Jayson Werth

47631094

CS 3353

**Lab 2 – Search Algorithms**

Comparison Tables for Adjacency List vs Adjacency Matrix:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Adjacency List Source -> Destination** | | **Search Algorithm** | | | | | |
| DFS Iterative | DFS Recursive | BFS Iterative | BFS Recursive | Dijkstra | A\* |
| **Average Normalized Results** | Nodes in Path | 5212.78 | 5212.78 | 65.96 | 65.96 | 197.4823 | 66.14 |
| Nodes Explored | 5212.89 | 5212.89 | 5090.81 | 5090.81 | 6518.77 | 5137.07 |
| Execution Time | 6.718755983 | 6.195284422 | 6.640676939 | 6.653615958 | 11.06003 | 11.22913 |
| Distance | 5211.78 | 5211.78 | 64.96 | 64.96 | 196.4823 | 65.14 |
| Cost | 0 | 0 | 0 | 0 | 96.8339 | 67.73257 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Adjacency Matrix Source -> Destination** | | **Search Algorithm** | | | | | |
| DFS Iterative | DFS Recursive | BFS Iterative | BFS Recursive | Dijkstra | A\* |
| **Average Normalized Results** | Nodes in Path | 5212.78 | 5212.78 | 65.96 | 65.96 | 198.2 | 66.14 |
| Nodes Explored | 5212.89 | 5212.89 | 5090.81 | 5090.81 | 6549.92 | 5137.07 |
| Execution Time | 3.41270274 | 1.608385859 | 3.59274312 | 3.602048773 | 8.136609 | 8.168439 |
| Distance | 5211.78 | 5211.78 | 64.96 | 64.96 | 197.2 | 65.14 |
| Cost | 0 | 0 | 0 | 0 | 97.16218 | 97.49331 |

* The cost row represents the heuristic/raw cost of the returned path for the cost-based graph searches. For Dijkstra, the cost is purely the cumulative weights between nodes in the final path. For A\*, my heuristic was the cumulative weight of the path so far plus the distance between the two current nodes quantity squared to cause the heuristic to affect the total weight more than the raw cost.

Graphs of Collected Statistics:

Data Analysis – Differences Between Algorithms in Measured Stats:

* Number of Nodes and Distance in Returned Path
  + As was to be expected, the DFS searches had the largest average number of nodes in the result path, since its algorithms merely finds a random path and often times the algorithm goes through the majority of the nodes due to the specific path it takes. For example, my implementation always first went with the “last child,” usually causing the last checked node to be 2 or 3. For this reason, the final distance of the return path from the DFS searches were also the largest on average.
    - Both the Iterative and Recursive DFS searches do not differ much at all on these stats due to their similar implementations, one using a stack object and other using the recursive call stack instead of an object. Both return the basically same path so the stats will be consistent.
  + The BFS and A\* search both had the lowest average number of nodes in the final path and the lowest average distance of the final path. These result make sense since we know that the BFS searches return the path with the least number of edges, so they will have the lowest distance and number of nodes. A\* follows similarly suit to BFS searches since it is optimizing for both cost plus the heuristic of distance between each node, so the path that it finds will be cost-effective and contain minimal travel distance.
    - Both Iterative BFS and Recursive BFS searches do not differ much, similar to the DFS. Both use similar implementation with the queue object used to radial search, only one used the queue recursively. Both return basically the same path so the stats will be consistent.
  + Dijkstra’s data on number of nodes and distance of path was a little more confusing at first since I was thinking it would be very similar to the results of A\*. However, Dijkstra is not trying to optimize based on number of edges as BFS was and it doesn’t take into account the extra heuristic as A\* does, so although its average path is longer than both BFS and A\*, it is looking for the path of least cost, which is shown that is not going to be the path of least nodes.
* Number of Nodes Explored
  + On average, all the algorithms except for Dijkstra contained data that averaged out to show that they explored around half of the nodes in the given list.
    - For DFS, if the source and destination nodes were close or on the correct path that it was searching, the path would barely explore any nodes, but if they happened to be “far” in the sense that the path will have to almost loop around to find it (like with 1 -> 2 for me), then DFS will explore almost all the nodes. These two cases average out to half the data set size.
    - For BFS, if the source nodes were close in proximity (i.e. not on the other side of the graph) then it will barely explore any nodes and find a path quickly, but it were far on the opposite side of the graph, due to the radial search pattern, it would have to explore most nodes to arrive. These two cases average out to an average explored set of one-half the data set size.
      * Similarly to the Number of Nodes and Distance of Path, Nodes Explored will not be affected much but iterative vs recursive implementation of these since they will return basically the same path (by exploring the same nodes).
    - A\* is similar to the BFS searches where it will search most of the nearby nodes first before radiating outwards since the added heuristic of distance affects the cost-based search a good amount, causing distance to play a larger part rather than purely just cost.
  + Dijkstra is the outlier in this batch of data and I’m not completely sure as to why this is so. It may be similar to the reasoning behind the difference in the Number of Nodes and Path Distance as it doesn’t care about how far the nodes are in a xyz-coordinate sense, but rather checks for the lowest cost, changing and reverting back to previous spots when a new path of lowest cost appears. However, this could very well be an instance of improper addition to the tree/exploration that appeared as a common problem when tackling this search.
* Execution Time
  + Based on the data, the two DFS and two BFS searches all have similar execution times (with an outlier with DFS Recursive using Adjacency Matrix) and both A\* and Dijkstra both have similar times which are a bit slower than all the DFS and BFS algorithms. The difference in the timing between these two groups may be due in part to the more expensive function calls that must be used in Dijkstra and A\* that being the extra comparison function within the priority queue that sorts itself whenever a new item is added or the cost/heuristic cost calculation/retrieval from the matrix/graph.
    - For the DFS and BFS algorithms, they all shown to have average execution times ranging 0.2-0.4 seconds of each other. This data remains consistent with the ideas that DFS and BFS both can have very efficient runtimes (when the end is in the path/radially close the start) or have very bad runtimes (if the end is towards the end of the “path”/is radially far away) which when averaged out should provide execution times that are similar to each other (similar to the data of Number of Nodes and Path Distance and Explored Nodes).
    - However, there lies a bit of discrepancy when it comes the DFS Iterative vs DFS Recursive, since, unlike BFS where its iterative and recursive definitions have pretty consistent times, these two algorithms differ a large amount compared to that of the BFS. With the Adjacency List, it ranges 0.6 seconds rather than 0.1 seconds and the Adjacency Matrix poses a larger issue of 1.8 second difference. This is very strange because there isn’t a similar gap with the Recursive BFS, and generally recursion is slower than running a block of code compared to using a loop to iterate through it.
      * This could be due issues that arise with the stack management that I used with the iterative implementation that wouldn’t arise with the natural recursive stack or it could be due to my inclusion of a second conditional Boolean vector that checks if a node is on the tree already before inserting. This is only in the recursive definition since it caused the iterative DFS to gain the properties of a BFS algorithm (i.e. it would return a path with the least amount of edges instead of the long, redundant path that it got beforehand) whenever I tried to add it to the iterative definition as well.
    - A\* and Dijkstra both average about 5 seconds faster than all the DFS and BFS searches. This is one of the more confusing stats to me, since I would expect these to reduce the time to execute since it would be able to find the shortest/least costly path through the use of edge relaxation. However, these algorithms also aren’t time-based optimization algorithm and rather they are cost-based optimization algorithms which look for the path with least cost for Dijkstra or least heuristic cost for A\*. In addition, these algorithms use more methods as stated before when calculating the cost of the path by grabbing data from the graphs and/or calculating a heuristic which also might add to the overhead of each. Although, since they both average out to have similar average execution times, it seems to make sense that they both execute longer due to fact they search for the most optimized path rather than purely a path (DFS) or the shortest path (BFS).

Data Analysis – Differences Between the Types of Graphs

* Number of Nodes, Path Distance, Explored Nodes, Heuristic Cost
  + The paths that are found by the algorithms, regardless of graph type, appear to be the same, or at least the same length in regards to number of nodes, distance, and the exploration path. In fact, the data shows that both DFS, both BFS, and A\* have the exact same averaged stats for both types of graphs and Dijkstra shows minimal variation in all the stats, showing a slightly larger average for all three stats when utilizing the Adjacency Matrix.
    - The consistency in these statistics makes sense since I have implemented both graphs to perform the same functionality regardless of their differences as data structures. That is, when retrieving a node from the graph to use its data or when retrieving the children of a node from the graphs, both will either find the proper node and return it or get all the children of the requested node and they will retrieve the same data. In this way, the algorithms will be able to find the same paths with the same distances and explore the same amount of nodes since they do not recognize any change in the data they are given to work with.
    - The fact that Dijkstra is the only algorithm that has data that varies (basically at all) is very strange since when looking back through its code, it uses the exact same amount of function calls using the graph pointer that the A\* algorithm and when looking through the raw data , it seems that only one or two sets of data actually change between the data from the Adjacency List and Adjacency Matrix. Maybe towards the end of running the 100 different source -> destination nodes something messed up, or maybe when copying data from path object to the ofstream to print to the .csv file some of the data values got mixed up or changed accidentally.
  + The Heuristic cost between A\* and Dijkstra is consistent when using the Adjacency Matrix but has a big discrepancy when using the Adjacency List. This is strange since, like with collecting the nodes and their associated data for constructing the paths, this data for the xyz-position and cost to move from node to node should be consistent between the graphs. I think this may be due to incorrect retrieval of the position for each node for the list, which was a little more complicated to do in the list structure since the children don’t get stuffed with all the data in construction unlike that of the matrix where all the defined nodes do get stuffed with all the need info due to the difference in the storage of “children.” This is somewhat supported when looking through the raw data of the Adjacency List where A\* has some tuples where the cost somehow is evaluated to be something really low (<5) where as all the cost data in the Adjacency Matrix tuples have data mostly greater than 20.
* Execution Timing
  + The average Execution Times of the algorithms are shown to be about 3 seconds slower when utilizing the Adjacency List compared to using the Adjacency Matrix. As discussed before, the A\* and Dijkstra algorithms perform on average 5 seconds slower than the DFS and BFS algorithms and there also is an outlier data point with the timing of the DFS Recursive algorithm utilizing the Adjacency Matrix, which comes out to be 4.5 seconds faster than the same algorithm using the list.
    - Although all the methods of the two graphs utilize the same functionality for constructing the paths/calculating the distance/cost of algorithms, the retrieval of data for children and general nodes is a bit different due to their natural differences as data structures. For example, when constructing the Adjacency Matrix, a vector of vectors of Node pointers is used which takes a bit to construct (especially for a large data set), every instance of each node (since each appears twice) contains all needed info. Alternatively, the Adjacency List utilizes a Linked List of Nodes, which means that only the vertex nodes contain all the info from the files where as the child edges will not have everything.
      * One method which I believe is one of the major overheads between the two is the getChildren method. In the Adjacency Matrix, we can go instantly to the source node’s row and then find all its children and return them without much effort. However, with the Adjacency List, we much iterate through the Linked List to the proper vertex and then grab the children vector from the vertex AND then retrieve the positions from other source nodes and add them to the children since it wasn’t stored from the beginning since the edges aren’t a second copy like in the matrix.
    - When it comes to the outlier that appears with the Adjacency Matrix DFS Recursive call, I don’t have much of an explanation, since as a “basic fact” recursion should be slower (in general) compared to a similar iterative implementation but this is much faster than the iterative version. The raw data shows that the average is not brought down by a simple outlier and rather a full shift of data. The difference in iterative vs. recursive implementation may be due to the lack of a traditional stack and retrieval of a tempNode from the graph based on this dependence of the stack object. With the recursive calls, this call to grab a tempNode based on the top of the stack is no longer needed since we just recur back with the node at the “top of stack” already defined for us. This retrieval could account for the difference shown in both graph’s data, but also, as speculated beforehand about how expensive the methods are for the list compared to the matrix, the fact this one graph function call is not in each recursion call (which will run as many times as the while loop with the stack will execute) could explain the big difference in execution time that arises between the Adjacency List vs. Adjacency Matrix runs of this algorithm.