(i) provide a pseudo-code description of the basic algorithm (DPLL) and all the major heuristics implemented (BCP, conflict clause learning, non-chronological backtracking, etc.)

Before we get to the pseudocode, we want to describe the algorithm and each heuristic in detail.

In our project we implemented DPLL, CHAFF BCP, DLIS, and our own idea which is Unate branching.
We implemented **DPLL** as a recursive algorithm. DPLL encompasses these three heuristics and has many helper functions to help do its job. It handles all the branching (updating solution, clauses) and backtracking (undoing solution and clause modifications).
We considered passing variables by value instead of reference to reduce backtracking time. Although that was achieved, the overall time to solve was lengthened due to having to copy massive amounts of data between DPLL calls, so we went back to passing by reference which required undoing changes for backtracking.

We had our own twist of **CHAFF BCP**. Instead of using watch variables which required extra overhead, we instead made note of each clause when it became a unit clause. Upon the next call of DPLL, we would then branch on it. This way we still had the entire effect of branching on unate clauses while keeping overhead to a bare minimum. We did this by checking the number of unassigned variables in a given clause and whether that clause was not yet satisfied (see part ii below for more details).

We also applied **DLIS** in our own way. At the start of the program, while parsing the .cnf file, we increment counters of each literal (a, a', b, b', …) of how many times they occur. Instead of recounting literals on every decision, we simply updated the counters whenever a clause was satisfied (or un-satisfied upon backtracking). Upon deciding, we check the most common **variable** (sum of complemented and uncomplemented occurences) that has not been assigned yet. Then we branch on that variable's most common literal. If this fails we then branch upon it's complement (say a' is more common then a, we branch on a', if it fails then we branch on a). We found this implementation of DLIS effective and it removed applying the same operations repeatedly in the original DLIS method that only based branching on the literal rather than variable count.

Lastly, we came up with our own heuristic, which we called **Unate branching**. It uses the counters we made in the DLIS heuristic (see above). At any point in the algorithm, when there occurs a variable that has not been assigned and one of its literals does not appear (in the remaining unsatisfied clauses) while its other literal does appear (say count of a = 0 but count of a' = 3)' we would then branch on the literal that does appear. Since the variable is unate in the remaining unsatisfied clauses, it only speeds up finding a SAT solution or that the current subsolution is UNSAT. We found this to be very efficient and a good way to further utilize DLIS' overhead and apply concepts we learned in Unit 3 of this class!

Pseudo-code:
Note: This is the pseudo-code of the DPLL function and the heuristics mentioned above, not the many helper functions we used to implement these parts of the algorithm.
Color code:
Entire pseudocode = DPLL
CHAFF BCP (green)
DLIS (blue)
Unate branching (yellow)

Input explanation:
DPLL takes all clauses (vector of all Clauses); curr_solution (vector of current variable assignments), uncomp (vector of vector containing clause #s that contain each uncomplemented literal), comp (vector of vector containing clause #s that contain each complemented literal)

```
int DPLL(all clauses, curr_solution, uncomp, comp) {
        While (∃ unit clauses || ∃ unate variables) {
                while(∃ unit clauses ) {
                        If (unit_handle_duplicates() == -1) backtrack; // finds unit clauses;
backtracks on conflicts
                        For ∃ unit variable :
                                update_solution()
                                If (update_clauses() == -1) backtrack;
                }
                find_unate() // finds all unate variables
                While (! ∃ unit clauses && ∃ unate variables) {
                        // never causes a conflict since unate
                        For ∃ unate variable :
                                update_solution()
                                update_clausse()
                }
                find_unate() // finds all unate variables
        }

        If (! ∃ unsatisfied clauses) return SAT

        // figure out what to branch on
        Variable = DLIS_choice()
        update_solution(variable)
        If (unit_handle_duplicates() == -1) backtrack uncomplemented Variable + skip to
complemented section;
        If (DPLL(clauses, solution, uncomp, comp) == 0) return SAT
        Else backtrack uncomplemented variable
```

```
        // here: use compliment of Variable
        update_solution(variable')
        If (unit_handle_duplicates() == -1) backtrack complimented Variable + skip to backtrack
big while loop section;
        If (DPLL(clauses, solution, uncomp, comp) == 0) return SAT
        Else backtrack complemented variable

// backtracking the while loop changes
        backtrack (unit variables)
        Backtrack (unate variables)
        Return -1;
}
```

(ii) explain the data structures used and justify your choices

```
struct Clause {
    vector<int> literals;
    vector<bool> assigned_literals; // keep track of which literals have an assigned value
    unsigned unassigned = 0; // count of unassigned variables // might not be useful if we go for the map BCP
    bool is_satisfied = false; // whether clause is satisfied

};
```

We made a struct "Clause" and within it are a vector of literals, a vector of booleans to mark which variables have been assigned, # of unassigned variables, and a boolean for whether a clause is satisfied.

In our functions we use:
Vectors of clauses
Vector holding the solution
Vectors of vectors for un/comp clauses - for each literal stores which clause #s it exists in
An example:

```
int DPLL(vector<Clause> & clauses, vector<int> & solution, const vector<vector<unsigned>> & uncomp, const vector<vector<unsigned>> & comp, bool & sat);
```

We originally considered maps (int, vector<Clause>); where each mapElement.first referred to # of unassigned variables in that clause, and mapElement.second had a vector containing all clauses with that number of unassigned variables; however: this consumed a lot of memory and made backtracking INCREDIBLY COMPLEX. Using Vectors for our data structure was clearly the best option.

We made Clause a struct instead of a class since we didn't need member functions nor inheritance for this application.

(iii) describe the experimental methodology, benchmarks used to evaluate your implementation, and the results obtained

We used this site: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html for our benchmarks found in the Project Overview.
We used all of the following cases in our testing and results section:

- **Uniform Random-3-SAT**, phase transition region, unforced filtered - description (html)
  - uf20-91: 20 variables, 91 clauses - 1000 instances, all satisfiable
  - uf50-218 / uuf50-218: 50 variables, 218 clauses - 1000 instances, all sat/unsat
  - uf75-325 / uuf75-325: 75 variables, 325 clauses - 100 instances, all sat/unsat
  - uf100-430 / uuf100-430: 100 variables, 430 clauses - 1000 instances, all sat/unsat
  - uf125-538 / uuf125-538: 125 variables, 538 clauses - 100 instances, all sat/unsat
  - uf150-645 / uuf150-645: 150 variables, 645 clauses - 100 instances, all sat/unsat
  - uf175-753 / uuf175-753: 175 variables, 753 clauses - 100 instances, all sat/unsat
  - uf200-860 / uuf200-860: 200 variables, 860 clauses - 100 instances, all sat/unsat
  - uf225-960 / uuf225-960: 225 variables, 960 clauses - 100 instances, all sat/unsat
  - uf250-1065 / uuf250-1065: 250 variables, 1065 clauses - 100 instances, all sat/unsat

We tested all of the above 6399 benchmarks and passed 100% of them.
Our final version ran all of them.
Our initial version (mentioned further in Results) became incredibly slow past 150 (though it would pass them, it may have taken several days for all the benchmarks 250 variables).

Note: 20 variables 91 clauses only had SAT benchmarks, no UNSAT benchmarks. The rest have both SAT and UNSAT.

Results

We had two versions of our SAT solver.
The first version implemented DPLL and CHAFF BCP.
The second (final) version further implemented DLIS and Unate branching on top of the first version.
We **did not** have a version that **only** had DPLL. The reason is because DPLL alone is far too slow past 50 variables, so we decided to implement one heuristic alongside it. In hindsight, we should have only done DPLL first as that would have reduced the difficulty of debugging DPLL significantly. Our debugging of the first version included debugging of DPLL and BCP at once which made that challenging.

In this section we compare the first version and the second version of the project. Tables and graphs are titled “BCP Only …” or “All Heuristics …” to distinguish the first from the final version of our project.

Our tables report the maximum, minimum, mean, and median run times.
To reduce the impact of outliers, our graphs display only the median run times.

Lastly before getting to the results, our first version was too slow to complete 175+ variables so we only took the data for 20-150. It would pass the larger CNFs but would take *incredibly* long. Our final version passes all of them with speed, so we have all benchmarks of that version included!

Note: Runtimes do not include the time it took to run our homemade parser. However, this time was negligible. Even the largest CNF files (250 variables, 1065 clauses) took at most 0.003 seconds to parse. Therefore, runtimes are only made up of the algorithm runtime (from completion of parse to returning SAT/UNSAT).

Table 1. BCP Only - Runtimes in seconds on SAT Benchmarks

# Variables - # Clauses	MAX	MIN	MEAN	MEDIAN
20-91	3.73E-04	1.43E-05	8.00E-05	6.42E-05
50-218	0.0287676	0.00004754	0.003307986683	0.00213686
75-325	0.284778	0.000657095	0.05686943709	0.0388557
100-430	14.7962	0.000134844	0.8729614439	0.448873
125-538	227.075	0.000305504	10.76350706	4.12283
150-645	623.608	0.176603	98.52053809	38.281
175-753	—	—	—	—
200-860	—	—	—	—
225-960	—	—	—	—
250-1065	—	—	—	—

Table 2. All Heuristics - Runtimes in seconds on SAT Benchmarks

# Variables - # Clauses	MAX	MIN	MEAN	MEDIAN
20-91	1.47E-04	1.39E-05	3.65E-05	2.55E-05
50-218	0.00293119	0.000045796	0.000437734498	0.000289484
75-325	0.0142135	0.000087584	0.00262295793	0.00165962
100-430	0.103567	0.000122541	0.01338220803	0.00754917
125-538	0.476919	0.000249018	0.08431459452	0.0551328
150-645	1.60915	0.00149464	0.3038949173	0.158569
175-753	13.1443	0.000394391	1.416601568	0.664528
200-860	56.1974	0.00588604	8.419698067	5.72324
225-960	140.58	0.0020372	22.19976325	8.25611
250-1065	701.405	0.0305682	106.7612237	47.5223

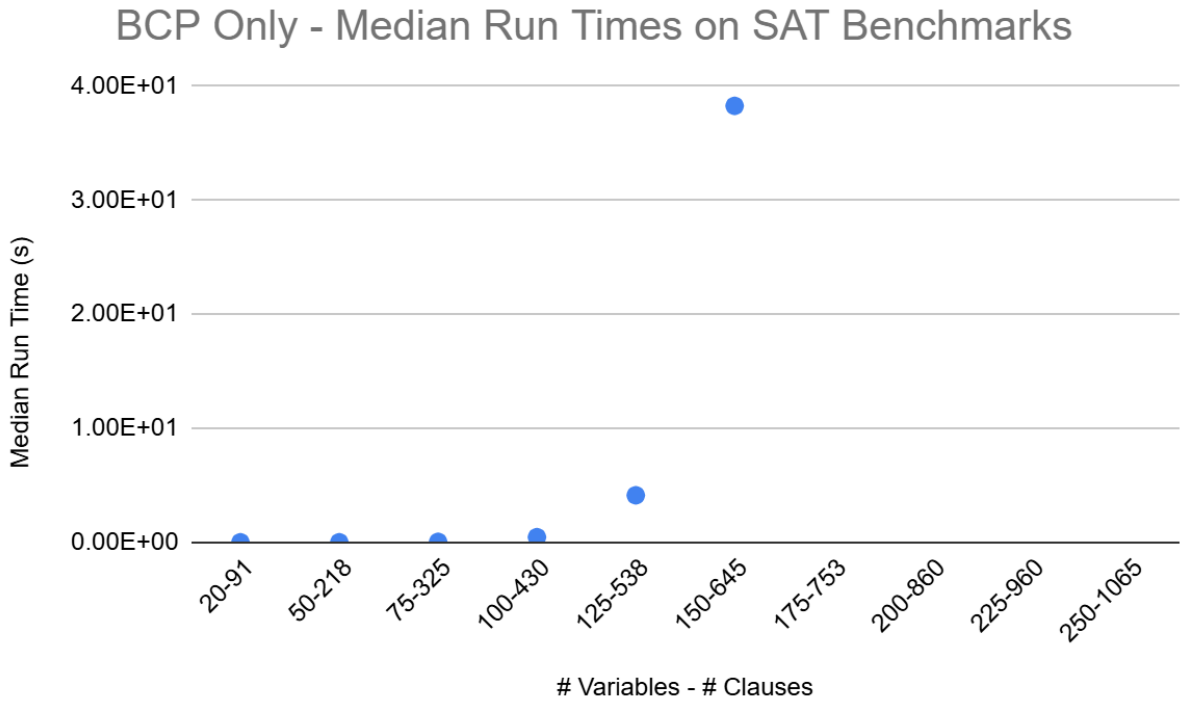
Table 3. BCP Only - Runtimes in seconds on UNSAT Benchmarks

# Variables - # Clauses	MAX	MIN	MEAN	MEDIAN
20-91	N/A	N/A	N/A	N/A
50-218	0.0604721	0.00118021	0.00832736633	0.00710716
75-325	0.410407	0.0188474	0.123230701	0.100803
100-430	23.9167	0.102291	2.258021853	1.59537
125-538	134.881	2.00338	25.7110653	20.4425
150-645	2778.63	19.2898	329.894108	187.65
175-753	—	—	—	—
200-860	—	—	—	—
225-960	—	—	—	—
250-1065	—	—	—	—

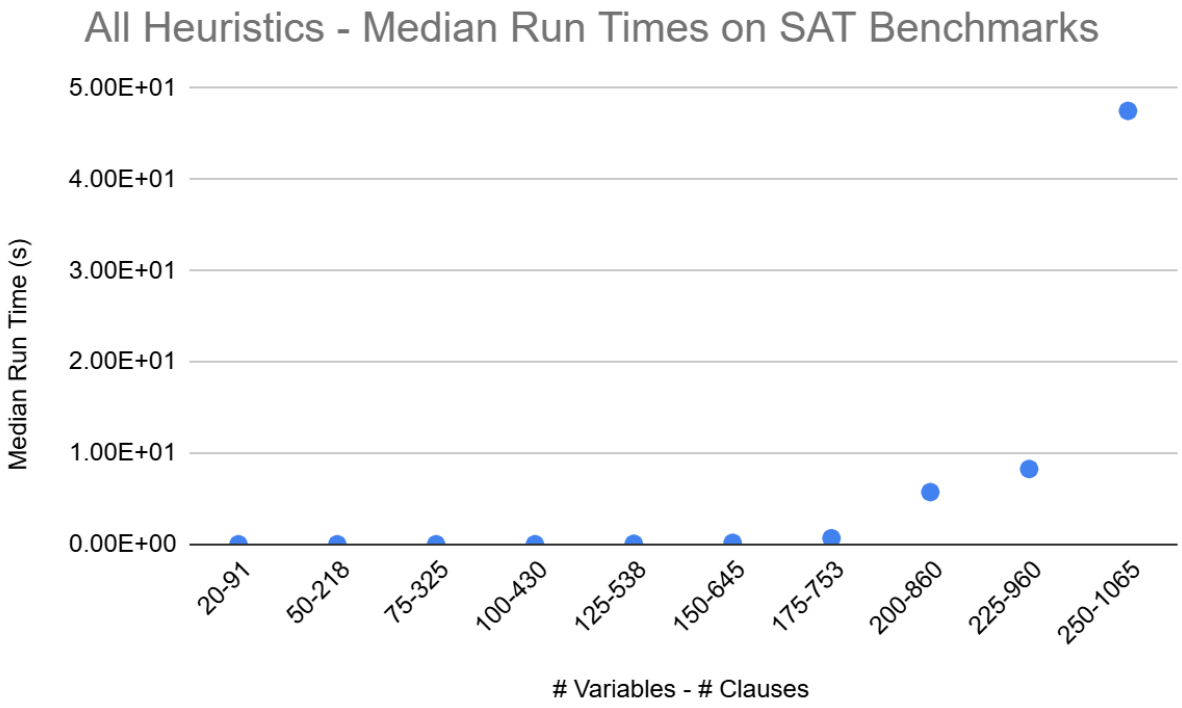
Table 4. All Heuristics - Runtimes in seconds on UNSAT Benchmarks

# Variables - # Clauses	MAX	MIN	MEAN	MEDIAN
20-91	N/A	N/A	N/A	N/A
50-218	0.00354868	0.000368382	0.001292051276	0.00121768
75-325	0.0164374	0.00320569	0.0078220645	0.00743252
100-430	0.136448	0.00884739	0.04392721429	0.0405293
125-538	0.700592	0.0490665	0.229994716	0.202027
150-645	5.53575	0.212215	1.0570973	0.828269
175-753	21.3295	1.00088	4.612633	3.90961
200-860	51.7233	3.27576	20.16662848	18.42065
225-960	302.177	16.9911	96.542992	83.8528
250-1065	1836.5	29.8907	429.322217	355.578

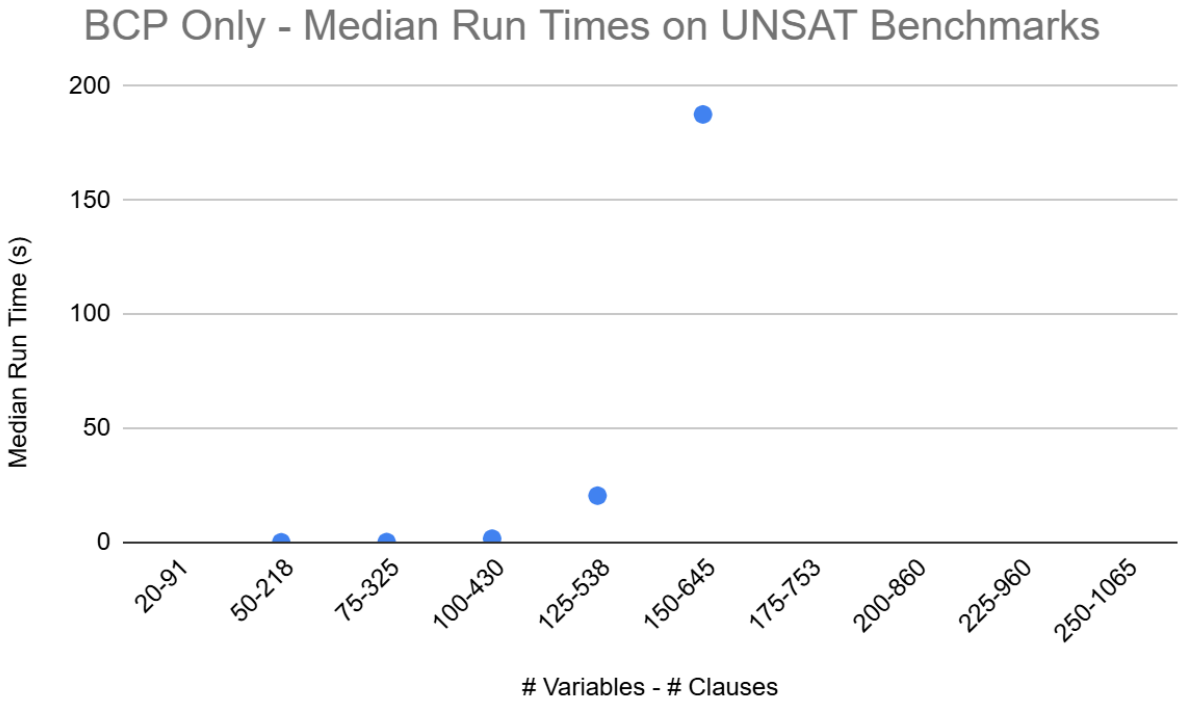
Graph 1. BCP Only - Median Runtimes in seconds on SAT Benchmarks (Table 1 data)



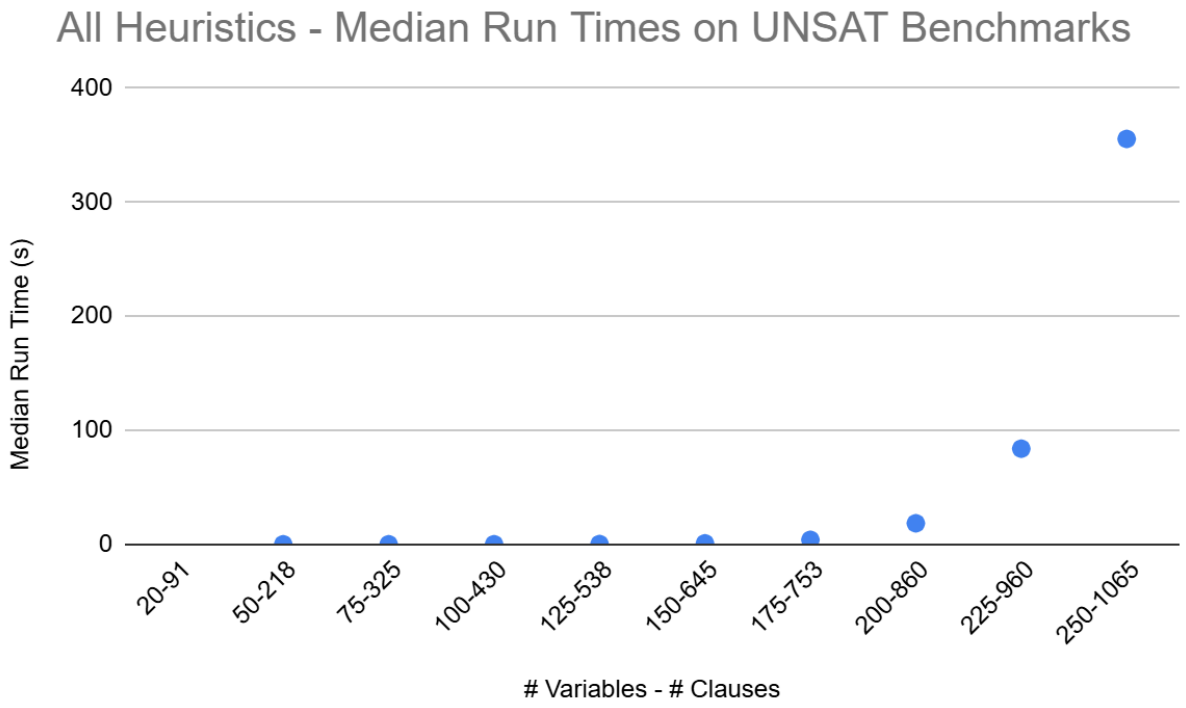
Graph 2. All Heuristics - Median Runtimes in seconds on SAT Benchmarks (Table 2 data)



Graph 3. BCP Only - Median Runtimes in seconds on UNSAT Benchmarks (Table 3 data)

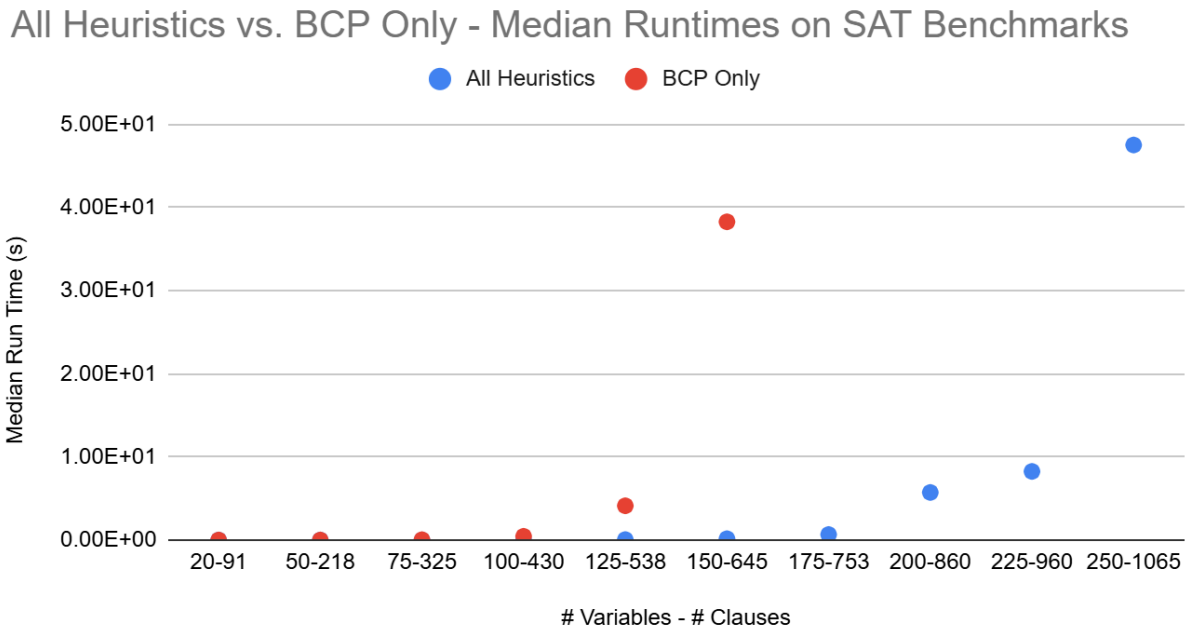


Graph 4. All Heuristics - Median Runtimes in seconds on UNSAT Benchmarks (Table 4 data)

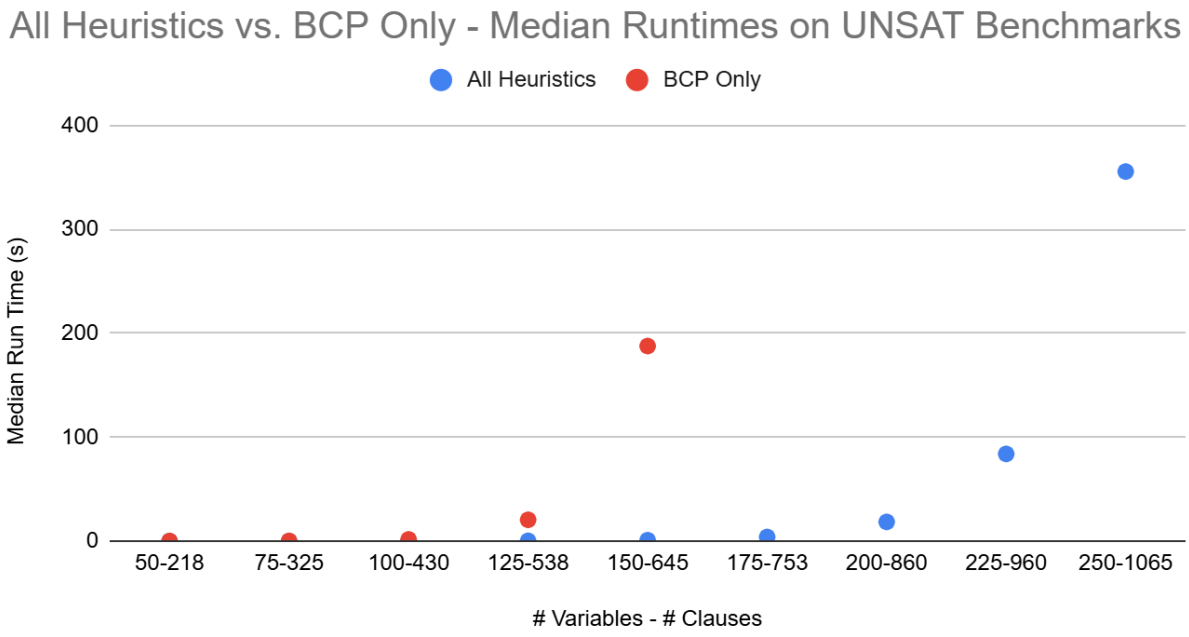


Now to compare the speedup from our first version to our final version (for SAT and UNSAT), we plot Graph 1 and 2 onto Graph 5 and Graph 3 and 4 onto Graph 6 below:

Graph 5. All Heuristics vs. BCP Only - Median Runtimes in seconds on SAT Benchmarks (Table 1, 2 data)



Graph 6. All Heuristics vs. BCP Only - Median Runtimes in seconds on UNSAT Benchmarks (Table 3, 4 data)



(iv) analyze the results (e.g., how does the run-time of your implementation vary with problem size, what is the effect of turning on/off each of the heuristics, etc.)

Comparing our program's benchmarks in the graphs above truly shows how much faster it became with more heuristics. Remember, the first implementation of our program had BCP as well as DPLL. So, if we tested with just DPLL, then we would see an even larger run-time and thus an even greater speed-up.

Take for example the largest case that ran (in reasonable time) on our first implementation. At 150 variables, 645 clauses: it ran at a median of 38.3 seconds (for SAT) and 187.7 seconds (for UNSAT). The final implementation of the algorithm (now with DLIS and unate branching heuristics) ran this same case at a median of 0.16 and 0.83 seconds. That's over 250 and over 225 times speed up, respectively! That shows how powerful these heuristics are!

From the graphs, it is evident that UNSAT expressions take far longer to complete. This makes sense since as soon as a solution is found, SAT is returned instantly, whereas if a solution is not yet found, the algorithm continues to search for one until (a) a solution is found or (b) all possibilities are reached and no solution is found, thus returning UNSAT.

Furthermore, it is clear from the graphs that the **runtimes grow exponentially** as the CNF gets larger both in terms of the number of variables and the number of clauses. This is also as expected, as there is more memory and time overhead for updating values, as well as larger CNFs will generally go deeper into DPLL's recursion.

In our experiments, enabling each of the implemented heuristics consistently improved runtimes. Each heuristic we chose handled another heuristic's weaknesses.

Unit clause propagation was written based on CHAFF BCP we learned in class. It was effective in early backtracking upon seeing a unit clause conflict which saved many recursive calls. It also quickly reduced the unsatisfied clause count as each unit variable was a necessary decision. It also cost very little overhead as the unit clauses were collected upon updating clauses, which is something DPLL must do anyways.

DLIS was added to make smart decisions when we were forced to branch. Before implementing DLIS we simply would choose the next unassigned variable to branch upon which was very inefficient. After implementing DLIS, the solver was guided to quickly reduce the unsatisfied clause count early in recursion, therefore reducing overhead in future recursive calls.

Unate branching further improved runtime by enhancing both DLIS and BCP. It used data that was already gathered by DLIS, so overhead was minimal. It also made free decisions that quickly reduced a subsolution's set of remaining unsatisfied clauses. This allowed for a subsolution to return a SAT solution faster as well as whether that subsolution was even valid at all so as to start searching for other subsolutions earlier. It also led to unit clauses being found earlier which improved BCP.

(v) summarize the significant conclusions that can be drawn from your results.

The DPLL algorithm determines the satisfiability of a CNF formula by returning SAT and a satisfying assignment or UNSAT after an exhaustive search. In its most basic form, DPLL has a worst-case runtime of O(2^n), as such an implementation simply is a brute-force binary search.

To address this and reduce runtime, heuristics are added to guide the search process and reach the SAT/UNSAT result sooner. However, each of these heuristics incur their own overhead. Too few heuristics will not benefit the runtime by much and adding too many could degrade performance due to excessive processing and overhead. Adding the right amount will give maximum speedup.

We chose to integrate three heuristics: CHAFF BCP, DLIS, and unate branching. Our results section showed that each heuristic addition reduced runtime considerably.

Benchmarks for UNSAT CNFs are consistently slower than those for SAT CNFs no matter the CNF's size. This is because as soon as a SAT solution is found, the algorithm ends. Whereas, the algorithm will run an UNSAT expression until it exhausts all cases.

It seems that it is useful to apply heuristics that cover each others' weaknesses as well as that expand on each other:

- BCP performs unit propagation at the cost of not making decisions

- DLIS makes decisions at the cost of not having the fine tuning of BCP or Unate branching
- Unate branching enhances both by identifying guaranteed-valid assignments earlier and leveraging DLIS data

Overall, this heuristic combination proved to be synergistic and very likely more effective than, for instance, implementing DLIS and VSIDS alone which do mostly the same thing and only cover each other's weaknesses slightly.

We conclude that a careful selection of heuristics is key to optimizing DPLL performance and writing a well-rounded algorithm for a SAT solver!

(vi) AI Usage Disclosure

1. Parser

For the code, AI (ChatGPT-4) was used for help on writing the parser.
This was **only** because we were allowed to copy/paste and use MiniSAT Parser directly. However, when going through MiniSAT's parser, we got lost in the legacy code and decided to write our own parser. Our parsing skills are not the strongest specifically with sstream library, so we needed help as without a perfect parser, the entire program fails.
The parser consists of ~90 lines of code only around 35 of which were written by AI (just the sstream part). The rest of the parser code that updated different data structures was written entirely by us.

The prompt given was: "SAT DIMACS in C++ questions" and a mostly correct parser was provided.

2. Report language

For the last section of part iv and part v, AI (ChatGPT-4) was used to make what we wrote more concise. To be clear, we had written those sections already but were far too verbose.

The prompt given was: " <given sentence/paragraph> please make this more concise"