# How to Run

Each algorithm is implemented in three separate python files. I ran Python 3.11.5 on a Windows 11 machine using Anaconda with the following libraries imported:

* time
* pandas
* random
* copy
* *python dpll.py*

By default, each program will look for the hard CNF formulas in the following path from the directory it is currently in:

*./PA3\_Benchmarks/PA3\_Benchmarks/HARD CNF Formulas*

The results for each formula will be printed to the screen and be written to a CSV file that will appear in the program’s parent directory.

# Design Description

## DPLL Description

I chose to represent each CNF formula as a list of lists in python where each sub list represents a clause that contains literals, and the main list contains all of the clauses. Before every algorithm runs, I perform unit propagation and pure literal elimination by continuously iterating through the expression until no pure literals or unit clauses remain.

The function responsible for eliminating pure literals returns an array of all of the undefined literals from the expression that was most recently passed to dpll. When it’s time to actually run dpll, I choose the first literal stored in this array to test. The chosen literal is set to 1 and a new and smaller expression is created with that literal removed (because it has been defined). I then call dpll with that expression to recursively check that branch of the tree. If dpll returns false, the chosen literal is set to zero and dpll is called on that expression instead. Stopping conditions at the beginning of the algorithm return whether or not the expression is satisfiable. This is easy to determine by checking the main list to see if it contains any clauses. If all clauses have been removed, the expression is satisfied.

## Walk SAT Description

Walk SAT is called with an expression, the probability of making a random choice, and the maximum number of flips allowed. I first iterate through the expression and make a list of every unique literal that appears in the expression. I use this to create a new list to hold the assignments (0 or 1) for each variable, that can quickly be indexed by the value of each variable.

First, every variable is randomly assigned the value of 0 or 1. Indices in the assignments list that are not indexed by a variable are wasted space and populated with -1. The algorithm then loops the specified number of times, I chose a maximum of 1000 loops for the hard cnf formulas to prioritize speed over accuracy.

For each iteration until the expression is satisfied or the maximum is reached, a random clause is selected and a random number from 1-100 is chosen. If the random number is greater than the probability of making a random choice that was passed into the function, a literal from the selected clause will be flipped randomly, otherwise the literal in the clause that maximizes the number of satisfied clauses when flipped will be flipped. I chose to run with a value of 50 to give equal probability to both.

A separate function takes the expression and an assignment list and returns two lists containing the satisfied and unsatisfied clauses. This function is called at the start of each loop, and the algorithm can return that the expression is satisfied if the unsatisfied clause list is empty.

If a random literal is flipped, the assignment list is just updated and the loop restarts. Otherwise, for every literal in the random clause, a new assignment list is created with the literal flipped and is passed into the function to apply the assignments. The literal that caused the most satisfied clauses to be returned is flipped in the original assignments array and the algorithm loop restarts.

## GSAT Description

GSAT is called with an expression, a maximum number of attempts, and a maximum number of flips. It uses the same assignment list structure and function to return satisfied and unsatisfied clauses for a given assignment.

For every attempt, each literal is randomly assigned to be 0 or 1. Next the algorithm loops until the maximum number of flips is reached or the expression is satisfied. Each iteration, I go through the expression and make a list containing all of the literals that appear in an unsatisfied clause. For each of these literals, a new assignment list is created with their value flipped. This new assignment is passed with either the list of unsatisfied clauses or satisfied clauses to measure whether flipping the literal has a positive impact. A literal that satisfies the most unsatisfied clauses and disrupts the fewest satisfied clauses is chosen to be flipped in the main assignment list and the loop restarts.

I chose to set both the number of attempts and the number of flips allowed to 100. I found this to give the highest accuracy while testing, although after running it I would choose different numbers to prioritize speed. My design also includes a feature to prematurely terminate an attempt if the same literal is being flipped over and over again (achieving nothing). If this is found to be the case, any literal that has a positive impact on the number of satisfied clauses will be chosen randomly to be flipped. This can happen a maximum of 10 times before the attempt is terminated. This feature is part of the reason my unsatisfied expressions will have different CPU times in the graphs below.

# Results

The following three graphs show how the number of satisfied clauses and the ability of a given formula to be satisfied impact the average amount of CPU time required to evaluate the formula. Times from satisfied formulas are shown in orange, times from unsatisfied formulas are shown in blue.

The first graph below shows the results for a single run of DPLL. The most obvious difference is that satisfiable formulas take less computation time on average than unsatisfied formulas, this makes sense because the search performed on the tree can stop early once a solution is found. Interestingly, CPU time also appears to decrease as a function of the number of clauses satisfied. This could be because harder formulas, which have larger search trees, are also more likely to have a larger number of unsatisfiable clauses.

The second graph below shows the results for ten runs of the Walk SAT algorithm. The accuracy of the conclusion of satisfiability over these ten runs was 92%. If I had run the algorithm longer, this accuracy would have increased and approached 100%. Unlike DPLL, the average CPU time is much more uniform. This is expected since the search is artificially limited and semi-random, adding additional clauses does not drastically expand a search tree the way DPLL does. Walk SAT was also faster than DPLL on average.

The third graph below shows the results for approximately five runs of the GSAT algorithm. Unfortunately, time constraints and a mistake in data handling lost a large amount of data. Despite that limitation, the algorithm had an overall accuracy of 96% across those five runs. Like with Walk SAT, the average CPU time has a more even distribution that is not related to the number of satisfied clauses. As with all three of the algorithms, the average computation time was longer for unsatisfied formulas than for satisfied ones. For Walk SAT and GSAT, unsatisfiable formulas will always run until the timeout limits I established. GSAT is slower than Walk SAT, likely because I programmed it first and was not remotely conservative about resource usage.

# Learning Outcome

The biggest lesson I took away from this project is how implementing an incomplete but clever algorithm can often be a better approach than implementing a complete one. For example, my DPLL algorithm took about two hours to run a single time and produced an absolute result of which formulas were satisfiable and which were not. Meanwhile, my Walk SAT algorithm had a relatively low chance of finding a satisfiable solution for each formula, but it only took about an hour to run it a total of ten times.

The key difference between these two algorithms is that the speed of Walk SAT is not as significantly impacted when you have more clauses to check, it will take about the same amount of time per formula no matter what. Therefore, if I wanted to check an even harder set of CNF formulas, it might be more efficient to just run Walk SAT over and over to approach the correct solution rather than to spend the time to find it explicitly using DPLL.

This lesson also gives a clue about how to approach real world problems that are either too difficult or too impractical to solve completely. In many circumstances, like modeling weather or creating a chess bot, coming up with an algorithm that approaches 100% correctness is more useful than one that actually is 100% correct.