MiniSat Report

Cosmin Pascaru, Rafael Bota, Andrei Bota, and Radu Todosan

West University of Timișoara

Abstract. This report is about *MiniSat*, a tool for solving SAT problems. It explains how to use MiniSat and provides an overview of its code structure, detailing how the main classes are organized and how they work together. The report also includes performance results from the 2024 SAT Competition to show how well MiniSat performs compared to other solvers.

Keywords: Minisat · SAT Solver · Benchmark

1 Introduction

SAT problems are an important part of computer science and are used in various fields such as in software verification, optimization, and artificial intelligence. A SAT problem asks whether it's possible to assign true or false values to variables in a logical formula so that the whole formula becomes true. Despite its simple appearance, SAT is the foundation for solving many complex problems in computing.

This report is about MiniSat, a popular SAT solver known for its simplicity. It explains how to use MiniSat to solve SAT problems and looks at its internal structure. We also included benchmarks from the 2024 SAT Competition.

The report starts with an overview of SAT and why it matters, then explains how to install and use MiniSat. It also describes important parts of MiniSat, like the Solver class and how is the CDCL algorithm implemented. Finally, we present benchmark results to show how well MiniSat works on different computers.

2 Contributions

- Source Code Analysis (Todosan Radu, Rafael Bota, Andrei Bota): Conducted an examination of MiniSat's source code, identifying key components and their roles within the SAT solving process.
- Class Analysis and Diagram Creation (Todosan Radu, Rafael Bota): Worked together to analyze the structure of MiniSat's classes and created a class diagram.
- Flow Understanding (Everybody): Mapped out the execution flow of MiniSat to better understand its core components, such as clause propagation, conflict analysis, and learning mechanisms.

- Algorithm Understanding (Todosan Radu, Andrei Bota): Tried to understand where and how the CDCL algorithm operates within MiniSat.
- Report Documentation (Cosmin Pascaru): Formatted the findings into the report.
- Benchmark Testing (Cosmin Pascaru, Andrei Bota): Ran performance benchmarks using MiniSat.

The complete project, including the benchmark results and the source files for the report, are available on GitHub.

3 The Satisfiability Problem

The SAT problem asks whether there is a way to assign true or false values to a set of variables in a logical formula so that the entire formula becomes true. This problem is important because many real-world tasks can be simplified into SAT problems.

3.1 Why is SAT Important?

SAT is important because many complex problems in computer science, from designing circuits to solving problems in artificial intelligence, can be reduced to SAT problems. Being able to solve SAT efficiently helps solve these larger, more complicated problems.

3.2 A Simple Example

Here's a simple example of a SAT problem:

$$(A \vee B) \wedge (\neg A \vee C)$$

The question is: Can we find a set of true or false values for A, B, and C that make the entire formula true? If such values exist, the problem is called *satisfiable*; if no such values exist, the problem is *unsatisfiable*.

SAT solvers like MiniSat use smart techniques to figure out whether a formula can be satisfied, and if it can, to find the values that make it true.

4 Configuration & Mainflow

MiniSat is a popular tool used to solve SAT problems. It is known for its small size and simple design. Despite its minimalistic approach, MiniSat is effective at solving SAT problems.

MiniSat is open-source and because of this it became a starting point for other SAT solvers and has influenced the design of similar tools.

This section will explain how to install and use MiniSat, and will provide an overview of its main features. We will also take a closer look at the main components that make MiniSat work.

4.1 Installation

To install MiniSat, users can download and build it from its source repository. The following steps outline a typical installation process:

1. Clone the MiniSat repository from GitHub or another source:
   ```
   git clone https://github.com/niklasso/minisat.git
   ```
2. Navigate to the cloned directory:
   ```
   cd minisat
   ```
3. Build MiniSat using `make`:
   ```
   make install
   ```

There is also the possibility of installing it like this if running on linux OS or `WSL` (Windows Subsystem for Linux):

```
sudo apt install minisat
```

After these steps, MiniSat will be ready for use.

4.2 Usage

MiniSat is used from the command line. It takes a CNF (Conjunctive Normal Form) file as input and outputs whether the formula is satisfiable or unsatisfiable. If the formula is satisfiable, MiniSat will also include the model in the output. The general usage format is:

```
./minisat input.cnf output.txt
```

– **input.cnf**: The input file containing the SAT problem in CNF format.
– **output.txt**: (Optional) A file where the solution will be written.

4.3 Important Arguments

MiniSat accepts various command-line arguments that modify its behavior. Here are some of the most commonly used ones:

- `-verb=<level>`: Sets the verbosity level of the output. Higher levels provide more detailed logs.
- `-cpu-lim=<seconds>`: Specifies a CPU time limit for solving the problem..
- `-mem-lim=<megabytes>`: Sets a memory usage limit.

4.4 Running More Tests Using an Automated Script

To make it easier to test multiple CNF files, we created a Bash script to automate the process of running MiniSat. This script was used to run the benchmark, helping us manage the time limit and keep logs of each test. Below is an explanation of how the script works and its role in our testing. The script can be found in our Git repository at: script.sh.

How the Script Works: The script goes through all the CNF files in the folder and runs MiniSat on each file with a set time limit. Here is how the script works step by step:

1. The time limit is set (e.g., 5 hours) and converted to seconds.
2. For each CNF file, the script:
   - Creates output and log file names based on the name of the CNF file.
   - Runs MiniSat with the time limit and saves the output and any messages to the log file.
   - Checks the exit code from MiniSat to see if the time limit was reached or if there were any errors.
   - Adds a note in the log if the time limit was exceeded or if an error occurred.
3. The script saves the results and logs for each file.

This script made it easy to run the benchmark, ensuring that each run had an output file and a log file for reviewing later.

4.5 Overall Architecture

The architecture of MiniSat is built around two main parts: the `Main.cc` file and the `Solver.cc` file. The `Main` function handles the setup for the entire process. It is responsible for reading the input, setting up options, and managing how the program runs. Once everything is prepared, the `Solver` class takes over and actually works on solving the SAT problem. It uses CDCL to find a solution or determine that there is no solution.

4.6 Main

The `Main.cc` file in MiniSat serves as the primary entry point. It is responsible for managing the overall flow of the program, from reading the input file to outputting the results. Below is a detailed breakdown of its main components and functionality.

4.7 File Overview

The `Main.cc` file manages the entire SAT solving process, including:

- Parsing command-line arguments to configure options.
- Reading SAT problems in the DIMACS format.
- Initializing and configuring the solver, including setting verbosity levels and resource limits.
- Running the solving process, handling interrupts, and outputting the solution.

4.8 Key Components

Command-Line Options The program allows users to configure various parameters via command-line options:

- **Verbosity**: Controls the level of detail printed during execution (silent, some output, or more detailed output).
- **CPU Limit**: Sets a limit on the CPU time the solver is allowed to use.
- **Memory Limit**: Specifies the maximum amount of memory the solver can consume.
- **Strict Parsing**: Ensures the correct format of the input DIMACS file.

Reading the Input: The program reads the input SAT problem. The input is parsed and transformed into a suitable format for solving.

Solving the Problem: After parsing the input, the solver attempts to simplify the problem using basic logical operations. It then attempts to solve the SAT problem by performing a search for a satisfying assignment, if one exists. The solving process is interrupted if necessary, allowing the solver to exit early when resource limits are exceeded or if a signal is received.

Output and Results: The results are printed on the console or written to an output file, depending on the user's specifications. If the solver finds a satisfying assignment, the solution is output as a list of variable assignments. If the problem is unsatisfiable, a corresponding message is printed. The program also provides detailed statistics about the solver's performance when verbosity is enabled.

4.9 Integration of the Solver in Main.cc

The `Main.cc` file demonstrates how the `Solver` class is instantiated and used. It sets up the solver, parses the input in DIMACS format through `parse_DIMACS()`, and manages resource constraints (e.g., CPU time and memory) using options like `cpu_lim` and `mem_lim`.

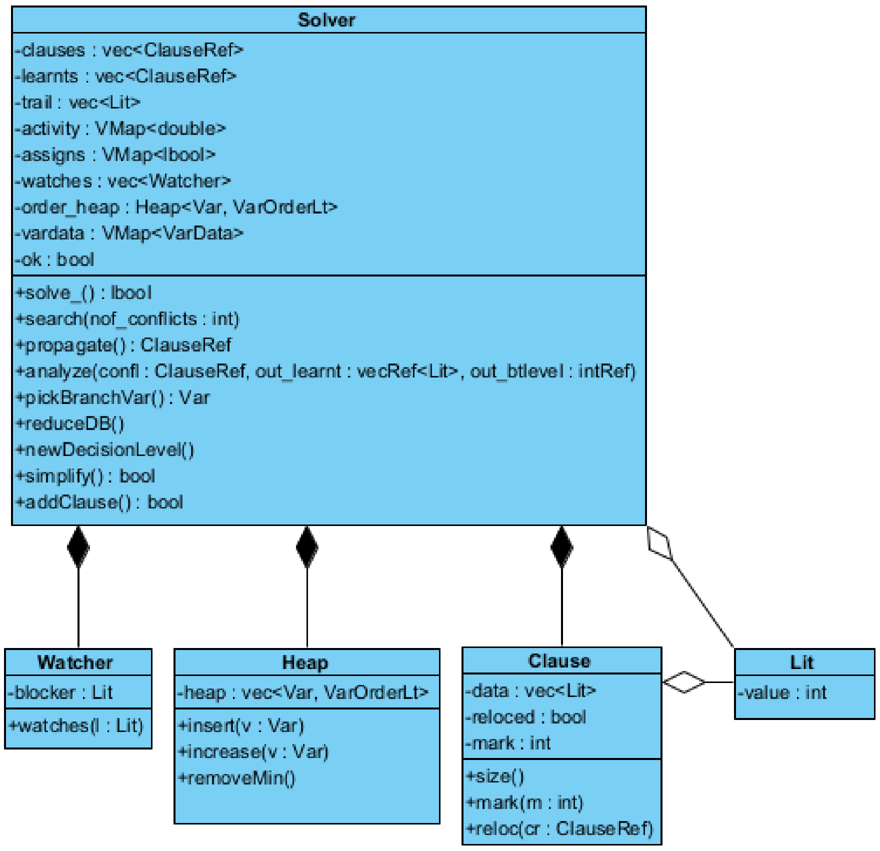
5 Overall Classes Structure

This section explains the classes structure based on the provided class diagram. The classes are grouped into three categories: the main class (**Solver**), secondary classes (**Clause**, **Lit**, **Heap**, and **Watcher**), and auxiliary classes (**VMap**, **lbool**, **Var**, **VarData**, and **VarOrderLt**), which are not shown in the diagram.

The **Solver** class is the most complex, containing the majority of attributes and methods, while the other classes act as helpers to support its functionality. Most relationships between classes are compositional, as many attributes, such as *clauses* (for **Clause**), *order_ heap* (for **Heap**), or *watches* (for **Watcher**), are created when a **Solver** object is initialized.

For **Lit**, the relationship is aggregation with both **Solver** and **Clause**, as **Lit** objects are initialized independently and do not fully depend on the other classes. This structure highlights how the classes interact and contribute to the overall functionality of the project.

Fig. 1. Class Diagram



5.1 The Solver Class

The `Solver` class in MiniSat is the core structure responsible for implementing the SAT-solving process. It supports the CDCL algorithm, which incorporates various methods for propagation, conflict resolution, decision-making, and backtracking. In the following, we detail its most important methods, including those essential to CDCL.

Important Attributes in the Solver Class

- `clauses`: Stores the original problem clauses. It is used during initialization and conflict analysis, working closely with the watches to ensure efficient propagation.
- `learnts`: Holds the learnt clauses generated through conflict-driven learning, helping to avoid revisiting similar conflicts. These clauses are managed during conflict analysis and pruning (reduceDB()).
- `trail`: A vector that records the assignments made during the search, enabling backtracking. It works with trail_lim to manage decision levels.
- `activity`: Represents a heuristic metric to measure the importance or "activity" of variables. It influences the decision heuristic by prioritizing variables with higher activity, often linked with order_heap and updated during conflict analysis.
- `assigns`: Tracks the current assignment of each variable, indicating whether a variable is true, false, or unassigned. It is essential for propagation, conflict detection, and decision-making.
- `watches`: A list that keeps track of the clauses that are "watching" each literal. This supports efficient Boolean constraint propagation (BCP) and interacts with the trail during unit propagation to identify satisfied clauses.
- `order_heap`: A priority queue that stores variables sorted by their activity, driving the decision heuristic for selecting the next variable to branch on. It is updated based on activity levels and influenced by conflict-driven learning.
- `vardata`: Stores data about each variable, including reasons for assignments and their decision levels. This is crucial for conflict analysis and backtracking, working closely with trail, trail_lim, and clauses.
- `ok`: Indicates whether the problem is satisfiable or trivially unsatisfiable. This attribute helps to avoid unnecessary computation by halting the solver when the problem is determined to be unsolvable.
- `conflict_budget / propagation_budget`: Limits the number of conflicts or propagations that can occur during search, allowing controlled exploration of the solution space.

Important Methods in the Solver Class

- `solve()`: The entry point method that initiates the solving process. It prepares the solver for execution by resetting the relevant parameters and then calls `search()` to find a solution.
- `search()`: The primary loop method that drives the CDCL algorithm. It repeatedly:
  - Invokes `propagate()` to perform unit propagation and detect conflicts.
  - Calls `analyze()` to generate learned clauses when conflicts are found.
  - Utilizes `cancelUntil()` for non-chronological backtracking to a previous decision level informed by the learned clause.
  - Selects the next variable for assignment using `pickBranchLit()` if no conflict is present.
  - Manages termination conditions: returns `l_True` if a solution is found, `l_False` if unsatisfiability is proven, or exits if resource limits are reached.
- `propagate()`: Performs Unit Propagation and Boolean Constraint Propagation (BCP) by propagating implications of the current assignments, identifying conflicts or satisfied clauses. It works closely with trail, watches, and assigns and returns the conflicting clause when a conflict is found.
- `analyze(CRef confl, vec<Lit>Ref out_learnt, intRef out_btlevel)`: Used for conflict analysis, this method identifies the cause of a conflict and produces a learned clause that prevents the same conflict from happening again. It utilizes trail, trail_lim, vardata, and activity and produces clauses stored in learnts.
- `pickBranchLit()`: Selects the next decision variable and its polarity based on a decision heuristic that prioritizes variables with high activity. It relies on activity and order_heap, ensuring that the solver explores promising areas of the search space.
- `reduceDB()`: Periodically called to reduce the number of learned clauses. It helps maintain solver performance by retaining only high-activity clauses that contribute to conflict resolution.
- `newDecisionLevel()`: Starts a new decision level by separating decision levels in the trail using trail_lim. It prepares the solver for making a new assignment, supporting decision-making and backtracking.
- `addClause(vec<Lit>Ref ps)`: Adds a new clause to the solver, updating the solver's state with additional constraints. The clause is appended to the clauses, and watches are adjusted to account for the new constraints.
- `simplify()`: Simplifies the problem by removing satisfied clauses and literals, optimizing the solver's state for better efficiency. It interacts with clauses, learnts, and assigns, and can mark the problem as unsatisfiable if inconsistencies are found.
- `setConfBudget(int64_t x) / setPropBudget(int64_t x)`: Sets resource constraints for the solver by limiting the number of conflicts or propagations allowed during the search. These methods are useful for iterative or resource-constrained solving, linking with conflict_budget and propagation_budget.

Summary of Importance :

Decision-making in MiniSat is guided by `activity`, `order_heap`, and `decision`, while *propagation* relies on `trail`, `watches`, and `assigns`. *Conflict analysis* is supported by `vardata`, `trail`, and `learnts`, forming the core of its SAT-solving process.

The *critical methods* in MiniSat include `solve()`, `propagate()`, `search()`, and `analyze()`, while *efficiency boosters* are `simplify()`, `reduceDB()`, and `backtrack()`. *Utility functions* such as `addClause()`, `newDecisionLevel()`, and `setConfBudget()` further enhance its functionality.

5.2 Secondary Classes

Clause Class – Clause Representation The Clause class represents a disjunction of literals and is central to the SAT problem. A clause is true if at least one literal is true. It supports propagation, conflict analysis, and clause learning, guiding the solver's search process.

Attributes:

– `data : vec<Lit>`: A vector storing the literals of the clause.
– `reloced : bool`: Indicates whether this clause has been relocated (used for memory management in *ClauseAllocator*).
– `mark : int`: A marker for special operations (e.g., reduceDB or simplification).

Methods:

– `size()`: Returns the number of literals in the clause.
– `reloc(cr : ClauseRef)`: Relocates the clause in memory and updates its reference.
– `mark(m : int)`: Sets the marker of the clause.
– `operator[] (i : int)`: Overload facilitates iteration and manipulation of literals.

Lit Classes - Literal Representation The Lit class represents a literal (variable or its negation) and is the basic building block for clauses and propagation. It simplifies logical operations and variable management.

Attribute:

– `value : int`: Encodes the literal as an integer (positive/negative variable).

Heap Class – Priority Management for Variables Role: Implements a heap to prioritize variables based on activity. Used by VSIDS heuristic in `pickBranchVar()` to select variables that are likely to lead to a solution.

Attribute:

– `heap : vec<Var>`: Stores variables in the heap, ordered by activity.

Methods:

- `insert(v : Var)`: Inserts a variable into the heap.
- `increase(v : Var)`: Increases the activity score of a variable.
- `removeMin()`: Removes the variable with the lowest priority.

Watcher The Watcher class monitors literals to efficiently trigger propagation when they become true. It optimizes the solver by focusing on relevant clauses during unit propagation, enhancing performance.

Attribute & Method:

- `blocker : Lit`: A literal used to optimize propagation. If this literal is already true, propagation is stopped immediately.
- `watches(Lit)`: Lists the literals being watched.

5.3 Auxiliary Classes

VMap - Template Class: Extends IntMap<Var, T>

Key Points:

- `Purpose`: To associate data with variables (e.g., activity scores, assignments, polarities).
- `Usage`: Provides efficient access and storage for per-variable data structures.
- `Template Parameter (T)`: Specifies the type of data stored for each variable.

Context Examples:

- `activity: VMap<double>`: Maps variables to their activity scores.
- `assigns: VMap<lbool>`: Maps variables to their current truth assignments.

lbool - Class Extends boolean logic *true/false* with a third state, *undefined.*

Attribute `value : uint8_t`: A variable of type uint8_t (an 8-bit unsigned integer) that stores the logical state. It can have the following values:

- `0: true (l_True)`
- `1: false (l_False)`
- `2: undefined (l_Undef)`

Utility: `lbool` allows the solver to keep track of unassigned variables, which is essential for the decision-making process `pickBranchVar()` and propagation `propagate()`. With lbool, the solver can quickly identify which variables need to be evaluated to satisfy all clauses.

Var - typedef

Purpose: Each SAT variable is identified by a unique integer.

VarData - struct

Purpose: Tracks why and when a variable was assigned to help in backtracking and conflict analysis.

VarOrderLt - struct

Purpose: Defines the sorting criteria for order_heap, prioritizing variables with higher activity for decision-making.

Usage: Works with Heap<Var, VarOrderLt> to guide the solver's branching decisions.

6 SAT Solving Process

This section provides an in-depth look at the SAT solving process in MiniSat, focusing on its core algorithm, Conflict-Driven Clause Learning (CDCL), and the methods that enhance its efficiency. CDCL integrates key components such as decision making, propagation, conflict analysis, and backtracking to navigate the search space. Supporting mechanisms such as activity-based heuristics, clause database management, and restart policies further optimize the solving process. We also discuss essential techniques such as Boolean Constraint Propagation (BCP) and variable ordering, which ensure logical consistency and effective exploration of potential solutions. Together, these strategies enable MiniSat to solve complex SAT problems efficiently.

6.1 CDCL Algorithm

The CDCL algorithm in MiniSat is implemented through various methods, these being the most important:

- **Decision Making:** Handled by `pickBranchLit()`.
- **Unit Propagation:** MiniSat uses the propagate() method with a **watcher-based propagation mechanism** to also handle **Boolean Constraint Propagation** across all clauses.
- **Conflict Analysis and Clause Learning:** MiniSat's analyze() method performs conflict analysis and generates learnt clauses. These clauses are stored in the learnts structure and used to avoid revisiting the same conflicts.
- **Non-Chronological Backtracking:** MiniSat backtracks to the appropriate decision level through the cancelUntil() method, using learnt clauses and skipping irrelevant levels.
- **Clause Database Management:** MiniSat periodically simplifies the learnt clause database using the reduceDB() method to ensure efficiency.
- **Restart Policies:** MiniSat implements restarts to explore different parts of the search space effectively, which occurs solve_() method.

These components are combined in the search() method.

6.2 Propagation & Heuristics

Unit Propagation and Boolean Constraint Propagation (BCP) in MiniSat

Unit Propagation and Boolean Constraint Propagation (BCP) are essential techniques in SAT solvers like MiniSat, enabling the simplification of the formula by deducing variable assignments based on logical constraints.

- *Unit Clause*: A unit clause is a clause that contains only one unassigned literal. When a unit clause is found, the only literal in it must be assigned a value that satisfies the clause.
- **Unit Propagation** assigns values to variables based on unit clauses forcing their satisfaction and potentially triggering a chain reaction of further assignments.
- **Boolean Constraint Propagation (BCP)** is a broader process that propagates all logical consequences of current assignments, including unit propagation, simplifying the formula, detecting conflicts, and uncovering new unit clauses.

In MiniSat, BCP, including unit propagation, is implemented in the `propagate()` method. This method iterates over the clauses and checks for unit clauses, while also managing broader logical implications of current assignments. MiniSat achieves efficient propagation using a **watching mechanism** (through the `watches` attribute), which tracks which clauses are watching a particular literal. When a literal is assigned, the solver propagates its implications across all relevant clauses, performing both unit propagation and more general constraint propagation, potentially triggering further assignments or identifying conflicts.

Conflict Detection and Backtracking in MiniSat

Conflict detection occurs during unit propagation. If the solver encounters a situation where a clause is unsatisfied (i.e., a conflict), it backtracks to explore other possible assignments.

- *Conflict Example*: A conflict occurs when a clause cannot be satisfied regardless of how the literals in it are assigned. If, during unit propagation, a clause like *(x1 OR x2)* is assigned *both x1 = false and x2 = false*, a conflict is detected.

In MiniSat, conflict detection happens in the `propagate()` method. If unit propagation leads to an unsatisfiable clause, the solver then calls the `analyze()` method to perform **conflict analysis**, generating **a learnt clause** that helps avoid the same conflict in the future. The solver then **backtracks** by using a code section from `analyze()` body, which reverts the solver's state to a previous decision level, allowing the solver to try different assignments. The conflict-driven learning process improves efficiency by pruning invalid search paths.

Variable Ordering and Activity-Based Heuristics in MiniSat
The order in which variables are assigned significantly impacts the solver's performance. MiniSat dynamically determines this order using activity-based heuristics, specifically the VSIDS (Variable State Independent Decaying Sum) heuristic.

- **Variable Activity**: Each variable's activity score increases when it participates in a conflict. The more frequently a variable is involved in conflicts, the higher its score becomes, making it more likely to be selected for assignment. The activity scores are decayed over time to ensure the solver doesn't focus too much on older conflicts.
- **Priority Queue**: MiniSat maintains a priority queue (`order_heap`) that ranks variables based on their activity scores. This allows the solver to prioritize exploring the search space that is more likely to lead to a solution.

By using this approach, MiniSat efficiently balances exploration and exploitation of the search space, avoiding irrelevant paths and focusing on promising areas.

7 Benchmark Results

To evaluate MiniSat performance, we conducted 15 test runs using benchmarks from the software verification category of the 2024 SAT Competition. These tests were executed on two different computers. This comparison helps understand how MiniSat's performance varies across different system configurations.

7.1 System Specifications

The benchmarks were run on the following systems:

- **Computer 1**: Intel Core i7-7700k, 32GB 3200MHz, Win 10
- **Computer 2**: AMD Ryzen 7 5800X3D, 64GB 3200MHz, Win 11

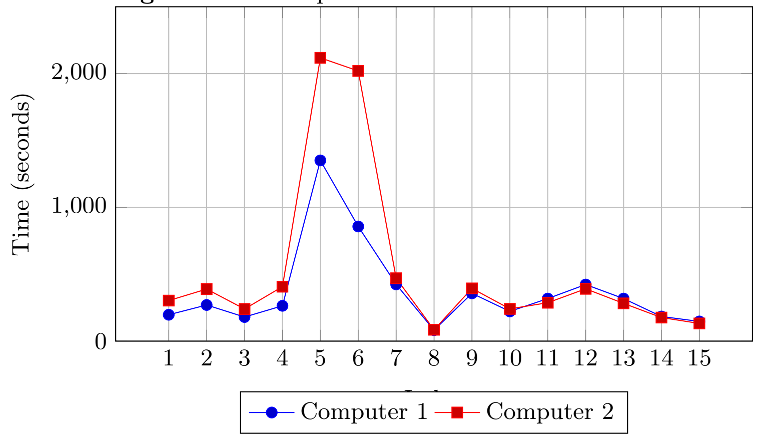
7.2 Results Overview

Table 1 summarizes the results of the benchmark tests, showing the CPU time and memory usage for each execution on both computers.

Table 1. Benchmark Results for 15 Executions from the Software Verification Family

Execution	CPU Time (Computer 1)	Memory Used (Computer 1)	CPU Time (Computer 2)	Memory Used (Computer 2)
1	197 s	5383 MB	303 s	5384 MB
2	270 s	6091 MB	389 s	6090 MB
3	180 s	4907 MB	240 s	4906 MB
4	264 s	5823 MB	407 s	5825 MB
5	1351 s	5365 MB	2118 s	5366 MB
6	857 s	739 MB	2020 s	738 MB
7	424 s	7617 MB	470 s	7618 MB
8	82 s	2495 MB	85 s	2495 MB
9	357 s	7201 MB	394 s	7203 MB
10	222 s	4785 MB	241 s	4783 MB
11	318 s	5772 MB	287 s	5771 MB
12	423 s	7485 MB	391 s	7488 MB
13	318 s	5876 MB	282 s	5875 MB
14	184 s	3780 MB	176 s	3778 MB
15	147 s	3636 MB	132 s	3637 MB

Fig. 2. Visual comparison of the execution times



Based on these results, it appears that there is a correlation between the complexity of the problems and the execution time. Generally, as the size or complexity of the problem (reflected in MB and execution time) increases, the execution time also tends to rise.

The execution time seems to grow along with the size of the problem or the number of solutions that need to be checked. For instance, problem 5 (with 1351 seconds and 5365 MB) seems significantly more complex than others, which is expected for larger problems.

Memory usage remains relatively constant, though there are minor variations. For example, in problem 6, where the execution time is much shorter, memory does not vary significantly.

Additionally, it can be observed that the performance difference between the two computers appears to increase as the execution time grows. This suggests that as problems become more complex, the disparity in performance between the systems becomes more pronounced.

8 Challenges

When going through the source code, it was a bit tricky to pinpoint where exactly is the CDCL algorithm implemented. We also came across variables like "ws", "sr", "wc", and others that do not immediately make it clear what is happening. So, we had to dig deeper to understand how the CDCL algorithm was applied, especially since it is not separated in the code, but is combined with other methods of the solver class. Another challenge was that the project uses C++, and none of us on the team had much experience with it. We were all more familiar with other programming languages, so getting used to C++ specific syntax and concepts took a bit of time. The complicated algorithms also made our job harder.

References

1. Mr. 4th Dimension. (2024). *Code Reading, MiniSat [Part 1]*. Retrieved from `https://www.youtube.com/watch?v=VZiSX8qIAFc&ab_channel=Mr.4thDimention`
2. Mr. 4th Dimension. (2024). *Code Reading, MiniSat [Part 2]*. Retrieved from `https://www.youtube.com/watch?v=VZiSX8qIAFc&ab_channel=Mr.4thDimention`
3. Mr. 4th Dimension. (2024). *Code Reading, MiniSat [Final]*. Retrieved from `https://www.youtube.com/watch?v=7mpkihV7CrA&ab_channel=Mr.4thDimention`
4. MiniSat. (2024). *MiniSat Manual*. Retrieved from `http://minisat.se/downloads/MiniSat.pdf`