Charles Bolton
Artificial Intelligence
1/19/2020

Best-First Search:

I’ve implemented Best-First Search in the following way: I begin by adding the “initial puzzle” onto a stack; next, I check the possible “successor” puzzles, with a range of 2-4 possible successors based on the location of the blank space. From the input puzzle, the possible successor puzzles are expanded and their corresponding heuristic values are calculated. The successor with the lowest h-value is returned. Ties are not broken, so if two or more puzzles have the same h-value, the greedy algorithm just takes the first one found. This heuristic performs poorly whether or not the number of inversions is high or low. In one puzzle I found it loops forever with only two inversions while it can solve a puzzle with at least 10 inversions. It seems all it needs is to get stuck between two states with the same best h values to endlessly oscillate.

Best First Search With Memory:

I initially misunderstood the problem and wrote a different version of Best First With memory. This algorithm could solve puzzles more quickly than A* by simply keeping track of already-seen successor states and just popping puzzle configurations off a stack. I have included some data in the “Extra Findings” addendum. It doesn’t include the puzzle path but it’s interesting because it could reach the goal state very quickly compared to A*. However, the path it found may or may not be the optimal path. There is little point in averaging the number of steps taken for each heuristic because it gets caught in infinite loops far too often for any interesting or useful data analysis.

A*:

A star is able to solve even really tough puzzles (I tested up to 28 inversions). It works as described in the text, by expanding the node with the best g(n) (depth) + heuristic value. It is worth noting that it can take a very long time to find the solution, especially when there are a lot of inversions. Had I used a faster data structure it would have been more efficient for all three heuristics, but since I used a list of tuples, it does take a very long time on really difficult puzzles.

Manhattan:

For A* The manhattan distance performed much better than the misplaced tiles. In one puzzle I tested for 16 inversions and the difference between the two was stark (see the summary findings. For Best First Search, if it could find a solution at all, it performed about the same as the other heuristics.

Misplaced Tiles:

This heuristic is noticeably worse when it comes to A*, at times performing quite terribly. I waited for one solution for 10 minutes, another I had to terminate because it was taking far too long.

Out of Row, Out of Column:

This heuristic is admissible since if a tile is out of its own row or column (or both) it will need to be moved at least once for each case it is out of order. It seems to perform rarely worse, sometimes the same, but mostly better than the misplaced tiles. It is for most puzzles not as good as Manhattan distance.

15 Puzzle:

I wrote a program for the 15-puzzle and it works (fifteen_puzzle.py), and I’ve been able to run it and solve puzzles with all three heuristics. I did not have time before the deadline to run it on 5 different puzzles but there is one solved puzzle in the findings_fifteen_puzzle.txt.