COMP 4106 Assignment 1

Marc teBoekhorst

100925246

Commodity Transportation

For this problem, the state space of the system was represented as an array of binary digits. The first index of this array would represent the torch, and the remaining indices would represent the people. If a value at a specific index was a “1”, this meant that the entity (whether it was a person or torch) had crossed the bridge. A “0” represented any entity that had not crossed the bridge. The first step in completing the problem was to generate the possible moves. To do this, I used a method that generated every possible permutation, and would compare it to the current state to verify that it was a valid move. Depth First Search (DFS) was quick, however it failed to find the fastest method of travel on nearly every case. Breadth First Search (BFS) was slower, and found the exact same result as the DFS because it stopped once it found a solution, and did not traverse throughout the entire tree. The first heuristic I used was to always send the fastest person with the torch. If the fastest person always travelled with the torch, this would ensure the fastest return rate of the torch after two people have crossed. The second algorithm uses a heuristic function that calculated the time spent crossing for each valid move, then expanded the fastest branch to look at its children. This is very similar to a best-first search, as it allows for back-tracking to previous branches if need be. This algorithm always found the optimum path, but was very slow in the process. One key strategy I found in this method was that it sent the fastest two people on the first iteration, then the slowest two people on the second iteration. The torch would then travel back with the person from the first iteration, which reduced the total time for everyone to cross. The goal in mind was to now create a third algorithm that would take the computation time of the first algorithm with the accuracy of the second algorithm. To do this, I first found the fastest trip for the first iteration and expanded that branch. However, unlike the second algorithm, I discarded each of the other branches with no potential of back-tracking. By eliminating nodes located higher in the tree, this tripled the speed of the second algorithm, and still found the fastest route. The two heuristics worked well together, however the computation time still took a few minutes when using upwards of seven people. When using many people, the first algorithm was the fastest in discovering a solution.

Space Management

This problem was much slower in computation time compared to the first problem due to the number of possible horse moves that could be made on each iteration. To represent the state of the system, a two-dimensional array of integers would be used. The blank square in the puzzle would be represented by a “0”. The BFS algorithm was the most accurate of the five algorithms. It consistently found the best solution, but at the cost of computation time. On very complex puzzles (where every number is out of position), the program would run out of memory when trying to execute the BFS, which made it less reliable then some of the others. DFS was very quick, however made far too many moves to complete the puzzle (sometimes over 9000!). This was because it would exhaust every possible move until it reached the solution, which is not optimal. The first algorithm uses a heuristic function that calculates the number of squares out of position, and makes moves to minimize that number each iteration. This algorithm was fast and accurate, but sometimes did not find the optimal solution that BFS would find. The second algorithm used a heuristic function that calculated the total Manhattan distance (sum of the distances of all incorrect tiles), and expand the branch that would minimize this distance. This algorithm was similar to the fist algorithm, and was sometimes better and sometimes worse in terms of finding the optimal solution. This algorithm was slower on average, as it required a lot of back tracking when finding an optimal solution. For the third algorithm, I added a tie-breaking heuristic to the first algorithm. If two potential moves would lower the number of tiles out of position, it would compare the Manhattan distance of the two moves, and would select the alternative that further lowered the total distance. This algorithm was very inconsistent, and would sometimes perform worse than the first algorithm alone. In terms of reliability, the first algorithm would always find a solution to the problem without having to worry about memory issues, and could complete it in a timely manner. Due to the vast amount of back-tracking used by the other three algorithms (excluding DFS), they were prone to failing due to memory issues, and would take very long to solve the puzzle. DFS is the worst solution, as it does not solve the puzzle using any logic, but by trial and error until a solution is reached. The searches work in some cases for 2x4 and 2x5, but the program is not designed to solve anything outside of the requirements of the assignment.