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, 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 item for this project was to write a BDD based SAT-solver. Creating the algorithm for this solver was easy, and the implementation was fairly simple.

BDDSat Algorithm:

First, each OR clause is converted to a BDD. Then, the BDDs are ORed together, resulting in one BDD. Finally, parent references are set for each BDD node, and the BDD is traversed upward from the terminal true node to the root node. For this component, variable order is fixed - an ascending order is chosen.

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). I have been able to demonstrate that this solver has some very tangible benefits for a number of input files, including those generated from the reachability inputs.

DPLL Algorithm:

According to the 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 union third subset and second union third subset are unsat with p removed, then the whole thing is unsat. Else, we set p to the appropriate value (ie: if the side with $\neg p$ removed was sat, we set p to true to satisfy the other side), and the whole thing is sat.

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. The final algorithm has two components: an unrolling algorithm, which converts the input state machine into a directed acyclcic graph, and a reachability to sat encoder.

Unrolling Algorithm:

Given N nodes and S steps, we must first create a new node representing a node at a given step. If the node n is reachable from the initial node in exactly s steps, then the new node n^s is reachable from the inital node in the initial state. This unrolling can be considered to encode the time component as part of the graph. Any edge from node n_1 to n_2 specified in the input edge list will now be encoded as an edge from n_1^s to n_2^{s+1} where s represents a given step number. It also creates a S-partite directed acyclic graph (it's S-partite, because every node in step 1 must be reachable from some node in step 0 - nodes with no edges can be removed). This graph also has a natural topological sort, which means conceptually, the sat solver scans it from left to right to find reachability. In order to encode the original graph as such a graph, we generate a new edge list, where, given two nodes nuumbered $x, y \in \mathbb{N}$ for each edge (x, y), for each step s$\in$ range(0,S), we create an edge from node $(x + s * N, y + N * (s + 1))$. Prof. Ivancic provided the basis for this algorithm witht the picture he drew me, which is also referenced in the seminal SAT BMC paper by Biere et al.

Graph SAT Conversion Algorithm:

The basis for this algorithm is a prolog reachability algorithm I found here (and had written similar versions of in undergrad), and its conversion to SAT example. In order to convert our Graph to a SAT reachability expression, we give every node a boolean variable. This variable represents whether the node is reachable. A node is reachable either if it is reachable as part of the problem statement, or if any of its direct parents are reachable (ie: if there is an edge from x to y, x is a parent of y). we can also think of this as

$\forall x, y \in Nodes, edge(x, y) \wedge reachable(x) \rightarrow reachable(y)$ and by extension

$\forall x,_1, x_2...x_n, y \in Nodes \wedge edge(x_i, y), \bigvee\limits_{i=0}^{n} \text{reachable(x}_i) \rightarrow reachable(y)$. Because

$p \rightarrow q \iff p \vee \neg q$, we can rewrite this statement as

$\forall x,_1, x_2...x_n, y \in Nodes \wedge edge(x_i, y), \bigvee\limits_{i=0}^{n} reachable(x_i) \vee \neg reachable(y)$. Therefore, if we have a list of parents for each node, we can obtain a cnf clause for each node that tells us that either one of its parents is reachable (at which point it can be reachable), or it is not reachable if we have edges (1,2), and (3,2) pointing into node 2, this clause looks like $(1 \vee 3 \vee \neg 2)$. Note: the sat solver can assign not reachable to the node, but reachable to the parents, but if it is needs to find a solution, it will be forced to assign at least one path from the start node to the end node. Now that we have converted each node to a clause in the DAG, all we need is to have a clause that says that the initial state is reachable in the initial step(so for example, if the initial state is 0, the clause just looks like (0)) and a similar clause for the bad state being reachable at any step (so for example, if we have bad state 1, and n nodes, and 3 steps, this clause might look like $(n + 1 \vee 2n + 1 \vee 3n + 1)$. Finally, we need a clause for each state in step 0 other than the initial state that says that this state is not reachable in the initial step, so, if 2 is a state that is neither initial, nor the bad state, we have a clause that says $(\neg 2)$.

We can use this algorithm to specify liveness as well, we can specify that a certain state MUST be reached within the first n steps by adding a single literal clause that says the given state must not be reachable in step 1, and another that it must not be reachable in step 2, and so on until step n. Then, we expect the sat solver to tell us this is unsatisfiable, or provide a counterexample that provides a path that avoids this state for n steps.

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. Furthermore, the splitting mechanism can work well for nodes that are reachable from many other nodes.

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 (around 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 isuboptimal 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 take 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. For references other than the ones inlined here (which are the truly relevant ones), please look at the last slide in the powerpoint.