CSC 491-2 Senior Project **Winter/Spring 2025**

SAT Solving & Visualization

Advisor: John Clements *Author: Fisher Lyon*

1 Introduction

My initial goal for this senior project was to develop a truth tree generator for propositional logic and first-order logic. I had taken a class on symbolic logic and thoroughly enjoyed it. Another class that I had taken and really enjoyed was programming languages. I thought that this initial project idea was going to be a great way to combine my liking for both of these classes. That was until I proposed the idea to my advisor and he proposed the idea of SAT solvers, which at the time, I had never heard of. More on SAT solvers will be discussed in the background section, but until then, it will suffice to say that truth trees and SAT solvers have some commonalities.

Truth trees evaluate boolean formulas completely and display truth value assignments (TVA) for the boolean variables that ultimately satisfy the whole formula. As for SAT solvers, their main goal is to find out if a given boolean formula is satisfiable or not—meaning that their task is to find a single TVA that satisfies the whole formula.

That being said, I ultimately chose to explore the world of SAT solving, and I am glad I did. This project required that I absorb a lot of new and interesting information that I would not have been able to take part in if I decided to pursue a project in truth tree generation.

Through the duration of the first quarter of senior project, I took on the task of learning the basics of SAT solving. This required that I understand the following concepts: propositional logic, depth-first search (DFS), conjunctive normal form (CNF), Tseitin Transformation, the David-Putnam-Logemann-Loveland Algorithm (DPLL), and the Conflict-Driven Clause Learning (CDCL) algorithm. During the duration of second quarter, I chose to take my new knowledge and create a learning tool in the form of the visualization of the DFS SAT solver and the CDCL SAT solver. I will explore my journey over the course of these past two quarters in greater depth later, but first, I would like to explain SAT solving in more depth.

2 SAT Solving

The goal of SAT solvers is to solve the Boolean Satisfiability Problem (also called the Propositional Satisfiability Problem) [Wikipedia, 2025a]. The Boolean Satisfiability Problem (BSP) is a problem that asks if there exists a truth-value assignment that satisfies some given Boolean formula. A Boolean formula is a formula that is made up of variables that can either be True or False, and are connected via logical connectives like conjunction (AND, $\wedge$), disjunction (OR, $\vee$), negation (NOT, $\neg$), conditional (IF... THEN..., $\rightarrow$), and biconditional (... IF AND ONLY IF ..., $\leftrightarrow$). A Boolean formula can be satisfied by giving it a valid truth-value assignment that causes the formula to evaluate to True [Wikipedia, 2025b].

When feeding a Boolean formula into a SAT solver, it is typically in Conjunctive Normal Form (CNF), meaning that the formula is a conjunction of disjunctions [Wikipedia, 2025c]. Any Boolean formula can be converted to CNF via the naive approach through the use of logical equivalences and De Morgan's law. When running through example problems with the different solvers, I used Tseitin transformation to convert the Boolean formulas to CNF. This approach is better than the naive approach because the naive approach

to conversion can result in an exponential increase in the size of the formula, while Tseitin transformation results in a formula whose size grows linearly [Wikipedia, 2024].

Having discussed some high level basics of SAT solvers, lets get into what I learned and built over the course of the first quarter of working on this senior project.

3 Quarter 1: SAT Solving Basics

3.1 Introduction

During this quarter of work, I learned the basics of SAT solving and I built a couple of tools and solvers using the Racket programming language. All of the work from this quarter can be observed in a public GitHub repository here: `https://github.com/fisherlyon/SAT_SeniorProject/tree/main`. The tools and solvers made are designed to run from the command line.

3.2 Data Definitions

The Boolean formula Typed Racket implementation can be defined by the following EBNF:

```
<Formula> ::= <var> | <aux>          // Boolean variable or auxiliary variable
        | (~ Formula)                // Negation of a boolean formula
        | (& (Listof Formula))       // Conjunction of boolean formulas
        | (v (Listof Formula))       // Disjunction of boolean formulas
        | (-> Formula Formula)       // Conditional of two boolean formulas
        | (<-> Formula Formula)      // Biconditional of two boolean formulas
```

3.3 SAT Tools

3.3.1 Tseitin Transformation

This tool takes as input a Boolean formula and converts it to CNF using Tseitin transformation. It can take as input a Boolean formula either from the command line or from an input file. The resulting formula is returned either via the command line or via a file to be written to.

3.3.2 CNF File Creation

This tool takes as input a Boolean formula in CNF and writes it to the given file in DIMACs CNF file format. This file format is used commonly for SAT solver input and it is the file format that is used for my SAT solver implementations.

3.3.3 Make Check

This program was made as a tool for myself so that I could test my SAT solver implementations with a known, working SAT solver implementation. When testing satisfiability using one of my SAT solver implementations, the user has the option to write the output to a file. That output file is used, in conjunction with a copy

of the originally tested CNF file to make a "SAT-Check" file. The SAT-Check file is also a CNF file, but at the end, there is the appended, given SAT output. By appending the SAT output to the original CNF file, it adds a series of unit clauses to the original knowledge base that correspond to the TVA of the SAT output. When running this new CNF file through a known, working SAT solver, it ensures that the given SAT output is valid, and it helped me ensure that my SAT implementations are correct.

3.4 Solvers

Throughout my discussions of the following SAT solvers, I will commonly refer to a "knowledge base", which is just the given Boolean formula. The solver implementations interact with the Boolean formulas in integer-clausal CNF. In this form, each Boolean variable is given an integer to represent it: positive for non-negated variables and negative for negated variables. The overall formula is then stripped of its logical operators and converted to a set of clauses, where each clause is a set of Boolean variables, representing a single disjunction. These clauses are part of a larger set, representing the greater overall conjunction. When programming with Boolean formulas, it is easier to work with them in this form.

Each of the following SAT solver implementations has a command-line tool and their inputs are CNF files.

3.4.1 Brute Force SAT Solver

This SAT solver implementation checks every possible truth value assignment (TVA) starting by setting all variables in a Boolean formula to false, then working towards setting all of them to true. For a formula with n Boolean variables, there are 2^n possible TVAs. So, in order to test all possible TVAs, the program uses the binary representation of integers from 0 to $2^n - 1$, assigning each bit in the integer to a variable in the Boolean formula. After an UNSAT, the current TVA integer is incremented to TVA+1, then that is tested. Once one of the TVAs returns SAT, that TVA is returned and each of the bits in the integer is mapped back to either True or False.

3.4.2 DFS SAT Solvers (SAT-I & SAT-II)

The SAT-I solver implementation utilizes Depth-First-Search (DFS) and returns a complete variable instantiation (truth value assignment). It also uses the idea of *conditioning*. When traversing the tree, at each level, a TVA is assigned to a variable. By conditioning the knowledge base on the variable given its TVA, we can reduce the size of the knowledge base until either SAT or UNSAT is returned. As stated previously, this variation returns a complete variable instantiation, which means that we have conditioned the knowledge base on every variable.

The SAT-II solver implementation is just an extension of the previously mentioned SAT-I. The only difference between SAT-II and SAT-I is that SAT-II returns a partial variable instantiation. This can be done due to the fact that it is not always necessary to traverse the tree all the way down to a leaf node in order to have SAT returned.

3.4.3 DPLL SAT Solver

This SAT solver implementation utilizes the David-Putnam-Logemann-Loveland (DPLL) algorithm and heavily relies on the idea of unit resolution, which reduces the clausal form of a knowledge base based on unit clauses (clauses of size one). Unit resolution returns a set of literals that were either present as unit

clauses in the knowledge base or derived by unit resolution (I), and a new knowledge base which results from conditioning the knowledge base on I (Γ).

When Γ is empty, SAT is returned, and if there is a contradiction in Γ, then UNSAT is returned. For this method to progress, literals are chosen, using a heuristic, to be added to Γ as a clause for unit resolution. The heuristic used in my implementation is called "Maximum Occurrence in Minimum-sized Clauses" (MOM). The MOM heuristic helps select a variable that is likely to influence the satisfiability of the formula as quickly as possible, ideally reducing the search space early. When SAT is achieved, I is returned.

3.4.4 CDCL SAT Solver

This SAT solver implementation utilizes the idea of Conflict-Driven Clause Learning (CDCL). This version also uses unit resolution, but it is different than the one used in DPLL as it also takes in a decision sequence (D) and a set of learned clauses (Γ). The general idea of this algorithm is that many decisions—literals to be added as clauses to the knowledge base for unit resolution—are made and once a conflict/contradiction is reached, we analyze it to create *learned clauses* that are implied by the knowledge base that we will ultimately add to the knowledge base.

This is done by constructing an implication graph (IG) to see what decisions and implications ultimately led up to the contradiction. From analyzing the IG, we are able to derive learned clauses that we can add to Γ that are implied by the knowledge base. In my implementation, an asserting learned clause is chosen based on a heuristic called the first UIP. A UIP (unique implication point) is a dominator of the path in the IG from the decision made at the highest decision level to the contradiction. A dominator is a node in the graph that is passed through in all possible paths from the source node (decision node made at the highest decision level) to a target node (contradiction). The first UIP is the UIP that is the closest to the contradiction.

Once learned clauses are derived from creating cuts in the IG, we choose the one that is asserting and that has the first UIP in it. We then back-track to the decision level where the contradiction originated. When a literal is chosen to be a decision, it must be the case that neither itself or its negation is implied by unit resolution. This process runs until either unit resolution doesn't detect a contradiction and there aren't any more literals to be chosen, in which SAT is returned along with I, or if unit resolution detects a contradiction and the decision sequence D is empty, in which UNSAT is returned.

4 Quarter 2: Visualization

4.1 Introduction

During this quarter of work, I set out to create a visualization tool for the DFS and CDCL SAT solving algorithms. For the DFS algorithm, the tree is displayed and different decisions are tested until either SAT or UNSAT is achieved. For the CDCL algorithm, the tree is built and displayed based on decisions made and once a contradiction/conflict is reached, we construct the implication graph and derive the learned asserting clause. That clause is then added to the initial knowledge base and the process starts over. As with the DFS algorithm, it runs until either SAT or UNSAT is achieved. All of the work from this quarter can be viewed in a public GitHub repository here: `https://github.com/fisherlyon/SAT-Start`.

One of the challenges that I had to face throughout the duration of this portion of the project was the transition from using big-step semantics to small-step semantics. For the solvers I built during the first quarter, they are given an input and return an output, without displaying how they got from point A to

point B. When visualizing the process of these algorithms during this quarter, I had to step through each of the processes and display the different states along the way.

For this next portion of the paper, I am going to explain some things about each of the tools, and then go through an in-depth explanation of the DFS and CDCL algorithms through the visualization of a given example. We will start with the DFS algorithm first and will be using the same Boolean formula example for each of the algorithm visualizations.

4.2 DFS SAT Solver Visualization Tool

4.2.1 Tool Traversal

The "Home" page of the DFS algorithm visualization has three buttons labeled "Small", "Medium", and "Large", each pointing to their own DFS SAT example. Due to the fact that complete binary trees grow exponentially, the small example has a formula that consists of two Boolean variables, the medium example has a formula that consists of three Boolean variables, and the large example has a formula that consists of four Boolean variables.

Upon choosing one of the examples, the user will see a complete binary tree where the depth of the tree has to do with how many unique Boolean variables the corresponding formula is made up of. For example, the formula $\{\{A\} \wedge \{\neg B\}\}$ will create a complete binary tree of depth two, where Boolean variable A corresponds to level zero and Boolean variable B corresponds to level one. The final level, level 2, which consists of the leaf nodes, displays the ends of the possible paths from the root node. The user will also see the input Boolean formula, current decision, current DFS path, and current stage in the DFS SAT algorithm. (See Figure 1)

There are also a couple of buttons on this screen:

- Home - Returns you to the home page where the example selection is.
- Legend - Displays a screen that explains the meaning of the different colors, lines, and symbols used.
- Next - Steps forward in the DFS SAT algorithm.
- Undo - Steps backward in the DFS SAT algorithm.

"Next" and "Undo" operations can also be conducted via the left and right arrow keys, where the left arrow key is "Undo" and the right arrow key is "Next".

That wraps up the general structure of the tool, so let's get into going over an in-depth example.

4.2.2 Visualization and Explanation

The example Boolean formula that we will testing satisfiability of is the following (medium example):

$$\Delta = \{\{A \vee C\} \wedge \{\neg A \vee B\} \wedge \{\neg A \vee \neg B\}\}$$

Figure 1 shows the initial state of the DFS SAT algorithm for the given Boolean formula (Δ). The complete binary tree has a depth of three, where level zero corresponds to Boolean variable A, level one corresponds to Boolean variable B, level two corresponds to Boolean variable C, and level three corresponds to the end of the possible paths. Solid-lined edges from a Boolean variable node mean that we will take that variable

to be $True$ for our decision, and dotted-lined edges mean that we will take that variable to be $False$ for our decision.

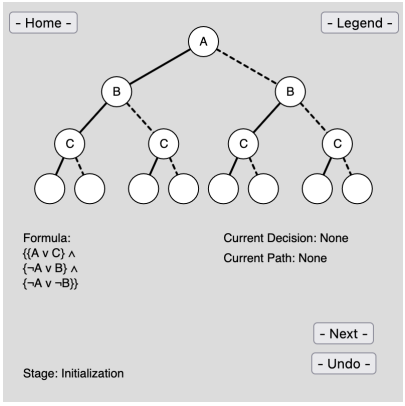

Figure 1: Initial view after clicking an example button

The visualization implementation has three major steps:

1. Make a decision via DFS
2. Condition the formula on the current decision
3. Check the formula for SAT/UNSAT/Contradiction

As stated in the first major step, the algorithm makes literal decisions based on DFS. Recall that DFS uses a stack data structure for its implementation, based on the tree displayed in the Figure 1, The first path to be tested will be where all Boolean variables are $True$. If contradictions occur, the path decided by DFS will traverse the tree from left to right until either SAT is achieved, or all possibilities have been tested, and none return SAT, in which UNSAT is returned.

Conditioning the formula based on the decision made is how the formula is reduced. Because the formula is in CNF, conditioning is made easy. To show this, let's make the first step in the DFS SAT algorithm.

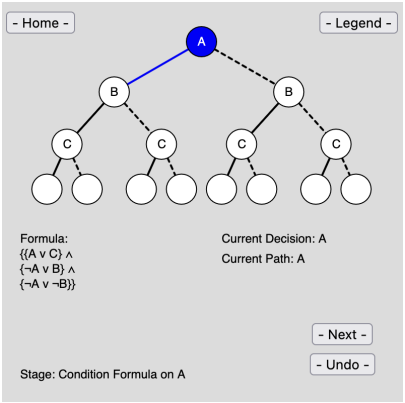

Figure 2: Formula and Tree before conditioning given decision $A = True$

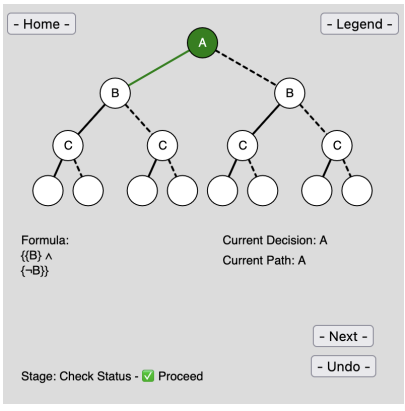

Figure 3: Formula and Tree after conditioning and after status check

Figure 2 shows that from DFS, the first decision is A and that we are preparing to condition the formula on A, which can be denoted by $\Delta|A$. The A node being blue displays that it is currently being processed, and the outgoing left solid-lined edge being blue displays that we are deciding on A to be $True$. Figure 3 shows the reduced formula after conditioning, as well as, the A node and its left outgoing solid-lined edge being green, representing the idea that making decision A is temporarily valid.

$$\Delta = \{\{A \vee C\} \wedge \{\neg A \vee B\} \wedge \{\neg A \vee \neg B\}\}$$

$$\Delta|A = \{\{B\} \wedge \{\neg B\}\}$$

By conditioning the formula on A, we treat A as $True$ in the context of the formula. Let's label the conjuncts in Δ.

1. $\{A \vee C\}$
2. $\{\neg A \vee B\}$
3. $\{\neg A \vee \neg B\}$

By treating A as $True$ in conjunct 1, it becomes satisfied because we are left with the disjunction, $True$ OR C, which reduces to $True$, by the definition of disjunction. By treating A as $True$ in conjunct 2, it reduces it to the disjunction, $False$ OR B, which reduces to B, by the definition of disjunction. Lastly, by treating A as $True$ in conjunct 2, it reduces it to the disjunction, $False$ OR $\neg B$, which reduces to $\neg B$, by the definition of disjunction.

1. $\{A \vee C\} \Rightarrow \{True \vee C\} \Rightarrow True$
2. $\{\neg A \vee B\} \Rightarrow \{False \vee B\} \Rightarrow \{B\}$
3. $\{\neg A \vee \neg B\} \Rightarrow \{False \vee \neg B\} \Rightarrow \{\neg B\}$

Thus, we are left with the following conjunction:

$$\Delta|A = \{True \wedge \{B\} \wedge \{\neg B\}\}$$

And by the definition of conjunction, conjunct 1 ultimately disappears from the formula, resulting in:

$$\Delta|A = \{\{B\} \wedge \{\neg B\}\}$$

And because conditioning hasn't resulted in a contradiction ($\Delta = \{\{\}\}$), this decision is temporarily valid. The next step results in the decision of taking B to be $True$.

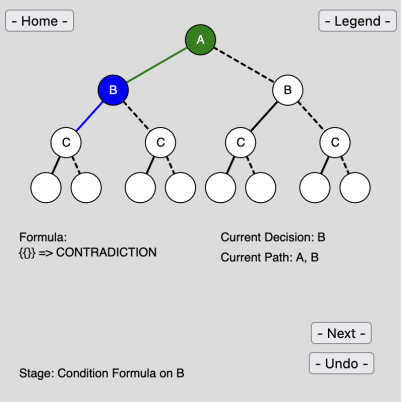

Figure 4: Formula and Tree before conditioning given decision $B = True$

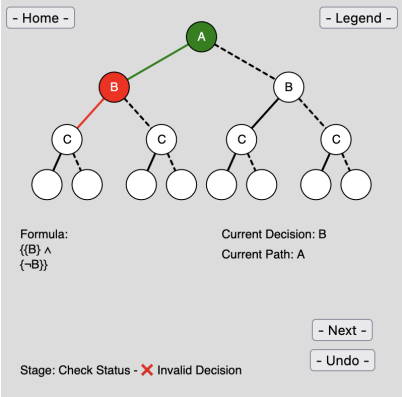

Figure 5: Formula and Tree after conditioning and after status check

As shown by Figure 4, conditioning the formula on B results in a contradiction. Let's once again label the conjuncts in Δ before the conditioning on B, and show what they reduce to after the conditioning on B.

1. $\{B\} \Rightarrow \{True\} \Rightarrow True$
2. $\{\neg B\} \Rightarrow \{\neg True\} \Rightarrow \{False\} \Rightarrow \{\}$

Because conjunct 1 reduces to $True$, it is removed from the formula, and we are just left with $\Delta|A, B = \{\{\}\}$, which displays a contradiction. This idea can be further illustrated by the fact that after a couple of reductions of the conjuncts above, we arrive at the formula, $\{\{True\} \wedge \{False\}\}$, which by the definition of conjunction, results in a contradiction.

As shown in Figure 5, because a contradiction occurs, we take our decision back. Figure 6 illustrates the choice of $\neg B$ instead, which also ultimately results in a contradiction, causing us to reverse our decision making to where we decided A to be $True$. Because choosing A to be $True$ ultimately leads to a contradiction given subsequent decisions, we will begin to traverse the right side of the tree by deciding A to be $False$.

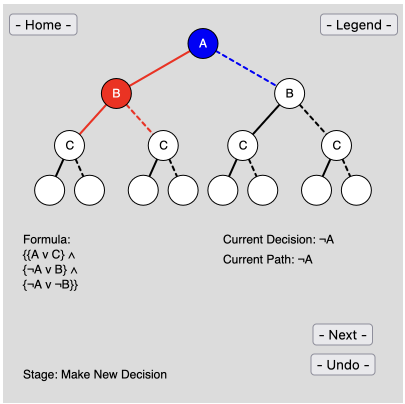

Figure 6: Formula and Tree after concluding that decisions B and $\neg B$ both result in a contradiction—shown by their red fill.

This decision ultimately leads to the formula being reduced to $\{\{C\}\}$, showing that all instances of B have been removed, which means that B can be either $True$ OR $False$, and it would not affect the outcome. Then, once we decide C to be true, we are left with $\{\}$, showing satisfiability of the formula. The complete DFS path can be illustrated via Figure 7, and the conditioning can be explained via the steps below:

- $\Delta = \{\{A \vee C\} \wedge \{\neg A \vee B\} \wedge \{\neg A \vee \neg B\}\}$
- Decide $\neg A \Rightarrow \Delta|\neg A = \{\{False \vee C\} \wedge \{True \vee B\} \wedge \{True \vee \neg B\}\} = \{\{C\} \wedge True \wedge True\} = \{\{C\}\}$
- Decide $B \Rightarrow \Delta|\neg A, B = \{\{C\}\}$
- Decide $C \Rightarrow \Delta|\neg A, B, C = \{True\} = \{\}$

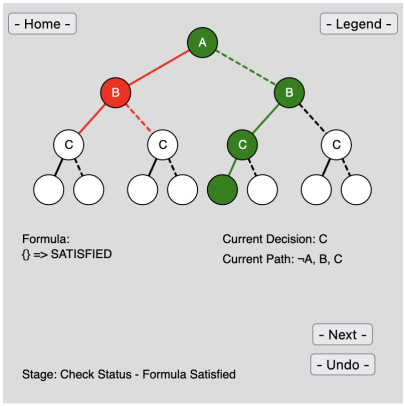

Figure 7: Illustration of satisfied formula given TVA: $A = False, B = True, C = True$

Because SAT has been reached, the DFS SAT algorithm returns the current path, which is also the truth value assignment (TVA) for the given Boolean variables.

Thus, we have reached the end of the DFS SAT algorithm visualization segment, and we will now begin the CDCL SAT algorithm visualization example run-through.

4.3 CDCL SAT Solver Visualization Tool

4.3.1 Tool Traversal

General traversal of the CDCL visualization tool is the same as the one for the DFS tool. There is one change, however, on the "Home" page. That is, the inclusion of a few text-boxes so that the user can input their own examples. In the first text-box, the user should enter the formula in integer-clausal conjunctive normal form (CNF). This form is when, for formulas already in CNF, the formula as a whole gets encapsulated, the conjuncts become encapsulated and are comma-separated, and the disjuncts become integers, where negations are negative integers and non-negations are positive integers, which are also comma-separated. The disjuncts should start at 1 and should increment by +1 for each unique variable. Then, in the second text-box, the user should enter the list of variable characters, as individual strings, that they wish to represent the corresponding integers in the formula. The number of listed variables should be equal to the set of the absolute value of the integers used in the integer-clausal CNF. If either of the given text-box inputs doesn't

match the needed form, a red box will appear around the text-box indicating that the input isn't valid. An example conversion is shown below for the following formula.

$$\{\{A \vee C\} \wedge \{\neg A \vee B\} \wedge \{\neg A \vee \neg B\}\}$$

- Integer-Clausal CNF: $[[1, 3], [1, 2], [-1, -2]]$
- List of Variables: [“A”, “B”, “C”]
- $S = \{\, |x| \,|\, x \in \bigcup \{\{1, 3\}, \{1, 2\}, \{-1, -2\}\}\,\} = \{1, 2, 3\}$
- $V = \{\, y \,|\, y \in \text{List of Variables}\} = \{\text{“}A\text{”}, \text{“}B\text{”}, \text{“}C\text{”}\}$
- $|S| = |V| = 3$

That being said, let's get into the example CDCL SAT visualization.

4.3.2 Visualization and Explanation

As stated previously, the example Boolean formula that we will testing satisfiability of is the same one that we used for DFS:

$$\Delta = \{\{A \vee C\} \wedge \{\neg A \vee B\} \wedge \{\neg A \vee \neg B\}\}$$

There are some key pieces of data that we carry around with us throughout the duration of CDCL:

- D: A decision sequence (list of decisions made thus far)
- Γ: A set of learned clauses (derived from implication graphs)
- I: A set of literals that were either present as unit clauses in the formula or as unit clauses derived by unit resolution

The CDCL algorithm can be broken down into two, repeating steps:

1. Perform unit resolution and make new decisions until SAT, UNSAT, or a Contradiction.
2. If SAT or UNSAT, return. Otherwise, build an implication graph to learn a new clause that is implied by the original Boolean formula that will get added to the formula in the next run.

I will start by explaining step 1 and what it entails.

The algorithm starts by performing unit resolution (UR). UR takes as input: Δ, D, Γ, and I and returns Δ' and I', the reduced Boolean formula, and the updated I. At the start, D is transformed into a set of unit clauses, where each unit clause consists of one of the made decisions—call this D'. A unit clause is a clause of length 1. Then, we perform the following union:

$$\Delta_{\Gamma, D'} = \Delta \cup \Gamma \cup D'$$

Then, $\Delta_{\Gamma, D'}$ is scanned for unit clauses so that it can be conditioned on by the literal contained within the found unit clause. Let's call the found literal—if one is found—l, where l is either $True$ or $False$. This means that l must be $True/False$ across the entirety of $\Delta_{\Gamma, D'}$, since it is its own individual conjunct within the formula. We will now condition $\Delta_{\Gamma, D'}$ on l.

This process runs recursively until $\Delta_{\Gamma,D'}$ no longer contains any more unit clauses, in which it becomes Δ'. Throughout the process, literals that were either present as unit clauses in the formula or derived by UR are added to I'. At the end, Δ' and I' are returned.

After returning, if $\Delta' = \{\{\}\}$—a contradiction—then we transition into constructing an implication graph. Otherwise, we will make a new decision and perform UR again. In this implementation, decisions are made based on the first literal observed in Δ'. The rule is that a new decision, d or $\neg d$, must not be implied by UR—not contained within I'. If d is a literal from Δ', it certainly won't be in I' since I' is the set of literals that were either present as unit clauses in the formula or as unit clauses derived by UR.

An illustration of UR can be shown below via the use of the visualization tool. During the first iteration of CDCL, when UR runs, nothing happens because Δ doesn't contain any unit clauses. So, because there isn't a reason to return or construct an implication graph, we make a new decision, which is A, since that is the first disjunct in the first conjunct of Δ.

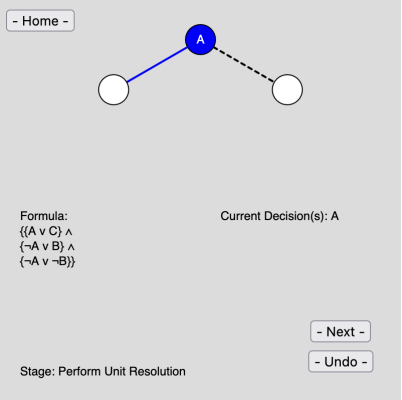

Figure 8: Formula and Tree before unit resolution given decision $A = True$

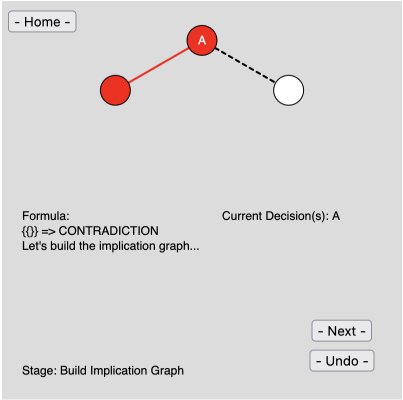

Figure 9: Formula and Tree after unit resolution and after status check

Now, the formula from Figure 9 shows that the the formula from Figure 8 reduced down to a contradiction after the call to UR. Let's breakdown just how that happened, recalling that unit resolution takes as input $\Delta, D, \Gamma,$ and I.

- Inputs to UR:
- $D = \{A\}$
- $\Gamma = \{\}$
- $I = \{\}$
- Beginning steps before UR:
- $D' = \{\{A\}\}$
- $\Delta_{\Gamma,D'} = \Delta \cup \Gamma \cup D' = \{\{A \vee C\} \wedge \{\neg A \vee B\} \wedge \{\neg A \vee \neg B\} \wedge \{A\}\}$
- Find a unit clause: $\{A\}$
- Conditioning: $\Delta_{\Gamma,D'}|A = \{\{B\} \wedge \{\neg B\}\}$

- Find a unit clause: $\{B\}$
- Conditioning: $\Delta_{\Gamma,D'}|A, B = \{\{\}\}$
- Find a unit clause: There aren't any, so return!
- Return $\Delta' = \{\{\}\}$ and $I' = \{A, B\}$

Now that we have arrived at a contradiction we need to check for UNSAT before proceeding to constructing the implication graph. To check for UNSAT, all we need to do is check if D is empty, and because it isn't we are good to proceed to implication graph construction. If D were to be empty, it would imply that we arrived at a contradiction without making any decisions, showing that the formula is unsatisfiable.

Let's now move on to constructing the implication graph (IG). When constructing the IG, we need to take D and I' with us. D tells us the decisions that we made that led to the contradiction, and I' tells us if there were any other literals that were derived from UR that helped cause the contradiction. We start by constructing the IG by placing nodes that represent our decisions made. This can be illustrated via the visualization tool in Figure 10. Since A was our only decision, it is the only one. We will also want to note the decision level in which each decision is made. This can be derived by simply taking the index of each of the decisions in D. Because D just contains A, decision node $A = True$ was made at decision level 0. Decision nodes are depicted in green.

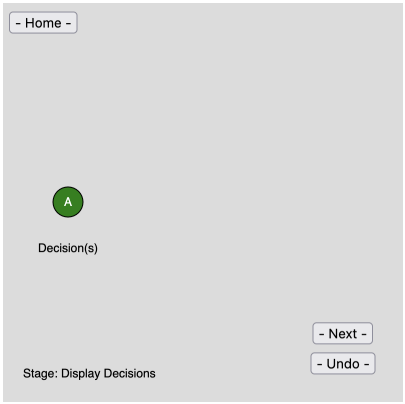

Figure 10: Displaying of the decision node(s) in the implication graph

Now, it's time to add the implication nodes. These are derived by finding conjuncts in the formula, that when paired with the decisions or other implications, allows you to imply something about another literal. To illustrate this, let's label the conjuncts in the original formula.

1. $\{A \vee C\}$
2. $\{\neg A \vee B\}$
3. $\{\neg A \vee \neg B\}$

Given that A is true based on our decision, we know that conjunct 1 becomes $True$ and gets removed from the formula. As for conjunct 2, $\neg A$ gets removed, leaving just $\{B\}$. Therefore, we can imply that, based our decision of A being $True$, B must also be $True$ in order for conjunct 2 to become $True$ as well. Hence, we will add an implication node to the graph for $B = True$. And because this implication came from conjunct

2, we will note that, and we will also note that it was made at decision level 0, since it came from decision node $A = True$, which was also made at decision level 0. Implication nodes in the visualization are depicted in orange.

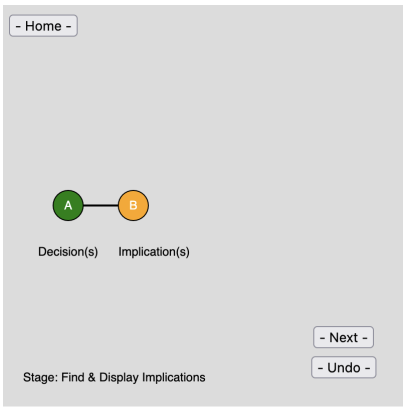

Figure 11: Displaying of the implication node $B = True$ in the IG

The process of adding implication nodes to the graph continues until we reach the contradiction implication. So, let's proceed with the same procedure. We know that conjunct 1 still results in $True$ due to decision node $A = True$. Now that we have implication node $B = True$, we know that conjunct 2 becomes $True$. So, now we just have conjunct 3 to review. With both A and B being $True$, conjunct 3 becomes $\{False \vee False\}$, which is equivalent to $\{False\}$, which is equivalent to $\{\}$. So there we have it, conjunct 3 is the source of our contradiction and it too was made at decision level 0 since it came from decision node $A = True$. Because the contradiction implication was derived from both, decision node $A = True$ and implication node $B = True$, it will have branches connecting to both nodes. The final IG looks like what is shown in Figure 12 and the contradiction implication node is depicted in red.

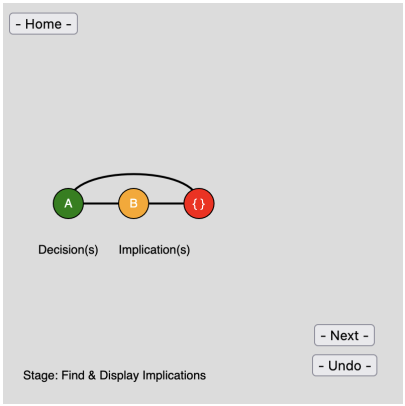

Figure 12: Displaying the contradiction node in the IG

Now that we have a complete IG, we can begin the process of finding a learned clause. Finding a learned clause is important in the CDCL algorithm because it allows us to learn new things about the formula, which allows us to progress towards finding SAT or UNSAT. To start this procedure, we need to identify the first unique implication point (UIP). This is just a common heuristic used when developing a learned clause. A UIP helps the CDCL algorithm construct a learned clause that cuts off just enough of the current

contradiction assignment to avoid the contradiction in the future—without back-tracking too far or learning a useless clause. To get more specific about what an UIP actually is, it is a node in the IG that dominates all paths from the decision node made at the highest level, to the contradiction node. Then, the first UIP is the UIP that is closest to the contradiction node. In the case of our IG, decision node A is the first UIP, as it is a part of both of the only two paths to the contradiction node from the decision node made at the highest level. Decision node B isn't a UIP because it doesn't lie on the path: $A \rightarrow \{\}$. In the visualization in Figure 13, the first UIP is depicted in yellow.

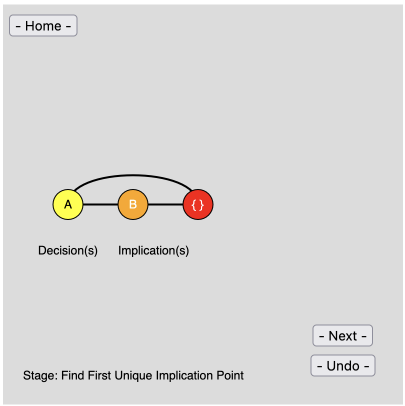

Figure 13: Displaying the first UIP on the IG

It is now time to begin the search for the learned clause. This procedure is done through a reverse Breadth-First Search (BFS) of the IG, starting at the contradiction node. Throughout this process, we are going to keep track of the current node, its incoming neighbors, the BFS queue, the nodes visited, and the conflict set. At the end of the BFS, the negation of the conflict set becomes the learned clause. In general, when a node becomes the current node, its incoming neighbors are added to the conflict set. More specifically, here is how the learned clause is put together based on the BFS:

- If the current node is the first UIP, we will mark it as visited, leave it in the conlict set, and proceed with the BFS.
- If the current node isn't the first UIP, but is in the conflict set, one of two things can occur:
  - The current node is a decision node, meaning it doesn't have any incoming nodes, in which we will keep it in the conflict set—treating it similarly to how we treated the interaction with the first UIP node.
  - The current node is an implication node, in which we will update the queue by adding any nodes from the current node's incoming neighbors that haven't already been visited, and by removing it from the current conflict set and replacing it with its incoming neighbor nodes if they haven't already been visited.
- If the current node isn't the first UIP and isn't in the conflict set, then we just add its incoming neighbor nodes to the queue if they haven't already been visited.

To start of this portion of the algorithm, the BFS queue and conflict set will be initialize to just contain the contradiction node. Then from there, the procedure runs until the BFS queue is empty. The five images below, show the complete step-by-step run of the reverse BFS traversal of the IG. Blue nodes represent the current node and gray nodes represent visited nodes.

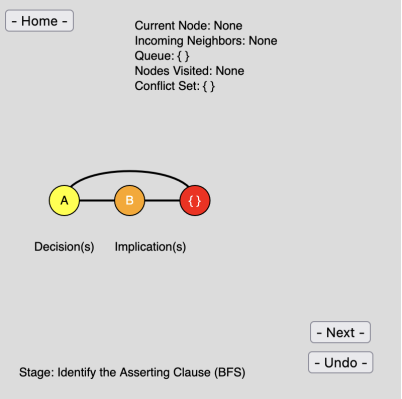

Figure 14: Initial state of the reverse BFS of the IG

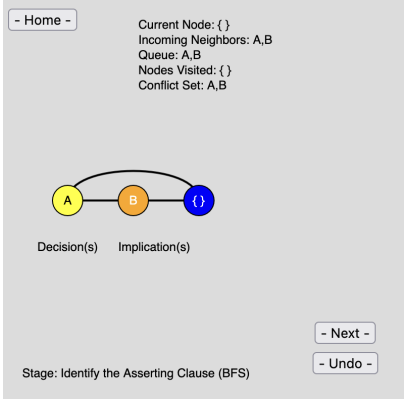

Figure 15: Step 1 of the reverse BFS of the IG

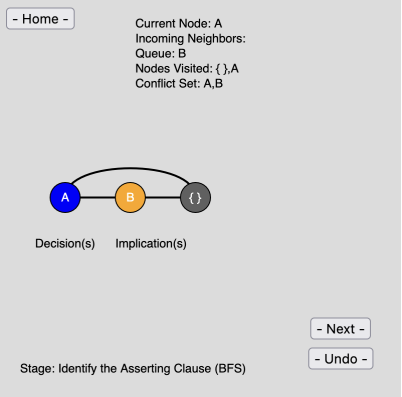

Figure 16: Step 2 of the reverse BFS of the IG

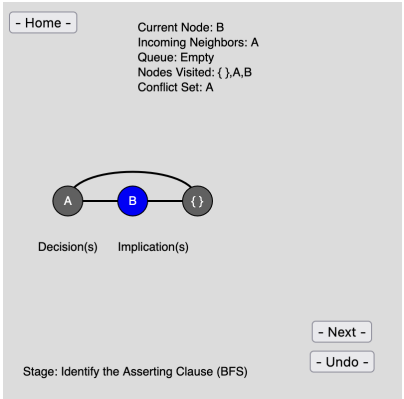

Figure 17: Step 3 of the reverse BFS of the IG

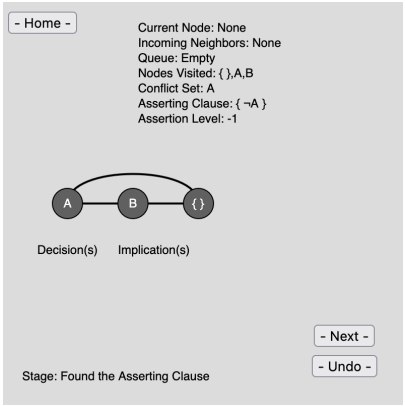

Figure 18: The 4th and final step of the reverse BFS of the IG

Figure 18 shows that the asserting learned clause was found: $\{\neg A\}$, and that the assertion level is -1. The

assertion level of an asserting learned clause is the second-highest decision level among the literals in the learned clause, after conflict analysis. The assertion level tells us how far we need to back-track in our decision making. Since our asserting learned clause is only of length one, and thus it doesn't have a second-highest decision level among its literals, the assertion level becomes -1, meaning that we will need to back-track to before all of our prior decisions were made.

So, now we go back to more decision making, and it would appear as if we are starting at square one again...

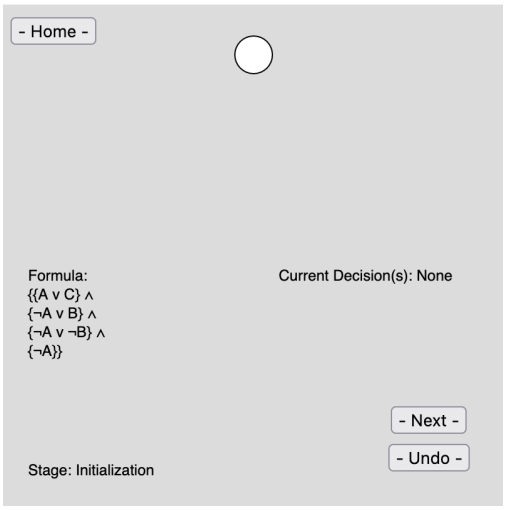

Figure 19: Beginning of decision making after finding a learned clause

But, after unit resolution runs, we see that we arrive at SAT.

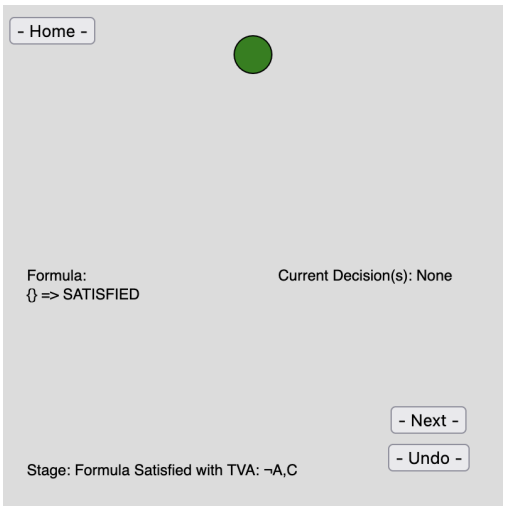

Figure 20: SAT!

This is because the learned asserting clause that we added to the formula was a unit clause, meaning that unit resolution runs smoothly all the way through! Here is an expanded view of how unit resolution ran:

- Inputs to UR:
- $D = \{\}$
- $\Gamma = \{\{\neg A\}\}$
- $I = \{\}$
- Beginning steps before UR:
- $D' = \{\}$
- $\Delta_{\Gamma,D'} = \Delta \cup \Gamma \cup D' = \{\{A \vee C\} \wedge \{\neg A \vee B\} \wedge \{\neg A \vee \neg B\} \wedge \{\neg A\}\}$
- Find a unit clause: $\{\neg A\}$
- Conditioning: $\Delta_{\Gamma,D'}|\neg A = \{\{C\}\}$
- Fins a unit clause: $\{C\}$
- Conditioning: $\Delta_{\Gamma,D'}|\neg A, C = \{\} = SAT$
- Find a unit clause: There aren't any, so return!
- Return $\Delta' = \{\}$ and $I' = \{\neg A, C\} \Rightarrow$ the TVA that satisfies Δ

And voila! There's the entire run through of an example of the CDCL algorithm illustrated by the visualization tool I created for this senior project.

5 What's Next

In the future, I would like to further develop these visualization tools and even create ones for my Brute Force algorithm and for DPLL. I would also like to make the structures more variable in size, as well as make the visuals richer. Lastly, I would also like to put this tool on its own webpage so that it isn't just sitting in a GitHub repository, and so that others can see just how neat SAT solving is.

6 Conclusion

Overall, this project was a fantastic experience. Starting with the SAT solving basics, programming my own implementations of DFS, DPLL, and CDCL in Typed Racket, as well as other SAT-related tools, and ending with visualizing the DFS SAT algorithm and the CDCL SAT algorithm using JavaScript. I am grateful to have had the experience to work on a project that I had genuine interest in, and I am looking forward to learning more about SAT solving in the future.

References

Wikipedia. Sat solver. https://en.wikipedia.org/wiki/SAT_solver, 2025a. Accessed: 26 May 2025.

Wikipedia. Boolean satisfiability problem. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem, 2025b. Accessed: 26 May 2025.

Wikipedia. Conjunctive normal form. https://en.wikipedia.org/wiki/Conjunctive_normal_form, 2025c. Accessed: 26 Mau 2025.

Wikipedia. Tseytin transformation. https://en.wikipedia.org/wiki/Tseytin_transformation, 2024. Accessed: 26 May 2025.