CompSci 2XC3 Final Project

Lab Section: L03

Yujia Wang #400356815

Louis Doan #400395183

Sam (Jia Wei) Liu #400022227

Table of Contents

Table of Figures……3

Summary……4

Part 1……5

Part 2……8

Part 3……9

Part 4……11

Appendix……13

Table of Figures

Figure 1: Graph displaying the difference between the approximations and their original versions when the size of the graph changes……5

Figure 2: Graph displaying the difference between the approximations and their original versions when the density of the graph changes……..6

Figure 3: Graph displaying the difference between the approximations and their original versions when the maximum number of relaxations allowed for the approximation function changes……..7

Figure 4: Graph displaying run time of the mystery function when run over random complete graphs with 1, 2, …, 150 nodes. ……8

Figure 5: Graph displaying the runtimes of A* (blue) vs Dijkstra's (orange) on pairs of stations……..9

Figure 6: Graph displaying the runtimes of A* using a bad heuristic (blue) vs Dijkstra's (orange) on pairs of stations……10

Figure 7: Graph displaying the runtimes of A* (blue) vs Dijkstra's (orange) on pairs of stations……..11

Summary

- Part 1: We ran experiments to test the relationship between total distance and size of graph for Dijkstra and Bellman-Ford. We also tested the density of the graph vs the accuracy of the approximation functions. Lastly, we tested the maximum number of relaxations and accuracy of approximations.
- For the Mystery Algorithm, dijkstra-approx(G, source, k) has $\theta(kV)$ run time complexity for dense graphs, and bellman-ford-approx(G, source, k) has $\theta(kV^2)$ run time complexity for dense graphs.
- Part 2: A* is an extension of Dijkstra's algorithm that tries to address the issue of large and complex graphs by using a heuristic function to guide the search towards the goal, rather than exploring all possible paths as Dijkstra's algorithm does.
- Part 3: A* performed better than Dijkstra's. However, this is not always the case as A*'s performance is heavily dependent on its heuristic function, as shown when an inefficient heuristic function was used.
- Part 4: The UML diagram utilizes the design principles of inheritance and polymorphism.

Part 1:

Experiment Suite 1:

In our first experiment, we generated random complete graphs with 10, 11, …, 100 nodes and observed the difference in total distance from a randomly selected source node to every node in the graph between both approximations and their original versions when the size of the input graph changes. We chose k to be 20 and found a positive relationship between the difference and the size of the graph for Dijkstra, and no relationship for the difference between the Bellman-ford approximation and its original version.

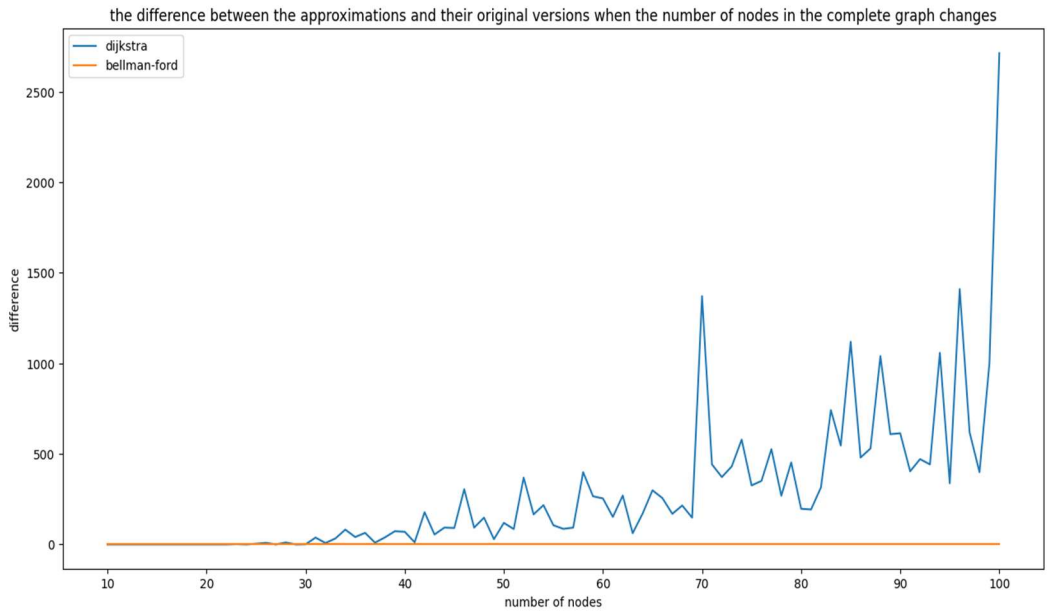

Figure 1: Graph displaying the difference between the approximations and their original versions when the size of the graph changes.

In our second experiment, we generated random graphs with 50 nodes and 50 to 100 edges that have at least one path from node 0 to every node and observed the difference between both approximations and their original versions when the density of the input graph changes. We did not use the total_dist() function as a means of measurement as sometimes the approximation cannot find a shortest path to from node 0 to another node within 20 relaxations. Instead we randomly select a node that the approximation did find a shortest path from node 0 to it. The value of k is still 20 and we found that the density of the graph does not affect the accuracy of the approximation for both algorithms.

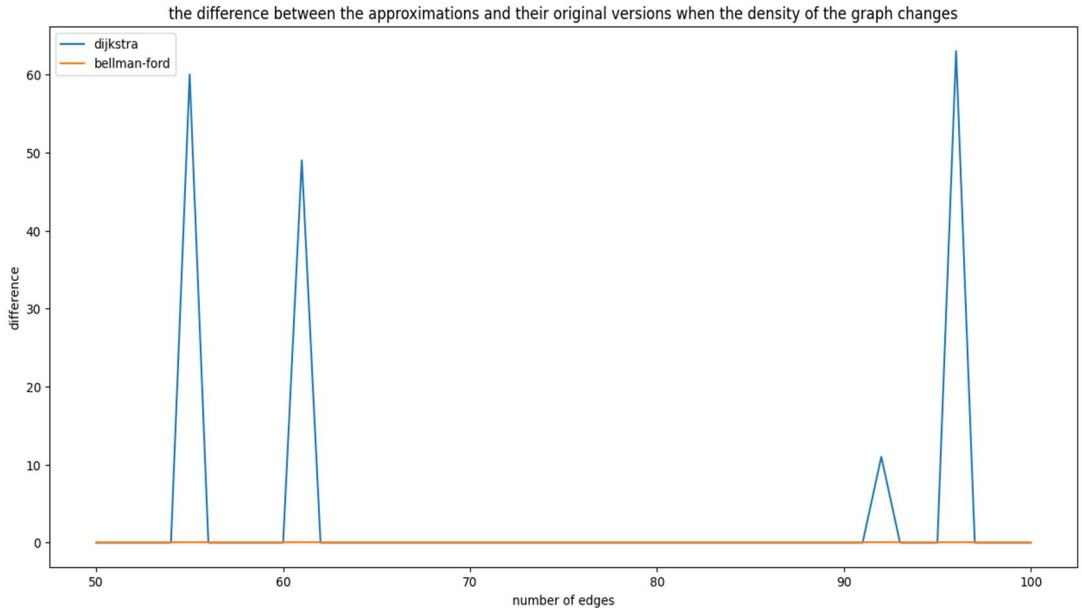

Figure 2: Graph displaying the difference between the approximations and their original versions when the density of the graph changes.

In our third experiment, we generated random complete graphs with 50 nodes and observed the difference between both approximations and their original versions when the maximum number of relaxations allowed for the approximation function changes from 0 to 100. Using the same method of measurement as the first experiment, we found that there is a definite negative relationship for both algorithms, but it is more pronounced for Dijkstra.

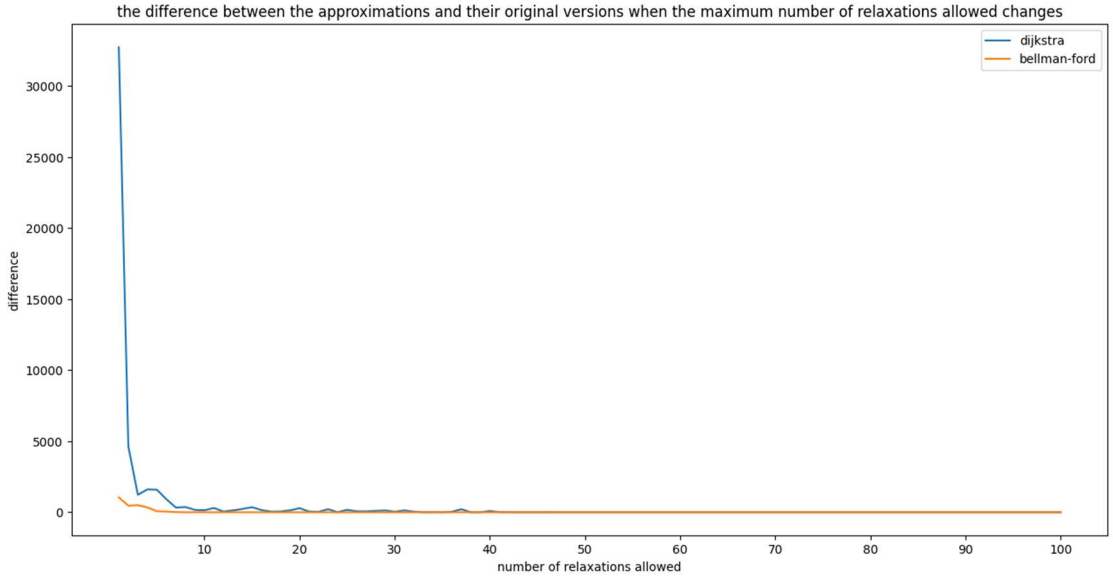

Figure 3: Graph displaying the difference between the approximations and their original versions when the maximum number of relaxations allowed for the approximation function changes

Mystery Algorithm:

The complexity of dijkstra-approx(G, source, k) is $\theta(kV)$ for dense graphs, where k is the maximum number of relaxations allowed($0 \leq k < V$), V is the number of edges in the graph. The complexity of bellman-ford-approx(G, source, k) is $\theta(kV^2)$ for dense graphs, where k is the maximum number of relaxations allowed($0 \leq k < V$), V is the number of edges in the graph. The reason for this is all else being equal, in the nested loop that makes up the core of both original algorithms, the code inside the if condition is executed a maximum of V times each time the innermost loop is executed, whereas in the approximation functions it is only executed up to k times.

To deduce what the mystery function outputs, we generated 100 random complete graphs without negative edge weights and used them as input for both the Dijkstra algorithm and the mystery function. In all cases, the nth element in the output of mystery contains a list of the shortest distance from node n to every node in the graph. Therefore, we hypothesize the mystery function calculates the shortest path to every node in the graph for node 0, 1, …, n in the graph. Next, we ran the function 10 times on graphs containing negative edge weights. Due to the min_heap functions not working with negative weights, we did not compare the output of the Dijkstra algorithm to the mystery function, but upon manual verification, the mystery function appears to produce the correct output for graphs containing negative edge weights as well.

By inspecting the code, G.number_of_nodes() has constant run time, init_d(G) has $\theta(V^2)$ run time complexity, and the triple nested loop has $\theta(V^3)$ run time complexity. We conclude the mystery function has a run time complexity of $\theta(V^3)$, where V is the number of nodes in the graph. We verified this with an experiment which graphs the run time of the mystery function when it is run on random

complete graphs with 1, 2, …, 150 nodes. This run time complexity is not surprising as the mystery function calculates the shortest path to every node in the graph for node 0, 1, …, n in the graph, and to determine whether there is a path that is shorter than directly from one node to another, it needs to consider the nodes in triples (i, j, k), hence the need for the triple nested loop mentioned earlier.

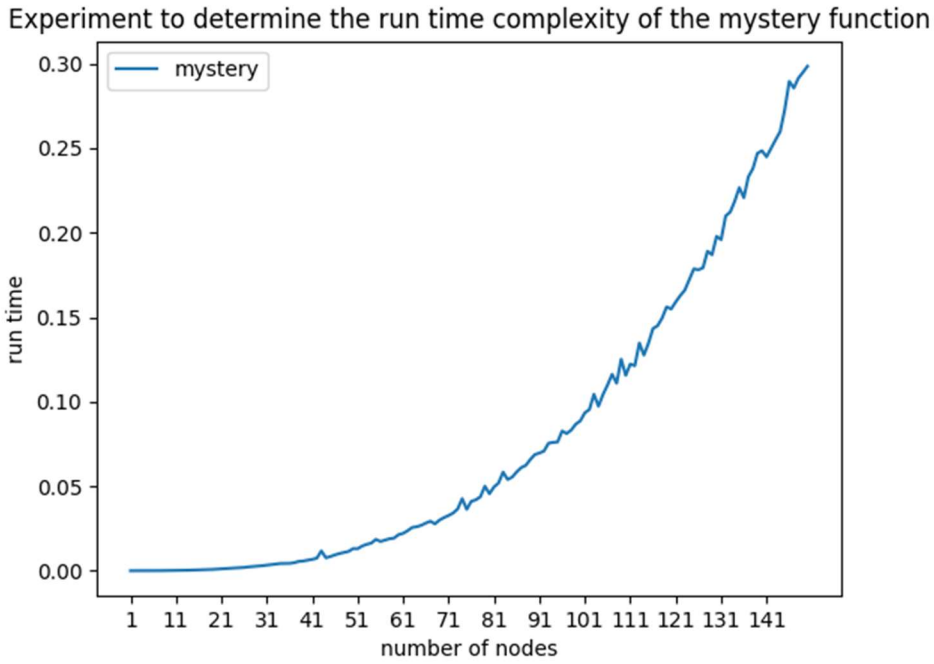

Figure 4: Graph displaying run time of the mystery function when run over random complete graphs with 1, 2, …, 150 nodes.

Part 2:

Dijkstra's algorithm can be very slow when dealing with large or complex graphs. A* is an extension of Dijkstra's algorithm that tries to address this issue by using a heuristic function to guide the search towards the goal, rather than exploring all possible paths as Dijkstra's algorithm does. The heuristic function typically provides an estimate of the distance from the current node to the goal node, allowing A* to prioritize exploring the most promising paths first.

To empirically test the performance of Dijkstra's algorithm vs A*, we could compare their running time and the number of nodes visited for a given graph and source-destination pair. We can randomly generate graphs with different properties such as number of nodes, edge density, and connectivity to compare the performance of Dijkstra and A* with different heuristic functions. We can measure the running time and the number of nodes visited for both algorithms for each graph and source-destination pair, and compare the results to determine which algorithm performs better in terms of time and space complexity.

If an arbitrary heuristic function is used that does not provide accurate estimates of the distance to the goal, A* may perform similarly to Dijkstra's. In this case, A* would explore many unnecessary paths, as the heuristic function would not be effective in guiding the search towards the goal. In the worst case, the heuristic function may cause A* to perform worse than Dijkstra's, as it can lead astray the algorithm.

A* is particularly useful in applications where the search space is large or complex and where it is important to find the optimal path quickly. As such, if there is a good heuristic function, A* would be preferred in situations such as navigation and satellite systems, where the algorithm needs to find the fastest or shortest route from one location to another in a city or country, such as in logistics and transportation. Another situation would be pathfinding in games or robotics, where the algorithm needs to find the shortest path from point to another in 2D or 3D space.

Part 3:

Experiment setup:

Graph:

- each station was added to the graph as a node.
- edges connect stations that are connected.
- weight of edges is the distance between the 2 connected stations, calculated using the latitude and longitude of the stations.

Algorithms are tested on an array of pairs (x, y) where each algorithm finds the shortest path between station x and y.

The array used in the experiment is ordered into 3 subparts a, b, c where:

- a: all possible pairs of stations on the same line.
- b: all possible pairs of stations on adjacent lines.
- c: all possible pairs of stations on lines that require several transfers.

Subparts a + b + c = all combinations of stations. The array is structured where part a comes first, then b, then c. So, the algorithm is first ran on pairs of stations on the same line, then pairs of stations on adjacent lines, then pairs of stations on lines that require several transfers.

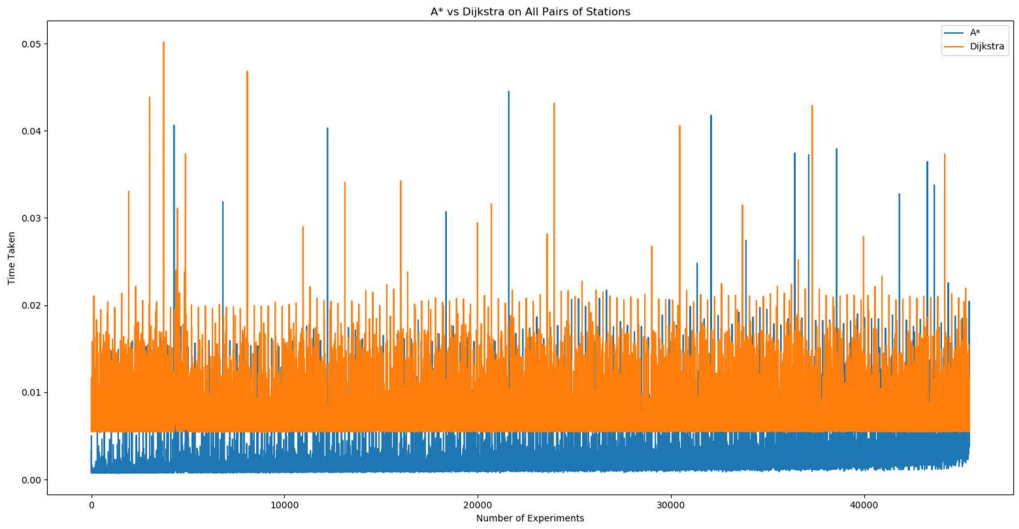

Figure 5: Graph displaying the runtimes of A (blue) vs Dijkstra's (orange) on pairs of stations.*

Overall, A* always appears to be better than Dijkstra's. At all points in the graph, whether pairs of stations were on the same line or not, A* was better. However, this does not actually mean that A* will always be better, as A* is very dependent on its heuristic function.

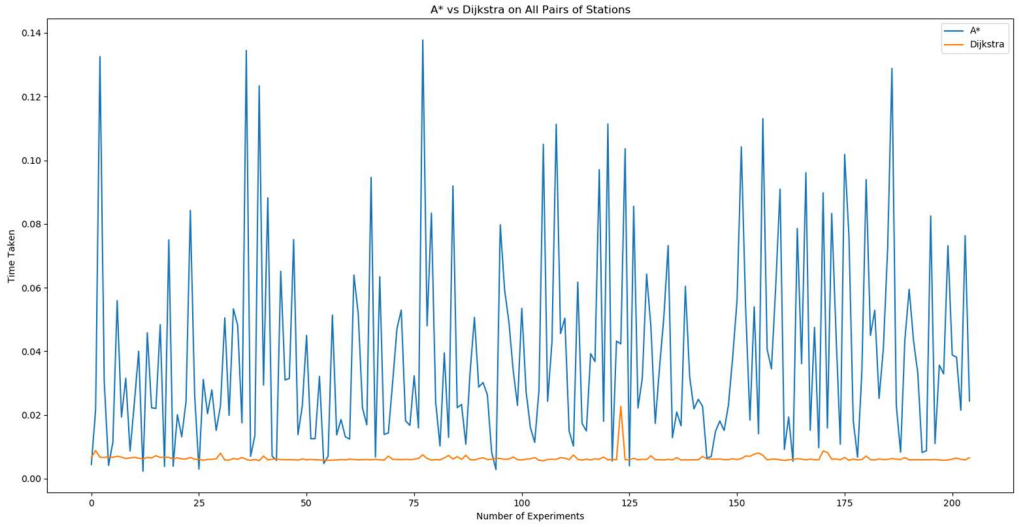

Figure 6: Graph displaying the runtimes of A using a bad heuristic (blue) vs Dijkstra's (orange) on pairs of stations.*

(Some experiments at the end of the array were skipped because running the tests took too much time).

In this graph, a badly optimized heuristic function was used. This resulted in Dijkstra's being clearly better, by even a bigger margin than above. This makes sense, as when using a good heuristic function (accurate and efficient), A* can explore better paths than Dijkstra's. However, when using a bad heuristic function (in this case, badly optimized), the benefits of the heuristic function might be marginal and can make A* worse than Dijkstra's.

Overall, A* with a good heuristic function has benefits over Dijkstra's, but having a badly optimized heuristic can be extremely costly.

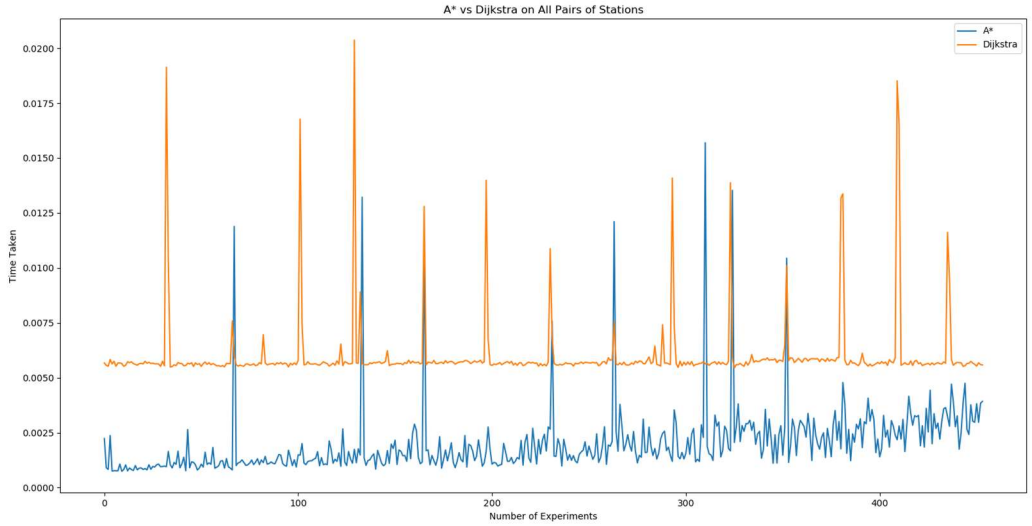

Figure 7: Graph displaying the runtimes of A (blue) vs Dijkstra's (orange) on pairs of stations.*

(Some experiments in between were skipped for visibility and better show the increase)

The combinations of stations appear to not affect Dijkstra's much. This could be explained by Dijkstra's not having to rely on something "external" like how accurate a heuristic function is on the pair of nodes. So, its efficiency is relatively stable. In this case, the pair of nodes do not affect runtime much.

However, while the runtime of A* is still always better than Dijkstra's, the runtime appears to gradually increase. This could be explained by the heuristic function not being as accurate as the path becomes more complex. When the stations are on the same line, the path is more likely to be simple and the heuristic can accurately guide the function and make a huge difference. But, when multiple transfers are needed, the path may be complex, in which case the heuristic may not be as accurate and reduce the advantages gained. Overall, this is still just based on the heuristic function, as better heuristic functions may still be accurate on more complex paths.

However, the number of lines and shortest path should be unrelated. For example, even if we are finding the path between 2 stations on the same line, the shortest path might take a route that visits stations on different lines. So, the minimum number of lines used is the minimum number of lines between the 2 stations, but the shortest path may use more. In conclusion, there isn't a relationship between the shortest path based on distance and number of lines. In fact, finding the shortest path based on number of lines is an entirely different shortest path problem.

Part 4

The UML diagram utilizes the design principles of inheritance and polymorphism. The *Graph* class is a generic class where its methods are abstract methods. The class can be inherited and have its methods overwritten by child classes. In the diagram, *WeightedGraph* is a child class that inherits from *Graph* class. It overwrites the *get_adj_nodes()*, *add_node()*, *add_edge()*, *get_num_of_nodes()*, and *w()*

methods of *Graph* by implementing its own functionality for those methods. Since Python does not support interfaces like Java to enable *Graph* to be inherited as a generic class, abstract class and abstract class methods were used instead.

Likewise, *SPAlgorithm* is an abstract class with a single abstract method that is intended to be inherited from. All three algorithms, Dijkstra, Bellman Ford, and A* are child classes of *SPAlgorithm* that overwrite the *calc_sp()* method with their own functionality. The UML diagram utilizes the modular Strategy pattern here, since the algorithms are under the *SPAlgorithm* class, which enables additional algorithms to be added in the future if needed.

To implement the A* algorithm under the UML diagram, an auxiliary class *A_star* was created with a method that calculated the heuristic function. After this heuristic function was calculated, the graph, source node, destination node, and heuristic function were given to the *A_star_helper* class which contains the actual A* algorithm.

For future work, to make the application accept String nodes instead of just integers, a new class called *Node* can be created to represent nodes. The *Node* class can be a generic class that could be inherited and extended by child classes to create specific *Node* types. Within the *Node* class, there would be an identifier state variable which would store an Integer, String, or other types of values. The graph class can be further extended by implementing other types of graphs such as unweighted graphs and undirected weighted graphs. Implementing the unweighted graph could automatically assign an edge value of 1 to all edges. For the undirected weighted graph, every time a new edge is created, the reverse of that edge would be automatically added in as well.

Appendix

Part 1:

Refer to part_1.py for part 1.

Part 2:

Refer to *a_star_part2.py*.

Part 3:

Refer to file *part3.py*.

Requires A* algorithm from a_star.py.

Requires Dijkstra's algorithm from shortest_path.py.

Part 4:

Refer to *part4* folder.