



# COMP3712 ASSIGNMENT 2

Code Documentation

Timothy Foong  
Foon0020

## Running the code

The code will take in user input to output the results of Dijkstra's algorithm. The input should be formatted as thus, [path to graph being read] [starting node] [true if you want to see the adjacency matrix, false if not] [true if you want to see the path and distance to each node, false if not] [1 if you only want to see the list path and distance, 2 if you only want to see the priority queue path and distance and 3 if you only want to see the custom priority queue path and distance. Leave blank if you want to see all three.]

A sample input would be: "data/graphs/random\_v10\_e10\_w50.graphml 0 false true 1".

This would read in the graph from data/graphs/random\_v10\_e10\_w50, find the shortest path from 0 to each node, not show the layout of the graph, would show the path and distances of the nodes and only show the list path.

## Part (a) Reading in data from GraphML file

The first thing that I did in this project is copy over all my code from the previous lab. This included the Graph, MyGraph (which comes from AdjacencyListDirectedGraph) and the Vertex data structure. Next, I read through the GraphBuilder class and the graph10.graphml file to understand how it reads in the file. The GraphBuilder class will read in the graphml file and separate the nodes from the edges, it then separates the edges into the edge id, the source node, the target node, and the weight of the edge. Now that I understood how the GraphBuilder worked it was easy to add in the necessary framework to build a graph. Initially when I built the graph, I only used the edges to build the graph but when done this way, the GraphBuilder does not include the nodes that have no edges. Thus, I changed the builder so that it would enter all the nodes into the graph first, and then enter in all the edges.

```
data/graphs/graph10.graphml 0 true
Vertices: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0: 1 6
1: 0 4 6
2: 0 4 5
3: 7 9
4: 2 6 7
5: 2
6: 0 1 8
7: 3 4
8: 5 6 9
9: 3 8
```

Figure 1 Code output showing the graph and all edges.

## Part (b) Building an adjacency list/matrix.

My code in lab 3 already had the implementation of an adjacency matrix in the form of the adjacencyMatrix HashMap. This HashMap takes in a Vertex and a HashMap, which itself takes a vertex and an integer. The first HashMap then accessed with a vertex will return a HashMap with all vertexes that the source connects with along with their weights. HashMap was used for its fast access time and because we know the maximum size of the vertices list (100). This means

that the HashMap will not have to be resized and rehashed which would lower the performance. The  $O(1)$  access time means that the constant accessing will not take a large amount of time.

## Part (c) Dijkstra's algorithm

Dijkstra's algorithm takes a lot of the code from the lab 3 searching function and repurposes it for Dijkstra's algorithm. The original idea was to create another adjacency matrix that could be used to hold the current shortest path to the node as well as a separate list that would track the status of the different nodes. I soon realised in the implementation that it would be much faster and easier to use the prebuilt VertexState variable to track whether the nodes distance had been finalised and simply make a HashMap for the current distances, using the node as a key and the distance as the value. When called the DijkSearch will initially set the distance for the source node as 0 and the rest will be set to 999999 as there is no edge with a weight that high. The function will then call a recursive function that will find all neighbours of the given node and update their tentative distances and tentative previous nodes. It will then iterate through those neighbours again and find the smallest one that has its state not set to FINISHED. If there is no available node then it will terminate but otherwise it will run again with the smallest unfinished neighbour node.

$n$  is equal to the total number of vertices in the graph.

For time complexity

Each Vertex can be connected to  $n-1$  vertices

Setting each nodes distance is  $n$

Running the actual search function is done  $n(n+1)/2$  times

Searching through neighbours is  $n$

Finding the smallest node to go to next is  $n$

The time complexity is  $O(n + ((n(n+1)/2) * (n+n)))$

Becomes  $O(n + ((n^2+n)/2n) * (2n^2))$

Becomes  $O(n + (2n(n^2+n)))$

Becomes  $O(2n^3 + 2n^2)$

Becomes  $O(n^3)$

Thus the time complexity of my dijkstra's algorithm is  $O(n^3)$

The prev and dist HashMaps are equal to  $n$

Not counting the space taken up by each individual nodes, the space complexity would be  $O(n(n-1) + n + n)$

Becomes  $O(n^2 - n + n + n)$

Becomes  $O(n^2+n)$

Becomes  $O(n^2)$

Thus the space complexity is  $O(n^2)$

When I first created the graph it would output the path backwards. In order to fix this I reversed the order of the output string. This had the side effect of also reversing all of the node labels, i.e. 10 would become 01. This was solved by reversing the node list and then casting it to a string so that it would output correctly.

1. random_v100_e500_w50.graphml.1	
Expected output:	Actual output:
1. shortest path to 0: 1 36 57 0: cost = 25	1. shortest path to 0: 1 63 75 0: cost = 25
2. shortest path to 1: 1: cost = 0	2. shortest path to 1: 1: cost = 0
3. shortest path to 2: 1 45 51 2: cost = 51	3. shortest path to 2: 1 01: cost = 10
4. shortest path to 3: 1 59 79 3: cost = 67	4. shortest path to 3: 1 01 57 11: cost = 26
5. shortest path to 4: 1 45 51 98 4: cost = 61	5. shortest path to 4: 1 01 36 21: cost = 42
6. shortest path to 5: 1 10 25 5: cost = 36	6. shortest path to 5: 1 63 75 0 77 44 31: cost = 65
7. shortest path to 6: 1 36 6: cost = 22	7. shortest path to 6: 1 01 57 27 41: cost = 63
8. shortest path to 7: 1 10 25 5 49 7: cost = 72	8. shortest path to 7: 1 63 51: cost = 38
9. shortest path to 8: 1 10 63 12 8: cost = 69	9. shortest path to 8: 1 63 6 61: cost = 61
10. shortest path to 9: 1 10 75 92 34 9: cost = 59	10. shortest path to 9: 1 63 92 71: cost = 56
11. shortest path to 10: 1 10: cost = 10	11. shortest path to 10: 1 01 57 47 81: cost = 61
12. shortest path to 11: 1 10 75 11: cost = 26	12. shortest path to 11: 1 01 57 29 43 91: cost = 57
13. shortest path to 12: 1 10 63 12: cost = 42	13. shortest path to 12: 1 54 15 2: cost = 51
14. shortest path to 13: 1 36 57 0 77 44 13: cost = 65	14. shortest path to 13: 1 54 15 89 53 02: cost = 69
15. shortest path to 14: 1 10 75 72 14: cost = 63	15. shortest path to 14: 1 63 12: cost = 51
16. shortest path to 15: 1 36 15: cost = 38	16. shortest path to 15: 1 63 22: cost = 48
17. shortest path to 16: 1 36 6 16: cost = 61	17. shortest path to 16: 1 01 57 68 32: cost = 52
18. shortest path to 17: 1 36 29 17: cost = 56	18. shortest path to 17: 1 63 6 42: cost = 38
19. shortest path to 18: 1 10 75 74 18: cost = 61	19. shortest path to 18: 1 01 52: cost = 21
20. shortest path to 19: 1 10 75 92 34 19: cost = 57	20. shortest path to 19: 1 01 57 29 62: cost = 54
21. shortest path to 20: 1 45 51 98 35 20: cost = 69	21. shortest path to 20: 1 63 75 0 77 72: cost = 58
22. shortest path to 21: 1 36 21: cost = 51	22. shortest path to 21: 1 01 36 21 8 82: cost = 71
23. shortest path to 22: 1 36 22: cost = 48	23. shortest path to 22: 1 63 92: cost = 53
24. shortest path to 23: 1 10 75 86 23: cost = 52	24. shortest path to 23: 1 95 97 3: cost = 67
25. shortest path to 24: 1 36 6 24: cost = 38	25. shortest path to 24: 1 63 92 03: cost = 61
26. shortest path to 25: 1 10 25: cost = 21	26. shortest path to 25: 1 01 52 85 59 13: cost = 49
27. shortest path to 26: 1 10 75 92 26: cost = 54	27. shortest path to 26: 1 54 15 23: cost = 46
28. shortest path to 27: 1 36 57 0 77 27: cost = 58	28. shortest path to 27: 1 54 15 89 53 33: cost = 61
29. shortest path to 28: 1 10 63 12 8 28: cost = 71	29. shortest path to 28: 1 01 57 29 43: cost = 46
30. shortest path to 29: 1 36 29: cost = 53	30. shortest path to 29: 1 54 15 89 53: cost = 56
31. shortest path to 30: 1 36 29 30: cost = 61	31. shortest path to 30: 1 63: cost = 14
32. shortest path to 31: 1 10 25 58 95 31: cost = 49	32. shortest path to 31: 1 01 36 87 73: cost = 40
33. shortest path to 32: 1 45 51 32: cost = 46	33. shortest path to 32: 1 83: cost = 47
34. shortest path to 33: 1 45 51 98 35 33: cost = 61	34. shortest path to 33: 1 01 57 29 62 93: cost = 62
35. shortest path to 34: 1 10 75 92 34: cost = 46	35. shortest path to 34: 1 54 15 89 4: cost = 61
36. shortest path to 35: 1 45 51 98 35: cost = 56	36. shortest path to 35: 1 63 6 04: cost = 41
37. shortest path to 36: 1 36: cost = 14	37. shortest path to 36: 1 01 57 27 07 14: cost = 72
38. shortest path to 37: 1 10 63 78 37: cost = 40	38. shortest path to 37: 1 01 36 21 78 24: cost = 48
39. shortest path to 38: 1 38: cost = 47	39. shortest path to 38: 1 01 36 21 34: cost = 54
40. shortest path to 39: 1 10 75 92 26 39: cost = 62	40. shortest path to 39: 1 63 75 0 77 44: cost = 64
41. shortest path to 40: 1 36 6 40: cost = 41	41. shortest path to 40: 1 54: cost = 26
42. shortest path to 41: 1 10 75 72 70 41: cost = 72	42. shortest path to 41: 1 01 52 85 59 35 64: cost = 85
43. shortest path to 42: 1 10 63 12 87 42: cost = 48	43. shortest path to 42: 1 01 52 5 74: cost = 78
44. shortest path to 43: 1 10 63 12 43: cost = 54	44. shortest path to 43: 1 63 25 84: cost = 67
45. shortest path to 44: 1 36 57 0 77 44: cost = 64	45. shortest path to 44: 1 01 52 5 94: cost = 48
46. shortest path to 45: 1 45: cost = 26	46. shortest path to 45: 1 01 52 5: cost = 36
47. shortest path to 46: 1 10 25 58 95 53 46: cost = 85	47. shortest path to 46: 1 01 52 85 59 13 38 05: cost = 92
48. shortest path to 47: 1 10 25 5 47: cost = 78	48. shortest path to 47: 1 54 15: cost = 27

Figure 2 The path function outputting in reverse.

Once this problem was solved I was able to test it against the given graphs where it all came out correct.

```

from clipboard
shortest path to 0: 0: cost = 0
shortest path to 1: 0 50 52 49 48 22 1: cost = 44
shortest path to 2: 0 50 97 2: cost = 42
shortest path to 3: 0 33 86 91 3: cost = 32

shortest path to 97: 0 50 97: cost = 22
shortest path to 98: 0 50 42 6 98: cost = 34
shortest path to 99: 0 50 62 99: cost = 11

1 1 << C:\Users\foong\.jdk\openjdk-17.0.2\bin\java.exe "-
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 17
18 18
19 19
20 20
21 21
22 22
23 23
24 24
25 25
26 26
27 27
28 28
29 29
30 30
31 31
32 32
33 33
34 34
35 35
36 36
37 37
38 38
39 39
40 40
41 41
42 42
43 43
44 44
45 45
46 46
47 47
48 48
49 49
50 50
51 51
52 52
53 53
54 54
55 55
56 56
57 57
58 58
59 59
60 60
61 61
62 62
63 63
64 64
65 65
66 66
67 67
68 68
69 69
70 70
71 71
72 72
73 73
74 74
75 75
76 76
77 77
78 78
79 79
80 80
81 81
82 82
83 83
84 84
85 85
86 86
87 87
88 88
89 89
90 90
91 91
92 92
93 93
94 94
95 95
96 96
97 97
98 98
99 99
100 100
101 101
102 102
103 103
104 104
105 105

```

Figure 3 comparison between the biggest graph and output of the function.

## Part (e) Comparison between Arrays and Priority Queue

To compare the performance of both the list and the priority queue, a new Dijkstra's function needed to be created that would run using a Priority Queue instead of a list. As HashMaps are built on arrays, we can use the original Dijkstra function as a comparison. While the DijkstraPrior function uses a integer array to hold distances, it does not have to search through the matrix to find the next node to run its function on, instead it uses a priority queue to hold all potential nodes to run the function on. This priority queue uses a custom object called vComp, which holds the label, the node, and the weight of the edge its associated with. This object was created as I did not have a way to store both the nodes and their associated distances in the priority queue. The distance is being held in an array whose size is equal to the total number of vertices in the graph. An array was chosen as it has an  $O(1)$  modify element time and we know how large the array needs to be when creating it. the first thing that is added to the priority queue is the source vertex with 0. Every other vertex is added to the queue with 999999. While the queue is not empty, the queue will pull out the smallest element and for each of the neighbouring vertices, the function will check if the distance from the smallest element to the neighbouring vertices is smaller than the saved distance. If it is then it will save the path between vertices, save the current distance as the smallest distance, and add the new path to the queue.

n is equal to the total number of nodes in the graph

Time complexity

First for loop  $O(n)$

While loop  $O(n(n-1))$

Becomes  $O(n^2-n+n)$

Becomes  $O(n^2)$

Thus the time complexity of the priority queue is  $O(n^2)$

Space complexity

Dist is  $O(n)$

Prev is  $O(n)$

dQueue is  $O(n^2)$

Becomes  $O(n^2+2n)$

Thus the space complexity of the priority queue is  $O(n^2)$

In comparison to the list based function the queue based function is much faster, being  $O(n^3)$  compared to  $O(n^2)$ . While the space complexity remains relatively the same for the two functions as, in the worst case scenario, they would both have to store all of the edges in the graph. As can be seen below in the smallest random graph the queue based function is over 1000000 nanoseconds faster than the list based function. When the functions are run on the largest graph the difference remains the same, about 1000000 nanoseconds.

```
List took 6902600 nanoseconds
Priority queue took 5013100 nanoseconds
```

Figure 4 Both list and queue-based functions running on random\_v10\_e10\_w50.

```
List took 2783300 nanoseconds
Priority queue took 1736700 nanoseconds
```

Figure 5 Both list and queue-based functions running on random\_v100\_e1000\_w50.

The function's output was compared to the given output, and it matched, confirming that the function worked.

```
From clipboard
shortest path to 0: 0: cost = 0
shortest path to 1: 0 50 52 49 48 22 1: cost = 44
shortest path to 2: 0 50 97 2: cost = 42
shortest path to 3: 0 33 86 91 3: cost = 32
shortest path to 4: 0 50 62 99 92 85 4: cost = 42
shortest path to 5: 0 50 62 99 92 81 5: cost = 34
shortest path to 6: 0 50 42 6: cost = 24
shortest path to 7: 0 50 62 99 92 28 15 7: cost = 27

shortest path to 92: 0 50 62 99 92: cost = 13
shortest path to 93: 0 50 62 93: cost = 25
shortest path to 94: 0 33 19 96 63 94: cost = 31
shortest path to 95: 0 50 62 95: cost = 40
shortest path to 96: 0 33 19 96: cost = 27
shortest path to 97: 0 50 97: cost = 22
shortest path to 98: 0 50 42 6 98: cost = 34
shortest path to 99: 0 50 62 99: cost = 11

Editor
1 1 << C:\Users\foong\.jdk\openjdk-17.0.2\bin\java.exe "-jav
2 2
3 3
4 4 shortest path to 0: 0: cost = 0
5 5 shortest path to 1: 0 50 52 49 48 22 1: cost = 44
6 6 shortest path to 2: 0 50 97 2: cost = 42
7 7 shortest path to 3: 0 33 86 91 3: cost = 32
8 8 shortest path to 4: 0 50 62 99 92 85 4: cost = 42
9 9 shortest path to 5: 0 50 62 99 92 81 5: cost = 34
10 10 shortest path to 6: 0 50 42 6: cost = 24
11 11 shortest path to 7: 0 50 62 99 92 28 15 7: cost = 27
12 12
13 13
14 14 shortest path to 92: 0 50 62 99 92: cost = 13
15 15 shortest path to 93: 0 50 62 93: cost = 25
16 16 shortest path to 94: 0 33 19 96 63 94: cost = 31
17 17 shortest path to 95: 0 50 62 95: cost = 40
18 18 shortest path to 96: 0 33 19 96: cost = 27
19 19 shortest path to 97: 0 50 97: cost = 22
20 20 shortest path to 98: 0 50 42 6 98: cost = 34
21 21 shortest path to 99: 0 50 62 99: cost = 11
22 22
23 23
24 24 Operation took 91795300 nanoseconds
25 25
26 26 Process finished with exit code 0
```

Figure 6 Comparison between the queue-based function's output and the given output.

## Part (f) Custom Priority Queue using heap.

The custom priority queue was created using a heap data structure. This heap was represented in two lists, one was for the distance from the source vertex to any given vertex and the other was for the matching vertex. A list was used to allow for easy adding and removal from the data structures. The priority queue utilised the formula given on the slides to determine the appropriate positions that need to be swapped. The formula to find the left child is  $2 \cdot \text{pos} + 1$ , the formula to find the right child is  $2 \cdot \text{pos} + 2$  and the formula to find the parent of the element is  $((\text{pos} - 1) / 2)$ . To shift up the element will compare itself to its parent, if it is bigger than its parent it will swap positions with it until it is smaller than its parent. The shift down function will first find the largest child, swap the two elements, and recursively call itself until the children are smaller than the current element. The swap function not only swaps the positions of the heap, but it also swaps the position of the matching vertex in the vertices list that way both lists remain matching to each other. When popping the element at the front of the queue it will copy the elements to a temporary variable, remove the first element of both lists, shift down the first element of both lists to remain the ordering of the lists and then return the vertex that was in the temporary variable. A new function to run the custom priority queue was created to test and verify that it works. Below is a screenshot verifying that the function works using the most complicated graph. There is one difference between the given output and the output from the function but, this difference is merely in the path chosen to get to the node, the total weight of the path remains the same.

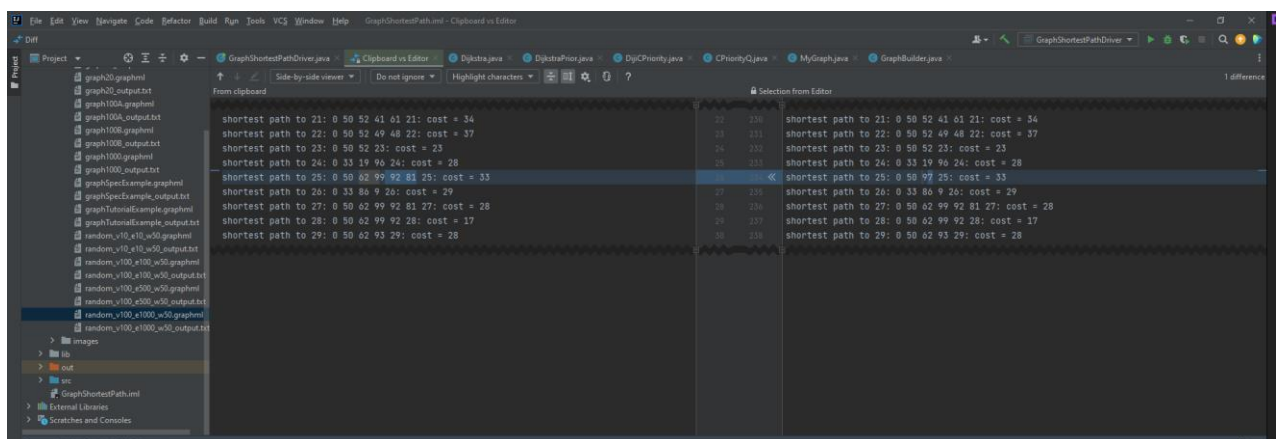
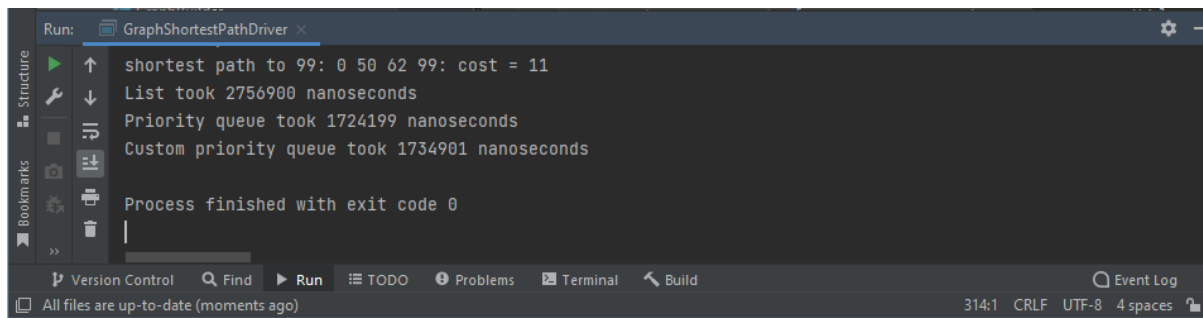


Figure 7 Comparison between the queue-based function's output and the given output.

Given that my implementation of a priority queue is inefficient compared to the inbuilt function, it is safe to assume that the time and space complexity will be slightly worse. As the big O notation only cares about the largest order of  $n$ , however, the big O of my custom priority queue will be the same. Thus, time complexity is  $O(n^2)$  and space complexity is  $O(n^2)$ .



The screenshot shows a terminal window titled 'Run: GraphShortestPathDriver'. The output text is as follows:

```
shortest path to 99: 0 50 62 99: cost = 11
List took 2756900 nanoseconds
Priority queue took 1724199 nanoseconds
Custom priority queue took 1734901 nanoseconds

Process finished with exit code 0
```

The terminal interface includes a sidebar with 'Structure' and 'Bookmarks' views, and a bottom status bar showing 'All files are up-to-date (moments ago)', '314:1 CRLF UTF-8 4 spaces', and an 'Event Log' icon.

Figure 8 Comparing the runtime of the three functions on the `random_v100_e1000_w50` graph.

As can be seen the custom priority queue almost equal to the inbuilt priority queue but is just 10000 nanoseconds slower. The custom priority queue could be improved further; however, I do not know how to implement batch inserting.