# 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

# 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.