Colin Weil

**GRAPH READING FOR SEARCHING ALGORITHMS:**

DFSI - Depth First Search Iterative

DFSR - Depth First Search Recursive

BFSI - Breadth First Search Iterative

BFSR - Breadth First Search Recursive

D - Djikstra

A – A\*

Adjacency List Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Average Normalized Results | Adjacency List | Search Algorithms | | | | | |
| Data Size = 16 | DFSI | DFSR | BFSI | BFSR | D | A |
| Nodes in Path | 0.952 | 0.952 | 0.712 | 0.712 | 0.716 | 1.000 |
| Nodes Explored | 0.705 | 0.705 | 0.742 | 0.742 | 0.996 | 1.000 |
| Execution Time | 1.000 | 0.994 | 0.509 | 0.391 | 0.650 | 0.675 |
| Cost | 1.000 | 1.000 | 0.617 | 0.617 | 0.234 | 0.384 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Average Normalized Results | Adjacency List | Search Algorithms | | | | | |
| Data Size = 2500 | DFSI | DFSR | BFSI | BFSR | D | A |
| Nodes in Path | 1.000 | 1.000 | 0.079 | 0.079 | 0.163 | 0.241 |
| Nodes Explored | 0.719 | 0.719 | 0.755 | 0.755 | 1.000 | 0.815 |
| Execution Time | 0.314 | 0.308 | 0.483 | 0.465 | 0.762 | 1.000 |
| Cost | 1.000 | 1.000 | 0.077 | 0.077 | 0.137 | 0.147 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Average Normalized Results | Adjacency List | Search Algorithms | | | | | |
| Data Size = 10,000 | DFSI | DFSR | BFSI | BFSR | D | A |
| Nodes in Path | 1.000 | 1.000 | 0.041 | 0.041 | 0.117 | 0.113 |
| Nodes Explored | 0.770 | 0.770 | 0.787 | 0.787 | 1.000 | 0.693 |
| Execution Time | 0.405 | 0.404 | 0.614 | 0.594 | 0.867 | 1.000 |
| Cost | 1.000 | 1.000 | 0.041 | 0.041 | 0.103 | 0.070 |

Adjacency Matrix Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Average Normalized Results | Adjacency Matrix | Search Algorithms | | | | | |
| Data Size = 16 | DFSI | DFSR | BFSI | BFSR | D | A |
| Nodes in Path | 1.000 | 1.000 | 0.709 | 0.709 | 0.725 | 0.996 |
| Nodes Explored | 0.706 | 0.706 | 0.732 | 0.732 | 1.000 | 0.992 |
| Execution Time | 5.258 | 4.977 | 1.000 | 1.344 | 1.811 | 1.863 |
| Cost | 1.000 | 1.000 | 0.616 | 0.616 | 0.240 | 0.383 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Average Normalized Results | Adjacency Matrix | Search Algorithms | | | | | |
| Data Size = 2500 | DFSI | DFSR | BFSI | BFSR | D | A |
| Nodes in Path | 0.331 | 0.298 | 0.568 | 0.530 | 0.529 | 1.000 |
| Nodes Explored | 0.807 | 0.807 | 0.755 | 0.755 | 1.000 | 0.815 |
| Execution Time | 0.618 | 0.536 | 1.000 | 0.959 | 0.995 | 2.086 |
| Cost | 1.000 | 1.000 | 0.049 | 0.049 | 0.088 | 0.095 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Average Normalized Results | Adjacency Matrix | Search Algorithms | | | | | |
| Data Size = 10,000 | DFSI | DFSR | BFSI | BFSR | D | A |
| Nodes in Path | 1.000 | 1.000 | 0.026 | 0.026 | 0.074 | 0.072 |
| Nodes Explored | 0.806 | 0.806 | 0.777 | 0.777 | 1.000 | 0.694 |
| Execution Time | 0.331 | 0.298 | 0.568 | 0.530 | 0.529 | 1.000 |
| Cost | 1.000 | 1.000 | 0.026 | 0.026 | 0.065 | 0.044 |

List Data Graphs:

Matrix Data Graphs:

Code Design:

I put all of my searching algorithms into the Search class because I wanted them to have access to the Matrix and Adjacency List that were created. In the lab handout, you said that the search algorithms should take in two parameters of the start and end id of the node, so there would be no way for the search algorithms to access the data structures by reference, and I didn’t want to have to call a function every time in order to get the information needed to run the algorithms.

In my general structure, I also had two sub-calls of the search functions, one for running the algorithm on the Adjacency List and one to run the algorithm on the Adjacency Vector. I did this in order to make it easier to read and change code, but also especially for the recursive functions. It would be very hard to have one recursive function run certain code based on one data structure, and other code based on another. To simplify and make it easier to write the code, I ended up just making two functions and followed this practice for the other algorithms.

My Node class was how I kept track of all of the data read in the files within my Adjacency List and Adjacency Matrix. Inside of it, I kept track of the id, weight, distance, and ranking of the node. The weight, id, and distance were used throughout all the functions, but the rank variable was only used for the Djikstra and A\* algorithms to keep the ranking of the nodes within the priority queue. I also had a pointer to the parent Node which is how I ended up finding the final path. I basically made the path a linked list of nodes.

DISCLAIMER: After getting all of my data from running each algorithm 100 times, I went to write my report. I double checked the data and realized that both the Djikstra and A\* algorithms were implemented incorrectly. I know exactly why but didn’t have time to fix them because of their run time is long so the data is incorrect. The ranking between the two doesn’t carry over so the data doesn’t in fact find the best route. In order to correct these functions, I would have changed the way that the ranking system worked by setting the child rank as adding the parent rank onto the child rank which would have just been the distance\*weight for Djikstra and distance\*weight + heuristic (position to final node) for A\*. I know that the data is incorrect, but I will pretend like they are while talking about the results.

Results:

Matrix vs. List – The difference between the two data structures didn’t end up affecting the general trends of the lines on the graph. Although they do seem to be the same, the adjacency list does end up being better as the data size grows. This is because while looking for nodes that a current node points to, the adjacency list only contains the known nodes that are pointed to while the adjacency matrix doesn’t know which node is pointed to. This means that the adjacency matrix could potentially search the whole data size of nodes if the node is the last node which makes it less efficient for the way I used in my program. Also, there can be a small difference in the number of nodes explored/ the path that the data structure is explored. This is because the linked list doesn’t take into account what order the ID is in, it only cares the order of when the node is added to the list. On the other hand, the matrix will always search from the first index to the last which can cause a difference in the path taken and some of the data may change accordingly.

Recursive vs Iterative – I was surprised by the result of the comparison of the two because I thought recursive would be slower. Recursive has to call itself over and over so I thought overall that this would end up making the algorithm slower. The interesting part is that while using the adjacency list, the difference in time is basically negligible as it comes down to milliseconds. But for the Adjacency matrix, the difference is obvious. This could be because the recursive part of the function just runs on my computer, or the iterative function gets punished more by the matrix’s nature. For all other data, the recursive and iterative functions return the same results.

A\* vs Djikstra – A\* is just a more developed version of Djikstra so it makes sense that it is more effective with timing (based on the quality of the heuristic). If the heuristic that is utilized is very effective, then the timing of the system will be much lower because the algorithm’s guesses of where to go are very accurate which speeds the whole algorithm up. Since the system also is guessing the correct path more, there will be less nodes to check and will affect the time to go down. Djikstra and A\* will both return the same path and cost because they will both return the best possible path based off the cost.

Overall Algorithm Effectiveness – DFS suffers the most on a consistent basis to find the correct path on every single aspect. This is because it basically guesses the direction, without being completely randomized, and goes. DFS is only be effective if the location is directly in the same location as the first guessed direction. This makes no sense because if the destination is very close in the opposite starting position, then it will take a very long time for a simple solution to be found. This is where BFS will improve on DFS. BFS does a radial search so it won’t go far away before checking its closest Nodes. This also can have a downside because if the destination is very far away in one direction, BFS will take a very long time to operate. Both of these algorithms have their positives and negatives, but both still won’t find the most effective path. Djikstra will always find the best path while using a BFS-type radial search but will check to make sure it is the most efficient path. Djikstra will sacrifice its timing in order to find the best path. A\* is the best of these algorithms because it improves on Djjkstra’s timing, while still finding the same path (the best path). A\* is only as good as how effective it’s heuristic is, so if the heuristic does well, then A\* will do well. The heuristic can also harm the timing in some cases if it’s guess is not actually in the right direction.