1 DPLL

1.1 Algorithm

The first algorithm that we implemented was vanilla DPLL. In our initial implementation, we created a `SATInstance` object with a set `Clauses` of integers, a map `VarCount` from *variable* to *positive counts/negative counts*, a set of `UnsatisfiedClauses`, an `AssignmentStack`, and a `ClauseStack`.

In this implementation, each time we make a branch we deep copy our `SATInstance` object and perform unit propagation and pure literal elimination. When we branch, we do not directly set any variable to True or False, we simply add a unit clause indicating our guess for the particular variable. Each time we backtrack, we revert to the previous copy of the state rather than making individual changes. In this implementation, we consider a CNF SAT once all the clauses have been removed and consider it UNSAT if we traverse the whole tree and all are unsolvable. Something is unsolvable if there is an empty clause in the CNF.

This implementation solves all but three of the CNFs with the RDLIS branching heuristic. However, the memory overhead of deep copying at each branching decision is significant. Moreover, we iterate through every clause and check it's length for this implementation which could be slow if there is a good amount of clauses. This is a linear time in the number of variables in a clause per clause. We next tried implementing Two Watched Literals to decrease the runtime complexity Tichy and Glase (2006).

Heuristics

We tried multiple heuristics for picking the splitting variable, DLCS, DLIS, RDLCS, and RDLIS. These were relatively simple to implement because we keep track of variable counts as we delete clauses and resolve variables. We decided that the overhead for some of the other heuristics would likely cause significant slowdowns.

1.2 Two Watched Literals

Our implementation of two watched literals required significant modification of the code. This modification hinged on the assumption that as long as we have 2 variables that are Unassigned, it is possible to for a particular clause to be solvable. We added a map of assignments `Vars`, a map from watched literal to clauses `LiteralToClause`, and a map from clause to the two watched literals `WatchedLiterals`. We stopped deep copying whenever we branched. The backtracking logic was reduced significantly and we only did pure literal elimination before any branching. However, this modification to the logic invalidated our heuristic, so we experimented a bit with extremely naive (choose based on initial variable counts) heuristics. In addition, we implemented restarts based on assignment history depth Chaff (1999).

1.3 Profiling

We achieved correctness, but did not see any runtime gain with two watched literals which surprised us. This suprised us because for each clause, we would no longer need a linear runtime in the count

of the variables unless we are moving the watched literal. One possible reason for no speedup in certain CNF examples is because all of the clauses are pretty short. Others are faster with two watched literals, possibly because there are very long clauses.

We profiled with **pprof** in order to better understand the execution timeline. We did not see any extraneous calls. Most of the time was spent moving watched literals and getting elements from the set. This figure is located in fig. 1.

2 CDCL

We attempted to implement several versions of CDCL, all trying to find the first Unique Implication Point (UIP). We did this by constructing an implication graph from the values we assigned to each variable. We also kept two maps of lists, for both the branching history and the propagation. Once we reach a conflict clause, we construct a queue of literals from the conflict clause. We perform a breadth first traversal to get the parents of each node in the queue, check if the parents are in the current level or a previous level and add them to the queue if they are in the current level, or add to the learned clause if they in a lower level Junttila (2020).

Then, until the queue of literals is empty we backtrack variable truth assignments by first finding the max level in the learned clause that is before the current level we are at. We unassign variables that are assigned in a level that is greater than this found level and we remove them from the implication graph.

Heuristic	Total Time	Instances Failed
DLCS	645.87 (1200.87)	2
DLIS	646.71 (946.71)	1
RDLCS	326.62 (1226.62)	3
RDLIS	601.02 (1501.02)	3

Table 1: A 4 rows by 3 columns table.

3 Conclusions

In total, our initial DPLL implementation seemed to perform the best. If we had more time, we would finish debugging CDCL to ensure correctness as this seemed like the most viable path to a more efficient algorithm. In addition, we found that the go overhead was higher than we originally thought. Overall, we were slightly disappointed that our initial implementation worked the best; however, we still felt that we learned much from this project!

4 Appendix

References

Chaff. Engineering an efficient sat solver. *IEEE Transactions on Computers*, 48:506–521, 1999.

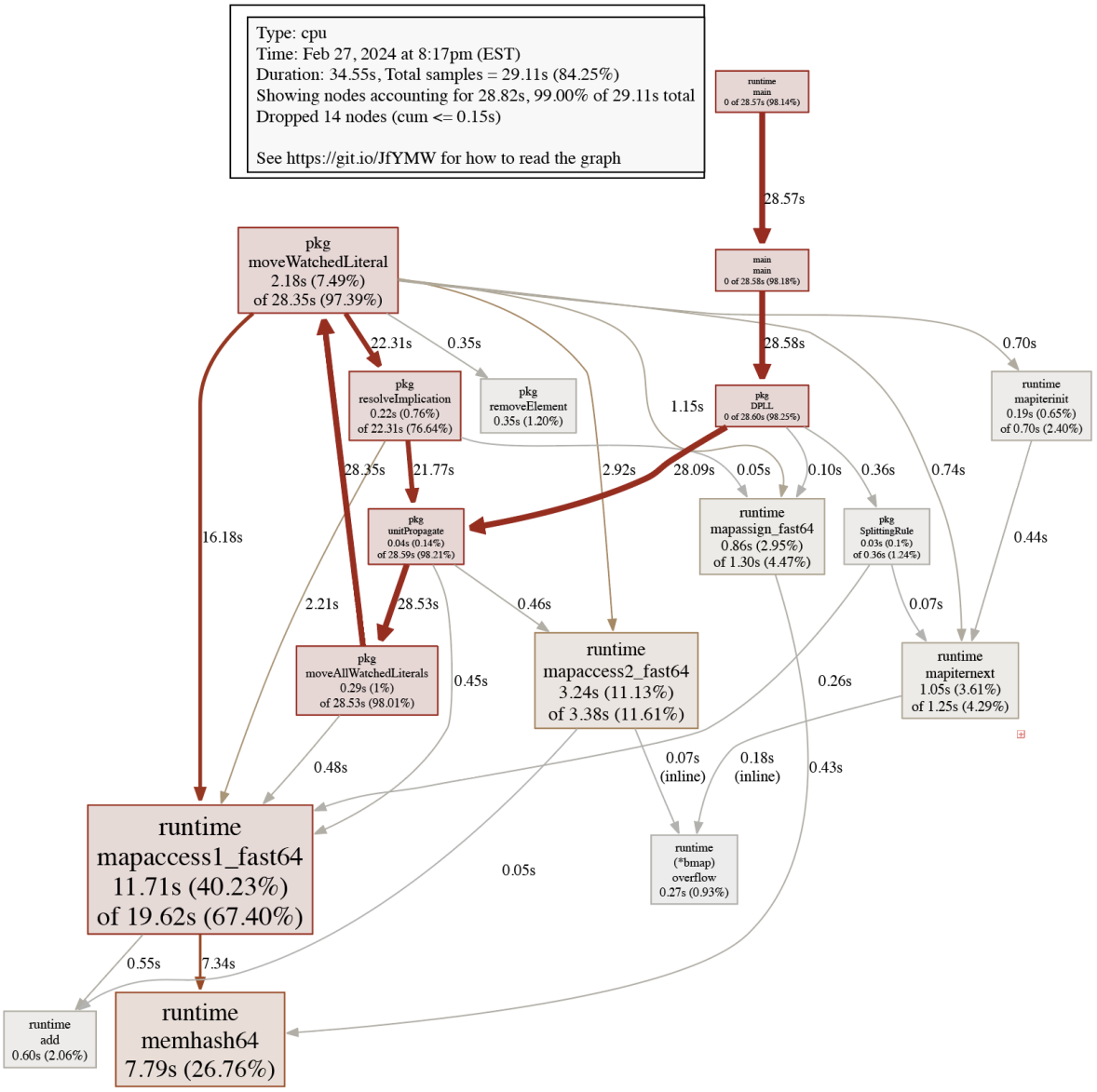

Figure 1: Profiling of the Two Watched Literals generated using PProf

Tommi Junttila. Conflict-driven clause learning (cdcl) sat solvers - cs-e3220: Propositional satisfiability and sat solvers, 2020. URL https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html. Accessed: February 2024.

Richard Tichy and Thomas Glase. Clause learning in sat. *Faculty of Computer Science Clause Learning in SAT Seminar Automatic Problem Solving WS 2005/06*, 2006.