SAT Benchmarks from Circuits with Random Gates

Modoacă Iulia, Todoca Ioana, and Şapcă Miruna

West University of Timişoara
https://info.uvt.ro/

Abstract. Satisfiability solving, the problem of deciding whether the variables of a propositional formula can be assigned in such a way that the formula evaluates to true, is one of the classic problems in computer science. It is of practical interest because modern SAT-solvers can be used to solve many important and practical problems.
In this paper, we aim to investigate the utility of the MiniSat solver, in solving complex propositional problems. We will look at technical details regarding the implementation and operation of the MiniSat solver, including basic algorithms such as DPLL (Davis-Putnam-Logemann-Loveland) and CDCL (Conflict-Driven Clause Learning). So, the main objectives of this paper are the presentation of the MiniSat solver and its installation on two different computing environments, followed by the running of a benchmark used in the SAT 2024 competition, the highlighting and interpretation of the results obtained and the enumeration of the challenges encountered during the realization of the project.

Keywords: MiniSAT · SAT solver · Propositional logic

1 Introduction

Boolean satisfiability (SAT) has been studied for the last twenty years. Advances have been made allowing SAT solvers to be used in many applications including formal verification of digital designs. However, performance and capacity of SAT solvers are still limited. From the practical side, many of the existing applications based on SAT solvers use them as blackboxes in which the problem is translated into a monolithic conjunctive normal form instance and then throw it to the SAT solver with no interaction between the application and the SAT solver. SAT-solvers based on the DPLL algorithm, backtracking with conflict analysis and clause recording (often referred to as learning), and Boolean constraint propagation (BCP) using watched literals are commonly known as conflict-driven SAT-solvers. These solvers can be conceptually divided into three main components:

1. Representation. The SAT instance must be internally represented using efficient data structures, along with any derived information.

2. Inference: While brute-force search alone is rarely sufficient, solvers rely on mechanisms to compute and propagate the direct implications of the current state of information. Inference is typically combined with search to ensure completeness.
3. Search. Search can be seen as another method of deriving information.

A standard conflict-driven SAT-solver manages clauses (of two or more literals) and assignments. While assignments can technically be seen as unit clauses, they are often treated separately due to their distinct nature. The primary inference mechanism used in these solvers is unit propagation: when a clause becomes unit under the current assignment (all literals except one are false), the remaining unbound literal is set to true. This action can make other clauses unit, triggering further propagation until no new information can be derived.

Modern SAT-solvers, like MiniSat, extend this basic framework by integrating techniques such as conflict-driven learning. When a conflict arises under the current assignment, the solver analyzes the conflict by tracing back through variable assignments to identify the cause. This process, involves constructing a conflict clause—a new clause that prevents the same conflicting assignment from reoccurring. This clause is then added to the clause database. Learned clauses serve a dual purpose: they guide backtracking and accelerate future conflict resolution by "caching" the reason for the conflict.

MiniSat employs Boolean constraint propagation using watched literals, a strategy derived from CHAFF(an algorithm designed by researchers at Princeton University for solving instances of the Boolean satisfiability problem in programming using the DPLL algorithm). In this approach, two unbound literals in a clause are marked as watched. These literals are monitored for changes, ensuring that only relevant clauses are revisited when propagation occurs. This efficient management of watched literals minimizes overhead during backtracking, as no adjustments are needed to the watcher lists when undoing decisions.

The learning process also benefits from strategic conflict analysis. When conflicts occur, backtracking is guided by conflict clauses to avoid redundant search paths. This backtracking can be enhanced through non-chronological backtracking, where the solver jumps back to the most relevant decision level, skipping intermediate levels that are not directly related to the conflict.

While the addition of learned clauses accelerates future inferences, an excessive number of such clauses can slow down propagation. To address this, MiniSat periodically reduces the number of learned clauses, retaining only those deemed useful based on heuristic criteria.

MiniSat's extensibility allows for the integration of custom Boolean constraints, which influence its representation, inference, and search mechanisms. By clearly delineating the dependencies between the SAT algorithm and constraint implementation, MiniSat maintains clarity and flexibility while accommodating a wide range of problem domains. This modular approach ensures that the solver remains both powerful and adaptable for diverse applications.

1.1 MiniSAT overview

MiniSat is a lightweight, efficient, and extensible Boolean Satisfiability (SAT) solver that has become very important in the SAT-solving community. Its primary function is to determine whether a Boolean formula in Conjunctive Normal Form (CNF) can be satisfied—that is, whether there exists an assignment of truth values to variables that makes the formula true. SAT problems are foundational in fields like formal verification, automated reasoning, artificial intelligence, constraint satisfaction, and optimization. MiniSat has been widely adopted in both academia and industry due to its combination of simplicity, robustness, and competitive performance, making it suitable for research prototyping, educational purposes, and practical applications.

At the core of MiniSat is the implementation of the Conflict-Driven Clause Learning (CDCL) algorithm, a state-of-the-art technique for SAT solving. CDCL enables the solver to learn from conflicts encountered during the search process, creating new clauses that prevent the same mistakes from being repeated. The learned clauses are generated using the First-UIP (Unique Implication Point) scheme, which ensures conciseness and efficiency. To facilitate fast clause propagation, MiniSat uses a two-literal watching scheme, a highly efficient mechanism that tracks only two literals per clause to reduce computational overhead during unit propagation. Decision-making is guided by activity-based heuristics like Variable State Independent Decaying Sum (VSIDS), which dynamically prioritize variable assignments based on their relevance to recent conflicts, ensuring that the solver focuses on the most promising parts of the search space.

MiniSat incorporates several strategies to enhance its efficiency and adaptability. It uses restart mechanisms to periodically reset the search, allowing it to escape local minima and explore new areas of the solution space. Simplification techniques, such as clause subsumption and self-subsuming resolution, are applied to reduce redundancy in the clause database, improving overall performance. Furthermore, MiniSat supports incremental SAT solving, enabling users to solve related problems iteratively by reusing previously learned clauses and variable states. This feature is particularly valuable in applications where problem instances evolve over time, such as in formal verification or optimization.

MiniSat has also served as a foundation for developing more advanced solvers, including Glucose and CryptoMiniSat, which build on its architecture and ideas while introducing specialized optimizations. Additionally, its influence is evident in SAT competitions, where its concepts and methodologies have shaped the development of other solvers.

MiniSat is used in a wide range of applications. In formal verification, it is employed for tasks like hardware model checking, software bug detection, and equivalence checking. It is also utilized in constraint satisfaction problems, where it can handle scheduling, resource allocation, and optimization challenges. Automated theorem proving and logic synthesis further benefit from its capabilities, and its incremental solving feature makes it a valuable tool for dynamic systems.

MiniSat was developed by Niklas Eén and Nikita Sörensson in 2003, and it

has inspired numerous derivative solvers and is often used as a benchmark or baseline in SAT-solving research. However, MiniSat has limitations: it is specialized for SAT problems and does not natively support extensions like MaxSAT, Quantified Boolean Formulas (QBF), or Satisfiability Modulo Theories (SMT). While it can be adapted to these tasks, doing so requires significant modifications. Additionally, although MiniSat remains competitive, newer solvers with advanced preprocessing or domain-specific optimizations may outperform it in certain specialized scenarios.

1.2 Installing and configuring MiniSAT

To install MiniSAT on Ubuntu through WSL on a Windows system, the first step was to ensure that the system packages were up-to-date. This was achieved by running the commands *sudo apt update* and *sudo apt upgrade* in the Ubuntu terminal, which refreshed the package list and installed any available updates. Next, MiniSAT was installed using the apt package manager with the command *sudo apt install minisat*. After the installation process was completed, the installation was verified by running the *minisat* command in the terminal, confirming that the program was installed successfully and was ready for use. This process allowed MiniSAT to be set up and ready for solving Boolean satisfiability problems within the Ubuntu environment on WSL, which was running on a Windows system.

2 Experiments and benchmarking

MiniSat has proven to be a reliable tool for conducting experiments and benchmarking in the domain of SAT solving.It is a good option for testing new concepts in algorithms and heuristics due to its simplicity and strong performance. MiniSat is often used as a baseline solver to evaluate the effectiveness of new techniques, such as alternative decision heuristics, preprocessing methods, or conflict analysis strategies. Also, it is easy to modify and integrate experimental features due to its compact codebase
MiniSat's efficiency have also made it a popular choice for benchmarking SAT-solving performance across a variety of problem domains. It has been used extensively in SAT competitions to evaluate the relative strengths of different solvers under standardized conditions. Its support for standard input formats, such as DIMACS for CNF, ensures compatibility with widely available benchmark suites like those from the SAT Challenge and SAT Race. These benchmarks include problems from real-world applications, such as hardware and software verification, planning and cryptography.

2.1 The random-circuits benchmark

Random circuits benchmarks typically consist of randomly generated satisfiability problems that are designed to evaluate the efficiency and scalability of SAT

solvrs. These problems are generated using random formulas in CNF), where the variables and clauses are randomly assigned. The idea behind these benchmarks is to provide a wide range of problems with different levels of difficulty, allowing researchers to test solvers on both easy and hard instances.Boolean circuits are composed of logic gates and can easily be represented as CNF formulas. It is well-known that SaT solvers can be more efficient on such circuits-based fromulas by identifying selected types of gates and processing them using dedicated techniques. A prominent example are XOR gates which can be proceed using Gaussian elimination.

A single random gate can be generated for a given

- number of input bits and
- number of output bits.

to generate an entire circuit, we can also consider

- the number of input bits of the circuit,
- the number of output bits of the circuit
- the number of gates, and
- a distance parameter that limits the distance between input variables and output variables of a gate.

2.2 Experimental setup

We ran the experiment on two computers with the following specifications:

Computer 1:

- CPU: Intel i7-8565U @ 1.99 GHz
- RAM: 16 GB
- Operating system: Windows 11

Also, a bash script was designed to automate the processing of .sanitized.cnf files using the SAT solver called minisat. Below is presented the actual implementation of this script:

```
#!/bin/bash

input_dir="/home/raluca/input"
output_dir="/home/raluca/output"

mkdir -p "$output_dir"

for cnf_file in "$input_dir"/*.sanitized.cnf; do
    base_name=$(basename "$cnf_file" .sanitized.cnf)
    output_file="$output_dir/${base_name}.output"
    #run minisat
    minisat "$cnf_file" "$output_file"
```

```
    echo "Processed $cnf_file -> $output_file"
done

echo "All CNF files processed."
```

The scrip reads .sanitized.cnf files from a designated input directory, with a for loop it iterates over all files in the input directory that have the .sanitized.cnf extension. Then it runs the minisat solver on the current file, the output of the solver is saved to the corresponding output file. After all files have been processed, a message is displayed in the terminal to confirm that the script has completed its task.

From the benchmark database we chose the family "random-circuits" to which it belonged 15 benchmarks. In 24 hours, with the Computer1, its specifications enumerated above, it was managed to run with minisat 8 benchmarks.

Benchmark Name	Running Time	File Size	MiniSAT Result
7ac7fabd8c078aea420087a0c80e5563-circuit_32in32out_with_400gates_6in6out_dist64_seed1	77.6646 s	98.635 kb	SAT
6f7a0e1cf94b6b26eafc08a827a692ce-circuit_64in64out_with_64gates_8in5out_dist256_seed1	125.385 s	27.676 kb	SAT
37ca184832fc6fa43a22ae900f1756a2-circuit_32in32out_with_350gates_6in6out_dist64_seed1	67.9899 s	85.275 kb	SAT
71ec94c233016219e12d671594dc88e5-circuit_32in32out_with_70gates_7in7out_dist128_seed1	1032.41 s	68.994 kb	SAT
170b13af977e962321c493544b2bd0a9-circuit_48in64out_with_800gates_4in4out_dist128_seed4	1088.31 s	8.123 kb	SAT
396fd56f3fd7b85afbba4254ea6e746c-circuit_32in32out_with_80gates_7in7out_dist128_seed1	512.985 s	79.170 kb	SAT
8942dca5dc0876fc3f723f738d72de1c-circuit_32in32out_with_64gates_8in6out_dist128_seed2	2186.12 s	61.300	SAT
303480ca7e8322d771c94caf4ebd4e95-circuit_48in64out_with_700gates_4in4out_dist128_seed1	13.2318 s	7.046 kb	SAT

3 MiniSAT code analysis

The MiniSat codebase is accessible to academics and developers because to its architecture, efficiency and simplicity. The code, which is written in C++, is compact, with fewer than 10,000 lines, making it easy to comprehend and modify. Clear interfaces and effective data structures are used in the implementation of important elements including the clause database, conflict analysis, and propagation mechanisms. To optimize performance, strategies like VSIDS heuristics and two-literal watching are skillfully combined. MiniSat is a perfect platform for testing SAT-solving algorithms or creating modifications for specific use cases because of its easy-to-customize restart techniques and support for incremental solving.

*solver function

This function acts as the entry point for solving a SAT problem and integrates CDCL components like decision-making, propagation, and conflict analysis.

The **solve** function begins by setting up the initial conditions for solving a given SAT instance and then enters a loop where the core processes of the CDCL algorithm—propagation, decision-making, conflict analysis, and learning—are

executed. It coordinates the entire SAT-solving lifecycle, from preprocessing to returning a result indicating whether the problem is satisfiable or unsatisfiable. solve function steps:

- Sets up the solver state, including decision levels, variable assignments, and the clause database.
- Handles any problem simplifications or preprocessing steps to reduce complexity (e.g., removing tautologies or subsumed clauses).
- Calls the propagate function to perform unit propagation, which deduces necessary variable assignments based on the current state.
- If a conflict is detected during propagation, it triggers conflict analysis.
- When a conflict occurs, the function invokes analyze to identify the root cause and generate a learned clause.
- The learned clause is added to the clause database to prevent the same conflict from occurring in the future.
- Non-chronological backtracking (via cancelUntil) resets the search state to a decision level relevant to the learned clause.
- If no conflict is found and not all variables are assigned, the pickBranchLit function selects the next variable to decide on, using heuristics like VSIDS. This step advances the search by exploring different potential solutions.
- Periodically, the solver may restart the search to escape local minima or a stagnant search space, reusing previously learned clauses for efficiency.
- Monitors conflict limits and calls reduceDB to prune the clause database, removing less useful clauses to improve performance and manage memory.
- The loop continues until the problem is solved (all variables are assigned without conflicts) or deemed unsatisfiable (no solution exists within the constraints).
- Returns the result of the SAT problem (SAT or UNSAT) and, if satisfiable, provides the variable assignments that satisfy the formula.

Psedocode example of this function:

```
Result Solver::solve() {
    while (true) {
        Clause* conflict = propagate();
        if (conflict != NULL) {
            if (decisionLevel() == 0) return UNSAT;
            Clause* learned = analyze(conflict);
            cancelUntil(level);
            addClause(learned);
        } else if (allVariablesAssigned()) {
            return SAT;
        } else {
            Literal decision = pickBranchLit();
            newDecisionLevel();
            enqueue(decision);
        }
```

```
        if (shouldRestart()) performRestart();
    }
}
```

***propagate function**

The propagate function in MiniSat is a core component of both the DPLL (Davis-Putnam-Logemann-Loveland) and CDCL (Conflict-Driven Clause Learning) algorithms. Its primary role is to perform unit propagation, a critical process that simplifies the SAT problem by deducing assignments to variables based on the current state of the solver.

Unit propagation is an essential part of SAT solving that takes advantage of the fact that if a clause has all but one of its literals assigned a value, the value of the remaining literal is automatically determined. This is because a clause can only be satisfied if at least one of its literals is true. If a clause is unsatisfied and only one literal is unassigned, the unassigned literal must be assigned in a way that satisfies the clause. This deduction reduces the search space and accelerates the solving process.

Steps of the propagation function:

- MiniSat uses the two-literal watching technique to efficiently handle clause propagation. In this method, instead of keeping track of all literals in a clause, the solver only watches two literals at a time. This reduces the overhead of propagating changes across the entire clause database.
- Each clause in the solver has two watched literals, which are chosen in a way that allows efficient detection of when a clause is unit (i.e., has only one unassigned literal). When a literal is assigned, the solver checks the watched literals and updates the clause accordingly. Triggering Propagation:
- The propagate function starts by examining all clauses with unassigned literals. If a clause has only one unassigned literal, it triggers unit propagation, forcing the assignment of the remaining literal. This is done recursively, as each new assignment might cause further assignments in other clauses.
- During propagation, if an assignment causes a clause to become unsatisfied (i.e., all literals in the clause are false), a conflict is detected. In that case, the solver will stop propagation and trigger conflict analysis. If a conflict is detected during propagation, the function will return a pointer to the conflicting clause, and the solver will enter the conflict resolution phase (via the analyze function).
- MiniSat's use of two-literal watching ensures that propagation is done efficiently. Only the minimal necessary checks are made, and updates to the clause database are handled quickly. The function also ensures that the solver avoids redundant checks by keeping track of the state of each literal in the current decision level.

***analyze function**

When a conflict is detected during unit propagation (typically via the propagate function), analyze is responsible for diagnosing the conflict, deriving a learned clause, and determining the appropriate backtracking level. This allows the solver

to avoid redundant searches and focus on more promising parts of the search space.
Steps if the analyze function:

- When a conflict is detected during propagation, the analyze function is invoked with the conflicting clause as input. The first step of analysis is identifying the conflicting literals—the literals that caused the conflict. This involves backtracking from the conflicting assignment through previous decisions and learned clauses to determine where the conflict originated.
- The analyze function traces the conflict backward using a technique called First-UIP (Unique Implication Point). The idea is to identify the point at which the conflict could have been avoided by making a different decision. First-UIP analysis traces the conflicting literals backward, determining the implication chain (i.e., which assignments lead to the conflict). This chain is analyzed to derive the learned clause, which will help prevent the solver from making the same mistake in future decision levels.
- After tracing the conflict, the solver creates a learned clause. This clause is derived from the conflict and added to the clause database. The learned clause typically represents the condition that must hold true to avoid the conflict in the future. The learned clause is typically a non-chronological clause, meaning it involves literals from multiple decision levels. This allows the solver to avoid the specific conflict by skipping over the erroneous search path.
- A critical aspect of CDCL solvers like MiniSat is non-chronological backtracking. After deriving a learned clause, the solver must determine the appropriate backtrack level. This level is not necessarily the most recent decision level but the one that allows the solver to skip over the conflict and explore different paths. MiniSat uses the backtrack level to reset the decision stack to a level where it can make a better choice and continue searching for a solution.
- Once the learned clause is derived and added to the clause database, it is used in subsequent propagations to prevent the same conflict from occurring. This significantly speeds up the solving process by reducing redundant searches.
- To ensure the learned clause is as small and concise as possible, MiniSat applies clause minimization techniques. This step attempts to remove any unnecessary literals from the learned clause, making it more efficient in future propagation.

Pseudocode example of this function:

```
void Solver::analyze(Clause* conflict) {
    // Initialize the learned clause
    vector<Literal> learnedClause;
    int backtrackLevel = 0;

    // Trace the conflict backwards to find the first UIP
    while (!conflict->isUnit()) {
```

```
        // Analyze the conflict and generate a learned clause
        Literal p = conflict->getFirstUIP();
        learnedClause.push_back(p);

        // Calculate the backtrack level
        backtrackLevel = max(backtrackLevel, decisionLevel(p));

        // Update the conflict clause
        conflict = conflict->getParentClause(p);
    }

    // Minimize the learned clause and add it to the database
    learnedClause = minimizeClause(learnedClause);
    addLearnedClause(learnedClause);

    // Backtrack to the calculated level
    cancelUntil(backtrackLevel);
}
```

simplify function

It is typically used to remove clauses or literals that are redundant or unnecessary, which helps in improving both the performance and memory usage of the solver. This function is particularly important in the context of the Conflict-Driven Clause Learning (CDCL) algorithm, where learned clauses accumulate over time, potentially leading to a larger clause database that could degrade performance.
Key Operations in Simplification;

- Clause Removal: Eliminate redundant clauses.
- Literal Removal: Remove literals that are no longer needed.
- Clause Minimization: Reduce the size of learned clauses for better performance.
- Dynamic Simplification: Perform simplification dynamically during the solving process to keep the clause database small and relevant.

Acknowledgments. Ioana Todoca was responsible for running the benchmarks in MiniSAT, for creating the bash script that was used for automatisation and efficiency and also for the writing of the documentation. Iulia Modoaca was responsible for running the benchmarks using MiniSAT and for writing the documentation as well. Miruna Sapca and Bogdan Jude did part of the research and contributed in writing the documentation.

References

1. An Extensible SAT-solver by Niklas Een, Niklas Soorensson http://minisat.se/downloads/MiniSat.pdf

2. MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization by Niklas Een, Niklas Soorensson http://minisat.se/downloads/MiniSat$_v$1.13$_s$*hort.pdf*
3. MINISAT - SAT ALGORITHMS AND APPLICATIONS by Niklas Soorensson http://minisat.se/downloads/escar05.pdf
4. http://www.cs.chalmers.se/ een/Tip/Temporal$_I$*nduction*$_b$*y*$_I$*ncremental*$_S$*AT*$_S$*olving.ps.gz*
5. http://minisat.se/downloads/PracticalSAT$_l$*inux.zip*$Scientific Objectives for a Minisat$: $CoRoT by Baglin, A., Auvergne, M., Barge, P., Deleuil, M., Catala, C., Michel, E https://adsabs.harvard.edu/full/re$
6. COROT: A minisat for pionnier science, asteroseismology and planets finding by Annie Baglin, COROT Team https://www.sciencedirect.com/science/article/abs/pii/S0273117702006245
7. MiniSAT User Guide: How to use the MiniSAT SAT Solver https://dwheeler.com/essays/minisat-user-guide.html
8. https://github.com/niklasso/minisat
9. Introduction to MiniSAT -University of Nebraska https://cse.unl.edu/ choueiry/S18-235H/files/SATslides03.pdf
10. https://www.stackage.org/package/minisat-solver