William Harmer

Registration number 100388574

2026

The Development and Comparison of SAT Solvers

Supervised by Laure Daviaud

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The design, implementation, and comparative evaluation of seven SAT solvers are presented, each developed to solve Boolean satisfiability problems (SAT) in conjunctive normal form (CNF). The solvers progress from a naive brute-force strategy, through enhancements incorporating unit propagation and pure literal elimination, to the Davis–Putnam–Logemann–Loveland (DPLL) algorithm, and culminate in a conflict-driven clause learning (CDCL) approach. A CNF generator was developed to produce large datasets of formulae with configurable parameters, enabling benchmarking for comparison between solvers. Results show that brute-force fails to scale beyond extremely small formulae. DPLL and brute-force variants with unit propagation all perform similarly, handling modest sized problems. In contrast, CDCL solves formulae up to six times larger than the modest approaches and does so in less than half the runtime, demonstrating its superior scalability and efficiency, and justifying its widespread use in practice.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr Laure Daviaud, for her guidance, support, and valuable feedback throughout the course of this project.

Contents

List of Figures

List of Tables

1. Introduction

Creating and evaluating Boolean formulae is how digital computers operate at a hardware level. The Boolean satisfiability problem, also known as SAT, asks whether there exists an assignment of truth values for the literals / variables of a given formula that satisfies (solves) the Boolean formula. As their name suggests, SAT solvers are specifically designed to answer SAT for given formulae(Biere et al., 2021).

SAT is classified as a nondeterministic polynomial-time complete (NP-complete) problem, meaning there is currently no known method for solving it in polynomial time. Consequently, optimising solvers to be capable of handling larger, more complex formulae within a reasonable time frame has become a significant area of research (Cook, 2000).

The importance of SAT solvers comes from their broad range of applications throughout computer science. If a question or problem can be encoded as a Boolean formula, a SAT solver can be used to determine the solution (Biere et al., 2021). For example, automated planning and scheduling, cryptanalysis, software testing and debugging, reverse engineering, resource and route allocation, mathematical theorem proving, financial portfolio optimisation, healthcare treatment planning, and answering constraint satisfaction problems such as Sudoku are just some of the ways in which SAT solvers are applied (Claessen et al., 2008).

More specifically, SAT solvers are well suited to solving problems with strict constraints and clear solutions. Questions involving approximation or prediction typically lie beyond the scope of SAT solvers and align more naturally with machine learning techniques (Biere et al., 2021).

In this paper, seven different SAT solvers were implemented and compared, focusing on differences in their implementations and how these impact performance on large datasets containing thousands of formulae. The process of generating these datasets and the methods used for comparing solver performance are also examined, with the ultimate goal of identifying which solver performs best in terms of both time and space complexity.

2. Background

2.1. What Is a SAT Solver?

SAT involves determining whether there exists a set of truth assignments (true or false) that will make a formula evaluate to true (satisfiable). A SAT solver is an algorithm designed to solve this problem for a given formula (Biere et al., 2021). The problem was first introduced by Stephen Cook in his paper titled 'The Complexity of Theorem-Proving Procedures' (Cook, 1971), proving that SAT was a NP-complete problem.

2.1.1. NP-Complete Problems

An NP-complete problem is one that belongs to both the NP and NP-hard categories. Specifically, an NP-complete problem:

- Requires more than polynomial time to solve, usually exponential time.
- Allows for a solution to be verified as correct or incorrect in polynomial time.
- Is one of the hardest problems within the NP class.

The significance of the third point lies in the fact that if we were to discover that a single NP-complete problem is actually solvable in polynomial time (that it belongs to P), it would imply all NP problems are solvable in polynomial time (P = NP) (Goldreich, 2010). We will revisit the significance of this later.

2.1.2. Real World Applications

The application of SAT solvers spans numerous areas of computer science. For example, in game theory, they are used to compute optimal strategies for winning. In automated theorem proving, they can verify the correctness of mathematical proofs. In cryptography, they test cryptographic protocols for potential flaws and attempt to break encryption schemes. SAT solvers are also applied to optimisation problems, such as the travelling salesman problem, job scheduling, and resource allocation. Additionally, they are crucial in software debugging, where they help identify bugs, and in circuit design to detect errors in hardware. In artificial intelligence, given certain constraints, SAT solvers assist in determining optimal outcomes (Claessen et al., 2008). All of these

tasks are solved through the same process of converting a problem or question into a Boolean formula, which is then solved by a SAT solver (Biere et al., 2021).

2.1.3. How Are SAT Solvers Measured?

SAT solvers are evaluated using various metrics, depending on their specific application. The most important criteria to consider will vary based on how the solver is being used (Biere et al., 2021).

Correctness is one such metric. It determines whether the answer provided is correct or not. All solvers must have one hundred percent correctness for every answer they give, otherwise, the solver is considered faulty. It is important to note that there can be hundreds of correct answers for a single formula, meaning multiple solvers can provide different answers, and both still be correct. Additionally, some formulae may not have a correct answer at all, and so just because a solver cannot find an answer, does not mean the solver lacks correctness (Biere et al., 2021).

Completeness is another metric. This criterion examines whether the solver will always find a solution to a formula if one exists. Some solvers do not explore the entire search space, meaning there will be instances where they cannot solve the formula, even if a solution exists. Typically, these solvers are designed with this limitation in mind, as they may excel in other areas. For example, walkSAT (Lozin, 2021), developed by Henry Kautz and Bart Selman in the early 1990s, does not always guarantee one hundred percent correctness. This is because it uses greedy and stochastic search methods to find solutions more quickly. This approach is advantageous when dealing with large formulae, where solving the problem by iterating through the entire search space in a reasonable amount of time may otherwise be unfeasible (Lozin, 2021). This paper only implements solvers that explore the entire search space (one hundred percent completeness), and so this metric is not analysed.

Runtime is a critical performance metric for SAT solvers. This measures how long it takes for a solver to reach an answer (Biere et al., 2021). Of course, runtime depends on both computational power and algorithmic efficiency. There will be a point, given a large enough formula, when a solver will not take a feasible amount of runtime to

solve the formula (Cook, 1971). We know this due to Stephen Cook's analysis of the SAT problem being NP-complete, discussed earlier. SAT solvers cannot, at this current point in time, have polynomial time complexity (Goldreich, 2010), and this is why the metric of scalability is also important. Scalability is the measure of how well a solver can handle increasingly large and complex instances of SAT problems, up to a point when it can no longer do so. A higher scalability in a solver means that larger formulae can be solved, and thus more complex questions, if expressed in Boolean formula, can be answered (Biere et al., 2021).

Finally, a solver must be capable of running with an appropriate allocation of computational resources (Biere et al., 2021). An efficient solver is of little use if it cannot be executed on the available computational power. Random access memory (RAM) is particularly crucial for scalability, as SAT solvers require sufficient memory to store formulae, intermediate results, and data structures. With more RAM, a solver can manage larger and more complex problems, ensuring greater scalability as the problem size increases (Manthey and Saptawijaya, 2016). All tests discussed in the following sections were conducted on the same device, ensuring consistent computational resources throughout. Consequently, only RAM usage will be benchmarked and analysed, as it represents the primary variable of interest in evaluating solver performance.

2.1.4. The Impact of Polynomial SAT Solvers

Many real-world applications that utilise SAT solvers are computationally expensive and take a long time to solve. If it were proven that SAT problems could be solved in polynomial time (P = NP), it could dramatically speed up the solution process for a wide range of challenging problems (Vega, 2025). This breakthrough could lead to significant advancements in areas we currently struggle with, such as accelerating medical research, improving artificial intelligence, and optimising renewable energy systems. However, such a breakthrough also brings potential risks, including the weakening of cryptographic systems and the potential for increased misinformation and AI manipulation (Vega, 2025).

While the idea of solving SAT in polynomial time is intriguing, it is important to note that some polynomial time solutions can still take an unreasonable runtime. These are

known as galactic algorithms (Lipton and Regan, 2013). For example, an algorithm with a time complexity of $O(n^{100})$ is technically polynomial, but in practice, its runtime is impractical. If $n = 100$ and a supercomputer capable of performing one trillion operations per second was used, even with one billion supercomputers working in parallel, they would only complete 10^{21} operations per second. At that rate, it would still take around 10^{179} seconds, which is approximately 3.17×10^{171} years, to finish 10^{200} operations. Therefore, even if P = NP, the real-world impact may not be as transformative as some suggest. However, with the rise of quantum computing, the possibility of solving these long-standing problems may gradually become more realistic. The debate over whether P = NP is an ongoing discussion that continues to attract attention today.

2.2. The History of SAT Solvers

2.2.1. The DPLL Era

One of the first major breakthroughs in SAT solving came in 1960 with the introduction of the Davis-Putnam algorithm by Martin Davis and Hilary Putnam. This algorithm was refined in 1961 by Martin Davis, George Logemann and Donald Loveland, resulting in the Davis–Putnam–Logemann-Loveland (DPLL) algorithm we know today (Davis et al., 1962). Unlike brute-force techniques, DPLL uses three pruning methods: backtracking, unit propagation, and pure literal elimination, which will be elaborated upon later, to systematically navigate the solution space while applying formula simplification where applicable. While DPLL was an important step forward, it still has its limitations: it does not learn from conflicts or mistakes, meaning it can revisit failing branches that are guaranteed not to satisfy the formula (Biere et al., 2021). Despite this issue, the DPLL algorithm remained a staple for three decades.

2.2.2. The CDCL Revolution

Clause learning was then proposed and introduced by Marques Silva and Karem Sakallah in their 1996 paper ‘GRASP-A New Search Algorithm for Satisfiability’ (Marques Silva and Sakallah, 1996) and Bayardo and Schrag in their 1997 paper ‘Using CSP Look-Back Techniques to Solve Real-World SAT Instances’ (Bayardo Jr and Schrag, 1997). Conflict-driven clause learning (CDCL) extends the DPLL algorithm by incorporating conflict-driven clause learning, which prevents the solver from making the same mis-

takes repeatedly. When the solver encounters a conflict, it learns a new clause using a decision heuristic and then employs non-chronological backjumping. This allows the solver to skip unnecessary searches, reducing runtime and improving scalability (Marques Silva and Sakallah, 1996). We will explore this process in further detail later in this paper.

2.2.3. Advanced Optimisations and Parallelism

In the last two decades, there have been multiple attempts to further optimise clause learning. One optimisation technique that is a staple today is SAT parallelism. Since 2002, the SAT competition has been identifying some of the fastest SAT solvers in the world for specific use cases (Franco and van Maaren, 2024). Originally, there was only one competition to apply for, the main track, which focused on optimising solvers for the use of a single CPU core. However, since 2013, there have been two additional competitions: the parallel track and the cloud track. The parallel track uses multiple cores and threads to explore different parts of the search space simultaneously, while the cloud track uses multiple machines with a general divide and conquer approach to the SAT problem to find a solution (Franco and van Maaren, 2024). These two optimisation techniques are widely used today to provide more computational resources to a SAT solver, enabling it to solve problems with faster runtimes.

Finally, further refinements to the CDCL algorithm have been implemented over the last two decades to fine-tune the algorithm. Key improvements include enhanced heuristics such as VSIDS, dynamic restarts, and non-chronological restarts to optimise search efficiency (Liang et al., 2015). Advances in clause learning, including clause database management and the use of clause blocking play a key role in reducing memory usage Haim and Walsh (2021).

3. Methodology

3.1. Work Process Overview

This paper compares and examines seven SAT solvers:

- Brute force (BF) solver
- Brute force with early stopping (BFES) solver
- Unit propagation and brute force (UPBFES) solver
- Pure literal elimination and brute force (PLEBFES) solver
- Unit propagation, pure literal elimination and brute force (UPPLEBFES) solver
- DPLL solver
- CDCL solver

All solvers are specifically designed to tackle formulae in conjunctive normal form (CNF), which is the most widely adopted Boolean algebra representation for SAT solvers (Biere et al., 2021). Furthermore, each solver guarantees absolute completeness and correctness.

In this section, we will explore the algorithmic workings of each solver and the exact design followed by each. The process of gathering formulae to provide to the solvers will also be described.

3.2. CNF

3.2.1. What is CNF?

CNF is a format for writing Boolean formulae that is easy to process by algorithms, especially SAT solvers. A CNF formula is a conjunction (AND, denoted by $\wedge$) of clauses, where each clause is a disjunction (OR, denoted by $\vee$) of literals. Clauses themselves are denoted as opening and closing parentheses. A literal is a single Boolean variable (e.g., A) or its negation (e.g., $\neg A$) (Biere et al., 2021). For example, given the formula:

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

We know this is not in CNF because the first two clauses are joined by a disjunction, and the third clause, $(D \wedge \neg B)$, joins the two literals with a conjunction. The correct CNF representation of this Boolean formula would be:

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

.

3.2.2. CNF generator

The CNF generator is designed to produce a dataset of generated CNF formulae. This dataset can then be fed into each solver, allowing them to determine whether each formula is satisfiable or unsatisfiable. The CNF generator is designed with multiple adjustable parameters, enabling the creation of different datasets tailored to various use cases that a solver may need to handle. The adjustable parameters include:

- The number of formulae in the dataset.
- The minimum number of literals in a formula.
- The maximum number of literals in a formula.
- The probability of a literal in a formula being the negation of the literal (NOT).
- The probability that a gate connecting two literals represents disjunction rather than conjunction in a CNF formula (higher probability creates larger average clause sizes).

Once these parameters are defined, the generator creates the specified number of formulae. Each formula's size is chosen at random within the given minimum and maximum literal counts. The type of logical gate connecting literals is determined by the probability that the gate represents disjunction rather than conjunction; a higher probability leads to longer average clause sizes. Additionally, each literal could be negated based on the specified probability.

This automated approach is essential, as modern SAT solvers, such as DPLL and CDCL, are highly efficient at solving formulae. Manually constructing thousands of large formulae would be impractical and time consuming. Once all formulae are generated, they are returned in a dataset ordered by the number of literals in the formula.

3.3. Brute Force

3.3.1. BF Solver

The most basic solver implemented in this paper is the brute force approach. This solver iterates through every possible combination of truth values for a given formula. For each generated assignment, it substitutes the corresponding truth values into the formula and evaluates the outcome using standard Boolean operations. If the formula evaluates to true under a particular assignment, that assignment is recorded as a satisfiable solution. Conversely, if none of the combinations result in a true evaluation, the formula is deemed unsatisfiable. This is the only solver that provides every possible combination of truth value assignments that make a formula satisfiable.

3.3.2. BFES Solver

BFES is an enhanced version of the BF solver, designed to terminate as soon as it identifies the first set of truth assignments that satisfy a given formula. This approach returns only one solution if the formula is satisfiable. For smaller formulae, this method may not always be ideal. The first satisfying assignment found may not always be the most optimal solution. For instance, in a business context, there could be an alternative assignment that offers a more cost effective outcome. That being said, this paper does not focus on finding optimal solutions, as our formulae lack this contextual constraint. Without a defined objective function or cost metric, determining the 'best' solution is not feasible. The focus of this paper is more specifically on comparison in terms of runtime efficiency and computational resource usage.

3.4. DPLL Algorithm

The DPLL algorithm implements three main optimisation techniques: unit propagation, pure literal elimination, and backtracking. Three solvers have been developed, each incorporating at least one of these optimisation techniques, while the fourth solver is the complete DPLL algorithm that includes all three optimisation techniques together.

3.4.1. UPBFES Solver

This solver first applies unit propagation to unit propagate the formula before using brute force with early stopping to solve the remaining problem. Unit propagation is an algorithm that can potentially simplify a formula. This simplification reduces formula size, which can shrink the search space, in turn decreasing solver runtime. It operates as follows:

First, a unit clause must be identified. This is a clause consisting of a single literal (let's call this literal l). The literal can be in one of two polarities, e.g., A or $\neg A$. If a unit clause is identified, the following rules can be applied to the formula:

- Every clause that contains l with the same polarity is removed from the formula.
- Every clause that contains l with the opposite polarity must have l removed from the clause.

This process is then repeated until all unit clauses have had this process applied to them (Biere et al., 2021). For example, given the formula:

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

we can see that A is in a unit clause. For the whole formula to be SAT, we know that A must always be true, and so we can follow the rules stated above to simplify the formula:

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

The formula still contains a unit clause that has not been checked. The literal $\neg B$. Again, we know that for this formula to be SAT, $\neg B$ must always be false, so B in the third clause can never be true. Thus, we can remove that instance of the literal without affecting the formula's satisfiability, further simplifying the formula by following the established rules to obtain:

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

There are now no more unit clauses that have not already been processed, and so the algorithm stops.

In this case, unit propagation significantly simplifies the formula. However, other formulae may undergo only minimal simplification or none at all. Therefore, once unit

propagation has been applied, this solver then employs the brute force algorithm to solve the remainder of the formula.

3.4.2. PLEBFES Solver

This solver first applies pure literal elimination and then uses brute force with early stopping to solve the remaining problem. Pure literal elimination works by identifying a literal in the formula that, throughout the whole formula, has only one polarity (e.g., it appears only as A, not $\neg A$). This literal is then deemed pure. Since the opposite polarity is absent from the formula, assigning a value to the pure literal cannot contradict any clause. Once assigned, every clause containing the pure literal is immediately satisfied, and all those clauses can be removed from the formula. The process continues until no more pure literals remain (Biere et al., 2021). Like unit propagation, this helps simplify the formula and reduce the problem size. For example, given the formula:

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

The algorithm first identifies A as a pure literal, and so a value is assigned to satisfy the literal, in this case, true. All clauses that include A can then be removed:

$$(\neg B \vee \neg C \vee B) \wedge (D \vee E) \wedge (\neg C \vee E) \wedge (\neg D \vee \neg E) \quad \text{where} \quad A = \text{true}$$

Again, we can see that $\neg C$ is pure:

$$(D \vee E) \wedge (\neg D \vee \neg E) \quad \text{where} \quad A = \text{true}, C = \text{false}$$

There are no more pure literals, and so the algorithm stops.

Just like unit propagation, pure literal elimination does not usually solve the formula entirely, it typically simplifies it. When this happens, the brute force algorithm is employed to handle the remaining formula. It is important that the truth values assigned during pure literal elimination are remembered and included in the final satisfying solution to provide the complete answer.

3.4.3. UPPLEBFES Solver

This solver uses both unit propagation and pure literal elimination algorithms, applying them sequentially to maximise formula simplification. First, unit propagation is applied

to the formula, followed by pure literal elimination. If the application of pure literal elimination introduces new unit clauses, unit propagation is applied again. Conversely, if unit propagation results in new pure literals, pure literal elimination is reapplied. This process continues until neither algorithm can simplify the formula any further. The remaining formula is then solved using brute force with early stopping, just like the two solvers mentioned above.

3.4.4. DPLL Solver

The DPLL solver utilises both unit propagation and pure literal elimination, but rather than relying on brute force to simplify the formula, it employs a recursive backtracking algorithm to explore the entire search space. Initially, like the UPPLEBF solver, it simplifies the formula as much as possible. Once no further simplification can be made, the algorithm recursively assigns a truth value to a chosen literal in the formula, initially setting it to true. This assignment allows the formula to be further simplified through another round of unit propagation and potentially pure literal elimination.

This recursion continues, repeatedly assigning a literal a truth value and simplifying the formula further. Conceptualising this as a binary tree, each recursive call represents a move down one branch of the tree. The algorithm keeps trying to move down one branch of the tree until the formula is completely solved, or a conflict arises while trying to unit propagate or pure eliminate (caused by the forced truth value of the chosen literal). This scenario can be visualised as reaching a leaf node in the tree. If the formula at this point has been successfully simplified enough that the formula is solved, the recursion terminates. However, if a conflict is detected, the algorithm must recursively backtrack its way back up the branch by revisiting previous decisions (literal assignments) made. At each node in this backtracking process, the literal that was previously assigned true is reassigned false, and the algorithm re-enters the recursive process of simplification, exploring down a new branch of the tree.

This method of recursively exploring different branches continues until either one of the leaf nodes identifies the formula as solved or all possible branches have been exhaustively explored, in which case the formula is determined to be unsatisfiable.

The choice of which literal to assign a truth value to, and whether to assign it as true or false first, is known as a decision heuristic. While there is always an optimal literal to select and truth value to set, it varies from case to case, and no algorithm can guarantee the best choice every time. Instead, heuristics use ‘rules of thumb’ to make decisions that are optimal in most cases but may be slightly suboptimal in others (Kullmann, 1999). A common heuristic for DPLL is to select the literal that appears most frequently throughout the formula, as this choice may have the greatest potential to simplify multiple clauses.

3.5. CDCL Solver

CDCL is a similar process to DPLL, however, the backtracking process is approached differently to prevent the solver from pursuing paths that would never make the formula satisfiable (Schlenga, 2020) (Biere et al., 2021).

When the formula is first given to the CDCL solver, the decision level is set to 0. At every decision level, unit propagation is applied to the formula. After each unit propagation, the formula will either be simplified or a conflict within the formula will be identified.

If the formula is simplified, the decision level is increased by one and, like in DPLL, a literal from the formula is assigned a truth value. This literal is known as a decision literal / node. The solver achieves assigning the truth value to the literal by appending a unit clause containing that literal to the formula. This allows unit propagation to be applied again. Just like DPLL, this process continues until the formula is either simplified enough to be solved or a conflict is identified at the current level. If a conflict is identified, this is where CDCL differs from DPLL. CDCL applies a conflict analysis algorithm aimed at producing what is known as a learned clause (Biere et al., 2021).

3.5.1. What Is a Learned Clause?

A learned clause is a clause that, once identified, is added to the end of the formula. Once the learned clause is included, it essentially acts as a "don’t go here again" rule. It ensures that the solver will never repeat the same poor decision that led to a conflict.

For example, if we have the formula:

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

If CDCL identified the learned clause as:

$$(A)$$

Once added, the formula would become:

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

In this case, the learned clause, when applied to the formula, is effectively saying, "You cannot assign A as false again, or you'll inevitably end up in a conflict". The solver has now learned from the mistake it made at the conflict level and, unlike DPLL, will not make that same mistake again (Biere et al., 2021).

3.5.2. How Is the Learned Clause Found?

Once we have a formula containing a conflict, we apply conflict analysis. The first step in conflict analysis is to identify the first Unique Implication Point (1-UIP). However, before this, it is important to explain what a trail is in CDCL.

3.5.3. The Trail

In CDCL, during the entire process of solving a formula, a trail is maintained in the background. Put simply, a trail is a sequence of all the decisions made by the algorithm, including why they were made and the order in which they occurred. A visual representation of this trail can be seen in Figure 1. This visual representation is known as an implication graph, which helps us, as humans, understand the trail with greater ease. All the information shown in the implication graph is contained within the trail, which the algorithm uses internally. Therefore, in practice, the algorithm never needs to construct the implication graph itself.

The nodes in bold represent the decision literals. For example, in Figure 1, A was set to true. Decision literals are the reason conflicts arise, since the downstream implications of a decision literal cannot be fully known in advance. The non-bold nodes represent

implications that resulted from the impact of the decision literal. These implications are connected by arrows, or edges, to indicate which node caused another node's assignment. The number inside each node indicates the level at which the truth assignment occurred. For example, we can see that the literal $\neg D$ was assigned false due to C being assigned true, which occurred because of the decision literal A being assigned true.

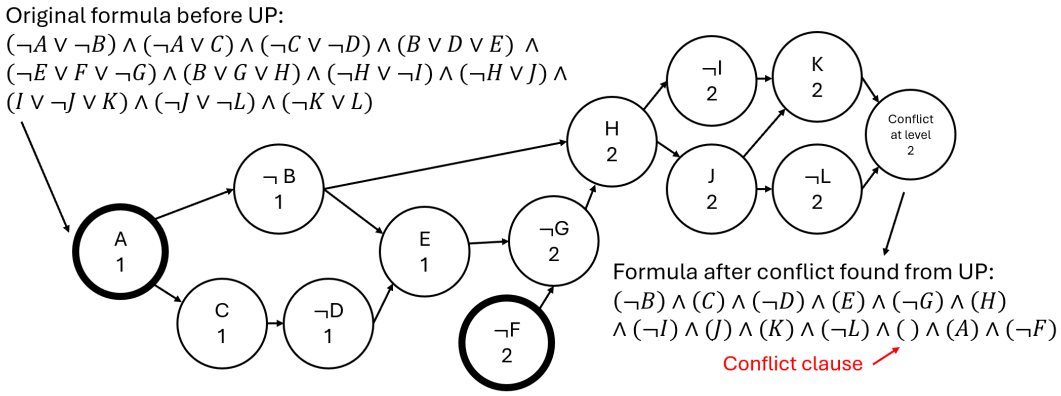

Figure 1: Visualisation of a trail, known as an implication graph, on an example formula.

3.5.4. The First Unique Implication Point

The first step in conflict analysis is finding the first unique implication point (1-UIP) from the trail. A unique implication point (UIP), in reference to an implication graph, is a single node through which all edges from the most recent decision node pass through. There can be multiple UIPs in some instances of trails. For example, in Figure 1 both $\neg G$ and H are UIPs. The 1-UIP is the UIP in the implication graph that is closest to the conflict. In the instance of Figure 1 it would be H. Identifying 1-UIP is a decision heuristic and the most common for CDCL (Schlenga, 2020).

We can see that in Figure 1, if the 1-UIP had not been assigned its particular truth value, the conflict would never have occurred, regardless of what other assignments were made at the same decision level. Because of this, we can now create a learned clause that will prevent this particular 1-UIP node from ever happening again. You could simply flip the polarity of the first UIP and use that as our learned clause. This would prevent the conflict from happening again, but only at that particular decision level. It does not take into account lower decision levels that could also cause the same conflict. This leads us

to the next stage of conflict analysis; identifying the appropriate learned clause.

3.5.5. Identifying the Learned Clause

Now that we have identified the node that is the 1-UIP, we can proceed to find the learned clause. To do this, we must follow the specific set of rules stated below. The learned clause must include the literals of:

- The negation of the 1-UIP's literal.
- The negation of any literal at a lower decision level that has a direct edge to a node that is past, or higher than, the 1-UIP.

Failure to include the second rule's literals in the learned clause may permit the same conflict to arise again. In Figure 1, it is evident that the second rule does not apply in this case, therefore the learned clause is $(\neg H)$ (Bayardo Jr and Schrag, 1997).

3.5.6. Backtracking

Once the learned clause is found, the algorithm must identify the lowest level literal that is in the learned clause and then backtrack to that level. So, for Figure 1, you would backtrack to level two. Backtracking means reverting all formula and trail progress made up until that level. Once the algorithm has backtracked, we add the learned clause to the end of the formula and continue the process of unit propagation from that level. In subsequent runs, this learned clause ensures that the same conflict cannot occur again. As a result, unit propagation will yield different results than before, effectively exploring more of the search space.

Finally, we know that the formula is not satisfiable if a conflict is identified at decision level 0, as we can no longer backtrack. This means the conflict cannot be resolved by changing any prior decisions. Therefore, the conflict is inherent to the formula itself, and no possible assignment of variables can satisfy all clauses. As a result, the solver can conclusively declare the formula unsatisfiable (Marques Silva and Sakallah, 1996).

4. Implementation

Having outlined the solvers and algorithms used in this paper, we will now examine how they were implemented in terms of programming logic, data structures, and overall design. We begin with the CNF generator.

4.1. CNF Generator

The CNF generator was implemented in C++ as one stand-alone script. Each execution of the script produced a dataset stored as a text file, with each line corresponding to a single formula.

Within these formulae, conjunctions were denoted by the caret symbol (‘^’), and disjunctions by the letter ‘v’. Each formula could contain up to 25 unique literals. These were selected from the letters ‘a’ through ‘z’, excluding ‘v’, which was reserved for representing disjunctions. Although this provided a limited number of unique literals, the literals could appear multiple times within a single formula. Negation of a literal was indicated by a hyphen (‘-’) preceding the literal. Clauses were enclosed in parentheses, with ‘(’ marking the beginning and ‘)’ marking the end of a clause. At the end of each formula, an exclamation mark (‘!’) was used as a delimiter, followed by the total count of literals present in the formula, regardless of repeating literals. For example, one line in the dataset might appear as follows:

```
(a v -g) ^ (a v c v -b) ^ (e v -c) ^ (a v -f) ^ (d)!10
```

Each formula was generated using the exact process and parameters outlined in the methodology section of this paper.

4.2. Solvers To CSV

The CNF dataset generated by the C++ script could then be used to feed into a Java project designed to evaluate each formula using the seven different SAT solvers discussed in the methodology, all implemented within the same project. For each solver, the system captures the key metrics: execution time, memory usage, satisfiability result and where applicable, the truth value assignment. These results were then recorded into a CSV file to be used for later analysis.

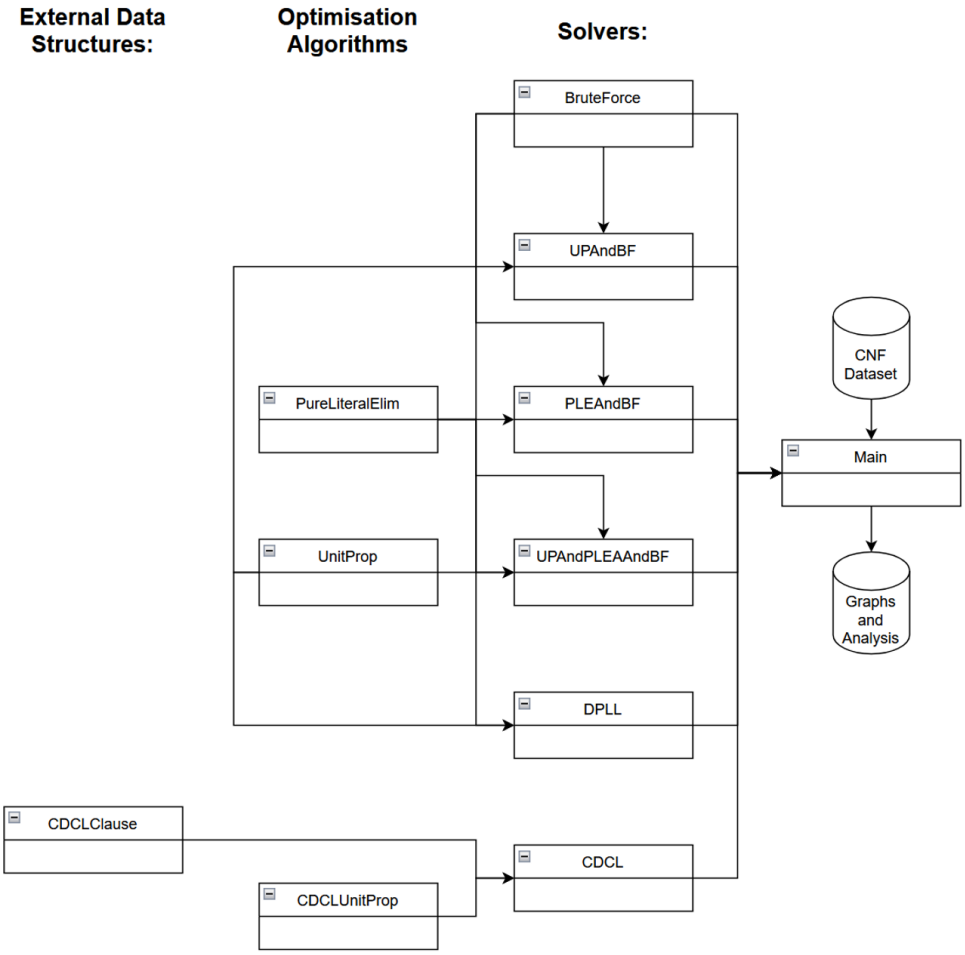

Figure 2: The project structure of the Java project.

4.2.1. Two Dimensional Array

The overall structure of the project is shown in Figure 2. The dataset is first read into the main class, where each formula is then processed using a function that transforms each formula into a 2D array, with the outer list representing the full formula and each inner list representing a clause. Within each clause, literals are stored in the order they appear, maintaining positional consistency.

Literals are still stored as characters, however, negated literals are represented by up-

percase letters. For example, the literal ¬g is stored as 'G', while g is stored as 'g'. Conjunctions and disjunctions are not explicitly stored in the data structure, as they are implicitly defined by the CNF format; literals within a clause are disjoined, and clauses are conjoined. The integer following the exclamation mark (!) indicating the total count of literals in the formula is not included in the formula structure but is extracted and saved in the same CSV file for use in performance analysis. Using the same example formula from the CNF generator:

```
(a v -g) ^ (a v c v -b) ^ (e v -c) ^ (a v -f) ^ (d)!10
```

The resulting two-dimensional array list would be:

```
[[a, G], [a, c, B], [e, C], [a, F], [d]]
```

We now turn to an explanation of how each solver was implemented within the Java project. As shown in Figure 2, each solver was implemented as its own class and was invoked into the main class.

4.2.2. BF and BFES Solver

The brute force implementation goes through all 2^n possible truth-value assignments for n distinct literals by looping an integer index i from 0 to $2^n - 1$. Each unique literal is assigned a bit position in the binary representation of i. At each iteration, we convert i into its binary representation and use that same bit position to set each unique literal to either true or false. The literal and value pairs are stored in a hash map keyed by literal names. For each iteration, we then evaluate the formula by substituting in the truth assignments for the literals and applying Java's logical operators. Whenever the formula evaluates to true, we add the current hash map to a hash set. After all iterations have run, the solver will return a hash set of hash maps with every combination of truth value assignments that make the formula satisfiable. BFES is the exact same process, however when one satisfying assignment is identified, the loop breaks early, and so only a single hash map needs to be returned. Table 1 illustrates how this works for the following formula:

```
[[a, G], [a, c, B], [e, C], [a, F], [d]]
```

i	Binary	A	G	C	B	E	F	D
0	0000000	F	F	F	F	F	F	F
1	0000001	F	F	F	F	F	F	T
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
127	1111111	T	T	T	T	T	T	T
$i_{\max} = 2^n - 1 = 127$								

Table 1: Mapping the 7-bit index i to truth values of the unique literals A, B, C, D, E, F, G.

4.2.3. Unit Propagation

Unit propagation is implemented as its own class that each solver invokes when unit propagation is required. The algorithm begins with an outer loop that scans each array (clause) inside of the 2D array. When it finds an array containing exactly one element (a unit clause) that has not yet been processed, it applies the propagation rules stated in the methodology. To do this, it enters an inner loop that revisits every inner array again. If one of the arrays contains the same element as the unit clause with the same polarity, that entire inner array is removed from the outer array because it is already satisfied. If an inner array contains the opposite polarity element to the unit clause, only that element is removed from the inner array. The outer and inner loops repeat until no unprocessed unit clauses remain.

Polarity checks are handled by a global function in a utility class that is commonly used throughout the whole project. The function takes two characters as arguments and returns true if both are uppercase or both are lowercase, and returns false if one is uppercase while the other is lowercase.

4.2.4. UPBF Solver

This solver already has all the necessary components implemented; they are simply integrated together. First, the unit propagation method is invoked inside this solver class. The 2D array of clauses is passed through unit propagation, and the resulting simplified 2D array is returned. Then, the BFES solver is called within the same class to compute the final truth value assignments for the formula.

4.2.5. Pure Literal Elimination

Pure literal elimination is also its own class like unit propagation that is invoked by other solvers when required. It works in three main stages. First, a hash set is created to store all unique elements found in the 2D array. This involves scanning through each inner array and adding each element to the set. Next, the algorithm checks the polarity of each literal in the hash set. For each element in the hash set, it loops through all inner array elements to determine if its opposite polarity element also exists. If it is found, the literal is removed from the set. Once this step is finished, only pure literals remain in the set. Finally, the algorithm loops through the 2D array again and removes any inner array that contains a pure literal from the set. The method returns both the updated 2D array and the hash set of pure literals, which is needed to track which literals have been completely removed from the formula.

4.2.6. PLEBF Solver

PLEBF is similar to the UPBF solver in the sense that all components are already implemented; they just need to be put together. Inside this new solver class, pure literal elimination is first invoked, and the 2D array is passed through it. This returns a new 2D array and a hash set containing pure literals. The BFES solver is then called, and the simplified formula is passed through it to obtain the final truth value assignments. The pure literals from the hash set are also assigned their corresponding truth values, and these are combined with the assignments from the brute force solver to produce the final result.

4.2.7. UPPLEBF Solver

This solver again invokes both unit propagation and pure literal elimination classes, applying them sequentially to the formula. If pure literal elimination modifies the formula, as indicated by a Boolean flag within the class, another round of unit propagation and pure literal elimination is invoked. This cycle continues until no further modifications are made to the formula. Each time pure literal elimination is invoked, it returns a hash set containing the pure literals identified in that round. These hash sets are combined and accumulated across successive rounds. Once the formula is fully simplified, it is passed to the brute force solver to compute the remaining truth value assignments. The

final result is then produced by combining these assignments with the accumulated pure literal assignments.

4.2.8. DPLL

The DPLL solver is implemented using a recursive function, as described in the methodology. Each call takes the 2D array and potentially returns true or false: true if the formula is satisfied (the 2D array has no inner arrays remaining), or false if it is unsatisfiable (an empty inner array is found). If it does not return true or false, it continues recursing. A global hash map is used to store the literals and their truth value assignments. When the recursive function returns true, a hash map with the appropriate truth value assignments will be returned by the class.

At the start of each call, unit propagation is invoked and applied followed by pure literal elimination. After these simplifications, we check for a satisfied or unsatisfied state. If neither apply, we proceed with a branching decision. To choose a branching literal, we call a function in the same class that scans through the formulas and uses a hash map to count occurrences of each distinct literal. It then returns the literal with the highest count.

This literal is used to create two new unit clauses, one for each polarity of the literal. Two instances of the 2D array are made with both unit clauses being appended to one of the instances. We then recurse on each of these two modified formulae to explore both branches of the binary decision tree described in the methodology. This recursive process continues until the formula either evaluates to true in one of the recursive calls, or is determined to be unsatisfiable when all recursive branches return false.

4.2.9. CDCL Solver

Unlike the other solvers, CDCL's implementation does not use a 2D array. Instead, each clause is an instance of a Java class made called CDCLClause. This means CDCL is using an array of CDCLClause objects instead.

Each CDCLClause encapsulates three internal data structures:

- One array to hold the clause.

- One array to hold the trail.
- An integer to store the clause level.

A new class has also been made called 'CDCLUnitProp' which refactors the unit propagation class to support the new data structure. The logic for the class behaves exactly the same, however when a CDCLClause contains a unit clause literal with opposite polarity, the literal is removed from the clause array and moved to the trail. This allows changes made to the clauses to be recorded which is needed to find the learned clause.

The CDCL implementation mirrors the methodology exactly. Starting at decision level 0, we use a stack data structure to store the formula at each level. After each loop of unit propagation, one of three outcomes occur:

- **Conflict:** A conflict is detected if unit propagation produces an empty CDCLClause (which shouldn't happen if the formula is solvable).
- **Satisfiable:** The formula is solved when there are only CDCLClause unit clauses left.
- **Undetermined:** The formula has no conflicts and still contains CDCLClauses that are not unit clauses, which will require further decisions and propagation.

In the case that the formula is undetermined, this implementation uses the 'first non-unit literal' heuristic. It scans the formula for the first literal that isn't a unit clause and appends it as a new unit clause at the end of the formula. Once this clause is added, the decision level is incremented, the formula is saved to the stack, and another round of unit propagation is triggered via a while loop that continues running as long as the decision level remains non-zero.

In the case that the formula is unsatisfiable, the decision level is checked. As stated in the methodology, a conflict at level 0 of the formula means it can never be satisfied, so the program terminates without returning any result. If the conflict occurs at a higher level, the code invokes a function used for conflict analysis (finding the learned clause). The first step in this function is to call another function that identifies the 1-UIP. This functions works by locating the CDCLClause whose clause array is empty (the conflict clause) and adding all of the elements from the trail array of that same CDCL clause

into a hash set. For each element in that set, the unit clause of that element is located in the formula and that CDCLClause's trail is also added to the set. This process continues until only one literal remains in the set, at which point the 1-UIP has been found.

After identifying the 1-UIP, the next step is to construct the learned clause. It uses a similar approach to before by adding elements to a hash set, but this time by traversing up the trail rather than down. The code finds every literal that is directly connected to another literal located higher in the trail than the 1-UIP. For such literals, its polarity is flipped, and the result is added to a new CDCLClause along with the flipped polarity of the 1-UIP's literal, which collectively forms the learned clause.

Lastly, the level of each literal in the learned clause was also tracked in the function. After identifying the lowest level, the code unwinds the stack by popping formulae off the stack until we reach the formula corresponding to the required lowest level. At that point, the learned clause is appended to the formula. Execution then proceeds as normal, and eventually, due to the structure of the while loop, the formula will either be satisfied, or determined to be unsatisfied at level 0, thus truly unsatisfiable.

4.3. Solvers To Plotting

Once a dataset of formulae is solved using the seven SAT solvers and the CSV file was generated by the Java project, it was passed to a small Python script. This script used pythons 'plotly' library to generate various plots, selecting different columns from the CSV for the x and y axes depending on whether runtime analysis or memory analysis was required. The results of these benchmark plottings are discussed in the following section.

5. Results and Evaluation

The results presented in this section were generated using the following CNF generator parameter values:

- Number of formulae: 500,000
- Minimum number of literals in a formula: 1
- Maximum number of literals in a formula: 500,000
- Probability of a literal in a formula being the negation: 50%
- Probability that a gate connecting two literals represents disjunction rather than conjunction (higher probability creates larger average clause sizes): 50%

5.1. Worst Performing Solvers

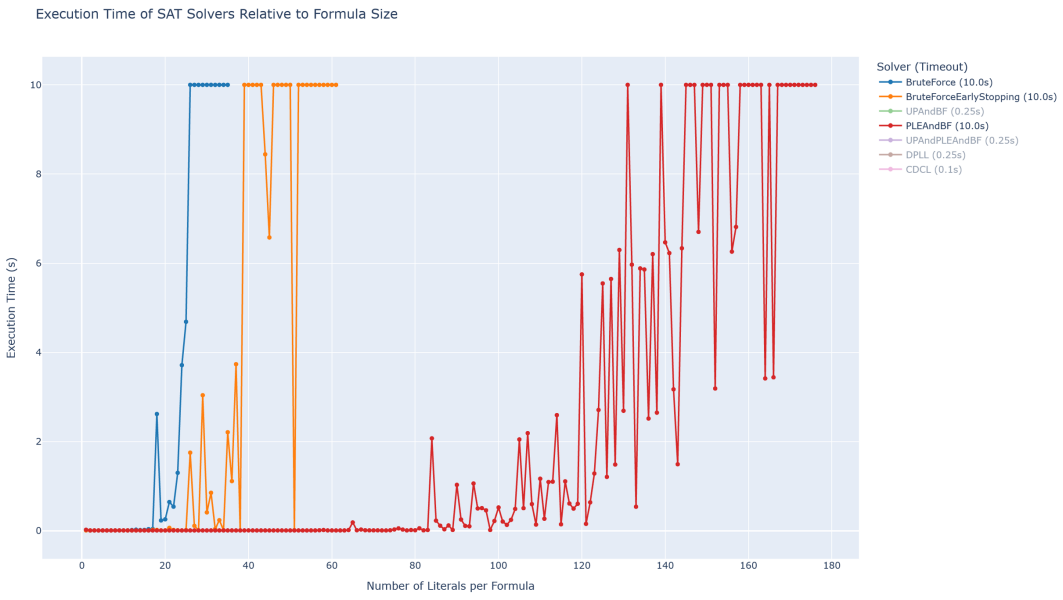

Figure 3: Execution time of the BF, BFES and PLEBF solvers relative to formula size.

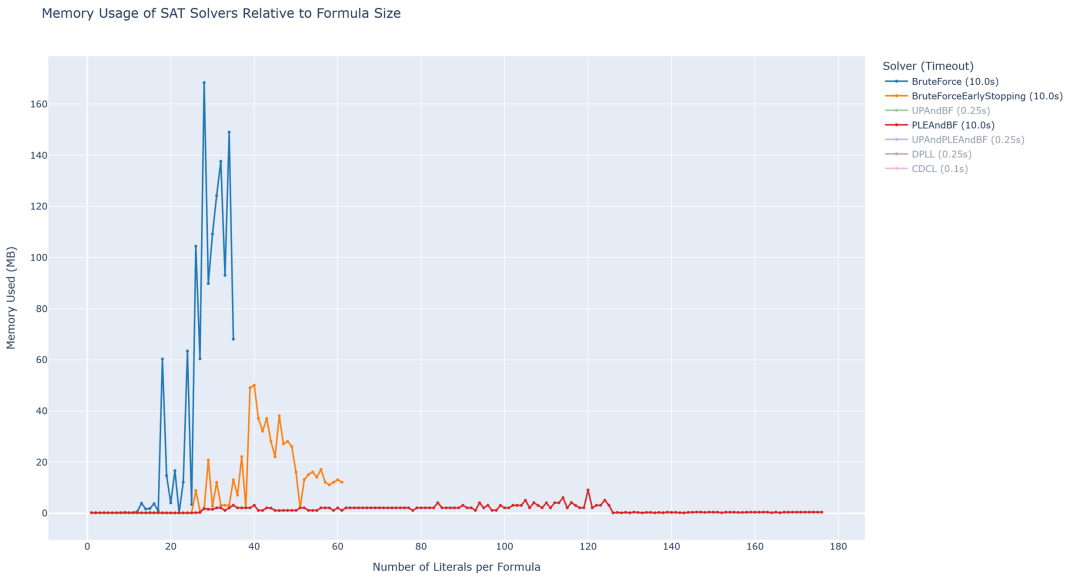

Figure 4: Memory usage of the BF, BFES and PLEBF solvers relative to formula size.

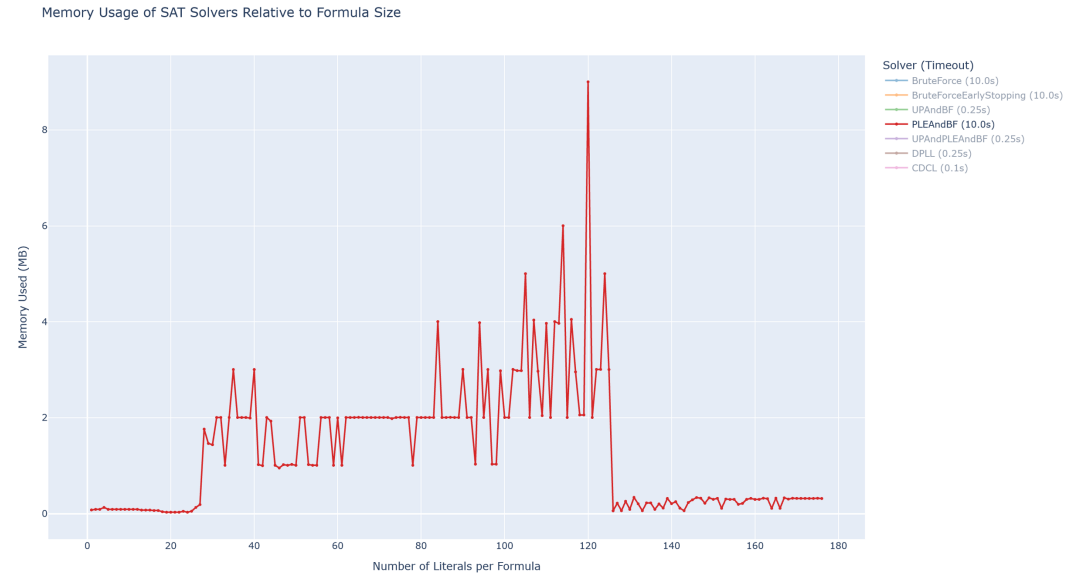

Figure 5: Memory usage of the PLEBF solver relative to formula size.

5.1.1. BF and BFES Solvers

We begin by evaluating the three poorest performing solvers: BF, BFES and PLEBF. These solvers were each given a 10 second timeout. The timeout meant that, if a solver runtime took longer than the timeout value for a specific formula, solving ended and the formula was deemed as unsolved. Additionally, in all cases examined in this section, if a solver timed out 10 times consecutively, it ceased attempting to solve the remaining, more challenging formulas. This helped speed up benchmarking time. This increased timeout limit compared to the other solvers as seen in Figure 3 was necessary as their performance was so poor that shorter timeouts yielded few or no results.

The BF solver exhibited the lowest scalability, managing to solve formulae with up to approximately 21 literals before hitting the timeout consecutively. It also had the highest memory usage, peaking at around 160 megabytes (MB). The BFES solver performed modestly better, solving formulae with up to around 40 literals within the timeout. This improvement is not due to increased efficiency in computation, but rather because the early stopping approach terminates after finding a single solution, while the BF solver continues until all solutions are found. Although both solvers reach the first answer at the exact same time, the BF solver continues processing all successive answers and thus takes longer to return an answer (as all answers are returned at once). We can also notice the memory usage is significantly reduced in the BFES variant, as it only needs to work with a single hash map, compared to the hash set of hash maps required by the BF approach.

5.1.2. PLEBF Solver

The significantly poorer performance of PLEBF compared to all other solvers, except BF and BFES, is particularly notable, with PLEBF reaching a timeout at around 140 literals. Relative to DPLL, this underperformance can be attributed to the limited use of pure literal elimination, which is applied only once in PLEBF. In contrast, DPLL uses branching to apply pure literal elimination repeatedly throughout the search process. One could argue that this repeated application significantly enhances the effectiveness of pure literal elimination. However, it is unclear how much of DPLL's performance improvement is due to unit propagation rather than pure literal elimination. Given how poorly PLEBF performed, it seems likely that pure literal elimination did not play a

major role in DPLL's overall success either.

We know that in UPBF, unit propagation is also applied only once to the formula. However, the performance difference between this solver and PLEBF is striking (identified through comparison of Figure 3 and 6). This indicates that unit clauses are far more likely to appear in formulae than pure literals. Such a disparity would likely become even more pronounced as the probability of conjunction to disjunction decreases. This is because it would then be even more likely to find unit clauses.

The reason for PLEBF's poor performance is that, as the formula size increases, the likelihood of encountering pure literals decreases, particularly under the constraint of a maximum of twenty five unique literals. This limitation becomes a significant bottleneck; in larger formulae restricted to the same twenty five set of unique literals, it becomes increasingly unlikely to find any that are pure. If the solvers were adapted to allow the use of more than 25 unique literals, performance would likely improve for this solver. The bad performance could also explain why CDCL solvers do not implement pure literal elimination. The potential performance gains are so minimal that they do not justify the additional complexity or overhead, making the technique largely irrelevant in modern SAT solving. One could argue however that this is simply due to the bias introduced by limiting the number of unique literals to twenty-five and in reality, it depends on the specific formula at hand.

5.2. Top and Mid Performing Solvers

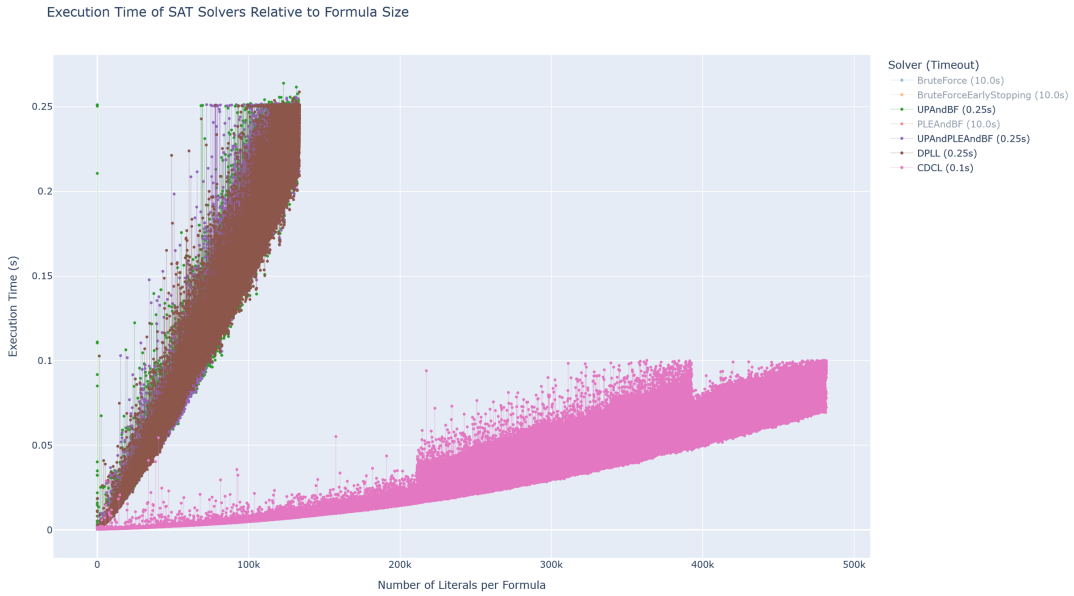

Figure 6: Execution time of the UPBF, UPPLEBF, DPLL and CDCL solvers relative to formula size.

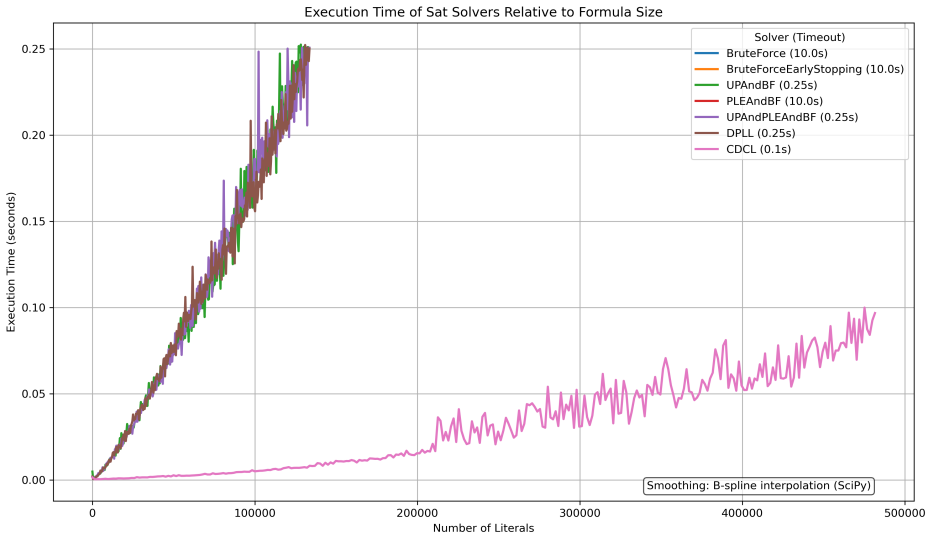

Figure 7: A refined version of the graph in Figure 6, incorporating B-spline interpolation.

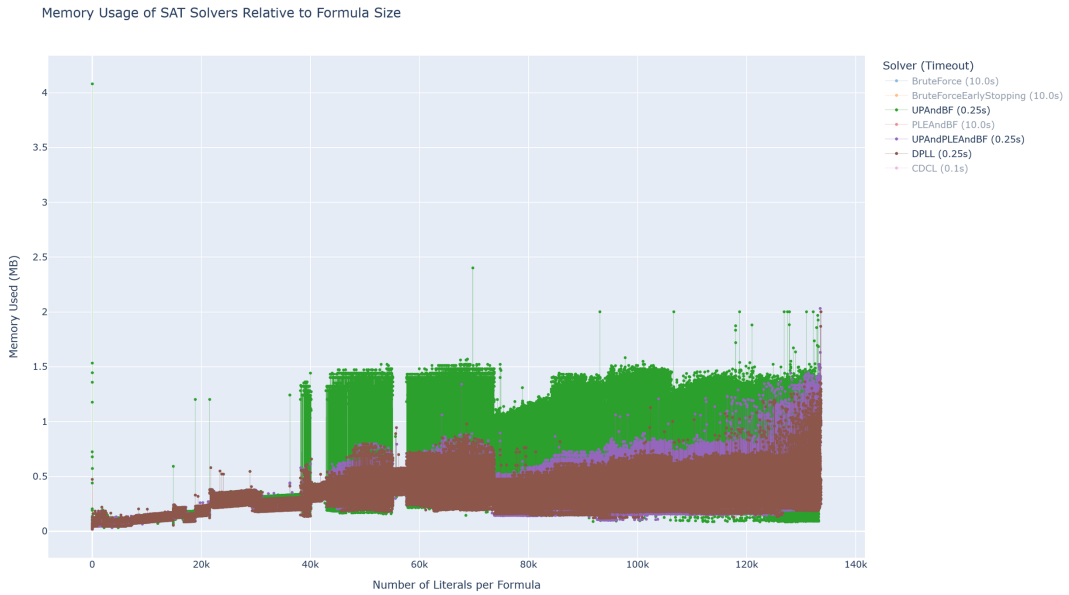

Figure 8: Memory usage of the UPBF, UPPLEBF and DPLL solvers relative to formula size.

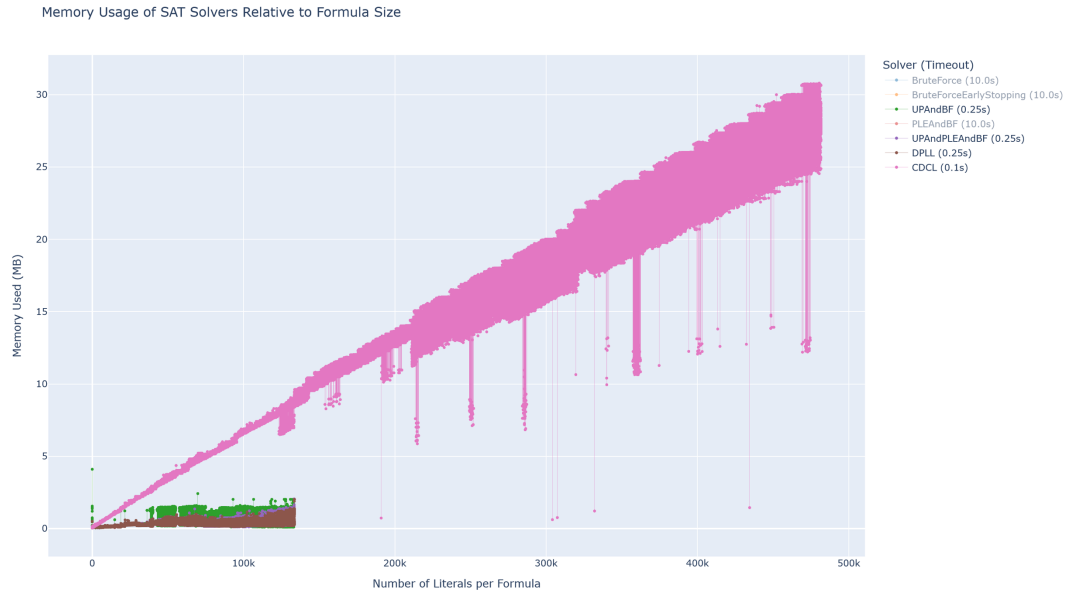

Figure 9: Memory usage of the UPBF, UPPLEBF, DPLL and CDCL solvers relative to formula size.

5.2.1. UPBF, UPPLEBF and DPLL Solvers

These three solvers demonstrated substantial runtime and memory gains compared to the previous three solvers (Figure 6). All three reached the 250 ms timeout threshold at a literal size of approximately 140,000, which is significantly better than the earlier solvers that peaked at around 160. Their performance was so strong that setting a 10 second timeout was not feasible, as running the experiments became excessively time consuming.

Interestingly, all three solvers demonstrated nearly identical runtime performance. Even in Figure 7, there is no noticeable out-performer. For UPBF and UPPLEBF, Figure 6 supports our earlier observation that pure literal elimination does not yield significant performance improvements. This confirms the suspicion that pure literal elimination did not provide significant performance gains for either UPPLEBF or DPLL. Regarding DPLL, its performance was unexpectedly similar to the other solvers. While there are minor reductions in runtime for certain formula instances, the average runtime gain is negligible. This indicates that the effectiveness of unit propagation is largely concentrated in its initial application. Recursive or repeated application of unit propagation appears to offer diminishing returns, likely because the initial application of unit propagation simplifies the formulae to such an extent that subsequent applications have minimal additional effect. This further underscores the strength and efficiency of unit propagation in SAT solving.

In terms of memory usage, DPLL consistently demonstrated the most efficient performance among the three solvers. UPBF exhibited the highest memory consumption, likely because the BF solver requires maintaining a large number of truth assignments in memory during the truth value assignment process. In contrast, DPLL stores truth values directly within the formula itself, using unit clauses to encode variable assignments. This approach reduces the need to explicitly retain separate assignments, resulting in lower overall memory usage. UPPLEBF showed a significant decrease in memory usage compared to UPBF, as illustrated in Figure 8. This indicates that pure literal elimination is indeed being applied and used in the formulas. The assumption is that, because unit propagation significantly simplifies a formula, it becomes much easier to identify pure literals afterwards. As a result, pure literal elimination allows far less brute-force

solving to be required once the formula is in its most reduced form, hence, considerably less memory to be used. Although pure literal elimination may not reduce runtime compared to a brute-force approach, it clearly has a significant impact on memory usage. In retrospect, its value may have been overlooked too quickly.

5.2.2. CDCL Solver

As shown in Figure 6 and 7, CDCL significantly outperformed all other solvers in terms of runtime. Its efficiency in handling large formulae was so pronounced that the timeout had to be reduced to 100 ms. Even with this stricter limit, CDCL was able to solve formulae with up to 480,000 literals before reaching the timeout. In contrast, the other three solvers would peak at approximately 80,000 literals if under the same 100 ms constraint. This demonstrates that CDCL could handle formulae approximately six times larger within the same time frame, which is a remarkable performance difference. This runtime performance advantage reflects the strength of clause learning in eliminating unnecessary search paths, confirming CDCL's status as the most effective SAT solver approach currently in use.

As shown in Figure 9, CDCL exhibited significantly higher memory usage compared to the other three solvers in the same figure. At a formula size of 100,000 literals, the other solvers used no more than approximately 2MB of memory, whereas CDCL consumed around 6MB. This disparity in memory usage becomes even more pronounced as the formula size increases. The higher memory consumption is likely due to the additional overhead of storing multiple instances of the formula on a stack, as well as the CDCLClause having to store a trail array and a clause array itself. While this results in greater memory usage, the substantial runtime advantage of CDCL more than compensates for this memory underperformance. As well as this, based on the results observed, as formula sizes increase, runtime becomes the primary bottleneck. In the context of Figure 9, even at 480,000 literals per formula, a memory usage of approximately 30 MB is relatively modest by today's standards and is well justified by the resulting improvements in solving time.

5.3. Parameter Adjustment

Benchmarks with varying average clause size and not probability were also retained, as seen in Appendix A, B, C and D. We can see that increased clause sizes on average caused runtimes to decrease for all solvers. This is likely due to the fact that, when unit propagation can remove a whole clause in a formula, having a much larger clause size means a larger chunk of the formula is removed and simplified. Clearly, finding unit clauses was still no problem even as the clause sizes increased. In terms of memory however, larger clause sizes also increased the amount of memory used. One reason for this could be, with fewer but much larger arrays, the underlying array sizes are bigger. Since these were implemented as ArrayLists in Java, more space may have been reserved than actually used compared to lower clause sizes, leading to higher memory consumption due to unused but allocated capacity. Variations in the not probability did not have a substantial enough effect to produce significant changes in either the memory or runtime graphs.

6. Conclusion and Future Work

In conclusion, on randomly generated formulae with at most 25 distinct literals, unit propagation provided the most notable runtime performance benefits, allowing UPBF, UPPLEBF, and DPLL to perform as mid-tier solvers in terms of runtime. When unit propagation was also combined with clause learning and conflict analysis, the benefits of learning from mistakes meant that the CDCL solver significantly outperformed all other solvers in terms of runtime, making it the highest performing solver out of the seven. In terms of memory however, CDCL exhibited higher consumption over the mid-tier solvers. It would therefore be interesting in future work to see another solver that combines the memory efficiency of pure literal elimination with the runtime performance of CDCL. Additional memory performance algorithms could also be looked at, for example, literal block distance (LBD)-based clause database reduction (Haim and Walsh, 2021) retains only the most valuable learned clauses and discards the rest. This approach reduces memory usage while aiming to minimise any negative impact on runtime by removing only the least impactful clauses.

In terms of runtime improvements, the CDCL decision heuristic used was a basic implementation. It would have been interesting to implement one of the more notable decision heuristics, such as VSIDS (Liang et al., 2015), to evaluate how much better the runtime and memory usage could have performed. Additionally, minor improvements to unit propagation could be explored in the future, such as the two-watched literals optimisation (Haim and Walsh, 2021), which only checks clauses when one of the two watched literals has changed. This technique offers significant runtime benefits while maintaining relatively low implementation complexity.

Finally, it would have been more valuable to give pure literal elimination a fairer opportunity to demonstrate its effectiveness by allowing a much larger number of unique literals within each formula as runtimes may have observed significant improvements. The power and potential of the algorithm can already be seen in how efficient it is in terms of memory usage. Other potential sources of bias may have been introduced by not having a sufficiently controlled environment when running the benchmarks. For instance, background tasks may have been active due to the length of time required to complete the benchmarks, which could have slightly affected some data points. More accurate data, particularly in tracking memory usage, might also have been achieved by using a library with a higher level of precision, enabling more stable and reliable graphical results.

References

Bayardo Jr, R. J. and Schrag, R. C. (1997). Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI)*, pages 203–208. AAAI Press.

Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2021). *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. IOS Press.

Claessen, K., Een, N., Sheeran, M., and Sorensson, N. (2008). Sat-solving in practice. In *2008 9th International Workshop on Discrete Event Systems*, pages 61–67.

Cook, S. A. (1971). 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. Association for Computing Machinery.

Cook, S. A. (2000). The p versus np problem. *Clay Mathematics Institute*.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397.

Franco, J. and van Maaren, H. (2024). SAT Competition. *https://satcompetition.github.io/*.

Goldreich, O. (2010). *P, NP, and NP-Completeness: The Basics of Computational Complexity*. Cambridge University Press.

Haim, S. and Walsh, T. (2021). *CDCL SAT Solver Heuristics: Clause Management, Instance Structure, and Restarts*. PhD thesis, Simon Fraser University.

Kullmann, O. (1999). Heuristics for sat algorithms: Searching for some foundations. Technical report, Unknown. Accessed via CiteSeerX.

Liang, J. H., Ganesh, V., Zulkoski, E., Zaman, A., and Czarnecki, K. (2015). Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In Biere, A., Nahir, A., and Vos, T., editors, *Hardware and Software: Verification and Testing*.

Lipton, R. J. and Regan, K. W. (2013). *David Johnson: Galactic Algorithms*, pages 109–112. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lozin, V. (2021). *Cliques, colouring and satisfiability: from structure to algorithms*, page 105–129. Encyclopedia of Mathematics and its Applications. Cambridge University Press.

Manthey, N. and Saptawijaya, A. (2016). Towards improving the resource usage of sat solvers. In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning (PAAR)*.

Marques Silva, J. and Sakallah, K. (1996). Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227.

Schlenga, A. T. (2020). Conflict driven clause learning. *Technical Report, Technical University of Munich*.

Vega, F. (2025). Sat in polynomial time: A proof of p = np. *HAL Open Science*.

Appendices

The dataset parameters used for the following benchmarks are outlined below:

- Number of formulae: 50,000
- Minimum number of literals in a formula: 1
- Maximum number of literals in a formula: 50,000
- Probability of a literal in a formula being the negation: 50%
- Probability that a gate connecting two literals represents disjunction rather than conjunction (higher probability creates larger average clause sizes): 50%

In cases where some dataset parameters have been modified, all such adjustments are stated in the corresponding figure caption.

Appendix A 25% Average Clause Size Benchmarks

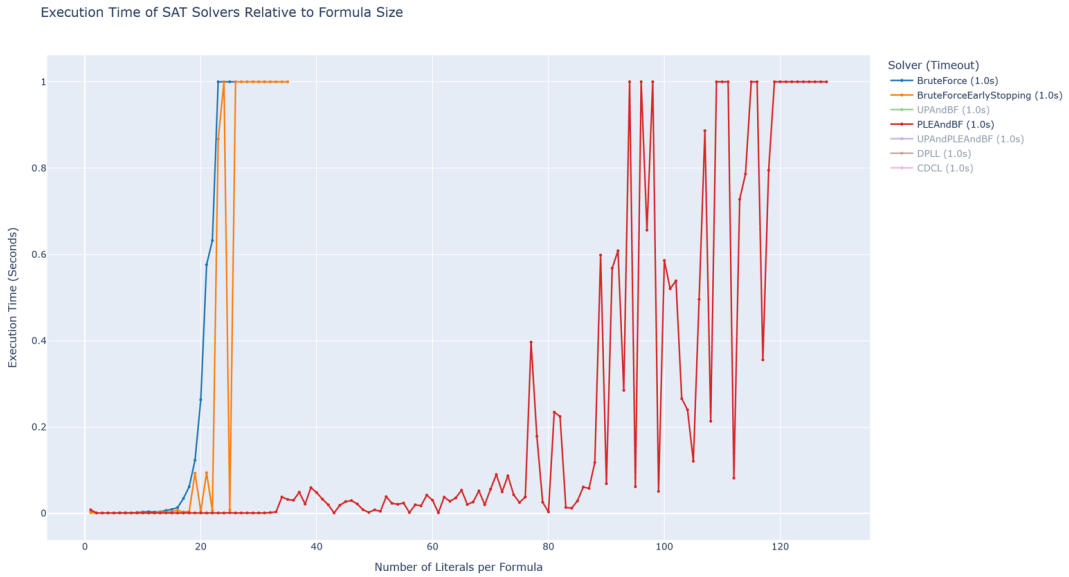

Figure 10: Execution time of the BF, BFES and PLEBF solvers relative to formula size with a 25% disjunction to conjunction chance.

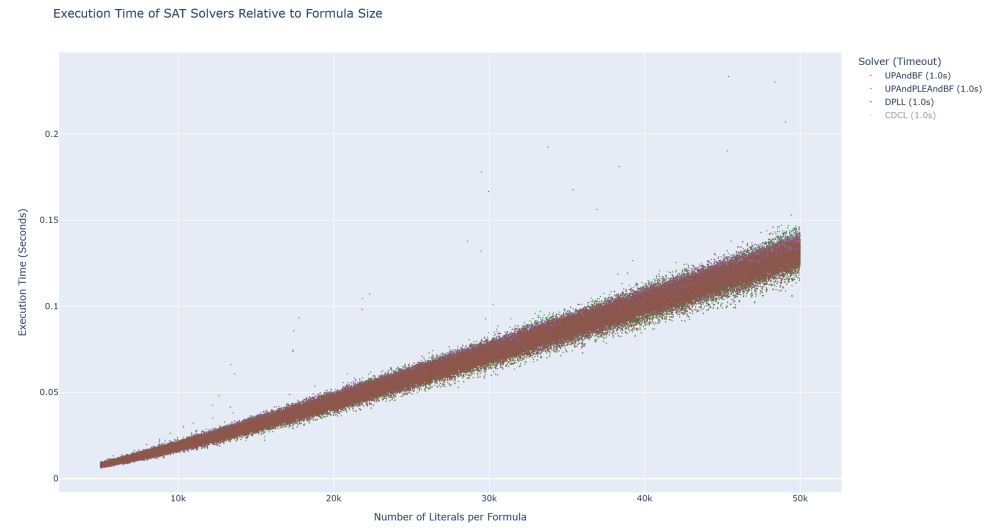

Figure 11: Execution time of the UPBF, UPPLEBF and DPLL solvers relative to formula size with a 25% disjunction to conjunction chance.

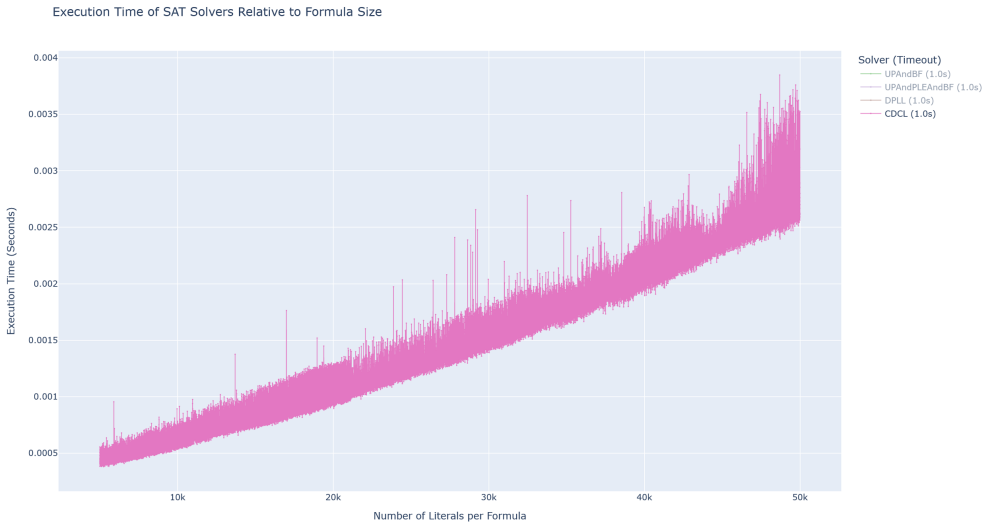

Figure 12: Execution time of the CDCL solver relative to formula size with a 25% disjunction to conjunction chance.

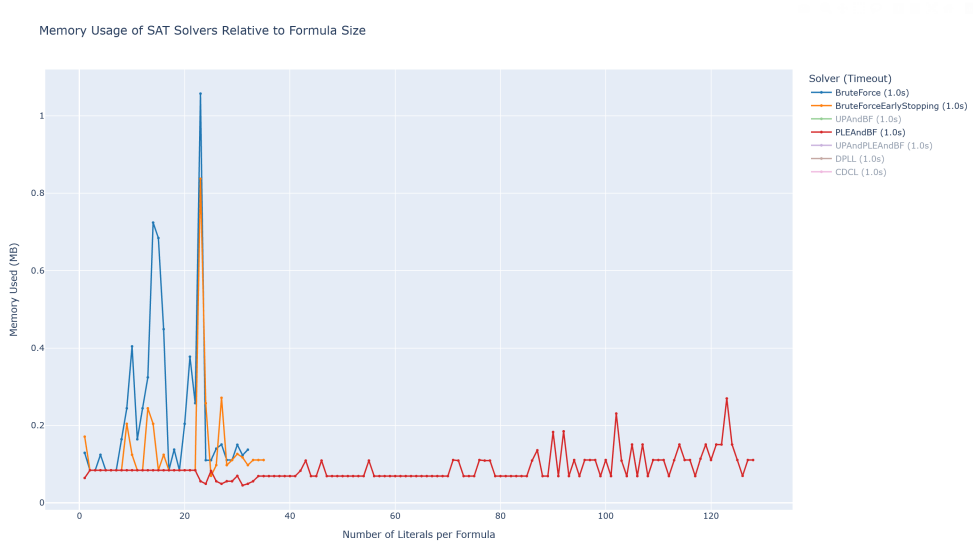

Figure 13: Memory usage of the BF, BFES and PLEBF solvers relative to formula size with a 25% disjunction to conjunction chance.

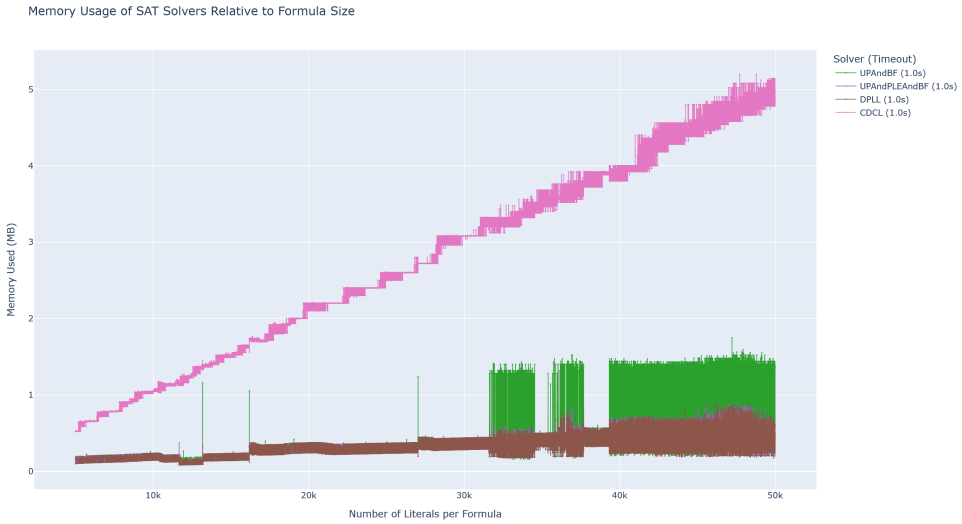

Figure 14: Memory usage of the UPBF, UPPLEBF, DPLL and CDCL solvers relative to formula size with a 25% disjunction to conjunction chance.

Appendix B 75% Average Clause Size Benchmarks

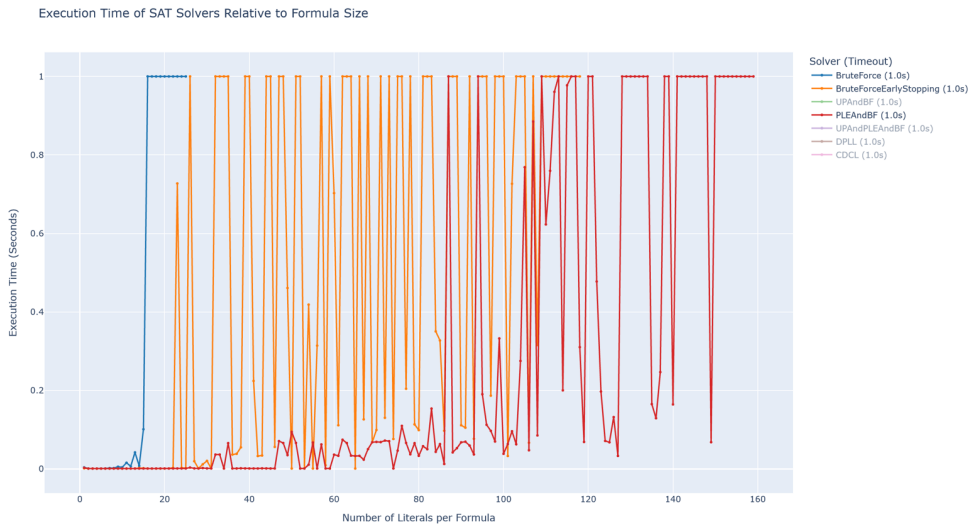

Figure 15: Execution time of the BF, BFES and PLEBF solvers relative to formula size with a 75% disjunction to conjunction chance.

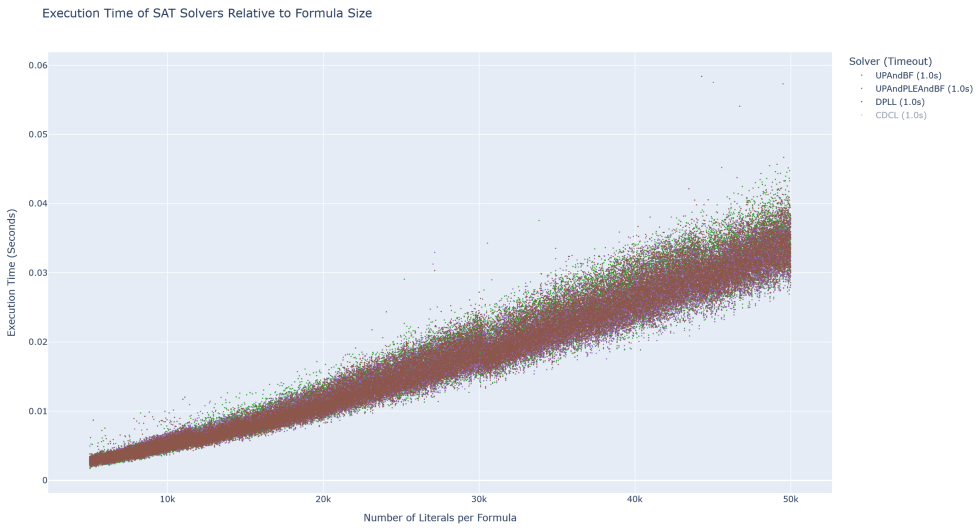

Figure 16: Execution time of the UPBF, UPPLEBF and DPLL solvers relative to formula size with a 75% disjunction to conjunction chance.

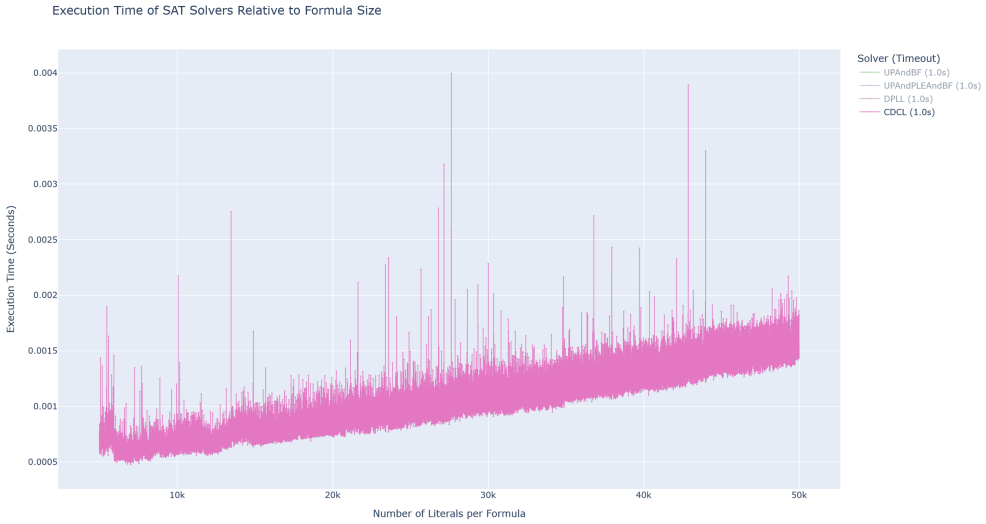

Figure 17: Execution time of the CDCL solver relative to formula size with a 75% disjunction to conjunction chance.

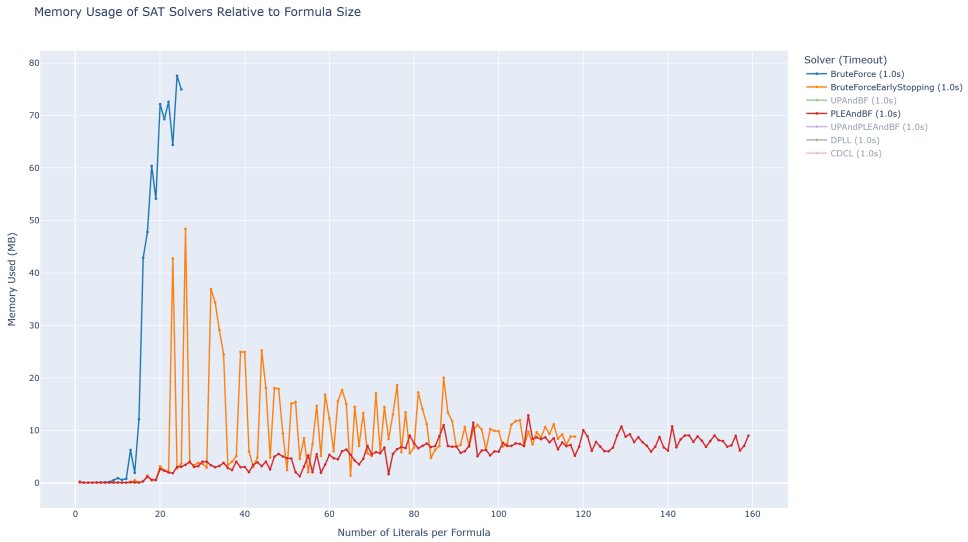

Figure 18: Memory usage of BF, BFES and PLEBF solvers relative to formula size with a 75% disjunction to conjunction chance.

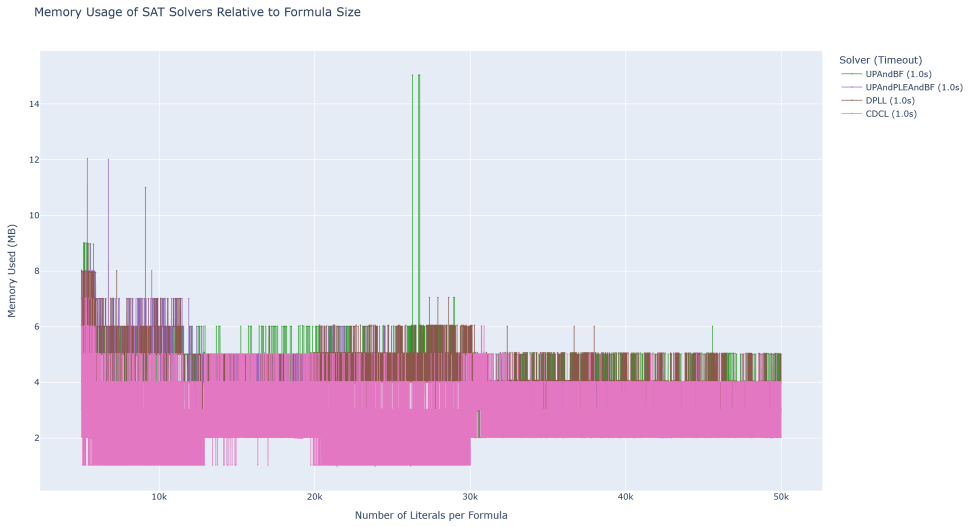

Figure 19: Memory usage of UPBF, UPPLEBF, DPLL and CDCL solvers relative to formula size with a 75% disjunction to conjunction chance.

Appendix C 25% Not Chance Benchmarks

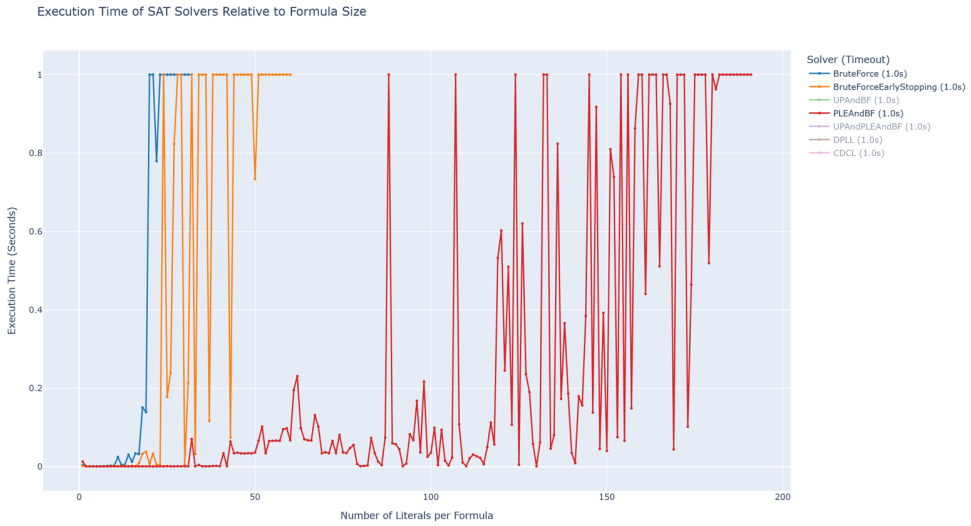

Figure 20: Execution time of the BF, BFES and PLEBF solvers relative to formula size with a 25% chance of each literal being negated.

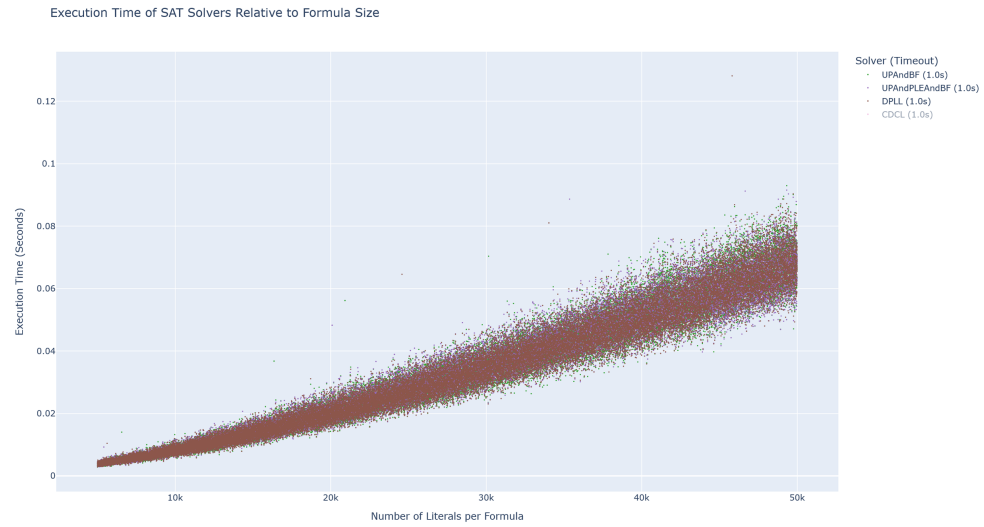

Figure 21: Execution time of the UPBF, UPPLEBF and DPLL solvers relative to formula size with a 25% chance of each literal being negated.

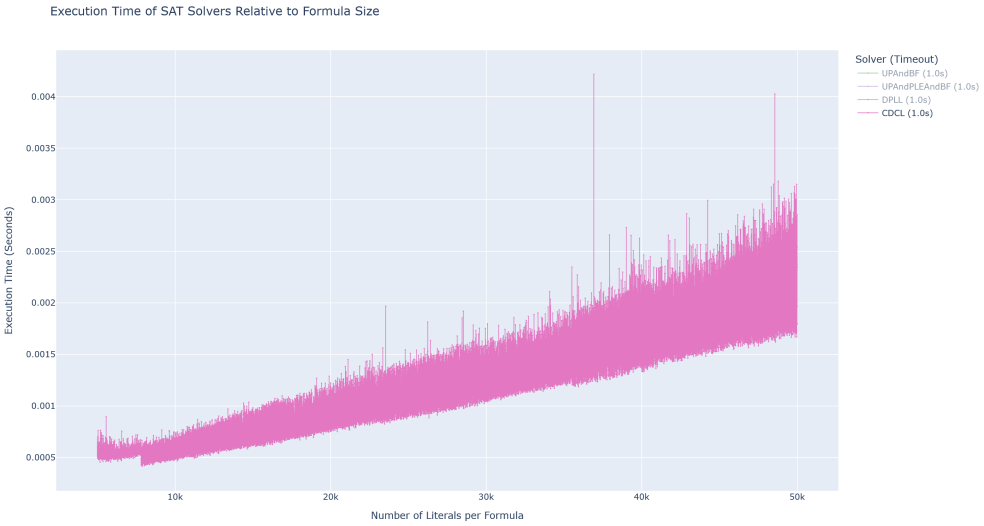

Figure 22: Execution time of the CDCL solver relative to formula size with a 25% chance of each literal being negated.

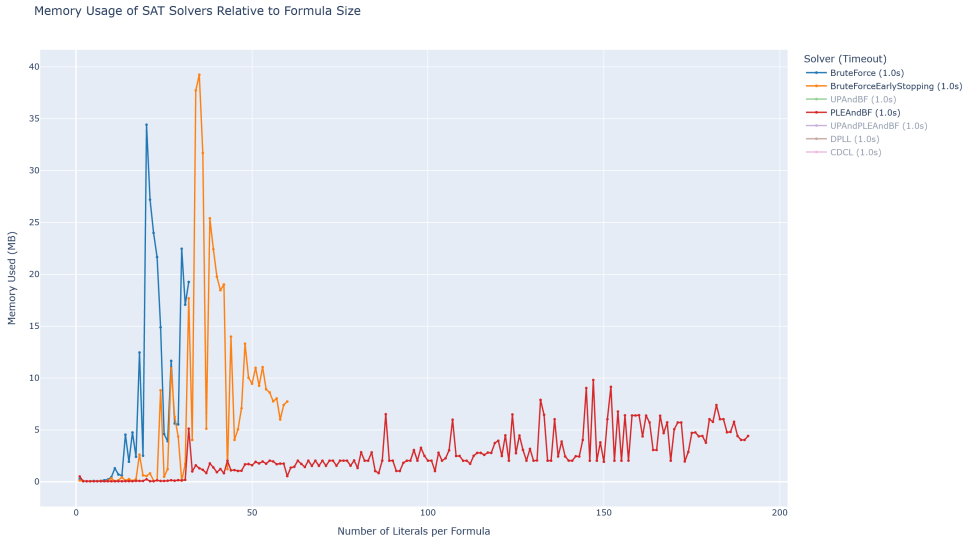

Figure 23: Memory usage of the BF, BFES and PLEBF solvers relative to formula size with a 25% chance of each literal being negated.

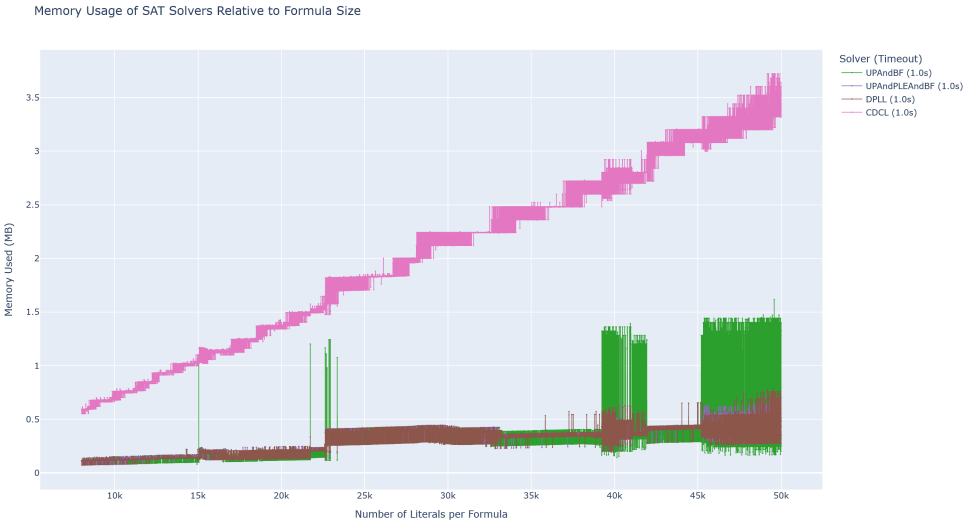

Figure 24: Memory usage of the UPBF, UPPLEBF and DPLL solvers relative to formula size with a 25% chance of each literal being negated.

Appendix D 75% Not Chance Benchmarks

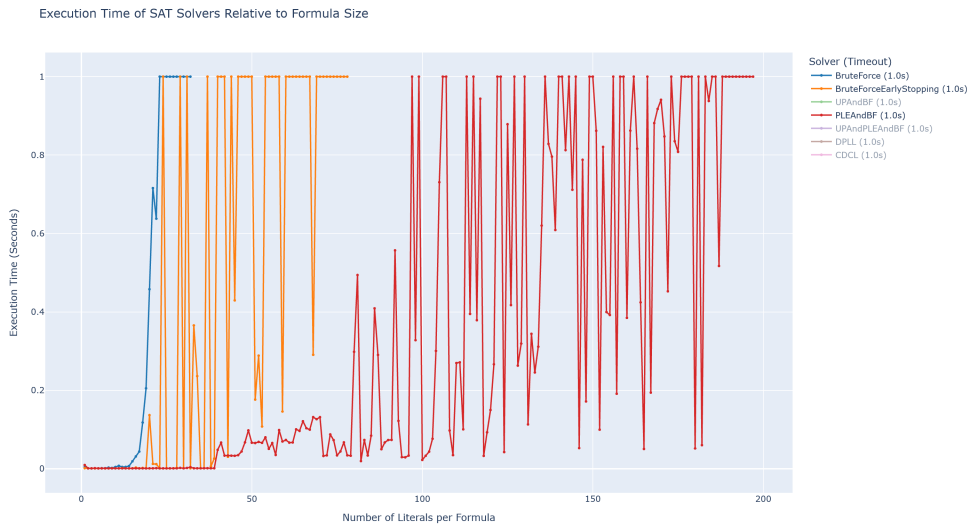

Figure 25: Execution time of the BF, BFES and PLEBF solvers relative to formula size with a 75% chance of each literal being negated.

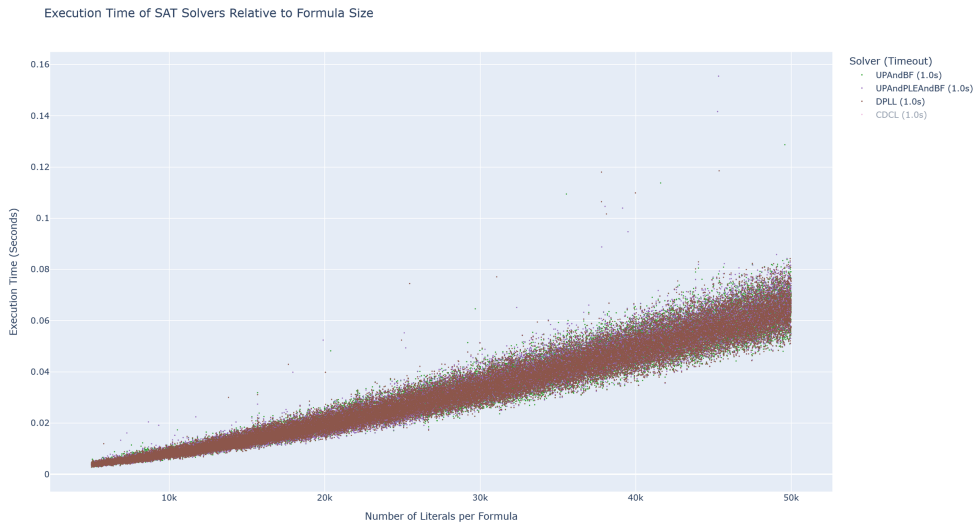

Figure 26: Execution time of the UPBF, UPPLEBF and DPLL solvers relative to formula size with a 75% chance of each literal being negated.

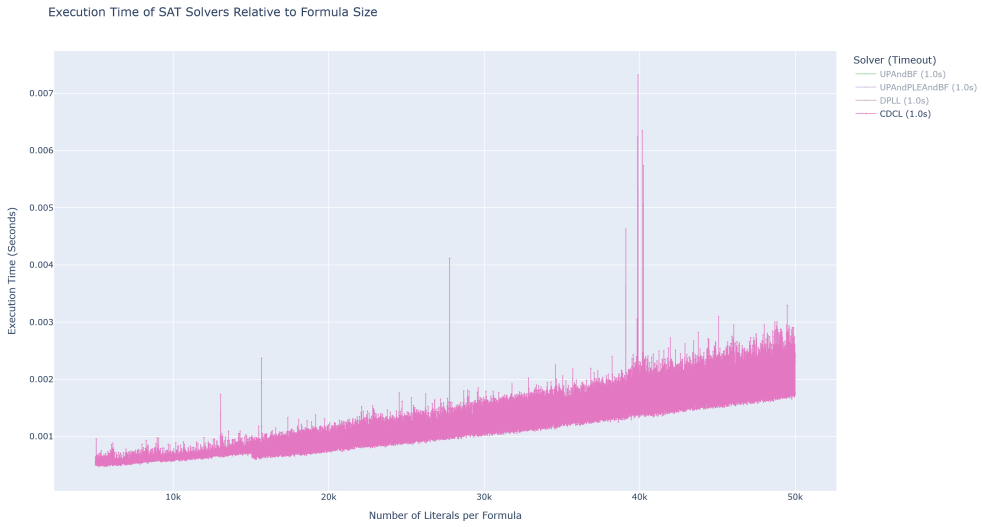

Figure 27: Execution time of the CDCL solver relative to formula size with a 75% chance of each literal being negated.

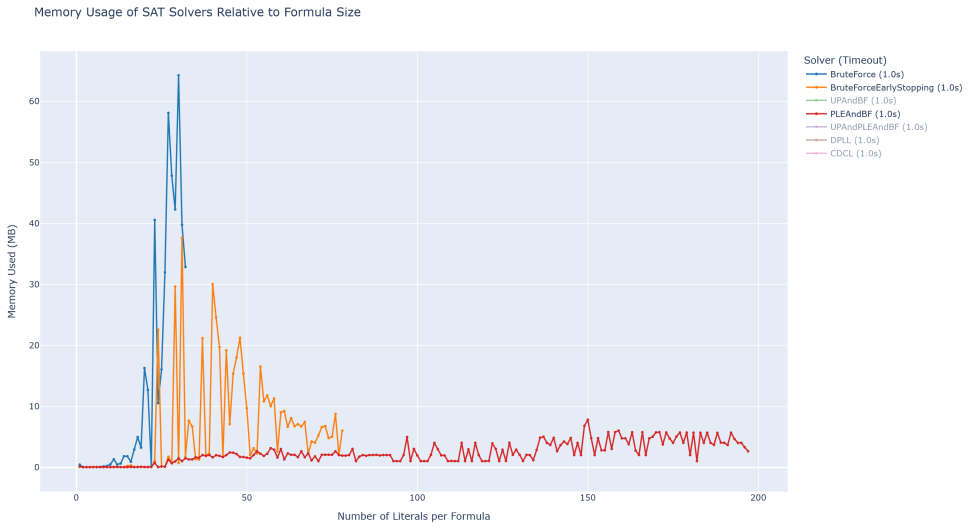

Figure 28: Memory usage of the BF, BFES and PLEBF solvers relative to formula size with a 75% chance of each literal being negated.

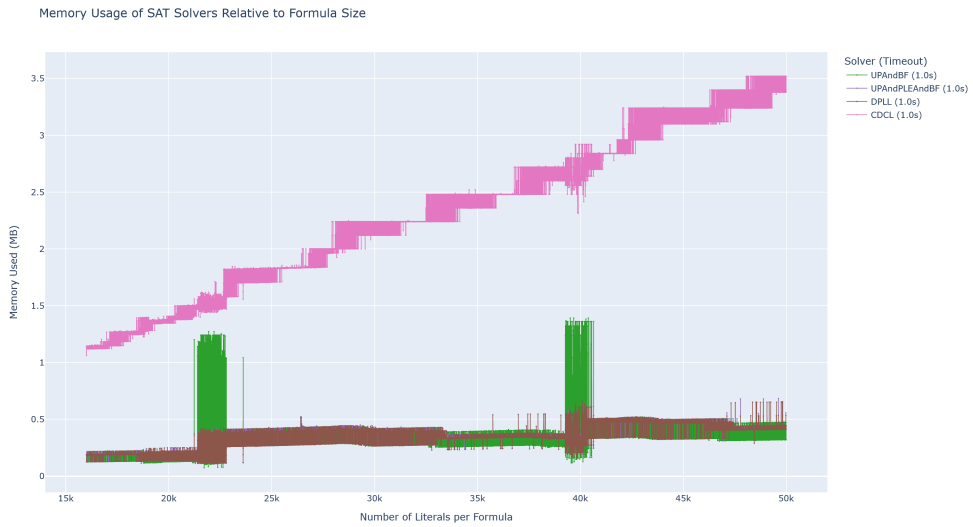

Figure 29: Memory usage of the UPBF, UPPLEBF, DPLL and CDCL solvers relative to formula size with a 75% chance of each literal being negated.