README

P3 Theorem Prover Heuristic Strategy

Wesley Gruenberg & Logan Barclay

Executive Summary:

The theorem prover program is a Java program that takes in a knowledge base in conjunctive normal form (CNF) in the form of a text file using the command line. The program attempts to solve the theorem and displays the steps taken to reach a successful resolution or returns failure if the given proof is not solvable. We implemented two heuristic strategies to improve the number of steps needed to resolve a theorem when compared to a random resolution strategy.

Unit Preference Heuristic Strategy

Overview

This heuristic strategy prefers to perform resolutions where one of the sentences is a single literal, or unit clause. The idea behind the strategy is that we are trying to produce an empty clause, so we want to use inferences that produce shorter clauses. Resolving a unit sentence with any other sentence always yields a clause that is shorter than the other clause. The unit preference strategy has shown dramatic speedups, making it possible to prove theorems that could not be handled without the strategy. True unit resolution is a restricted form of resolution in which every resolution step must involve a unit clause.

Implementation

Ranking our sentences and predicates was done easily with a Priority Queue (thank you Java) and by implementing Comparable.

The heuristic function we implemented adds a score to each clause, where lighter clauses are preferred. The score assigned to each clause is directly related to the number of predicates, and the number of parameters in each predicate. Unit clauses are treated as the lowest score and placed in a priority queue. This strategy is used as a generalization of the unit preference strategy. By adding heuristic scores to the sentences in the knowledge base, we were able to implement a random resolve by randomly assigning scores to the sentences and placing them in the priority queue. This random resolve strategy gave a us a base to test our heuristic strategies against.

Java made implementing comparable objects a breeze. We had our predicates and sentences compare favorably if they were more atomic. The compare to in sentences prioritized number of predicates and then the size of those predicates. To speed the process up more, we added Priority Queues of predicates to the sentences so that predicates were sorted in order of favorability. Predicates were considered

more favorable if they had less parameters and if number of parameters were equal, the one that was comprised of more constants was favored.

Fine tuning to optimize speed and number of resolutions

After broadly implementing some queuing and comparing, we were getting noticeable speed-up and improved resolution number from the random; however, on certain test cases, we were able to drop a couple more resolutions by adding better scoring heuristics and sorting our predicates correctly. One bug we had in predicate ranking that wasn't preventing problems being solved was that we were ranking them in the exact wrong order within a sentence. Multiplying their score by -1 solved this. In the case of test5.1, this saved us two resolutions (dropping us from 15 to 13).

Support Set Resolution Strategy

Overview

Our resolution algorithm directly utilizes the support set strategy to speed up the prover. The unit preference allows for prioritizing, but having a direct set of support to draw from helps quite a bit. The book states that if the support set is small relative to the whole knowledge base, the search space can be reduced dramatically.

Implementation

We had our prover keep track of a support set and store that for all subsequent resolutions. We utilized sets and converted to priority queues in order to maintain our unit preference strategy. We started by adding the refuted clause to our set of support and adding our resolved sentences to that set. If we reached our goal, we returned null and printed off our logic for getting there. Our implementation allows for random to utilize this strategy, so our timing of heuristics is mainly to focus on how our unit preference strategies can influence speed. This also allowed for quicker multiple tests.

Subsumption Heuristic Strategy

Overview

The book states that when adding sentences to the knowledge base, if the knowledge base already contains a more specific sentence, there is no point adding a less specific sentence containing the same predicate. To quote our text:

> **Subsumption**: The subsumption method eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small and thus helps keep the search space small.

Implementation

This makes a lot of sense, but proved more difficult to implement than we had hoped. We took a stab and created a subsumption flag so we could toggle on and off and leave our code for later tinkering. As of Sunday afternoon and the writing of this report, we do not have working subsumption heuristics.

What we tried and failed to do was implement our subsumption methods at the creation of the knowledge base. So when we parsed a predicate, we would check to see if it was subsumed. This may be part of the problem: we may need to actually check to see if an entire sentence is subsumed by a more specific sentence.

Testing

Testing Unit Preference Strategy

We ran our tests on all of the provided test files. Most of the files were fairly simple and we weren't able to see much of an improvement over random resolve. Tests test4.1, test4.2, test5.1 and test5.2 were complex enough that we had a noticeable improvement. While random would sometimes solve the theorem in less resolutions than our unit preference strategy, the majority of the time unit preference is the quicker solution.

File	Random Res.	Random Time	Heuristic Res.	Heuristic Time	Result
kb	0	0.05ms	0	0.23ms	Failure
kb2	100	503.41ms	100	1066.51ms	Failure
kb3	3	1.26ms	3	1.45ms	Success
test1.txt	2	0.45ms	2	1.67ms	Failure
test2.txt	6	4.89ms	5	2.89ms	Success
test3.txt	2	0.65ms	2	0.98ms	Success
test4.1	13	10.27ms	8	4.79ms	Success
test4.2	21	40.44ms	8	8.69ms	Success
test5.1	23	19.56ms	13	13.32ms	Success
test5.2	55	89.98ms	27	37.45ms	Success
test6.txt	2	2.98ms	2	2.83ms	Failure

Figure 1.2 Table of results for Unit Preference