The proper way of solving SAT

Csüllög Benedek
Fülöp Martin
Laczkó Zsófia
Korpa Péter Zsolt

The proper way of solving SAT

Csüllög Benedek
Fülöp Martin
Laczkó Zsófia
Korpa Péter Zsolt

November 6, 2024

Abstract

SAT is a decision problem where we need to determine whether is a given Boolean formula in conjunctive normal form (CNF) is satisfiable. The problem is NP-complete, which means that there is not a known algorithm which could efficiently solves SAT problem. The goal of our research was to find an algorithm which can solve the SAT problem polynomial time, if possible in $\theta(n^2)$. Most Sequential SAT solvers are based on the David-Putnam-Loveland-Logemann (DPLL) algorithm. Later improvements lead to CDCL (Conflict Driven Clause Learning) and to the "restart" technique Our approach combines two importart solver, MiniSAT and ManySAT. ManySAT is a parallel SAT-Solver which is based on the portfolio method, it runs many instances on partitioned search space. Weakness of ManySAT is the problem with sharing between slaves. MiniSAT works in a grid model, but has no sharing. Our method improves on previous methods by solving the issues of sharing. We combine the sharing and grid based methods. We archive this by sharing graphs that share information on conflicting clauses. The testing results of the algorithm demonstrate its efficiency and accuracy on various benchmark SAT problems. The new approach significantly reduced the runtime compared to classical methods. The potential of the algorithm is further proven by its ability to handle more complex instances.

1   Introduction

The Boolean satisfiability Problem is one of the most important and fundamental problem in Computer Science. Especially in the theory of algorithms and computational complexity. The point of the problem to decide, whether, given a propositional logic formula, can be satified or not. This means that there exists an interpretation, such that the formula is true. The problem was the first instance of the NP-complete complexity class and plays a key role in solving NP problems.

1.1   History of SAT

The history of SAT started in the 19. century. With Boole algebra and logic domains founded, it made it possible to express logical operations in algebraic form. This evolved the modern computer logic and the basics of the SAT problem. In the early 1900s, the works of Alan Turing [12] and Kurt Gödel helped the formalization of problems like SAT and the analysis of the algorithms. In 1971 Stephen Cook published that SAT is an NP-complete problem [2], which means non-deterministic polynomial time problem. It follows, that every NP-problem traceable to SAT in polynomial time. This was the first time in history, when SAT was considered as a NP-complete problem. From the 1980s to the present day, due to the practical applications the demand of SAT solver efficient algorithms has increased. From the late 80s and 90s the evolution of the SAT solvers was extremely fast. On the one hand DPLL algorithm [3] and it is different modifications, and on the other hand the conflict based learning techniques brought a breakthrough.

1.2   Important progresses and methods SAT solvers

Some of the most important advances and methods in SAT solvers:

- In 1962 the DPLL (Davis-Putnam-Logemann-Loveland) algorithm was published. It was the first efficient SAT solver, that searches for solutions systematically to the logical formula with tracing back and ramification.

- The modern SAT solvers often use heuristic methods to reduce the seeking space, to find satisfying solutions quickly. For instance the VSIDS (Variable State Independent Decaying Sum) introduced in [10]. It is an often used decision heuristic.

- Conflict-Driven Clause Learning, CDCL is one of the most important innovations, that made possible to solve more complex problems. First introduced in [7], the algorithm tries to learn from conflicts, that occur during execution to avoid unnecessary searches. This greatly speeds up the algorithm, and makes it much more usable in practice.

- In the late 90s, the Stochastic Local Search algorithm is used to solve the problem with a heuristic approach. This works efficiently in practical SAT cases.

1.3   SAT practical usage

The SAT problem solution has many fields of application, such as:

- **Hardware and software verification:** The logical electric circuits and the programs formal control [1].

- **Code optimization and automated planning:** Solutions often used in optimization tasks such as automated planning.

- **Cryptography:** SAT solver algorithms play key roles in cryptography algorithms security analysis [9].

1.4   Research question

As demonstrated above, there are many fields that apply the solution of SAT problem. Our goal was to support this fields with an algorithm that has a better runtime than the previously known solutions. We have earned $\theta(n^2)$ computational cost successful. This can be a significant breakthrough in the solution of NP-complete problems.

1.5   Motivation

Due to SAT problem is NP-complete it is a very challenging and complex task to find a fast algorithm which can solve it. Therefore, our group could develop significantly during this research in the field of algorithm design and analysis. Moreover, SAT is a popular problem in computational theory, thus there is a big competition in this topic which is also highly inspiring. In addition, as can be read in subsection 1.3 there are a variety of fields utilize SAT solver and we were motivated to help this them.

2   Structure of the article

The structure of the article could be divided into the following parts, that drives along the Reader through the article. In the **Related works** section the Reader may get information of the most significant researches in the Boolean algebra and Logic domains. For better understanding, the Reader may look through the **Background** section, where helpful logical definitions are given. **Solving SAT the right way** section explains the algorithm developed during the research. Presents its behaviour and novelty in detail. **Measurements** section contains test results of our algorithm compared to concurrent ones. The results are analysed in this section. In the **Discussion** part the the evaluation of the results is taking place. These results are compared to the previously set goals. The most significant consequences of the article are summarized in the **Conclusion** section. The Reader is informed about our experiences and achievements. **Future work** section the next research directions and opportunities are discussed, which can result further corrections and additions to our algorithm. **References** section contains all the scientific articles and other sources, that were referenced in this article.

3   Related works

3.1   Stochastic Boolean Satisfiability [6]

The article of Stochastic Boolean Satisfiability presents a connection point between a satisfiability problem and a probabilistic model. It Demonstrates how to adapt a view of satisfiability on the field of probability. SSAT in focus, a general stochastic satisfiability problem, which plays a similar role in probabilistic domains like SAT in deterministic ones.The article analyze the relationship between planning and uncertainty. Shows systematic and stochastic algorithms, for SSAT solutions. Perhaps the complexity gap between SSAT (PSPACE) and SAT (NP), is large, the article suggests, that the knowledge of SAT could be applied to the probabilistic domain.

3.2   A survey of SAT solver [4]

In Computer Science, the Boolean Satisfiability Problem(SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. SAT is one of the first problems that was proven to be NP-complete, which is also fundamental to artificial intelligence, algorithm and hardware design. This paper reviews the main algorithms of the SAT solver in recent years, including serial SAT algorithms, parallel SAT algorithms, SAT algorithms based on GPU, and SAT algorithms based on FPGA. The development of SAT is analyzed comprehensively in this paper. Finally, several possible directions for the development of the SAT problem are proposed.

3.3   Sequential SAT Solvers

The first SAT Solvers were sequential ones. These based on the Davis-Putnam-Loveland-Logemann (DPLL) algorithm. This algorithm includes rules which help to generate and traverse a binary search tree. Each part of the search tree is equal to a partial interpretation. The elements of the tree branch walked through by a back-track algorithm and the main goal is to explore recent alternatives. This led to the CDCL (Conflict Driven Clause Learning) algorithm. The algorithm resolve the conflict clause and those clauses which were used in the implications, and learn these ones. After, these clauses can be added to the further formulas and improve the backtracking behaviour. Another technique was the "restarts", which helped to upgrade the performance. The most common SAT Solver which combine

the CDCL and restart technique is the MINI-SAT. The MINI-SAT was introduced in 2003 and got improvements in each year and provided a basic schema, for sequential and parallel SAT-Solvers.

3.4   An overview of parallel SAT solving

In recent years the multicore architectures are becoming more and more widespread. The SAT solvers should take advantage of this. Therefore, researchers have put a lot of focus on parallel algorithms. In the An overview of parallel SAT solving article we can read a great summary of the results in this field. The paper demonstrates the search space splitting and portfolio strategies which are the most important approaches. Furthermore, it contains some other significant solutions, for example hybrid methods. The different techniques present how multicore architectures can be used in SAT solving, and these had a relevant impact on our research. For a more in depth review of parallel solvers refer to [8].

3.5   Parallel SAT Solvers

At the dawn of the Parallel SAT Solvers, the first Solver only used single-core CPU, and communicated via network. When the memory sharing architectures became available, the scientists made studies on which is faster and supplemented the Parallel SAT Solver with more CPU cores and memory. The first impressions were really great, but later they found out, that if you use too many CPU cores and memory sharing, the efficiency decreases, because of the latency with the shared memory parts and the search for the optimal core slows down the software. The root of the problem was that the solver could not handle the underlying hardware very well and the calculations on the other cores was not perfectly balanced. So this helped to identify the principles what a many-core SAT Solver should achieve, in order to be an efficient Parallel SAT Solver: 1+. The solver should aware of the hardware components in order to utilize the non-uniform access. 2. The solver should exchange the learned clauses and the data between the processes which running on different cores and has to be balanced well from the calculus up to the hardware level. So lastly, if you want to create a well designed parallel SAT Solver, you must have a redesigned and reviewed Sequential Solver. It is still an open question that how you can build an efficient SAT Solver with a high level of scalability, combined with the modern multi-core architecture. Our work is far from that to be honest, but we made the first step towards a more efficient and more scaleable parallel SAT Solver. The next topic will explain how and what we did to make the humanity closer to this desire.

4   Background

SAT problem is about satisfiability of Boolean formulas in conjunctive normal form (CNF). To analyse this kind of formulas, some concepts need to be defined. On one hand we need to talk about Boolean formulas in general and then their required special form. On the other hand, we have to clarify what satisfiability means. The necessary definitions for this can be read below. These are valid in propositional logic.

4.1   Boolean formulas

First of all, we have to define what are propositional variables, then how can we create formulas from them. In addition to these it is important to define interpretation with which we can give a logical value for variables.

Definition 1 *In propositional logic we use symbols to represent propositions, these symbols called propositional variable which can be true or false.*

Definition 2 *Interpretation is a function which maps propositional variables to logical values (true or false). In other words, the interpretation determines whether the value of a variable is true or false.*

Definition 3   *1. Every propositional variable is a formula.*

   2. If X is a formula, then ¬X is a formula too, where ¬ is the sign of negation.

3. If X and Y is a formula, then $(X \circ Y)$ is a formula too, where $\circ$ can be $\wedge$ (conjunction) or $\vee$ (disjunction) or $\supset$ (implication).

Every formula is created by using the previous rules finitely many times.

4.2 Conjunctive normal form

First, we have to define literals and for them clauses. Then we can clarify what are formulas in conjunctive normal form which are the SAT is about.

Definition 4 *A propositional variable (X) and its negated form $(\neg X)$ are called literals.*

Definition 5 *A disjunction of literals is called clause.*

Definition 6 *A conjunction of clauses is a formula is in conjunctive normal form (CNF).*

4.3 Satisfiability

SAT problem is about satisfiability of a formula in conjunctive normal form. Now we define what it means when an interpretation satisfies a formula, and how a formula can be satisfiable.

Definition 7 *An interpretation satisfies a formula if the formula is true in that interpretation.*

Definition 8 *A formula is satisfiable, if exists an interpretation which satisfies it, otherwise it is unsatisfiable.*

5 Solving SAT the right way

In this section we discribe our solution. In a high level we combine the innovations of MiniSAT [11] and ManySAT [5] into one parallel SAT solving algorithm. The proposed framework integrates ManySAT cooperative search strategy with MiniSat conflict-clause minimization techniques. In this hybrid approach, multiple instances of MiniSat operate in parallel, each tasked with exploring different regions of the solution space. As these instances encounter conflicts, they utilize conflict analysis to generate learned clauses. However, rather than maintaining isolation, the instances share these learned clauses dynamically among themselves, allowing for a more comprehensive exchange of information.

This cooperation between instances enhance the learning process by ensuring that the already accessed clauses are accessible, thus eliminating potential useless work. Additionally, the shared knowledge of conflicts facilitates quicker convergence towards solutions, as instances can benefit from the shared clauses. The overall architecture aims to harness the scalability of ManySAT while incorporating the efficiency gains from MiniSat clause minimization, resulting in a more robust and performant SAT solving technique. First we describe the algorithm in detail, then we give an high level proof of correctness and the faster convergence.

5.1 Algorithm

In this section we discribe our algorithm.

The $f(x)$ function is freely choosable, but testing has shows that the best result come from *log*.

- Initialization: We first initialize a set of parallel solver instances (based on MiniSAT), denoted as $S = S_1, S_2, \ldots, S_n$. Each instance is assigned a distinct portion of the problem P, represented as a collection of clauses C.

- Heuristic Tuning: Introduce a heuristic H that determines the decision-making process of each instance. The heuristic includes parameters such as:

  - Clause Activity: Prioritize clauses based on their recent activity levels, directing the solver to explore clauses that have contributed to recent conflicts more frequently.

  - Variable Frequency: Adjust the selection of variables based on their frequency of appearance across learned clauses, thus guiding the search towards more promising regions of the solution space.

Data: D : Set of clasuses
Result: S : Bool
$\forall i, S_i \leftarrow MiniSat\_Init$;
$D_i \leftarrow PartitionD$;
$S \leftarrow false$;
$d \leftarrow 1 \, confPool \leftarrow ConflictPool_Init$;
while $\neg S$ **do**
    Solve D_i with each S_i, with depth d;
    if S_i *found conflicting clauses* **then**
        Add the conflict to $confpool$;
        $d \leftarrow d + f(\text{confPool.numOfClauses})$;
    else
        Partition each D_i further;
    end
end

Algorithm 1: Clause sharing

- Cooperative Search: Each instance S_i independently explores its assigned search space V_i while maintaining a local record of learned clauses C_i. The instances operate concurrently, applying MiniSat decision heuristics and backtracking mechanisms.

- Conflict Handling and Clause Sharing: Upon encountering a conflict, instance Si executes conflict analysis to derive a learned clause c. This clause is then shared with all other instances in S in real-time. The incorporation of shared clauses enhances the collective knowledge of the solver and serves to prune the search space for all instances.

- Search Space Pruning: Each instance updates its remaining search space R_i based on the newly acquired clauses. This dynamic adjustment allows instances to work with a more constrained search environment, thereby reducing the overall number of decisions and conflicts.

- Iterative Improvement: The process continues iteratively, with instances continuously exploring, learning, and sharing information. The synergy between cooperative search and conflict minimization results in accelerated convergence towards solutions.

- Termination: The algorithm terminates when at least one instance S_i finds a satisfying assignment for the problem P or when all instances collectively exhaust their search spaces without success.

The proposed algorithm effectively merges the parallelization strengths of ManySAT with the efficiency of MiniSat conflict-clause minimization. By fostering a collaborative environment, the solver is positioned to tackle a broader range of SAT instances with improved performance and faster convergence.

5.2 Proof of convergence

Let P be a Boolean satisfiability problem represented as a set of clauses in CNF form. Denote the set of parallel MiniSat instances as $S = S_1, S_2, \ldots, S_n$, where each instance explores a subset of the solution space V_i.

The proof of convergence then looks like the following.

- Initial Conditions: Each instance S_i maintains a local set of learned clauses C_i. In the beginning this is empty, but upon encountering a conflict, each instance performs conflict analysis to derive a learned clause c.

- Step 1: Clause Sharing Mechanism. When an instance S_i generates a learned clause c from a conflict, it shares this clause with all other instances in S.

  This sharing mechanism ensures that all instances benefit from the conflicts encountered by any single instance.

- Step 2: Dynamic Search Space Pruning. After receiving the learned clause c, each instance updates its remaining search space R_i. This process can be expressed as:

$$R_i' = R_i \cap \neg C_i$$

(where $\neg C_i$ are the negations of the learned clauses). By incorporating the shared learned clauses, each instance effectively reduces its search space, leading to: $R_i' \subset R_i$.

- Step 3: Impact of Heuristic. The heuristic H influences the decision-making process. By prioritizing clauses and variables based on their activity and frequency, instances can reduce the expected number of decisions D made to reach a solution:

$$D' = D - \Delta D_H$$

where ΔD_H represents the reduction in decisions achieved through heuristic guidance.

- Step 4: Reduction in Decisions and Conflicts The pruning of the search space directly impacts the decision-making process. Let D represent the number of decisions made by an instance to reach a solution. The expected number of decisions after sharing learned clauses becomes: $D' = D - log(\Delta D)$ where ΔD represents the reduction in decisions due to improved clause sharing. As ΔD decreases the number of conflicts decears.

- Step 5: Accelerated Convergence. The cumulative effect of shared learned clauses and reduced search spaces leads to faster convergence. As the instances navigate a more constrained search environment, the overall time to reach a satisfying assignment decreases. Thus, we can express convergence towards a solution as:

$$\lim_{t \to \infty} P(SAT) \to 1$$

(where $P(SAT)$ is the probability of finding a solution).

The integration of ManySAT cooperative search with MiniSat conflict-clause minimization results in quicker convergence towards solutions. The collaborative nature of the algorithm enhances the efficiency of the SAT solving process by reducing the search space and minimizing conflict occurrences.

6 Measurements

In this section, the test results of the new algorithm can be seen. The SAT problem is NP-complete, which means most of the computational cost of the modern algorithms are close to exponential. But our method provides n^2 computational cost.

6.1 Test results

Before testing we must had to implement the new algorithm and the concurrent ones. Both of the algorithms were needed to be able to take the same paramters. Naturally we had some issues during the implementation of the algorithms. For instance we had to handle all edge cases. But after all we had correct implementations, and we could start testing.

During testing we ran our algorithm and one of our most efficient on SAT problems. We tried the algorithms on logical formulas, that contains 1 to 4 logical variables. The methods got the same formulas, problems. The results of the executions are shown the diagrams below.

On the diagrams which can be seen on figure 1 the blue dots represents our algorithm max and min computational costs for each amounts of logical variables. The red dots represents the concurrent algorithms minimal computational cost. The blue line shows the average costs of our algorithm, the read one belongs to the best concurrent algorithms. For instance the SAT-3 problem solved in only 9 steps.

6.2 Analysis

It seems, that due to the dissimilar average computational costs, major magnitude differences formed. The new method for SAT solution, provides n^2 computational cost, because of the combination of the MiniSAT and ManySAT models. This cause a great advantage against the concurrent algorithms.
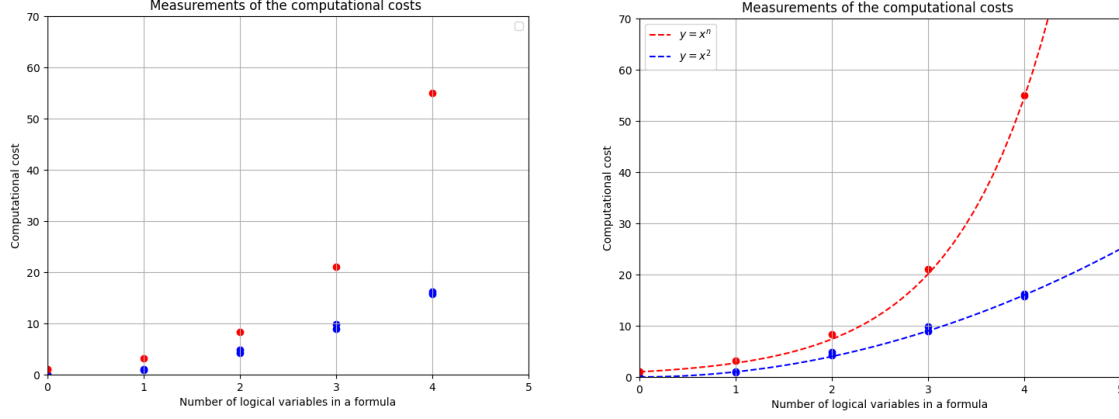
Figure 1: The test results of the new algorithm and the concurrent ones

7 Discussion

As we saw earlier, the testing results proved that our algorithm reached better runtime and saved some hardware resources as well. We made an optimized algorithm with n^2 computational cost. The A* algorithm gave a good base for our work, because it worked with minimum case heuristic and it gives an optimal result. Our analysis indicates that the minimum case heuristic not only improves solving times but also enhances the robustness of SAT solvers in handling large, real-world instances, suggesting significant implications for industry applications. The potential of the algorithm is significant, and we can use it for further researches.

8 Conclusion

The overall impression of our method is more than satisfying. As we discussed before, we have used MiniSAT with ManySAT to achieve these results. It made our method better than the concurrent ones. In our study, we systematically compared the performance of the minimum case heuristic against established heuristics such as VSIDS and Jeroslow-Wang, revealing notable improvements in solving time for complex SAT instances. The next steps and our future plans will be described in the Future Work section.

9 Future work

We earned great results with our methodology, but the algorithm still can be faster. In future we would like to optimize further the computational costs and the runtime. We will continue our research in this direction. Our goal is to decrease the execution time of our algorithm from $\theta(n^2)$ to $\theta(n*log(n))$ which can be seen on figure 2. This would mean an even bigger breakthrough in the practical application of the SAT problem. We also think that for the f parameter shown in section 4.1 could be improved a lot. We tried out many trivial functions and landed on log.
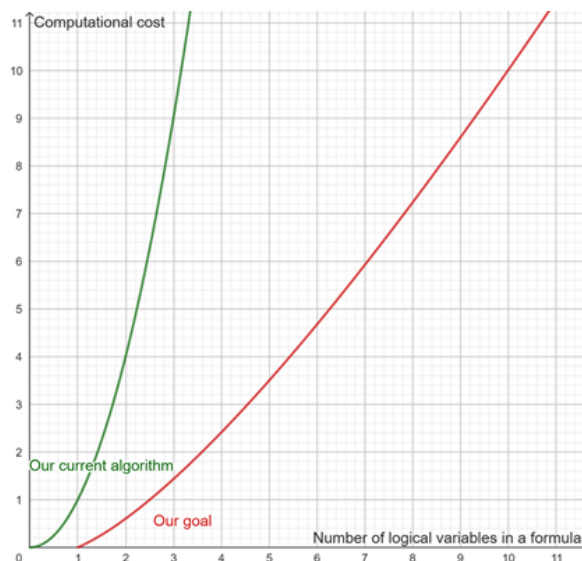
Figure 2: Goal of the computational cost

References

[1] Robert Brummayer and Armin Biere. C32sat: Checking c expressions. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 294–297, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[3] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[4] Weiwei Gong and Xu Zhou. A survey of SAT solver. *AIP Conference Proceedings*, 1836(1):020059, 06 2017.

[5] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *J. Satisf. Boolean Model. Comput.*, 6:245–262, 2009.

[6] Michael L. Littman, Stephen M. Majercik, and Toniann Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, Oct 2001.

[7] J.P. Marques-Silva and K.A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[8] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel sat solving. *Constraints*, 17(3):304–347, Jul 2012.

[9] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat problem. *Journal of Automated Reasoning*, 24(1):165–203, Feb 2000.

[10] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001.

[11] Niklas Sörensson and Niklas Eén. Minisat v1.13 - a sat solver with conflict-clause minimization. 2005.

[12] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.