Satisfiability Problem with MiniSat - Benchmarks, Comprehension and Challenges

Andries Rafael Gabriel, rafael.andries00@e-uvt.ro
Stoentel Alexandru-Eduard, alexandru.stoentel01@e-uvt.ro
Nenescu Eugeniu eugeniu.nenescu00@e-uvt.ro

West University of Timișoara, Bulevardul Vasile Pârvan 4, Timișoara 300223

Abstract. For as long as we have been alive, logic has helped us make sense of the world surrounding us, from our philosophical inquiries to its use in modern mathematics and informatics. Logic bridges the gap between formal and informal knowledge, proving essential in databases, programming languages (example (e.g), Prolog), and propositional logic. This paper focuses on a compelling area within informatics: the Boolean satisfiability problem (SAT). Our main focus will be on MiniSat, a SAT solver designed to address the challenge of satisfiability efficiently. Such solvers can tackle problems with millions of variables, benefiting research and industry.
This report will include the installation process for MiniSat, initial challenges faced with working with MiniSat, benchmarks from recent SAT competitions, mainly the Hamiltonian family of tests, a look into the algorithms found in the code of MiniSat, and suggestions on how the code might be modified to improve runtimes. Our analysis will highlight MiniSat's capabilities, while the code and benchmarks can be found in the following GitHub link:
`https://github.com/AndiSova/VF-Software-Engineering-2024-Project`

Keywords: MiniSat · SAT · algorithms.

Table of Contents

1 Introduction . . . 3
1.1 Motivation . . . 3
1.2 Scope and Objectives . . . 3
2 SAT problem and solutions . . . 4
2.1 Problem description . . . 4
2.2 MiniSat installation and first challenges . . . 4
2.3 First benchmark . . . 5
2.4 Running MiniSat on a Virtual Machine . . . 6
3 Code Documentation . . . 7
3.1 Algorithm structures . . . 7
3.2 Proposed Optimisation . . . 13
4 Experimental Results . . . 15
4.1 Hamiltonian family of problems . . . 15
5 Conclusion . . . 17
Acknowledgments. . . . 18

List of Tables

1 MiniSat benchmark results from the Hamiltonian family of tests . . . 16

List of acronyms

SAT Boolean satisfiability problem . . . 1

CDCL Conflict-Driven Clause Learning . . . 3

UML Unified Modeling Language . . . 3

CNF Conjunctive Normal Form . . . 4

WSL Windows Subsystem for Linux . . . 4

OS Operating System . . . 4

e.g example . . . 1

NP Nondeterministic Polynomial . . . 15

1 Introduction

SAT is a cornerstone in computational theory and practical applications, namely EDA, artificial intelligence, and combinatorial optimization (see works [3]). Since 2003, MiniSat has continued to be a small, efficient, and readable SAT solver, representing a perfect tool to introduce students and professionals alike to the nuances of SAT solving.

While MiniSat is often praised for its straightforward design, working with it as a beginner can present various challenges. For example, setting up MiniSat, understanding its algorithms, such as Conflict-Driven Clause Learning (CDCL), and running your first benchmark might challenge the uninitiated.

In this report, we will explore these challenges from the perspective of newly introduced users. Our study includes the steps for installing MiniSat, the steps required to run a test from the 2024 SAT competition, an analysis of MiniSats' main algorithms, possible improvements that can be made for the source code, and the results from running a family of tests using the original code.

For a better understanding of the main components of MiniSat, we will apply principles from software engineering, making use of Unified Modeling Language (UML) diagrams and detailing the code, using pseudocode, in an attempt to bridge the gap between theoretical concepts and practical implementation.

1.1 Motivation

Our main motivation for studying MiniSat stems from its influential role in advancing SAT solver technology and its accessibility as a learning tool. Since 2003, MiniSat has set a standard for efficiency in solving SAT problems, providing a straightforward and readable code that allows us to delve into its algorithms and structure.

It represents a perfect starting line for our introduction into the world of SAT solvers and their particularities, showcasing powerful algorithms, such as the CDCL algorithm, while also allowing us to explore potential improvements for the current code.

1.2 Scope and Objectives

With its ease of use in mind, without prior knowledge, the use and understanding of MiniSat might seem harder than it is. As such, our main aim for this paper will be to showcase how one can use MiniSat, as a first-time user, how to run a benchmark for the first time, how to understand the code that MiniSat runs on, and to start a discussion on how the code could be improved in the future.

2 SAT problem and solutions

2.1 Problem description

The SAT problem in propositional logic asks to determine in a given formula that combines atomic propositions using the Boolean operators "and" ($\wedge$), "or" ($\vee$), and "not" ($\neg$), whether there are truth values for the atoms in the formula such that the formula evaluates to true works [2]. In summary, the SAT problem is the challenge of determining if a logical formula, typically represented in Conjunctive Normal Form (CNF), has at least one solution.

While it is possible to be resolved by hand, after a certain number of atoms, the solution for such formulas becomes increasingly harder to be determined by ourselves alone, resulting in the need for solvers similar to MiniSat to exist. This exponential increase in difficulty for solving complex formulas has fueled the development of SAT solvers, which apply advanced algorithms to handle these formulas efficiently.

With SAT solvers like MiniSat, designed specifically to tackle these computational hurdles, we reached a level where it is possible to solve these problems within a reasonable time slot.

2.2 MiniSat installation and first challenges

Since the installation process for MiniSat on the Windows Operating System (OS) is not the most straightforward, this paper will showcase how a new user can install the solver on their machine with the following instructions:

To begin using the solver, we will go for the Windows Subsystem for Linux (WSL) approach, which allows us to run a Linux environment natively on Windows without the need for virtual machines or terminals such as Cygwin. Below are the steps followed for the installation process:

Firstly, WSL needs to be enabled, to do this, the following command is required to be run in PowerShell(mandatory, Powershell needs to be run as an Administrator):

```
wsl --install
```

This command installs the required features and the default Linux distribution (Ubuntu) system for WSL. However, in some cases, users may encounter issues where the installation hangs or doesn't complete. To resolve this, it's necessary to ensure that Windows is fully updated and that the proper components for WSL 1(or version 2 if the user chooses so) are manually enabled. This can be done again in PowerShell(running in Administrator mode) using the following commands:

```
dism.exe
```

This command was used to enable Virtual Machine Platform features. After successfully running it, the following command needs to be run to ensure WSL 1 is set as the default version.

```
wsl --set-default-version 1
```

In the possibility that neither command works, another possible solution would be to manually install the following update for Windows, wsl_update_x64, which allows Linux environments to run directly on Windows.

After enabling WSL, we need to install Ubuntu, a Linux-based OS. To do this, Ubuntu can be installed from the Microsoft Store or its home site(for this paper, version 22.04 of Ubuntu was chosen). To complete the installation process, a username and password need to be set up by the user. It is highly recommended to update Ubuntu after we are done with the setup process. We will use the following command to do so:

```
sudo apt update
```

Installing MiniSat on Ubuntu: First, we will install the MiniSat solver by running:

```
sudo apt install minisat
```

2.3 First benchmark

With MiniSat successfully installed, we can run the solver and evaluate its performance. To ensure that MiniSat is working properly, we can first test its basic functionality by running a sample input. To do this, we need to open the Ubuntu terminal and type the following command:

```
minisat
```

This should display MiniSat's usage instructions, confirming that the installation was indeed successful.

Next, to run a benchmark, we use test instances from recent SAT competitions. These competitions provide SAT instances that can be used to evaluate the efficiency and performance of various solvers, including MiniSat.

To run a benchmark, we can follow these steps:

- We will use the following link to download our tests:
`https://benchmark-database.de/?track=main_2024`.

- On the first page, we see several rows, each with a hash code on them. We need to select one of the hash codes to download the file in .cnf format.

- If we wish to search a specific family of tests, on the top of the page there is a 'Query for instances' search bar. Here, we can write a query to receive only the tests we are interested in. For example, if we wish to fetch only the tests from the 2024 competition of the Hamiltonian family, we need to run:

```
track=main_2024 and family like Hamiltonian
```

- After choosing which test we wish to download, we need to click on the hash code from the test and a download process will start. The downloaded file is a.zip file, which we need to unzip, and in it we can find the.cnf file that we need.

- Having downloaded the test file, we need to place it inside our Ubuntu folder first. To find it, in File Explorer, in the file search bar, we can type

```
\\wsl.localhost\Ubuntu-22.04\home\username
```

to easily access it.

- After placing the file in the correct folder, we can use the following command to have MiniSat try and solve it:

```
minisat <benchmark_file>.cnf output.stats > output.out
```

This command will create 2 files in the folder where we placed our test file, a .stats file where we can see the total runtime, whether the formula is satisfiable or unsatisfiable, and the memory used, and a .out file where the results are shown, showcasing as well which literals are true and which are false.

As seen, some challenges might arise from installing MiniSat, as such, we looked into installing MiniSat using different means, for example by using Cygwin, and by installing it using a Virtual Machine. However, we observed that installing it using WSL can prevent certain problems. Virtual machines do not use the system's full power, which is desirable when running benchmarks. Cygwin provides a harder and more complex installation process than PowerShell and Ubuntu.

2.4 Running MiniSat on a Virtual Machine

To analyze the Minisat code in more detail and to attempt to implement some modifications, we ran the MiniSat code on a virtual machine created in VMWare, whose range of functionalities made our work on the project easier. The OS installation was fast, and the state was preserved upon shutdown, allowing me to resume work from where I left off each time the virtual machine was relaunched. Running the MiniSat code was also a challenge for us, as we had to set up the environment to run this code on Linux by executing a series of commands. Additionally, we had to get accustomed to running the code on Linux and remember how to operate using the Linux terminal.

Commands for setting up the environment:

```
sudo apt update

sudo apt install git build-essential

sudo apt install zlib1g-dev

CXXFLAGS += -std=c++11 -fpermissive
```

Commands for running the code:

```
make clean

make
```

Commands for testing the functionality:

```
nano sample.cnf
```

```
nano output.out
```

```
./minisat sample.cnf output.out
```

It turned out that it is fairly hard to operate and make changes to this code, not only because of its dimensions (every key file has around one thousand lines of code, some even more) but also because of its age, MiniSat being developed more than 15 years ago.

3 Code Documentation

To better understand the MiniSat solver, we will delve into some of its most important algorithm, namely, the CDCL algorithms. We will also look in this section at possible ways in which we could improve the current code in the future.

The following diagram showcases the main use cases of MiniSats' code:

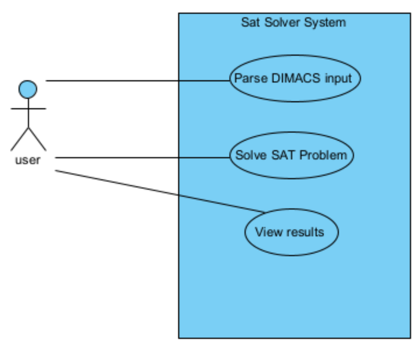

Fig. 1. Use Case Diagram

3.1 Algorithm structures

The CDCL algorithm is an advanced and sophisticated approach to solving Boolean satisfiability problems (SAT). It introduces many strong techniques to solve SAT problems in a faster way and handle more complex cases.

Conflicts in CDCL represent a main part of the algorithm and a key concept in modern SAT solving where the current variable assignments generate a logical contradiction. These contradictions occur when the solver's partial assignment violates the SAT formula's constraints, needing a plan to continue the search for a SAT solution.

What causes a conflict? A conflict appears during the DPLL search when a clause becomes unsatisfiable under the current variable assignments.

How are conflicts handled in CDCL? When a conflict is identified, the CDCL algorithm takes the next steps :

1. Conflict Analysis

The solver analyzes the conflict and traces the root cause of it, this involves :

- Identifying the sequence of propagation and decisions that lead up to the conflict

- Examining the implication graph which represents the variable assignment dependencies.

2. Clause Learning

The solver learns a new conflict clause which uses techniques like the first UIP rule to to prevent similar conflicts in the next iterations.

3. Backjumping

Instead of backtracking to the previous decision level, the solver strategically backjumps to an earlier decision level which optimizes the search efficiency by avoiding unnecessary backtracking.

4. Potential Restarts

Occasionally the solver restarts the search from the beginning, retaining the learned clauses to improve future exploration.

Why are conflicts so important in CDCL?

Conflicts aren't just obstacles, they are crucial problem solving opportunities that enable the CDCL solvers to :

- Reduce the search space through learned clause pruning

- Enhance solving efficiency

- Improve scalability to for complex SAT instances

With the core elements of the CDCL algorithm including conflicts explained, in the next parts we're going to break down how the CDCL algorithm is implemented in each file of the MiniSat code. The code is available on the project GitHub repository: `https://github.com/AndiSova/VF-Software-Engineering-2024-Project/tree/main/minisat`

The first file we're going to talk about is the main.cc file, located in the core folder.

The Solver class serves as the main implementation of the algorithm while the solverLimited() handling tasks like conflict analysis, decision making and propagation. The Lit data structure is used to represent Boolean variables and their negations while interrupt() allows the solver to be halted a common feature in CDCL solvers. In short here's what happens in the main.cc file: there are input files which are read, the solver is being set, some initial simplifications are being run which kick off the CDCL solving process.

Now we are moving to the Solver.cc in which we find the core components of the CDCL algorithm :

- Propagation: This is where the solver figures out which literals are implied by the current assignment, the propagate() function is the one that performs this function [3]. The following pseudocode shows how propagate()works [3]:

```
Constr Solver.propagate()
    while (propQ.size() > 0)
        // ‘p’ is now the enqueued fact to propagate
        lit p = propQ.dequeue();
        // ‘tmp’ will contain the watcher list for ’p’
        Vec(Constr) tmp;
        watches[index(p)].moveTo(tmp);

        for (int i = 0; i < tmp.size(); i++)
            if (!tmp[i].propagate(this, p))
                // Constraint is conflicting
                // copy remaining watches to ’watches[p]’
                // and return constraint:
                for (int j = i+1; j < tmp.size(); j++)
                    watches[index(p)].push(tmp[j]);
                propQ.clear();
                return tmp[i];
    return NULL;
```

- Conflict Analysis: When the solver hits a roadblock, the analyze() function helps to decide how to backtrack and learn from the conflict [3]. The following pseudocode shows how analyze() functions [3]:

```
void Solver.analyze(Constr confl, //
                Vec(lit) out_learnt, Int& out_btlevel)
    Vec(bool) seen(nVars(), FALSE)
    int counter = 0
    lit p = ~lit
    Vec(lit) p_reason
    // Leave room for the asserting literal
    out_learnt.push()
    out_btlevel = 0
    do
        p_reason.clear()
        // invariant here: confl != NULL
        confl.calcReason(this, p, p_reason)
        // Trace reason for p:
        for (int j = 0; j < p_reason.size(); j++)
            lit q = p_reason[j]
            if (!seen[var(q)]) {
                seen[var(q)] = TRUE
```

```
                if (level[var(q)] == decisionLevel())
                    counter++
                // exclude variables from decision level 0
                else if (level[var(q)] > 0)
                    out_learnt.push(~q)
                    out_btlevel = max(out_btlevel, level[var(q)])
        // Select the next literal to look at:
        do
            p = trail.last()
            confl = reason[var(p)]
            undoOne()
        while (!seen[var(p)])
        counter--
    while (counter > 0)
    out_learnt[0] = ~p
```

- Clause Database Management: The solver periodically cleans up its less active learned clauses to maintain efficiency with the help of the reduceDB() function [3]. The next pseudocode shows how reduced() helps clean less active clauses [3]:

```
void Solver.reduceDB()
    int i, j
    double lim = cla_inc / learnts.size()
    sortOnActivity(learnts)
    for (i = j = 0; i < learnts.size() / 2; i++)
        if (!learnts[i].locked(this))
            learnts[i].remove(this)
        else
            learnts[j++] = learnts[i]
    for (; i < learnts.size(); i++)
        if (!learnts[i].locked(this) && learnts[i].activity() < lim)
            learnts[i].remove(this)
        else
            learnts[j++] = learnts[i]
    learnts.shrink(i - j)
```

- Decision Heuristics: With the help of the pickBranchlist() function the next variables are chosen and their polarity is based on variable activity.

- Restarts: The search() function incorporates a strategy to reset the search periodically, which helps to solve tricky problems [3]. The pseudocode below shows the inner workings of the search() function [3]:

```
bool Solver.search(int nof_conflicts, //
            int nof_learnts, SearchParams params)
    int conflictC = 0
    var_decay = 1 / params.var_decay
    cla_decay = 1 / params.cla_decay
```

```
model.clear()

loop
    Constr confl = propagate()
    if (confl != NULL)
        // Conflict
        conflictC++
        Vec(lit) learnt_clause
        int backtrack_level
        if (decisionLevel() == root_level)
            return FALSE
        analyze(confl, learnt_clause, backtrack_level)
        cancelUntil(max(backtrack_level, root_level))
        record(learnt_clause)
        decayActivities()
    else
        // No conflict
        if (decisionLevel() == 0)
            // Simplify the set of problem clauses:
            // our simplifier cannot return false here
            simplifyDB()
        if (learnts.size() / nAssigns() >= nof_learnts)
            // Reduce the set of learned clauses:
            reduceDB()
        if (nAssigns() == nVars())
            // Model found:
            model.growTo(nVars())
            for (int i = 0; i < nVars(); i++)
                model[i] = (value(i) == TRUE)
            cancelUntil(root_level)
            return TRUE
        else if (conflictC >= nof_conflicts)
            // Reached bound on the number of conflicts:
            // force a restart
            cancelUntil(root_level)
            return FALSE
        else
            // New variable decision:
            // may have a heuristic for polarity here
            lit p = lit(order.select())
            // cannot return false
            assert(decisionLevel() > 0)
```

- Garbage Collection: The garbageCollect() function keeps memory usage in check by reclaiming unused memory, by relocating clauses and and resources.

The next file on our list is the Dimacs.h file is all about reading input. It handles parsing the standard DIMACS file format that SAT solvers use. The key functions include :

- readClause: This function extracts individual clauses from DIMACS file and stores them in the solver's internal database.

- parse_DIMACS_main(): This function represents the core parsing loop that processes the DIMACS file, reading its header and clauses and validating their addition to the solver.

- parse_DIMACS(): This represents a high-level wrapper function around the main parser and prepares the input stream.

The last but not least file we're going to have a look at is Solver.h, this file contains the core methods of the CDCL algorithm and supporting components that tie everything together.

Methods : - The Main Solving Methods: These methods in question are the solve() and solveLimited() functions which are the main entry points of for solving SAT problems using the CDCL algorithm. - The analyze() Method: This method performs conflict analysis and clause learning which are key features of CDCL. - The propagate Method: Handles unit propagation and detects conflicts during the solving process. - Decision Levels and Search State Tracking: Utilizes decision levels, reasons for variable assignments, and a trail to manage and monitor the current state of the search.

Supporting Components :

- Watcher Lists and Watcher Data Structures: Efficiently monitors watched literals to optimize propagation.

- order_heap Priority Queue: Maintains an ordering of decision variables based on variable activity, improving the efficiency of the decision heuristic.

- Clause Management: Includes operations for attaching, detaching, and removing clauses as needed.

- Variable Activity Maintenance: Tracks and adjusts variable activity with a decay mechanism, ensuring the decision heuristic remains effective

The combination of these methods and components in Solver.h and all the other files analyzed provide the foundation for robust SAT solving using the CDCL algorithm.

The following diagram showcases the relations between the classes that we talked about :

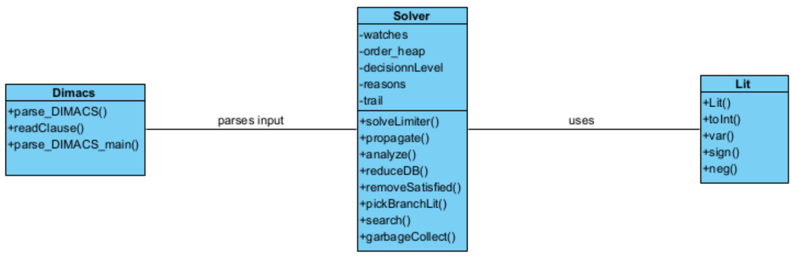

Fig. 2. Class Diagram

3.2 Proposed Optimisation

After multiple attempts to improve our code, we finally managed to optimize it by adding some if statements to the search function in the Solver.cc file (the file that does in itself the CDCL part). We improved the way MiniSat works, CDCL making MiniSat change the variable var_decay throughout the time the .cnf file is running, making some drastic changes to the logic of the code.

We will start with some print statements that will help us understand the way our code works:

If we have no conflicts we will get this print that will show that our clause didn't cause any conflicts: "Propagating -1"

If we have a benchmark that has conflicts, the code will start dynamically to adjust the var_decay variable.

How Dynamic Adjustment Works (Impact of var_decay on Search):

The optimization dynamically adjusts var_decay based on the number of conflicts the solver has encountered:

Early Phase ($<$ 1000 conflicts):

var_decay = 0.95 (The smallest var_decay)

The solver needs to find out about the structure of the problem and to explore it. Older variables are decayed faster, allowing the solver to test different parts of the search area. Useful in the early phase of the search, where the solver builds a global understanding of the problem.

Middle Phase (1000–10,000 conflicts):

var_decay = 0.99

Strike a balance between exploring new variables and exploiting variables from recent conflicts.

Late Phase ($>$ 10,000 conflicts):

var_decay = 0.999 (the largest var_decay)

Focus on exploitation. Retain activity information for recently conflicting variables for longer.

Useful in the late phase of the search, where the solver is focusing on resolving specific conflicts.

Here is the pseudocode for our proposed optimisation:

```
FUNCTION adjustVarDecay:
    IF conflicts < 1000:
        // Early phase: prioritize exploration
        SET var_decay = 0.95
    ELSE IF conflicts < 10000:
        // Middle phase: balance exploration and exploitation
        SET var_decay = 0.99
    ELSE:
        // Late phase: focus on exploitation
        SET var_decay = 0.999

    PRINT "Adjusted var_decay to:", var_decay, //
            "after", conflicts, "conflicts"
END FUNCTION
```

4 Experimental Results

4.1 Hamiltonian family of problems

We will briefly talk about what Hamiltonian Graphs are and their complexity.

In a graph G, a Hamiltonian circuit is a circuit that contains every vertex of G, and a Hamiltonian path is a path that contains every vertex of G. A graph containing a Hamiltonian circuit is called a Hamiltonian graph, and a graph containing a Hamiltonian path is called a traceable path(see [1]). A Hamiltonian cycle is a spanning cycle in a graph(a cycle through every vertex). We can also say that a graph G is Hamiltonian if G has a Hamiltonian cycle(see [5]).

As discussed in the works of Richard M. Karp, Hamiltonian graphs and cycles are Nondeterministic Polynomial (NP)-complete(see cite[4]). An NP-complete is a problem for which a solution can be verified in polynomial time. However, there remains no polynomial-time algorithm that can be used to solve an instance of this problem in a more efficient manner than NP-hard problems. An example of the nature of this problem is the Hamiltonian Cycle Problem. It is easy to confirm for a given cycle and is overall computationally difficult to obtain such an arbitrary graphs(see [2]). The NP-completeness of Hamiltonian problems suggests that considerable time and effort must be devoted to solving large instances of them, particularly when the graph size increases. Such problems exist in several disciplines such as logistics, network design, and bioinformatics, all of which require Hamiltonian problems to be solved – an essential area of study of practical relevance (see [4]).

MiniSat and similar SAT solvers can be useful in studying the complexity of graph problems like the Traveling Salesman Problem due to their capability to solve SAT instances. These benchmarks are insightful about the difficulty of the problem and the performance of the solver. In particular, we consider the hard small random Hamiltonian Cycle (HC) benchmarks that we tackled, which are graphs created by joining many copies of the xor9 graphs. A xor9 is a subgraph of P9, a nine-vertex simple path graph, except that edges are added to vertices(1,6) and (4,9) to enhance the graph's complexity and connectivity. These extra edges transform the structure into a dense one with a higher edge count plus 4 "outer vertices" (1,3,7,9), making it easy to build challenging problems close to the phase transition.

The balanced xor9-based Hamiltonian Cycle Problems HCPs are also worth emphasizing because of their peculiar constraints. The outer degrees of all the outer vertices are always 2 or 3, and each graph consists of k-linked xor9 subgraphs. This stringent restriction leads to hard instances that are few or uniquely Hamiltonian and are hard to solve. Such benchmarks are good for proving the performance of the SAT solvers like MiniSat and Kissat. In doing so, we have analyzed the benchmarks freshly defined and their issues are mainly with the increasing size of the graph and its complexity that leads to practical issues with SAT solvers dealing with Hamiltonian problems.

The following experimental results for the Hamiltonian family of tests were taken from the following link: `https://benchmark-database.de/?track=main_2024`. For the test, the following computer with the mentioned specs was used:

OS: Windows 11 Pro,

Processor: 12th Gen Intel(R) Core(TM) i5-12500H 2.50 GHz,

RAM: 16,0 GB (15,8 GB usable).

hash	SAT/UNSAT	Memory used (MB)	Runtime (s)
3a75ad246dbc750a7391ad887c5b0835	SAT	22.00 MB	76.3236 s
3c6e1d1c4b8d3d08aa4c1df3805f4f7d	UNSAT	51.00 MB	2206.42 s
5e1c11b77cdf3717b81b957120f0f477	SAT	38.00 MB	746.807 s
7fb202a51c0223f3119887a57086ca4d	SAT	22.00 MB	74.9458 s
8b18bb75459a4161633ba2a3c8ee183e	SAT	39.00 MB	771.64 s
09b61bbf19748094a7d896aac314ab36	SAT	40.00 MB	555.029 s
09c1b79b1cfe3522364fe60aef780703	SAT	32.00 MB	287.657 s
9d9c4fa425282759eb9e98b82fb5f56e	SAT	26.00 MB	148.588 s
13ae2628d8e113db1786dba41a65fe38	SAT	28.00 MB	428.061 s
14e4cfcf0d83b2185fad41684d00d4dc	SAT	43.00 MB	722.132 s
19e2c3a0865c8c1b4543d11213bebe5f	UNSAT	18.00 MB	114.904 s
54c2da6d387a6f5ad6e014ae4d4decfc	SAT	23.00 MB	185.681 s
57f4ea7ab160d996e38e69fac59869c4	UNSAT	18.00 MB	119.934 s
697c96ac45534726c7dbd96faa11a86a	SAT	21.00 MB	157.385 s
915a25bd189357e4c6d7771b69a6849f	UNSAT	26.00 MB	367.978 s
1507d9812624b3e0eaf15e40100be020	SAT	25.00 MB	142.535 s
5865fb9a6575d2ae6542c36ab96646a9	SAT	32.00 MB	636.069 s
0265448c232e3a25aa5bcd29b1b14567	UNSAT	43.00 MB	1693.69 s
195852083a05edee1902233698eec14a	SAT	32.00 MB	418.786 s
a45a0358685867bd4f1c7f7c0b0e379c	SAT	25.00 MB	274.111 s
aa9b67fd19d54ad51b93ee4ba5dc75fc	UNSAT	47.00 MB	1825.8 s
ad4e151d80c7012d88dd79bcfceaade5	UNSAT	15.00 MB	51.5779 s
af05a6b68a1cff165b684d9ff0ae3b3b	SAT	43.00 MB	967.203 s
b44ea915362c3a140269003d45b1d053	SAT	42.00 MB	1318.77 s
b7273af3d468ea2595f11a6dbd6ef6ce	UNSAT	40.00 MB	1139.49 s
bbfed8974655bca520259deb10d2347b	SAT	19.00 MB	114.361 s
c0e6e6eeebd48ca600cfc7d662fa804c	UNSAT	39.00 MB	1530.48 s
c3e53c353f30d9e2eb54ed93d6ce4f02	UNSAT	20.00 MB	149.637 s
c5a98231dd54cbca06135293bb7e1985	SAT	26.00 MB	151.881 s
c6568fc8805127e876c4c23551bf49fa	UNSAT	35.00 MB	1189.2 s
d195412a62cdcbb851136f60af76f463	UNSAT	25.00 MB	290.691 s
ded23680dfeab2879c05bc0e4de21126	UNSAT	21.00 MB	269.528 s
df1bd67978b9b0ec1d326ba174bc273c	SAT	36.00 MB	403.3 s
e08f11f0a3bd266ee5c78ce332de107f	UNSAT	39.00 MB	1232.28 s
edceb8782e72e290fa54757dbfdd0173	UNSAT	23.00 MB	283.357 s
eddd68e14d69cce7190b99f4e7abdafb	SAT	17.00 MB	52.973 s
f45e5faf1bcccbdd3065dd6367c3bd16	UNSAT	37.00 MB	1226.51 s
f296fe701a562022c0de0cf565fbca7d	UNSAT	17.00 MB	77.2704 s
f376d4c191518ed704326960b6b19a4b	UNSAT	42.00 MB	1608.78 s

Table 1. MiniSat benchmark results from the Hamiltonian family of tests

Total approximate runtime: 8 hours and 30 minutes.

Benchmark sum: 24(hours)/40(number of tests) = 0,6.

5 Conclusion

Logic and satisfiability have been a core part of computer science, serving as an intermediary between logical reasoning and computational practicality. We discussed in detail the internal workings of MiniSat, a very influential SAT solver, looking at its installation, usage, algorithms, and possible improvements. Despite its compact and readable design, MiniSat presents some notable challenges for beginners, ranging from installation hurdles to understanding its implementation of the CDCL algorithm and benchmarking processes.

Through our detailed analysis, we have pointed out the necessary steps to overcome such challenges, thus providing students and researchers alike an easy way to work with the SAT solvers. We have shown that MiniSat is still relevant and capable of running benchmarks and exploring Hamiltonian tests while proposing optimizations to its code base for solving complex logical problems. Its simplicity and efficiency maintain MiniSat as a very useful educational tool and a benchmark in order to advance the SAT-solving technologies.

As computational problems grow in complexity, the role of tools like MiniSat becomes ever more critical. Its ability to handle millions of variables efficiently underlines the potential of SAT solvers to solve real-world challenges in areas such as artificial intelligence, electronic design automation, and combinatorial optimization. Going forward, improvements in algorithms and usability for MiniSat could further enhance its utility, ensuring it remains a cornerstone in the exploration and application of satisfiability problems.

Acknowledgments.

References

1. Bermond, J.C.: Hamiltonian graphs. Selected topics in graph theory pp. 127–167 (1979)
2. Cook, S.A.: The complexity of theorem-proving procedures. In: Logic, automata, and computational complexity: The works of Stephen A. Cook, pp. 143–152 (2023)
3. Eén, N., Sörensson, N.: An extensible sat-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
4. Karp, R.M.: Reducibility among combinatorial problems. In: et al., M.J. (ed.) 50 Years of Integer Programming 1958–2008, pp. 219–241. Springer (2010)
5. Kawarabayashi, K.i.: A survey on hamiltonian cycles. Interdisciplinary information sciences **7**(1), 25–39 (2001)

Tasks distribution:

Stoentel Alexandru-Eduard: Procesul de instalare Minisat cu ajutorul WSL, rulare familia Hamiltoniana, explicarea aplicarii in scris a CDCL