Tony Diep
CS 2420
Analysis Document
3-30-17

Graphs and Pac-Man Analysis

Throughout this analysis, we explore the graph data structure and its application to the Pac-Man maze program. In our BFS algorithm, we are given the option to either choose LinkedList or the ArrayDeque as the queue. We choose to implement the ArrayDeque over the LinkedList. Firstly, the ArrayDeque is fast and prohibits any null elements on the queue. The ArrayDeque would then only add nodes that are in the path so when debugging and looking at the Nodes in the ArrayDeque, we do not have to worry about extra null space. The LinkedList implementation, however, is most familiar but takes more memory compared to ArrayDeque since each Node has references to another Node.

The straight line distance has an effect on the running time of our breadth first search (BFS) algorithm. If the straight line distance were to be small (meaning that the start and goal are adjacent to each other, with the best case being the start and goal right next to each other), then the running time to do BFS would be shorter. In terms of the best case, the time to run the algorithm is constant since the start is right next to the goal. However, if the straight line distance were to be big, then the running time would certainly be higher since BFS would have to explore all empty space in the maze text file, especially when ignoring walls. When ignoring walls for the running time, there are various paths for the algorithm to explore, meaning more nodes to explore (including the nodes' neighbors).

The **straight line distance** is the absolute distance from the start point to the goal point, ignoring any walls. The **actual solution path length** is the actual path length that the BFS algorithm searches with the walls. Consider the following maze derived from the text file "turn.txt":

```
15 9
XXXXXXXXXXXXXXX
X     ...     X
X     .X.     X
X     .X.     X
X     .X.     X
X     .X.     X
X     .X.     X
X     .X.     X
X     SXG     X
XXXXXXXXXXXXXXX
```

This maze shows a case where the straight line distance and actual solution path distance differs greatly. The straight line distance would be constant since the start and goal indicators are right next to each other, but in contrast, the actual solution path length would not be constant since there is a column wall inserted between the start and goal indicators. The BFS algorithm would have to traverse the workaround to reach the goal. The actual solution path length is more of an accurate representation of the running time of BFS because on average, most mazes do not have a start and goal indicator right adjacent to each other. Given the other mazes to test on our program, BFS had an actual solution path distance on almost all of them (except the maze with just a straight path).

The worse case type of maze that exists for Pac-Man to travel would be the following maze:

```
9 9

XXXXXXXXXX

XS         X

XXXXXXX    X

X   XXXXXXX

 XXXXXXX   X

X    XXXXXXX

X          GX

XXXXXXXXXX
```

This type of maze involves a zig zag traversal pattern, which means for N rows, there must be N columns that need to be traversed. Since the maze above has 7 rows and 7 columns (since we exclude the first row and last row because they are wall segments), every row would need roughly 9 traversals. As the density of the maze increases, the amount of traversals for each column per row also increases. This implies that at the worse case performance, we get a big O complexity of O(N^2). Additionally, if the maze were to be highly dense, then the maze will likely have one path to travel in order to reach the goal indicator. Highly dense mazes imply that there are various wall segments, and if there are more wall segments in the maze, then there are fewer paths to traverse to find the shortest path.

Now consider again the same maze mentioned above but instead of a 9 x 9 maze, the maze would be a 15 x 15 maze. Since the maze is the worst case, the big O complexity of the worst case maze is O(N^2), so there will be at least 13^2 individual nodes to traverse (note that it is 13^2 individual nodes to compare because there are two rows that are wall segments, so it is trivial to traverse them). However, when augmenting the code to keep track of how many nodes are compared (that is, each node being “looked at”), we notice there were 191 nodes that were being compared. With 169 nodes total and 191 total nodes being compared, the results support

that BFS performs the algorithm in the way we expected since it looks at every possible node that exists in the maze.

We hypothesize that when using depth first search (DPS), it would not necessarily explore all of the possible paths in the maze. Because of this, DPS is not guaranteed to find the shortest path but rather finds the deepest path of the maze. The reason for DPS to not be guaranteed to find the shortest path is because DPS essentially picks a node and finds the deepest path from that node while checking the neighbors of that same node. In essence, DPS does not traverse and compare neighboring nodes since it traverses similarly to a Binary Search Tree's (BST) traversal. With BFS, however, the algorithm guarantees to find the shortest path since every node existing in the maze is looked at.

Similar to the maze depicted on the previous paragraph, the type of maze will depend whether DFS will solve the maze in a fast or slow manner. However, the "snake" maze will perform at the worst case since the snake maze only has one path to travel. Additionally, the maze does not have any deep paths since each row has a straight path traversal. Thus, the amount of time for DFS and BFS to take on the maze will be similar.

What would be another way algorithm to implement if we are only given the neighboring nodes but not the edges? Since we originally check for north, east, south, and west neighbors, we can also check for diagonal neighbors (i.e. north/south west and north/south east neighbors). This works because we are given the north, east, south, and west neighbors so we can simply determine whether diagonal neighbors exist. Additionally, when calculating north, east, south, and west neighbors, they each have different rows and columns. This means our algorithm would still have to check the four neighbors, but now it would also check the diagonal neighbors.

For this assignment, me and my partner, Makenzie, spent between 6-8 hours completing the implementation. The assignment was not relatively challenging compared to the Poker assignment other than debugging all of the nodes and its neighbors and verifying that each node has the correct neighbors. I had a lot of fun doing this assignment because Pac-Man was one of my childhood favorite games.