Software Portfolio

Lachlan Gourlay - 201700039

Contents

Introduction

This report will be looking at the performance of a data structure which represents a transport network as well as a set of algorithms used to navigate said network. The network is made up of a series of nodes and arcs. Each node represents a place on a map such as a town/city, train station, airport, or seaport. Furthermore, each arc is a link between these places, such as a road, railway line, air corridor or sea lane. Additionally, each node had its own set of longitude and latitude coordinates which corresponded to its location on the map.

There were 7 operations which needed to be implemented on the network including an algorithm to find the maximum distance between two nodes. This meant finding the two farthest nodes as the crow flies within the data network, the coordinates stored within the node are then used to calculate the distance. The next operation needed to find the largest arc size, this meant using the coordinates of each node and calculating the distance of the link between every arc. This information is then used to determine which link has the largest size. Similarly, the find distance operation is given two places and using the longitude and latitude of each node gauges the direct distance between the two.

The fourth operation which needed to be implemented on the network data was named FindNeighbour, this evaluation needed to return all of the connecting nodes to a specified place on the network. The final evaluation algorithm is named check, this operation was passed a proposed route made up of nodes and a method of travel. From this it, it needed to check that first, this was a valid route by checking the neighbour arcs of each node in turn. Conjointly it must ensure the method of travel is valid at each stage.

The two final algorithms are similar in the fact they're both pathfinding operations. The first simply needs to find a path between two nodes within the network. It does this using a depth-first search, going from node to node starting on the left-hand side of the graph until it reaches the target node. The final operation is slightly different in that it needs to find the shortest route between two points. This is completed by solving the travelling salesman problem, one solution to this is using Dijkstra's shortest path. At each node, the path is decided depending on which arc is the least distance.

The aim of this report is to look at the performance of the data structure as well as each of these algorithms, this includes the time taken for each to complete. Along with the number of times each operation accesses each node record, as well as each arc record from within the data structure. From this, we can draw conclusions surrounding the number of times a record is visited and the time is taken for the process to complete.

Results

BuildNetwork

Command	Result	Time in microseconds	Number of records visited	
			Node	Arc
BuildNetwork	N/A	2548.0	78697	720

MaxDist

Command	Result (km)	Time in microseconds	Number of records visited	
			Node	Arc
MaxDist	411.279	850.0	47089	0

MaxLink

Command	Result (km)	Time in microseconds	Number of records visited	
			Node	Arc
MaxLink	356.309	24.0	217	720

FindDist

Command	Result (km)	Time in microseconds	Number of records visited	
			Node	Arc
FindDist 9361783 11391765	13.531	13.0	2	0
FindDist 9361783 12321385	48.432	9.0	2	0
FindDist 9121959 14431547	57.694	15.0	2	0

FindNeighbour

Command	Result	Time in microseconds	Number of records visited	
			Node	Arc
FindNeighbour 8611522	8631524 11251704 9361783 12321385 13491586	6.0	1	5
FindNeighbour 51889340	17191741 19151566	20.0	1	2
FindNeighbour 9081958	9361783 9121959 12032132	7.0	1	3

Check

Command	Result	Time in microseconds	Number of records visited	
			Node	Arc
Check Rail 14601225 12321385 8611522 9361783	14601225, 12321385, PASS 12321385, 8611522, PASS 8611522, 9361783, PASS	34.0	3	11
Check Ship 14601225 12321385 8611522 9361783	14601225, 12321385, FAIL	17.0	1	3
Check Car 12032132 12872098 14202070 15602006 15652001	12032132, 12872098, PASS 12872098, 14202070, PASS 14202070, 15602006, PASS	52.0	18	35

15761842 15941834 16141820 16321770 16411764 16431761 16391752 16531745 16561743 16721745 16861747 17151748 17191741 51889340	15602006, 15652001, PASS 15652001, 15761842, FAIL			

FindRoute

Command	Result	Time in microseconds	Number of records visited	
			Node	**Arc**
FindRoute Rail 9081958 15832241	9081958 9361783 8611522 11251704 12681748 14211727 14931717 15581717 16541744 15931781 15761842 15652001 18012084 18202128 16852174 17062210 15832241	38.0	18	67
FindRoute Ship 9081958 15832241	FAIL	7.0	1	3
FindRoute Foot 14601225 9361783	14601225 12321385 8611522 8631524	18.0	15	55

	9791429 12271393 13461591 10351609 10381699 8441694 9061761 9611790 9291784 9361783			

FindShortestRoute

Command	Result	Time in microseconds	Number of records visited	
			Node	Arc
FindShortestRoute Rail 9081958 15832241	9081958 12032132 15832241	619.0	6784	100
FindShortestRoute Ship 9081958 15832241	FAIL	222.0	653	3
FindShortestRoute Foot 16531745 15952280	16531745 16391752 16431761 16411764 16321770 16141820 15941834 15591825 15681844 15491853 15602006 15952280	1961.0	47363	720222.0

Discussion of Results

Review of data structure

To represent and store the transport network each node needs to be stored in an object. A class is created to represent the node, the Nodes are made up of a place name, reference number, longitude and latitude and a list of arcs. Furthermore, a vector is used to store pointers to each of the node objects which when created are added to the heap. An Arc object is also needed to represent each arc on the network. Each arc contains a pointer to the destination node, its mode of transport and the size of the arc which is the distance between the two nodes. The arc is then added to the corresponding nodes vector and the destination node is used to link the two.

A vector is used to store both the nodes and arcs. This is due to the fact it is dynamic, giving it the ability to grow and shrink in size utilizing exactly how much memory is needed depending on the size of the data set. Using a vector of arcs within the node object allows us to keep a single data structure which holds both the nodes and arcs. This increases performance when traversing the network data. It also makes it much easier to traverse when using a pathfinder as it allows you to jump from one node to the next consecutively.

Using pointers to store the Node and Arc objects also helps to improve performance. This is because when an operation uses a record, instead of copying the object in a deep copy where all of its data is copied to another place in memory, it simply creates a pointer to the place in memory and uses the original version in memory.

Analysis of commands

The first part of the code that can be measured for performance is the build network function. This method is in charge of reading in the data from CSV files and constructing the network. This is achieved by first looping through the places.csv and creating and storing Node objects into the large dynamic vector of pointers. Next, we read in the link.csv and create and add the Arc objects into the dynamic vector of pointers within each Node object. Before each Arc is added to the data structure the size of the Arc is calculated and is also added to the object. This process can be quite slow as we were accessing the node objects every time a record is added.

A series of algorithms are used to navigate the network. The first algorithm is named MaxDist, MaxDist is used to find the maximum direct distance between the two nodes which are the furthest apart. It does this by looping through every node that is stored within the data structure. It then goes through each node again each time it loops over the data structure. This allows the Pythagoras calculation to be applied to each combination of every record in the data set. The maximum value is stored and outputted. The reason we access the Node records 47,089 times is due to the fact there are 217 Nodes stored and because we have to compare each node with every other node that gives us 217 x 217 which equals 47,089 records accessed. This is quite inefficient so it takes 850.0 microseconds to perform this command.

The next operation which is performed on the data structure is called MaxLink. This is used to calculate the maximum single distance between all of the Nodes using the Arcs within the data structure. This algorithm like the previous MaxDist uses Pythagoras to calculate the distance between the nodes but this time is used to calculate the size of the Arcs. This time we only need to visit each node once so that we can get to the Arc records. So we visit 217 Node records and we visit each Arc once which is equal to 720 records. Due to the fact we're only accessing each record once, this algorithm is proportionally more efficient as it only takes 24.0 microseconds to complete. The reason this operation is so much more efficient than the last is because all of the calculations for the size of the arcs are done when we build the network. This means were simply doing a linear search on the records to retrieve the largest link.

The third algorithm which is used to search the transport network is named FindDist. This operation is passed two places and outputs the direct distance between the two nodes. It does this by performing a Pythagoras calculation using the longitude and latitude values stored within both Node objects. As this operation is only finding the distance between two Nodes only two Node object records are accessed. This means the time taken is consistent between each test as the number of records does not change. We can see this as the time in microseconds is consistently below the 20 microsecond mark for each time the operation is run.

The next algorithm named FindNeighbour is used to return all of the connecting Nodes to the Node it is given. It does this by looking at the Node that has been passed and following the connections to the Nodes which are joined by the Arcs. As there are no comparisons happening for this function, were simply doing a search and retrieve on the data structure. There will only ever be 1 Node record accessed to retrieve its Arcs. The number of Arc records which are accessed depends on how many links a specific node has. Due to the fact, the number of Arcs does not vary too much the time taken for the algorithm to finish does not fluctuate too heavily.

The final evaluation algorithm is named Check, this operation is used to verify a path through the network is correct. The first task it must complete is to check whether the path of nodes given are in the correct order and have arcs linking them together. This means they must be next to each other in the network graph. The algorithm must then check that those Arcs that link the nodes together can be travelled across with the selected mode of transport given in the command. The number of records visited depends heavily on the length of the route supplied. If the route fails the algorithm stops immediately meaning no more records are visited which is one way the operation has been made more efficient. The time is taken increases the longer the route given to check is. This increase is also not linear with the number of extra nodes, as for each node given we have to check all of its Arc records.

The first pathfinding algorithm was named FindRoute, this operation simply needed to find any route through the transport network as long as the transport method was valid. To ensure efficiency a recursive routine was used to solve this problem. The function is passed the current Node and checks its neighbour Nodes. If it hits a dead end it backtracks and takes the next available branch until it reaches the target. This means we visit the minimum number of Node records as well as the minimum Arc records. Due to the fact, we don't access many records the time taken for each command operation to execute is fairly low. There isn't a very big variance in the time taken as the number of record retrievals is never very high. Saying this, if we were to request a much longer route, for example from the two farthest points we would see an increase in the time taken to complete ass more records is needed. The increase is not linear due to the backtracking and the possibility of travelling the wrong way. One way I have increased efficiency and performance is that as soon as the route becomes invalid the algorithm stops, meaning no more records are accessed.

The final operation similar to the last is a pathfinding algorithm. The difference this time is the final output must be the shortest path from one node to the target. To complete this I have used Dijkstra’s shortest path algorithm and have adapted it to work with the data structure I constructed. To ensure the path is the shortest we must visit each Node and Arc record at least once. This means the efficiency of this algorithm is less than the previous pathfinding operation. Due to the way I store the distance from the starting Node, even if the route is found to be invalid early on, each Node is still visited once. But as soon as the route is invalid the operation stops searching for a path, this is to try and increase efficiency. The further away the target record is from the starting Node the longer the path and thus the more records that are accessed, and in turn the longer the process takes to finish.

Improvements

One way the data structure could be improved is by sorting it so it is no longer unordered. As our data set isn't very large, performing a linear search doesn't affect performance too heavily. But if I was to order the vector of objects, instead of using the find() function built into vectors I could use a binary search. This would be an order of magnitude faster if we had a big data set and a lot of records to search through. Another method I could use to improve the performance of my code is by using the reserve() function. This would be used when I add objects to a vector. The function guarantees vector capacity is at least big enough to fit the n number of elements requested. This ensures no memory is reallocated which can negatively affect performance quite heavily.

Additionally one way I could improve the pathing finding and shortest path algorithms is by using a queue to store the visited Nodes, as well as the distances for each of the Nodes. This would be more efficient as most of the Nodes would not be visited so storing this value within the object is wasted memory. Storing both of these values in a queue would also increase performance as it would stop the need for searching through every Node to find the shortest distance as it would be at the front of the queue.

Another method for increasing the performance for two of the operations, involves calculating the distance for the MaxLink and MaxDist processes in an alternate way. When calculating the maximum link we check the distance of every Arc both ways. In reality, we only need to perform this calculation one way for each link, this would halve the number of records being accessed. Similarly, for the maximum distance process, we only need to perform the calculation once for each pair of values. If we were to make this change we could halve the number of records accessed and in turn speed up the operation.

Currently, the shortest path operation is using Dijkstra to determine which route has the least amount of jumps. The issue with Dijkstra is that it is inefficient when choosing which node to visit next. A* uses heuristics to approximate the cost of reaching the goal node at each jump. This means if you have a good heuristic function the number of Nodes you would explore would be greatly reduced. With fewer records being accessed the performance of the function would improve significantly.

Finally, a performance improvement could be seen throughout each operation if it were to write the output to the file after it had finished each command operation. At the moment the program calculates the output and writes to the file after each command. If it were to instead do this after each operation had calculated its output and write all in one go it could see a performance improvement.

Conclusion

In conclusion, I have found that the more records I access within my data structure, the longer the process takes to complete. There is a positive correlation between accessing my data structure and the algorithm taking a higher number of microseconds to finish. There is evidence of this throughout my whole report. The maximum distance algorithm takes a performance hit as it has to access every Node object multiple times. The find distance algorithm takes consistently a similar time to complete as each time it runs it only access two Node Objects from within the data structure.

When looking at the FindNeighbour operation it can be seen that the time elapsed slightly increased when accessing more of the Arc records. The increase was discreet as the variance between the number of Arc records for each Node was minimal. One algorithm where a large increase in time elapsed can be seen between each time the operation is completed is the Check process. This is due to the fact when a route has a higher number of Nodes each of those Nodes has to be accessed along with its Arc records and such the time increases. The FindRoute routine has little variance between each time the operation is run. This is due to the minimal number of records that are accessed throughout the life of the process. Finally, the shortest route algorithm shows the largest variance in the time taken to complete. This depends on how far away the target record is from the beginning Node. If the path is far then the more records accessed and thus the time elapsed is larger.