Final Project Report

Team Members:

`Aarav Pant` (Part 4 & Report)

`Ankur Pandey` (Part 1 and Part 2 & Report)

`Megh Joshi` (Part 3 & Part 5)

Table of Content

Experiments / Analysis

Part 1.3

Introduction

- **Purpose:** We have analyze the algorithms. by experimentation, testing and/or theoretical explanation. The analysis done by experimentation and testing were on time complexity with three variables:
    - Graph Size
    - Density
    - Relaxation Limit (k)

  The analysis done by an experiment on accuracy was done only based on relaxation limit because for comparing accuracy, the number of relaxations is the main limitation which will cause the algorithm to give inaccurate results.

  We can't break down space complexity without importing sys or mem, but since that's not allowed we do that theoretically.
- **Background:** Mainly Dijkstra's algorithm gives the shortest path for a graph with positive weights. It can't handle negative weights because it's a greedy algorithm. Meanwhile, Bellman-Ford handles both negative and positive weights.

Functions Used

- **MinPriorityQueue Class:** Our Dijkstra's algorithm will be using a minimum priority queue in order to lessen the time taken to execute and bring the time complexity to $\Theta(E + VlogV)$.

```
class MinPriorityQueue:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def empty(self):
        return len(self.heap) == 0

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def insert(self, val):
        self.heap.append(val)
        self._heapify_up(len(self.heap) - 1)

    def delete_min(self):
        if len(self.heap) == 0:
            return None

        self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]
        min_val = self.heap.pop()
        self._heapify_down(0)
        return min_val

    def _heapify_up(self, index):
        while index > 0 and self.heap[self.parent(index)] > self.heap[index]:
            self.heap[index], self.heap[self.parent(index)] =self.heap[self.parent(index)], s
            index = self.parent(index)

    def _heapify_down(self, index):
        while index < len(self.heap):
            smallest = index
            left = self.left_child(index)
            right = self.right_child(index)

            if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
                smallest = left
            if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
                smallest = right

            if smallest != index:
                self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
                index = smallest
            else:
                break
```

- **Graph Generation:** We have a `generate_random_graph` function which uses the Graph class and then generates a random weighted graph based on `size` and `density` mainly. There are two additional optional parameters `directed` and `negative_weights` which can be used to alter how the graph is being created. The `directed` parameter by default is set to true because we'll be working with them only and the negative_weights is set to false because our design for time complexity and accuracy experiments don't require the use of negative weights.

```
def generate_random_graph(size, density, directed=True, negative_weights=False):
    if density < 0 or density > 1:
        raise ValueError("Density must be between 0 and 1.")

    graph = Graph(size, directed)

    max_edges = size * (size - 1) // 2  # Maximum possible edges in an undirected graph
    if directed:
        max_edges *= 2  # Double max edges for directed case

    num_edges = int(max_edges * density)

    for _ in range(num_edges):
        while True:
            u = random.randint(0, size - 1)
            v = random.randint(0, size - 1)
            if u != v and graph.adj_matrix[u][v] == 0:
                if negative_weights:
                    weight = random.randint(-5, 10)
                else:
                    weight = random.randint(1, 10)
                graph.add_edge(u, v, weight)
                break

    return graph
```

- **Time function:** It will be used to calculate the time each algorithm takes using the time library. The function asks for two letters ‘d’ or ‘b’ and based on the input calls the method on the graph given. If ‘d’ is chosen, then Dijkstra’s algorithm is called and Bellman-Ford is called if the other option is chosen.

```
def time_algorithm(graph, source, algorithm='d', relaxation_limit=3):
    if algorithm.lower() == 'd':
        start_time = time.time()
        graph.dijkstra(source, k=relaxation_limit)
        end_time = time.time()
        return end_time - start_time
    start_time = time.time()
    graph.bellman_ford(source, k=relaxation_limit)
    end_time = time.time()
    return end_time - start_time
```

- **Normal Dijkstra’s:** We have created a normal Dijkstra’s algorithm which doesn’t have the k limit restriction. This will be used for calculating accuracy.

```
def dijkstra_without_restriction(graph, start_vertex_data):
    start_vertex = graph.vertex_data.index(start_vertex_data)
    distances = [float('inf')] * graph.size
    predecessors = [None] * graph.size
    distances[start_vertex] = 0

    pq = MinPriorityQueue()
    pq.insert((0, start_vertex))

    while not pq.empty():
        (current_distance, u) = pq.delete_min()

        if current_distance > distances[u]:
            continue

        for v in range(graph.size):
            if graph.adj_matrix[u][v] != 0 and distances[v] > distances[u] + graph.adj_matrix
                alt = distances[u] + graph.adj_matrix[u][v]
                distances[v] = alt
                predecessors[v] = u
                pq.insert((alt, v))

    paths = {}
    for vertex in range(graph.size):
        current_node = vertex
        path = [current_node]
        while predecessors[current_node] is not None:
            path.insert(0, predecessors[current_node])
            current_node = predecessors[current_node]
        paths[vertex] = path

    return distances, paths
```

- **Normal Bellman-Ford:** We have created a normal Bellman Ford's algorithm which doesn't have the k limit restriction. This will also be used for calculating accuracy.

```
def bellman_ford_without_restriction(graph, start_vertex_data):
    start_vertex = graph.vertex_data.index(start_vertex_data)
    distances = [float('inf')] * graph.size
    predecessors = [None] * graph.size
    distances[start_vertex] = 0

    for i in range(graph.size - 1):
        for u in range(graph.size):
            for v in range(graph.size):
                if graph.adj_matrix[u][v] != 0:
                    if distances[u] + graph.adj_matrix[u][v] < distances[v]:
                        distances[v] = distances[u] + graph.adj_matrix[u][v]
                        predecessors[v] = u

    # Negative cycle detection
    for u in range(graph.size):
        for v in range(graph.size):
            if graph.adj_matrix[u][v] != 0:
                if distances[u] + graph.adj_matrix[u][v] < distances[v]:
                    raise ValueError("Graph contains a negative cycle")

    paths = {}
    for vertex in range(graph.size):
        current_node = vertex
        path = [current_node]
        while predecessors[current_node] is not None:
            path.insert(0, predecessors[current_node])
            current_node = predecessors[current_node]
        paths[vertex] = path

    return distances, paths
```

- **Measure Accuracy:** We use a function called `measure_accuracy` to calculate the total distance error (by subtracting the difference in distances of the limited algorithm and without limitation version of the algorithm) for a given k.

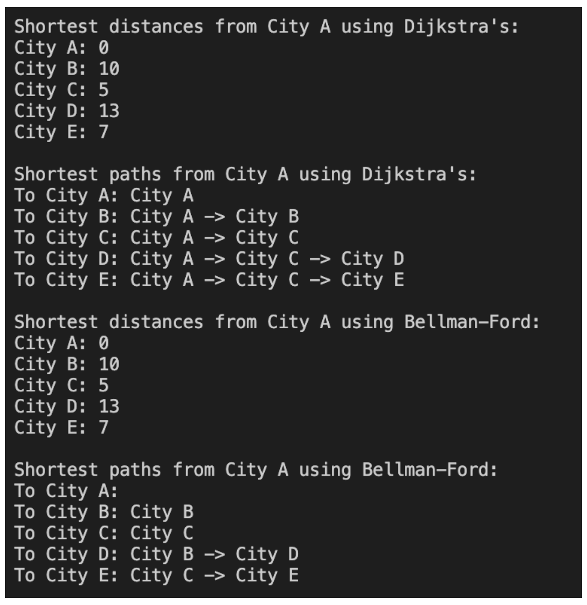
```
def measure_accuracy(graph, source, algorithm, relaxation_limit=4):
    if algorithm == 'dijkstra':
        distances, _ = graph.dijkstra(source, k=relaxation_limit)
        unrestricted_distances, _ = dijkstra_without_restriction(graph, source)

    elif algorithm == 'bellman_ford':
        distances, _ = graph.bellman_ford(source, k=relaxation_limit)
        unrestricted_distances, _ = bellman_ford_without_restriction(graph, source)
    else:
        return None

    distance_error_sum = 0

    for i in range(len(distances)):
        distance_error_sum += abs(distances[i] - unrestricted_distances[i])

    return distance_error_sum
```

Experiments

- Initially, we start by doing basic experimentation to check if the algorithms were implemented correctly and are functioning as expected. The initial experiments are as follows:-
    - Checking that both Dijkstra's and Bellman-Ford give the correct shortest path and distances(as far as possible for a given k) for the same graph.
    - Then we create a negative weighted graph and check if Dijkstra's and Bellman-Ford give the same answer
    - Finally we create a graph with negative cycle. We have designed the algorithm such that it should return empty list for distance and empty dictionary for paths in the case it encounters a negative cycle. We check if the output we get is as desired.
- Next we design three experiments to check time complexity. We break them down as follow:-
    - Firstly, we check what happens to the time complexity for variable graph sizes. We create a list with graph sizes from 5 to 100 skip 10 using the `generate_random_graph` function and plot the changes using `matplotlib` library and the `time_algorithm` function. A new graph will be created inside a loop for each size in `graph_sizes` list.
    - Secondly, we check what happens to the time complexity for variable densities. Again we create a list with different densities from 0.1 to 1 and use our `generate_random_graph` to create a graph inside a loop for each density in the `densities` list.
    - Finally, we check what happens to the time complexity for variable relaxation limits. Again we create a list with different relaxation limits from 1 to 10 and use our `generate_random_graph` to create a graph inside a loop for each k in the `relaxation_limit` list.
- We do one more experiment for checking how accurate the algorithms are. For this we use the `dijkstra_without_restriction` and `bellman_ford_without_restriction` . The logic behind this experiment is that if we first compare the difference in the distance given by the limited Dijkstra and without limitation Dijkstra and limited Bellman-Ford and without limitation Bellman-Ford and then sum all of the differences for a particular value of k, we can find how the accuracy drops when we attach the k limit.

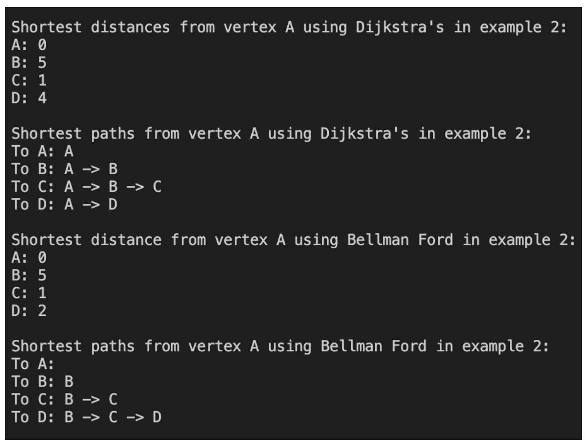
Results

- **Results for the initial experiments:-**
    - Dijkstra's and Bellman-Ford both give the same answer which means that our algorithm works as expected for positive weights.



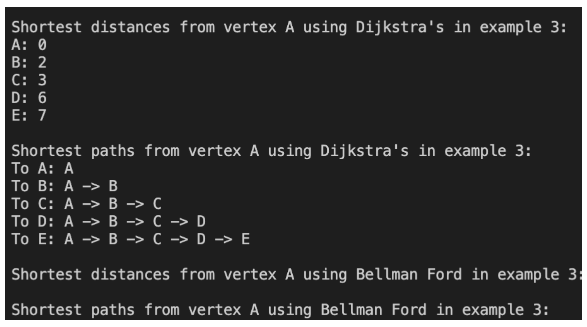
```
Shortest distances from City A using Dijkstra's:
City A: 0
City B: 10
City C: 5
City D: 13
City E: 7

Shortest paths from City A using Dijkstra's:
To City A: City A
To City B: City A -> City B
To City C: City A -> City C
To City D: City A -> City C -> City D
To City E: City A -> City C -> City E

Shortest distances from City A using Bellman-Ford:
City A: 0
City B: 10
City C: 5
City D: 13
City E: 7

Shortest paths from City A using Bellman-Ford:
To City A:
To City B: City B
To City C: City C
To City D: City B -> City D
To City E: City C -> City E
```

- Dijkstra's and Bellman-Ford give different answers with Bellman-Ford giving the correct answer. Hence, Bellman-Ford is the better algorithm for negative weights



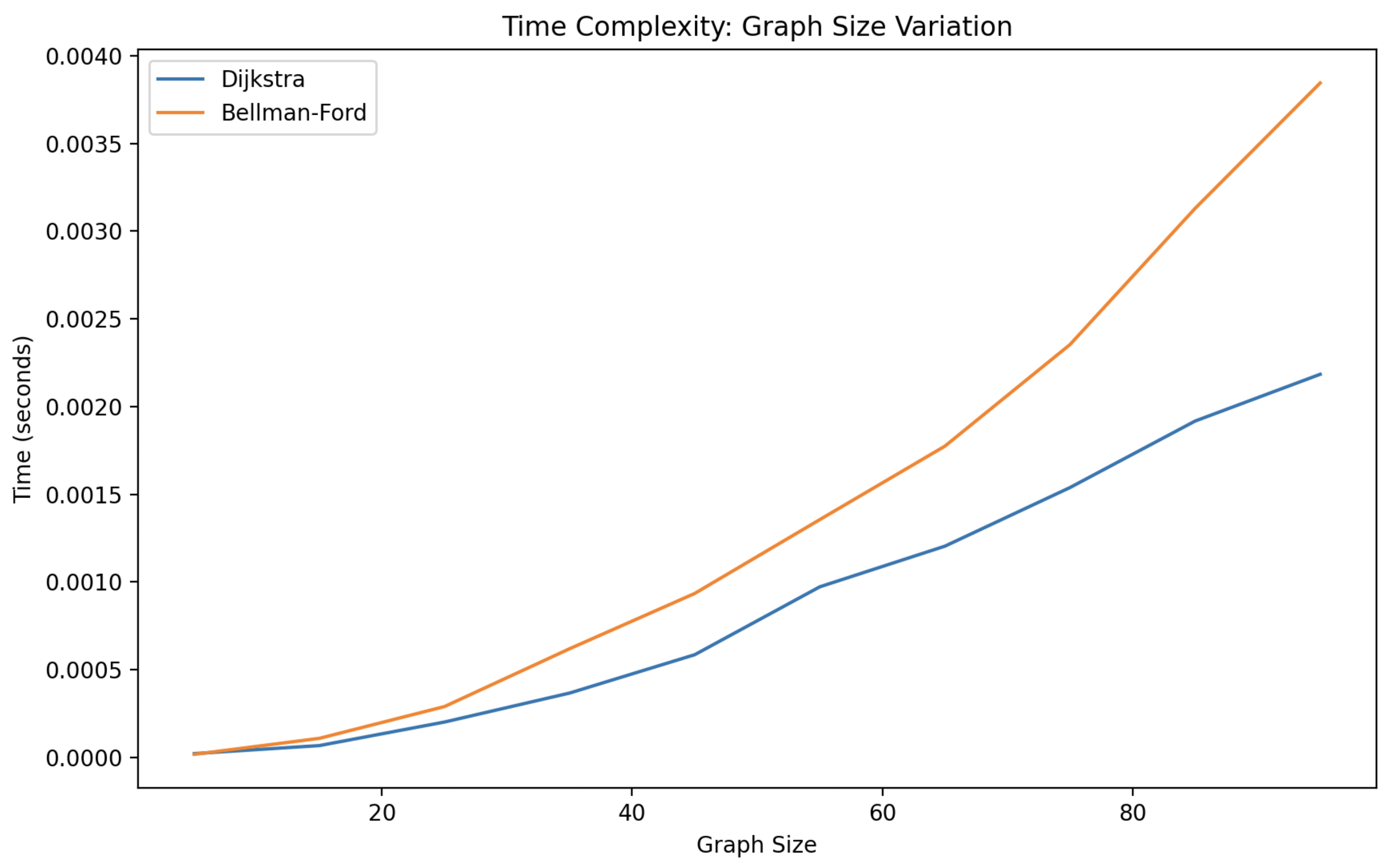
```
Shortest distances from vertex A using Dijkstra's in example 2:
A: 0
B: 5
C: 1
D: 4

Shortest paths from vertex A using Dijkstra's in example 2:
To A: A
To B: A -> B
To C: A -> B -> C
To D: A -> D

Shortest distance from vertex A using Bellman Ford in example 2:
A: 0
B: 5
C: 1
D: 2

Shortest paths from vertex A using Bellman Ford in example 2:
To A:
To B: B
To C: B -> C
To D: B -> C -> D
```

- Dijkstra gives wrong results whereas Bellman-Ford doesn't output anything because it detects a negative cycle.

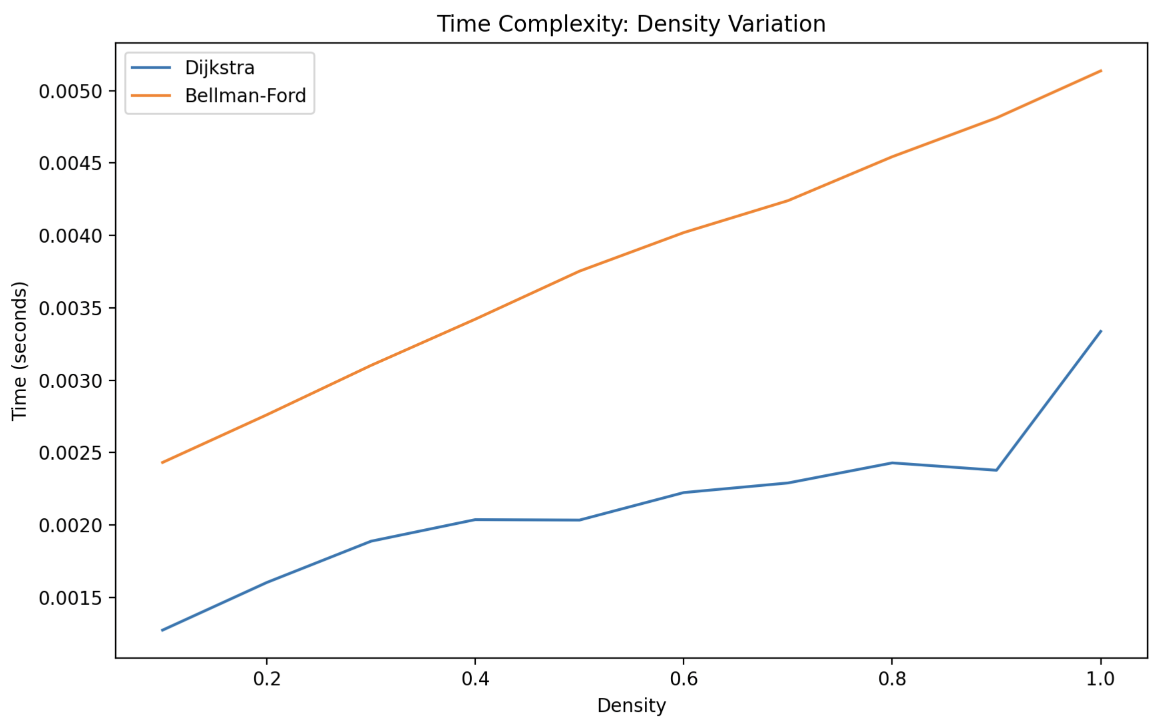


```
Shortest distances from vertex A using Dijkstra's in example 3:
A: 0
B: 2
C: 3
D: 6
E: 7

Shortest paths from vertex A using Dijkstra's in example 3:
To A: A
To B: A -> B
To C: A -> B -> C
To D: A -> B -> C -> D
To E: A -> B -> C -> D -> E

Shortest distances from vertex A using Bellman Ford in example 3:

Shortest paths from vertex A using Bellman Ford in example 3:
```

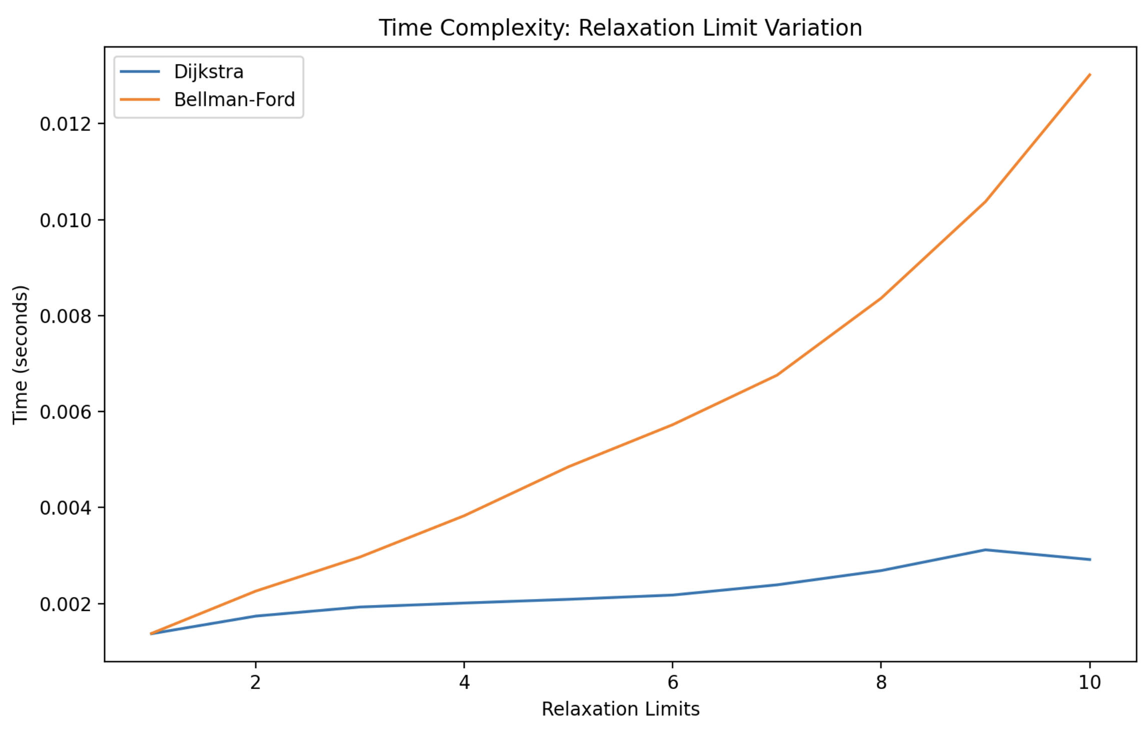
- **For the time complexities, the results are as follows:-**
  - We notice for increase in graph sizes, the time increases with Bellman-Ford displaying higher time than Dijkstra's
    - This result was as expected because theoretically their time complexities depend on Vertices and Edges and more the number of vertices and edges, the higher will be the time taken. Additionally, the time taken by Bellman-Ford is higher than Dijkstra as expected because even theoretically it should be higher.



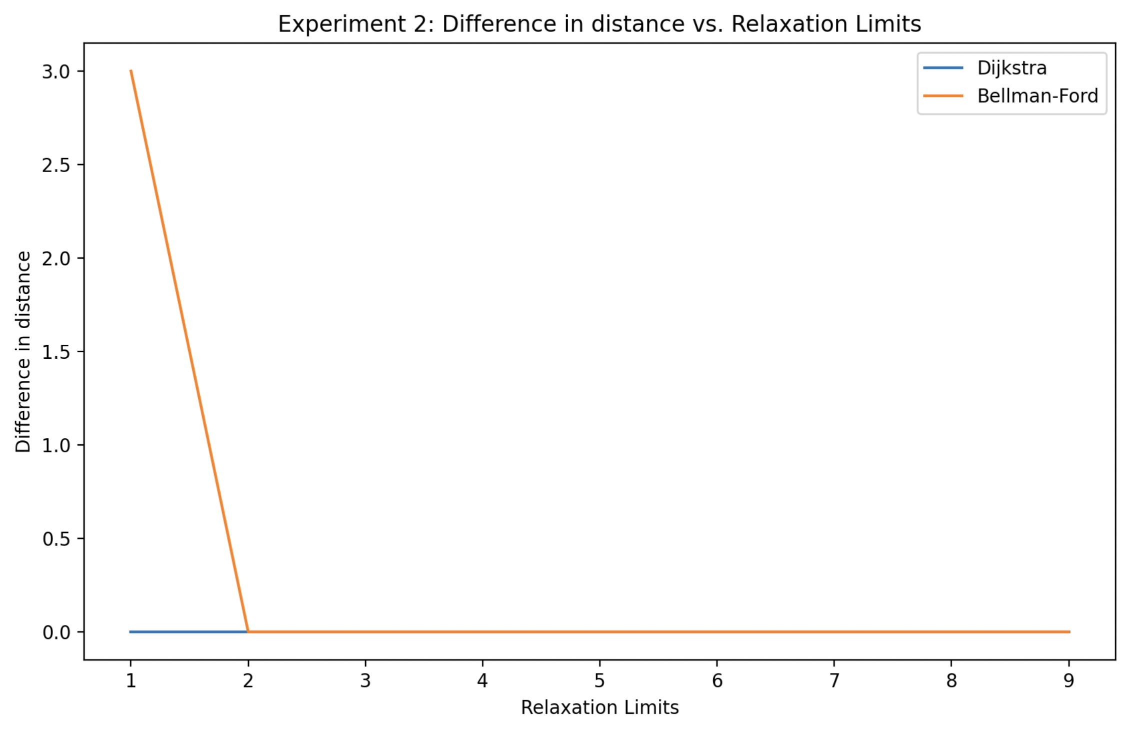
- We notice for increasing densities, the time increases again but this time the Bellman-Ford has significantly higher time than Dijkstra's.
  - This result was as expected because theoretically their time complexities depend on Edges and more the number of edges (i.e. dense graph), the higher will be the time taken. Additionally, the time taken by Bellman-Ford is higher than Dijkstra as expected because even theoretically it should be higher.



- For increase in relaxation the time taken increases again and as usual, the Bellman-Ford algorithm has higher time than Dijkstra's
  - Here we don't have theoretical proof, but logically if we have set a relaxation limit, then obviously it will stop running/relaxing edges after the limit has been reached. Thus higher the relaxation limit, higher will be the time taken to execute the algorithm. This time the time taken by Bellman-Ford is higher than Dijkstra's is not affected by relaxation limit, but rather due to the graph size and density because even for fixed density and size, Bellman-Ford theoretically takes more time to execute than Dijkstra for the same relaxation limit.



- For the accuracy experiment, we find that if k = 1 or 2 the distance error is higher (If you increase the graph size then even for k = 3 or maybe more the distance error will be higher). After the initial k values, the distance error comes about to be 0 because those relaxations are enough to find the accurate shortest distance values.
  - This is expected because if k is small enough then obviously there is a very high chance in big graphs that the distance chosen is not the shortest and thus is different than the normal Dijkstra's. Also if we keep on increasing graph size(not done automatically but rather manually in experiments) we find that the least value of k for which there is a graph error increases for eg. for graph size 100 it shows distance error for k = 1 and k = 2 but for graph size 200 it shows distance error for k = 1, 2 and 3 as well. This is most likely due to the fact that if graph size increases then there must be more relaxations needed to navigate the increasing number of edges in order to find the shortest path. The increase in k fact also holds for increasing densities



- Finally for space complexity, we didn't do an experiment but we can say theoretically that for our implementation both algorithms should have a space complexity of O(V) since they both require tentative storing of distances. The graph class has the space complexity of O(V^2) due to the adj matrix.

Part 2

Introduction

- In this report, we will explore strategies to design all-pairs shortest path algorithms capable of handling both positive and negative edge weights. As told in the clarifications, we have designed two functions to handle the positive and negative weights.

Functions Used

- **Graph class:** We use the same graph class as in part 1.
- **Dijkstra's Algorithm:** We use the `dijkstra_without_restriction` from part 1 along with the MinPriorityQueue class.
- **Bellman-Ford Algorithm:** We use the `bellman_ford_without_restriction` from part 1.
- **Graph generator:** We use the same `random_graph_generator` function from part 1.
- The two main algorithm functions to handle positive and negative weights are as follows:-
    - **all_pairs_dijkstra:** It handles the graphs with positive weights. The function goes through all vertices in a graph and finds the distance, paths using the `dijkstra_without_restriction` function where each vertex acts as the source. Then we add values to the dictionaries (distance, path, previous) with (u, v) being the key where u is the source and v is the destination vertex.

```
def all_pairs_dijkstra(graph):
    """Assumes the graph only has positive edge weights"""
    shortest_paths = {}
    previous = {}
    distances = {}

    for i in range(graph.size):
        source = graph.vertex_data[i]
        dist, paths = graph.dijkstra_without_restriction(source)

        for vertex in range(graph.size):
            shortest_paths[(i, vertex)] = paths[vertex]
            previous[(i, vertex)] = paths[vertex][len(paths[vertex])-2]
            distances[(i, vertex)] = dist[vertex]

    return distances, shortest_paths, previous
```

    - **all_pairs_bellman_ford:** It is similar but handles negative weights using the `bellman_ford_without_restriction` .

```
def all_pairs_bellman_ford(graph):
    shortest_paths = {}
    previous = {}
    distances = {}

    for i in range(graph.size):
        source = graph.vertex_data[i]
        dist, paths = graph.bellman_ford_without_restriction(source)

        for vertex in range(graph.size):
            shortest_paths[(i, vertex)] = paths[vertex]
            previous[(i, vertex)] = paths[vertex][len(paths[vertex])-2]
            distances[(i, vertex)] = dist[vertex]

    return distances, shortest_paths, previous
```

Time complexity analysis for dense graphs

1. `all_pairs_dijkstra`
    - **Inner Loop:** We're calling `dijkstra_without_restriction` for each vertex as the source. Since Dijkstra's complexity is O(V^2) for dense graphs, the inner part has O(V^2) complexity.
    - **Outer Loop:** The outer loop executes V times.
    - **Total Complexity:** O(V * V^2) = **O(V^3)**
2. `all_pairs_bellman_ford`

- **Inner Loop:** `bellman_ford_without_restriction` executes for each vertex. Bellman-Ford's complexity for dense graphs is O(VE), and E is approximately V^2, resulting in O(V^3) for the inner part.
- **Outer Loop:** The outer loop executes V times.
- **Total Complexity:** O(V * V^3) = **O(V^4)**

Summary

Due to the nested loop structure where we execute the single-source algorithms (Dijkstra's or Bellman-Ford) for each vertex in the graph, the overall complexity increases. In a dense graph, the single-source algorithms themselves have complexities of O(V^2) and O(V^3) respectively.

Conclusion

For dense graphs, both of our all-pairs shortest path algorithms have a higher time complexity than their single-source counterparts:

- `all_pairs_dijkstra` has a complexity of O(V^3).
- `all_pairs_bellman_ford` has an even worse complexity of O(V^4).

Part 3.2

Introduction

In this report, we worked on the A* algorithm, which is an enhanced version of Dijkstra's algorithm that incorporates a heuristic function to optimize pathfinding between two nodes. The function `A_Star` will be implemented to accept a directed weighted graph, source, destination, and heuristic, returning a predecessor dictionary and the shortest path.

This report will address the following:

- The limitations of Dijkstra's algorithm that A* resolves.
- Comparison methods for Dijkstra’s and A* under various conditions.
- Effects of using an arbitrary heuristic on A* performance.
- Situations where A* is preferable over Dijkstra’s algorithm.

Functions used

- `A_Star`: Implements the A* algorithm to find the optimal path from a source to a destination node using heuristics.

```
def A_Star(graph, source, destination, heuristic):
    open_list = MinPriorityQueue()
    open_list.insert((0 + heuristic[source], source))
    predecessors = {source: None}
    costs = {source: 0}

    while len(open_list.heap) > 0:
        _, current = open_list.delete_min()

        if current == destination:
            break

        for neighbor, weight in graph[current].items():
            new_cost = costs[current] + weight
            if neighbor not in costs or new_cost < costs[neighbor]:
                costs[neighbor] = new_cost
                priority = new_cost + heuristic[neighbor]
                open_list.insert((priority, neighbor))
                predecessors[neighbor] = current

    path = reconstruct_path(predecessors, source, destination)
    if not path:
        return predecessors, path, float('inf')

    return predecessors, path,costs [destination]
```

- `reconstruct_path` : Finds the optimal path from destination back to source using the predecessor data from A*.

```
def reconstruct_path(predecessors, start, end):
        if end not in predecessors:
            return []
        path = []
        while end is not None:
            path.append(end)
            end = predecessors.get(end)
        path.reverse()
        return path
```

- `MinPriorityQueue` : Used for efficiently managing the open list in the A* algorithm.

```
class MinPriorityQueue:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def insert(self, val):
        self.heap.append(val)
        self._heapify_up(len(self.heap) - 1)

    def delete_min(self):
        if len(self.heap) == 0:
            return None

        self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]
        min_val = self.heap.pop()
        self._heapify_down(0)
        return min_val

    def _heapify_up(self, index):
        while index > 0 and self.heap[self.parent(index)] > self.heap[index]:
            self.heap[index], self.heap[self.parent(index)] = self.heap[self.parent(index)],
            index = self.parent(index)

    def _heapify_down(self, index):
        while index < len(self.heap):
            smallest = index
            left = self.left_child(index)
            right = self.right_child(index)

            if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
                smallest = left
            if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
                smallest = right

            if smallest != index:
                self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
                index = smallest
            else:
                break


    def put(self, node, priority):
        self.insert((priority, node))

    def is_empty(self):
        return len(self.heap) == 0
```

Time complexity **analysis**

`MinPriorityQueue` **Operations:**

- **Insertion (** `insert` **)**: O(log n) - Inserting an element is done by adding it at the end and then executing a heapify-up operation, which requires logarithmic time in relation to the number of elements in the heap.
- **Deletion of the minimum element (** `delete_min` **)**: O(log n) - Removing the root of the heap is done by swapping it with the last element, removing the final element, and then conducting a heapify-down operation, which takes logarithmic time.

`A_Star` **Function:**

- **Overall Complexity**: O((E + V) log V) - Each vertex is theoretically added to the priority queue once (O(V log V)), and each vertex's edges are investigated once (O(E log V)), where E is the total number of edges and V is the total number of vertices. The logarithmic factor results from priority queue operations (insertion/deletion).

`reconstruct_path` **Function:**

- **Linear in the number of nodes in the path (O(p))**: The path is constructed by tracing back from the destination node to the source node, which takes time proportional to the number of nodes in the path (p).

Questions

Ques 1: What issues with Dijkstra's algorithm is A* trying to address ?

Ans 1:

1. `Heuristic Guidance` : Dijkstra's algorithm explores all possible paths to find the shortest path without any direction. A* uses a heuristic to guide its search towards the destination, making it significantly faster for many applications, especially in large graphs or maps.
2. `Performance in Pathfinding` : In practical pathfinding problems, especially in game development or robot navigation, where a map can be vast, Dijkstra's algorithm can be slow because it treats all directions equally and explores all possible paths. A* focuses on promising paths, reducing the exploration of unnecessary nodes.
3. `Efficiency in Large Graphs` : For very large graphs with one specific target, Dijkstra's algorithm may perform unnecessary computations, whereas A* can drastically reduce the search space using its heuristic function

Ques 2: How would you empirically test Dijkstra's vs A*?

Ans 2:

1. `Execution Time` : Measure the time it takes for each algorithm to find the shortest path in a variety of graphs, ranging from sparse to dense, and from small to very large. This comparison can highlight A*'s efficiency improvements.
2. `Number of Nodes Explored` : Track and compare the number of nodes each algorithm must explore before reaching the destination. A* should explore fewer nodes due to its heuristic guidance.
3. `Accuracy and Optimality` : Verify that both algorithms return the optimal path. A*'s heuristic must be admissible (never overestimate the cost to reach the goal) for this to hold true.

4. `Varying Heuristics`: For A*, test different heuristics to observe how the choice of heuristic affects performance and node exploration. Compare these results with Dijkstra's consistent performance.

Ques 3: If you generated an arbitrary heuristic function (like randomly generating weights), how would Dijkstra's algorithm compare to A*?

Ans 3:

Using an arbitrary (randomly generated) heuristic function in A* could deteriorate its performance, making it comparable to or even worse than Dijkstra's algorithm because:

- If the heuristic is not admissible (it overestimates costs), A* could fail to find the shortest path.
- A random heuristic might not provide meaningful guidance, causing A* to explore paths similar to Dijkstra's exhaustive search.
- The benefit of A* comes from its heuristic being informative. An arbitrary heuristic likely removes this benefit, negating A*'s advantage over Dijkstra's algorithm.

Ques 4: What applications would you use A* instead of Dijkstra's ?

Ans 4: A* is preferred over Dijkstra's algorithm in scenarios where:

1. `Pathfinding on Maps:` `A*` is widely used in GPS and game development for finding paths between locations, as it can efficiently handle large maps.
2. `Robotics`: In robotics, A* is used for navigation and obstacle avoidance, where quick and efficient pathfinding is crucial.
3. `Puzzle and Game AI`: A* is suitable for solving puzzles (e.g., sliding puzzles, Sokoban) and for game AI where entities need to navigate complex environments efficiently.
4. `Graph Traversal with Known Goal`: Whenever there's a specific goal and the graph is large, A* is advantageous because it can significantly reduce the search area using heuristics that estimate distances to the goal.

Summary

In summary, A* is used over Dijkstra's when the goal is known and an appropriate heuristic can be applied to guide the search, making the process faster and more efficient for many practical applications.

Part 4

Introduction

In this part of my report, we examine the performance of Dijkstra's and the A* algorithms on the London Subway system as a practical dataset. Unlike tests with randomly produced graphs, this real-world scenario underlines the significance of the heuristic function in A*. The dataset contains around 300 stations, each represented as a node, with connections as edges weighted by geographical distance between stations. For the A* algorithm, we estimated the heuristic using the Euclidean distance, which is the straight-line distance between stations. This method will assist analyze and compare how each algorithm performs across different station combinations in the subway network, showing when A* outperforms Dijkstra's algorithm and when their performances are comparable.

This report will address the following:

1. **Performance Comparison:**
    - *When does A outperform Dijkstra?* Explore scenarios where the heuristic-driven nature of A* reduces the search space and leads to faster solution times.
    - **When are they comparable?** Analyze conditions under which both algorithms perform similarly, likely due to low variability in path choices or small dataset sizes.
2. **Impact of Network Layout on Algorithm Performance:**
    - **Stations on the same line:** Examines how both algorithms handle straightforward, linear routes with minimal junctions.

    - **Stations on adjacent lines:** Assess performance when routes require simple transfers, which might introduce complexity in pathfinding.
    - **Stations requiring several transfers:** Investigate how increased route complexity affects algorithm efficiency, especially where multiple line changes are necessary.
3. **Line Usage Analysis:**
    - **Number of lines used in the shortest paths:** Utilizes the line information provided in the dataset, calculate how many different lines are typically involved in finding the shortest paths by each algorithm. This will help understand how effectively each algorithm navigates the network topology.

Functions used

- `MinPriorityQueue` **Class**: Implements a priority queue using a binary heap structure, supporting operations like insertion, deletion, and heap adjustments to manage priorities efficiently.
- `dijkstra(graph, start, end)`: Dijkstra's algorithm to find the shortest path and distance from a start to an end node in a graph.
- `euclidean_distance(coord1, coord2)`: Calculates the Euclidean distance between two points (coordinates) given as tuples.
- `heuristic(station_id, destination_id)`: Calculates a heuristic based on the Euclidean distance between two stations, used by the A* algorithm.
- `A_Star(graph, source, destination, heuristic)`: Implements the A* algorithm, using a priority queue to efficiently find the shortest path from source to destination with the help of a heuristic function.
- `reconstruct_path(predecessors, start, end)`: Constructs the path from start to end using a map of predecessors, typically used after pathfinding algorithms to trace the route.
- `parse_stations(file_path)`: Parses a CSV file to extract station data, creating a dictionary of station IDs mapped to their properties.
- `parse_connections(file_path)`: Reads connection data from a CSV file to list the connections between stations, including details like distances.
- `build_graph(stations, connections)`: Constructs a graph from station and connection data, where nodes represent stations and edges are weighted by the Euclidean distance between them.
- `measure_performance(graph, start_id, end_id, heuristic)`: Measures and compares the execution times of Dijkstra's and A* algorithms for the given start and end nodes.
- `plot_performance_comparison(labels, dijkstra_times, astar_times, title, image_name)`: Plots and saves a bar chart comparing the performance (execution time) of Dijkstra's and A* algorithms across different scenarios.
- `count_line(graph, source, destination, connections_filename)`: Determines the number of line changes required for the shortest path from source to destination using Dijkstra's algorithm.
- `categorize_station_pairs_by_transfers(graph, stations, connections_filename)`: Categorizes all station pairs into groups based on the number of transfers required, organizing them into same-line, adjacent-transfer, and multiple-transfer categories.

Experiments

Once you have generated the weighted graph and the heuristic function, use it as an input to both A* and Dijkstra's algorithm to compare their performance. It might be useful to check all pairs shortest paths, and compute the time taken by each algorithm for all combination of stations. Using the experiment design, answer the following questions

Ques 1: When does A* outperform Dijkstra? When are they comparable? Explain your observation why you might be seeing these results ?

Ans 1:

- *`A Outperforms Dijkstra`*:
    - **Effective Heuristic**: A* typically outperforms Dijkstra in scenarios where the heuristic is effectively guiding the algorithm towards the goal. This reduces the number of nodes explored, making A* faster especially in larger or more complex graphs.

    - **Long Distances with Clear Goals**: In situations where the destination is far from the starting point and the heuristic can accurately estimate distances to the goal, A* leverages this to prioritize more promising paths, thus reducing the overall computation time compared to Dijkstra's exhaustive node exploration.
    - **Sparse Graphs**: In less densely connected graphs where paths are not straightforward, A*'s heuristic helps it skip irrelevant parts of the graph, unlike Dijkstra's, which would still explore them.
- `A and Dijkstra Are Comparable :`
    - **Dense and Small Graphs**: In dense networks where destinations are relatively close or the graph is small, both algorithms tend to perform similarly because the advantage of the heuristic in A* is minimized.
    - **Poor or Non-Informative Heuristic**: If the heuristic used by A* does not provide a good estimate of the actual shortest path to the destination (e.g., in highly irregular topographies or misleading geographic data), its performance advantage diminishes, making it comparable to Dijkstra's.
    - **Uniform Cost Paths**: When all paths have similar or uniform costs, the heuristic does not significantly impact the decision process in A*, making its performance similar to Dijkstra's, which does not use a heuristic.

These observations suggest that the performance of A* relative to Dijkstra's hinges significantly on the characteristics of the graph and the effectiveness of the heuristic used in guiding the search process.

Ques 2: What do you observe about stations which are 1) on the same lines, 2) on the adjacent lines, and 3) on the line which require several transfers?

Ans 2:

Methodology and Thinking

Graph and Connection Data: We Utilize the graph constructed from the subway stations and their connections, where each station is linked by edges representing direct paths, weighted by distance.

Categorization of Station Pairs: By iterating over all pairs of stations, we calculate the number of line changes required using the shortest path between them (determined by Dijkstra's algorithm). This data was used to categorize pairs into three groups:

- **Same Line**: Pairs that require no line changes to reach one another.
- **Adjacent Line**: Pairs that require exactly one line change.
- **Multiple Transfers**: Pairs that require more than one line change.

```
def count_line(graph, source, destination, connections_filename):
    distance, shortest_path = dijkstra(graph, source, destination)

    if shortest_path == []:
        return -1


    with open(connections_filename, mode='r') as file:
        csv_reader = csv.DictReader(file)
        connections = {(int(row['station1']), int(row['station2'])): int(row['line']) for row

    num_line_transfers = 0
    current_line = None

    for i in range(len(shortest_path) - 1):
        station1, station2 = shortest_path[i], shortest_path[i + 1]
        line = connections.get((station1, station2)) or connections.get((station2, station1))

        if line is not None:
            if current_line is not None and line != current_line:
                num_line_transfers += 1
            current_line = line
    return num_line_transfers

def categorize_station_pairs_by_transfers(graph, stations, connections_filename):
    categories = {
        'same_line': [],
        'adjacent_transfer': [],
        'multiple_transfers': []
    }

    for start_id in stations:
        for end_id in stations:
            if start_id != end_id:
                num_transfers = count_line(graph, start_id, end_id, connections_filename)

                if num_transfers == 0:
                    categories['same_line'].append((start_id, end_id))
                elif num_transfers == 1:
                    categories['adjacent_transfer'].append((start_id, end_id))
                elif num_transfers > 1:
                    categories['multiple_transfers'].append((start_id, end_id))

    return categories
```

Observations

Stations on the Same Line:

- *A Often Faster*: A* generally shows better performance (faster times) for stations on the same line. This improved efficiency is attributed to the heuristic effectively guiding the search. Since geographic proximity typically correlates closely with the travel path on the same subway line, the heuristic helps A* prioritize more direct paths over less optimal ones. By focusing search efforts where they are most likely to yield quick results, A* reduces the computational overhead compared to Dijkstra's algorithm, which does not utilize heuristic guidance and must methodically explore all possible paths.

Adjacent Line Transfers:

- **Competitive Performance**: For station pairs like Aldgate to Edgware Road and Acton Town to Amersham, which involve just one line transfer, A* demonstrates slight improvements or comparable performance to Dijkstra. This observation suggests that the heuristic provides a marginal benefit in these scenarios by potentially guiding the search towards more effective transfer points. The heuristic's effectiveness in these cases, though not as significant as in single-line scenarios, still helps in reducing the search space and focusing on more promising paths, thereby enhancing the efficiency of the search process.

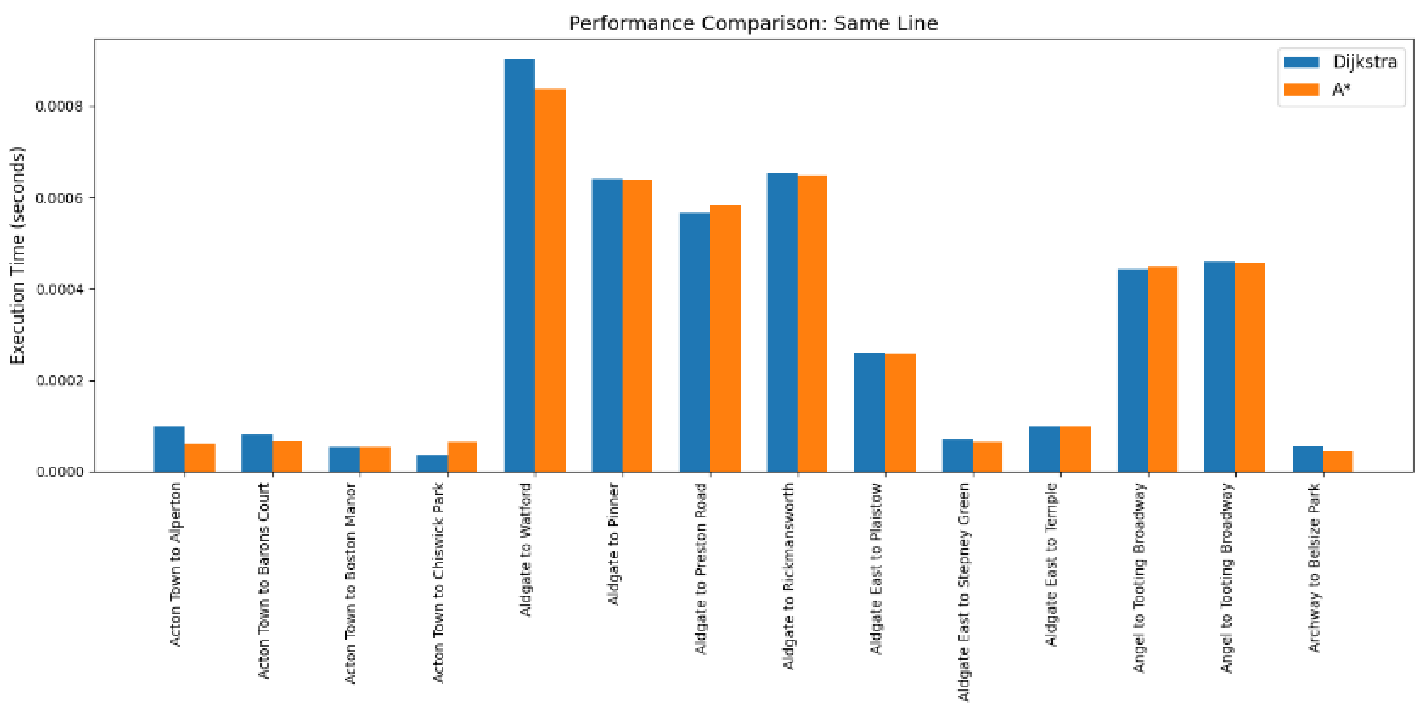
Multiple Line Transfers:

- **Mixed Results**: In scenarios involving multiple line transfers, such as Acton Town to Aldgate and Seven Sisters to Mornington Crescent, the performance results are mixed. A* may be slower or faster depending on the specific route and configuration of the subway network. This variability highlights the dependence of A*'s performance on the effectiveness of the heuristic in navigating complex decision trees involving multiple transfers. When the heuristic aligns well with efficient transfer strategies, A* can outperform Dijkstra's. However, if the heuristic misguides the search or if the route complexity overburdens the heuristic's simplicity, A* may not achieve a noticeable advantage or could even underperform compared to Dijkstra's exhaustive approach.

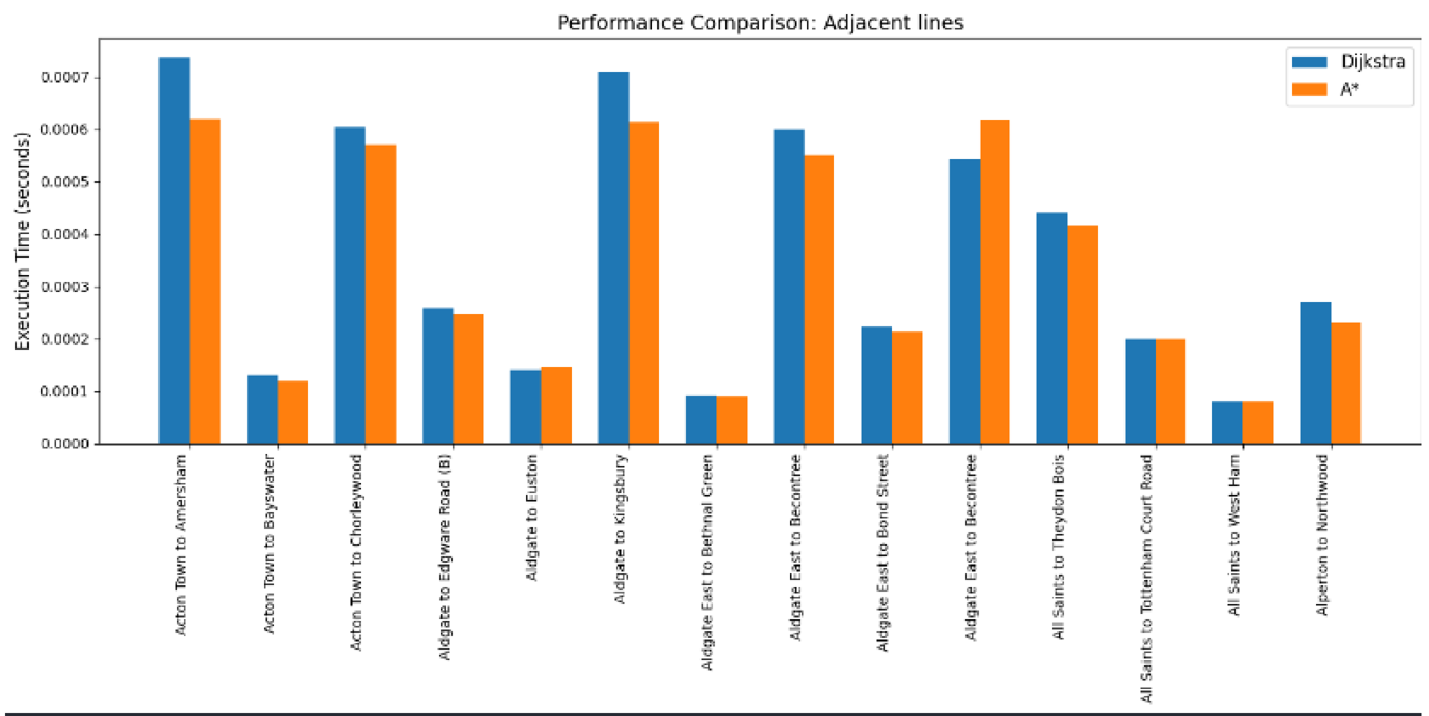
```
#Experiments

same_pairs = [
    (1,5),
    (1,17),
    (1,30),
    (1,52),
    (2,280),
    (2, 199),
    (2,202),
    (2, 214),
    (3, 200),
    (3, 244),
    (3, 255),
    (7, 258),
    (7, 258),
    (8,22)

]

adjacent_pairs = [
    (1,6),
    (1,18),
    (1,53),
    (2,82),
    (2,89),
    (2,144),
    (3, 24),
    (3, 21),
    (3, 28),
    (3, 21),
    (4, 256),
    (4, 259),
    (4, 289),
    (5, 179),

]

multi_pairs = [
    (1,2),
    (1,10),
    (1, 39),
    (2, 8),
    (2, 15),
    (2, 16),
    (2, 17),
    (2, 40),
    (223, 21),
    (223, 27),
    (224, 170),
    (224, 172),
    (224, 183),
    (225, 98),


]


# No transfer (Same line)
labels_same_line = []
dijkstra_times_same_line = []
astar_times_same_line = []
station_pairs_adjacent_lines = []
```

```
print("Time Comparison between Dijkstra's and A* Algorithms: same line")
print("----------------------------------------------------------------------------------------------------
for start_id, end_id in same_pairs:

    dijkstra_time1,_ = measure_performance(graph, start_id, end_id, lambda x, y: 0)
    astar_time1,_ = measure_performance(graph, start_id, end_id, heuristic)
    labels_same_line.append(f"{stations[start_id]['name']} to {stations[end_id]['name']}")
    dijkstra_times_same_line.append(dijkstra_time1)
    astar_times_same_line.append(astar_time1)
    print(f"Pair: {labels_same_line[-1]}, Dijkstra Time: {dijkstra_time1:.6f}, A* Time: {astar_t
print("----------------------------------------------------------------------------------------------------

plot_performance_comparison(labels_same_line, dijkstra_times_same_line, astar_times_same_line, '

# One transfer (Adjacent lines)
dijkstra_times_adjacent_lines = []
astar_times_adjacent_lines = []
labels_adjacent_lines = []
print("Time Comparison between Dijkstra's and A* Algorithms: Ajacent Line transfers")
print("----------------------------------------------------------------------------------------------------
for start_id, end_id in adjacent_pairs:

    dijkstra_time3,_ = measure_performance(graph, start_id, end_id, lambda x, y: 0)
    astar_time3,_ = measure_performance(graph, start_id, end_id, heuristic)
    labels_adjacent_lines.append(f"{stations[start_id]['name']} to {stations[end_id]['name']}")
    dijkstra_times_adjacent_lines.append(dijkstra_time3)
    astar_times_adjacent_lines.append(astar_time3)
    print(f"Pair: {labels_adjacent_lines[-1]}, Dijkstra Time: {dijkstra_time3:.6f}, A* Time: {as

print("----------------------------------------------------------------------------------------------------

plot_performance_comparison(labels_adjacent_lines, dijkstra_times_adjacent_lines, astar_times_ad


# Multiple transfers (Multiple lines)
dijkstra_times_multiple_transfers = []
astar_times_multiple_transfers = []
labels_multiple_transfers = []
print("Time Comparison between Dijkstra's and A* Algorithms: Multiple Line transfers")
print("----------------------------------------------------------------------------------------------------
for start_id, end_id in multi_pairs:

    dijkstra_time2,_ = measure_performance(graph, start_id, end_id, lambda x, y: 0)
    astar_time2,_ = measure_performance(graph, start_id, end_id, heuristic)
    labels_multiple_transfers.append(f"{stations[start_id]['name']} to {stations[end_id]['name']
    dijkstra_times_multiple_transfers.append(dijkstra_time2)
    astar_times_multiple_transfers.append(astar_time2)


    print(f"Pair: {labels_multiple_transfers[-1]}, Dijkstra Time: {dijkstra_time2:.6f}, A* Time
print("----------------------------------------------------------------------------------------------------

plot_performance_comparison(labels_multiple_transfers, dijkstra_times_multiple_transfers, astar_

# Testing the count_line function
#whether the function is working correctly or not
for i in range(len(same_pairs)): #Zero transfers
```

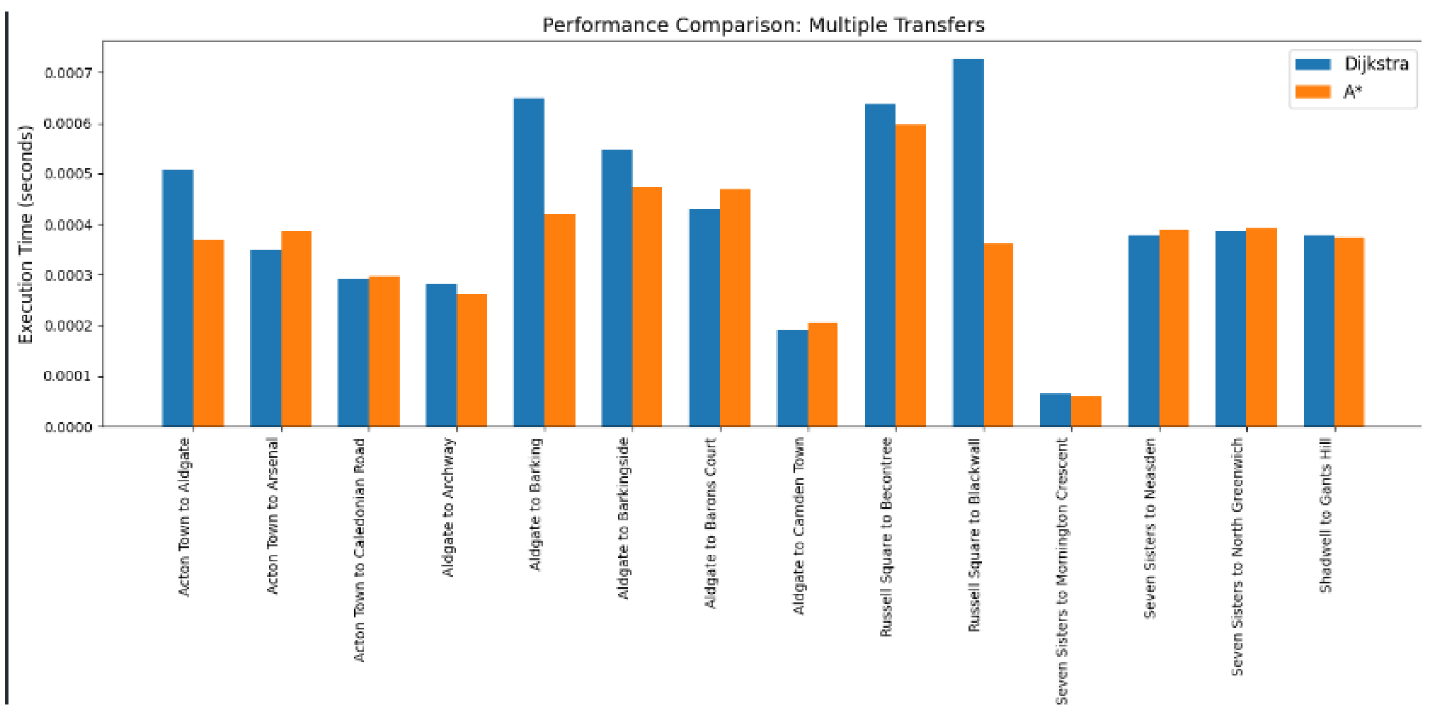
```
    print(count_line(graph, same_pairs[i][0], same_pairs[i][1], 'london_connections.csv'))
print('---------------------------------------------------------------------------------
for i in range(len(adjacent_pairs)): #One transfer
    print(count_line(graph, adjacent_pairs[i][0], adjacent_pairs[i][1], 'london_connections.csv
print('---------------------------------------------------------------------------------
for i in range(len(multi_pairs)): #Multiple transfers
    print(count_line(graph, multi_pairs[i][0], multi_pairs[i][1], 'london_connections.csv'))
print('---------------------------------------------------------------------------------
```



Same line (Dijkstra vs A*)



Adjacent Lines (Dijkstra vs A*)



Multiple Transfer (Dijkstra vs A*)

Ques 3: Using the "line" information provided in the dataset, compute how many lines the shortest path uses in your results/discussion?

Ans 3:

The line change data provided indicates the number of times a traveler would need to switch lines when taking the shortest path between specific pairs of stations. So, we iteratively find all pairs of vertices and edges using Dijkstra's and A* algorithm, and append the pair to our list. Working through our consecutive pairs of shortest path, it correspondingly checks if there happens to be a line connecting them in the network. It then returns the count of these distinct lines travelled through. These counts helps us analyze the connectivity and transfer efficiently within the London Subway network:

- **Zero Line Changes**: For pairs like Acton Town to South Ealing and Baker Street to Regent's Park, where no line changes are required, the shortest path uses only one line. This is the ideal scenario for passengers, as it ensures a seamless journey without the need to switch trains.
- **Moderate Line Changes (1-3)**: Pairs such as Arsenal to Latimer Road (3 line changes), Bank to Woodford (2 line changes), and Chesham to Plaistow (1 line change) illustrate paths where passengers must make one to three transfers. Each line change potentially introduces a point of friction, extending total travel time and complicating the journey.
- **High Line Changes (8)**: An extreme example is the route from Epping to Wimbledon, requiring eight line changes. This path represents a highly complex journey, likely impacting passenger convenience and indicating significant gaps in direct connectivity across the network.

Analysis:

To compute and validate these line changes, the `count_line_changes` function systematically processes each segment of the path to detect when a switch to a different line is necessary. This is how the function operates:

1. **Map Line Connections**: The function builds a map of station pairs to possible lines, allowing quick lookup of which lines can be taken between any two consecutive stations on a path.
2. **Track Line Transitions**: As the journey progresses from station to station, the function checks if the current line (if any) continues to the next station. If not, it identifies the new line to be taken and counts this as a line change.
3. **Aggregate Results**: The total number of line changes is then aggregated for the entire path, giving a clear picture of how many different subway lines a traveler would use.

```
#Question 3 Line changes, I took some random pairs, and made line change function below, inorde
station_pairs = [
    (1, 234),  # Short distance
    (10, 150),  # Medium distance
    (50, 200),  # Long distance
    (11, 212),  # Same Line, Short Distance
    (13, 301),  # Same Line, Long Distance
    (11, 87),   # Different Lines, No Transfers
    (3, 295),   # Different Lines, Multiple Transfers
    (117, 42),  # Heathrow Terminals 1, 2 & 3 to Canary Wharf
    (282, 247), # Wembley Park to Stratford
    (88, 299),  # Epping to Wimbledon
    (35, 192),  # Brixton to Oxford Circus
    (280, 167),  # Watford to Moorgate
    (6,70)

]


def count_line_changes(path, connections):
    station_pairs_to_lines = {}
    for conn in connections:
        station1, station2, line, _ = conn
        if (station1, station2) not in station_pairs_to_lines:
            station_pairs_to_lines[(station1, station2)] = set()
        station_pairs_to_lines[(station1, station2)].add(line)

        if (station2, station1) not in station_pairs_to_lines:
            station_pairs_to_lines[(station2, station1)] = set()
        station_pairs_to_lines[(station2, station1)].add(line)

    line_changes = 0
    current_line = None
    for i in range(len(path) - 1):
        station1, station2 = path[i], path[i + 1]
        possible_lines = station_pairs_to_lines.get((station1, station2), set())


        if current_line not in possible_lines:

            for line in possible_lines:
                current_line = line
                break

            if i > 0:
                line_changes += 1

    return line_changes

labels = []
for start_id, end_id in station_pairs:
    labels.append(f"{stations[start_id]['name']} to {stations[end_id]['name']}")
line_changes_list = []
print("Line Change Comparison between Dijkstra's and A* Algorithms:")
print("------------------------------------------------------------------------------------
for (start_id, end_id), label in zip(station_pairs, labels):
    _,dijkstra_path = dijkstra(graph, start_id, end_id)
    _, astar_path = A_Star(graph, start_id, end_id, heuristic)


    dijkstra_line_changes = count_line_changes(dijkstra_path, connections)
    astar_line_changes = count_line_changes(astar_path, connections)

    line_changes_list.append((dijkstra_line_changes, astar_line_changes))

    print(f"Pair: {label}, Dijkstra Line Changes: {dijkstra_line_changes}, A* Line Changes: {ast
#Output
"""
Pair: Acton Town to South Ealing, Dijkstra Line Changes: 0, A* Line Changes: 0
Pair: Arsenal to Latimer Road, Dijkstra Line Changes: 3, A* Line Changes: 3
Pair: Chesham to Plaistow, Dijkstra Line Changes: 1, A* Line Changes: 1
Pair: Baker Street to Regent's Park, Dijkstra Line Changes: 0, A* Line Changes: 0
Pair: Bank to Woodford, Dijkstra Line Changes: 2, A* Line Changes: 2
Pair: Baker Street to Embankment, Dijkstra Line Changes: 2, A* Line Changes: 2
Pair: Aldgate East to Whitechapel, Dijkstra Line Changes: 0, A* Line Changes: 0
Pair: Heathrow Terminals 1, 2 & 3 to Canary Wharf, Dijkstra Line Changes: 2, A* Line Changes: 2
Pair: Wembley Park to Stratford, Dijkstra Line Changes: 2, A* Line Changes: 2
Pair: Epping to Wimbledon, Dijkstra Line Changes: 8, A* Line Changes: 8
Pair: Brixton to Oxford Circus, Dijkstra Line Changes: 0, A* Line Changes: 0
Pair: Watford to Moorgate, Dijkstra Line Changes: 0, A* Line Changes: 0
Pair: Amersham to Devons Road, Dijkstra Line Changes: 3, A* Line Changes: 3"""
```

Conclusion

In summary, the A* algorithm generally outperforms Dijkstra's algorithm in navigating the London Subway system due to its heuristic function. This heuristic effectively guides the search, focusing on the most promising paths and significantly reducing unnecessary calculations. This results in faster and more efficient route finding, especially in scenarios with direct paths or minimal complexity. The advantage of A*'s heuristic function is evident in both simpler routes and more complex scenarios involving transfers, highlighting its effectiveness in optimizing pathfinding in a dense urban transit network.

Part - 5

Introduction

In this report, we explore the structuring and implementation of graph theory algorithms using object-oriented design principles, particularly focusing on a Unified Modeling Language (UML) diagram. The discussion encompasses the use of design patterns such as the Adapter pattern for integrating the A* algorithm, and considerations for enhancing the flexibility of the graph models to handle diverse types of node identifiers and attributes. Additionally, I will discuss potential extensions of the `Graph` abstraction to accommodate different graph types and their practical implementations.

This report will address the following:

1. **Use of Design Patterns**: Specifically, how the Adapter pattern is applied to integrate the A* algorithm within our existing graph framework.
2. **Graph Model Flexibility**: Discuss modifications to the UML diagram to support various node types beyond integers, such as strings or objects containing multiple attributes.
3. **Graph Types**: Explore potential additional implementations of the `Graph` interface to accommodate different kinds of graph structures and use cases.

Functions made:

- **Graph and its Subclasses**: The `Graph` class defines the basic structure of a graph, with methods for adding nodes and edges. The subclasses `WeightedGraph` and `HeuristicGraph` extend this functionality to accommodate weighted edges and heuristic values, respectively.
- **Shortest Path Algorithms**: The abstract class `SPAlgorithm` defines a common interface for various shortest path algorithms. The concrete implementations `Dijkstra`, `A_Star_Adapter`, and `BellmanFord` represent different algorithms for finding the shortest path.
- **MinPriorityQueue**: This class provides a priority queue data structure, primarily used by the A* and Dijkstra algorithms to efficiently select the next node to explore based on its priority.

- **ShortPathFinder**: This class serves as a high-level interface for finding the shortest path in a graph using a specified algorithm. It encapsulates the graph and algorithm objects and provides methods to set them and compute the shortest path.

Questions

Ques 1: Discuss what design principles and patterns are being used in the diagram.

Ans 1: As mentioned in the description of Part 5, instead of re-writing the A* algorithm for this particular part, we can treat the class from UML as an **Adapter**. **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate. It helps in conversion of one object, so that another object can understand it. It helps in conversion of To conceal the intricate conversion process taking place in the background, an **adaptor** is wrapped around one of the objects. Our design uses abstract base classes (`Graph` and `SPAlgorithm`) that other classes implement or extend. Hence, ensuring that our system components depend on abstractions rather than concrete implementations, giving us flexibility.

Structural Pattern : Adapter

In our code, and as discussed above, `A_Star_Adapter` serves as an adapter, making the external A* algorithm compatible with our internal representations without requiring changes to be made to the algorithm itself. The class `SPAlgorithm` acts as an interface which inherits our adapter. Composite object like `Graph` can be extended to `WeightedGraph` and `HeuristicGraph`, each adding additional behaviors while complying to their base class structure.

```
class A_Star_Adapter(SPAlgorithm):
    def calc_sp(self, heuristic_graph: HeuristicGraph, source: int, dest: int) -> float:
        graph = {node: {} for node in heuristic_graph.nodes}
        for node, neighbors in heuristic_graph.edges.items():
            for neighbor, weight in neighbors.items():
                graph[node][neighbor] = weight

        heuristic = {node: heuristic_graph.get_heuristic(node) for node in heuristic_graph.nodes

        _, cost = self.a_star_algorithm(graph, source, dest, heuristic)

        return cost

    def a_star_algorithm(self, graph, source, destination, heuristic):
        open_list = MinPriorityQueue()
        open_list.put(source, heuristic[source])
        predecessors = {source: None}
        costs = {source: 0}

        while not open_list.is_empty():
            _, current = open_list.delete_min()

            if current == destination:
                break

            for neighbor, weight in graph[current].items():
                new_cost = costs[current] + weight
                if neighbor not in costs or new_cost < costs[neighbor]:
                    costs[neighbor] = new_cost
                    priority = new_cost + heuristic[neighbor]
                    open_list.put(neighbor, priority)
                    predecessors[neighbor] = current

        if destination not in costs:
            return [], float('inf')

        return self.reconstruct_path(predecessors, source, destination), costs[destination]
    def reconstruct_path(self, predecessors, start, end):
        path = []
        current = end
        while current != start and current is not None:
            path.append(current)
            current = predecessors.get(current)
        if current is None:
            return [], float('inf')
        path.append(start)
        path.reverse()
        return path
```

Ques 2: The UML is limited in the sense that graph nodes are represented by the integers. How would you alter the UML diagram to accommodate various needs such as nodes being represented Strings or carrying more information than their names.? Explain how you would change the design in Figure 2 to be robust to these potential changes.

Ans 2: As mentioned in the question, currently the nodes are represented as integers, but to accommodate various needs such as nodes being represented as strings or carrying more information than their needs, we will need to make changes to our design to incorporate the changes. Few of the changes are as below, to fulfill the requirements of the question:

- **Abstract Node Type**: We can define a base `Node` class or interface with properties that all nodes must have. Specific node types can then inherit from this base class, allowing them to gain custom attributes and methods specific to different kinds of nodes. For instance, a `NodeFactory` class could have a method `createNode(type, properties)` which, based on the type, instantiates different subclasses of `Node`.

- **Factory Pattern for Node Creation**: We can utilize a factory pattern to abstract the creation process of different types of nodes. This can help in managing the instantiation of various node types based on runtime conditions, making the system more adaptable and easier to manage. The factory ensures that all nodes are created following a consistent process, which can include validation checks, default settings, and proper initialization. This is crucial for maintaining the integrity of the graph's data.

Ques 3: Discuss what other types of graphs we could have implement "Graph". What other

implementations exist?

Ans 3: Since our "Graph" is very flexible as described in our UML, it gives us malleability to design different specific kind of graphs to fulfill the needs and requirements. The other types of graphs we could implemented for "Graph" are as follows:

- **Directed Graph:** A directed graph, or digraph, is a graph where edges have a direction associated with them, indicating what vertex is the start and what vertex is the end. In this graph, edges are represented as ordered pairs of vertices. To implement this, we would have to add an attribute to the `Graph` class to differentiate between directed and undirected graphs. In directed graphs, the `add_edge` method would only add an edge from `src` to `dst` and not vice versa, reflecting the one-way relationship.
- **Weighted Graph:** Weighted graphs assign a cost or weight to each edge, which can assist with routing algorithms where distances or costs between nodes are needed. This is designed with our `WeightedGraph` class where the `add_edge` method also takes a `weight` parameter, and edges are stored with their corresponding weights.
- **Tree:** A tree **is an undirected graph G** that satisfies any of the following equivalent conditions: G is connected and is acyclic. So to implement this, we can add methods to ensure that each node (except the root) has exactly one parent and methods to traverse the tree efficiently using BFS and DFS.
- **Additional Implementation**: We can ensure methods for `add_node`, `add_edge`, `remove_node`, and `remove_edge` are robust and handle all updates to the graph structure efficiently.

Each of these implementations would inherit from the abstract `Graph` class defined in your UML. These are some of the implementations of other types of graphs for "Graph".

Appendix (Code Navigation)

Dijkstra's algorithm

The `MinPriorityQueue` Class

- **Key Data Structure:**
  - `heap`: A list-based representation of a binary heap.
- **Methods:**
  - `__init__(self)`: Constructor to initialize an empty priority queue.
  - `parent(self, i)`: Returns the index of the parent node of the node at index `i`.
  - `empty(self)`: Returns `True` if the priority queue is empty, otherwise `False`.
  - `left_child(self, i)`: Returns the index of the left child node of the node at index `i`.
  - `right_child(self, i)`: Returns the index of the right child node of the node at index `i`.
  - `insert(self, val)`: Inserts `val` into the priority queue, maintaining the heap property.
    - Appends the value to the heap.
    - Calls `_heapify_up` to restore the heap property by moving the new element upwards if needed.
  - `delete_min(self)`: Removes and returns the element with the minimum value (highest priority) from the priority queue.
    - Checks if the heap is empty.
    - Swaps the first (min) and last element.
    - Removes the last element (the former minimum).
    - Calls `_heapify_down` to restore the heap property by moving the swapped element downwards.
  - `_heapify_up(self, index)`: Restores the heap property (from the bottom up) after an insertion. Compares the node at `index` with its parent and swaps them if the parent is larger.
  - `_heapify_down(self, index)`: Restores the heap property (from the top down) after a deletion. Compares the node at `index` with its children and swaps it with the smaller child if necessary.

The `Graph` Class

This represents a graph data structure, which can be either directed or undirected.

- **Key Data Structures**
  - `adj_matrix`: A 2D matrix representing the adjacency relationships between vertices. If `adj_matrix[u][v]` has a non-zero weight, that signifies an edge exists from vertex `u` to vertex `v` with that weight.
  - `size`: The number of vertices in the graph.
  - `vertex_data`: A list to store optional data associated with each vertex.
  - `directed`: Boolean flag to indicate whether the graph is directed or undirected.

Methods

- `__init__(self, size, directed=False)`
  - Constructor to initialize the graph.
  - `size`: The number of vertices in the graph.
  - `directed`: (Optional) Specifies if the graph is directed (True) or undirected (False).
- `add_edge(self, u, v, weight)`
  - Adds an edge between vertices `u` and `v` with the specified `weight`.
  - For undirected graphs, adds the edge in both directions.
- `add_vertex_data(self, vertex, data)`
  - Associates `data` with the specified `vertex` index.
- `num_edges(self)`

    - Returns the total number of edges in the graph.
- `num_vertices(self)`
    - Returns the total number of vertices in the graph.
- `dijkstra(self, start_vertex_data, k)`:
    - `start_vertex_data`: The data associated with the starting vertex.
    - `k`: The maximum number of edges in the shortest paths to consider.
    - Initialize distances to infinity for all vertices, except the start vertex set to 0.Initialize a `visited` array to keep track of visited vertices.
    - Initialize a `paths` dictionary to track shortest paths to each vertex.
    - Initialize a `counts` array to track the number of times a vertex is considered.
    - Create a `MinPriorityQueue`. Insert the starting vertex with a distance of 0.
    - **Loop:**
        - While the priority queue is not empty:
            - Remove the vertex (`u`) with the minimum distance.
            - If `u` has been visited `k` or more times, skip it; this limits path length.
            - Mark `u` as visited and increment its count in `counts`.
            - For each unvisited neighbor `v` of `u`:
                - Calculate the potential distance to `v` through `u`.
                - If the new distance is smaller, update the distance in `distances`, update the path in `paths`, and add `v` to the priority queue with the new priority (distance).
    - **Returns:** `distances` (list of shortest distances to each vertex), and `paths` (dictionary of shortest paths).

How to Create & Use a Graph

1. **Create a `Graph` object:**

```
my_graph = Graph(5, directed=True)  # A directed graph with 5 vertices
```

2. **Add edges:**

```
my_graph.add_edge(0, 1, 5)  # Edge from vertex 0 to 1 with weight 5
my_graph.add_edge(2, 4, 2)
# ...add more edges
```

3. **Add optional vertex data:**

```
my_graph.add_vertex_data(0, "City A")
my_graph.add_vertex_data(1, "City B")
# ...add data for other vertices
```

Bellman Ford's algorithm

Methods

- ... the graph methods are same as before.
- `bellman_ford(self, start_vertex_data, k)`
    - Implements the Bellman-Ford algorithm to find the shortest distances from a starting vertex to all other vertices in the graph. Also finds the shortest paths.
    - `start_vertex_data`: The data associated with the starting vertex.
    - `k`: The maximum number of edges in the shortest paths to consider.

- **Key Logic:**
    - Initializes an array `distances` to track shortest distances to each vertex.
    - Initializes a dictionary `paths` to track shortest paths to each vertex.
    - Iteratively relaxes edges: If adding an edge shortens the distance to a vertex, update the distance and path.
    - Performs negative cycle detection
- **Returns:** `distances` (a list of shortest distances to each vertex), and `paths` (a dictionary of shortest paths). Returns empty lists if a negative cycle exists.

Find shortest paths with Bellman-Ford:

```
distances, paths = my_graph.bellman_ford("City A", 3)
```

How to Use the Classes

1. **Creating a Graph:**

```
my_graph = generate_random_graph(8, 0.6, directed=True, negative_weights=True)
# Or create a graph manually by adding vertices and edges
```

2. **All-Pairs Shortest Paths (Positive edge weights):**

```
distances, paths, previous = all_pairs_dijkstra(my_graph)
print(distances[(2, 5)])  # Distance from vertex 2 to vertex 5
print(paths[(2, 5)])  # Shortest path from vertex 2 to vertex 5
```

3. **All-Pairs Shortest Paths (Negative edge weights):**

```
distances, paths, previous = all_pairs_bellman_ford(my_graph)
```

Part 2 Experiment

... the MinPriorityQueue class is the same as part 1

... the Graph class is also the same as part 1

- **`dijkstra_without_restriction` (within `Graph`)**
    - A standard implementation of Dijkstra's algorithm.
    - **Inputs:** graph - constructed from the graph class and a source: any vertex
    - **Returns:** `distances` (list of distances to each vertex), `paths` (dictionary of shortest paths)
- **`generate_random_graph`**
    - Helper function to create random graphs.
    - **Parameters:**
        - `size`: The number of vertices in the graph.
        - `density`: How densely connected the graph should be (0 = no edges, 1 = fully connected)
        - `directed`: Whether the graph should be directed.
        - `negative_weights`: Whether to allow negative edge weights.
- **`all_pairs_dijkstra`**
    - Computes all-pairs shortest paths with positive edge weights using Dijkstra's algorithm.
    - Loops through each vertex as a potential source, calls `dijkstra_without_limit`.
    - **Returns:**
        - `distances`: A dictionary where keys are (source, target) vertex pairs and values are shortest distances.

        - `shortest_paths`: A dictionary where keys are (source, target) and values are the actual shortest paths.
        - `previous`: A dictionary where keys are (source, target) pairs and values are the second-to-last vertex on the shortest path.

- **`all_pairs_bellman_ford`**
    - Computes all-pairs shortest paths with negative edge weights using the Bellman-Ford algorithm.
    - Similar logic to `all_pairs_dijkstra`, but calls `bellman_ford_without_restriction`.
    - **Returns:** (same as `all_pairs_dijkstra`)

Code Navigation

- **Main Execution Flow:**
    1. A `Graph` is generated (either randomly or manually).
    2. Either `all_pairs_dijkstra` or `all_pairs_bellman_ford` is called, depending on if the graph has negative weights.
    3. These functions repeatedly execute either `dijkstra_without_restriction` or `bellman_ford_without_restriction` respectively, for each vertex as a source.

A* Algorithm (According to the prof implementation in the slides)

Introduction

This report discusses the implementation of the A* algorithm to find the shortest path between two nodes in a graph. The A* algorithm is a popular choice for pathfinding and graph traversals due to its efficiency and accuracy. It combines features of Dijkstra's algorithm and a heuristic function to optimize pathfinding.

Implementation Details:

1. **MinPriorityQueue Class**

In developing the `MinPriorityQueue` class, we aimed to efficiently manage the nodes during the graph traversal process, which is crucial for both Dijkstra's and A* algorithms. Here's how I implemented the various components of this class:

- **Initialization (`__init__`):**
    - We start by initializing the `heap` as an empty list, which will store the elements in a heap structure. This setup ensures efficient priority-based access and removal of elements.
- **Helper Methods**:
    - `parent(self, i)`:
        - We created this method to calculate the parent's index of a node in the heap, which is essential for maintaining the heap structure during insertions and deletions.
    - `left_child(self, i)`:
        - This method determines the left child's index of a given node, aiding in the traversal of the heap during reordering.
    - `right_child(self, i)`:
        - Similarly, this method finds the right child's index, which is necessary for the same reasons as the left child calculation.
    - `insert(self, val)`:
        - When adding a new element `val` to the heap, I append it to the end of the list to maintain a complete tree structure.
        - I then call `_heapify_up` to adjust the heap from bottom to top, ensuring the minimum value remains at the root, preserving the heap property.
    - `delete_min(self)`:

        - To remove the minimum element, which is the root of the heap, I swap it with the last element and then pop the last element from the list. This method preserves the heap structure while efficiently removing the root.
        - I follow this by calling `_heapify_down` to adjust the new root downward, reestablishing the heap property.
    - `_heapify_up(self, index)`:
        - I designed this method to move the element at the specified `index` up the heap. It continues moving up until it is no longer less than its parent, maintaining the heap order.
    - `_heapify_down(self, index)`:
        - This method moves the element at `index` down the heap, swapping it with its smallest child until it properly adheres to the heap properties, either by settling into a position where it's smaller than its children or becoming a leaf.
    - `put(self, node, priority)`:
        - We created this method as a wrapper for `insert`, handling the input as a tuple of `priority` and `node`. This adaptation facilitates specific priority queue operations needed in pathfinding algorithms.
    - `is_empty(self)`:
        - This utility method checks if the heap is empty, which is crucial for controlling the loop conditions in the pathfinding algorithms.

2. **A* Algorithm Function**

   In implementing the A* algorithm, my goal was to enhance the efficiency of pathfinding by incorporating a heuristic:

   - **Initialization**:
     - I initialize a `MinPriorityQueue` to manage nodes that need to be evaluated.
     - I start with the source node, assigning it a priority calculated by adding 0 (cost from start) to the heuristic estimate to the destination.
   - **Main Loop**:
     - The main loop continues until the priority queue is empty.
     - We remove the node with the highest priority and evaluate its connections. If the destination is reached, we break out of the loop.
     - For each neighbor of the current node, we update the cost if a cheaper path is found, calculated by adding the current path cost to the edge weight.
     - We then insert the neighbor into the priority queue with a new priority, comprising the updated cost plus the heuristic estimate to the destination.
   - **Path Reconstruction**:
     - Using the `reconstruct_path` method, I trace the path back from the destination to the source using the predecessors' dictionary. If no path is found, an empty list is returned.
   - **Return Values**:
     - The function returns a tuple containing the predecessors' dictionary, the reconstructed path, and the total cost to the destination.

3. **Conclusion**

   The implementation of the A* algorithm demonstrates its effectiveness in pathfinding by combining Dijkstra's algorithm's guarantee of the shortest path with the heuristic function's ability to speed up the search. This report provides a detailed overview of implementing the A* algorithm for route finding, illustrating the critical role of data structures like the priority queue and the importance of a heuristic function in optimizing pathfinding.

Part 4 Experiment (According to the prof implementation in the slides)

Introduction:

This report details the development of a pathfinding system for the London Underground network using both Dijkstra's and A* algorithms. The system calculates the shortest path between stations and evaluates the number of line changes required, optimizing for both time and convenience.

Priority Queue Implementation:

The `MinPriorityQueue` class is a fundamental component for the efficient execution of both Dijkstra's and A* algorithms. Here's a detailed breakdown of its implementation:

- **Initialization (`__init__`)**:
  - A Python list is used to implement the heap structure, starting empty.
- **Insertion (`insert`)**:
  - When a new element is added, it's appended to the end of the list (heap), maintaining the complete tree property.
  - The `_heapify_up` method is then called to restore the heap property by comparing the newly added element with its parent and swapping if necessary, continuing until the root is reached or no more swaps are needed.
- **Deletion (`delete_min`)**:
  - The root of the heap (minimum element) is swapped with the last element, and then the last element is removed. This preserves the complete tree structure.
  - The `_heapify_down` method is invoked to restore the heap property by swapping the new root with its smallest child until the heap property is satisfied throughout the tree.
- **Utility Methods**:
  - `is_empty` checks if the heap list is empty, which is crucial for the loop conditions in the pathfinding algorithms.
  - Internal methods like `parent`, `left_child`, and `right_child` are used to navigate the indices of the heap structure efficiently.

Graph Representation

The graph is represented using dictionaries to map station IDs to their neighbors along with associated weights (travel times or distances). Key functions in this setup include:

- **Parsing Stations and Connections:**
  - The `parse_stations` function reads station data from a CSV file and stores it in a dictionary, mapping station IDs to their corresponding attributes such as name, latitude, and longitude.
  - The `parse_connections` function processes another CSV file detailing connections between stations, including travel time and line ID for each connection. It outputs a list of tuples representing connections.
- **Building the Graph:**
  - The `build_graph` function utilizes the parsed station and connection data to construct a graph representation of the London Underground network.
  - Each station (node) in the graph has a dictionary of neighboring stations (edges) with distances calculated using the Euclidean formula.

Pathfinding Algorithms

1. **Dijkstra's Algorithm:**

- **Functionality:** Dijkstra's algorithm is used to find the shortest path between two stations on the London Underground network.
- **Implementation:**
  - It initializes dictionaries to track the minimum distance to each station (`distance`) and the predecessor stations (`predecessor`).
  - A priority queue (`MinPriorityQueue`) is used to explore stations in order of increasing distance.
  - For each station, it updates the distance to its neighbors if a shorter path is found.

    - Once the exploration is complete or the destination is reached, the shortest path is reconstructed in reverse from the destination to the start.

2. *A Algorithm:*
    - **Functionality:** A* algorithm is an extension of Dijkstra's algorithm that incorporates a heuristic function to guide the search towards the destination more efficiently.
    - **Implementation:**
        - Similar to Dijkstra's algorithm, it initializes dictionaries to track the predecessors and the cost from the start to each station.
        - It utilizes a priority queue (`MinPriorityQueue`) to explore stations based on a combined cost function, considering both the actual cost from the start and the heuristic estimate to the destination.
        - By incorporating the heuristic function, A* algorithm can potentially reduce the number of nodes explored compared to Dijkstra's algorithm.

Performance and Utility Functions

1. **Performance Measurement (`measure_performance`):**
    - This function times the execution of both Dijkstra's and A* algorithms for a given start and end station, providing empirical data on their efficiency.
2. **Path Reconstruction (`reconstruct_path`):**
    - A utility function used by both algorithms to trace back the shortest path from the destination to the start using the predecessor map.

Visualization and Analysis

- **Performance Comparison Visualization (`plot_performance_comparison`):**
    - This function plots the execution times of both algorithms across different pairs of stations, visually showcasing the relative efficiency of each algorithm.
    - It helps in understanding how the performance of Dijkstra's and A* algorithms varies for different station pairs.
- **Line Changes Analysis:**
    - The `count_line` function is used to determine the number of line changes required to travel between two stations.
    - By comparing the line changes between paths found by Dijkstra's and A* algorithms, the effectiveness of each algorithm in minimizing line changes can be assessed.
    - This analysis provides insights into the route optimization capabilities of both algorithms within the London Underground network.

Part 5 UML

```
from abc import ABC, abstractmethod
from typing import Dict, Tuple, List


class Graph(ABC):
    def __init__(self):
        self.edges = {}

    @property
    def nodes(self):
        return list(self.edges.keys())

    def add_node(self, node: int):
        if node not in self.edges:
            self.edges[node] = []
    def add_edge(self, src: int, dst: int):
        if src not in self.edges:
            self.add_node(src)
        if dst not in self.edges:
            self.add_node(dst)
        self.edges[src].append(dst)
        self.edges[dst].append(src)

    def get_num_of_nodes(self) -> int:
        return len(self.edges)

    def get_adj_nodes(self, node: int) -> List[int]:
        return self.edges.get(node, [])

    @abstractmethod
    def w(self, node: int) -> float:

        pass


class WeightedGraph(Graph):
    def __init__(self):
        super().__init__()
        self.edges = {}
    def add_edge(self, src: int, dst: int, weight: float):
        if src not in self.edges:
            self.edges[src] = {}
        if dst not in self.edges:
            self.edges[dst] = {}
        self.edges[src][dst] = weight
        self.edges[dst][src] = weight

    def w(self, node1: int, node2: int) -> float:
        try:
            return self.edges[node1][node2]
        except KeyError:
            return float('inf')



class HeuristicGraph(WeightedGraph):
    def __init__(self, _heuristic: Dict[int, float]):
        super().__init__()
        self._heuristic = _heuristic

    def get_heuristic(self, node) -> Dict[int, float]:
        return self._heuristic.get(node, float('inf'))

class SPAlgorithm(ABC):
    @abstractmethod
    def calc_sp(self, graph: Graph, source: int, dest: int) -> float:
        pass

class Dijkstra(SPAlgorithm):
    def calc_sp(self, graph: WeightedGraph, source: int, dest: int) -> float:
        distance = {vertex: float('infinity') for vertex in graph.nodes}
        distance[source] = 0
        pq = MinPriorityQueue()
        pq.put(source, 0)
        visited = set()
        predecessor = {vertex: None for vertex in graph.nodes}

        while not pq.is_empty():
            _, current_vertex = pq.delete_min()

            if current_vertex in visited:
                continue
            visited.add(current_vertex)

            if current_vertex == dest:
                break

            for neighbor in graph.get_adj_nodes(current_vertex):
                if neighbor in visited:
                    continue
                alt_route = distance[current_vertex] + graph.edges[current_vertex][neighbor]
                if alt_route < distance[neighbor]:
                    distance[neighbor] = alt_route
                    pq.put(neighbor, alt_route)
                    predecessor[neighbor] = current_vertex

        # Reconstruct path
        path = []
        current = dest
        while current is not None and current in predecessor:
            path.append(current)
            current = predecessor[current]
        path.reverse()

        return distance[dest] if distance[dest] != float('infinity') else float('inf')


class A_Star_Adapter(SPAlgorithm):
    def calc_sp(self, heuristic_graph: HeuristicGraph, source: int, dest: int) -> float:
        graph = {node: {} for node in heuristic_graph.nodes}
        for node, neighbors in heuristic_graph.edges.items():
            for neighbor, weight in neighbors.items():
                graph[node][neighbor] = weight

        heuristic = {node: heuristic_graph.get_heuristic(node) for node in heuristic_graph.nodes

        _, cost = self.a_star_algorithm(graph, source, dest, heuristic)

        return cost

    def a_star_algorithm(self, graph, source, destination, heuristic):
        open_list = MinPriorityQueue()
        open_list.put(source, heuristic[source])
        predecessors = {source: None}
        costs = {source: 0}

        while not open_list.is_empty():
            _, current = open_list.delete_min()

            if current == destination:
                break

            for neighbor, weight in graph[current].items():
                new_cost = costs[current] + weight
                if neighbor not in costs or new_cost < costs[neighbor]:
                    costs[neighbor] = new_cost
                    priority = new_cost + heuristic[neighbor]
                    open_list.put(neighbor, priority)
                    predecessors[neighbor] = current

        if destination not in costs:
            return [], float('inf')

        return self.reconstruct_path(predecessors, source, destination), costs[destination]

    def reconstruct_path(self, predecessors, start, end):
        path = []
        current = end
        while current != start and current is not None:
            path.append(current)
            current = predecessors.get(current)
        if current is None:
            return [], float('inf')
        path.append(start)
        path.reverse()
        return path




class BellmanFord(SPAlgorithm):
    def calc_sp(self, graph: WeightedGraph, source: int, dest: int) -> float:
        distances = {v: float('inf') for v in graph.nodes}
        predecessors = {v: None for v in graph.nodes}
        distances[source] = 0

        for _ in range(len(graph.nodes) - 1):
            for src in graph.edges:
                for dst, weight in graph.edges[src].items():
                    if distances[src] + weight < distances[dst]:
                        distances[dst] = distances[src] + weight
                        predecessors[dst] = src

        for src in graph.edges:
            for dst, weight in graph.edges[src].items():
                if distances[src] + weight < distances[dst]:
                    raise ValueError("Graph contains a negative-weight cycle")

        path = []
        current = dest
        while current is not None:
            path.append(current)
            current = predecessors[current]
        path.reverse()

        return distances[dest] if distances[dest] != float('inf') else None
```

```
# For A* and Dijkstra
class MinPriorityQueue:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def insert(self, val):
        self.heap.append(val)
        self._heapify_up(len(self.heap) - 1)

    def delete_min(self):
        if len(self.heap) == 0:
            return None

        self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]
        min_val = self.heap.pop()
        self._heapify_down(0)
        return min_val

    def _heapify_up(self, index):
        while index > 0 and self.heap[self.parent(index)] > self.heap[index]:
            self.heap[index], self.heap[self.parent(index)] = self.heap[self.parent(index)], sel
            index = self.parent(index)

    def _heapify_down(self, index):
        while index < len(self.heap):
            smallest = index
            left = self.left_child(index)
            right = self.right_child(index)

            if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
                smallest = left
            if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
                smallest = right

            if smallest != index:
                self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
                index = smallest
            else:
                break


    def put(self, node, priority):
        self.insert((priority, node))

    def is_empty(self):
        return len(self.heap) == 0


class ShortPathFinder:
    def __init__(self, graph: Graph, algo: SPAlgorithm):
        self.graph = graph
        self.algo = algo

    def calc_short_path(self, source: int, dest: int) -> float:
        return self.algo.calc_sp(self.graph, source, dest)


    def set_graph(self, graph: Graph):
        self.graph = graph
        self.algo.graph = graph

    def set_algorithm(self, algo: SPAlgorithm):
        self.algo = algo
```

Test Cases:

```
"""Detailed testing to ensure that everything is working or not"""
def create_test_graph():
    graph = WeightedGraph()
    graph.add_edge(1, 2, 1.0)
    graph.add_edge(2, 3, 2.0)
    graph.add_edge(3, 4, 3.0)
    graph.add_edge(4, 5, 4.0)
    graph.add_edge(1, 5, 10.0)
    return graph

def create_heuristic_graph():
    heuristic = {1: 9, 2: 7, 3: 4, 4: 2, 5: 0}
    graph = HeuristicGraph(heuristic)
    graph.add_edge(1, 2, 1.0)
    graph.add_edge(2, 3, 2.0)
    graph.add_edge(3, 4, 3.0)
    graph.add_edge(4, 5, 4.0)
    graph.add_edge(1, 5, 10.0)
    return graph

def test_dijkstra_simple_path():
    graph = create_test_graph()
    dijkstra_algo = Dijkstra()
    finder = ShortPathFinder(graph, dijkstra_algo)
    cost = finder.calc_short_path(1, 5)
    print(cost)

def test_dijkstra_disconnected_graph():
    graph = WeightedGraph()
    graph.add_edge(1, 2, 1.0)
    graph.add_edge(3, 4, 1.0)
    dijkstra_algo = Dijkstra()
    finder = ShortPathFinder(graph, dijkstra_algo)
    cost = finder.calc_short_path(1, 4)
    print(cost)


def test_bellman_ford_negative_cycle():
    graph = WeightedGraph()
    graph.add_edge(1, 2, 4.0)
    graph.add_edge(2, 3, -6.0)
    graph.add_edge(3, 1, 2.0)
    bellman_ford_algo = BellmanFord()
    finder = ShortPathFinder(graph, bellman_ford_algo)
    try:
        cost = finder.calc_short_path(1, 3)
        assert False, "Negative cycle detection failed"
    except ValueError as e:
        assert str(e) == "Graph contains a negative-weight cycle", "Incorrect error message"
    print("Test Case 4 Passed: Bellman-Ford Negative Weight Cycle")
def test_a_star_adapter():
    # Test case 1
    graph1 = HeuristicGraph({0: 2, 1: 1, 2: 0})
    graph1.add_edge(0, 1, 1)
    graph1.add_edge(1, 2, 1)
    a_star1 = A_Star_Adapter()
    print(a_star1.calc_sp(graph1, 0, 2))

    # Test case 2
    graph2 = HeuristicGraph({0: 2, 1: 1, 2: 1, 3: 0})
    graph2.add_edge(0, 1, 1)
    graph2.add_edge(0, 2, 5)
    graph2.add_edge(1, 3, 1)
    graph2.add_edge(2, 3, 1)
    a_star2 = A_Star_Adapter()
    print(a_star2.calc_sp(graph2, 0, 3))

    # Test case 3
    graph3 = HeuristicGraph({0: 3, 1: 2, 2: 1, 3: 0})
    graph3.add_edge(0, 1, 1)
    graph3.add_edge(1, 2, 1)
    graph3.add_edge(2, 1, 1)
    graph3.add_edge(2, 3, 1)
    a_star3 = A_Star_Adapter()
    print(a_star3.calc_sp(graph3, 0, 3))

    # Test case 4
    graph4 = HeuristicGraph({0: 1, 1: 2, 2: 1, 3: 0})
    graph4.add_edge(0, 1, 1)
    graph4.add_edge(0, 2, 2)
    graph4.add_edge(0, 3, 3)
    a_star4 = A_Star_Adapter()
    print(a_star4.calc_sp(graph4, 0, 3))

def test_a_star_simple_weighted_graph():
    heuristic = {0: 3, 1: 2, 2: 1, 3: 0}
    graph = HeuristicGraph(heuristic)
    graph.add_edge(0, 1, 1)
    graph.add_edge(1, 2, 1)
    graph.add_edge(2, 3, 1)
    graph.add_edge(0, 3, 4)

    a_star_algo = A_Star_Adapter()
    finder = ShortPathFinder(graph, a_star_algo)

    cost = finder.calc_short_path(0, 3)
    print("Test A* Simple Weighted Graph:", "Passed" if cost == 3 else "Failed", "- Cost:", cos
def test_bellman_ford_no_negative_cycle():
    graph = WeightedGraph()
    graph.add_edge(0, 1, 5)
    graph.add_edge(1, 2, 3)
    graph.add_edge(2, 0, 2)

    bellman_ford_algo = BellmanFord()
    finder = ShortPathFinder(graph, bellman_ford_algo)

    cost = finder.calc_short_path(0, 2)
    print("Test Bellman-Ford No Negative Cycle:", "Passed" if cost == 8 else "Failed", "- Cost:'

def test_sp_algorithm_variety_graph_types():
    graph = WeightedGraph()
    graph.add_edge(0, 1, 2)
    graph.add_edge(1, 2, 2)

    # Dijkstra
    dijkstra_algo = Dijkstra()
    finder = ShortPathFinder(graph, dijkstra_algo)
    cost_dijkstra = finder.calc_short_path(0, 2)

    # Bellman-Ford
    bellman_ford_algo = BellmanFord()
    finder.set_algorithm(bellman_ford_algo)
    cost_bellman = finder.calc_short_path(0, 2)

    print("Test SPAlgorithm with Dijkstra and Bellman-Ford:", "Passed" if cost_dijkstra == cost_

test_dijkstra_simple_path()
test_dijkstra_disconnected_graph()
test_bellman_ford_negative_cycle()
test_a_star_adapter()
test_a_star_simple_weighted_graph()
test_bellman_ford_no_negative_cycle()
test_sp_algorithm_variety_graph_types()
```

Graph Classes

Abstract Base Graph

I started with an abstract base class `Graph` to define a general structure for graph implementations. The key components are:

- `__init__()`: Initializes the `edges` dictionary to store the connections between nodes.
- `nodes`: A property that lists all nodes in the graph.
- `add_node(node: int)`: Adds a new node to the graph if it's not already present.
- `add_edge(src: int, dst: int)`: Establishes a bidirectional (undirected) connection between two nodes.
- `get_num_of_nodes()`: Returns the number of nodes.
- `get_adj_nodes(node: int)`: Provides the list of adjacent nodes for a given node.
- `w(node: int)`: An abstract method that should return the weight of a node in derived classes. This method forces any subclass to implement how weights are handled.

Weighted Graph

I derived `WeightedGraph` from `Graph` to handle graphs where edges have weights:

- `__init__()`: Calls the base initializer and prepares the `edges` dictionary to store edge weights in addition to connections.
- `add_edge(src: int, dst: int, weight: float)`: Overrides the base class method to include the weight of the edge.
- `w(node1: int, node2: int)`: Returns the weight between two nodes, handling missing entries by returning infinity.

3. Heuristic Graph

For A* implementations, I created `HeuristicGraph`, extending `WeightedGraph` to integrate heuristics:

- `__init__(_heuristic: Dict[int, float])`: Besides the base initialization, it sets up a dictionary for heuristic values.
- `get_heuristic(node)`: Returns the heuristic value for a node, defaulting to infinity if the node is unknown.

Pathfinding Algorithms

1. **Abstract Base Algorithm**

   `SPAlgorithm` is an abstract class ensuring all pathfinding algorithms implement a `calc_sp()` method to calculate the shortest path.

2. **Dijkstra's Algorithm**

   Implemented as `Dijkstra`, it uses a priority queue to efficiently find the shortest paths:

   - `calc_sp(graph, source, dest)`: Initializes distances and predecessors, processes each node using the priority queue, and updates paths and distances as it explores the graph. It reconstructs the path from destination to source if reachable.

3. *A Algorithm*

   I adapted the A* algorithm within `A_Star_Adapter`:

   - `calc_sp(heuristic_graph, source, dest)`: Converts the heuristic graph to a standard format and calls the A* algorithm.
   - `a_star_algorithm(graph, source, destination, heuristic)`: Similar to Dijkstra but uses heuristics to prioritize the frontier. It also constructs the path if the destination is reached.

4. **Bellman-Ford Algorithm**

   Implemented as `BellmanFord`, it handles graphs with negative weights:

   - `calc_sp(graph, source, dest)`: Iteratively relaxes edges and checks for negative weight cycles. If a cycle is detected, it throws an error.

Utility Classes and Methods

1. **MinPriorityQueue**

   A custom priority queue implementation used by both Dijkstra's and A*:

   - `insert(val)` **and** `delete_min()`: Manage elements in the heap, ensuring the minimum element can always be accessed efficiently.
   - `_heapify_up()` **and** `_heapify_down()`: Maintain the heap property after insertion or deletion.

2. **ShortPathFinder**

   A facade that uses any `SPAlgorithm` to find paths in a `Graph`:

   - `calc_short_path(source, dest)`: Delegates to the `calc_sp()` method of the contained algorithm.
   - `set_graph(graph)` and `set_algorithm(algo)`: Allow switching of graphs and algorithms dynamically, enhancing flexibility.

Testing Functions

I included several functions to test the functionality and robustness of the algorithms with different graph configurations:

1. `create_test_graph()`:

- Creates a simple weighted graph with five nodes and edges of varying weights.

2. `create_heuristic_graph()`:
   - Creates a heuristic graph with heuristic values associated with each node, along with weighted edges.
3. `test_dijkstra_simple_path()`:
   - Tests the Dijkstra's algorithm on a simple path from node 1 to node 5 in the test graph.
   - Verifies if the calculated cost matches the expected value.
4. `test_dijkstra_disconnected_graph()`:
   - Tests Dijkstra's algorithm on a disconnected graph where there is no direct path from the source to the destination.
   - Verifies if the algorithm handles disconnected graphs gracefully.
5. `test_bellman_ford_negative_cycle()`:
   - Tests the Bellman-Ford algorithm on a graph containing a negative-weight cycle.
   - Verifies if the algorithm correctly detects the negative-weight cycle and raises an appropriate error.
6. `test_a_star_adapter()`:
   - Tests the A* algorithm adapter with various heuristic graphs and weighted edges.
   - Verifies if the calculated costs match the expected values.
7. `test_a_star_simple_weighted_graph()`:
   - Tests the A* algorithm on a simple weighted graph.
   - Verifies if the calculated cost matches the expected value.
8. `test_bellman_ford_no_negative_cycle()`:
   - Tests the Bellman-Ford algorithm on a graph without any negative-weight cycle.
   - Verifies if the calculated cost matches the expected value.
9. `test_sp_algorithm_variety_graph_types()`:
   - Tests both Dijkstra's and Bellman-Ford algorithms on the same graph.
   - Verifies if both algorithms yield the same shortest path cost.