Introduction to Artificial Intelligence Final Practicum

Eoin Doherty
Kyle Rosenberg

Problem

The goal of our project was to create an artificial intelligence system to automate the verification of propositional logic equivalence statements. For example, showing that (~P^~R)v((P^~Q)^~R) and ~R^(Q->~(P^~R)) are logically equivalent. The most simple yet time consuming way of solving the problem is using truth tables and comparing the columns for each expression[4]. However, when the expressions get bigger and have more variables it can be faster and easier to use logical equivalence laws[3].

Methods

To represent logical expressions in python, we made an abstract syntax tree. A parser parsed text input into this AST to make user input easier. Our AST supports not, and, or, conditional, and biconditional statements. Each expression in the AST class generates neighbors, logically equivalent expressions, based on basic logical equivalence laws. Using AST representations of expressions and the list of neighboring expressions, we are able to generate a graph of expressions mapping one expression to another. To prove that one expression is equivalent to another we simply find a path from the starting expression to the ending expression. If no path exists, then the expressions are not logically equivalent.

Although generating the entire graph would be solving the problem since we could just check if the node we want has been generated, doing so can be computationally expensive. A full graph of all equivalent expressions would be very dense and potentially infinite. Generating it generally requires more computing time than solving the problem. Instead, we iterated through the neighbors of a node, checked if any of them were the goal, then traversed the neighbors' neighbors. We would have to traverse the entire graph if no path exists from the start to the goal. Some actions, like double negation, are valid for every expression, causing an infinite number of neighbors and thus an infinitely large graph. To prevent this infinite graph problem, we limited the number of double negations generated in the neighbors to two in a row.

We tried multiple algorithms and heuristics to traverse the graph and find a series of steps to prove that the inputs are logically equivalent.

Breadth First Search (BFS)

As a baseline, we tried simple breadth first search. This search algorithm pops nodes off of a queue, generates the neighbors of the current node, and adds them to the queue in order. We also kept a hash table of visited nodes, so if a node is already on the stack it won't be added twice.

Depth of AST

An important distinction we made was that we don't necessarily care about finding the shortest path, just finding whether a path exists as fast as possible, and thus didn't implement exactly A*. We simply used a min heap instead of a stack and scored nodes by their 'distance' from the target, not caring about the steps taken. The first heuristic we tried was comparing the depth of the current node in the search to the depth of the goal node. The depth of the AST corresponds to the number of operators used in the expression. By taking the absolute value of the difference between the depth of the goal node and the depth of the current node, we could quickly compare how similar the two expressions were. For this search and all of the other heuristic searches, we replaced the queue with a priority queue and added the node's heuristic score as well as the node itself to the queue.

Number of Operations

To find a better similarity metric, we tried counting the number of each operation in the expression. This heuristic iterates through the input node's AST, storing the number of each operation in a dictionary. To find a single numerical value for the difference between the current node and the goal node, the dictionary representation of the current node and the dictionary representation of the goal node are passed to a function that adds up the absolute value of the difference between each operation in the dictionaries. This heuristic is similar to AST depth, but favors expressions that contain a lot of the same operations as the goal. This bias is useful as an expression containing similar operations to the goal is likely only a few logical operations away from the goal.

Weighted Combination

The weighted combination combines the scores of multiple heuristics to hopefully create an even better heuristic. In order to find the optimal weights, we could have just experimented with our own until we got better results, but the faster solution was to use a genetic algorithm. We used a modified version of Daniel Shiffman's genetic algorithm[5]. We created an initial

population of 50 random weight vectors where each weight ranged from 0-10 and ran the search on 5 different inputs to score their fitness. Then for the mutation to a new population, we kept the 5 best scoring weight vectors. One of the 5 was picked randomly, and each weight was changed randomly from [-3, 3], and added to the population (repeat 40 times) for 40 more members. The last 5 were generated randomly to try to avoid reaching local minima. This process was repeated for 20 generations, and the resulting best weight vector ([15.33613225, 22.53582815, -19.79393251]) was saved and then hard coded for the test cases used to gather results.

Depth Limited Search

Another thought we had was that a lot of computation was wasted when the amount of steps from the start to the target was much higher than the actual solution. We picked a couple different cases where we knew the number of steps needed, and ran some tests on different limits for each case.

Results

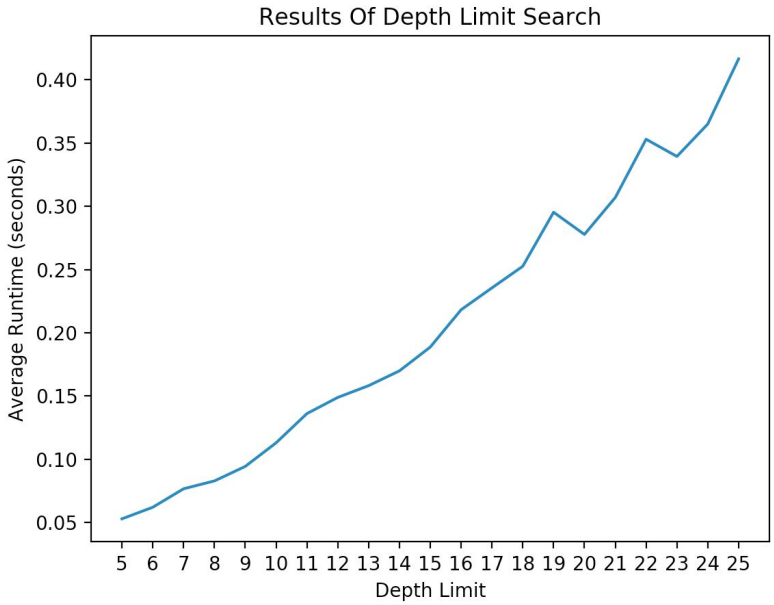
Method	Average Runtime (seconds)
Breadth First	3.28
Depth of AST	0.17
Number of Operations	0.26
Weighted Combination	0.06
Depth Limited Search	0.18

We used average runtime to measure the performance of each algorithm. As expected, breadth first search was the slowest. Other searches performed much better and the weighted combination performed best.

Though heuristic searches yielded better runtimes, there was still a significant difference between the heuristics we tried. The depth of the AST had the best performance with a 0.09 second increase over the number of operations heuristic. Comparing the depth of the ASTs worked well because it ignores complicated, rarely used expressions while still expanding expressions that are different from the start and goal. Some difference in the types of operations is necessary to get from start to goal and this heuristic allows that. The number of operations

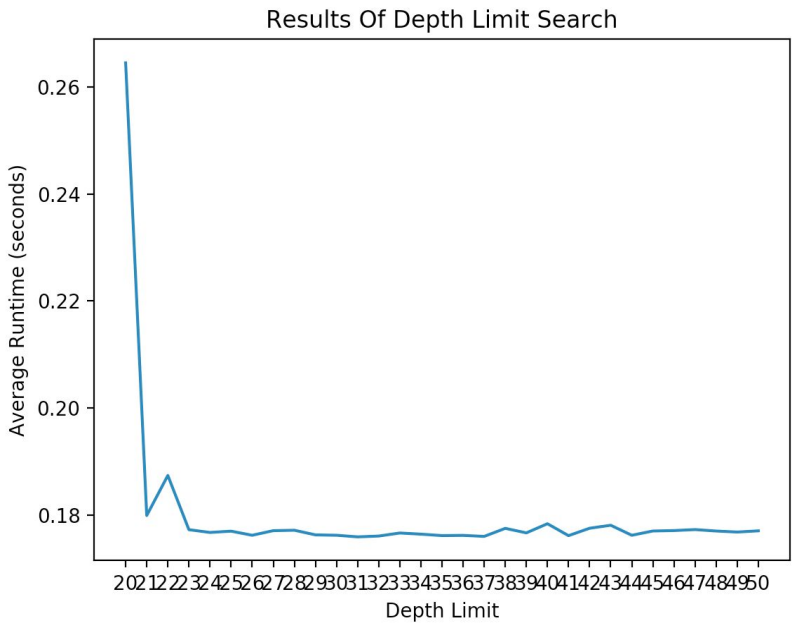
heuristic was likely too biased toward operations present in the goal expression which lead to a lot of dead ends. This heuristic restricts search to nodes with the same operations and even favors expressions that are more complicated but contain similar operations to the goal.

The depth limited search performed well with a runtime slightly slower than the AST depth heuristic, but faster than the number of operations heuristic. The depth limited search algorithm in the table above used the AST depth heuristic to score the nodes that were not pruned.



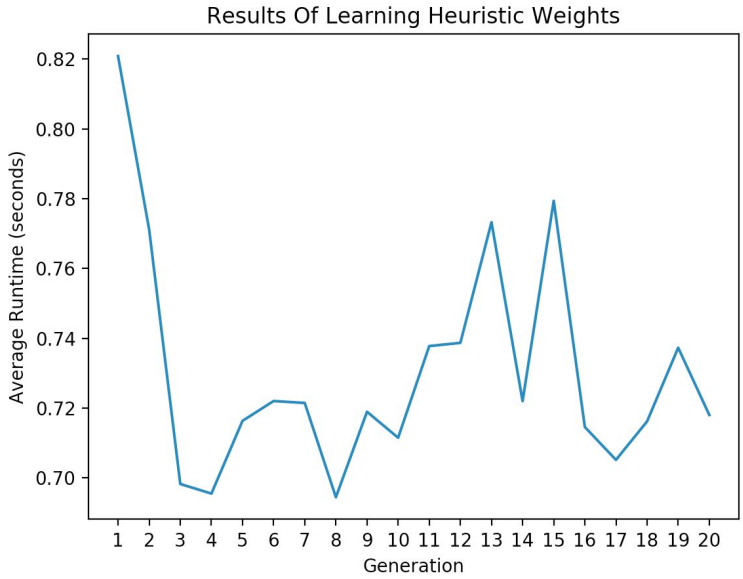
Uninformed depth limited search (BFS) for solving (pv(pv(pv(pvT)))) = T (a 4 step proof)

When depth limiting is applied to an uninformed search, such as the breadth first search in the figure above, the performance increase is more noticeable. Regular breadth first search took 4 seconds on average to prove the equivalence of (pv(pv(pv(pvT)))) and T. Limiting the depth of that search to 24 or below yields an average runtime of under 0.4 seconds, an order of magnitude better. As the figure above illustrates, there is a linear relationship between the depth limit and the average runtime of BFS. This depth limited BFS also yielded solutions with fewer steps. The problem used to generate this graph can be solved in 4 steps. Regular BFS generates a fifteen step solution; depth limited BFS generates a shorter solution, though the exact number of steps depends on the maximum depth.



Depth limited search performance with AST depth heuristic

When a heuristic is applied to depth limited search, the results are not as impressive. Though limiting the depth pruned dead ends, the search rarely encountered those dead ends since the heuristic it was using was fairly reliable. In this case, calculating the distance of each node from the start added unnecessary computational work that increased the average runtime.



Weighted heuristic combination performance per genetic algorithm generation

By far our best result came from combining multiple heuristics into one. When we trained the data as in the graph above, we used 5 test cases that always terminated in under 10

seconds. When we ran tests to create the result table above, we had to omit the last test case because it was taking way too long with BFS. The graph above shows how in just a few generations we made great improvements over the initial random weights, and the table shows how the results were far superior to any single heuristic. The fluctuations in the graph might have something to do with the relatively small population, but given that it ends on a dip the average runtime would probably still converge with more generations. Unfortunately, training takes a long time and further improvements might not be as drastic, so we decided we were happy with these results.

Specifically, the combination of heuristics was probably important because of their nature. There are a lot of ASTs that have the same depth and many with the same operators in different orders, so combining the two allowed the heuristic to be more precise. Another effect is breaking ties; for example if two trees have the same depth, combining it with the operation count would show a clear winner. Given more time, it would definitely be interesting to explore more ways of scoring a node, and more training to improve performance even more.

Conclusions/Discussion

We have tried multiple search algorithms to solve logical equivalence problems. This problem can be difficult to solve since the graph of possibilities is potentially infinite. Our results show that a weighted combination of multiple heuristics is the most efficient search algorithm to find a valid path from the starting logical expression to the goal. AST depth is the most efficient heuristic function when comparing heuristics side by side. If no heuristic can be used, limiting the depth of an uninformed search can be a useful pruning mechanism to ensure the final proof is limited to a certain number of steps. This also causes the algorithm to fail relatively quickly if no proof exists, a property that the other algorithms do not have. Even with the improvements we made to performance for the relatively small expressions, the time complexity of these algorithms is still exponential, and expressions like (~P^~R)v((P^~Q)^~R) = ~R^(Q->~(P^~R)) still don't terminate in a reasonable amount of time (we don't have the answer to this one specifically). The system we have implemented so far is a good starting point for solving more complicated equivalences, and the techniques we have used here can be adapted to solve other, more practical problems using AI planning.

Code Repository

- https://github.com/SunDove/LogicalEquivalences

References

(1) Coles, Andrew, and Amanda Smith. "Greedy Best-First Search When EHC Fails." *Carnegie Mellon School of Computer Science*, 2007, www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html/node11.html#modifiedbestfs.

(2) Newborn, Monty. *Automated Theorem Proving*. Springer Science+Business Media New York, 2001.

(3) Rosen, Kenneth H. "Propositional Equivalences." *Discrete Mathematics and Its Applications*, by Kenneth H. Rosen, McGraw-Hill, 2013, pp. 25–34.

(4) Russell, Stuart, and Peter Norvig. "Logical Agents." *Artificial Intelligence: A Modern Approach*, by Stuart Russell et al., Prentice Hall, 2010, pp. 234–279.

(5) Shiffman, Daniel. "The Evolution of Code." *The Nature of Code*, 2012, natureofcode.com/book/chapter-9-the-evolution-of-code/.