2XC3: Final_Lab

Zicheng Li, Yan Lu, Cosmo Xi

Table Of Contents:

1. Experiment Suite 1 ……..…. [3-6]
2. Mystery Algorithm …….. [6-10]
3. Part2……..……... [10-13]
4. Part3 ……..……. [13-15]
5. Part4 ……..…….. [15-16]
6. Executive Summary ……... [17]
7. Appendix …….. [17]

Table Of Figures:

1. Experiment Suite ...……. [3-6]
2. Mystery Algorithm …….. [7-10]
3. Part 2 ……..……..……. [12]
4. Part 3 ……..…. [13-15]

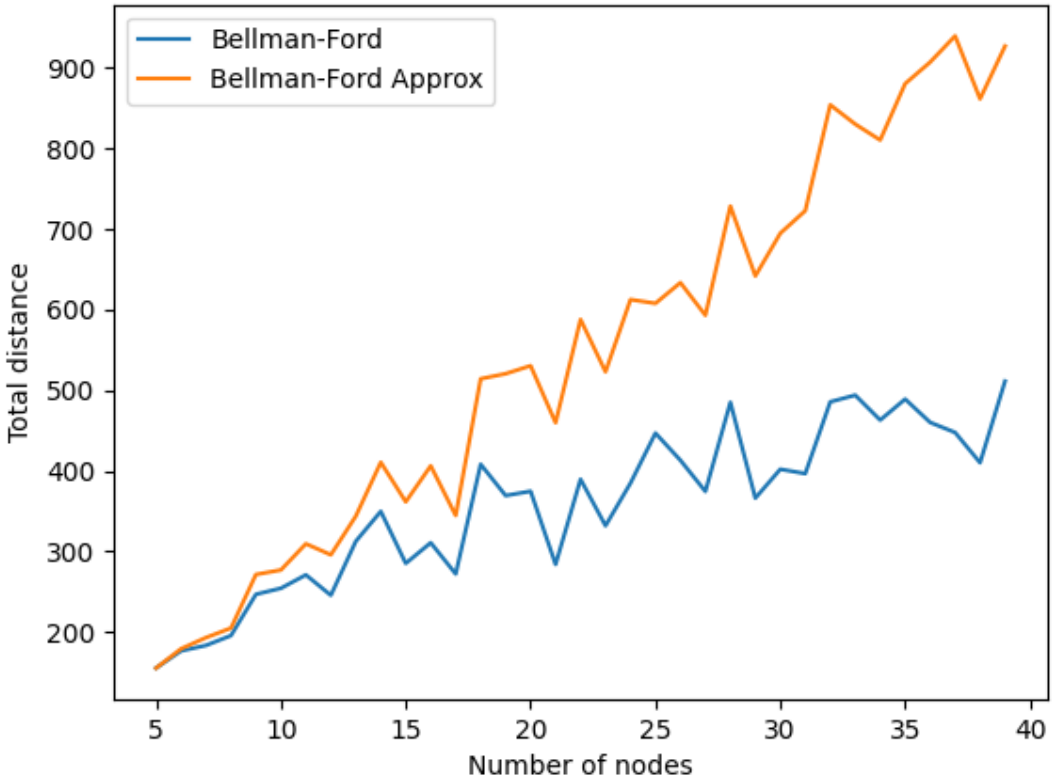
Part 1:

Experiment suite 1

a) In this experiment, I compare the total distance of Dijkstra and dijkstra_approx when increasing the number of nodes and fix the value of k for the graphs.
The number of nodes = [10, 11, 12,..., 99, 100]
K = 3
Number of runs = 10

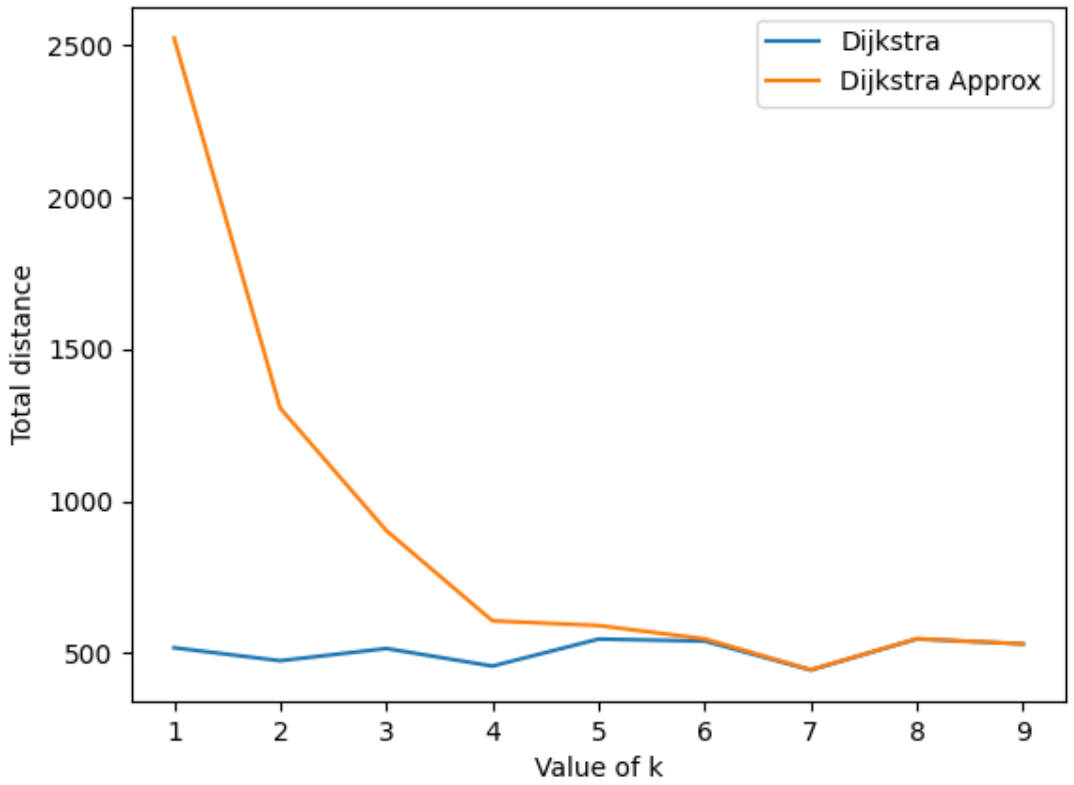
We can see that when the number of nodes is small, the total distance of both algorithms is almost the same. As the number of nodes increases, the total distance of both increases but the difference between the two algorithms becomes larger where the total distance of dijkstra_approx is significantly larger than Dijkstra.

b) In this experiment, I compare the total distance of bellman_ford and bellman_ford_approx when increasing the number of nodes and fix the value of k for the graphs.
The number of nodes = [5, 6, 7,..., 39, 40]
K = 3
Number of runs = 10



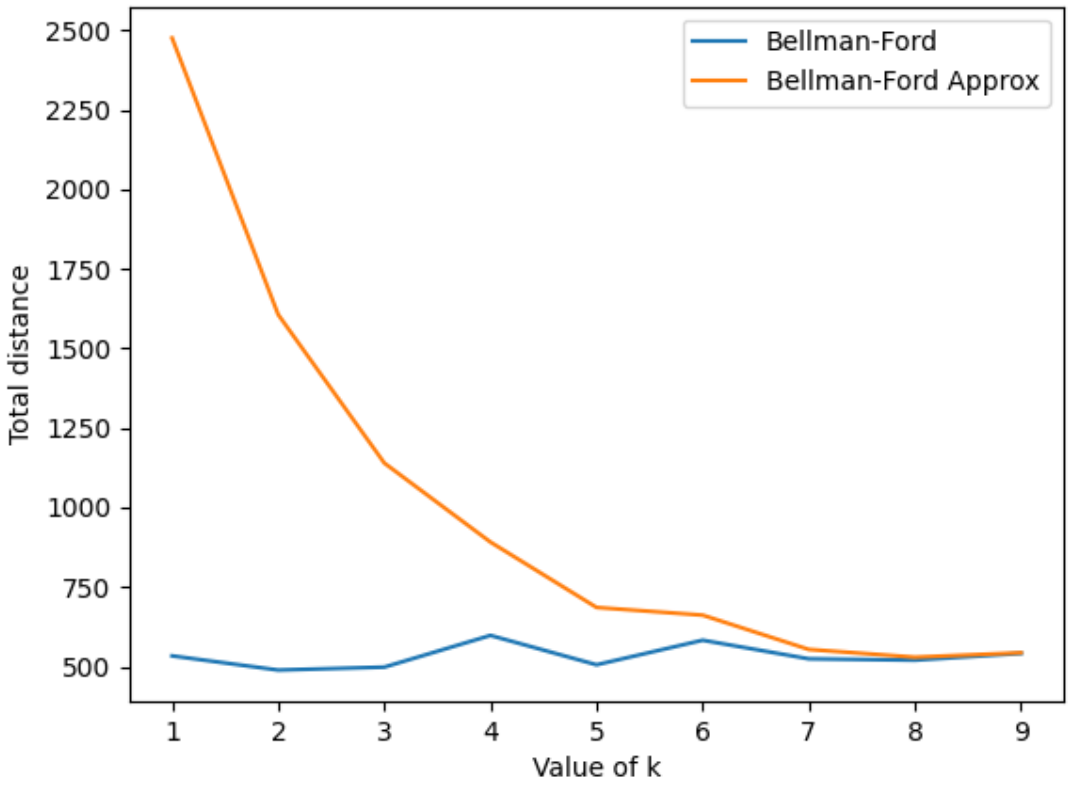
We can see that when the number of nodes is small, the total distance of both algorithms is almost the same. As the number of nodes increases, the total distance of both increases but the difference between the two algorithms becomes larger where the total distance of bellman_ford_approx is significantly larger than bellman_ford.

c) In this experiment, I compare the total distance of Dijkstra and dijkstra_approx when increasing the value of k and fixing the number of nodes of the graphs.
The number of nodes = 50
K = [1,2,3,...,10]
Number of runs = 10



We can see that when the value of k is small, the total distance difference between the two is very large where the total distance of dijkstra_approx is significantly larger than that of Dijkstra. As the value of k increases, the total distance of Dijkstra approaches dijkstra_approx.

d) In this experiment, I compare the total distance of bellman_ford and bellman_ford_approx when increasing the value of k and fixing the number of nodes of the graphs.
The number of nodes = 50
K = [1,2,3,...,10]
Number of runs = 10



The result of this experiment is similar to experiment c: the difference in total distance is very large at first where bellmam_ford has the greater distance, then the difference becomes smaller as the value of k increases.

Mystery Algorithm

For positive edge weights, we could use Dijkstra's algorithm as a single-source algorithm for each vertex, which has a complexity of Θ(E + VlogV) or Θ(V^2) for dense graphs. Therefore, the complexity of the algorithm would be Θ(V(E + VlogV)) or Θ(V^3) for dense graphs.

For potentially negative edge weights, we could use the Bellman-Ford algorithm as a single-source algorithm for each vertex, which has a complexity of Θ(VE) or Θ(V^3) for dense graphs. Therefore, the complexity of the algorithm would be Θ(V(VE)) or Θ(V^4) for dense graphs.

The mystery() function is implementing the Floyd-Warshall algorithm that finds the all-pairs shortest path in a graph. It initializes a distance matrix d using the init_d() function, where the element d[i][j] is the weight of the edge from node i to j,

float("inf") means there is no edge connecting the two nodes. d[i][i] is set to zero because the shortest path from a node to itself is always zero.
Then, this algorithm iterates over all nodes k and updates the distance matrix d by considering all possible paths from node i to node j through node k. If d[i][j] > d[i][k] + d[k][j], then the distance matrix is updated to be d[i][k] + d[k][j].

The graph with negative edge weights I use is:

```
G = DirectedWeightedGraph()
G.add_node(0)
G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_edge(0, 1, 2)
G.add_edge(1, 3, -3)
G.add_edge(3, 2, -1)
G.add_edge(1, 2, 1)
```

I get this result:

```
[[0, 2, -2, -1], [inf, 0, -4, -3], [inf, inf, 0, inf], [inf, inf, -1, 0]]
```

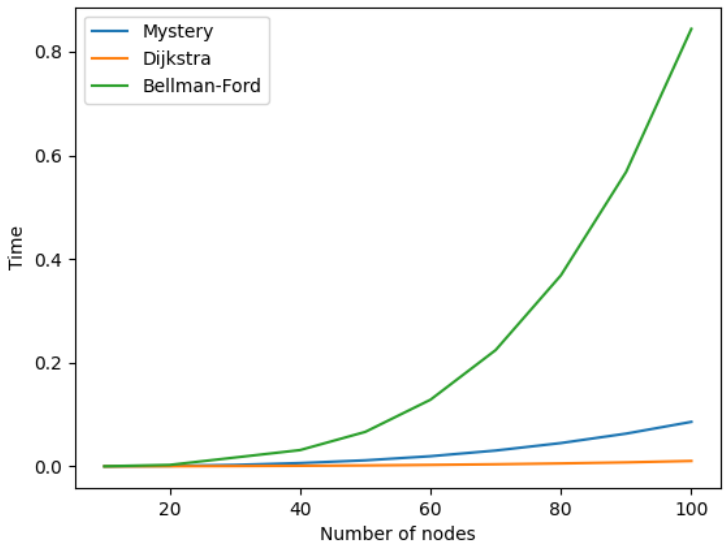
Which is correct.

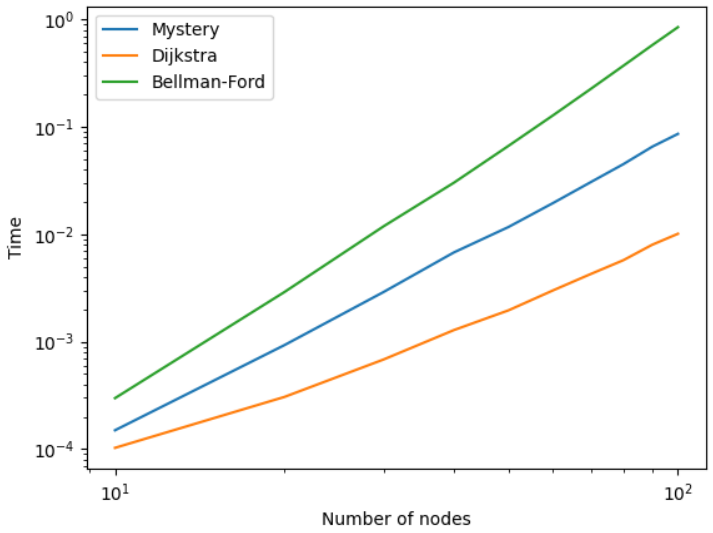
To find the complexity of the mystery algorithm, I designed the following 2 experiments:

```
sizes = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
times_mystery = []
times_dijkstra = []
times_bellman_ford = []

for n in sizes:
    G = create_random_complete_graph(n, 100)
    start_time = timeit.default_timer()
    mystery(G)
    end_time = timeit.default_timer()
    times_mystery.append(end_time - start_time)
    start_time = timeit.default_timer()
    dijkstra(G, 0)
    end_time = timeit.default_timer()
    times_dijkstra.append(end_time - start_time)
    start_time = timeit.default_timer()
    bellman_ford(G, 0)
    end_time = timeit.default_timer()
    times_bellman_ford.append(end_time - start_time)

plt.plot(sizes, times_mystery, label="Mystery")
plt.plot(sizes, times_dijkstra, label="Dijkstra")
plt.plot(sizes, times_bellman_ford, label="Bellman-Ford")
plt.xlabel("Number of nodes")
plt.ylabel("Time")
plt.legend()
plt.show()
```

```python
for n in sizes:
    G = create_random_complete_graph(n, 100)
    start_time = timeit.default_timer()
    mystery(G)
    end_time = timeit.default_timer()
    times_mystery.append(end_time - start_time)
    start_time = timeit.default_timer()
    dijkstra(G, 0)
    end_time = timeit.default_timer()
    times_dijkstra.append(end_time - start_time)
    start_time = timeit.default_timer()
    bellman_ford(G, 0)
    end_time = timeit.default_timer()
    times_bellman_ford.append(end_time - start_time)

plt.loglog(sizes, times_mystery, label="Mystery")
plt.loglog(sizes, times_dijkstra, label="Dijkstra")
plt.loglog(sizes, times_bellman_ford, label="Bellman-Ford")
plt.xlabel("Number of nodes")
plt.ylabel("Time")
plt.legend()
plt.show()
```

And get the following results:





The first graph is a regular plot and the second one is log/log plot. We can see that for both plots, the complexity of the mystery function is between Dijkstra and bellman_ford, which means the complexity of the mystery function is Θ(V^3).
This is not surprising because when we look at the code, there is a triple loop, which makes this algorithm run in cubic polynomial time.

Part 2:

Implementation

I modify Dijkstra's algorithm by editing the min_heap Q. The values of Q is the sum of edge weights and heuristic values. Everything else stays the same. See the Python file. There is the test case for part 2 in final_project_part2.

Explanation

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

Dijkstra's algorithm just follows the path currently shortest and doesn't pay any attention to the direction it is going. So Dijkstra will just go through some unnecessary path, and it does not aim to the destination and find the shortest path. Instead, it will go all the possible paths and go to every single one to find the shortest path. It will flood fill the graph and it is not very optimal. Whereas A* will go toward the destination using the heuristic function.

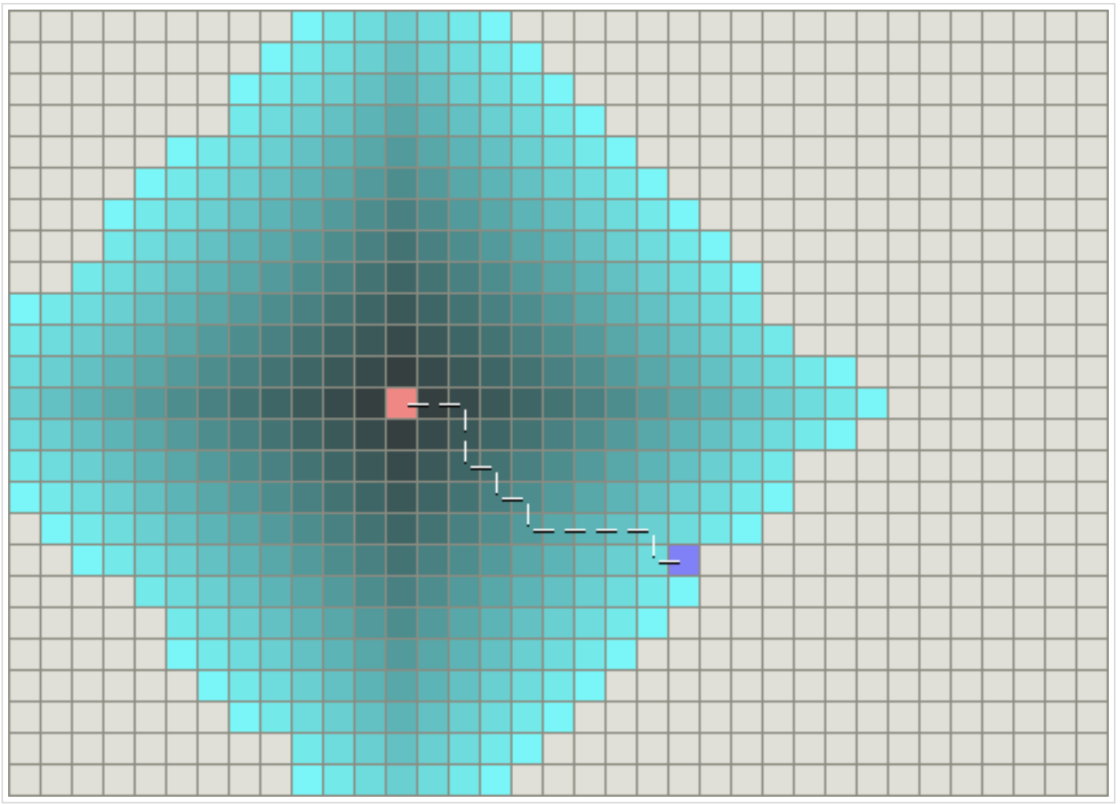
How would you empirically test Dijkstra's vs A*?

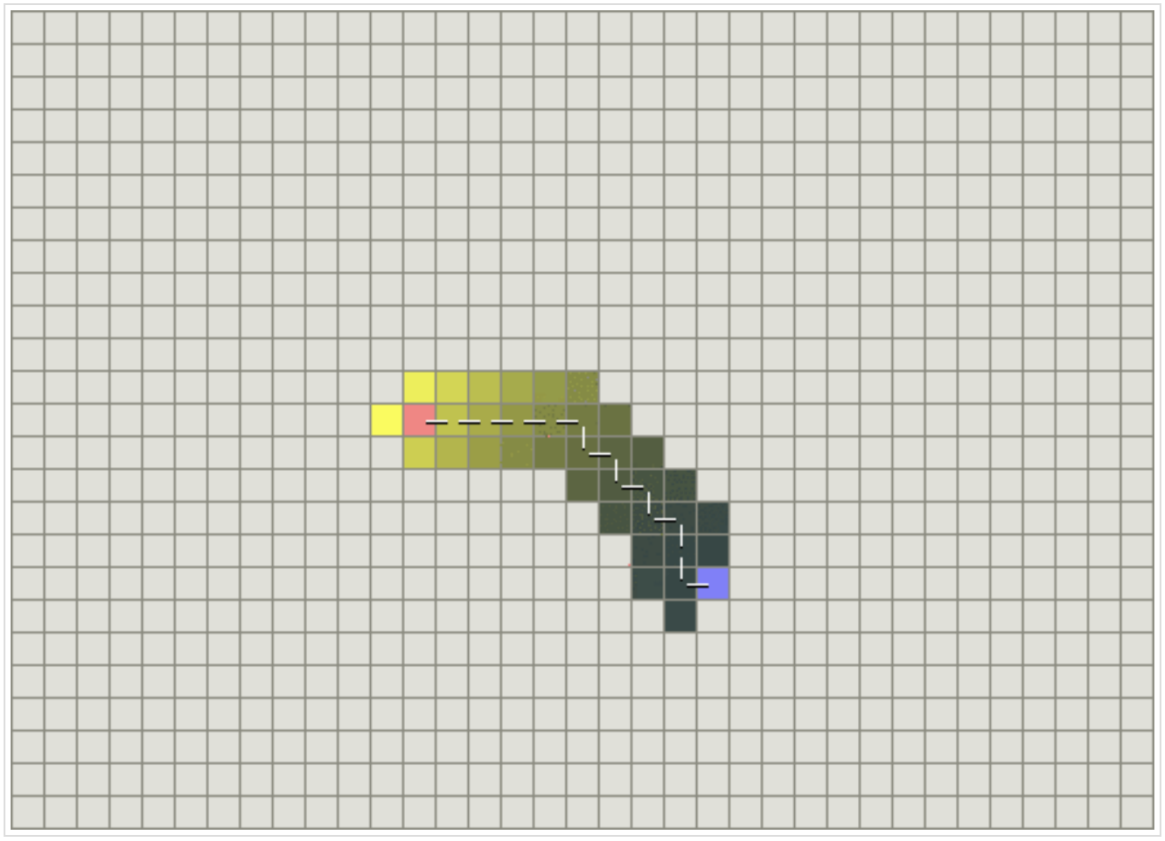
I can make a DirectedWeightedGraph with a heuristic function to test the different paths that Dijkstra gives versus A*. I can also test the runtime of Dijkstra versus A*.

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

Dijkstra will run the whole graph flood fill to explore every single node. Whereas A* will choose the smallest heuristic values and then find the shortest path while it is aiming for the destination. Also, Dijkstra will run slower than A* because Dijkstra loops all nodes.

The first picture shows Dijkstra finding the path from red to blue. The second picture shows the A* finding the path from red to blue.



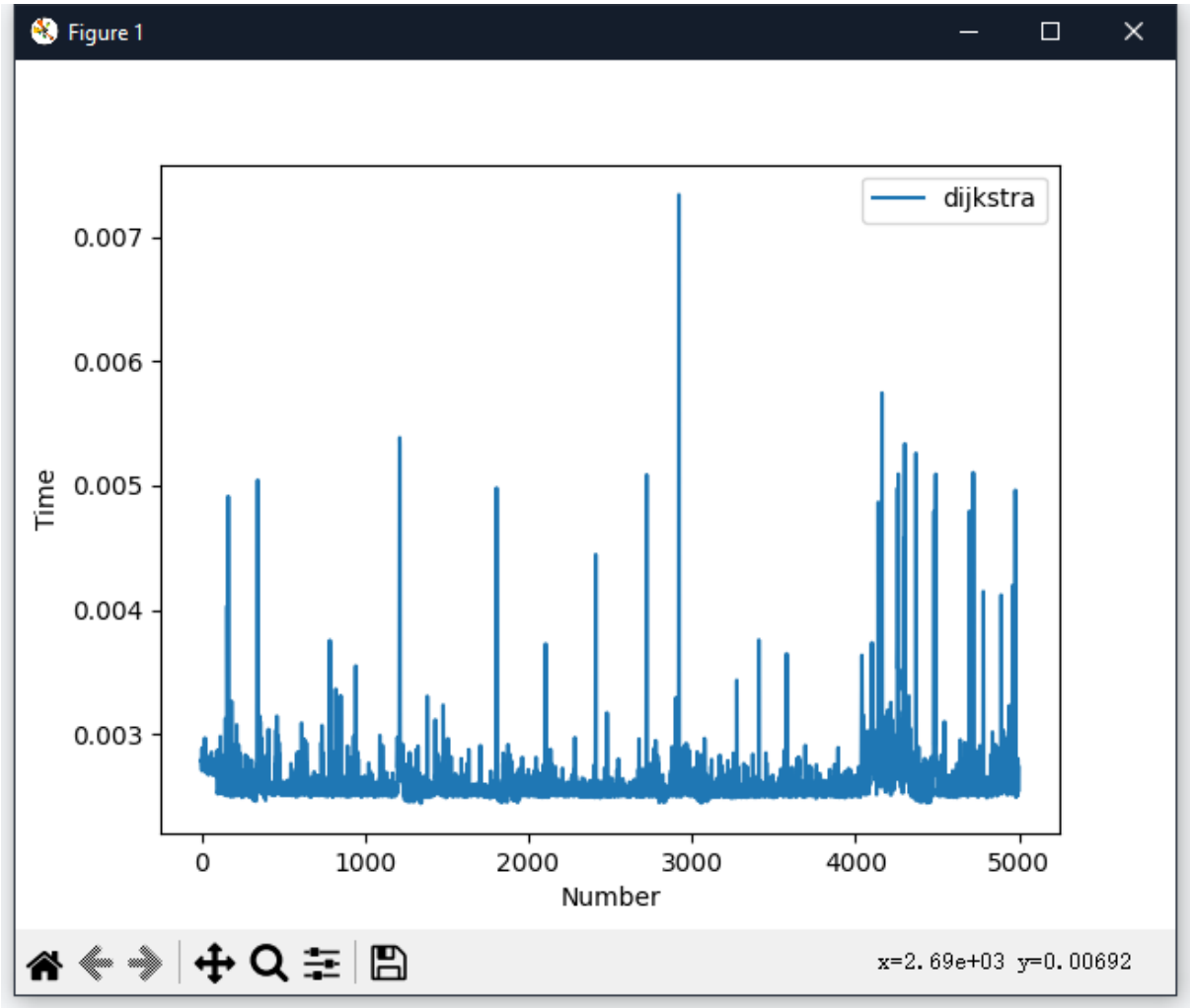


What applications would you use A* instead of Dijkstra's?

When I want to find the shortest path between 2 nodes, I will use A* because it is a lot faster. For example, when I want to find the shortest from Vancouver to Toronto, I will use A* to find the shortest path. Whereas Dijkstra will go through all roads in Canada, which is impractical.

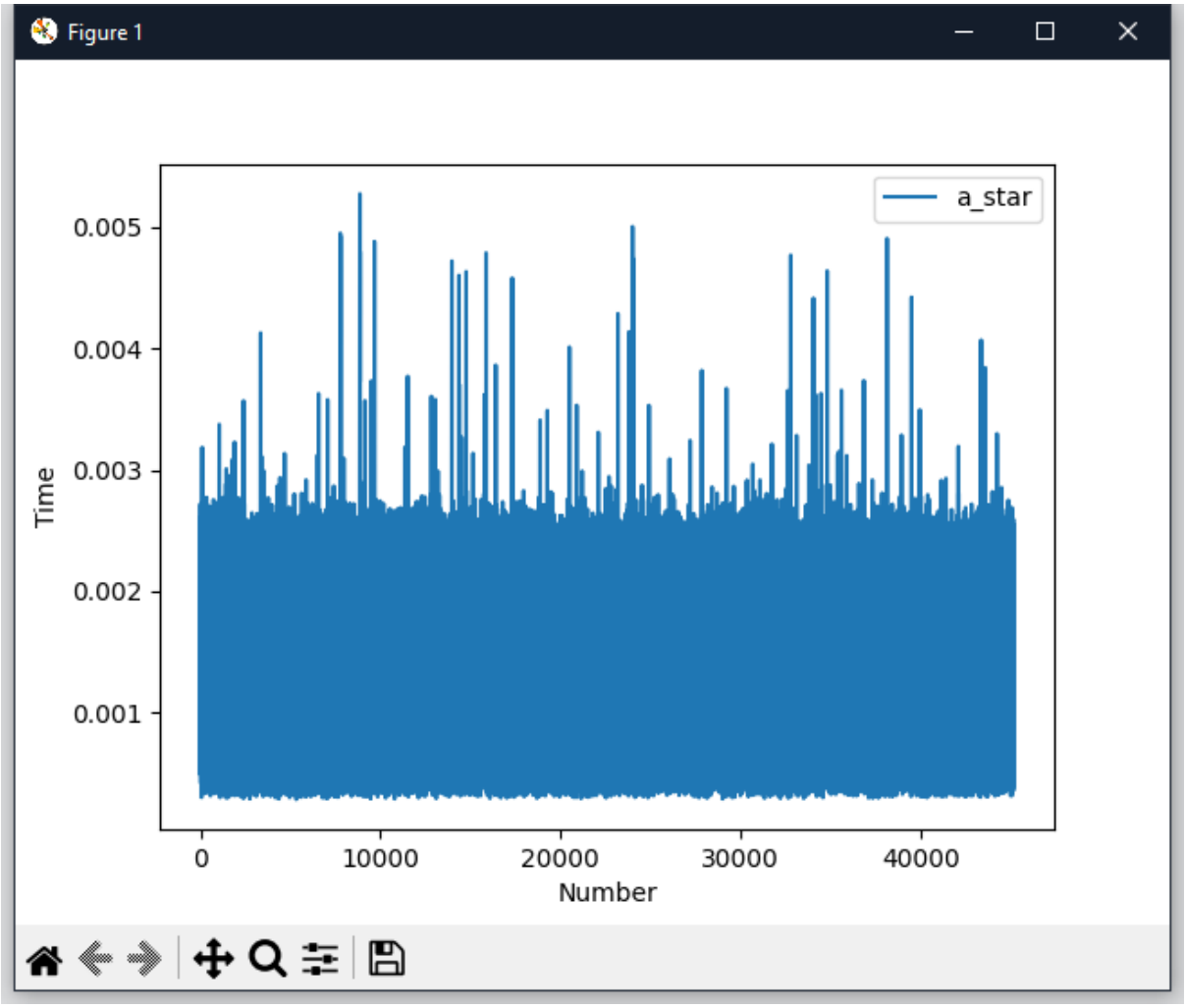
Part 3:

First, I run 5000 times London Station Directed Weighted Graph with the Dijkstra algorithm. Got the following graph.



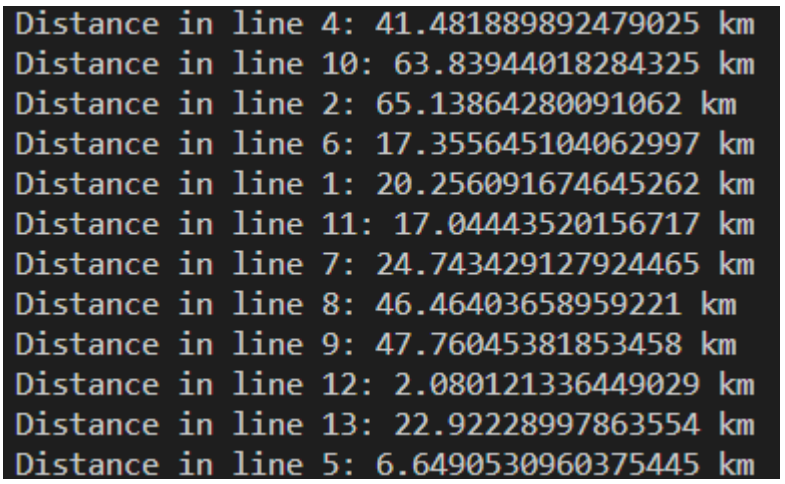
It uses 13.09633580002992s in total, the fastest is around 0.0024s. Most of the time it runs below 0.003s.

Then, I take all combinations of stations for the A* algorithm and see the result below.



It can clearly see that still most of the pairs are even below 0.0027 and some reach 0.0002.
So most of the time A* outperforms Dijkstra. This is making sense because A* is just getting the shortest path between two certain situations, once it reaches then stops. So it should somehow take less time than Dijkstra.

Then based on my observation, stations on the same line take less time. Usually around 0.0003s even 0.0002 for adjacent stations. Then for stations on adjacent lines, usually takes slightly more time. For example, situations 224 and 266 are on line 11 and line 10, but still, take 0.00036s. Finally, lines which require several transfers take the most time.



Above I used a function total_distance_by_line method with a modified DirectedWeightedGraph to provide the line information for the shortest path.

Part 4:

There are two test cases in the ShortestPathFinder.py that are available to test the result.

1.
    a. We use the **strategy pattern** to design the SPAlgorithm. So users can choose the different shortest-path algorithms from the ShortestPathFinder. We make the class for each algorithm. We create classes for Dijkstra and Bellman_Ford and they are all inherited from SPAlgorithm interfaces. And for each algorithm, we override the calc_sp method.
    b. We also use the **adapter pattern** to design the A_Star method. This is because A_Star requires a heuristics dictionary as an argument to run, so we can use the adapter so that the user can still use the same manner in ShortestPathFinder class to call the A_Star method.
    c. We use the **Open/Closed principle** as Design principles. Our SPAlgorithm class are open for extension, so user can add more different kind of Algorithm as they like. But we close for modification, the user can not change anything on the SPAlgorithm directly. They only have access to ShortestPathFinder class.

d. We use **Encapsulation** as the Design principle. We make the heuristics dictionary private in the HeuristicGraph class. And we make a setter method for the user to modify the heuristics dictionary. We also make a getter method for the user to get the heuristics dictionary.
e. We use **Inheritance** as the Design principle. For the Weightedgraph and HeuristicGraph, all inherit from graph interfaces. In the SPAlgorithm class, all shortest-path algorithms are inherited from SPAlgorithm.

2. We can create a new class for nodes. So when each time when we want to change something to a node, we can always design a more setter method to extend our implementation in the future. So we can also use Factory Pattern or strategy pattern to create more different kinds of nodes.

3. We can also create DirectedWeightedGraph class that is inherited from Weightedgraph class. Therefore we can refactor the Weightedgraph class in the part1 Python file that the professor provided. We need another data structure to store the direction, which is the direction between two nodes.

Executive Summary:

- The total distance of the shortest path is large when k (max relaxation value) is small.
- When the value of k is fixed, the total distance difference for both the shortest path algorithm and their approximation algorithm would increase when the number of nodes increases.
- When the number of nodes is fixed, the total distance difference for both the shortest path algorithm and their approximation algorithm would decrease when k increases.
- A* is going toward the destination using the heuristic function. A* is faster than Dijkstra in cases where the graph is more complex.
- A* is an extension of Dijkstra, but it is very useful to increase efficiency.
- Design patterns and principles are very powerful in terms of refactoring the code into something reasonable.
- When getting the distance of stations, just learned what is and how to use haversine_distance with trigonometry
- In order to get the line information for the shortest path of the subway system, some modifications of the data structure and both algorithms are useful and convenient.
- Furthermore, it is possible to have a special total_distance_by_line for the A* algorithm.

Appendix:

1. Part1: final_project_part1.py
2. Part2:final_project_part2.py
3. Part3:final_project_part3.py
4. Part4:graph.py, ShortestPathFinder.py, SPAlgorithm.py