For both parts of the question, we take n = |V| = number of nodes in the graph and e = |E| = number of edges in the graph.

We use Python in this project for its ease in utilising data structures. Additionally, we start every shortest-path traversal from node 0.

PROJECT 2 (A)

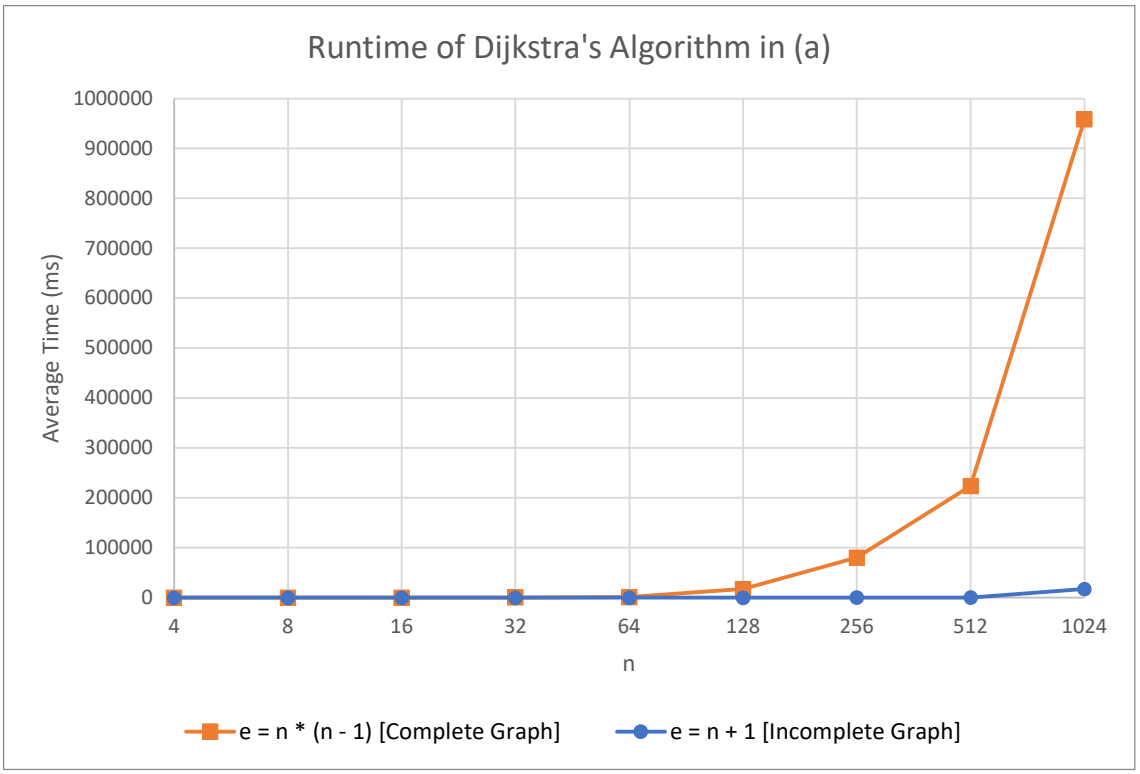
Given n, e, our Adjacency Matrices for our directed graphs are generally randomised in the following algorithm:

1. Generate e random tuples (as vertices in the matrix), without repetition and excluding the diagonal vertices, where edges will be produced.
2. Assign a random value 1-100 as the weight to the aforementioned edges.
3. If there is no edge, i.e. no connection between any two nodes, assign a sentinel value to the matrix vertex.
4. Assign 0 to all vertices in the diagonal of the matrix.

As you may be able to see, there may be cases where the graph is not fully connected. Our Dijkstra's Algorithm code accounts for this, and returns the infinity value as a distance should a path from 0 to some node not be available.

For this part, we look at two types of graphs: a completed graph, i.e. e = n * (n – 1) and a graph with a relatively low number of edges, in this case we take e = n + 1.

First, the complete graph. We iterate n in powers of 2 from 4 to 1024 and take e = n * (n – 1), the maximum number of edges a graph can have. We do 5 trials and take the average CPU time (in microseconds).



Naturally, as n (and by extension e) increases, the runtime of Dijkstra's Algorithm generally increases for both graphs. The general trend is that as n increases from 512 to 2014, the runtime taken takes a larger jump.

As it is difficult to properly view the data points of the incomplete graph, the data points are given below. Note that the variance of runtime throughout the 5 trials across a single n, e is relatively high.

n	e	$time_1$	$time_2$	$time_3$	$time_4$	$time_5$	average
4	5	21.21925354	31.9480896	28.61022949	23.36502075	24.31869507	25.89225769
8	9	14.30511475	31.70967102	12.87460327	43.86901855	12.15934753	22.98355103
16	17	46.49162292	72.47924805	32.90176392	18.59664917	24.31869507	38.95759583
32	33	40.29273987	292.0627594	54.35943604	243.9022064	52.21366882	136.5661621
64	65	57.45887756	89.64538574	58.1741333	56.98204041	52.92892456	63.03787231
128	129	103.2352448	453.9489746	461.5783691	84.400177	58.88938904	232.4104309
256	257	133.5144043	418.4246063	310.6594086	466.3467407	160.2172852	297.832489
512	513	229.8355103	486.3739014	2177.476883	329.4944763	458.4789276	736.3319397
1024	1025	475.6450653	3078.460693	709.0568542	804.6627045	83123.68393	17638.30185

Sampled runtimes for e = n + 1 (ms)

For instance, when n = 1024 and e = 1025, the timings can range from 475 microseconds to 83123 microseconds. We believe this is not a case of outliers however; this is rather expected. When e is relatively small, there is a much lower chance for the randomly-generated matrix to represent a strongly connected graph – as a result, a graph that is not strongly connected may undergo more iterations in Dijkstra's Algorithm, resulting in a higher runtime. Likewise, a randomly-generated matrix which happens to represent a strongly connected graph may require lesser runtime, mainly also due to the fact that there are lesser iterations to be covered especially when e is small.

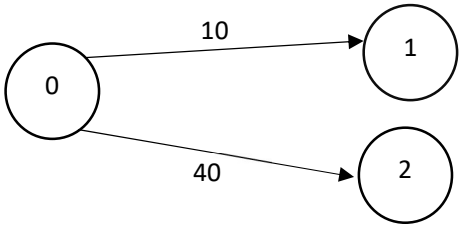
PROJECT 2 (B)

Given n, e, our Adjacency Lists are generally created using the Python Dictionary data type by the following algorithm:

1. Initialise an empty dictionary.
2. Each key in the dictionary represents the starting node (of the edge), whereas the value to each key stores an array of arrays in the form [target node of the edge, edge weight]
   Eg.

KEY	VALUE
0	[[1, 10], [2, 40]

   denotes

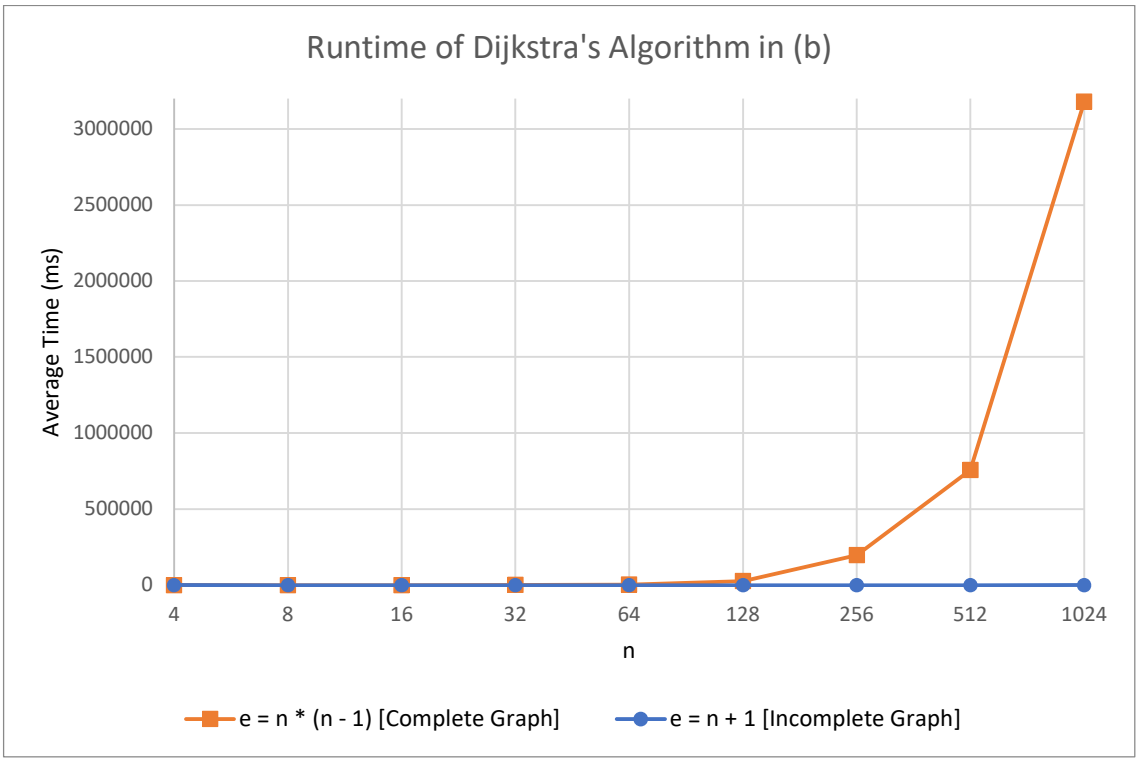


3. Randomise the starting and target node of which edges are touching.
4. Append the starting node as a key, with its value as the list in the form [target node, edge weight] where the edge weight is a random number 1-100.

Like in (a), we accept our graphs to not be strongly connected. An infinity value will be output should there not be a path from 0 to some node.

In this Dijkstra's Algorithm, we take the Priority Queue as a heap using the heapq library (which defaults to a minimising heap), where each value is a tuple in the form (distance from 0, node). After a node is visited, it is removed from the queue and its neighbouring node with the shortest distance is pushed into the queue.

Again, we will be looking at the complete graph (e = n * (n – 1)) and an incomplete graph with a relatively low number of edges (e = n + 1).



Once again, the complete graph takes on a steep jump in runtime as n increases from 512 to 1024. For the incomplete graph, the runtimes appear to take on relatively low values as n increases. Focusing on its data points,

n	e	$time_1$	$time_2$	$time_3$	$time_4$	$time_5$	average
4	5	5.960464478	4.529953003	5.006790161	10.49041748	8.106231689	6.818771362
8	9	6.675720215	6.198883057	19.31190491	17.40455627	16.68930054	13.256073
16	17	6.437301636	6.198883057	15.25878906	5.722045898	11.68251038	9.059906005
32	33	35.04753113	19.31190491	13.11302185	12.63618469	10.96725464	18.21517944
64	65	71.52557373	13.58985901	13.35144043	29.80232239	39.33906555	33.52165222
128	129	14.7819519	33.85543823	23.36502075	15.25878906	27.89497375	23.03123474
256	257	25.03395081	25.74920654	30.27915955	37.90855408	44.82269287	32.75871277
512	513	54.83627319	52.69050598	104.1889191	101.0894775	52.92892456	73.14682007
1024	1025	109.4341278	118.4940338	158.5483551	105.1425934	185.251236	135.3740692

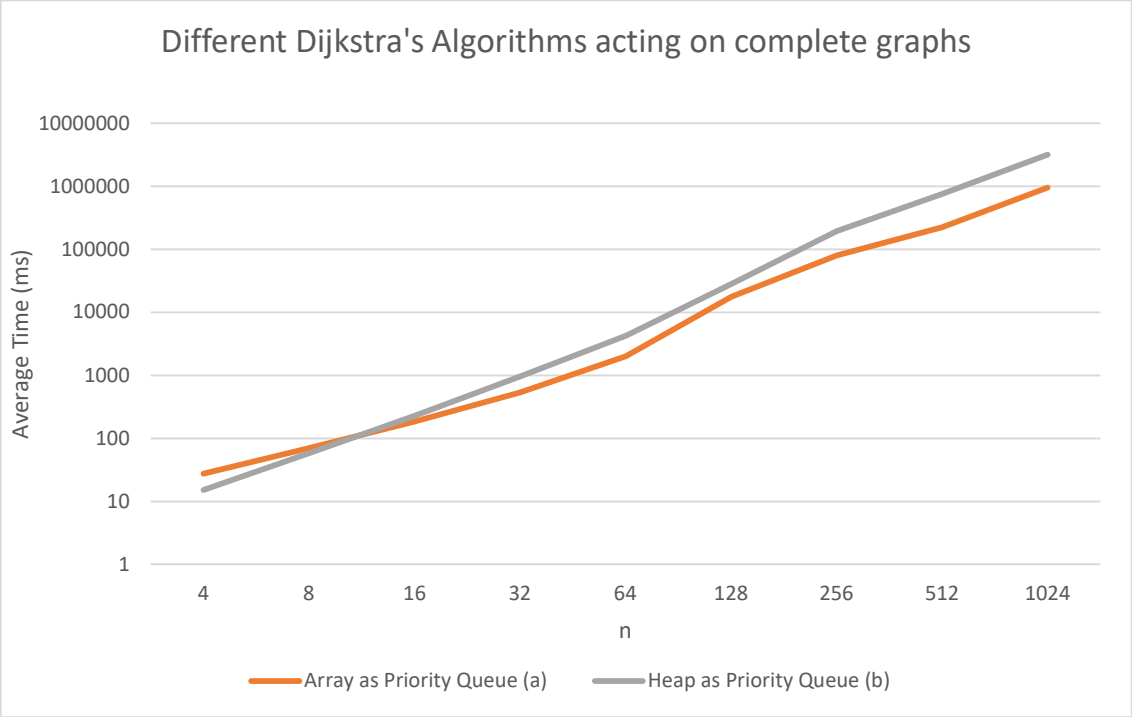
Sampled runtimes for e = n + 1 (ms)

we are actually able to see a jump in runtime from n=512 to n=1024 as well. Like before, we can see that the variance in timings is relatively large. This would again be due to the connectivity of the randomly-generated graph leading to the differing amounts of iterations during Dijkstra's Algorithm.

Overall

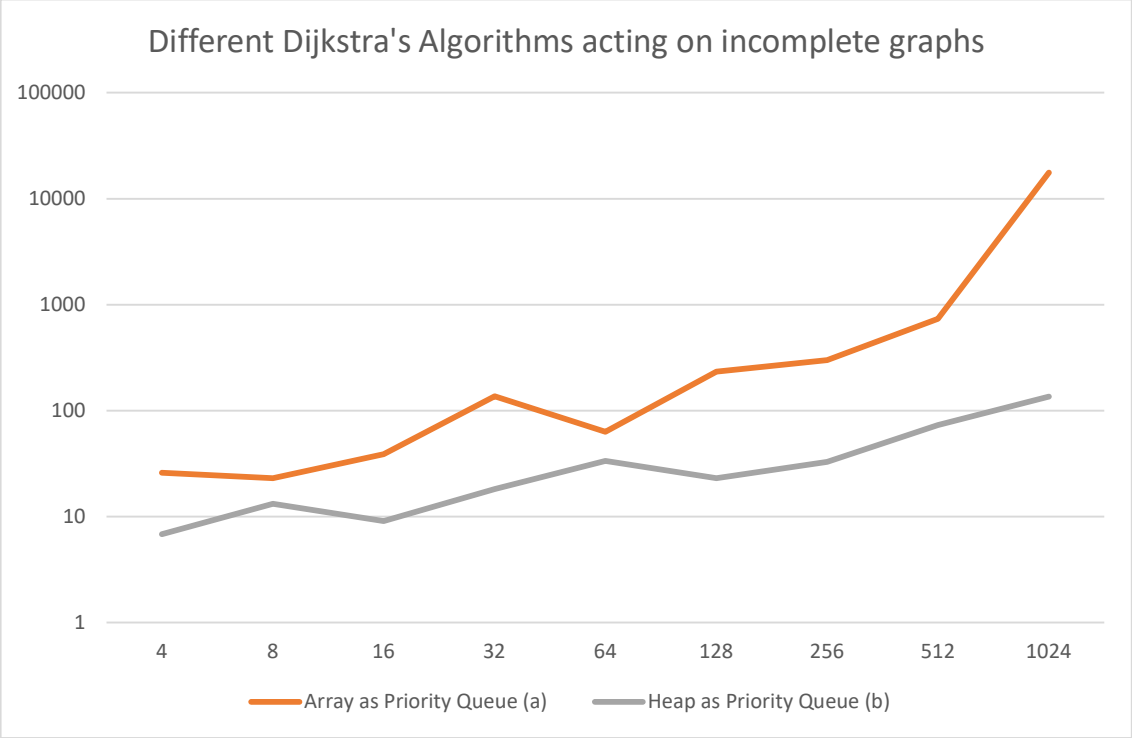
We need to do a comparison of both (a) and (b). Though they are both Dijkstra's Algorithms, the difference in implementation creates largely different runtimes for various values of n.

We fix the input graph to be complete graphs, and vary our n as before.



Note: the y-axis is $\log_{10}$

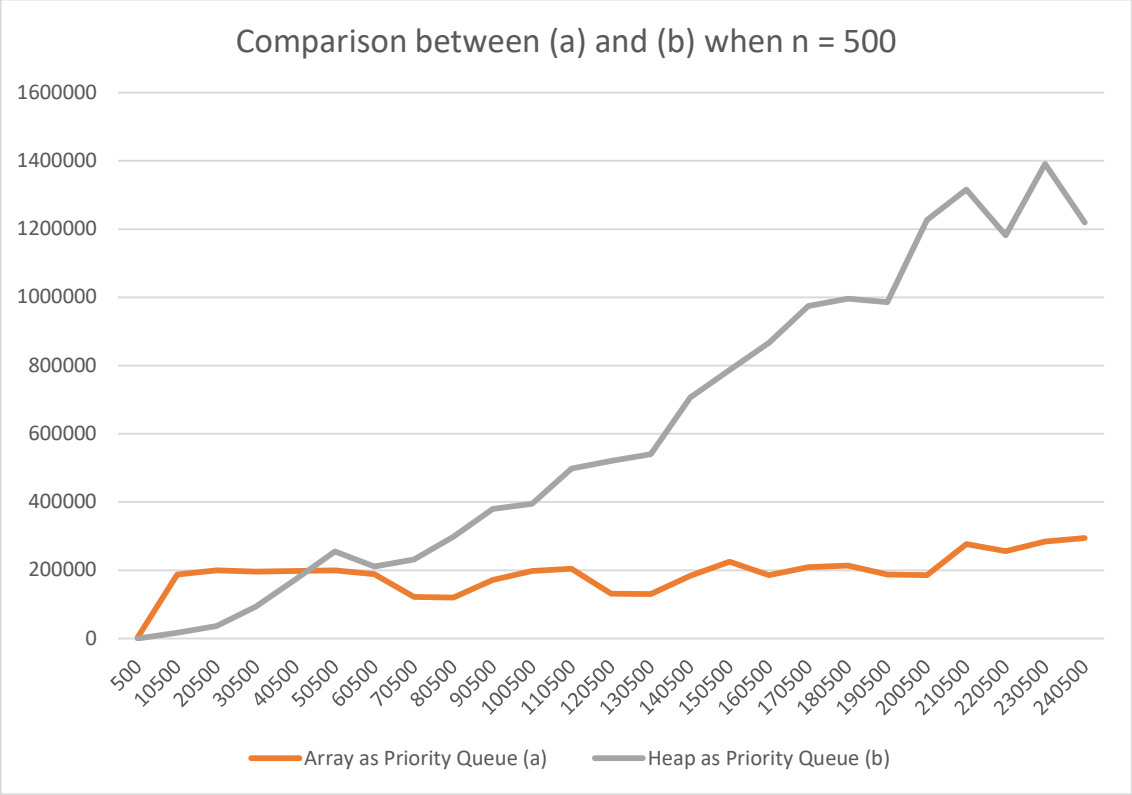
It can be observed that while (b) acts faster when n is small, the turning point comes when n reaches around 10, giving way for (a) to execute faster.



Note: the y-axis is $\log_{10}$

It appears that (b) executes faster on incomplete graphs with lower e values.

Next, we fix the input to be n = 500, and vary the e, which starts at 500 and is incremented by 1000 until we reach a complete graph.



We can see that while (b) starts out as a faster algorithm, it is quickly overtaken by (a) around e = 50500. When n is fixed, the runtime for (b) grows faster while that of (a) appears to remain relatively low – it appears to be independent of e.

To conclude, it appears that (a) is a faster algorithm when we have a large completed graph, whereas (b) is the superior algorithm when we have a smaller graph with little amount of edges. Moreover, (a) also appears to be largely unaffected by the size of e when we fix n an n.