Landon Crowther
U0926601
3/30/17

Assignment 9 Analysis Document:

Straight Line Distance

The "straight line distance" from the start point to the goal has little to no effect on the run time of our algorithm. Because our algorithm uses a breadth-first search, the "straight line distance" between the start point and end-point is never calculated. Larger maps will generally have a larger runtime, because more work must be done to calculate the shortest path.

Straight Line vs. Actual Path

Straight line distance can be defined as the shortest path between the start point and the end-point. Actual solution path length is the distance that was "traveled" to reach the goal. Because there were walls/boundaries separating the start and goal points, the straight-line path could not be taken. As a practical example, Salt Lake City lies on a mountain range. To get on the other side of the mountain range, one must either travel around the mountains, or through a special canyon where roads have built. One could not simply drive in a straight line over the mountains. The mountain analogy is like this assignment, where if a wall lied between the start and end points, the path to the end-point had to go around the wall.

As far as runtime is concerned in our program, runtime is dependent only on the length of the actual path. For example, a map may exist where the start and end points are nearly adjacent, but separated by a wall. To get to the end-point, the shortest path may have to travel a very far distance, even though the end-point is very close initially. The straight-line path did not affect the runtime, as runtime was determined by the shortest path.

Square Maze:

Assuming each length of the square is N, then the area of the square would be N^2. The worst possible scenario would be a situation where the start and end points are at different corners of the square, and there are no walls in between them. In theory, to get to the end from the start, four nodes will have to be checked at each level, as the only walls are those that create a boundary. If N represents the length of the side, then the Big-O behavior would be order O(4*N^2) = O(N^2). Note, the reason for N^2 is because there are at most (excluding boundaries) ~N^2 nodes.

To confirm this, we implemented a field in the node class called "timesLookedAt." For example, every time the alreadySeen method was called on "Node a," this counter was updated. We made a 28x28 maze, where there was the start point in one corner and the end-point in the other. This maze was constructed as a graph, and the findShortestPath method was called. After the method was called, we had a loop check ever node in the graph, and compute the average number for the timesLookedAt field. This average came out to be 3.84. This makes sense, because for all internal nodes (nodes that are not adjacent to a boundary), there are four neighboring nodes. We would hypothesize that if the dimensions were to increase (i.e. 100x100), and this method was run, the average would come out to be much closer to 4.0.

On the other end of the spectrum, if there was a maze where there was ONLY one path from the start to end, the Big-O notation would be much closer to order O(2*N*2), because the path is at most 2N (make one turn) in length, and each time the path is expanded, only two nodes would have to be checked: the one in front and the one in back.

Landon Crowther
U0926601
3/30/17

Depth First Search vs. Breadth First Search

Depth first search is very like tree traversal, where the "search" goes as far as it can before it gets stuck, and then back tracks. In breadth first search, the "search" uses the first "layer" of options, and then explores the second "layer" of options, and so on until a solution is found, making sure skip over any possibility that has already been explored.

In the context of the maze, if the solution is close to the start point, a breadth first search would make sense. To illustrate, take an example where the end-point is directly east of the start point. In this situation, it would be impractical to use depth first search because there is the possibility that the algorithm searches North first, which could lead the search completely astray and travel in unnecessary directions, whereas if the search went East first, the solution would have been found immediately.

Alternatively, depth first search could be more efficient because it has a much lower memory requirement. Take a very large maze for example. Let's say this maze has multiple solutions, each equal in length and complexity. In this case, a depth first search algorithm might be faster. This is because rather than having to keep track of multiple possible solutions, the depth first search would theoretically be able to find a solution on the first run.

In general, a breadth first search may be able to find the *shortest* path the quickest, because it checks each level until a solution is found; however, a depth first search may be able to find *a* solution faster than the breadth first, but it may not be the *shortest* solution.

Thought Problem:

Assuming the word "edge" in the question is referring to a connection between two nodes:
Because there is no reference to other nodes in the maze, *some* guesses will have to be made to see if adjacent cells are walls or nodes. One algorithm, which is highly efficient and will be very like a depth first search, would be to pick an order of directions (N, S, E, W). Then, check for a node in the given direction. If a node exists, *store some sort of reference to this path*, then continue in the initial direction until a wall is reached. Once this wall is reached, go the second direction until another wall is reached. At this point, try the original direction, then the second, then third and finally fourth. This would continue until the path leads to a dead end. At this point, the path would backtrack until it found the most recent time it changed direction, and it would try the *next* direction. Eventually, this would come to a solution, but because references to previous nodes are being stored as the algorithm progresses, it would eventually become more efficient and be able to learn from previous mistakes.