Lizzy Hanna
CS 3330
Algorithms
10/17/19

Algorithms Lab 2

I noticed that I would sometimes get slightly different paths from source to destination between Adjacency List and Adjacency Matrix. This is because the children in Adjacency Matrix are ordered but that is not always the case with an Adjacency List.

Dijkstra and A* greatly outperformed BFS and DFS in terms of speeds, nodes explored, distance, cost, and nodes in path on smaller data sets. In fact, my BFS and DFS algorithms were so slow and strenuous that it took me over 20 minutes to run a single search in the 2500-Node graph before my memory manager killed the program. Because of this, my summary table is largely empty because I was unable to get the data I needed before the program would crash, even when running the program Lyle genuse servers.

One way I would revise this is to have a stack/queue that stores Nodes or ints instead of possible paths to the destination. Storing so many vectors (paths) and having to continually call the copy constructor to add onto the path is likely the source of the huge amount of time and memory spent on BFS and DFS. A back-tracking function would likely be much more efficient.

However, even if I were to revise and optimize my BFS and DFS, I can safely say that this is the ranking in terms of speed of these algorithms:

1. A*
2. Dijkstra
3. BFS/DFS Iterative
4. BFS/DFS Recursive

Since A* and Dijkstra are not blind searches, they are not as costly nor as slow as BFS/DFS on average. Dijkstra chooses the next node based on the least amount of cost, whereas A* chooses the next node based on a heuristic value (in this program, the heuristic was F(n) = distance(1+cost)). At first, I implemented a Dijkstra algorithm that chose the next

node based on the shortest distance between the nodes. I believe that this would be a fine way to implement Dijkstra but would cause a decrease in performance because I was having to calculate the distance between the two Nodes each and every time I viewed a child of the Node.

When it comes to BFS and DFS, I found that the iterative searches performed better in terms of time than the recursive searches (whenever I ran random values, the iterative searches tended to finish while the recursive searches timed out). Recursive searches, especially in large data sets, caused a huge strain on the stack because a new activation record was being pushed on the stack every time the search moved to the next node.

For the following tests on BFS and DFS, I used the Sample Graph of 16 nodes and randomly selected nodes as source and destination each iteration. If I were to redo this project, I would try to optimize BFS and DFS so they could run in a reasonable time (i.e. not hours) on larger graph sizes. Here is the data I collected:

Adjacency List:

	Search Algorithm					
Adjacency List Source->Destination	DFS Iterative	DFS Recursive	BFS Iterative	BFS Recursive	Dijkstra	A*
Nodes in Path	8	8	3	3	x	x
Nodes Explored	8	8	9	18	8	8
Execution Time	53	104	98	322	41	44
Distance	7	7	2	2	x	x
Cost	n/a	n/a	n/a	n/a	x	x

Adjacency Matrix:

	Search Algorithm					
Adjacency Matrix Source->Destination	DFS Iterative	DFS Recursive	BFS Iterative	BFS Recursive	Dijkstra	A*
Nodes in Path	8	8	7	3	x	x
Nodes Explored	x	9	x	17	9	8
Execution Time	52	96	159	336	33	32
Distance	x	x	x	x	x	x
Cost	n/a	n/a	n/a	n/a	x	x

(X means data was not collected. Execution time is in microseconds.)

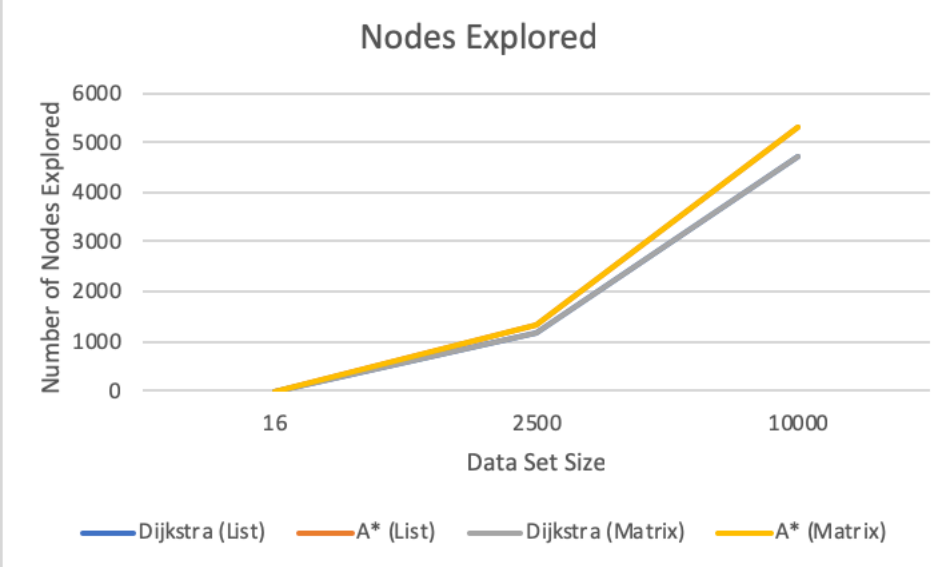
I was able to test the 2500-Node graph size and 10,000-node graph size with Dijkstra and A*. Here are the results I got for the List and Matrix versions of each search algorithm:

Graph Size: 2500	Dijkstra (List)	A* (List)	Dijkstra (Matrix)	A* (Matrix)
Nodes Explored	1177	1333	1184	1336
Execution Time	3238	3334	27639	30586

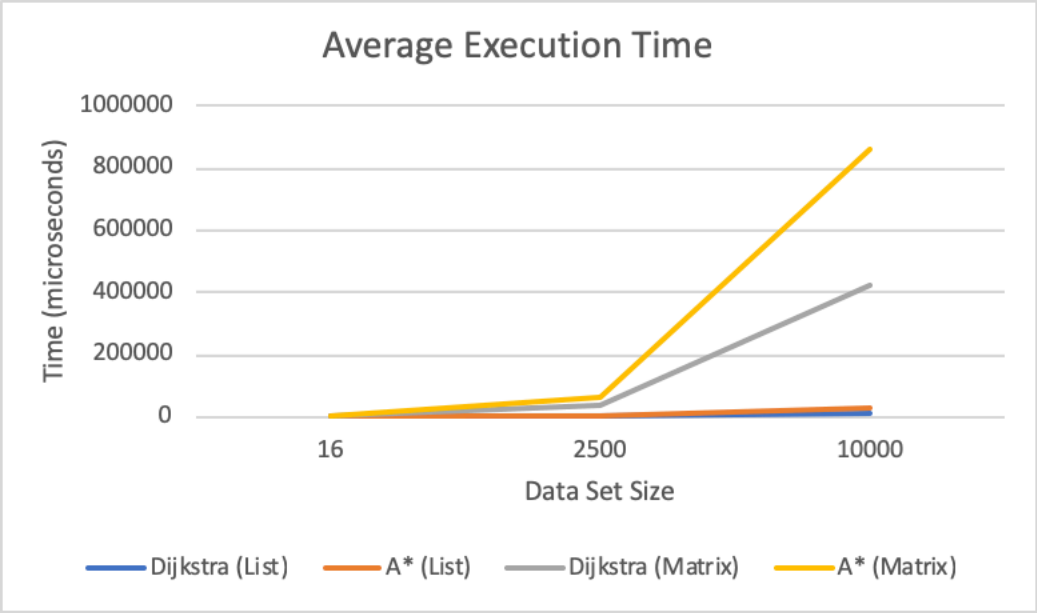
Graph Size: 10000	Dijkstra (List)	A* (List)	Dijkstra (Matrix)	A* (Matrix)
Nodes Explored	4737	5310	4723	5317
Execution Time	14194	14222	396311	434281

(Execution time is in microseconds)

Here are the plots for Nodes Explored and Execution Time as the data set size increases:



It is interesting to notice how A* explored more Nodes on average than Djikstra in both Adjacency Matrix and Adjacency list form



As the data set size increased, the search algorithms ran using an Adjacency Matrix suffered in performance.

Overall, guided searches like Dijkstra and A* outperformed blind searches like BFS and DFS, and iterative functions outperformed recursive calls. Additionally, searches using a Adjacency List performed noticeably better than searches performed using an Adjacency Matrix. I prefer the Adjacency List because there is no need to store duplicate data and it is easier in my opinion to access the children of a Node in an Adjacency List than in an Adjacency Matrix. To find the all the children of a Node in an Adjacency Matrix, one needed to loop through the entire row to find every child, meaning that finding the children of a Node was O(n) time where n is the size of the graph whereas Adjacency List already stores the children in a vector or LinkedList, making accessing a child nearly constant time. This is likely the reason why the timing of the Adjacency List shot up so quickly as the data set size increased.