# FINAL LAB PROJECT

COMPSCI 2XC3
April 16th , 2023
Dr. Vincent Maccio

Martin Matlok - matlokm
Nihal Singh - singhn77
Insiyah Ujjainwala - ujjainwi

# Table of Contents

# Table of Figures

# Executive Summary

- Dijkstra's approximation algorithm always outperforms Bellman-Ford's approximation algorithm. This statement has proved to be true when testing the algorithms on graphs of varying sizes (and graphs with varying edges for fixed nodes), varying relaxation times, and varying sources.
- Both the Dijkstra's approximation algorithm and the Bellman-Ford's approximation algorithm are quite accurate as the difference between the true shortest path and the estimated shortest path is quite small.
- A* algorithm is a heuristic version of Dijkstra's algorithm. Dijkstra's algorithm would outperform A* in the case where a random heuristic is being generated as A* would end up checking the nodes/paths in a random order whereas Dijkstra would have some plan/order to check the nodes. Hence, making it faster.
- When navigating through a dense map like that of the London subway system, we find that A* algorithm performs more efficiently than Dijkstra as A* knows the general direction to travel in.
- Code should be designed with design principles in mind before work is done, as refactoring code into proper design patterns with proper design principles is a tedious process that can easily be avoided with pre-planning.
-

# Part 1

For the first experiment, we observed the runtime for both the algorithms when the number of nodes and edges were varying. For each combination of nodes and edges, we set the source to 0 and the number of relaxations, k to 6. We then calculated the runtime for graphs with nodes starting from 10 and going up to 40 incrementing by 10 each time. For each case of the number of nodes, we added an increasing number of edges beginning with 20, going up to 80 in increments of 20.

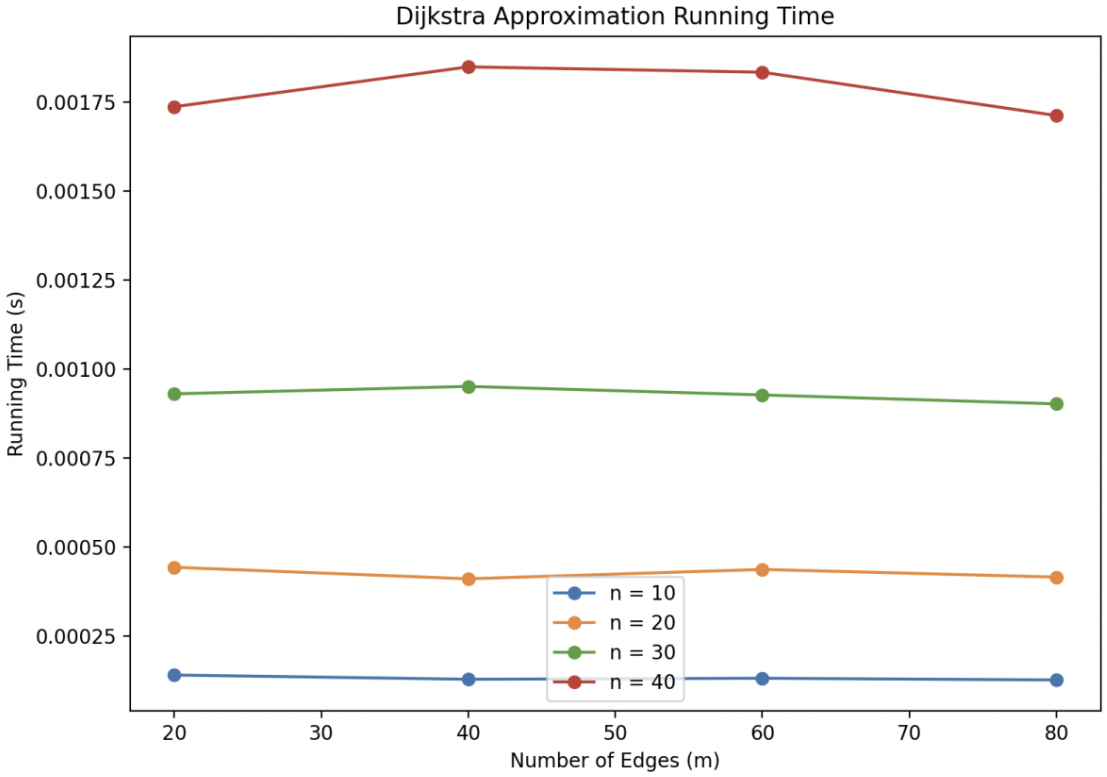


**Figure 1**

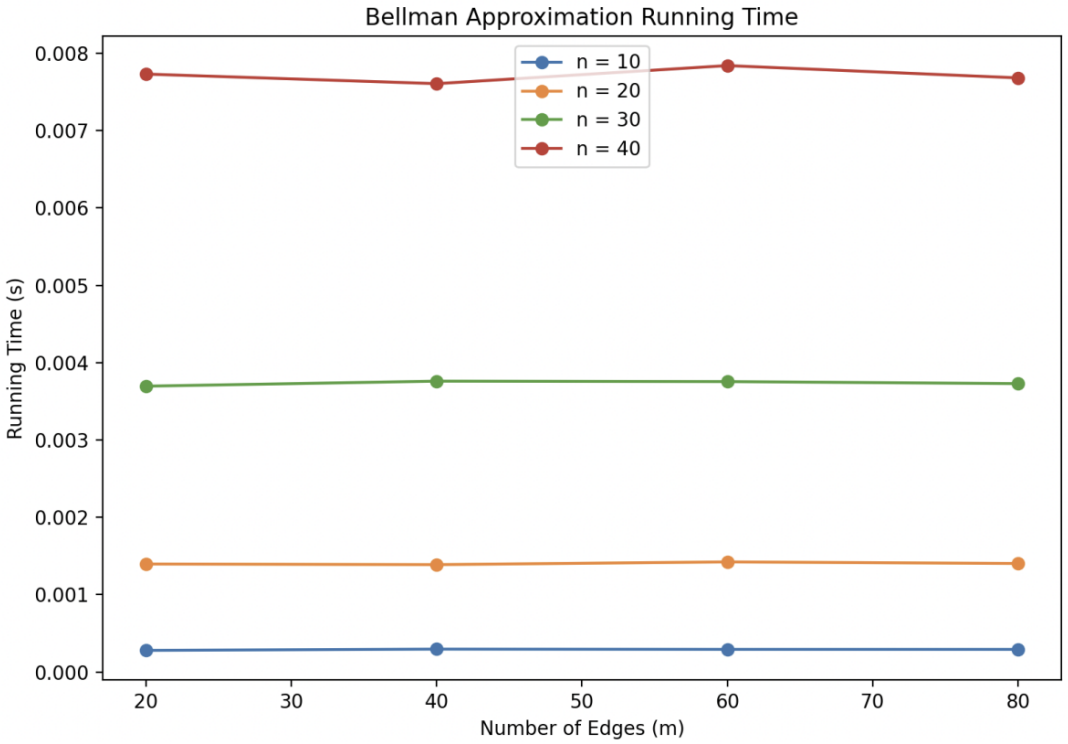


**Figure 2**

From the graphs above, we see that in both cases the runtime of the algorithm increases when the number of nodes increases. However, the runtime remains almost unchanged for a particular number of nodes. While both the algorithms portray the same general trends for increasing number of nodes and edges, we see that Dijkstra's algorithm always outperforms Bellman-Ford's algorithm.

For the second experiment, we observed the runtimes for both the algorithms for varying values of k. For this experiment, we create a single graph consisting of 100 nodes and 1000 edges, and set the source to be 0 for all computations. We have taken values of k that range from 1 to 10.

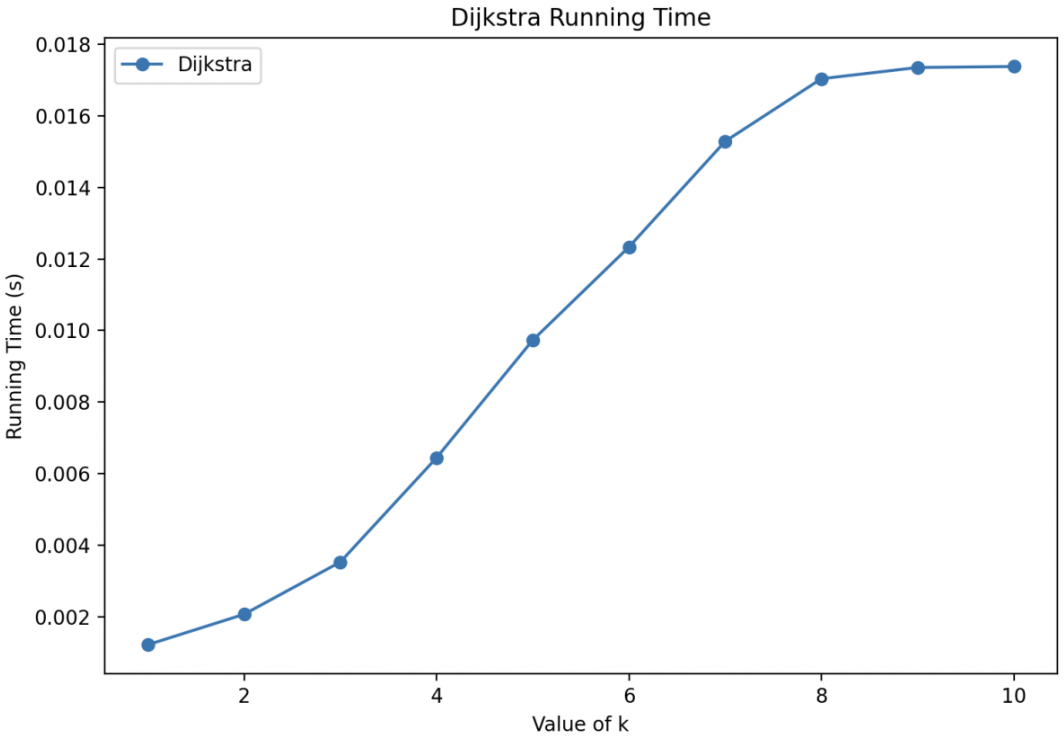


**Figure 3**

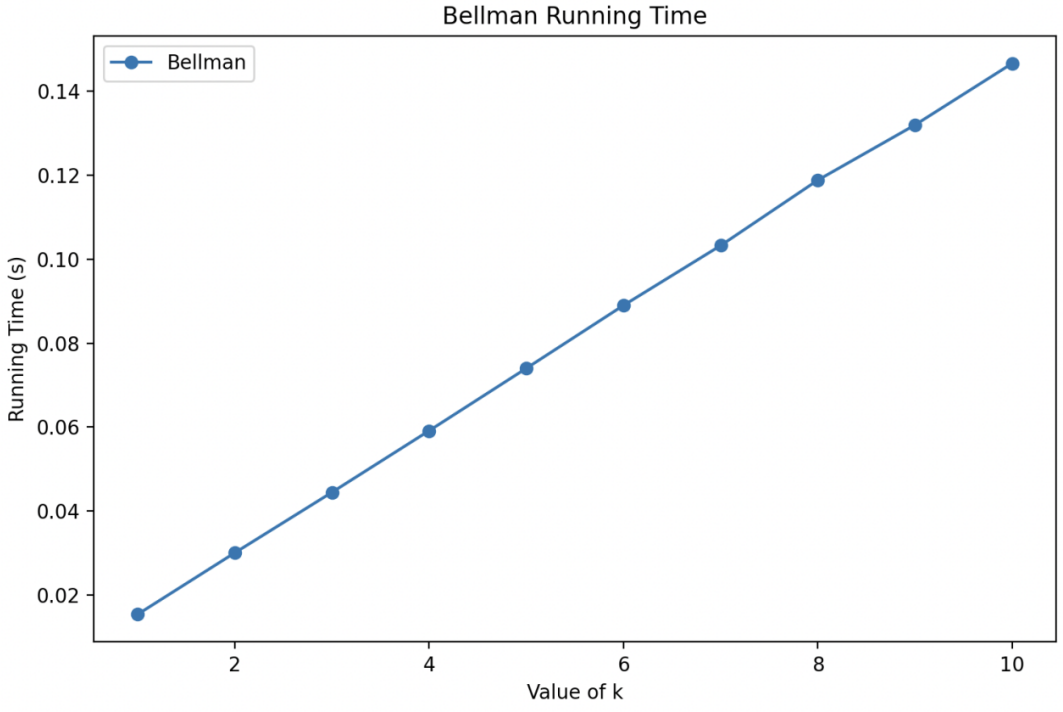


**Figure 4**

In general, we see that as k increases from 1 to 10 the runtimes of both algorithms will increase linearly with k. However, the actual runtimes of the algorithms depend on their implementations. We see that for the same graph, Dijkstra's approximation algorithm has a much lower runtime than the Bellman-Ford algorithm.

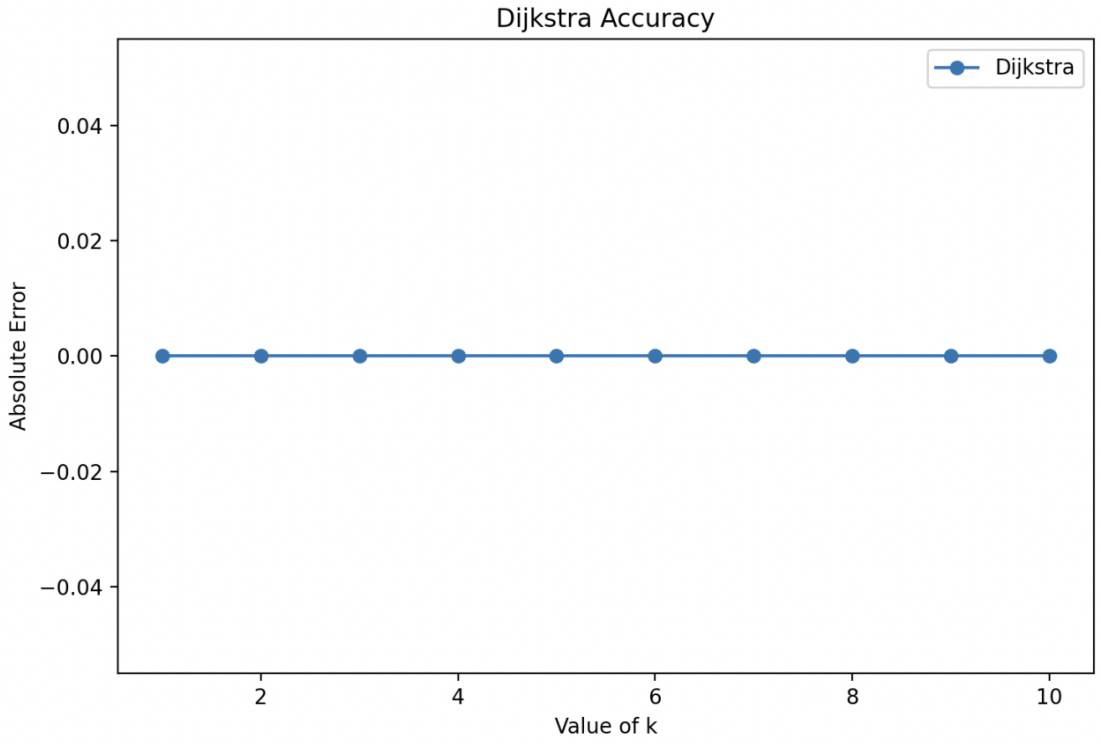


**Figure 5**

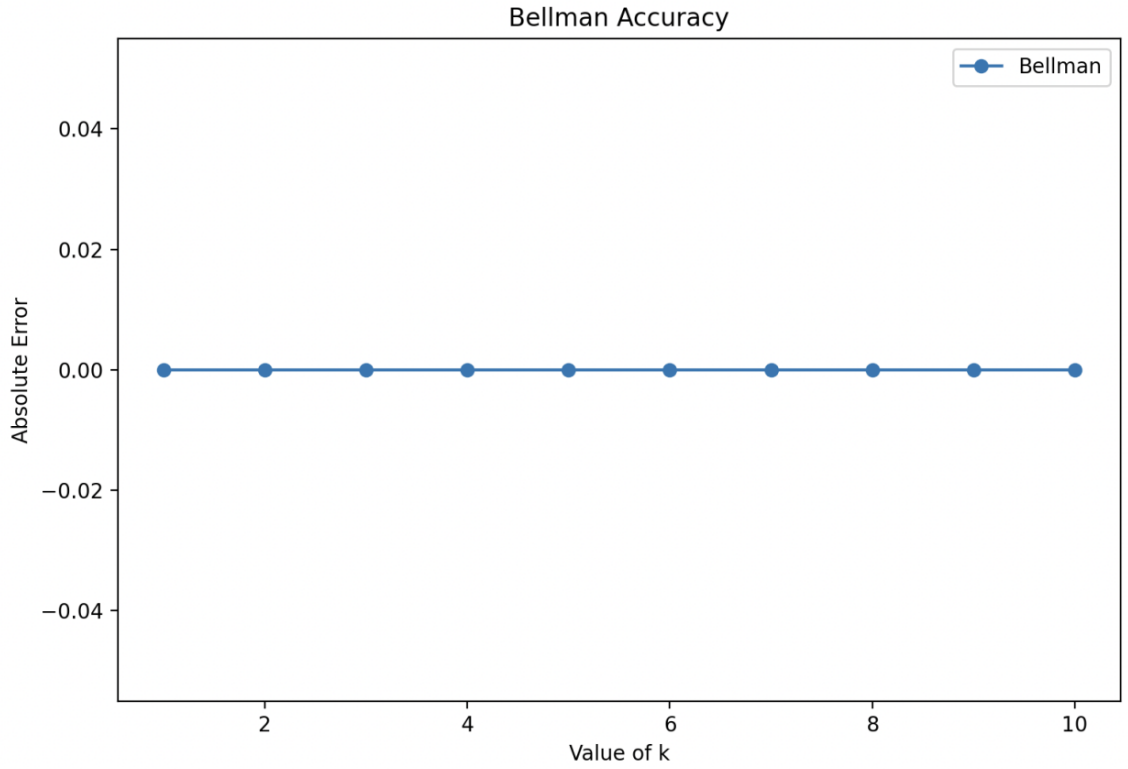


**Figure 6**

In this experiment, from the graphs above you can notice that we have calculated the absolute error based on the sum of distances returned by the dijkstra_approx() and

bellman_ford_approx() function. The real reason this is being done is to determine the accuracy of both Dijkstra and Bellman approximation algorithms with respect to the true shortest path distances in the graph. What the approximation algorithm is returning is an estimate of the shortest path distance between the source node and all the other nodes in the graph, which is not necessarily the true shortest path distance. The absolute error is the difference between the estimated path distance and the true path distance, the smaller the absolute error the more approximate the algorithm is.

For experiment 3, we observe the runtimes for both the algorithms when the shortest path is computed from a different source in the graph. For this experiment, we made 2 graphs, a small graph with 10 nodes and 20 edges, and a relatively larger graph with 100 nodes and 1000 edges. In both cases, each node could be relaxed up to 6 times.

**Case(i) : Graph with 10 nodes and 20 edges**

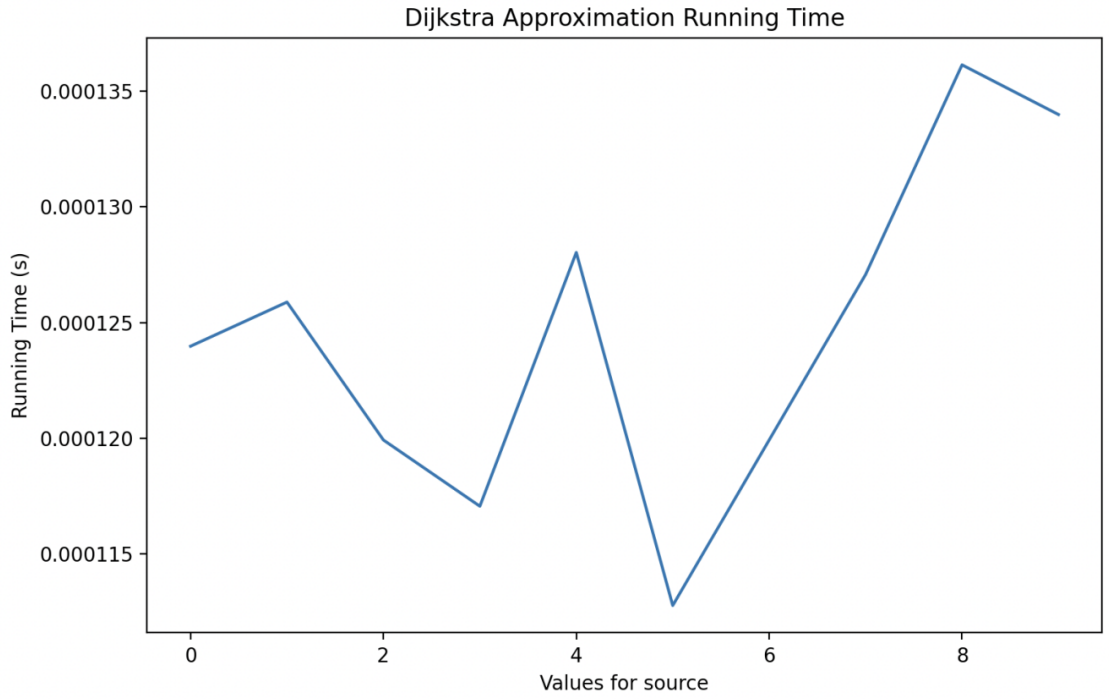


**Figure 7**

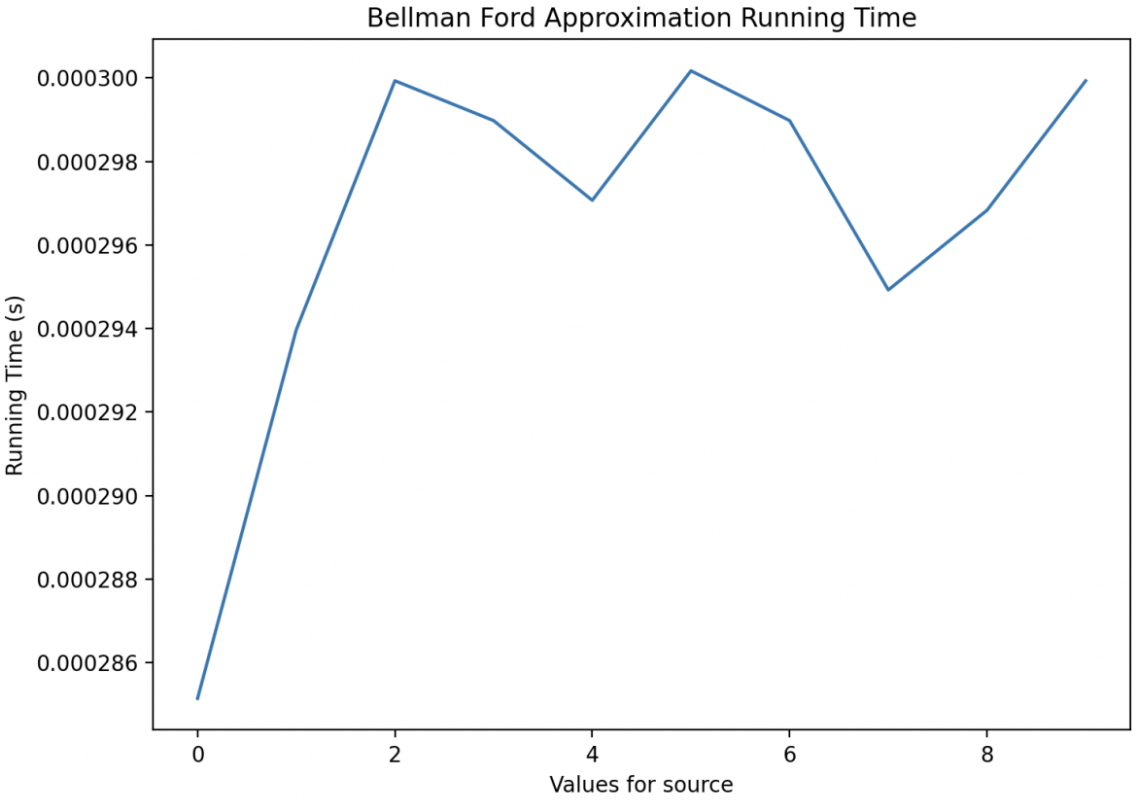


**Figure 8**

For both these graphs, we calculated the runtimes for all possible values of source (i.e. each node value from the graph). We see the runtimes for each algorithm is dependent on the source value and it is difficult to find a general trend in these graphs. However, once again, Dijkstra's algorithm outperforms Bellman-Ford in for all values of source.

**Case(ii): Graph with 100 nodes and 1000 edges**

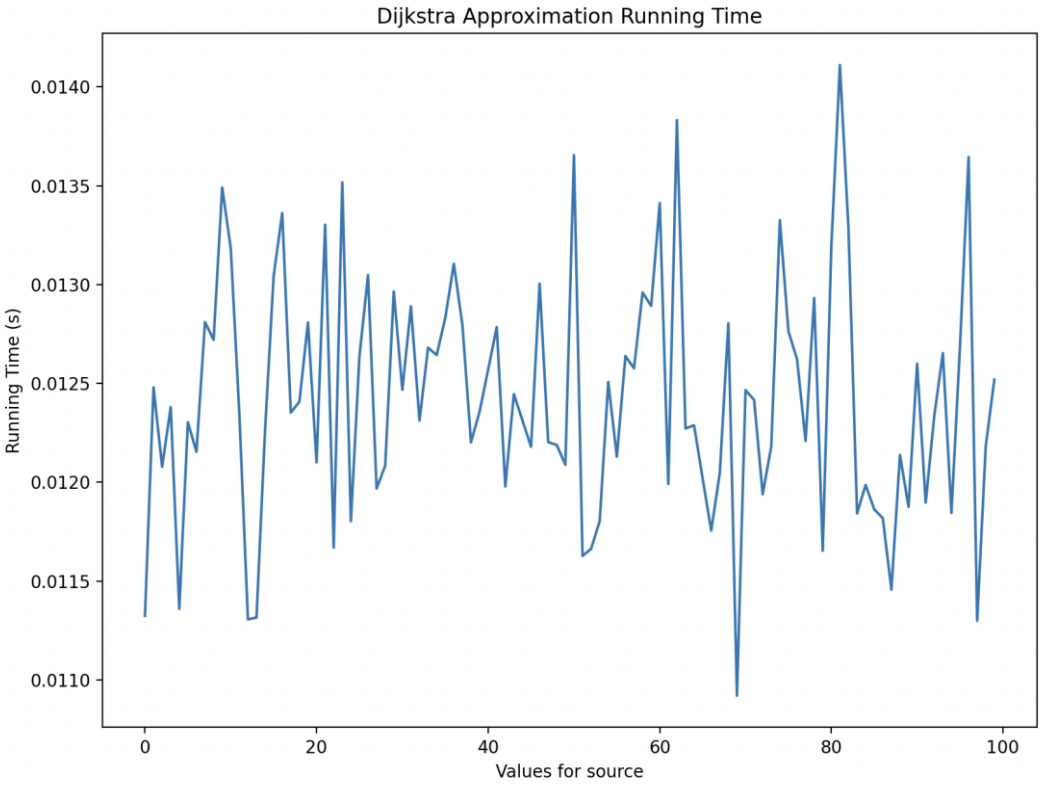


**Figure 9**

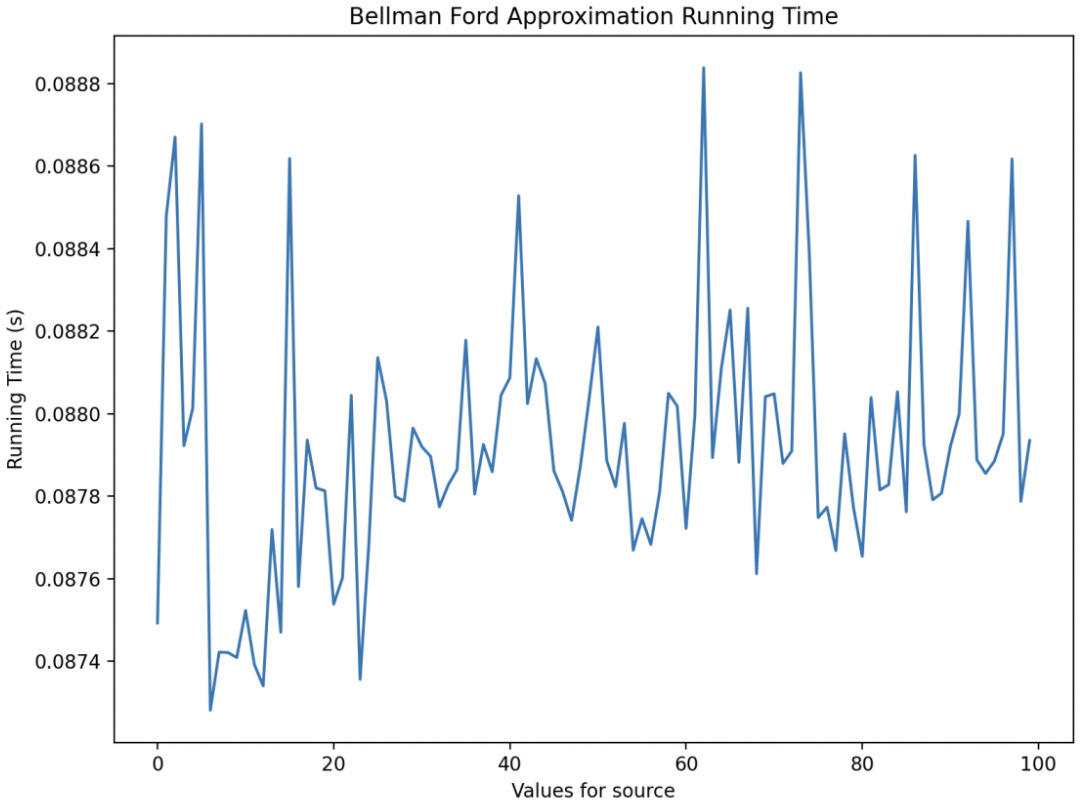


**Figure 10**

Similar to the previous case, for both these graphs, we calculated the runtimes for all possible values of source (i.e. each node value from the graph). Once again, we see that the runtimes are dependent on the source value, but Dijkstra's approximation algorithm performs far more efficiently than Bellman-Ford's approximation algorithm.

---

# Part 2

A* and Dijkstra's are essentially the same algorithm, but with one key difference: A* follows a heuristic, which essentially acts as a guide towards the right direction. The heuristic is some form of dictionary that returns the distance that each node is from the target node, ignoring anything in between them. In other words, if I were trying to get from MUSC to the Subway off campus, the Heuristic would provide the distance between the two as if I could walk through walls to get there.

Within the algorithm, it adds the distance of the node from the heuristic to the total distance from the starting node, and uses this total number for the priority queue. This means that nodes that go in the wrong direction are usually lower in the priority queue, and that nodes that go in the right direction are higher up. This results in the algorithm analyzing paths that are more direct to the final node first, which should speed up the processes of finding the path in most graphs.

The problem that A* is trying to fix with Dijkstra's is that Dijkstra's will always follow the shortest path, and ignore the direction it is going. This means in situations where we have two paths : One with a small weight in the wrong direction, and another with a larger weight in the correct direction, Dijkstra's will check the shorter path, realize it's wrong, and then go to the other, correct path. A* aim's to correct this by giving a sense of direction using the heuristic.

To empirically test A*, as its goal is to decrease the runtime of Dijkstra's, running comparisons of the runtime of the two on a graph would be sufficient. Realistically, it should be a large graph, with a combination of some "maze-like" areas and some areas that are very well connected. This gives the "optimal" conditions for both algorithms, as with a "maze-like" area, A*'s heuristic would not be that helpful. This idea will hopefully be proven in Part 3. In well-connected areas, A* should shine, as it should know which nodes are closer, giving it an edge over Dijkstras.

You could also keep track of how many nodes the two check and compare, but I believe comparing runtime would give more interesting and easy to analyze results.

If a heuristic function was randomly generated, I believe that Dijkstra's would outperform A*. With a random heuristic, A* would likely end up checking the nodes/paths in a random order, as the heuristic weight being added to each path would mean each node could be placed essentially anywhere in the priority queue. As Dijkstra's would still have a plan/order to check every node, I believe it would be more consistent, which means it would be faster on average.

Finally, when would you want to use A*? I believe that A* would be best used in mapping / navigation situations in very dense locations. Take for example, downtown Hamilton. Treat every intersection there as a node in the graph. There are a *lot* of nodes there, and they are all connected in a grid-like fashion. This means that Dijkstra's would struggle, as if each node is similar in distance to each other, it may cascade through almost the entire city before finding the correct path. A*, however, would know the general direction to take, making it perform much faster.

---

# Part 3

We began this experiment by making a list of pairs that contained all possible combinations of stations on the London subway map. For the sake of this experiment, we made slight modifications to the code of the Dijkstra algorithm provided to us. The modified function is **dijkstra(G, source, dest)** and it returns the distance from the source node to the destination node.

We then plotted a graph comparing the runtimes for A* and Dijkstra's algorithm for all possible pairs of stations. We obtained the following results:

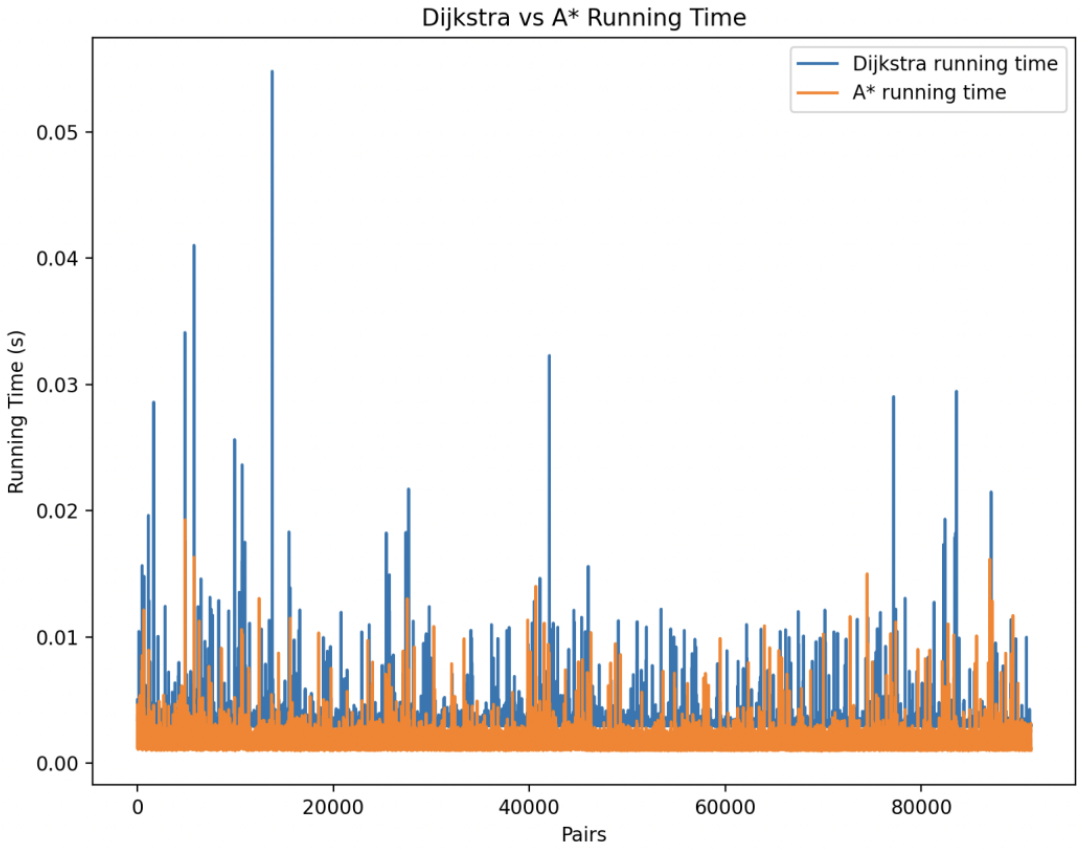


**Figure 11**

We obtain quite a large list of 91204 pairs of stations. This large value makes it nearly impossible to identify the pairs directly from the graph. For this purpose, the code outputs pair-wise runtimes for each algorithm and also specifies the station information for these pairs. To identify the spikes yourself, run the code, as it prints out the pair of stations that each pair number is associated with.

We have learned from part 2 that heuristic approaches tend to be useful in cases when the system is extremely dense. The results of this graph help us verify that statement. The general trend observed from this graph is that A* almost always outperforms Dijkstra. However, there are 638 cases where Dijkstra runs faster than A*. As there are around 90000 total cases, it is fair to say that A* is far superior when it comes to runtime. We see that for both algorithms, there are certain spikes in runtimes for certain pairs. These spikes are larger when it is required to change several lines in the process of getting from station A to station B, and smaller when there are fewer line changes required.

---

# Part 4

Many design principles are being adhered to in Figure 2. By separating the Algorithms and graphs for the shortest path finder into interfaces, it leaves the ShortPathFinder, algorithms, and previously established graphs closed for modification but open for extension, validating the open-closed-principle. Now, many different shortest-path algorithms can be added and modified, but the actual main program of ShortPathFinder will remain unchanged.

The single responsibility principle is also upheld. SPAlgorithms all have one responsibility: finding the shortest path. Graphs also only have one purpose: storing the node and edge data. Finally, ShortPathFinder only has one purpose: combining the graph and SPAlgorithm to find the shortest path for a graph.

Designing for interfaces is also clearly in effect. Algorithms are based off the SPAlgorithm interface, and different graph types are based off of the Graph interface.

As for design patterns, there is the Strategy pattern being used for the SPAlgorithms. It is used to swap out the different shortest path algorithms at runtime, through ShortPathFinder. The use of this pattern enforces the previously discussed design principles of designing for interfaces and the open-closed principle.

One key issue in figure 2 is the representation of nodes. Right now, every node is represented by an int. This may cause issues, as what if someone wants to represent nodes by names, for example. To apply information hiding, as Nodes are exposed as ints, Nodes should be their own class that Graph uses. This way, no matter how the node is represented within Node (be it an int, string, or other), the algorithms are expecting a Node type, and will be able

to work with it anyway. This would also allow Nodes to carry more information than just their name/id.

As for Graph, there are many other possible graphs that could be implemented. It may be smart to separate WeightedGraph into WeightedBidirectional and WeightedDirected, as both types have their own uses. You could also implement NonweightedDirected and NonweightedBiDirectional graphs, though it wouldn't make much sense to do so as we're mainly concerned with shortest path algorithms, which usually rely on weights to give interesting results.

You could also possibly introduce a set of those graphs represented as different things under the hood, such as a set represented by adjacency lists, a set represented by adjacency matrices, etc. This could be useful for testing if how the graph is stored has any effects on the algorithm's runtime.

# Appendix

| Part | File |
|---|---|
| Part 1 (containing dijkstra_approx(), bellman_ford_approx(), and experiments 1-3 for each algorithm) | part1.py |
| Part 2 | a_star.py |
| Part 3 | AstarVSDijk.py , london_h.py , london_map.py , |

| | |
|---|---|
| Part 4 | graph.py , heuristicgraph.py , ShortPathFinder.py , SPAlgorthm.py , weightedgraph.py , RefactoredDijkstra.py , RefactoredBellman.py , RefactoredAstar.py |