\{v, v_2'\}$, $\{v_1, v_1'\}$, $\{v_2, v_2'\}$ and $\{v_1', v_2'\}$. The result depends on the order of the vertices and the choices of the neighbours. In general, there may be multiple pairs of neighbours to choose. The generator trisat.c has a deterministic way for doing it. In Figure 3.2, (b), (c), (d) and (e) are obtained by working on the vertices v_1, v_2, v_3 and v_4 in that order.

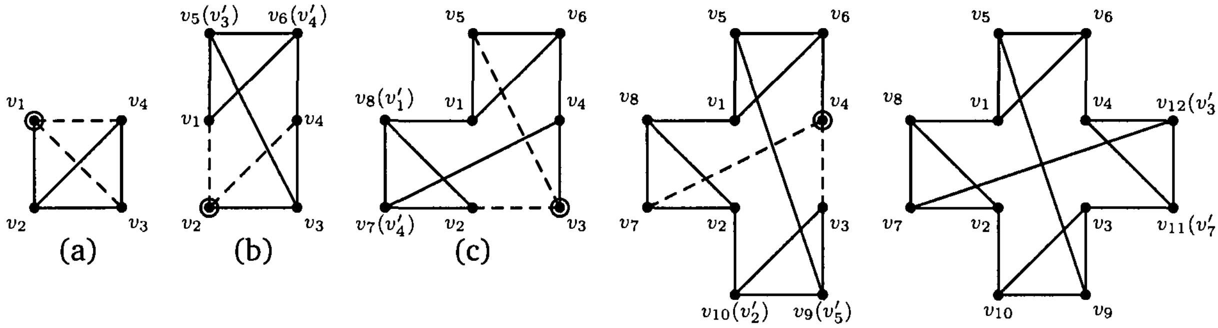

Figure 3.2: EXAMPLES OF GRAPHS CORRESPONDING TO pret INSTANCES

Table 3.3: PERFORMANCE ON THE pret INSTANCES

INSTANCE (S/U)	DLCS	MINLEN	2SJW
pret60-25.cnf(U)	68157438 (33.18S)	**11491010 (6.35s)**	16354642 (6.72S)
pret60-40.cnf(U)	68157438 (33.31S)	**11491010 (6.42s)**	16354642 (6.48S)
pret60-60.cnf(U)	68157438 (33.77S)	**11491010 (6.35s)**	16354642 (6.58S)
pret60-75.cnf(U)	68157438 (33.20S)	**11491010 (6.36s)**	16354642 (6.54S)

3.5.1.4 par **instances**

The PAR formulas, contributed by James Crawford, encode parity learning problems. Consider the parity functions over subsets of $\{x_1, x_2, \ldots, x_n\}$. The inputs to the functions are vectors in $\{0,1\}^n$, and the function computes the parity of a subset, V, of $\{x_1, x_2, \ldots, x_n\}$. The parity learning problem is, given m pairs of sample inputs and corresponding outputs, identify the subset V that determines the function values. These instances are all satisfiable, and the satisfying assignments can be translated to the incidence vectors of V.

Table 3.4: PERFORMANCE ON THE par INSTANCES

INSTANCE	DLCS	MINLEN	2SJW
par16-1.cnf(S)	5054704 (3.81s)	**3137702 (2.90s)**	3906055 (2.67s)
par16-2.cnf(S)	2013579 (1.43s)	**1960298 (1.69s)**	3547112 (2.07s)
par16-3.cnf(S)	1774584 (1.32s)	1823274 (1.77s)	**1143446 (0.76s)**
par16-4.cnf(S)	7317874 (5.18s)	4302527 (3.65s)	**4075472 (2.55s)**
par16-5.cnf(S)	10849808 (8.72s)	**6141771 (5.79s)**	13093507 (9.20s)
par16-1-c.cnf(S)	2210695 (2.61s)	1732677 (1.90s)	**1660189 (1.33s)**
par16-2-c.cnf(S)	6722992 (8.90s)	6510934 (7.82s)	**1506899 (1.29s)**
par16-3-c.cnf(S)	**69938 (0.07s)**	2031979 (2.53s)	1286371 (1.18s)
par16-4-c.cnf(S)	4094161 (5.13s)	2546039 (3.68s)	**1794838 (1.58s)**
par16-5-c.cnf(S)	5896518 (7.07s)	**1720827** (1.93s)	1842087 **(1.43s)**

We observe that each of the three branching rules perform better than the other two on some of the par instances.

3.5.1.5 Other DIMACS instances

The ii instances are described in Kamath, Karmakar, Ramakrishnan, and Resende [37] and have been contributed to the DIMACS collection by Mauricio Resende.

Table 3.5: PERFORMANCE ON THE ii INSTANCES

INSTANCE (S/U)	DLCS	MINLEN	2SJW
ii16b2.cnf(S)	**1491641 (14.00s)**	3713456 (34.03s)	3894963 (37.41s)
ii16c2.cnf(S)	438235 (3.27s)	**47149 (0.90s)**	128723 (1.14s)
ii16d2.cnf(S)	**24103 (0.24s)**	1460837 (24.10s)	1454699 (10.51s)
ii16e2.cnf(S)	352332 (2.54s)	12337 (0.21s)	**12209 (0.09s)**
ii32b1.cnf(S)	1107 (0.01s)	**222 (0.00s)**	2988 (0.02s)
ii32b2.cnf(S)	**10498 (0.04s)**	21009 (0.08s)	278634 (0.89s)
ii32b3.cnf(S)	**2660 (0.02s)**	3266 (0.09s)	121849 (0.60s)
ii32b4.cnf(S)	**7252 (0.07s)**	6342 (0.14s)	2083859 (14.58s)
ii32c1.cnf(S)	216 **(0.00s)**	**184** (0.01s)	217 (0.01s)
ii32c2.cnf(S)	368 **(0.00s)**	**220** (0.01s)	2498 (0.03s)
ii32c3.cnf(S)	446 (0.01s)	**271 (0.01s)**	10747 (0.06s)
ii32d1.cnf(S)	527 (0.00s)	**476 (0.00s)**	167183 (0.30s)
ii32d2.cnf(S)	155344 (0.40s)	**1730 (0.01s)**	5013175 (9.36s)
ii32e1.cnf(S)	211 (0.00s)	**195 (0.00s)**	210 (0.00s)
ii32e2.cnf(S)	622 (0.01s)	**267 (0.00s)**	3300 (0.03s)

Continued on Next Page...

Table 3.5: PERFORMANCE ON ii INSTANCES (CONTINUED...)

ii32e3.cnf(S)	**764 (0.01s)**	3046 (0.08)	634718 (2.66s)
ii32e4.cnf(S)	16042 (0.16s)	**652 (0.10s)**	758354 (4.45s))
ii32e5.cnf(S)	**1171 (0.04s)**	9066 (0.31s)	1561632 (17.25s)

We observe that DLCS performs better on some of the ii instances and MinLen performs better on the others.

The jnh instances are contributed to the DIMACS collection by John Hooker and are described in [57].

PERFORMANCE ON THE jnh INSTANCES

INSTANCE	DLCS	MINLEN	2SJW
jnh1.cnf(S)	1965 (0.01s)	**257 (0.00s)**	1189 (0.00s)
jnh2.cnf(U)	160 (0.00s)	113 (0.00s)	**102 (0.00s)**
jnh3.cnf(U)	1522 (0.00s)	**805 (0.00s)**	1390 (0.01s)
jnh4.cnf(U)	667 (0.00s)	**526 (0.00s)**	629 (0.01s)
jnh5.cnf(U)	214 (0.00s)	**163 (0.01s)**	437 (0.00s)
jnh6.cnf(U)	1131 (0.01s)	**483 (0.00s)**	1188 (0.01s)
jnh7.cnf(S)	121 (0.00s)	**97 (0.00s)**	256 (0.00s)
jnh8.cnf(U)	170 (0.00s)	**96 (0.00s)**	150 (0.00s)
jnh9.cnf(U)	315 (0.01s)	**261 (0.00s)**	440 (0.00s)
jnh10.cnf(U)	820 (0.00s)	**206 (0.00s)**	702 (0.00s)
jnh11.cnf(U)	842 (0.01s)	**238 (0.00s)**	295 (0.00s)
jnh12.cnf(S)	409 (0.00s)	**157 (0.00s)**	157 (0.00s)
jnh13.cnf(U)	186 (0.00s)	**156 (0.00s)**	268 (0.00s)
jnh14.cnf(U)	385 (0.00s)	**154 (0.00s)**	245 (0.00s)
jnh15.cnf(U)	723 (0.01s)	**431 (0.00s)**	581 (0.00s)
jnh16.cnf(U)	10115 (0.05s)	**5272 (0.02s)**	9290 (0.04s)
jnh17.cnf(S)	810 (0.01s)	422 (0.01s)	**122 (0.00s)**
jnh18.cnf(U)	1060 (0.01s)	**467 (0.00s)**	1149 (0.01s)
jnh19.cnf(U)	1334 (0.00s)	**410 (0.00s)**	904 (0.01s)
jnh20.cnf(U)	361 (0.00s)	**257 (0.00s)**	388 (0.01s)
jnh201.cnf(S)	97 (0.00s)	98 (0.00s)	**92 (0.00s)**
jnh202.cnf(U)	155 (0.00s)	**103 (0.00s)**	155 (0.00s)
jnh203.cnf(U)	504 (0.00s)	**419 (0.00s)**	472 (0.00s)
jnh204.cnf(S)	759 (0.01s)	608 (0.01s)	**323 (0.00s)**
jnh205.cnf(S)	**100 (0.00s)**	396 (0.00s)	120 (0.00s)
jnh206.cnf(U)	1055 (0.00s)	**806 (0.00s)**	1052 (0.00s)
jnh207.cnf(S)	1644 (0.01s)	**1089 (0.00s)**	1661 (0.01s)
jnh208.cnf(U)	1555 (0.01s)	**513 (0.00s)**	997 (0.00s)
			Continued on Next Page...

PERFORMANCE ON THE jnh INSTANCES (CONTINUED...)

jnh209.cnf(S)	285 (0.00s)	**201 (0.00s)**	227 (0.00s)
jnh210.cnf(S)	129 (0.00s)	**119 (0.00s)**	132 (0.00s)
jnh211.cnf(U)	175 (0.00s)	**149 (0.00s)**	193 (0.00s)
jnh212.cnf(S)	2448 (0.01s)	**205 (0.00s)**	1837 (0.00s)
jnh213.cnf(S)	**98 (0.00s)**	149 (0.01s)	116 (0.00s)
jnh214.cnf(U)	416 (0.00s)	**388 (0.00s)**	413 (0.00s)
jnh215.cnf(U)	345 (0.00s)	**172 (0.00s)**	329 (0.00s)

Branching rule MinLen performs consistently better on the jnh instances.

It seems reasonably difficult to find a branching rule that works well on most classes of instances. In the following section, we use MINLEN as the branching rule (since it performs consistently better on the DIMACS instances) to compare our solver with other well-known solvers.

3.6 Performance of our solver

In this section, we compare the running time of our solver with SATZ [44], zCHAFF [49], and MINISAT [58] on some DIMACS satisfiability instances [20] and some other instances from the SATLIB collection [55]. We have compiled and run them on a 2.2 GHz AMD Opteron 64-bit processor machine in the `cirrus` cluster at Concordia University.

3.6.1 On DIMACS instances

In the previous section, we have discussed some of the DIMACS SAT instances (`aim`, `pret`, `dubois`, and `par`) while comparing the performance of different branching rules on our solver. For detail of other instances, see [20]. In this section, we compare the performance of our solver with other well-known solvers (SATZ, zCHAFF, and MINISAT) on some DIMACS instances.

Table 3.7: PERFORMANCE OF OUR SOLVER ON `dubois` INSTANCES

INSTANCES (S/U,#VARS,#CLAUSES)	SATZ	zCHAFF	MINISAT	OUR SOLVER
`dubois20.cnf` (U, 60, 160)	3.90s	0.01s	0.00s	0.00s
`dubois21.cnf` (U, 63, 168)	4.35s	0.01s	0.00s	0.00s
`dubois22.cnf` (U, 66, 176)	11.74s	0.00s	0.00s	0.00s
`dubois23.cnf` (U, 69, 184)	31.22s	0.00s	0.00s	0.00s
`dubois24.cnf` (U, 72, 192)	35.18s	0.00s	0.00s	0.00s
`dubois25.cnf` (U, 75, 200)	>60s	0.00s	0.00s	0.00s
`dubois26.cnf` (U, 78, 208)	>60s	0.00s	0.00s	0.00s
`dubois27.cnf` (U, 81, 216)	>60s	0.00s	0.00s	0.00s
`dubois28.cnf` (U, 84, 224)	>60s	0.01s	0.00s	0.00s
`dubois29.cnf` (U, 87, 232)	>60s	0.01s	0.00s	0.00s
`dubois30.cnf` (U, 90, 240)	>60s	0.01s	0.00s	0.00s
`dubois50.cnf` (U, 150, 400)	>60s	0.01s	0.00s	0.00s

Table 3.8: PERFORMANCE OF OUR SOLVER ON pret INSTANCES

INSTANCES (S/U,#VARS,#CLAUSES)	SATZ	ZCHAFF	MINISAT	OUR SOLVER
pret60-25.cnf (U, 60, 160)	5.92s	0.01s	0.00s	6.35s
pret60-40.cnf (U, 60, 160)	5.55s	0.01s	0.00s	6.42s
pret60-60.cnf (U, 60, 160)	5.41s	0.01s	0.00s	6.35s
pret60-75.cnf (U, 60, 160)	5.28s	0.01s	0.00s	6.36s

Table 3.9: PERFORMANCE OF OUR SOLVER ON par INSTANCES

INSTANCES (S/U,#VARS,#CLAUSES)	SATZ	ZCHAFF	MINISAT	OUR SOLVER
par8-1.cnf (S, 350, 1149)	0.00s	0.00s	0.00s	0.00s
par8-2.cnf (S, 350, 1157)	0.00s	0.00s	0.00s	0.00s
par8-3.cnf (S, 350, 1171)	0.00s	0.00s	0.00s	0.00s
par8-4.cnf (S, 350, 1155)	0.00s	0.00s	0.00s	0.00s
par8-5.cnf (S, 350, 1171)	0.00s	0.00s	0.00s	0.00s
par8-1-c.cnf (S, 64, 254)	0.00s	0.00s	0.00s	0.00s
par8-2-c.cnf (S, 68, 270)	0.00s	0.00s	0.00s	0.00s
par8-3-c.cnf (S, 75, 298)	0.00s	0.00s	0.00s	0.00s
par8-4-c.cnf (S, 67, 266)	0.00s	0.00s	0.00s	0.00s
par8-5-c.cnf (S, 75, 298)	0.00s	0.00s	0.00s	0.00s
par16-1.cnf (S, 1015, 3310)	1.45s	0.76s	0.04s	2.90s
par16-2.cnf (S, 1015, 3374)	0.08s	1.07s	0.31s	1.69s
par16-3.cnf (S, 1015, 3344)	2.86s	0.28s	0.20s	1.77s
par16-4.cnf (S, 1015, 3324)	1.84s	0.28s	0.01s	3.65s
par16-5.cnf (S, 1015, 3358)	0.26s	0.76s	0.17s	5.79s
par16-1-c.cnf (S, 317, 1264)	0.39s	0.36s	0.01s	1.90s
par16-2-c.cnf (S, 349, 1392)	0.15s	0.77s	0.16s	7.82s
par16-3-c.cnf (S, 334, 1332)	0.48s	0.13s	0.11s	2.53s
par16-4-c.cnf (S, 324, 1292)	0.10s	0.01s	0.00s	3.68s
par16-5-c.cnf (S, 341, 1360)	0.30s	0.58s	0.10s	1.93s

Table 3.10: PERFORMANCE OF OUR SOLVER ON phole INSTANCES

INSTANCES (S/U,#VARS,#CLAUSES)	SATZ	ZCHAFF	MINISAT	OUR SOLVER
hole6.cnf (U, 42, 133)	0.00s	0.01s	0.00s	0.00s
hole7.cnf (U, 56, 204)	0.02s	0.03s	0.02s	0.01s
hole8.cnf (U, 72, 297)	0.17s	0.25s	0.26s	0.19s
hole9.cnf (U, 90, 415)	1.62s	1.04s	1.70s	1.90s
hole10.cnf (U, 110, 561)	16.74s	5.57s	28.88s	20.78s

Table 3.11: PERFORMANCE OF OUR SOLVER ON ssa INSTANCES

INSTANCES (S/U,#VARS,#CLAUSES)	SATZ	ZCHAFF	MINISAT	OUR SOLVER
ssa0432-003.cnf (U, 435, 1027)	0.00s	0.01s	0.00s	0.26s
ssa6288-047.cnf (U, 10410, 34238)	0.14s	0.01s	0.01s	0.26s
ssa7552-038.cnf (S, 1501, 3575)	0.05s	0.01s	0.00s	0.00s
ssa7552-158.cnf (S, 1363, 3034)	0.03s	0.00s	0.00s	0.00s
ssa7552-159.cnf (S, 1363, 3032)	0.04s	0.00s	0.00s	0.00s
ssa7552-160.cnf (S, 1391, 3126)	0.03s	0.00s	0.00s	0.00s

Table 3.12: PERFORMANCE OF OUR SOLVER ON ii INSTANCES

INSTANCES (S/U,#VARS,#CLAUSES)	SATZ	ZCHAFF	MINISAT	OUR SOLVER
ii16b2.cnf (S, 1076, 16121)	0.39s	0.36s	0.00s	34.03s
ii16c2.cnf (S, 924, 13803)	0.44s	0.01s	0.01s	0.90s
ii16d2.cnf (S, 836, 12461)	0.46s	0.01s	0.00s	24.10s
ii16e2.cnf (S, 532, 7825)	0.58s	0.01s	0.01s	0.21s
ii32b1.cnf (S, 228, 1374)	0.05s	0.00s	0.00s	0.00s
ii32b2.cnf (S, 261, 2558)	0.15s	0.00s	0.00s	0.06s
ii32b3.cnf (S, 348, 5734)	0.87s	0.00s	0.00s	0.06s
ii32b4.cnf (S, 381, 6918)	1.14s	0.02s	0.00s	0.13s
ii32c1.cnf (S, 225, 1280)	0.05s	0.00s	0.00s	0.00s
ii32c2.cnf (S, 249, 2182)	0.15s	0.01s	0.00s	0.00s
ii32c3.cnf (S, 279, 3272)	0.34s	0.01s	0.00s	0.00s
ii32d1.cnf (S, 332, 2703)	0.09s	0.01s	0.00s	0.00s
ii32d2.cnf (S, 404, 5153)	0.24s	0.01s	0.00s	0.00s
ii32e1.cnf (S, 222, 1186)	0.04s	0.01s	0.00s	0.00s
ii32e2.cnf (S, 267, 2746)	0.16s	0.00s	0.00s	0.00s
ii32e3.cnf (S, 330, 5020)	0.51s	0.01s	0.01s	0.01s
ii32e4.cnf (S, 387, 7106)	1.22s	0.01s	0.01s	0.07s
ii32e5.cnf (S, 522, 11636)	2.05s	0.00s	0.01s	0.25s

We do not list the performance on the aim and jnh instances as all four solvers perform well on them.

3.6.2 Other instances from SATLIB solvers collection

3.6.2.1 Uniform Random 3-SAT

3-SAT instances with m clauses over n variables in the SATLIB collection have been generated in the following way:

1. Each of the m clauses is constructed by drawing a literal uniformly at random from the $2n$ possible literals.

2. Clauses containing duplicate literals are not added.

3. Clauses containing both a literal and its complement are not added.

We compare our solver with MINISAT on several instances (both satisfiable and unsatisfiable) of the SATLIB collection. Here, we list the total running time, taken by each solver on several instances.

Table 3.13: PERFORMANCE ON THE uf INSTANCES

INSTANCE (S/U)	OUR SOLVER	MINISAT
uf125-538 (100 satisfiable instances)	0.60s	0.10s
uuf125-538 (100 unsatisfiable instances)	2.25s	0.34s
uf150-645 (100 satisfiable instances)	2.65s	0.45s
uuf150-645 (100 unsatisfiable instances)	7.83s	0.50s
uf175-753 (100 satisfiable instances)	9.74s	1.72s
uuf175-753 (100 unsatisfiable instances)	27.02s	3.45s

We observe that our solver does not perform well, compared to ZCHAFF and MINISAT on well-known instances from DIMACS and SATLIB. In the following chapter, we discuss a class of instances (vdw instances) where our solver (with a suitable branching rule) performs better than any other known SAT-solver. We have used the solver to compute some previously unknown van der Waerden numbers (defined in section 4.1). These numbers are published in Ahmed [1].

Chapter 4

SAT and van der Waerden numbers

When we started coding for DPLL, the instances we used to test the performance were instances related to the van der Waerden numbers, defined in the following section. In various phases, we have improved and optimized our solver to perform in these instances as efficiently as possible. Eventually, we have been able to compute thirty new van der Waerden numbers.

4.1 Van der Waerden numbers

The *van der Waerden number* $w(r; k_1, k_2, \cdots, k_r)$ is the least integer m such that for every partition $P_1 \cup P_2 \cup \cdots \cup P_r$ of the set $\{1, 2, \cdots, m\}$, there is an index j in $\{1, 2, \cdots, r\}$ such that P_j contains an arithmetic progression of k_j terms. A list of van der Waerden numbers known so far is given in Table 4.5 at the end of this chapter.

4.2 SAT encoding of van der Waerden numbers

Given positive integers $r, k_1, \cdots, k_r$, and n, we construct a SAT formula (an instance of the satisfiability problem), which is satisfiable if and only if $w(r; k_1, k_2, \cdots, k_r) > n$. We consider the following two cases:

When $r = 2$, we have variables x_i for $1 \leqslant i \leqslant n$ and the following clauses:

(a) $\{\overline{x}_a, \overline{x}_{a+d}, \cdots, \overline{x}_{a+d(k_1-1)}\}$ with $a \geqslant 1, d \geqslant 1, a + d(k_1 - 1) \leqslant n$,

(b) $\{x_a, x_{a+d}, \cdots, x_{a+d(k_2-1)}\}$ with $a \geqslant 1, d \geqslant 1, a + d(k_2 - 1) \leqslant n$.

Here, $x_i =$ TRUE encodes $i \in P_1$ and $x_i =$ FALSE encodes $i \in P_2$ (if x_i is not assigned but the formula is satisfied, then i can be arbitrarily placed in either of the blocks of the partition). Clauses (a) prohibit the existence of an arithmetic progression of length k_1 in P_1 and clauses (b) prohibit the existence of an arithmetic progression of length k_2 in P_2.

When $r > 2$, we take one variable for each integer and each block of the partition. Each variable $x_{i,j}$ with $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant r$, takes value TRUE if and only if the integer i belongs to a block P_j of a partition. This generates nr variables. The double subscripts i, j can be routinely encoded as single subscripts such as $r(i-1)+j$ or $i + n(j-1)$. We have the following clauses:

(a) INTEGER i IS IN AT LEAST ONE BLOCK: For each integer i, we have the clause $\{x_{i,1}, x_{i,2}, \cdots, x_{i,r}\}$ to ensure that i belongs to at least one block of the partition.

(b) NO ARITHMETIC PROGRESSION OF LENGTH k_j IN BLOCK P_j: This is the most important constraint. For $1 \leqslant j \leqslant r$, $1 \leqslant a \leqslant n - k_j + 1$ and $1 \leqslant d \leqslant$

$\lfloor (n-a)/(k_j-1) \rfloor$, we add the following clauses:

$$\{\overline{x}_{a,j}, \overline{x}_{a+d,j}, \cdots, \overline{x}_{a+d(k_j-1),j}\}.$$

(c) INTEGER i IS CONTAINED IN AT MOST ONE BLOCK: We want an integer not to be contained in more than one block of the partition. To do so, we add the following clauses: $\{\overline{x}_{i,s}, \overline{x}_{i,t}\}$ for $1 \leqslant i \leqslant n, 1 \leqslant s < t \leqslant r$.

Clauses of the third kind are not necessary, but their presence may steer the branching rules towards better decisions.

4.3 Experiments on some van der Waerden formulas

We denote a van der Waerden instance by wr-k_1-$\cdots$-k_r-n.cnf, where r is the number of blocks in the partition and n is an integer. The instance is satisfiable if and only if $n < w(r; k_1, \ldots, k_r)$. In this section, we report the results of the experiment on some known values of van der Waerden numbers to evaluate the performance of different branching rules on these instances. In this experiment, we run our solver on 2.2 GHz AMD Opteron 64-bit processors of the `cirrus` cluster at Concordia University. Preprocessing (as described in section 3.4.5.1) does not help in simplifying these instances. From Table 4.1, we see that `2sJW` consistantly performs better (in terms of the number of calls to `SetVar` and running time) than the other two branching rules on the `vdw` instances. So, we fix `2sJW` as the branching rule for further experiments on the `vdw` instances.

Table 4.1: PERFORMANCE ON THE vdw INSTANCES

INSTANCE	DLCS	MINLEN	2SJW
w2-3-3-9.cnf(U)	34 (0.00S)	34 (0.00S)	**32 (0.00s)**
w2-3-4-18.cnf(U)	157 (0.00S)	**116 (0.00s)**	123 (0.00S)
w2-3-5-22.cnf(U)	452 (0.00S)	420 (0.00S)	**396 (0.00s)**
w2-3-6-32.cnf(U)	1889 (0.00S)	1898 (0.00S)	**1432 (0.00s)**
w2-3-7-46.cnf(U)	24597 (0.02S)	36976 (0.03S)	**20174 (0.02s)**
w2-3-8-58.cnf(U)	55668 (0.08S)	47103 (0.09S)	**28326 (0.05s)**
w2-3-9-77.cnf(U)	386856 (0.83S)	217512 (0.56S)	**109984 (0.27s)**
w2-3-10-97.cnf(U)	4505603 (12.96S)	1635291 (5.30S)	**749378 (2.30s)**
w2-3-11-114.cnf(U)	42613428 (147.50S)	10145290 (38.99S)	**4249781 (15.31s)**
w2-3-12-135.cnf(U)	459501234 (1807S)	73592941 (343.64S)	**25027457 (109s)**
w2-3-13-160.cnf(U)	(> 6 HRS)	616727175 (3208S)	**204929576 (971s)**
w2-4-4-35.cnf(U)	3684 (0.00S)	1490 (0.00S)	**1334 (0.00s)**
w2-4-5-55.cnf(U)	79428 (0.13S)	27284 (0.10S)	**20842 (0.04s)**
w2-4-6-73.cnf(U)	6312526 (13.47S)	1567336 (6.92S)	**936838 (2.39s)**
w2-4-7-109.cnf(U)	3389336998 (11476S)	166908653 (979S)	**68788298 (297s)**
w2-5-5-178.cnf(U)	(> 6 HRS)	(> 6 HRS)	**8177796 (125.20s)**

4.4 New van der Waerden numbers found by Kouril

In 2006, Kouril [39] found seven new van der Waerden numbers, one of which was $w(2;5,6)$. Unaware of Kouril's progress, we were also trying to determine this number. Once we have found in 2007 that this number is 206, we tried to improve the running time of our solver on `w2-5-6-206.cnf`. It turned out that in proving the instance `w2-5-6-206.cnf` to be unsatisfiable, our solver (using 2sJW as branching rule) performs (takes 6.2 days) significantly better than any other known solver (for example, MINISAT takes 35 days).

Table 4.2: RUNNING TIME ON VAN DER WAERDEN INSTANCES

INSTANCE	S/U	SATZ	ZCHAFF	MINISAT	OUR SOLVER
w-2-4-7-109.cnf	(U)	25.8 mins	>100 mins	4.1 mins	4 mins
w-2-3-13-160.cnf	(U)	-	-	20.6 mins	15.9 mins
w-3-3-4-4-89	(U)	-	-	>10 days	4.1 days
w-4-3-3-3-76.cnf	(U)	-	-	>15 days	3.9 days
w-2-5-6-206.cnf	(U)	-	-	35 days	6.2 days

Table 4.2 shows that our solver performs better than other well-known solvers on hard van der Waerden instances.

Table 4.3 provides a good partition related to all the van der Waerden numbers found by Kouril and also $w(2;6,6)$ found by Kouril and Paul [40]. Here a *good partition* means a partition $P_1 \cup P_2 \cup \cdots \cup P_r$ such that no P_j contains and arithmetic progression of k_j terms. We will use strings to denote partitions; for example, 11221122 denotes $P_1 = \{1,2,5,6\}$ and $P_2 = \{3,4,7,8\}$.

Table 4.3: VAN DER WAERDEN NUMBERS FOUND BY KOURIL

$w(r;k_1,k_2,\cdots,k_r)$			EXAMPLE OF A GOOD PARTITION
$w(2;3,14)$	=	186	22121222 22222222 22112222 21222222 21222222 22121222 22222122 22222122 22222221 12221222 22222222 22122112 22222222 21212222 21222222 22122222 12122222 22222122 21222222 2222A211 22221222 22222212 22212222 2 (where A is arbitrary).
$w(2;3,15)$	=	218	22222222 21222122 22222122 21122222 12222222 22221222 21222222 22211222 22222212 22222212 22212221 22222222 22221122 22222212 22222212 22222222 21222222 22212222 12122222 12112222 22222221 22222122 22222221 12222222 22221122 22222212 22222222 2
$w(2;3,16)$	=	238	2222A221 22222222 22222212 1222B222 22212211 22222222 22222221 22222112 12222222 22221221 21222222 22222221 22222222 22221222 12222222 22122122 22222122 22222222 22212212 22221121 22222222 22222122 12222222 12222222 222221C2 21222221 12222212 22222222 21222222 22222 (where ABC is arbitrary).
$w(2;4,8)$	=	146	112221A2 12222112 22222122 12222211 12212222 12111222 21221121 21222222 21122222 12211222 21222212 11222112 22222211 21222222 21122212 11222221 12212222 122B1121 2

Continued on Next Page...

Table 4.3: VAN DER WAERDEN NUMBERS FOUND BY KOURIL

			(where AB is arbitrary).
$w(2;5,6)$	=	206	21112111 22122221 11222211 11211122 21222212 11222221 12122221 22211121 11121221 12112212 11112111 22212222 12112222 21121222 21222111 21111212 21121122 12111121 11222122 22121122 22211212 22212221 11211112 22211122 22122AB1 21112 (where AB is arbitrary).
$w(2;6,6)$	=	1132	A1222211 21111121 22221222 11222221 22122212 11122212 11212112 21122121 22121112 22121112 11211111 22111211 11212222 21221111 21211112 21222221 21111211 12211111 21121112 12221112 12212122 11221121 21121222 11121222 12212222 21122212 22212111 11211222 21212222 11211111 21222212 22112222 21221222 12111222 12112121 12211221 21221211 12221211 12112111 11221112 11112122 22212211 112B2111 12212222 21211112 11122111 11211211 12122211 12122121 22112211 21211212 22111212 22122122 22211222 12222121 11112112 2221C122 22112111 11212222 12221122 22212212 22121112 22121121 21122112 21212212 11122212 11121121 11112211 12111121 22222122 11112221 11122122 22212111 12111221 11112112 11121222 11121221 21221122 11212112 12221112 12221221 22222112 22122221 21111121 12222111 22221121 11112122 22122211 22222122 12221211 12221211 21211221 12212122 12111222 12111211 21111122 11121111 21222221 2211112D 21111221 22222121 11121112 21111121 12111212 22111212 21212211 22112121 12122211 12122212 21222221 12221222 21211111 21122221 E1222211 21111121 22221222 11222221 22122212 11122212 11212112 21122121 22121112 22121112 11211111 22111211 11212222 21221111 2F211112 21222221 21111211 12211111 21121112 12221112 12212122 11221121 21121222 11121222 12212222 21122212 22212111 11211222 21G

Continued on Next Page...

Table 4.3: VAN DER WAERDEN NUMBERS FOUND BY KOURIL

			(where ABCDEFG is arbitrary).
$w(3;2,3,8)$	=	72	33333233 23233323 33333233 32323323 33331323 32323332 33333323 33232332 3333333
$w(3;2,4,7)$	=	119	33333322 23223332 33233233 33332233 32333232 33233333 22232233 22233323 23313333 33232323 32233232 33333322 32223333 23323332 333333

4.5 Some new van der Waerden numbers found by us

We have found thirty previously unknown van der Waerden numbers. These numbers and the corresponding good partitions are listed in Table 4.4.

Table 4.4: VAN DER WAERDEN NUMBERS FOUND BY US

$w(r;k_1,k_2,\cdots,k_k)$			EXAMPLE OF A GOOD PARTITION
$w(3;2,3,9)$	=	90	33333332 33332333 33332322 32333332 33322333 33332333 33323333 33133223 23333323 33223333 33323233 3
$w(3;2,3,10)$	=	108	33333233 33333332 33233323 33333323 33322333 32323333 32333233 13333333 33233333 22323333 33233322 33333323 33333333 233
$w(3;2,3,11)$	=	129	33333322 33323333 33322333 22333333 33332323 33333233 33333332 33233323 23333333 33323233 32331333 33333223 33333233 33233333 33323333 23323333
$w(3;2,3,12)$	=	150	33333333 33323233 23333333 33323223 33233333 33333323 33333223 23333333 33332333 32323333 33333332 23333333 33233333 32333332 12233333 33333322 33332333 23333333 33332
$w(3;2,3,13)$	=	171	33333333 33332333 33323333 33332232 23333333 33233333 23333333 32332323 33333323 33333332 32333333 33333321 33333332 23233333 33323233 33223333 33332332 32333333 32333333 33233333 33233333 33
Continued on Next Page...			

Table 4.4: VAN DER WAERDEN NUMBERS FOUND BY US

$w(3;2,5,5)$	=	180	33232332 32223233 32222322 22322323 33232223 33323333 23323222 32333222 23222232 23233323 22233332 33332333 32223233 32322322 22322223 33232223 23323333 23333212 32333232 23222232 22233323 22232332 223
$w(4;2,2,3,8)$	=	83	44444434 44433434 44434443 43444334 44444144 24344344 44434344 44344434 44443344 44434444 44
$w(4;2,2,3,9)$	=	99	43443444 44444343 44343444 44443433 43444344 44442444 44444144 44443444 34334344 44444343 44343444 44444344 34
$w(4;2,2,3,10)$	=	119	34434444 43443444 44444434 44443344 44444344 44334424 44344444 44433444 44444341 43444444 34344344 44444334 44434444 44434444 344444
$w(4;2,2,4,5)$	=	75	43434444 34434441 33343343 33444434 43444433 34334333 44443443 44443334 33443424 44
$w(4;2,2,4,6)$	=	93	33343344 44433434 34444343 33444434 44434444 43334444 34333423 43314444 43433344 44343344 44343444 3343
$w(4;2,3,3,5)$	=	86	43433444 34444224 33232444 43442424 32244232 43434444 14444343 42324422 34242443 44442324 34224
$w(5;2,2,2,3,4)$	=	29	54554555 44143555 45544255 5445
$w(5;2,2,2,3,5)$	=	44	55544545 55454425 55345555 45555144 54555454 455
$w(5;2,2,2,3,6)$	=	56	45555545 55545455 54455555 45551423 44555554 55544555 5545555
$w(5;2,2,2,3,7)$	=	72	55555544 54555455 55552445 44555545 55515555 45555445 44355555 54555454 4555555
$w(5;2,2,2,3,8)$	=	88	55455455 55544555 45455554 55555535 55555454 41455555 55454455 55545555 55255544 55455555 4555545
$w(5;2,2,2,4,4)$	=	54	54554544 45544454 55255454 44554445 45315545 44455444 54554
$w(5;2,2,2,4,5)$	=	79	55554554 55554445 44544455 55455455 55444544 54442555 35555155 44454454 44555545 445555
$w(5;2,2,3,3,4)$	=	63	55443453 53543545 55332335 45553455 54543144 55535335 35445553 544343

Continued on Next Page...

Table 4.4: VAN DER WAERDEN NUMBERS FOUND BY US

$w(6;2,2,2,2,3,3)$	=	21	66556655 43216655 6655
$w(6;2,2,2,2,3,4)$	=	33	65656655 46566366 56215565 66655666
$w(6;2,2,2,2,3,5)$	=	50	56665655 65666525 66636666 46665565 56166565 56665666 6
$w(6;2,2,2,2,3,6)$	=	60	66665666 66556665 66666552 35466656 66665566 65656666 56616656 665
$w(6;2,2,2,2,4,4)$	=	56	56656555 66555656 63665655 56655565 66166465 25665556 5666555
$w(6;2,2,2,3,3,3)$	=	42	55446465 56655646 44531246 46556655 64644565 4
$w(7;2,2,2,2,2,3,3)$	=	24	67766776 16345727 6677667
$w(7;2,2,2,2,2,3,4)$	=	36	77676677 76646277 57773616 67776676 777
$w(8;2,2,2,2,2,2,3,3)$	=	25	87877883 78126578 77488787
$w(9;2,2,2,2,2,2,2,3,3)$	=	28	89899828 99597148 86889399 898

4.6 Van der Waerden numbers known so far

Table 4.5 contains a complete listing of known van der Waerden numbers.

Table 4.5: VAN DER WAERDEN NUMBERS KNOWN SO FAR

$w(r;k_1,k_2,\cdots,k_r)$		REFERENCE
$w(2;3,3)$	9	CHVÁTAL [9]
$w(2;3,4)$	18	CHVÁTAL [9]
$w(2;3,5)$	22	CHVÁTAL [9]
$w(2;3,6)$	32	CHVÁTAL [9]
$w(2;3,7)$	46	CHVÁTAL [9]
$w(2;3,8)$	58	BEELER AND O'NEIL [6]
$w(2;3,9)$	77	BEELER AND O'NEIL [6]
$w(2;3,10)$	97	BEELER AND O'NEIL [6]
$w(2;3,11)$	114	LANDMAN, ROBERTSON AND CULVER [41]
$w(2;3,12)$	135	LANDMAN, ROBERTSON AND CULVER [41]
$w(2;3,13)$	160	LANDMAN, ROBERTSON AND CULVER [41]
$w(2;3,14)$	186	KOURIL [39]
$w(2;3,15)$	218	KOURIL [39]
$w(2;3,16)$	238	KOURIL [39]
$w(2;4,4)$	35	CHVÁTAL [9]
		Continued on Next Page...

Table 4.5: VAN DER WAERDEN NUMBERS KNOWN SO FAR

$w(2;4,5)$	55	CHVÁTAL [9]
$w(2;4,6)$	73	BEELER AND O'NEIL [6]
$w(2;4,7)$	109	BEELER [5]
$w(2;4,8)$	146	KOURIL [39]
$w(2;5,5)$	178	STEVENS AND SHANTARAM [59]
$w(2;5,6)$	206	KOURIL [39]
$w(2;6,6)$	1132	KOURIL AND PAUL [40]
$w(3;2,3,3)$	14	BROWN [7]
$w(3;2,3,4)$	21	BROWN [7]
$w(3;2,3,5)$	32	BROWN [7]
$w(3;2,3,6)$	40	BROWN [7]
$w(3;2,3,7)$	55	LANDMAN, ROBERTSON AND CULVER [41]
$w(3;2,3,8)$	72	KOURIL [39]
$w(3;2,3,9)$	**90**	AHMED [1]
$w(3;2,3,10)$	**108**	AHMED [1]
$w(3;2,3,11)$	**129**	AHMED [1]
$w(3;2,3,12)$	**150**	AHMED [1]
$w(3;2,3,13)$	**171**	AHMED [1]
$w(3;2,4,4)$	40	BROWN [7]
$w(3;2,4,5)$	71	BROWN [7]
$w(3;2,4,6)$	83	LANDMAN, ROBERTSON AND CULVER [41]
$w(3;2,4,7)$	119	KOURIL [39]
$w(3;2,5,5)$	**180**	AHMED [1]
$w(3;3,3,3)$	27	CHVÁTAL [9]
$w(3;3,3,4)$	51	BEELER AND O'NEIL [6]
$w(3;3,3,5)$	80	LANDMAN, ROBERTSON AND CULVER [41]
$w(3;3,4,4)$	89	LANDMAN, ROBERTSON AND CULVER [41]
$w(4;2,2,3,3)$	17	BROWN [7]
$w(4;2,2,3,4)$	25	BROWN [7]
$w(4;2,2,3,5)$	43	BROWN [7]
$w(4;2,2,3,6)$	48	LANDMAN, ROBERTSON AND CULVER [41]
$w(4;2,2,3,7)$	65	LANDMAN, ROBERTSON AND CULVER [41]
$w(4;2,2,3,8)$	**83**	AHMED [1]
$w(4;2,2,3,9)$	**99**	AHMED [1]
$w(4;2,2,3,10)$	**119**	AHMED [1]
$w(4;2,2,4,4)$	53	BROWN [7]
$w(4;2,2,4,5)$	**75**	AHMED [1]
$w(4;2,2,4,6)$	**93**	AHMED [1]
		Continued on Next Page...

Table 4.5: VAN DER WAERDEN NUMBERS KNOWN SO FAR

$w(4;2,3,3,3)$	40	BROWN [7]
$w(4;2,3,3,4)$	60	LANDMAN, ROBERTSON AND CULVER [41]
$w(4;2,3,3,5)$	**86**	AHMED [1]
$w(4;3,3,3,3)$	76	BEELER AND O'NEIL [6]
$w(5;2,2,2,3,3)$	20	LANDMAN, ROBERTSON AND CULVER [41]
$w(5;2,2,2,3,4)$	**29**	AHMED [1]
$w(5;2,2,2,3,5)$	**44**	AHMED [1]
$w(5;2,2,2,3,6)$	**56**	AHMED [1]
$w(5;2,2,2,3,7)$	**72**	AHMED [1]
$w(5;2,2,2,3,8)$	**88**	AHMED [1]
$w(5;2,2,2,4,4)$	**54**	AHMED [1]
$w(5;2,2,2,4,5)$	**79**	AHMED [1]
$w(5;2,2,3,3,3)$	41	LANDMAN, ROBERTSON AND CULVER [41]
$w(5;2,2,3,3,4)$	**63**	AHMED [1]
$w(6;2,2,2,2,3,3)$	**21**	AHMED [1]
$w(6;2,2,2,2,3,4)$	**33**	AHMED [1]
$w(6;2,2,2,2,3,5)$	**50**	AHMED [1]
$w(6;2,2,2,2,3,6)$	**60**	AHMED [1]
$w(6;2,2,2,2,4,4)$	**56**	AHMED [1]
$w(6;2,2,2,3,3,3)$	**42**	AHMED [1]
$w(7;2,2,2,2,2,3,3)$	**24**	AHMED [1]
$w(7;2,2,2,2,2,3,4)$	**36**	AHMED [1]
$w(8;2,2,2,2,2,2,3,3)$	**25**	AHMED [1]
$w(9;2,2,2,2,2,2,2,3,3)$	**28**	AHMED [1]

4.7 Immediate future work

(*i*) Computing $w(2;3,17)$, $w(2;4,9)$, and $w(2;5,7)$,

(*ii*) Computing $w(5;3,3,3,3,3)$: which is $\geqslant 171$ [32],

(*iii*) Computing $w(3;4,4,4)$ the current lower bound ($\geqslant 293$) of which is 30 years old [52].

Chapter 5

Conclusion

In this chapter, we describe the summary of the thesis and future work in this direction.

5.1 Summary of the thesis work

We have contributed the following:

(i) We have presented an improved variant of the DPLL algorithm.

(ii) We have described efficient implementation of our version of DPLL.

(iii) We have computated thirty new van der Waerden numbers.

(iv) We have done a survey of some extremal properties of random k-SAT formulas and described two easily verifiable counting conditions under which a k-SAT formula is satisfiable.

(v) We have done a survey of the known deterministics k-SAT algorithms and described some of them in order of running times.

5.2 What we have not done?

(i) Conflict-clause recording,

(ii) VSIDS branching rule and random restarts

5.3 Future work

(i) Using the solver in attempts to compute new van der Waerden numbers and similar partition-related problems, for example, computing the 5th Schur number [1] $s(5)$. It can be also used in attempts to compute Ramsey numbers[2] $r(m, n)$.

(ii) Implementation of new ideas in branching rules.

(iii) Implementation of new ideas for parallel processing.

(iv) Implementation of new ideas on the data structure.

[1] A *Schur number* $s(k)$ is the largest integer m such that $\{1, 2, \cdots, m\}$ can be partitioned into k sum-free sets (A set S is sum-free if the intersection of S and $S + S$ is empty).

[2] A *Ramsey number* $r(m, n)$ is the minimum integer ν such that all undirected graphs of order ν contain a complete subgraph (all vertices are adjacent to each other) of order m or an independent set (no vertices are adjacent to each other) of order n.

Appendix A

Some satisfiable instances of SAT

In this section, we discuss some easily verifiable counting conditions under which a SAT formula is satisfiable. In each case, we discuss the condition and an efficient algorithm to find a satisfying assignment. We also discuss the optimality of the conditions and compare them mutually by examples.

A.1 Counting clauses

A.1.1 The condition

Theorem A.1.1 provides a simple condition for satisfiability of a SAT formula. The proof of the condition and an efficient algorithm to find a satisfying assignment (as described in the following section) are implicit in Erdös and Selfridge [24].

THEOREM A.1.1. *If a formula F satisfies the condition*

$$\sum_{C \in F} 2^{-|C|} < 1, \tag{A.1}$$

then F is satisfiable.

Proof of theorem A.1.1. Let x be an unassigned variable in F. Let F_0 be the set of clauses that contain $\overline{x}$ as a literal, and F_1 be the set of clauses that contain x as a literal. Then,

$$\sum_{C\in F|x} 2^{-|C|} = \sum_{C\in F_0} 2^{-|C|+1} + \sum_{C\in F-(F_0\cup F_1)} 2^{-|C|} \tag{A.2}$$

$$\sum_{C\in F|\overline{x}} 2^{-|C|} = \sum_{C\in F_1} 2^{-|C|+1} + \sum_{C\in F-(F_0\cup F_1)} 2^{-|C|} \tag{A.3}$$

From (A.2) and (A.3), we get:

$$\frac{1}{2}\left(\sum_{c\in F|x} 2^{-|C|} + \sum_{C\in F|\overline{x}} 2^{-|C|}\right) = \sum_{C\in F} 2^{-|C|} \tag{A.4}$$

From (A.4), since F satisfies (A.1), at least one of $F|x$ or $F|\overline{x}$ satisfies (A.1) in place of F. So, we set $x = \texttt{TRUE}$, and $F = F|x$ if $\sum_{C\in F|x} 2^{-|C|} \leqslant \sum_{C\in F|\overline{x}} 2^{-|C|}$; otherwise we set $x = \texttt{FALSE}$, and $F = F|\overline{x}$. Since we proceed satisfying (A.1), the assignment obtained at the end is satisfying. □

A.1.2 Optimality of the condition

The result in Theorem A.1.1 is tight since there are unsatisfiable SAT formulas with $\sum_{C\in F} 2^{-|C|} = 1$. For example, let F be a SAT formula with variables x_1,x_2, and x_3 and clauses $\{x_1, x_2, x_3\}$, $\{x_1, x_2, \overline{x}_3\}$, $\{x_1, \overline{x}_2, x_3\}$, $\{x_1, \overline{x}_2, \overline{x}_3\}$, $\{\overline{x}_1, x_2\}$, and $\{\overline{x}_1, \overline{x}_2\}$. Here, $\sum_{C\in F} 2^{-|C|} = 1$ and F is unsatisfiable.

A.2 Counting number of occurrences of variables

A.2.1 The condition

Let r, s-SAT denote the class of instances with exactly r literals per clause and each variable x appearing either as literal x or as literal $\overline{x}$ at most s times.

THEOREM A.2.1 (Tovey [62]). *Every instance of k, k-SAT is satisfiable.*

In the proof of Theorem A.2.1, we will require a few definitions. A graph is called *bipartite* if its vertices can be labeled "left" and "right" in such a way that each edge has one end among the left vertices and the other end among the right vertices. A *matching* M in a graph G is a set of pairwise non-adjacent edges (no two edges have a common vertex). A *cover* in a graph G is a subset K of the vertices such that every edge of G has at least one end in K. We state the following theorem, as this will be used to prove Theorem A.2.1.

THEOREM A.2.2 (König-Egerváry [38, 23]). *In a bipartite graph, the largest number of edges in a matching is equal to the smallest number of vertices in a cover.*

Proof of theorem A.2.1. Given a k, k-SAT formula F with clauses $C_1, C_2, \cdots, C_m$ over variables $x_1, x_2, \cdots, x_n$, we construct a bipartite graph G with $C_1, C_2, \cdots, C_m$ as the left nodes, $x_1, x_2, \cdots, x_n$ as the right nodes, and by adding an edge between C_i and x_j if and only if $x_j \in C_i$ or $\overline{x}_j \in C_i$. Let $var(C_i)$ denote $\{x : x \in C \text{ or } \overline{x} \in C\}$.

Given $I \subseteq \{1, 2, \cdots, m\}$, let t equal the number of pairs (x, C_i) such that $i \in I$ and C_i contains either x or $\overline{x}$ as a literal. Since every clause contains exactly k literals, we have

$$t = |I|k \tag{A.5}$$

Again, every variable occurs at most k times. So,

$$t \leqslant k|\bigcup_{i \in I} var(C_i)| \tag{A.6}$$

From (A.5) and (A.6), we get

$$|\bigcup_{i \in I} var(C_i)| \geqslant |I| \tag{A.7}$$

We want to show that if G satisfies condition (A.7) for all $I \subseteq \{1, 2, \cdots, m\}$, then G has a matching of size m. Suppose G does not have a matching of size m. We show that there exists a set $J \subseteq \{1, 2, \cdots, m\}$ such that G does not satisfy condition (A.7) for J. Let $\mathbb{M}$ be a matching in G with the largest number of edges such that $|\mathbb{M}| < m$. Let $\mathbb{K}$ be a cover in G with the smallest number of vertices. By Theorem A.2.2, $|\mathbb{M}| = |\mathbb{K}|$, and so $|\mathbb{K}| < m$. From the set $\{1, 2, \cdots, m\}$, we put $j \in J$ if and only if C_j is not in $\mathbb{K}$. So, all the edges incident on vertices C_j, where $j \in J$, are covered by the $|\mathbb{K}| - (m - |J|)$ right vertices in $\mathbb{K}$. So,

$$|\bigcup_{j \in J} var(C_j)| = |\mathbb{K}| - m + |J| < |J|.$$

Therefore, G has a matching of size m.

If G has a matching $\mathbb{M}$ of size m, then every C_i is matched to a distinct x_j. For each edge (C_i, x_j) in $\mathbb{M}$, set x_j to `TRUE` if $x_j \in C_i$, and x_j to `FALSE` if $\overline{x}_j \in C_i$. Hence,

F is satisfiable. $\square$

Graph G has $m+n$ vertices and at most kn edges. A matching of size m can be computed in time $\mathcal{O}\left((m+n)^{1/2}kn\right)$ using the Hopcroft-Karp Algorithm [33]. Then from the matching, we can obtain a satisfying assignment.

A.2.2 Optimality of the condition

For $k=3$, the condition in Theorem A.2.1 is tight. Let F be a 3-SAT formula with variables x, y, and z and clauses: $\{x,y,z\}$, $\{x,y,\overline{z}\}$, $\{x,\overline{y},z\}$, $\{x,\overline{y},\overline{z}\}$, $\{\overline{x},y,z\}$, $\{\overline{x},y,\overline{z}\}$, $\{\overline{x},\overline{y},z\}$, and $\{\overline{x},\overline{y},\overline{z}\}$. This formula is unsatisfiable and each variable appears 8 times in the formula. Now, we construct a 3,4-SAT instance F_1 from F, which is unsatisfiable.

For $i=1,\cdots,8$, we replace the i-th occurrence of x by new variable x_i, the i-th occurrence of y by new variable y_i, and the i-th occurrence of z by new variable z_i. We add the following 8 clauses to F_1:

$$\{x_1,y_1,z_1\},\quad \{x_2,y_2,\overline{z}_2\},\quad \{x_3,\overline{y}_3,z_3\},\quad \{x_4,\overline{y}_4,\overline{z}_4\},$$
$$\{\overline{x}_5,y_5,z_5\},\quad \{\overline{x}_6,y_6,\overline{z}_6\},\quad \{\overline{x}_7,\overline{y}_7,z_7\},\quad \{\overline{x}_8,\overline{y}_8,\overline{z}_8\}.$$

For $i=1,\cdots,8$, we introduce variables p_i, q_i, and r_i, and add the following 24 clauses to F_1:

$$\{x_1,\overline{x}_2,\overline{p}_1\},\quad \{x_2,\overline{x}_3,\overline{p}_2\},\quad \{x_3,\overline{x}_4,\overline{p}_3\},\quad \{x_4,\overline{x}_5,\overline{p}_4\},$$
$$\{x_5,\overline{x}_6,\overline{p}_5\},\quad \{x_6,\overline{x}_7,\overline{p}_6\},\quad \{x_7,\overline{x}_8,\overline{p}_7\},\quad \{x_8,\overline{x}_1,\overline{p}_8\},$$

$$\{y_1,\overline{y}_2,\overline{q}_1\},\quad \{y_2,\overline{y}_3,\overline{q}_2\},\quad \{y_3,\overline{y}_4,\overline{q}_3\},\quad \{y_4,\overline{y}_5,\overline{q}_4\},$$
$$\{y_5,\overline{y}_6,\overline{q}_5\},\quad \{y_6,\overline{y}_7,\overline{q}_6\},\quad \{y_7,\overline{y}_8,\overline{q}_7\},\quad \{y_8,\overline{y}_1,\overline{q}_8\},$$

$$\{z_1, \overline{z}_2, \overline{r}_1\}, \quad \{z_2, \overline{z}_3, \overline{r}_2\}, \quad \{z_3, \overline{z}_4, \overline{r}_3\}, \quad \{z_4, \overline{z}_5, \overline{r}_4\},$$
$$\{z_5, \overline{z}_6, \overline{r}_5\}, \quad \{z_6, \overline{z}_7, \overline{r}_6\}, \quad \{z_7, \overline{z}_8, \overline{r}_7\}, \quad \{z_8, \overline{z}_1, \overline{r}_8\},$$

To force $x_1, \cdots, x_8$ to all TRUE or all FALSE, we need to force each of $p_1, \cdots, p_8$ to TRUE. To force p_1 to TRUE, we introduce variables a_1, a_2, a_3, b_1, b_2, b_3, d_1, d_2, and d_3 and the following 13 clauses:

$$\{p_1, a_1, b_1\}, \quad \{d_1, a_1, \overline{b}_1\}, \quad \{d_1, \overline{a}_1, b_1\}, \quad \{d_1, \overline{a}_1, \overline{b}_1\},$$
$$\{p_1, a_2, b_2\}, \quad \{d_2, a_2, \overline{b}_2\}, \quad \{d_2, \overline{a}_2, b_2\}, \quad \{d_2, \overline{a}_2, \overline{b}_2\},$$
$$\{p_1, a_3, b_3\}, \quad \{d_3, a_3, \overline{b}_3\}, \quad \{d_3, \overline{a}_3, b_3\}, \quad \{d_3, \overline{a}_3, \overline{b}_3\},$$
$$\{\overline{d}_1, \overline{d}_2, \overline{d}_3\},$$

In this way, variables $p_1, \cdots, p_8$, can all be forced to TRUE with 72 new variables and 104 new clauses. We can do the same to force $y_1, \cdots, y_8$ to all TRUE or all FALSE, and $z_1, \cdots, z_8$ to all TRUE or all FALSE.

Finally, we get an unsatisfiable 3,4-SAT instance F_1 with 344 clauses over 264 variables.

For $k > 3$, the condition in Theorem A.2.1 is not tight. For example, we do not have an unsatisfiable instance of $4, 5$-SAT.

A.3 Comparing the conditions by example

EXAMPLE A.3.1. *Here we give an example of a satisfiable formula that satisfies the condition in Theorem A.1.1, but not the conditions given by Theorem A.2.1. Let F_1 be a 4-SAT formula with 15 clauses over the variables x_1, x_2, x_3, and x_4.*

$$\{x_1,x_2,x_3,x_4\}, \quad \{x_1,x_2,x_3,\overline{x}_4\}, \quad \{x_1,x_2,\overline{x}_3,x_4\}, \quad \{x_1,x_2,\overline{x}_3,\overline{x}_4\}, \quad \{x_1,\overline{x}_2,x_3,x_4\},$$
$$\{x_1,\overline{x}_2,x_3,\overline{x}_4\}, \quad \{x_1,\overline{x}_2,\overline{x}_3,x_4\}, \quad \{x_1,\overline{x}_2,\overline{x}_3,\overline{x}_4\}, \quad \{\overline{x}_1,x_2,x_3,x_4\}, \quad \{\overline{x}_1,x_2,x_3,\overline{x}_4\},$$
$$\{\overline{x}_1,x_2,\overline{x}_3,x_4\}, \quad \{\overline{x}_1,x_2,\overline{x}_3,\overline{x}_4\}, \quad \{\overline{x}_1,\overline{x}_2,x_3,x_4\}, \quad \{\overline{x}_1,\overline{x}_2,x_3,\overline{x}_4\}, \quad \{\overline{x}_1,\overline{x}_2,\overline{x}_3,x_4\}.$$

Formula F_1 is satisfied by $\{x_1 \mapsto \text{TRUE}, x_2 \mapsto \text{TRUE}, x_3 \mapsto \text{TRUE}, x_4 \mapsto \text{TRUE}\}$. *In* F_1,

- *The number of clauses is* 15, *which is less than* 2^4. *So the condition in Theorem A.1.1 is satisfied.*
- *Each variable* x_i *for* $1 \leqslant i \leqslant 4$, *occurs* 15 *times (which is bigger than* 4*). So the condition in Theorem A.2.1 is not satisfied.*

EXAMPLE A.3.2. *Here we give an example of a satisfiable formula that satisfies the condition in Theorem A.2.1, but not the other condition given by Theorem A.1.1. Let F_2 be a 3-SAT formula with variables* $x_1, \cdots, x_9$ *and clauses*

$$\{x_1,x_2,x_3\}, \quad \{x_1,x_2,\overline{x}_3\}, \quad \{x_1,\overline{x}_2,x_3\},$$
$$\{x_4,x_5,x_6\}, \quad \{x_4,x_5,\overline{x}_6\}, \quad \{x_4,\overline{x}_5,x_6\},$$
$$\{x_7,x_8,x_9\}, \quad \{x_7,x_8,\overline{x}_9\}, \quad \{x_7,\overline{x}_8,x_9\},$$

Here, F_2 is satisfied by $\{x_1 \mapsto \text{TRUE}, x_4 \mapsto \text{TRUE}, x_7 \mapsto \text{TRUE}\}$.

- *The number of clauses is* 9, *which is greater than* $2^3 - 1$. *So the condition in Theorem A.1.1 is not satisfied.*
- *Each variable occurs exactly* 3 *times in the formula. So the condition in Theorem A.2.1 is satisfied.*

Appendix B

Deterministic k-SAT algorithms other than DPLL

In this section, we describe some known deterministic algorithms (other than DPLL) for k-SAT. We briefly discuss the ideas behind these algorithms.

B.1 2-SAT algorithms

Cook [12] observed (from Davis-Putnam [19]) that 2-SAT can be solved in polynomial time.

B.1.1 Polynomial-time algorithm based on Davis-Putnam [19]

Two clauses C_1 and C_2 are said to *clash* if there is exactly one literal u, such that $u \in C_1$ and $\overline{u} \in C_2$. If C_1 and C_2 clash, then their *resolvent* is defined as $C_1 \cup C_2 - \{u, \overline{u}\}$ and is denoted by $C_1 \nabla C_2$. If clauses C_1 and C_2 are satisfied by some truth assignment z, then their resolvent is also satisfied by z. Adding $C_1 \nabla C_2$ does not change the satisfiability status of the formula. If two clauses of length at most two clash, then their resolvent is also of length at most two. So if we keep adding resolvents to a ($\leqslant$ 2)-SAT formula F over n variables, then the resulting formula may have at most $1 + 2n + 4\binom{n}{2} = 2n^2 + 1$ clauses. Thus the process terminates adding at most $\mathcal{O}(n^2)$ resolvents. If we encounter an empty clause, then F is not satisfiable; otherwise it is satisfiable.

B.1.2 Limited-backtracking DPLL-like polynomial-time algorithm

Even, Itai and Shamir [25] suggested a limited-backtracking DPLL-like algorithm (Algorithm B.19) for 2-SAT that runs in polynomial time.

Algorithm B.19 SOLVING 2-SAT WITH LIMITED BACKTRACKING

procedure LIMITED-BACKTRACKING-DPLL-2SAT(F)
  while there is a clause of length at most one in F **do**
    if F contains an empty clause **then return** UNSATISFIABLE
    if F contains a unit clause $\{u\}$ **then** $F = F|u$
  end while
  if F is empty **then return** SATISFIABLE
  choose an unassigned literal u
  $F' = F|u$
  while there is a unit clause $\{v\}$ in F' **do** $F' = F'|v$
  if F' does not contain an empty clause **then**
    return LIMITED-BACKTRACKING-DPLL-2SAT(F')
  else
    return LIMITED-BACKTRACKING-DPLL-2SAT($F|\overline{u}$)
  end if
end procedure

The idea is that if setting a literal u to TRUE does not immediately lead to a contradiction by unit-propagation, then the assignment may be fixed. In that case, the set of clauses in the resulting formula is a subset of the set of clauses in the original formula and the resulting formula is satisfiable if and only if the original formula is satisfiable.

B.1.3 A linear-time algorithm

Aspvall, Plass and Tarjan[3] came up with a linear time algorithm for ($\leqslant 2$)-SAT as described in Algorithm B.20. Let F be a ($\leqslant 2$)-SAT formula with m clauses over variables $\{x_1, \cdots, x_n\}$. Let $G(F)$ be the directed graph, as defined in [3], with vertices $\{x_1, \cdots, x_n, \overline{x}_1, \cdots, \overline{x}_n\}$ and edges $\{(u, v) | \{\overline{u}, v\} \in F\}$. So $G(F)$ has $2n$ vertices and at most $2m$ directed edges.

Let $u \rightsquigarrow v$ denote a directed walk $u \to \cdots \to v$ in $G(F)$. We observe that if $u \rightsquigarrow v$, then every satisfying assignment setting u to TRUE has to set v to TRUE as well. If $u \rightsquigarrow \overline{u}$, then every satisfying assignment sets u to FALSE. If $u \rightsquigarrow v \rightsquigarrow u$, then every satisfying assignment sets u and v to the same truth value. Also by construction of $G(F)$, we observe that $u \rightsquigarrow v$ if and only if $\overline{v} \rightsquigarrow \overline{u}$.

Lemma B.1.1 (Aspvall, Plass, Tarjan [3]). *A 2-SAT formula F is unsatisfiable if and only if $G(F)$ contains a directed walk $x \rightsquigarrow \overline{x} \rightsquigarrow x$.*

Proof of lemma B.1.1. Let F be a 2-SAT formula over n variables. The resolvent of clauses $\{x, u\}$ and $\{\overline{x}, v\}$ in F is $\{u, v\}$. Adding the resolvent to F will introduce edges $\overline{u} \to v$ and $\overline{v} \to u$ to $G(F)$. But $\overline{u} \to x \to v$ and $\overline{v} \to \overline{x} \to u$ were already in $G(F)$. Let F' be the formula obtained after adding resolvents to F as long as possible. We have $u \to v$ in $G(F')$ if and only if we have $u \rightsquigarrow v$ in $G(F)$ with $u \neq v$. We know that F is unsatisfiable if and only if F' contains $\{x\}$ and $\{\overline{x}\}$. Formula F'

containing $\{x\}$ and $\{\overline{x}\}$ is equivalent to the existence of $x \to \overline{x} \to x$ in $G(F')$, which in turn, is equivalent to $x \rightsquigarrow \overline{x} \rightsquigarrow x$ in $G(F)$. □

Algorithm B.20 (Aspvall, Plass, Tarjan [3]) constructs a satisfying assignment in time $\mathcal{O}(m+n)$ provided $G(F)$ contains no directed walk of the form $x \rightsquigarrow \overline{x} \rightsquigarrow x$.

A graph is *strongly connected* if every two vertices are mutually reachable. The maximal strongly connected subgraphs of a graph are vertex-disjoint and are called *strongly connected components*. The strongly connected components of a directed graph can be computed in time $\mathcal{O}(m+n)$ (Tarjan [61]) using depth-first-search.

If S_1 and S_2 are strongly connected components such that an edge leads from a vertex in S_1 to a vertex in S_2, then S_1 is a *predecessor* of S_2 and S_2 is a *successor* of S_1. Each clause $\{u, v\}$ in F contributes two edges $\overline{u} \to v$ and $\overline{v} \to u$ in $G(F)$. So, for each strongly connected component S in $G(F)$, there is a strongly connected component $\overline{S}$ (which is S with labels of vertices complemented and directions of edges reversed) in $G(F)$. If S_1 and S_2 are two strongly connected components in $G(F)$ and S_1 is a predecessor of S_2, then $\overline{S}_1$ is a successor of $\overline{S}_2$.

Algorithm B.20 SOLVING ($\leqslant$2)-SAT IN $\mathcal{O}(m+n)$ TIME

procedure LINEAR2SAT(F)
    $\mathcal{S}$ = strongly connected components of $G(F)$
    for each unassigned component S in $\mathcal{S}$ **do**
        if S contains literals u and $\overline{u}$ as vertices **then**
            return UNSATISFIABLE
        end if
        set each literal labelling vertices of S to TRUE
        set each literal labelling vertices of $\overline{S}$ to FALSE
    end for
    return SATISFIABLE
end procedure

If any strongly connected component S does not contain two vertices labelled by a literal and its complement, then $S \neq \overline{S}$.

If any strongly connected component is set to `TRUE`, then its successors are also set to `TRUE`. If any strongly connected component is set to `FALSE`, then its predecessors are also set to `FALSE`. So complementary components have complementary truth values and no path leads from a `TRUE` component to a `FALSE` component.

B.2 Monien-Speckenmeyer Algorithm

Monien and Speckenmeyer [48] came up with the very first algorithms for k-SAT that run in less than 2^n steps. The basic idea was to branch on a shortest unsatisfied clause. Algorithms B.21, B.22, and B.23 are three variants of Monien-Speckenmeyer algorithm with gradual improvements in running time. In this section, we use $\mathcal{O}^*(c^n)$ (where $c > 1$) instead of $\mathcal{O}(c^n \cdot poly(n))$ to indicate that the polynomial factor is suppressed.

B.2.1 $\mathcal{O}^*\left((2^k - 1)^{n/k}\right)$-time k-SAT algorithm

This algorithm comes from the simple observation that any clause of length k has $2^k - 1$ possible satisfying assignments.

Algorithm B.21 SOLVING K-SAT IN TIME $\mathcal{O}^*(c_k^n)$ WITH $c_k = (2^k - 1)^{1/k}$

procedure MS1(F)
  if $F = \emptyset$ **then return** SATISFIABLE
  if F contains an empty clause **then return** UNSATISFIABLE
  if F is a 2-SAT **then return** LINEAR2SAT(F)
  C = shortest unsatisfied clause $\{u_1, u_2, \cdots, u_\ell\}$ in F
  for each of the $2^\ell - 1$ satisfying assignments of C **do**
    compute simplified formula F_i
    if MS1(F_i)=SATISFIABLE **then return** SATISFIABLE
  end for
  return UNSATISFIABLE
end procedure

Let $T_k(n)$ be the complexity of Algorithm B.21. Now, ignoring polynomial factors, we get the recurrence

$$T_k(n) \leqslant (2^k - 1)T_k(n - k),$$

which gives the upper bound $\mathcal{O}^*\left(c_k^n\right)$ with $c_k = (2^k - 1)^{1/k}$. In particular, the running time for 3-SAT is $\mathcal{O}^*\left(1.913^n\right)$.

B.2.2 $\mathcal{O}^*\left(\beta_k^n\right)$-time k-SAT algorithm, where β_k is the biggest number satisfying $\beta_k = 2 - 1/\beta_k^k$

Algorithm B.22 K-SAT ALGORITHM (FASTER THAN ALGORITHM B.21)

```
1: procedure MS2(F)
2:     if F = ∅ then return SATISFIABLE
3:     if F contains an empty clause then return UNSATISFIABLE
4:     C = shortest unsatisfied clause {u_1, u_2, ··· , u_ℓ} in F
5:     for i = 1 to ℓ do
6:         F_i = {C − {u_1, ··· , u_{i−1}, ū_i} : C ∈ F, C ∩ {ū_1, ··· , ū_{i−1}, u_i} = ∅}
7:         if MS2(F_i)=SATISFIABLE then return SATISFIABLE
8:     end for
9:     return UNSATISFIABLE
10: end procedure
```

If F consists of n variables, then each F_i for $1 \leqslant i \leqslant \ell$ (line 6 of Algorithm B.22) consists of $n - i$ variables. Let the running time be $T_k(n)$, where n is the number of yet-to-be-assigned variables. Omitting constants that lead to sub-dominant polynomial factors, we get

$$T_k(n) \leqslant T_k(n-1) + T_k(n-2) + \cdots + T_k(n-k).$$

We have the running time $\mathcal{O}^*\left(\beta_k^n\right)$, where β_k is the largest zero of

$$1 - x^{-1} - \cdots - x^{-k}.$$

In particular, for 3-SAT, $\beta_3 = 1.8393\ldots$

B.2.3 $\mathcal{O}^*(\alpha_k^n)$-time k-SAT algorithm, where α_k is the biggest number satisfying $\alpha_k = 2 - 1/\alpha_k^{k-1}$

A truth assignment z over a subset V of the set of variable is *autark* in F if and only if every clause C in F that shares one or more variables with V is satisfied by z. Determining auturkness of a given assignment is not expensive.

Algorithm B.23 K-SAT ALGORITHM (FASTER THAN ALGORITHM B.22)

procedure MS3(F)
    if $F = \emptyset$ **then return** SATISFIABLE
    if F contains an empty clause **then return** UNSATISFIABLE
    C = shortest unsatisfied clause $\{u_1, u_2, \cdots, u_\ell\}$ in F
    for $i = 1$ to ℓ **do**
        t = assignment induced by $\{u_1 \mapsto 0, u_2 \mapsto 0, \cdots, u_{i-1} \mapsto 0, u_i \mapsto 1\}$
        if t is AUTARK **then**
            $\widehat{F} = \{C : C \in F, var(C) \cap var(\{u_1, \cdots, u_i\}) = \emptyset\}$
            return MS3($\widehat{F}$)
        end if
    end for
    for $i = 1$ to ℓ **do**
        $F_i = \{C - \{u_1, \cdots, u_{i-1}, \overline{u}_i\} : C \in F, C \cap \{\overline{u}_1, \cdots, \overline{u}_{i-1}, u_i\} = \emptyset\}$
        if MS3(F_i)=SATISFIABLE **then return** SATISFIABLE
    end for
    return UNSATISFIABLE
end procedure

In Algorithm B.23, we observe that if the first `for` loop contains no autark assignment, then in the second `for` loop, every subformula F_i contains a clause of length at most $k - 1$. This behaviour is sufficient to guarantee a better estimation than the one given by Algorithm B.22. The recurrence for Algorithm B.23 is

$$T_k(n) \leqslant T_k(n-1) + T_k(n-2) + \cdots + T_k(n-k+1).$$

We have the running time $\mathcal{O}^*(\alpha_k^n)$, where α_k is the largest zero of

$$1 - x^{-1} - \cdots - x^{-k+1}.$$

In particular, for 3-SAT, $\alpha_3 = 1.618\ldots$

B.3 Local search based k-SAT algorithms

Let F be a k-SAT formula with variables $x_1, x_2, \cdots, x_n$. The *Hamming distance* between two truth assignments z_1 and z_2 is

$$\sum_{i=1}^{n} z_1(x_i) \oplus z_2(x_i).$$

The *Hamming ball* of radius r around an assignment z in $\{0,1\}^n$ is the set of all assignments whose Hamming distance to z is at most r. Each Hamming ball of radius r has $\sum_{i=0}^{r} \binom{n}{r}$ assignments in it (let this number be denoted by $V(n,r)$). From Stirling's approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, with $0 \leqslant \alpha < 1$, we get

$$\binom{n}{\alpha n} \approx \frac{1}{\sqrt{2\pi n\alpha(1-\alpha)}} \left(\frac{1}{\alpha^{\alpha}(1-\alpha)^{1-\alpha}}\right)^n \tag{B.1}$$

Function $-\alpha \log_2 \alpha - (1-\alpha)\log_2(1-\alpha)$, denoted by $h(\alpha)$, which is maximum at $\alpha = 1/2$, is known as the *binary entropy function*. With $r = \rho n$ and $0 < \rho \leqslant 1/2$, we get

$$V(n,r) \leqslant 2^{h(\rho)n}$$

A *covering code* of radius r is a subset of $\{0,1\}^n$ that covers all the 2^n assignments by Hamming balls of radius r. Constructing an optimal covering code is NP-complete. But a near-optimal covering code can be constructed using a greedy approximation algorithm, as described in [17]. For any covering code $\mathscr{C}$, we have $|\mathscr{C}| \cdot V(n,r) \geqslant 2^n$. So,

$$|\mathscr{C}| \geqslant \frac{2^n}{2^{h(\rho)n}} = 2^{(1-h(\rho))n}.$$

Algorithm B.24 LOCAL SEARCH BASED K-SAT ALGORITHM

procedure HSEARCH(F, z, r)
    if $F = \emptyset$ **then return** TRUE
    if $r \leqslant 0$ **then return** FALSE
    if F contains an empty clause **then return** FALSE
    Pick a clause C that is false under z
    for each literal $u \in C$ **do**
        if HSEARCH($F|u$, z, $r-1$)=TRUE **then return** TRUE
    end for
    return FALSE
end procedure

Once we have a covering code of radius r, for every assignment z in the covering code, we can search for a satisfying assignment locally in the Hamming ball of radius r around z. But it is not necessary to search through all $V(n,r)$ assignments inside the ball. If the formula F is not satisfied by z, then there is a clause C which is not satisfied by z. Then F has a satisfying assignment in the Hamming ball of radius r around z if and only if there is a literal u in C such that $F|u$ has a satisfying assignment in the Hamming ball of radius $r-1$ around z.

B.3.1 $\mathcal{O}^*\left(\left(\frac{2k}{k+1}\right)^n\right)$-time algorithm for k-SAT by Dantsin et al. [17]

Dantsin et al. [17] gave algorithm B.25 for k-SAT, which runs in time $\mathcal{O}^*\left(\left(\frac{2k}{k+1}\right)^n\right)$.

Algorithm B.25 LOCAL SEARCH BASED k-SAT ALGORITHM

```
procedure HAMMINGBALLSAT(F, n)
    ρ = 1/(k+1)
    Generate a covering code 𝒞 using a greedy approximation algorithm
    for each assignment z in 𝒞 do
        if HSEARCH(F, z, ρn)=TRUE then return SATISFIABLE
    end for
    return UNSATISFIABLE
end procedure
```

Function HSEARCH(F, z, ρ) runs in time $\mathcal{O}^*(k^r)$. Therefore, Algorithm B.25 has a running time:

$$
\begin{aligned}
T(n,\rho) &\leqslant poly(n)\cdot 2^{(1-h(\rho))n}\cdot k^{\rho n}\\
&= poly(n)\cdot 2^{(1-h(\rho))n}\cdot 2^{\rho n\log_2 k}\\
&= poly(n)\cdot 2^{n(1+\rho\log_2\rho+(1-\rho)\log_2(1-\rho)+\rho\log_2 k)}\\
&= poly(n)\cdot 2^{n\left(1+\frac{1}{k+1}\log_2\frac{1}{k+1}+\frac{k}{k+1}\log_2\frac{k}{k+1}+\frac{1}{k+1}\log_2 k\right)}\\
&= poly(n)\cdot 2^{n\left(1+\log_2\frac{k}{k+1}\right)} = poly(n)\cdot\left(\frac{2k}{k+1}\right)^n
\end{aligned}
$$

For 3-SAT, Algorithm B.25 runs in time $\mathcal{O}^*(1.5^n)$ (Here $\rho = 0.25$).

B.3.2 $\mathcal{O}^*(1.481^n)$-time algorithm for 3-SAT by Dantsin et al. [17]

Algorithm B.24 can be modified to run in time $\mathcal{O}^*(2.848^r)$ instead of $\mathcal{O}^*(3^r)$, which improves the running time of Algorithm B.25 to $\mathcal{O}^*(1.481^n)$ for 3-SAT. Here, the

HSEARCH$(F,\ z,\ r)$ is modified so that if there is a clause $\{u_1, u_2, u_3\}$, which is false under z and F contains a clause $\{\overline{u}_i\}$ for some i in $\{1, 2, 3\}$, then we do not run HSEARCH$(F|u_i,\ z,\ r-1)$. To estimate the number of leaves of the recursion tree, let the function be $H(r)$. The recurrence is

$$H(r) = 6 \cdot (H(r-2) + H(r-3)), \tag{B.2}$$

for $r \geqslant 3$ with $H(0) = 1$, $H(1) = 3$ and $H(2) = 9$. Now, $H(r) = \mathcal{O}^*(\alpha^r)$, where α is $\sqrt[3]{4} + \sqrt[3]{2} \approx 2.848$, the largest root of $\alpha^3 - 6\alpha - 6 = 0$. With $\rho = 0.26$, for 3-SAT, Algorithm B.25 runs in time

$$T(n, 0.26) \leqslant poly(n) \cdot \left(2.848^{0.26} \cdot 2^{1-h(0.26)}\right)^n = \mathcal{O}^*(1.481^n).$$

Brueggemann and Kern [8] improved the recurrence (B.2) to

$$H(r) = 6 \cdot H(r-2) + 5 \cdot H(r-3), \tag{B.3}$$

Here, $H(r) = \mathcal{O}^*(\beta^r)$, where β is 2.792, the largest root of $\beta^3 - 6\beta - 5 = 0$. With $\rho = 0.264$, for 3-SAT, Algorithm B.25 runs in time

$$T(n, 0.264) \leqslant poly(n) \cdot \left(2.792^{0.264} \cdot 2^{1-h(0.264)}\right)^n = \mathcal{O}^*(1.473^n).$$

Bibliography

[1] AHMED T., Some new van der Waerden numbers and some van der Waerden-type numbers, INTEGERS, **9** (2009), #A06, 65–76, MR2506138.

[2] ASAHIRO Y., IWAMA K., MIYANO E., Random generation of test instances with controlled attributes, *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, **26** (1996), 377–394.

[3] ASPVALL B., PLASS M. F., TARJAN R. E., A linear-time algorithm for testing the truth of certain quantified boolean formulas, *Information Processing Letters*, **8(3)** (1979), 121–123, MR0526451.

[4] BAYARDO R., SCHRAG R., Using CSP look-back techniques to solve real-world SAT instances, *Proceedings of the 14th Nat. (US) Conf. on Artificial Intelligence*, 1997, 203-208.

[5] BEELER M., A new van der Waerden number, *Discrete Applied Math.* **6** (1983), 207, MR0707027.

[6] BEELER M., O'NEIL P., Some new van der Waerden numbers, *Discrete Math.* **28** (1979), 135-146, MR0546646.

[7] Brown T. C., Some new van der Waerden numbers (preliminary report), *Notices American Math. Society* **21** (1974), A-432.

[8] Brueggemann T., Kern W., An improved deterministic local search algorithm for 3-SAT, *Theo. Computer Science*, **329** (2004), 303–313.

[9] Chvátal V., Some unknown van der Waerden numbers, in: Combinatorial structures and their applications, Proc. Calgary Internat. Conf., Calgary, Alta., 1969, *Gordon and Breach, New York*, 1970, 31–33, MR0266891.

[10] Chvátal V., Reed B., Mick Gets Some (The Odds Are on His Side), *Proceedings of the 33rd Annual Symposium on FOCS*, 1992, 620–627.

[11] Chvátal V., Szemerédi E., Many hard examples for resolution, *J. ACM*, **35(4)** (1988), 759–768, MR1072398.

[12] Cook S. A., The complexity of theorem proving procedures, *Third Annual Symposium on the Theory of Computing, ACM*, 1971, 151-158.

[13] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., Introduction to Algorithms, 2nd ed., *The MIT Press and McGraw-Hill*, 1990.

[14] Coudert O., On solving covering problems, *Proceedings of the ACM/IEEE Design Automation Conference*, 1996, 197–202.

[15] Coudert O., Madre J. C., New ideas for solving covering problems, *Proceedings of the ACM/IEEE Design Automation Conference*, 1995.

[16] Crawford J. M., Auton L. D., Experimental results on the crossover point in random 3-SAT, *Artificial Intelligence Journal*, **81** (1996), 1–2, MR1396824.

[17] DANTSIN E., GEORDT A., HIRSCH E. A., KANNAN R., KLEINBERG J., PAPADIMITRIOU C., RAGHAVAN P., SCHÖNING U., A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search, *Theoretical Comp. Sci.*, **289** (2002), 69–83, MR1932890.

[18] DAVIS M., LOGEMANN G., LOVELAND D., A machine program for theorem-proving, *Comm. ACM*, **5** (1962), 394–397, MR0149690.

[19] DAVIS M., PUTNAM H., A computing procedure for quantification theory, *J. ACM*, **7** (1960), 201–215, MR0134439.

[20] DIMACS Implementation Challenges, `<http://dimacs.rutgers.edu/Challenges>`.

[21] DRANSFIELD M. R., LIU L., MAREK V. W., TRUSZCZYNSKI M., Satisfiability and computing van der Waerden numbers, *The Electronic J. of Combinatorics*, **11(1)** (2004), R41, MR2097307.

[22] DUBOIS O., ANDRE P., BOUFKHAD Y., CARLIER J. SAT vs UNSAT, *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, **26** (1996), 415–436.

[23] EGERVÁRY J., Matrixok kombinatorikus tulajdonságairól, *Matematikai és Fizikai Lápok*, **38** (1931), 16–28.

[24] ERDÖS P., SELFRIDGE J. L., On a combinatorial game, *J. Combinatorial Theory, Series A*, **14** (1973), 298–301, MR0327313.

[25] EVEN S., ITAI A., SHAMIR A., On the complexity of timetable and multicommodity flow problems, *SIAM J. Computing*, **5(4)** (1976), 691–703.

[26] FREEMAN J. W., Improvements to Propositional Satisfiability Search Algorithms, *Ph. D. Thesis*, University of Pennsylvania, 1995.

[27] GAREY M., JOHNSON D., Computers and Intractability; A Guide to the Theory of NP-Completeness, *ISBN 0-7167-1045-5*, 1979, MR0519066.

[28] GASCHNIG J., Performance measurement and analysis of certain search algorithms, *Ph. D. Thesis, CMU*, 1979.

[29] GOLDBERG E., NOVIKOV Y., Berkmin: A Fast and Robust SAT Solver *DATE*, 2002.

[30] GOMES C. P., SELMAN B., KAUTZ H., Boosting Combinatorial Search Through Randomization, *Proc. of AAAI*, 1998.

[31] GOWERS T., A new proof of Szemerédi's theorem, *Geom. Funct. Anal.*, **11(3)** (2001), 465–588, MR1844079.

[32] HERWIG P., HEULE M. J. H., LAMBALGEN M. VAN, AND MAAREN H. VAN., A new method to construct lower bounds for Van der Waerden numbers, *The Electronic Journal of Combinatorics*, **14** (2007), #R6.

[33] HOPCROFT J. E., R. M. KARP, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAM Journal on Computing*, **2(4)** (1973), 225–231.

[34] HOOKER J. N., VINAY V., Branching rules for satisfiability, *J. Automat. Reason.*, **15(3)** (1995), 359–383, MR1356629.

[35] JEROSLOW R., WANG J., Solving propositional satisfiability problems, *Annals of Mathematics and AI*, **1** (1990), 167–187.

[36] KAHN, A. B. Topological sorting of large networks, *Communications of the ACM*, **5 (11)** (1962), 558–562.

[37] KAMATH A. P., KARMAKAR N. K., RAMAKRISHNAN K. G., RESENDE M. G. C., A Continuous Approach to Inductive Inference, *Math. Programming*, **57(2)** (1992), 215–238, MR1195025.

[38] KÖNIG D., Graphen und Matrizen, *Matematikai és Fizikai Lápok*, **38** (1931), 116–119.

[39] KOURIL M., A Backtracking Framework for Beowulf Clusters with an Extension to Multi-Cluster Computation and Sat Benchmark Problem Implementation, *Ph. D. Thesis*, University of Cincinnati, Engineering : Computer Science and Engineering, 2006.

[40] KOURIL M., PAUL J.L., The van der Waerden number W(2,6) is 1132, *Experimental Mathematics*, **17(1)** (2008), 53–61, MR2410115.

[41] LANDMAN B., ROBERTSON A., CULVER C., Some new exact van der Waerden numbers, *Integers: Electronic J. Combinatorial Number Theory*, **5(2)** (2005), A10, MR2192088.

[42] LI C. M., A constraint-based approach to narrow search trees for satisfiability, *Information Processing Letters*, **71** (1999), 75–80.

[43] Li C. M., Equivalent literal propagation in Davis-Putnam Procedure *Discrete Applied mathematics*, **130/2** (2003), 251–276.

[44] Li C. M., Anbulagan, Heuristics Based on Unit Propagation for Satisfiability Problems *Proceedings of 15th International Joint Conference on Artificial Intelligence*, 1997, 366–371.

[45] Lynce I., Marques-Silva J., Efficient data structures for backtrack search SAT solvers, *Annals of Mathematics and Artificial Intelligence*, **43** (2005), 137–152.

[46] Marques-Silva J. P., Sakallah K., A new search algorithm for satisfiability, *Proceedings of ACM/IEEE International Conference in Computer Aided Design*, (1996), 220–227.

[47] Marques-Silva J. P., Sakallah K., GRASP: A search algorithm for propositional satisfiability, *IEEE Transactions on Computers*, **48** (1999), 506–521.

[48] Monien B., Speckenmeyer E., Solving satisfiability in less than 2^n steps, *Discrete Applied Math.*, **10** (1985), 287–295.

[49] Moskewicz N., Madigan C., Zhao Y., Zhang L., Malik S., Engineering an efficient SAT solver, *Proceeding of the Design Automation Conference*, (2001), 530–535.

[50] Ouyang M., How good are branching rules in DPLL?, *Discrete Applied Math.*, **89(1-3)** (1998), 281–286, MR1663116.

[51] Ouyang M., Implementation of the DPLL algorithm, *PhD Thesis*, Rutgers University, 1999.

[52] RABUNG J. R., Some progression-free partitions constructed using Folkman's method, *Canad. Math. Bull.*, **22(1)** (1979), 87–91.

[53] ROBINSON J. A., A machine-oriented logic based on the resolution principle, *J. ACM*, **12(1)** (1965), 23–41.

[54] SAT Competitions web page,
`<http://www.satcompetition.org>`.

[55] SATLIB - Benchmark Problems,
`<http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html>`.

[56] SCHIEUX T., VERFAILLIE G., Nogood recording for static and dynamic constraint satisfaction problems, *Proc. Inter. Conf. Tools on AI*, 1 (1993), 48–55.

[57] SELMAN B., MITCHELL D. G., LEVESQUE H. J., Generating Hard Satisfiability Instances, *Artificial Intelligence*, **81** (1996), 17–29.

[58] SÖRENSSON N., EEN N., MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, *MiniSat Page*, `<www.minisat.se>`.

[59] STEVENS R., SHANTARAM R., Computer-generated van der Waerden partitions, *Math. Computation*, **32** (1978), 635–636, MR0491468.

[60] STALLMAN R. M., SUSSMAN G. J., Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence*, **9** (1977), 135–196.

[61] TARJAN R. E., Depth first search and linear time graph algorithms, *SIAM J. Comput.*, **1(2)** (1972), 126–160.

[62] TOVEY C. A., A simplified NP-complete satisfiability problem, *Discrete Applied Math.*, **8** (1984), 85–89.

[63] VAN DER WAERDEN B. L., Beweis einer Baudetschen Vermutung, *Nieuw Archief voor Wiskunde*, **15** (1927), 212–216.

[64] VAN DER WAERDEN B. L., How the proof of Baudet's conjecture was found, in: Studies in Pure Mathematics, Papers presented to Richard Rado on the occasion of his sixty-fifth birthday, *Academic Press, London and New York*, 1971, 251–260.

[65] VAN GELDER A., TSUJI Y. K., Satisfiability testing with more reasoning and less guessing, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, **26** (1996), 559–586, MR1423138.

[66] ZHANG H., SATO: An efficient propositional solver, *Proceedings of the International Conference on Automated Deduction*, 1997, 272–275.

[67] ZHANG H., STICKEL M., Implementing the Davis-Putnam method, *Proceedings of SAT 2000, eds. I. Gent, H. van Maaren and T. Walsh*, 2000, 309–326.

Index