Algorithm Design and Urban Traffic System Application

April 15, 2023

Zhanxu Ye

Table of Contents

Table of Figures

Executive Summary

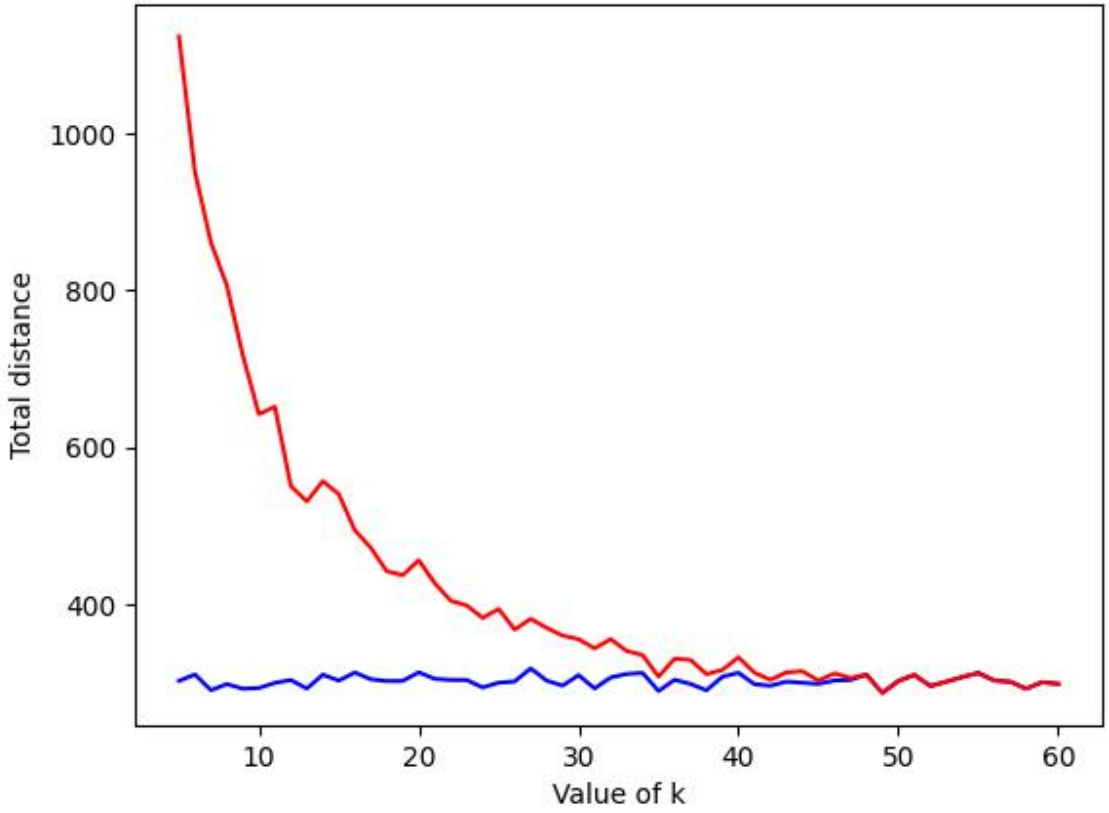
- The less we allow pathfinding algorithms to “relax” an edge, the worse their overall distance performs on graphs with large amounts of edges and nodes.

- Mystery is actually a Floyd-Warshall algorithm whose performance is in between that of Bellman-Ford and Dijkstra.

- A-Star is a modified Dijkstra’s algorithm that adds a given heuristic to its weight calculation in order to bias it towards paths closer to its destination. Given a good heuristic, it can cut down on runtime significantly.

- Good design principles and patterns, such as encapsulation and Strategy, can be implemented in Python as well.

Experiment 1a

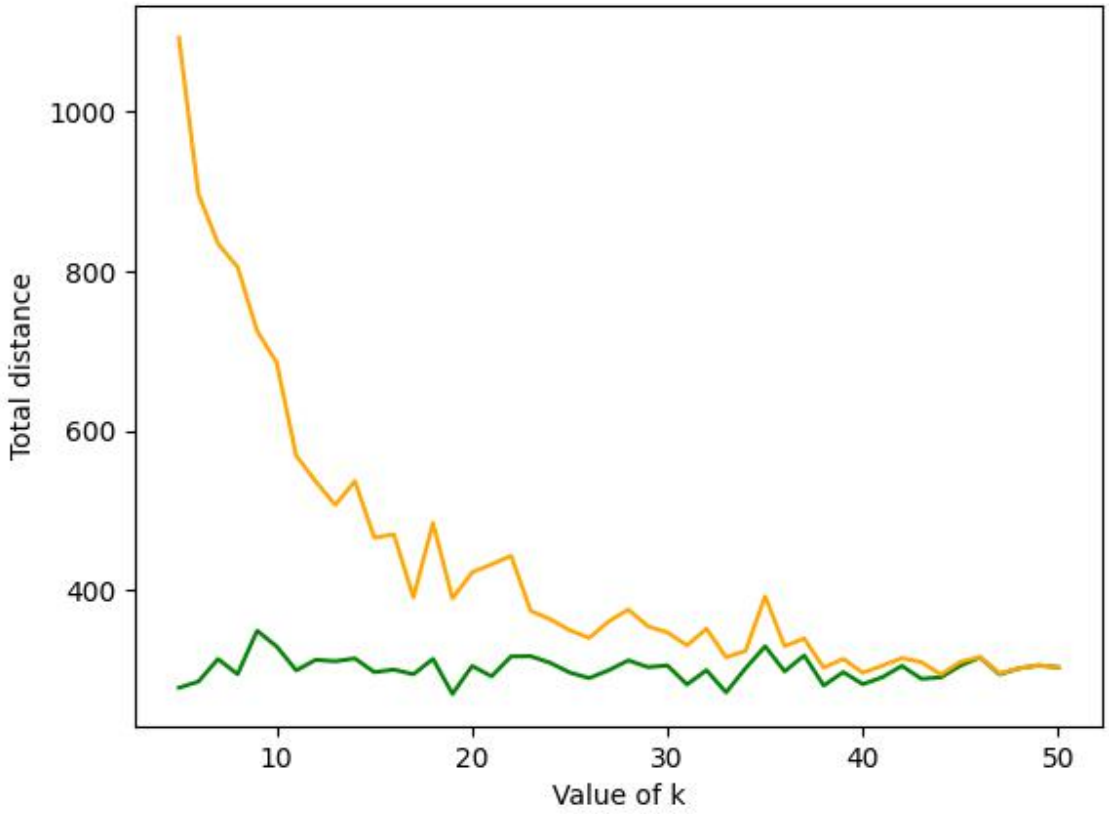


Graph nodes: 50
Graph vertices: 2450
Weight range: 1 to 50
Values of k: 5 to 60
Algorithm runs: 50
Source node: 0 in all cases

Blue: Dijkstra's Algorithm total distance (averaged)
Red: Dijkstra's Algorithm Approximation total distance (averaged)

I excluded early k values since the total distance was exponentially high.
As k value reaches the number of nodes, total distance of the approximation approaches the real total distance of the algorithm.

Experiment 1b

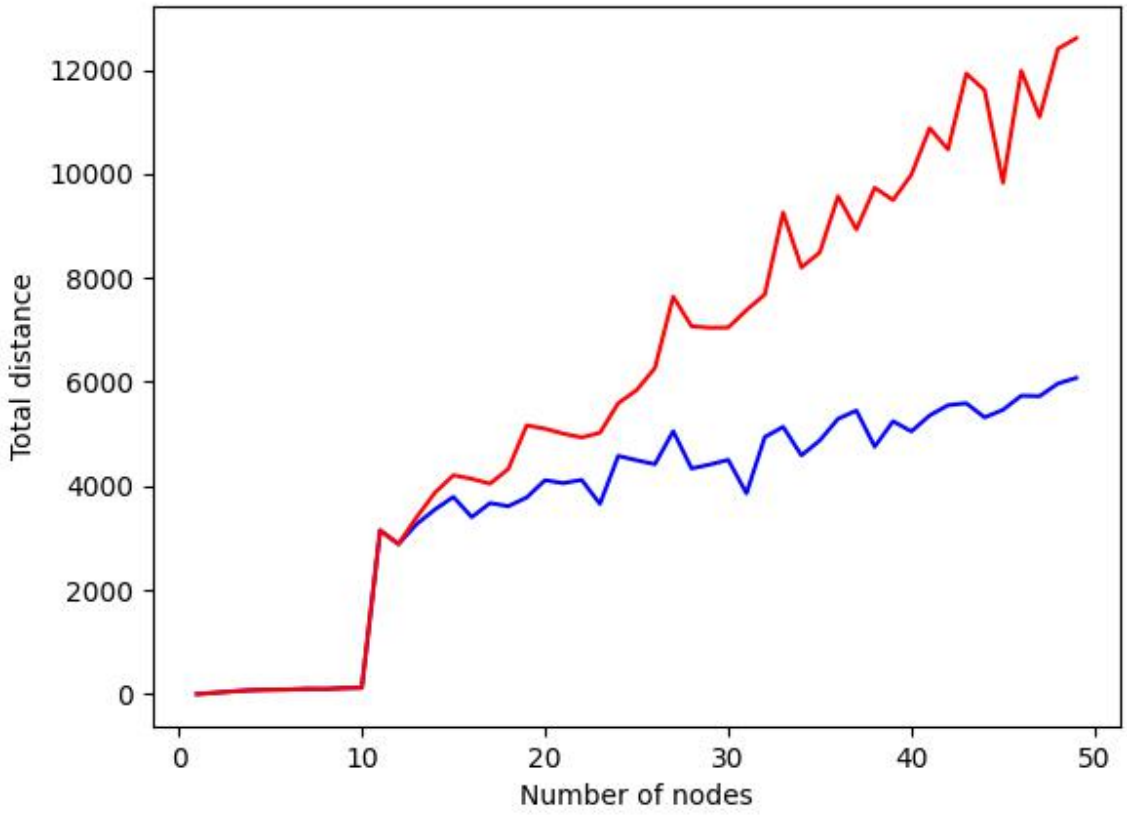


Graph nodes: 50
Graph vertices: 2450
Weight range: 1 to 50
Values of k: 5 to 50
Algorithm runs: 20
Source node: 0 in all cases

Green: Bellman-Ford Algorithm total distance (averaged)
Orange: Bellman-Ford Algorithm Approximation total distance (averaged)

Again, I excluded early k values since the total distance was exponentially high.
As k value reaches the number of nodes, total distance of the approximation approaches the real total distance of the algorithm.

Experiment 1c



Graph nodes: 1 to 50
Graph vertices: 2450
Weight range: 1 to 50
Values of k: 10
Algorithm runs: 20
Source node: 0 in all cases

Blue: Dijkstra's Algorithm total distance (averaged)
Red: Dijkstra's Algorithm Approximation total distance (averaged)

In very small complete graphs (N < 10), the approximation occurs to have the same total distance as Dijkstra, but after said amount of nodes increases past that threshold, the total distance difference between the two only seems to widen as N increases.

Mystery Function

Imagine if you were tasked with devising two different all-pairs-shortest path algorithm; one for positive edge weights and one for potentially negative edge weights. If correctness was your only concern, what would you do? It should not take too much convincing that running a single source algorithm for each potential “source” solves the all-pairs-shortest path problem.

I’d use Johnson’s Algorithm, an algorithm which adds vertexes one by one and reweights them to get rid of all negative weights. Then it uses Dijkstra’s algorithm on each pair to find the shortest path.

From 2C03 (and Wikipedia) you know Dijkstra has complexity $\Theta(E + V\log V)$, or $\Theta(V^2)$ if the graph is dense. Moreover, Bellman-Ford has complexity $\Theta(VE)$, or $\Theta(V^3)$ if the graph is dense. Knowing this, what would you conclude the complexity of your two algorithms to be for dense graphs? Explain your conclusion in your report. You do not need to verify this empirically.

The complexity for Johnson’s algorithm should be $O(V^2 E)$ (from Bellman-Ford) + $O(E)$ to re-weight a graph, and $O(V^3 \log V)$ to run Dijkstra’s V times. This adds up to $O(V^2 E + E + V^3 \log V)$, which simplifies to $O(V^2 E + V^3 \log V)$ for dense graphs.

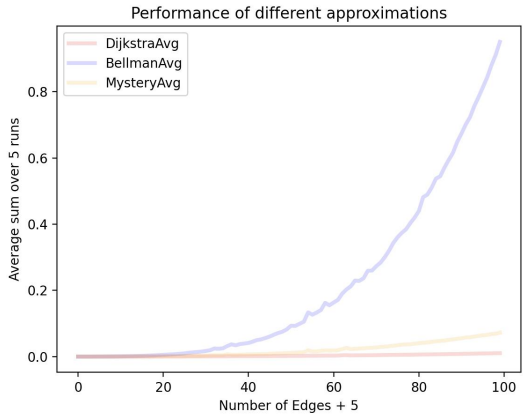
The Mystery Function:

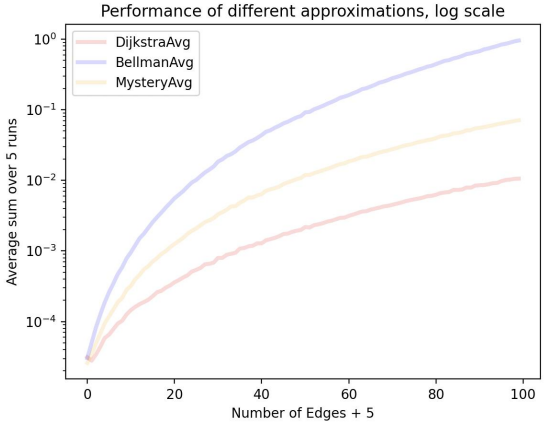
This mystery function seems to be an implementation of the Floyd-Warshall algorithm. It first initializes a solution matrix that starts the same as the input matrix. The idea is to one by one pick each vertex and find the shortest path. For each vertex, we go through each pair and find the shortest pairs. Either the pair is the shortest and gets added or it doesn’t and it gets discarded. The mystery function is capable of handling negative values, but there cannot be any negative value cycle.

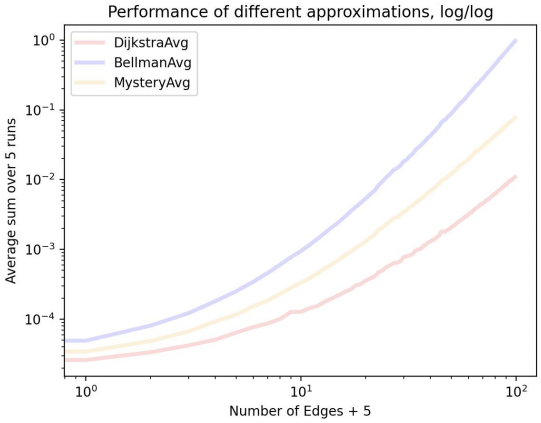
Below are the experiments that were run to determine runtime complexity.

For the first two graphs below, we created 100 sets of 5 complete graphs. Each graph was then tested on each shortest path algorithm, and the amount of time it took to complete by each method was averaged on 5 attempts. The experiment started with graphs that have 5 nodes, but each try, the number of nodes was increased to 105 nodes by the end here are the results. On the graph on the left, the y axis is on a linear scale while on the right its on a logarithmic scale.

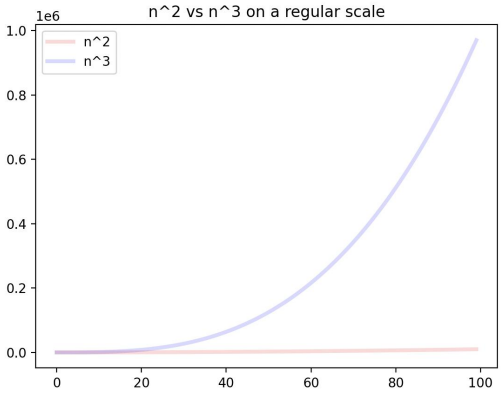
Graph nodes: 5 to 105
Graph vertices: 5 to 105
Weight: 1
Algorithm runs: 5
Source node: 1 in all cases

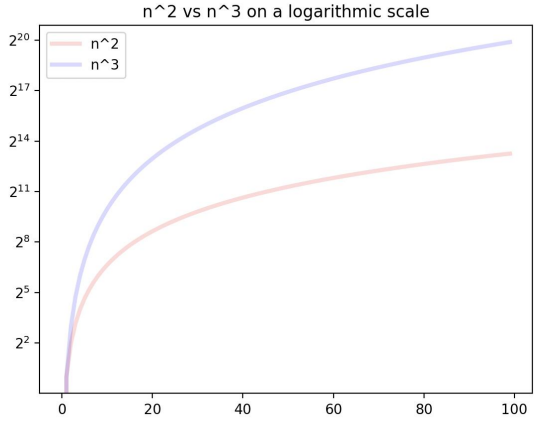


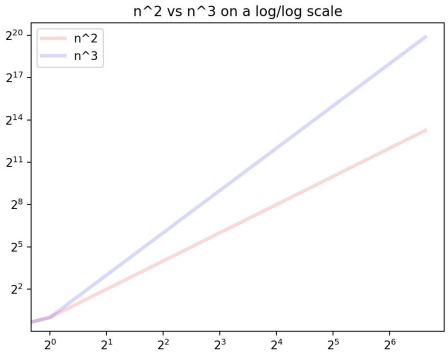




Below this are graphs of n^2 and n^3 on both a linear and logarithmic scale. As can be seen, the blue line is n^3 and while it is visibly growing at an exponential, on the logarithmic scale, it just looks like the same shape but scaled up. As we can see, the line that represents the Bellman Ford runtimes appears like its growing exponentially at a higher rate than mystery while Dijkstra's line looked almost straight. When displaying the data on a logarithmic y axis, it can be seen how all 3 algorithm's produce runtime data that's a similar shape, but scaled differently. Since BellmanFord has a n^3 runtime, its graph looks most similar to the n^3 lines in the graphs below. As you can see, the rate at which it increases is higher compared to the n^2 lines.


n^2 vs n^3 on a regular scale
1e6
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
n^2
n^3


n^2 vs n^3 on a logarithmic scale
2^{20}
2^{17}
2^{14}
2^{11}
2^{8}
2^{5}
2^{2}
0
20
40
60
80
100
n^2
n^3


n^2 vs n^3 on a log/log scale
2^{20}
2^{17}
2^{14}
2^{11}
2^{8}
2^{5}
2^{2}
2^{0}
2^{1}
2^{2}
2^{3}
2^{4}
2^{5}
2^{6}
n^2
n^3

A-Star Report

➔ Basic explanation about the A-star algorithm.

The A-star algorithm is an extension of Dijkstra's algorithm that has a heuristic function. So we will add the cost of heuristic into our algorithm, and once we find the target, we will stop searching immediately. And then return the shortest path.

➔ What issues with Dijkstra's algorithm is A* trying to address?

The main problem with Dijkstra's algorithm is that it doesn't have heuristic information. And A* perfectly solve these issues, and it can help us to avoid some unnecessary nodes and paths.

➔ How would you empirically test Dijkstra's vs A*?

I will make an experiment to test Dijkstra's vs A*.
Same graph, same target, I will figure out the runtime by using Dijkstra's and A* respectively.

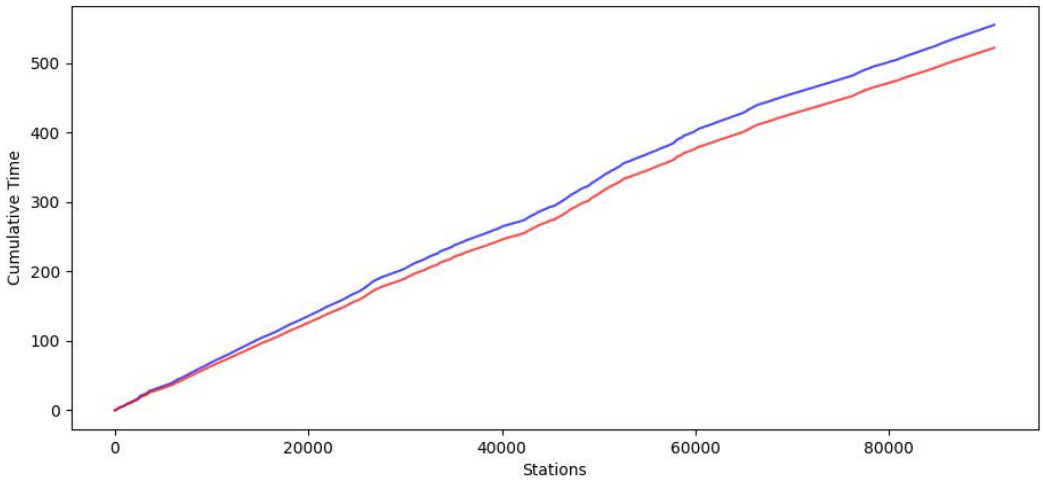
➔ If you generated an arbitrary heuristic function (similar to randomly generating weights), how would Dijkstra's algorithm compare to A*?

Because we just generated an arbitrary heuristic function, we can't make sure of the precise heuristic function, and that would reduce the efficiency of algorithm A*. Compare to A* algorithm, Dijkstra's algorithm will not use a heuristic function. Therefore, in this case, Dijkstra's algorithm could potentially be more efficient than A*.

➔ What applications would you use A* instead of Dijkstra's?

I will use A* instead of Dijkstra in some video game. I can use A* to find the shortest path between two point, for example, help novice players find the shortest path to their destination.

Experiment 2



Runs: 1, all station pairs
Red: A* Algorithm
Blue: Dijkstra's Algorithm
Y-axis: time in seconds

➔ When does A* outperform Dijkstra; when are they comparable in runtime; when does Dijkstra out perform A*?

In the case of an accurate, positive heuristic function with optimistic assumptions, A* can outperform Dijkstra in the majority of cases. Since A* is simply a modification of Dijkstra's algorithm, if a heuristic was uniform or zero, then their runtimes would be identical. Given arbitrary or random heuristics as input, A* may perform worse, since the point of a heuristic function is to bias towards the destination and prioritize certain nodes that could potentially lead you there with less summed weight.

➔ What could explain the results above?

A* was given an accurate heuristic using the details of latitude and longitude for every station to estimate the distance as the crow flies. Therefore, the time taken to find a given destination can be cut slightly shorter than the time taken for Dijkstra in some scenarios and in that case, A* edges out to be slightly faster overall.

➔ Is there anything you note about certain combinations of stations? For example, what do you note about stations which are:
On the same line?

Stations on the same line may take less time, since the route is extremely straightforward and is often designed to be the fastest route between two stations.

➔ On adjacent lines?

Similar to stations on the same line, the reasoning still applies, although this can expand the number of stations covered significantly since lines may lead to different routes. Considering they often run parallel to one another, these can be considered when pathfinding as a singular, long line.

➔ On lines which require several transfers?

Multiple transfers have the potential to slow down the overall route taken, since multiple lines are required in the route which may not directly lead to the destination nicely.

UML Design

➔ Discuss what design principles and patterns are being used in Figure 2.

Design principles:

1. Inheritance:

For the implementation of the graph class, we used a graph parent class. Then, since a weighted graph uses a lot of the same methods and has many of the same variables, it inherits from Graph. Heuristic graph is also a weighted graph with extra functions and a dictionary, so it inherits from both.

2. Encapsulation:

We used encapsulation by keeping the init functions of each object separate and inaccessible by the rest of the code. Polymorphism:

3. Open/Closed:

Heuristic Graph uses the Open/Closed principle in the way it inherits from weighted graph. Weighted graph is open to extension as seen from Heuristic Graph but closed to modification.

Design Patterns:

1. Adapter

For the implementation of A_Star, an adapter was needed. This is because it takes in a different set of arguments. A_Star requires a heuristics graph to run, so the adapter was used so that you could still call the same function from SPAlgorithm.

2. Strategy

For the implementation of the different shortest path algorithms, we used the strategy pattern. The SPAlgorithm class contains a base shortest path algorithm that just returns 0. It is meant to be overloaded by another algorithm that implements this function. In this case, Bellman Ford and Dijkstra inherit the calc_sp method, and replaces it with its own strategy. They all also specify the type of graph they take in as an argument.

➔ Figure 2 is a step in the right direction, but there is still a lot of work that can be done. For example, how graph Nodes are represented is being exposed to the rest of the application. That is, the rest of the application assume nodes are represented by integers. What if this needs to be changed in the future? For example, what if later on we need to switch nodes to be represented at Strings? Or what if nodes should carry more information than their names?
Explain how you would change the design in Figure 2 to be robust to these potential changes.

We could use another interface for nodes. This way, we could use adapter or strategy patterns to extend this implementation in the future. This adapter would need a get_node method that could deal with all the different types, and some way to compare nodes despite the types they may be.

➔ Discuss what other types of graphs we could have implemented using “Graph”. What other implementations exist?

We could have implemented weighted directed graphs. They would inherit from weighted graphs but would specify in both their adj and weights dictionaries which direction each arc would be. So Node1 to Node2 would have its own connection and weight while Node2 to Node1 would have a different weight or might not be connected at all.

Appendix

- Part 1-3 (Python-Driven Subway Path Optimization.py)

Experiment 1 Suite: Lines 186-270
Mystery Experiment: Lines 272-325
A* Algorithm: Lines 354-384
Experiment 2: Lines 386-444

- Part 4

Short Path Finder: UML_ShortPathFinder.py

Graph: graphInterface.py
Weighted Graph: graphInterface.py, Lines 34-56
Heuristic Graph: graphInterface.py, Lines 59-73

SPAlgorithm: SPAInterface.py,
Dijkstra: SPAInterface.py, Lines 18-43
Bellman Ford: SPAInterface.py, Lines 45-67
A Star: SPAInterface.py line, Lines 69-132