Team Information

- **Screen Name(s)**: \n\n\n\n\n
- **Time Spent**: 20+

Introduction

Our SAT solver implementation is based on the DPLL (Davis-Putnam-Logemann-Loveland) algorithm. We chose C++ for its performance benefits, particularly the lack of overhead, which we theorized may be crucial to efficiently solving large and complex SAT instances in relatively short amounts of time.

As for the simple heuristics that we considered first, we implemented unit clauses and pure literals early on, as we figured it would help prune and simplify the problem relatively quickly. Later in the project, we explored the use of watch literals, as we thought that adding them would help speed up performance on SAT problems.

To tackle choosing good heuristics for branching, we tried several different methods, ranging from frequency heuristics, maximum occurrences in clauses of minimum size, the dynamic largest individual sum (see below), as well as assigning literals that appeared in smaller clauses. After trying different weighting schema, we found that up-weighting the branching on literals that appear in smaller clauses worked the best, and therefore unweighted these values, until the point we found out that these values would work the best.

Another observation that we made was that our heuristics for branching appeared to be complementary to our watch literal implementation, and tailoring these two approaches into two distinct solvers was more effective than trying to distill or unify them into one solver. Therefore, we decided that, in approaching our problems, rather than applying one model that used all the heuristics and methodologies together, it was optimal to decide the nature of the given problem and apply the optimal specialized solver for it. Through this method, although we would later realize that our implementations had bugs, the number of unsolvable problems and bottlenecks that we previously had no longer existed.

Some of the biggest challenges included the debugging phase, particularly for backtracking before the first deadline. At the second deadline, we faced problems with our watch literal, as we realized that our watch literal was incorrectly declaring certain assignments to be SAT despite conflicts. Upon some unit testing, we realized that the bug was caused by a declaration error, that is, our code declared something to be SAT as long as all the literals received assigned, irrespective of whether they caused conflicts. This error was difficult to locate in the given test cases and the corpus that we were tasked to solve, as well as the toy examples, but upon creating our own unsat problem, we located the bug.

In addition, to ensure the validity of our results, we created a script that evaluated the correctness. However, upon inspection of the discrepancies between our verifier and assignments that were deemed to be SAT, we realized that for the input documents that our solver approached with the Jeroslow-Wang clause size heuristic, the assignment results were incorrect and would create conflicts. This caused a lengthy debugging period but validated our decision to create our own verification script.

Solution Strategy

Our solver employs several heuristics to improve the efficiency of the DPLL algorithm:

- **Jeroslow-Wang (JW) clause size Heuristic**: Assigns higher weights to literals in smaller clauses.
- **Frequency Heuristic**: Counts the occurrences of each literal in the formula.
- **Maximum Occurrences in clauses of Minimum Size (MOM) Heuristic**: Focuses on literals in the smallest clauses.

- **Dynamic Largest Individual Sum (DLIS) Heuristic**: Counts the number of clauses that each literal alone can satisfy.

Implementation Techniques

- **Unit Propagation**: Simplifies the formula by assigning values to unit clauses.
- **Pure Literal Elimination**: Removes clauses containing pure literals.
- **Heuristic-Based Literal Selection**: Combines multiple heuristics to choose the best literal for branching.

Experimental Observations

- **Performance**: The combination of heuristics significantly improved the solver's performance on various benchmarks.
- **Scalability**: The solver handled larger instances more efficiently due to the optimized literal selection process.
- **Accuracy**: The solver consistently found correct solutions or identified unsatisfiable instances.

Comparison of Techniques

- **Jeroslow-Wang (counting the size of clauses) vs. Frequency**: JW provided better performance for smaller clauses, while frequency was more effective for larger formulas.
- **MOM Heuristic**: Helped in quickly reducing the formula size by focusing on the smallest clauses.
- **DLIS Heuristic**: Provided a good balance between simplification and branching decisions.

Challenges and Difficulties

- **Heuristic Tuning**: Finding the optimal weights for combining heuristics required extensive experimentation.
- **Edge Cases**: Handling edge cases, such as formulas with many unit clauses or pure literals, was challenging.
- **Performance Optimization**: Ensuring the solver remained efficient for both small and large instances required careful optimization of data structures and algorithms.

Conclusion

Our SAT solver implementation successfully leveraged multiple heuristics to enhance the DPLL algorithm's performance. While tuning the heuristics and handling edge cases were challenging, the final solver demonstrated significant improvements in efficiency and scalability.

Additional Information

We chose C++ for its performance benefits, particularly the lack of overhead, which is crucial for solving large and complex SAT instances efficiently.