MiniSat Report [DRAFT]

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:

$$(A \lor B) \land (\lnot A \lor C)$$

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 MiniSat

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:

```
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.

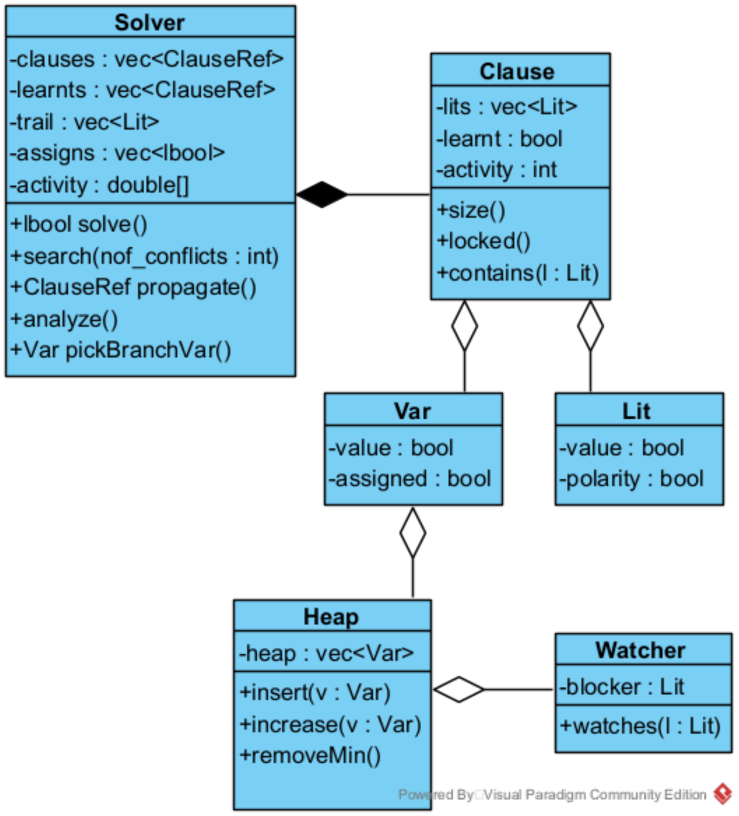
This script made it easy to run the benchmark, ensuring that each run had an output file and a log file for reviewing later.

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.

In this section, we will explain these two main parts in more detail and provide information about the helper classes. First, we'll look at how the `Main` function controls the program and works with the solver. Afterwards, we will dive into the `Solver` class, which is where the actual solving happens and then talk about the helper classes.

Fig. 1. Simplified Class Diagram



5.1 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.

5.2 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.

5.3 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 format suitable 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 to 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.

5.4 The Solver Class

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

5.5 Important Methods in the Solver Class

- `solve()`: The entry point method that initiates the solving process. It prepares the solver for execution by resetting 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()`: Executes unit propagation. If it encounters a clause that cannot be satisfied under the current assignments, it reports a conflict.
- `analyze()`: 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.
- `cancelUntil()`: Handles backtracking by reverting variable assignments up to a specified decision level. This supports non-chronological backtracking, allowing the solver to jump back to an optimal level rather than the last decision point.
- `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.
- `pickBranchLit()`: Implements the decision heuristic that selects the next variable to be assigned.

5.6 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`.

5.7 CDCL Algorithm Implementation

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

– **Decision Making:** Handled by `pickBranchLit()`.
– **Unit Propagation:** Managed by `propagate()`.
– **Conflict Analysis and Clause Learning:** Conducted by `analyze()` and integrated into the search loop.
– **Backtracking:** Executed through `cancelUntil()` for strategic return points.
– **Clause Database Management:** Periodically optimized by `reduceDB()`.

These components are combined in the search() method.

5.8 Helper Classes

Clause Class – Clause Representation Role: Represents a SAT clause, formed by a set of literals. A clause is true if at least one literal is true.

Attributes:

– `vec<Lit> lits`: A vector storing the literals in the clause.
– `bool learnt`: Indicates if the clause was learned during solving or is from the original input.

Methods:

– `size()`: Returns the number of literals in the clause.
– `locked()`: Checks if the clause is locked (used in a solution).

Var and Lit Classes – Variable and Literal Representation Role: Manage boolean variables (Var) and their literals (Lit).

– **Var:** Represents an unassigned or assigned boolean variable.
– **Lit:** Represents a literal of a variable (positive or negated).

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 likely to lead to a solution.

Attributes:

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

Methods:

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

6 Benchmark Results

To evaluate the performance of MiniSat, 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.

6.1 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

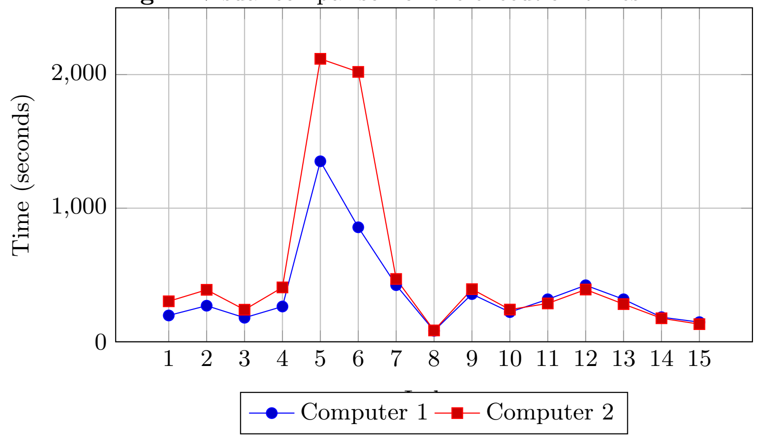
6.2 Results Overview

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.

7 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`