Sat-Based Reachability

1 Project Overview

The scope of this project was to implement a sat-solver-based graph reachability checker. In order to do this, I had to tackle two subtasks: to write a sat solver that takes in CNF input, and to write a conversion program that takes as input a file that encodes a graph by providing a number of states, a number of transitions, and a list of all the transitions (with the assumptions that the states would be numbered 1..n), and convert it into a SAT problem. The original input files for the project were provided by Prof. Ivancic and had one line of specification (namely the number of states, and the number of transitions) followed by a line for each transition (in the format "source destination"). My target was to convert this input into a DIMACS format Conjunctive Normal Form (cnf) sat problem. By using DIMACS, I would be able to use an existing SAT Solver (namely Minisat), to get a benchmark of performance and a solution. I also wrote two simple SAT solvers - one which implements a simple version of DPLL (as of this writing, I haven't implemented variable splitting but I might get to it eventually), and one which builds a BDD of the CNF input, and then walks from the terminal true node up the parent references to the root to get a possible solution. I have done some very simple benchmarks of how my implementations of SAT perform against both the input files provided, some simple reachability test files I have created, and also some CNF files I found on the internet.

2 BDD Sat

My first work for this project was to write a BDD based SAT-solver.This solver was very simple to implement from a conceptual perspective. Each OR clause of the CNF is trivial to convert to a BDD - the "true" child of any variable node that is true is connected to the true terminal node, and the false node is connected to the node for the next literal (if the node literal is false, then the children are swapped), with the final node's false child being connected to the false terminal node. After such a bdd is constructed for every clause, an AND is performed on all the BDDs (by ANDing the first two, then the result being ANDed with the third, and then the fourth, and so on). I wrote this component before the SAT lecture so I had yet to learn about the better way.

3 DPLL Sat

For this component, I followed an online slide deck I found here. I have not yet implemented an implication graph (this deck also didn't mention it, although it might be implicit in this explanation, rather than a component that has to be explicitly coded). From my understanding of the slide deck, a DPLL solver works through recursion and has a few simple rules that get applied to a CNF input in order to solve it:

1. If there are clauses with exactly one literal, then the truth value of the literal must be the same as in the clause. Because the inputs are ANDed together, if this clause is not satisfied, then the whole expression cannot be satisfied. This also means that if there is a separate clause somewhere with just this one literal, but the opposite truth value, then the expression is unsat.

2. If we have "pure" literals (ie: literals where only one truth value is present throughout the entire CNF), we can set them to their truth value.

3. If we have a variable present in multiple clauses with two elements, we can set it to the opposite truth variable to create clauses with one element, which forces many assignments. If thi is unsat, then we set it to true, and this causes those clauses to be satisfied.

4. We can perform the splitting rule, where we split the cnf into 3 parts: a subset where p is present, a subset where $\not p$ is present, and a subset where neither is present. If both the first and second subset are unsat with p removed, then the whole thing is unsat. Else, if either is sat with p removed, we can set p such that the other is sat, and we are left with just solving the 3rd subset.

Once variables are set with the above rules, the variable assignments are propagated, and a new smaller clause is checked to be satisfiable recursively. In order to generate the smaller problem, we remove the variable from every clause where the truth value doesn't match what we have set it as, and then we remove the variable from the clause; if it does match, then we can remove the whole clause. If we have reached an empty cnf, this is vacuously satisfiable, so everything up the stack is satisfiable, with each stack frame remembering what variables it set and how.

Although modern sat solvers are more complex, even my simplistic implementation of DPLL (which doesn't do splitting at the time of this writing - I might try to do it on the plane ride home, but this will be after the deadline) has shown its benefits in certain cases.

4 Converting to Sat

A very large portion of my time on this project was spent tryinng to understand how to convert the inputs to the model checker to a SAT instance. I followed a number of dead-end paths, before finally converging on an algorithm that seems correct. There is more information about this and how it works in my slide deck. I will also add it here later.

5 Why DPLL works well with Converted Reachability

In my implementation of converted reachability, any node which has just one parent automatically has a two-variable clause. If there are many such cases, DPLL is able to get rid of those clauses quickly, and assign the variables referenced in those clauses, potentially also getting rid of clauses they are in, and creating a cascade effect.

6 Experiments

I compared my BDD-based solver and my SimpleDPLL solver with each other and Minisat. The input files can be found in the res folder of this project. I found that in general , SimpleDPLL scaled much better than BDD. Simple DPLL solved the input given for the first test file (reach1trans.cnf) in under a second, while BDD encountered a Java Out Of Memory error while trying. For a much simpler input (reach3.cnf), SimpleDPLL solved it in under one millisecond, while BDD took 37ms. However, it is possible to create test files which stump SimpleDPLL (for example par.cnf), but which BDD can solve with ease (1.4) seconds. The same test file, however, took Minisat around 0.2 seconds to solve, so the issue here is more likely with my incomplete and suboptimal implementation of DPLL than with DPLL itself.

7 Conclusion

SAT Solvers are a very powerful tool, and demonstrate that sometimes going up in the worst-case order of complexity can mean vast improvements in terms of average case performance (quicksort is another good example). Also, as these solvers are based on heuristics, it is possible for an adversary to find an input that will takke much longer for one solver with certain optimizations to solve than another solver with other optimizations. Because formal verification generally aims to verify software and hardware created by humans, the solver can exploit the recurrence of certain patterns to obtain solutions for the kinds of problems that it is expected to receive. Especially in the case where we have a sparse transition relation, even a simplified subset of DPLL is able to solve a sat instance that woud take longer than the time of the known universe to solve (at least I think $2^10,000$ is that big) down to under a second. Furthermore, the optimizations required for this are fairly simple (they can probably be explained to a first year undergraduate) which makes their power even more exciting.