Optimizing DPLL with Adjacency Lists, H/T Lists, and Watched Literals

Sophie Andrews
sophie1@stanford.edu

Colin Schultz
colinrs@stanford.edu

***Abstract*—The Boolean satisfiability or SAT problem is fundamental to many interesting problems in computer science. While solving the problem in general is known to be NP-Complete, there is a lot of value in producing faster SAT solvers that can solve practical problems in less time. In this work, we implement a baseline SAT solver based on the DPLL algorithm, and then improve its performance with three different optimization techniques: adjacency lists, Head-Tail lists, and watched literals. We compare the effects of the optimizations on solver runtime for toy and practical benchmarks. Each optimization provides a speedup over the previous implementation.**

The code for our solver can be found at https://github.com/colin-r-schultz/CS257-Final-Project.

I. Introduction

Solving a SAT formula is an important technique that can be used in many areas such as program verification or artificial intelligence. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is one procedure for solving a SAT formula. Many modern SAT solvers expand on the DPLL procedure.

The DPLL algorithm works by applying three actions until either a model is found which satisfies the SAT formula or the formula is found to be unsatisfiable. The first action is **decide**, where an arbitrary unassigned variable is given an arbitrary assignment. When this rule is applied, it is important to note this arbitrary decision with a *decision point*. If this arbitrary decision ultimately leads to a conflict, the decision point allows us to determine which choice was incorrect.

The next rule is **propagate**. Executing this rule involves finding unit clauses (also called *implied* clauses) where no literals are assigned to true and only one literal is unassigned. In this case, the single unassigned literal must be assigned true in order to satisfy the clause. This rule is applied after every assignment, and may introduce zero or more additional assignments. Any assignments resulting from *implied* clauses will trigger this rule to be run again, until a conflict emerges.

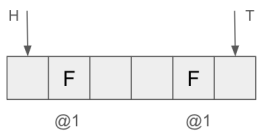
Conflicts occur if the propagate rule finds an implied assignment that is the opposite of a previously made assignment. In this case, we apply the third rule **backtrack**. This rule undoes all assignments that follwed the previous decision point. Then the decision associated with that point is reversed and the decision point is removed. If the solver attempts to backtrack while there are no previous decision points, then the formula is found to be unsatisfiable.

In this project, we explore several optimizations for performing the propagate rule [4] [5]. The naive approach for propagation is to use a linear search to find which clauses contain the newly assigned literal and update them accordingly. But by using three different data structures, we can perform propagation more efficiently.

II. Background

The first optimization is adjacency lists. Adjacency lists were introduced by the GRASP solver [6], and work by maintaining a list for each literal which contains the clauses in which the negation of the literal appears. Now when we assign a literal, instead of doing a linear search for the clauses with the opposite of the literal, we can immediately find all relevant clauses. For each literal, we care about clauses with its negation since those are the clauses that now have an additional literal assigned false and may become unit clauses. The next two optimizations build on top of the adjacency list implementation.

The second optimization is using a Head-Tail data structure. The Head-Tail data structure maintains two pointers into each clause, allowing more efficient backtracking and discovery of unit clauses. With the Head-Tail data structure, we start by maintaining a pointer to the first and last literal of a clause. If one of the head or tail literals is assigned false, we move the respective pointer towards the center of the clause until it reaches a satisfied or an unassigned literal or the other pointer. If the pointer location is now a satisfied literal, then the whole clause is satisfied. The pointers stop moving when they have both moved to the same literal. Compared to just using adjacency lists, the use of pointers into each clause means that instead of traversing a clause to determine if it is unit, we can just check to see if the head and tail point to the same location. It also means that we can more efficiently backtrack to a state with unassigned literals. However, one downside is that we need to store the state of the head-tail pointers at each decision point to make sure each clause is in the correct state when we backtrack. The SATO Solver [8] is an example of a solver using the Head-Tail data structure.



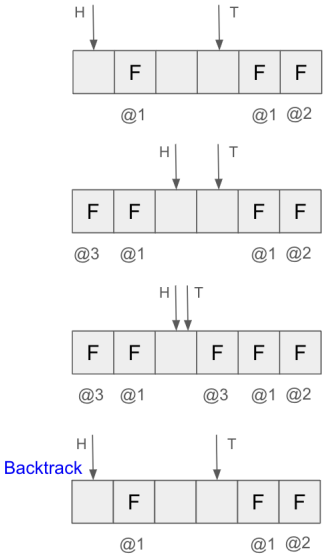
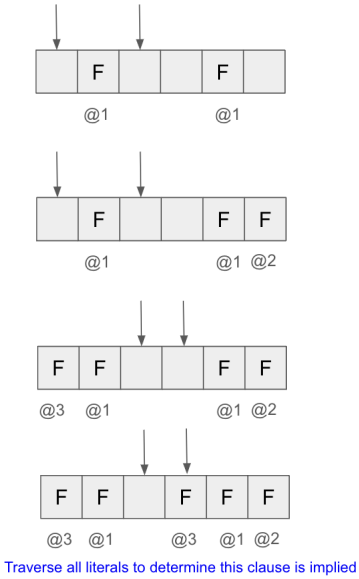


Fig. 1. Schematic showing the movement of H/T pointers on a clause. Boxes with F represent a literal assigned false. Blank boxes are unassigned literals.

This third optimization uses watched literals. Like the Head-Tail structure, the watched literals strategy greatly decreases the amount of work needed to find implied clauses. It does this by similarly maintaining two pointers into each clause, so that as long as both literals are unassigned, the clause cannot be implied. If the clause is implied, then one of the watched literals is assigned to false and the other is the implied variable. Unlike the Head-Tail structure, the watched literals strategy depends on fewer invariants about the pointers into each clause. One benefit of this is that there is no ordering to how the pointers move which means we do not have to update any of the literal references when backtracking. One downside is to identify a unit clause, we must traverse all of a clause's literals. Watched literals were introduced by the Chaff solver [7].



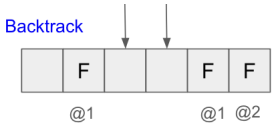


Fig. 2. Schematic showing the movement of Watched Literals on a clause. Boxes with F represent a literal assigned false. Blank boxes are unassigned literals. Compared to H/T lists, the backtrack step is easier.

III. Methodology

We began by creating a baseline implementation of DPLL. We chose to use an object-oriented approach in C++ so that we could override only specific features of the baseline algorithm in order to implement each optimization. This allowed for a fairer evaluation of the impacts of each individual optimization.

Our implementation relies on inputs in the DIMACS CNF format. We rely on parsing the number of variables out of the `p cnf` header line and we assume that each clause will be on its own line that terminates with a 0, and each clause is made of space-delimited literals. In each implementation, our first step is to handle any unit clauses in the original formula and assign those literals to true. Our subsequent implementation of the solving procedure is described in pseudocode below:

```
solve():
  loop:
    while there are pending assignments:
      a = next assignment
        if a conflicts:
          if previous decision point:
            backtrack()
          else:
            return UNSAT
          R = findRelevantClauses(a)
          for c in R:
            a' = getImpliedLiteral(c,a)
            if a':
              add a' to pending
      if there are unassigned variables:
        decide()
      else:
        return SAT
```

- `backtrack()` resets the variable assignments made after the most recent decision point, inverts the decision associated with the decision point, and then removes the decision point.
- `findRelevantClauses(a)` finds all clauses that may become *implied* as a result of assignment `a`.
- `getImpliedLiteral(c,a)` returns the literal that is known to be true in clause `c` due to assignment `a`. This only returns a literal if `c` is now *implied*. If `c` is already satisfied or has multiple unassigned literals, this function does not return anything.
- `decide()` finds an unassigned variable and assigns it arbitrarily, introducing a new decision point. We know

our formula is SAT if all variables have been assigned and we have not reached a conflict.

A. Baseline Solver

The baseline solver was created to be very naive to allow for future optimizations. The implementation of `findRelevantClauses(a)` performs a linear search over all of the clauses and returns the subset which contain the negation of the literal represented by `a`. Only these clauses may possibly become *implied* as a result of `a` being assigned.

The `getImpliedLiteral(c,a)` method scans all of the literals in `c`. If all but one literal are false, then the remaining literal is implied. If any literals are true or more than one literal is unassigned, then there is no implied assignment from this clause. The baseline implementation does not make use of the `a` parameter but it is useful for the later optimizations.

The `decide()` procedure searches for the first unassigned literal in the first unsatisfied clause, and sets the variable such that that clause is satisfied. This remains consistent across optimizations.

B. Adjacency Lists

The first optimization we implemented was adjacency lists. For this variation, we precompute a mapping from each literal l to the list of clauses which contain the negation of l. Then, we override `findRelevantClauses(a)` to return this list without needing to search all of the clauses each time the same assignment is made.

C. Head/Tail Lists

The Head/Tail lists optimization provides a faster implementation of `getImpliedLiteral(c,a)`. For each clause, we maintain a list of H/T pairs. The list represents the positions of the head and tail pointers into the clause at each decision point. The list is necessary so that we can revert the state of the pointers upon backtracking to a previous decision point. When `getImpliedLiteral(c,a)` is called we can check if `a` is pointed to by the current head or tail pointer associated with `c`. If it is, then we push that pointer inwards until we find the next unassigned variable. If the pointers now coincide on the same literal, then that literal is implied by the clause. If not, we know there are at least two unassigned variables.

For this optimization, we also had to add some logic to the `backtrack()` method. Namely, we have to move the pointers back to their position at the time the previous decision point was made. This guarantees that the pointers will still be on the outside of the unassigned variables in a clause, even after the variable assignments are reset by `backtrack()`.

Because not every clause will be affected over the lifetime of a decision level, it is inefficient to update the H/T pointer lists for every clause whenever a decision is made or reverted. Thus we actually stored the H/T pointer histories for each clause as a list of (decision point, head pointer, tail pointer) triples. A new triple is only added to a clause's H/T history when the pointers were moved during a new decision level. And in backtracking, we must only remove the new triples that were added during the decision level being reverted.

D. Watched Literals

Like H/T lists, the watched literals optimization overrides `getImpliedLiteral(c,a)`. We also maintain a pair of pointers for each clause, but there is no need to maintain a history for these pointers. When `getImpliedLiteral(c,a)` is called, we check if `a` was associated with either pointer in the pair. If it was, we perform a linear search over all literals in the clause to find a different unassigned variable that is not already associated with the other pointer. If we find one, then the clause has at least two unassigned variables and therefore implies no assignments. If we cannot find one, then the clause implies the literal pointed to by the second pointer.

Because `backtrack()` only removes assignments, it is clear that it will not invalidate any of the watched literals pointers (by making them point to assigned variables). Thus this optimization does not need to override `backtrack()`.

E. Composition of Optimizations

The adjacency list optimization only overrides `findRelevantClauses(a)`, which means it can be composed with the other optimizations. On the other hand, both H/T Lists and Watched Literals override `getImpliedLiteral(c,a)` which means they are not composable. We chose to include the Adjacency lists optimization when evaluating both H/T Lists and Watched Literals.

IV. Evaluation Methodology

To ensure that our implementations were correct, we checked the outputs of our four solvers against each other and against the results of CaDiCaL [3], a state-of-the-art solver.

In our initial evaluation, we used toy benchmarks such as SAT formulas representing the pigeonhole problem [2] or the N-queens problem [1]. These smaller benchmarks ensured that we saw the optimizations we were expecting while also running relatively quickly. Then we turned to more practical benchmarks from the 2002 SAT competition to see how our solver fared on harder problems. These benchmarks were in our desired CNF format. Certain benchmarks ran close to instantaneously even using our baseline implementation, so we ignored those for evaluation purposes. We were restricted to older benchmarks as even our fastest implementation, with watched literals, took many minutes to run on newer benchmarks. This emphasizes how many optimizations go into SAT solving beyond the ones we implemented, such as enhancing DPLL with the CDCL procedure.

V. Results

We report the solving time for five benchmarks with the baseline implementation, just the adjacency list optimization (AL), the H/T lists composed with the adjacency lists (HT + AL) optimizations, and the watched literals composed with the adjacency lists (WL + AL) optimizations.

Our first four benchmarks each took a substantial amount of time to solve, and were each 1625 clauses. Each of these formulas were UNSAT. We see that each optimization provides

Benchmark	Baseline	AL	HT + AL	WL + AL
Benchmark 1	>300s	49.98s	21.21s	11.49s
Benchmark 2	>400s	173.23s	74.68s	39.65s
Benchmark 3	>400s	123.70s	52.49s	28.33s
Benchmark 4	>600s	307.25s	131.67s	71.37s
Benchmark 5	39.05s	1.34s	1.74s	1.05s

TABLE I
OUR OPTIMIZATIONS PROVIDE THE EXPECTED SPEEDUP.

the expected speedup over the previous implementation. This shows the benefit of using extra data structures to store the states of the clauses to reduce the number of times we have to traverse the formula. These results are evidence of the classic trade-off between space and time. The wide range in solving times between benchmarks show that the effort required to solve a formula is dependent on much more than its number of clauses. Our naive baseline implementation was so slow that we terminated it after enough time had passed to demonstrate that even our simplest optimization, the adjacency lists, was necessary for a significant speedup. Adjacency lists are so powerful since we observe that a given literal is only part of relatively few clauses throughout the formula.

The last formula, 684 clauses, solved significantly faster than the first four. This may partially be due to the fact that this benchmark was SAT and about one-third the size of the earlier benchmarks. While the baseline implementation still took much longer than each optimization, interestingly, the using H/T lists solver took slightly longer than the implementation that just employed adjacency lists. We believe that this is because if a file is small enough, the overhead of storing the H/T configurations and determining where to backtrack to is not advantageous.

It makes sense that the watched literals implementation is the most optimal. Its biggest downfall is that the solver must traverse each literal in the clause to determine if it is implied. However, in practice, each clause is quite short (often, just two or three literals), so traversing the clause is not a computationally expensive operation. The benefits of a watched literals implementation, including lack of ordering of pointers and quick backtracking, far outweigh any cost.

VI. FUTURE WORK

In the future, we will explore further optimizations. We implemented a simple adjacency list structure to provide a foundation for the H/T lists and Watched Literals structures that we were most interested in exploring, but the adjacency lists can be further optimized. As a further speedup when implementing adjacency lists, when we assign a literal, we do not need to search every literal in a clause to determine the status of a clause (satisfied, unsatisfied, unresolved). For each clause, we can maintain a counter for the number of satisfied and unsatisfied literals. Any remaining literals are unassigned. By comparing the number of satisfied literals to the number of literals, we can conclude the status of the clause without checking each literal individually. For example, if the number of unsatisfied literals is equal to the total number of literals in the clause, the clause is unsatisfied. For our solver, it may also be interesting to see if we can pinpoint exactly when the H/T list implementation consistently provides a speedup over just using adjacency lists. Perhaps this can be characterized with number of clauses in the file or the number of decision points required in the procedure. We will also investigate whether similar data structures provide the same speedup for CDCL procedures given that DPLL and CDCL are very similar.

VII. CONCLUSION

We successfully implemented a naive DPLL implementation, and layered optimizations on top of it. Each optimization made use of a new data structure. We were specifically interested in optimizing the boolean constraint propagation step, so our implementations target clever ways to identify an implied clause. When using adjacency lists, we reduce the search space of clauses that may now be implied after the latest assignment. With both H/T lists and watched literals, we keep track of two pointers inside the clause for more efficient identification of unassigned literals and backtracking. The watched literals implementation is ultimately best as it provides the cleverest approach to backtracking and reduces the amount of record-keeping that needs to be "undone" at this step.

VIII. ACKNOWLEDGEMENTS

We would like to thank Caroline Trippel, Andrew Wu, and Hanna Lachnitt for teaching a wonderful class!

IX. CODE

Code for our SAT solver, optimizations, and the benchmarks used for evaluation can be found at https://github.com/colin-r-schultz/CS257-Final-Project.

REFERENCES

[1] N queens puzzle encoder in cnf-sat dimacs. https://user.it.uu.se/~tjawe125/software/pigeonhole/.
[2] Pigeonhole. https://user.it.uu.se/~tjawe125/software/pigeonhole/.
[3] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
[4] Matt Fredrikson. Lecture notes on optimized sat-solving techniques. https://www.cs.cmu.edu/~15414/f18/lectures/19-optimized-sat.pdf.
[5] Ines Lynce Joao Marques-Silva and Sharad Malik. Conflict-driven clause learning sat solvers. https://cse.usf.edu/~haozheng/teach/cda5416/misc/CDCL-sat.pdf.
[6] J.P. Marques Silva and K.A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.
[7] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001.
[8] Hantao Zhang. Sato: An efficient propositional prover. In William McCune, editor, *Automated Deduction—CADE-14*, pages 272–275, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.