Hayden Donofrio

CSE 3353

LAB02 Report

**Introduction:**

Today I will be analyzing my personal implementation of BFS, DFS, Dijkstra, and the A\* algorithm. I will analyze the amount of time it takes to run each of them through an adjacency list and a matrix, and the amount of operations that each of them use. I have used two different data sets for this report: 16 nodes and 2500 nodes.

With the smaller data set it is difficult to discern exactly why A\* is so much more efficient than the other graphs, but once the set expands, the heuristic that A\* uses really gives itself an edge over the other algorithms. The graphs that were provided were in a simple 2-dimensional plane with connections at a length of one. Dijkstra is dependent on these calculations, and therefore with a travel distance of one for every node, I believe that in my testing Dijkstra will take a similar amount of time to BFS, but it will return the shortest path while BFS returns the first path it finds.

With many different little operations added, the performances for these algorithms may not be completely pure to their original Big(O) for time. Also, as we dive into each chart below. I will mention how much recursive calls can slow down the processor because of all of the pushing onto the system stack. Now we will go in depth on each of the algorithms one-by-one and then I will do an overall comparison at the end.

I have represented each graph in a histogram format since I think it is much easier to read than a line graph with each node being a different algorithm. Through the graph, I only calculated the cost on the A\* because the report requirement only called for collecting cost for the A\* graph, however I have distance for all the graphs.

**DATA:**

All start/end points in this program are randomly generated to provide a good clear representation of how the algorithms act between each other. I will provide the graph normalized (average/maximum value for each of their own graphs) between 0 and 1. I will post both a representation of each data itself and then normalized across its own values.

**16 NODE ADJACENCY LIST SOURCE -> DESTINATION**

I believe that the reason the Dijkstra graph visited fewer nodes than A\* is because of the randomly generated set. They were very close, but its impossible to generate one hundred percent similar sets randomly for each graph.

**16 node matrix START -> DESTINATION**

Once again. We see the A\* algorithm and Dijkstra explored the fewest nodes out of all the algorithms, especially DFS.

**16 NODE LIST DESTINATION -> SOURCE**

**16 NODE MATRIX DESTINATION -> SOURCE**

**16 NODE CONCLUSION:**

With a very small sample size of 16 we can already begin to draw conclusions of how these algorithms are going to perform at the larger data set based on their raw values for time. Both Dijkstra and A\* may take slightly longer to calculation all distances and the use of a priority queue is slightly slow. However, they beat out the competition when it comes to the path that they return and the number of nodes that they search.

DFS is nearly the worst in every category except sometimes it runs fast I believe this is because that the amount of operations that my version of DFS allows it to run very fast. However the amount of nodes it searches for and returns is less than optimal. I believe at the larger graphs that DFS will exponentially perform worse as data sets increase, while we will see an exponential increase in Dijkstra and especially A\* as the data sets increase. Although the time for A\* and Dijkstra are slower now than DFS and BFS, I believe when the sets increase they will perform faster.

**2500 NODE LIST START -> DESTINATION**

**2500 NODE MATRIX START -> DESTINATION**

It should be obvious by now that A\* is superior in every single way except for the amount of time it takes to execute. It may take slightly longer than most of the other algorithms, but the number of nodes it’s searching/storing is significantly fewer.

**2500 NODE LIST DESTINATION -> SOURCE**

**2500 NODE MATRIX DESTINATION -> SOURCE**

**CONCLUSION:**

Throughout the tests today I have come to the easy conclusion that when wanting the most efficient route to take in a decent amount of time with a larger graph A\* is a must. Otherwise BFS will work very well. In our tests Dijkstra acted very much like BFS, I think that is because that all of the distances were rated at 1 instead of really varying much. If the distances were varied, we would see a much larger difference between distance travelled between BFS and Dijkstra. If we were simply worried about execution time for a random sized graph. DFS would be fine since it’s a very simple algorithm that runs fast at O(E+V) since we are only visited each edge maximum of once.

I firmly believe that the adjacency list worked much better than the array since our graph was not very dense. Had our graph been denser we would have seen a reasonable comparison between the two. However, the adjacency list completely outperformed the matrix due to our low density graph.

BFS would work well as well for returning somewhat of a fast path, it nearly returned a path as well as Dijkstra’s algorithm with complexity O(E+V) as well.

Dijkstra runs slower since we need to check many different vertices and update their values if they have not been visited, so we will not be checking all of them. This leads Dijkstra to be a complexity of around O(E + V\*log(V))

A\* has many different calculations to check. However, it is much faster at larger data sets than its partners because it only needs to check each edge and move on. This leads A\* to have a complexity of around O(E) which will end up being a lot faster in larger graphs than DFS, BFS, or Dijkstra. For smaller graphs however, the user may just want to stick to BFS or DFS.