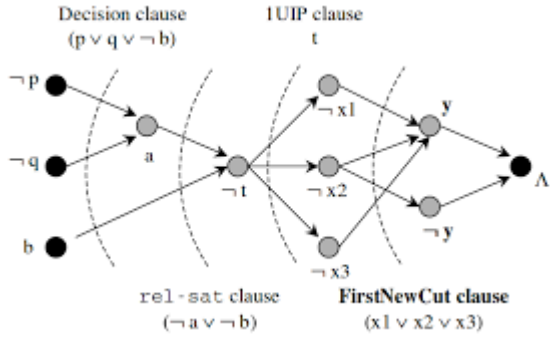
SAT solvers find solutions for Boolean Satisfiability Problems (aka Boolean Expressions) which can express many problems in computer science, engineering, logistics, scheduling, number theory, and many other fields. The Boolean Expressions are transformed into CNF (Conjunctive Normal Form), where smaller Boolean expressions, known as clauses, are all connected with and operators. The clauses are Boolean expressions where variables are all connected with "OR" operators. SAT # solvers are a type of SAT solvers that finds all solutions to the given Boolean Satisfiability Problems. For example:

$$x_1 \wedge (\bar{x}_1 \vee \overline{x_2}) \wedge (x_2 \vee x_3 \vee x_5) \wedge (x_1 \vee x_4 \vee \overline{x_6} \vee \overline{x_7}) \wedge (x_1 \vee x_2 \vee \overline{x_3} \vee x_5 \vee x_7).$$

Currently, the industry stander SAT solvers are based on the CDCL (Conflict Driven Clause Learning) algorithm. Which guess a variable's state (0/False or 1/True). Then CD propagation occurs, and the variables' states are compared to the clauses to find what other variables' states are implied to be true for the full expression to be true. If CD propagation leads to a variable being implied to be both true and false, then CL occurs. CL finds the variables' states that implied the conflict (aka already implied variable) to generate a new clause to avoid that conflict again and backtracks the earliest variable in the new clause.



An alternative to CDCL is the Flash algorithm. The Flash algorithm is like the CDCL algorithm but instead of guessing single variables blindly one at a time, the Flash algorithm solves a cutset of the SAT problem to choose what variables' states to set. Conflict Driven propagation and Clause Learning are also used to expedite the solving. A cutset is a section of the overall SAT problem with a sub-set of variables and clauses from the original SAT problem. Only clauses where every variable in the clause is in the sub-set apply to that cutset.

These cutsets are solved for all possible solutions. These solutions are then modified/reduced to only the variables' states that provide useful information for speeding up the search. Multiple variable groupings are generated and solved to find which has the smallest branching factor. Variable groupings are generated by using a dynamic selection function based on the initial variable selection seed, which are the variables found in each clause. The dynamic selection function iteratively selects clauses with the minimum number of new variables to the current variable grouping.

After variable grouping is generated, the variables are mapped from the variable grouping set to values between 1 and the size of the variable grouping. This format is needed for the cutset SAT # solvers. The mapping of the variables is determined by the frequency of a variable in the cutset's clauses. The greater the frequency of a variable higher of a number it would be mapped too. This results in a significate speed up in the cutset SAT # solvers.

The default cutset SAT # solver used in this program is DTD (Dead Tree Decomposition). The DTD algorithm focuses on calculating the variable state combinations that make a given clause false. It does this for every clause, and if any combinations overlap it skips to the next unique variable state combinations for that given clause to avoid repeating work. DTD is the default solver because it has a very quick and consistent cutset SAT # solver regardless of how many solutions there are to the cutset. There are other cutsets SAT # solvers that can be faster than DTD only when there is a smaller number of solutions. As mentioned above the solutions are modified/reduced to only the variables' states that provide useful information for speeding up the search. This is done by only returning the bits that were in both the low and high values of the bound values. The CD propagation is applied to all reduced solutions generated from the cutset, if there is no conflict then the solution becomes a branch, and the process repeats until all branches die or all solutions are identified.

During this project I did extensive testing on which data structure would work best for the DTD solver. The main concern was speed. The data structure had to hold a large numerical value while being able to access binary bit values quickly. Some of the data structures investigated were arrays of Boolean variables, an array of integers confined to values 0 or 1, Bitset (C++ only), and large integers. It was found that arrays of integers confined to values 0 or 1 resulted in the fastest solutions (C++ testing and more detail about the results can be found on my GitHub).

I also ran many tests using different languages and compilers. For testing, I only focus on the DTD solver. Once again, my main concern was speed. I tested python (basic), python (pypy), C++ (g++), and C++ (C-Lang). I was found that python using pypy was the quickest. Pypy is a JIT (just in time) compiler for python, allowing code to have a comparable speed to non - interpreted languages. The original expectation was C++, an AOT (ahead of time) complied langued, would beat a JIT compiler. C++ ran notifiable slower even with all optimization flags on the compiler turned on. (Testing and more detail about the results can be found on my GitHub). I believe the reason that the JIT compiler was better suited to this task is due to the scale and complexity of the program. JIT compilers, unlike AOT, compilers can perform pogo (profile guided optimization) which only becomes advantageous for large programs. JIT compilers also perform better at pseudo-constant propagation, indirect-virtual function inlining, etc.

This is still an ongoing project and additional improvements still can be made. The main bottle neck of the code is the cutset variable grouping. Four possible solutions are GCN (Graphical Convolutional Networks) and GNN (Graphical Neural Networks), pre-planned grouping, different greedy search algorithms, and coding the search algorithms to be GPU compliant. The next big improvement that could be made is to include clause learning in the algorithm. Allowing the program to learn new clauses from CD propagation conflict and possibly from dead cutset (cutset found to have no solutions). CD propagation can be improved upon by only propagating over the clauses which had a variable recently assigned.

Making the code able to run in parallel would increase the speed of cutset variable group searching significantly. This could be done multiple ways. Either using multi-threading, multi-processing, or making the code able to run on a GPU. Parameter tuning may also improve the speed of the program by selecting the optimal cutset's size, variable grouping algorithm, and SAT # solver for each branch. Finally, a computation time estimator would be a useful addition to the code.