CCS592 Advanced Algorithms and Complexity Assignment 2

Group member:

Name	**No USM**
Chan Khai Shen	24101339

Table of Contents

1 Introduction

In computer science, graph is a type of data structure that contains a set of vertices and a set of edges. Each edge connects two different vertices. Graph can represent real world problems, such as flight routes between airports, where vertex represents airport and edge represents the price of flight ticket. By finding the shortest path from one starting vertex to all other vertices, one can find the combination of routes that yields the lowest ticket price from one starting airport to all other airports in the network. So, finding shortest path in a network is an important task related to graph data structure.

The shortest path problem can be solved by implementing Djikstra's algorithm and Bellman-Ford algorithm. Djikstra's algorithm utilizes greedy approach to find the shortest path by start selecting the vertex with the shortest distance first and build the list of selected vertices incrementally. On the other hand, Bellman utilizes dynamic programming approach to find the shortest path by repeatedly updating the distance of all vertices in a bottom-up manner. The time complexity of Djikstra's algorihm is $O((n + m) \times \log n)$, which is lower than that of Bellman-Ford algorithm ($O(n \times m)$). This means Djikstra's algorithm is faster than Bellman-Ford algorithm.

Sometimes the graph can have negative weight edges to represent discount in flight ticket price. The problem with Djikstra's algorithm is it fails when there is a negative weight edge in the graph. Contrarily, Bellman-Ford algorithm is able to handle graphs with negative weight edges. This property highlights the importance and usefulness of Bellman-Ford algorithm even though its higher time complexity. On top of that, Bellman-Ford algorithm can be modified to detect and track negative weight cycle in a graph. This is important because negative cycle can indicate that there are some mistakes in setting the flight ticket price discounts that results in a total negative price when travelling between certain airports in cycle.

This brings to the goal of this report, which is to simulate Bellman-Ford algorithm and Djikstra's algorithm on finding the shortest path from one starting vertex to all other vertex. A weighted directed graph with some negative weight edges and a negative cycle is used as the sample graph in the simulation. The simulation will reveal how Djikstra's algorithm fails in handling negative weight edges and how Bellman-Ford algorithm succeeds. The simulation will also explore the ability of Bellman-Ford algorithm in detecting, tracing and reporting the negative cycle.

2 Algorithm explanation

This part of the report will explain how Bellman-Ford algorithm finds the shortest path, handles negative weight edges and detects negative cycle. Besides that, this part of the report will also explain how Djikstra's algorithm find the shortest path.

2.1 Bellman-Ford algorithm with negative cycle tracing

Pseudocode 2.1 depicts the steps for Bellman-Ford algorithm in finding the shortest path between the starting vertex and all other vertices and detecting negative cycle if there is any negative cycle in the graph.

Pseudocode 2.1: Bellman-Ford algorithm

```
BellmanFord(start):
   # Initialize distances and previous paths
   distances[start] = 0
   for vertex in all vertices:
      previousEdge[vertex] = None
      if vertex != start:
         distances[vertex] = infinity

   # Find shortest path
   for iteration in range(n – 1):
      for edge in all edges:
         if distances[edge.source] + edge.weight < distances[edge.destination]:
            distances[edge.destination] = distances[edge.source] + edge.weight
            previousEdge[edge.destination] = edge

   # Detect negative cycle
   for edge in all edges:
      if distances[edge.source] + edge.weight < distances[edge.destination]:
         cycleText = edge.source + '-' + edge.name + '->' + edge.destination
         previous = edge.source
         while previous != edge.destination:
            cycleText = previousEdge[previous].source + '-' +
                           previousEdge[previous].name + cycleText
            previous = previousEdge[previous].source
```

As shown in Pseudocode 2.1, Bellman-Ford algorithm takes an input parameter, which is the starting vertex and find the shortest path from the starting vertex to all other vertices. Firstly, the algorithm sets the distance to the starting vertex to 0 and the distance to all other vertices to infinity. Infinity indicates that the vertex is currently not reachable. Besides that, for all vertices, the previous edge is set to “None”. Previous edge keeps track of the last edge connecting to the vertex that forms the shortest path.

After that, the search for the shortest path starts. The search process is repeated for n-1 iterations, where n is the number of vertices. In the n-th iteration, the shortest distance recorded for all vertices is correct up to the path with n edges. The algorithm uses a bottom-up approach, which builds the knowledge on the shortest distances with 2 edges based on the knowledge of the shortest distances with 1 edge, and the knowledge on the shortest distances with 3 edges based on the knowledge of the shortest distances with 2 edges. The cloud of shortest path is a tree, and the maximum number of edges for a tree is n-1, so the ultimate shortest distance for all vertices can be found after n-1 iterations, with the condition that there is no negative cycle in the graph. This is because negative cycle causes an infinite loop with decreasing shortest distance.

In each iteration, every edge in the graph is examined. If the sum of the weight of the edge and the current shortest distance to the source vertex is smaller than the current shortest distance to the destination vertex, then there exists a shorter path, so the shortest distance to the destination vertex is updated to the sum of the weight of the edge and the shortest distance to the source vertex. Besides that, the previous edge of the destination vertex is also updated to the currently examined edge.

After searching the shortest distance for n-1 iterations, if there is still room of improvement in the shortest distance, then a negative cycle exists. Negative cycle causes the shortest distance to infinitely decrease. When a negative cycle is detected, the cycle is traced and reported. Firstly, the algorithm records the name of the edge that brings to a shorter distance after n-1 iterations and its source vertex and destination vertex. This is followed by backtracking to the previous edge by getting the previous edge of the source vertex and record the name and the source vertex of the edge. The backtracking process is repeated until the destination vertex of the first edge is met. At this time, a cycle is formed and this is the negative cycle.

Above are all the steps for Bellman-Ford algorithm in finding the shortest path from the starting vertex to all other vertices and detecting and tracing a negative cycle in the graph. The specification that allows Bellman-Ford algorithm to handle negative weight edge is explained in section 2.3.

2.2 Djikstra’s algorithm

Pseudocode 2.2 depicts the steps for using Djikstra’s algorithm to find the shortest path from one chosen starting vertex to all other vertices.

Pseudocode 2.2: Djikstra’s algorithm

```
Djikstra(start):
    # Initialize distances and previous paths and enqueue every vertex
    distances[start] = 0
    distanceQueue.enqueue(start, 0)
    for vertex in all vertices:
        previousEdge[vertex] = None
        if vertex != start:
            distances[vertex] = infinity
            distanceQueue.enqueue(vertex, infinity)

    # Find shortest path
    while distanceQueue is not empty:
        smallestKeyItem = distanceQueue.dequeue()
        source = smallestKeyItem.vertex
        for edge in all outgoing edges from source:
            if (edge.destination is in distanceQueue and
            distances[edge.source] + edge.weight < distances[edge.destination]):
                distances[edge.destination] = distances[edge.source] + edge.weight
                previousEdge[edge.destination] = edge
                distanceQueue.reduceKey(vertex, distances[edge.source]
```

As shown in Pseudocode 2.2, Djikstra’s algorithm takes a starting vertex as input and finds the shortest distance from that starting vertex to all other vertices. Firstly, the algorithm sets the distance to the starting vertex to 0 and enqueue the starting vertex into a priority queue named distanceQueue with key 0. Then, for all vertices, the previous edge is set to None. For all vertices except the starting vertex, the distance to that vertex is set to infinity and that vertex is enqueued into the priority queue with key infinity.

After that, the search for the shortest path starts. The process is repeated as long as the priority queue is not empty. In other words, the process is repeated as many times as the length of the priority queue, which is equivalent to the number of vertices. In every iteration, the priority queue dequeues the vertex with the smallest key and include that vertex into the cloud of shortest paths. Since the queue is a priority queue, the vertex selected in each iteration is the vertex that have the shortest distance to the starting point among those vertices that are still not yet included in the cloud of the shortest path. By applying greedy aproach, the algorithm builds the knowledge of shortest distances with the assumption that the vertices that is included into the cloud earlier will always have shorter distance than the vertices that is included later, and will not re-visit the vertices that is already in the cloud. This assumption is true, with the condition that all edges have positive weights.

In each iteration, the algorithm sets the vertex dequeued by the priority queue as the source vertex. Then, the algorithm examines every outgoing edge from the source vertex. If the destination vertex of the edge is still in the queue and the sum of the weight of the edge and the distance to the source vertex is smaller than the distance of the destination vertex, then there exists a shorter path, so the shortest distance to the destination vertex is updated to the sum of the weight of the edge and the distance to the source vertex. The previous edge of the vertex is also changed to currently examined edge. Besides that, the key of the destination vertex in the priority queue is also updated to reflect the change in the shortest distance to the vertex. The key of the vertex in the priority queue is equal to the shortest distance to the vertex. Importantly, only if the destination edge is still in the priority queue, then the change of its shortest distance is allowed to happen. In other words, all previously selected vertices are omitted because the algorithm assumes that the shortest distance to those vertices are already found at the time when it is included into the cloud of shortest path. This is fine with graph with all positive weight edges, but can cause error when there is negative weight edge.

2.3 Handling negative weight edge

Djikstra's algorithm cannot handle negative weight edge. This is because Djikstra's algorithm's greedy approach which includes the vertex with the shortest distance in the cloud of shortest path in the first iteration and the vertex with the second shortest distance in the second iteration, and will not update the shortest distance of the vertices in the cloud of shortest path in the later iterations. If a negative weight edge connecting to a vertex that is already in the cloud of shortest path is discovered in later iterations, then the actual shortest distance of the vertex might be shorter than the recorded shortest distance. This results in a wrong shortest distance.

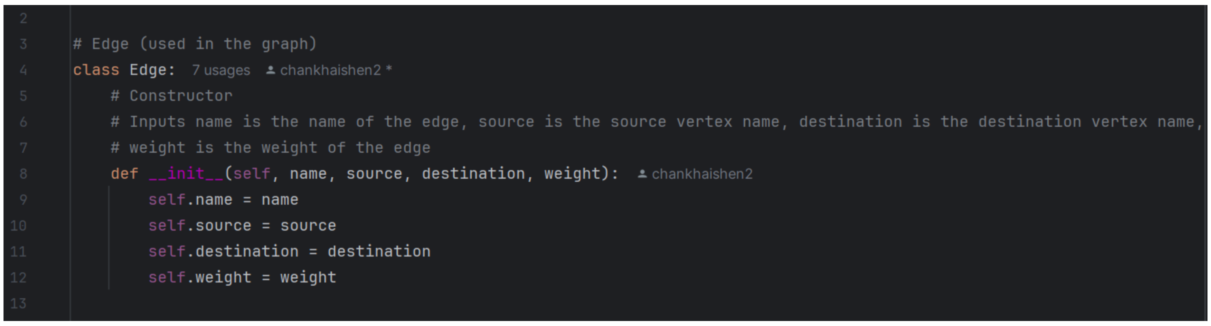
Contrarily, Bellman-Ford algorithm is able to handle negative weight edge by updating the shortest distance to each vertex in a bottom-up manner for n-1 iterations. Bellman-Ford algorithm is able to update every vertex whenever a shorter distance is found in later iterations. The shortest distance to any vertex is never fixed and changes are allowed until the n-1-th iteration. This provides flexibility to update the distance when a negative weight edge connected to a vertex brings reduction of distance to the vertex.

3 Source code overview

The programming language used in the source code is Python. The source code is implemented and run using PyCharm. One library, which is the Math library, is used in the source code. “math.inf” from the Math library is used to represent positive infinity. In both Bellman-Ford algorithm and Djikstra’s algorithm, initially the shortest distance of all vertices except the start vertex are set to positive infinity. The source code for implementing the graph, Bellman-Ford algorithm and Djikstra’s algorithm are discussed in sections 3.1, 3.2 and 3.3 respectively.

3.1 Graph

The graph is implemented using adjacency list. The adjacency list is implemented using Python’s dictionary data type, where the key is the vertex and the value is a list data type carrying all outgoing edges from the vertex. The vertex is represented by the name of the vertex in string. For example, vertex 0 is represented as a string ‘0’. The edge is represented by a custom edge object. Figure 3.1 depicts the details of the custom edge class.

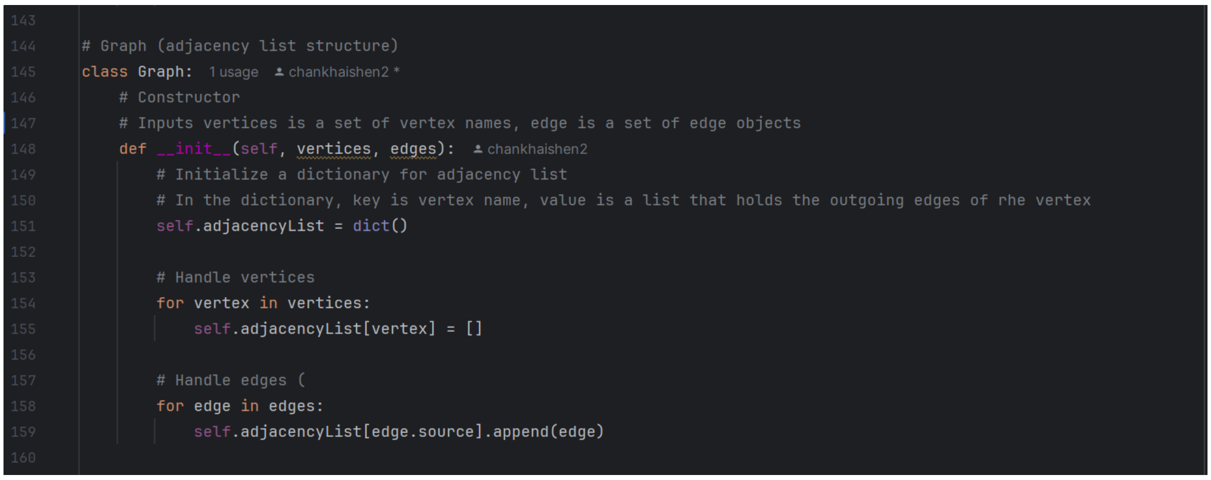


```python

# Edge (used in the graph)
class Edge:  7 usages  chankhaishen2 *
    # Constructor
    # Inputs name is the name of the edge, source is the source vertex name, destination is the destination vertex name,
    # weight is the weight of the edge
    def __init__(self, name, source, destination, weight):  chankhaishen2
        self.name = name
        self.source = source
        self.destination = destination
        self.weight = weight

```

Figure 3.1 Edge class

As shown in Figure 3.1, the edge class has 4 attributes, namely name, source, destination and weight. The edge is a weighted directed edge, which is in line with the sample graph given. The name attribute stores the name of the edge in string, whereas the source attribute stores the name of the source vertex in string, whereas the destination attribute stores the name of the destination vertex in string, whereas the weight attribute stores the weight of the edge in integer. The object created from this edge class is used as the edge in the adjacency list in the graph.

The graph is implemented using a graph class. Figure 3.2 shows the constructor for the graph class.

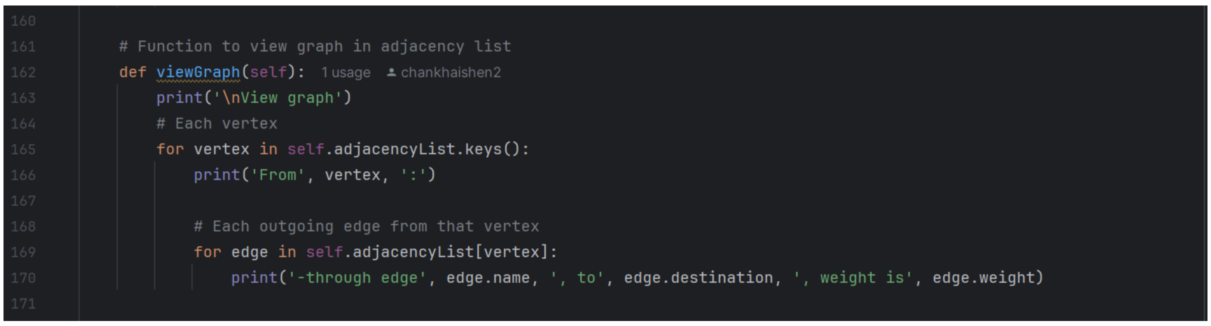


```python

# Graph (adjacency list structure)
class Graph:  1 usage  chankhaishen2 *
    # Constructor
    # Inputs vertices is a set of vertex names, edge is a set of edge objects
    def __init__(self, vertices, edges):  chankhaishen2
        # Initialize a dictionary for adjacency list
        # In the dictionary, key is vertex name, value is a list that holds the outgoing edges of rhe vertex
        self.adjacencyList = dict()

        # Handle vertices
        for vertex in vertices:
            self.adjacencyList[vertex] = []

        # Handle edges (
        for edge in edges:
            self.adjacencyList[edge.source].append(edge)

```

Figure 3.2 Graph class constructor

The graph class has only one attribute named adjacencyList, which is a dictionary that stores the adjacency list of the graph. As shown in Figure 3.2, the constructor of the graph accepts two inputs, namely vertices and edges, and builds the adjacency list from the inputs. The vertices input is the list of vertices in the graph, whereas the edges input is the list of edges in the graph. Firstly, the constructor initializes the adjacencyList attribute as an empty dictionary. The constructor then loops through the list of vertices and for each vertex, the constructor adds new key-value pair with the name of the vertex as the key and an empty list as the value. Finally, the constructor loops through the list of edges and for each edge, the constructor appends the edge to the list which the source vertex of the edge is the key. After all, an adjacency list, which is a dictionary which each key is the name of a vertex and the value is the list of outgoing edges from the vertex, is formed.

The adjacency list structure can be printed by using the viewGraph() function in the graph class. Figure 3.3 depicts the viewGraph() function.



```
    # Function to view graph in adjacency list
    def viewGraph(self):  1 usage  chankhaishen2
        print('\nView graph')
        # Each vertex
        for vertex in self.adjacencyList.keys():
            print('From', vertex, ':')

            # Each outgoing edge from that vertex
            for edge in self.adjacencyList[vertex]:
                print('-through edge', edge.name, ', to', edge.destination, ', weight is', edge.weight)
```

Figure 3.3 View graph function

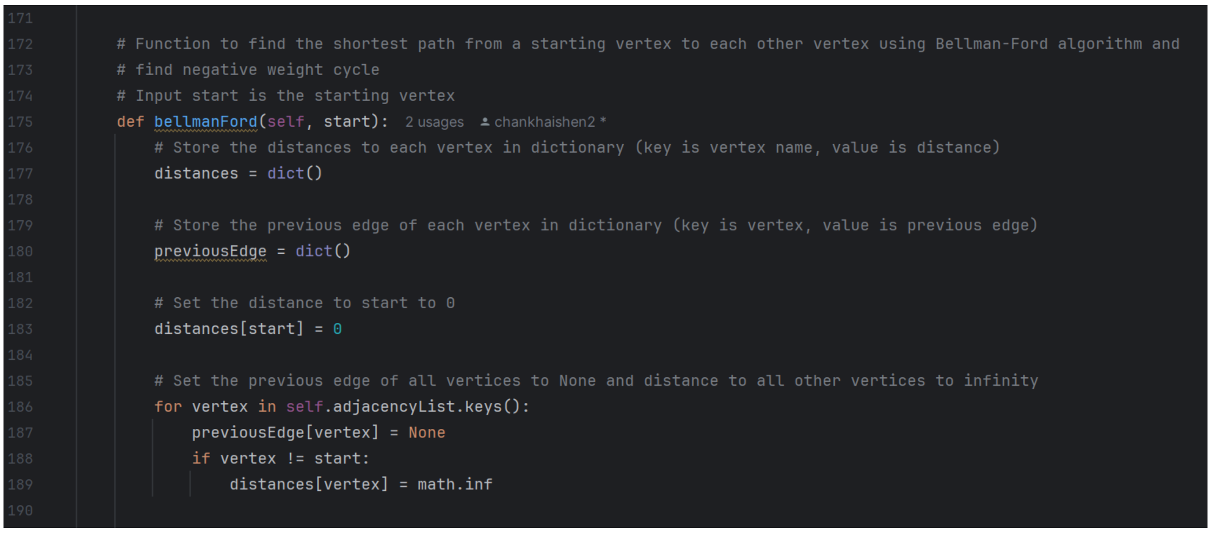
As shown in Figure 3.3, the function loops through the list of vertices and for each vertex, the function loops through the list of outgoing edges to print the name of the edge, the destination vertex of the edge and the weight of the edge.

3.2 Bellman-Ford algorithm

The Bellman-Ford algorithm is implemented as a function in the graph class. The function is the bellmanFord() function.

3.2.1 Initialization and finding the shortest path

When calling the bellmanFord() function, there is an input parameter named start, which is the starting vertex for the Bellman-Ford algorithm. Figure 3.4 shows the initialization for the Bellman-Ford algorithm.



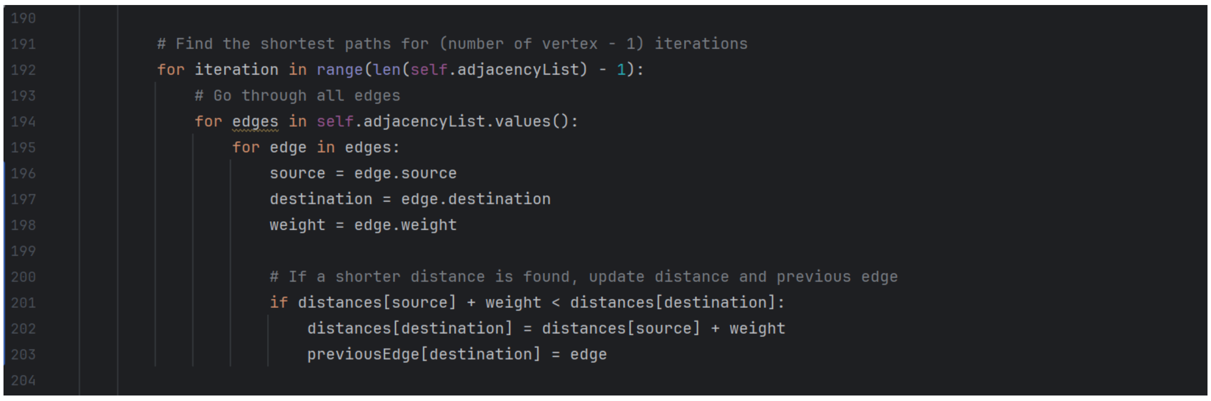
```python
    # Function to find the shortest path from a starting vertex to each other vertex using Bellman-Ford algorithm and
    # find negative weight cycle
    # Input start is the starting vertex
    def bellmanFord(self, start):  2 usages  chankhaishen2 *
        # Store the distances to each vertex in dictionary (key is vertex name, value is distance)
        distances = dict()

        # Store the previous edge of each vertex in dictionary (key is vertex, value is previous edge)
        previousEdge = dict()

        # Set the distance to start to 0
        distances[start] = 0

        # Set the previous edge of all vertices to None and distance to all other vertices to infinity
        for vertex in self.adjacencyList.keys():
            previousEdge[vertex] = None
            if vertex != start:
                distances[vertex] = math.inf
```

Figure 3.4 Initialization (Bellman-Ford algorithm)

As shown in Figure 3.4, two empty dictionaries, namely distances and previousEdge, are initialized. The distances dictionary stores the shortest distance to each vertex from the start vertex, where the key is the name of the vertex and the value is the shortest distance to the vertex. The previousEdge dictionary stores the previous edge of each vertex in the path that results in the shortest distance, where the key is the name of the vertex and the value is the previous edge (edge class).

After that, as the part of initialization for the Bellman-Ford algorithm, the distance to the starting vertex is set to 0, whereas the distance to all other vertex is set to infinity by using “math.inf” from the Math library. Besides that, the previous edge of all vertices is set to “None”.

After initialization, the next step is finding the shortest path from the start vertex to each vertex. Figure 3.5 depicts the code for finding the shortest path using Bellman-Ford algorithm.



```python
        # Find the shortest paths for (number of vertex - 1) iterations
        for iteration in range(len(self.adjacencyList) - 1):
            # Go through all edges
            for edges in self.adjacencyList.values():
                for edge in edges:
                    source = edge.source
                    destination = edge.destination
                    weight = edge.weight

                    # If a shorter distance is found, update distance and previous edge
                    if distances[source] + weight < distances[destination]:
                        distances[destination] = distances[source] + weight
                        previousEdge[destination] = edge
```

Figure 3.5 Find the shortest path (Bellman-Ford algorithm)

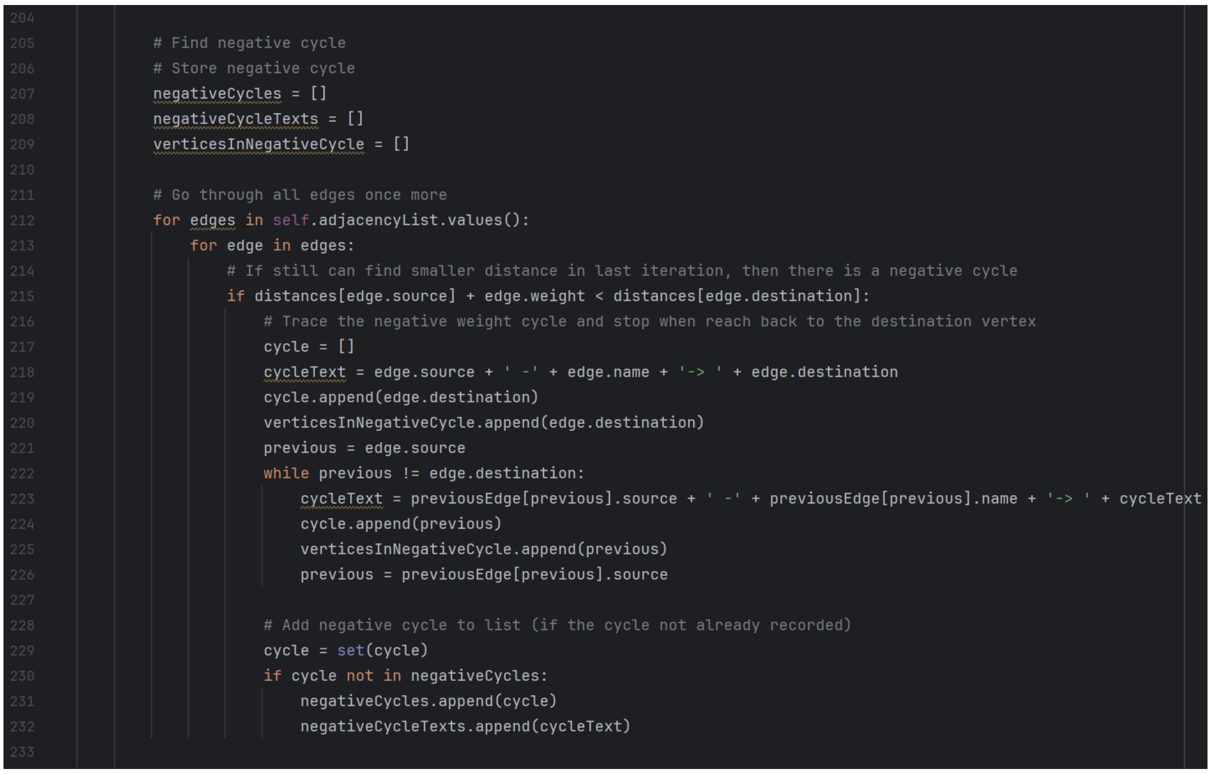
As shown in Figure 3.5, the information of the shortest path is incrementally built in several iterations. The number of iterations is equal to the length of the adjacency list minus one, which is synonym to the number of vertices minus one. This is due to the fact that in the i-th iteration, the shortest path is correct up to the path with i edges and the maximum number of edges a path can have without repeating any vertex is the number of vertices minus one.

In every iteration, the programme loops through all values in the adjacency list, which is equivalent to the collection of all the list of outgoing edges from each vertex. Then, in each list of outgoing edges, the programme loops through each edge. Importantly, combining these two for loops, all edge in the graph is processed exactly once.

For each edge, if the sum of the weight of the edge and the shortest distance to the source vertex of the edge is smaller than the shortest distance to the destination vertex of the edge, then there exists a shorter distance to the destination vertex. In this case, the shortest distance to the destination vertex will be changed to the sum of the weight and the shortest distance to the source vertex. The previous edge of the destination vertex will also be changed to the currently processed edge. The programme updates the distances dictionary and previousEdge dictionary to reflect the changes.

3.2.2 Detecting and tracing negative cycle

After all edges has been examined for “number of vertices minus one” times, the shortest distance from the starting vertex to all other vertices has been found, with the condition that the graph does not have a negative cycle. If the graph has a negative cycle, then the shortest distance to the vertices in the negative cycle is able to be decreased infinitely. In other words, if after “number of vertices minus one” iterations, the shortest distance of a vertex can still be decreased, then the vertex is in a negative cycle. Figure 3.6 depicts the code to detect and trace negative cycles in the graph.



```python

        # Find negative cycle
        # Store negative cycle
        negativeCycles = []
        negativeCycleTexts = []
        verticesInNegativeCycle = []

        # Go through all edges once more
        for edges in self.adjacencyList.values():
            for edge in edges:
                # If still can find smaller distance in last iteration, then there is a negative cycle
                if distances[edge.source] + edge.weight < distances[edge.destination]:
                    # Trace the negative weight cycle and stop when reach back to the destination vertex
                    cycle = []
                    cycleText = edge.source + ' -' + edge.name + '-> ' + edge.destination
                    cycle.append(edge.destination)
                    verticesInNegativeCycle.append(edge.destination)
                    previous = edge.source
                    while previous != edge.destination:
                        cycleText = previousEdge[previous].source + ' -' + previousEdge[previous].name + '-> ' + cycleText
                        cycle.append(previous)
                        verticesInNegativeCycle.append(previous)
                        previous = previousEdge[previous].source

                    # Add negative cycle to list (if the cycle not already recorded)
                    cycle = set(cycle)
                    if cycle not in negativeCycles:
                        negativeCycles.append(cycle)
                        negativeCycleTexts.append(cycleText)

```

Figure 3.6 Detect and trace negative cycle (Bellman-Ford algorithm)

As shown in Figure 3.6, firstly, the programme initializes three empty list, namely negativeCycles, negativeCycleTexts and verticesInNegativeCycle. The negativeCycle list stores all negative cycles, where each negative cycle is a set of vertices that forms the cycle. This list is used to check duplicate negative cycle. The negativeCycleText list stores all negative cycle texts, which contains the trail of a cycle in text, which is in term of name of vertex and name of edge (e.g. 3 -e-> 4 -f-> 5 -g-> 3). All strings in this list, which contains the trail of every negative cycle, will be printed later. The verticesInNegativeCycle list stores the list of all vertex names that is in any negative cycles. This list is used as a reference when printing the results because the shortest distance of the vertices that is in a negative cycle will not be printed.

In the two for loops (which is same as the for loop in Figure 3.5, which is explained previously), all edges are passed exactly once. Importantly, this happens after the programme had checked all edges for “number of vertices minus one” iterations. For each edge, if the sum of the weight of the edge and the shortest distance to the source vertex is smaller than the shortest distance to the destination vertex, then the edge is in a negative cycle.

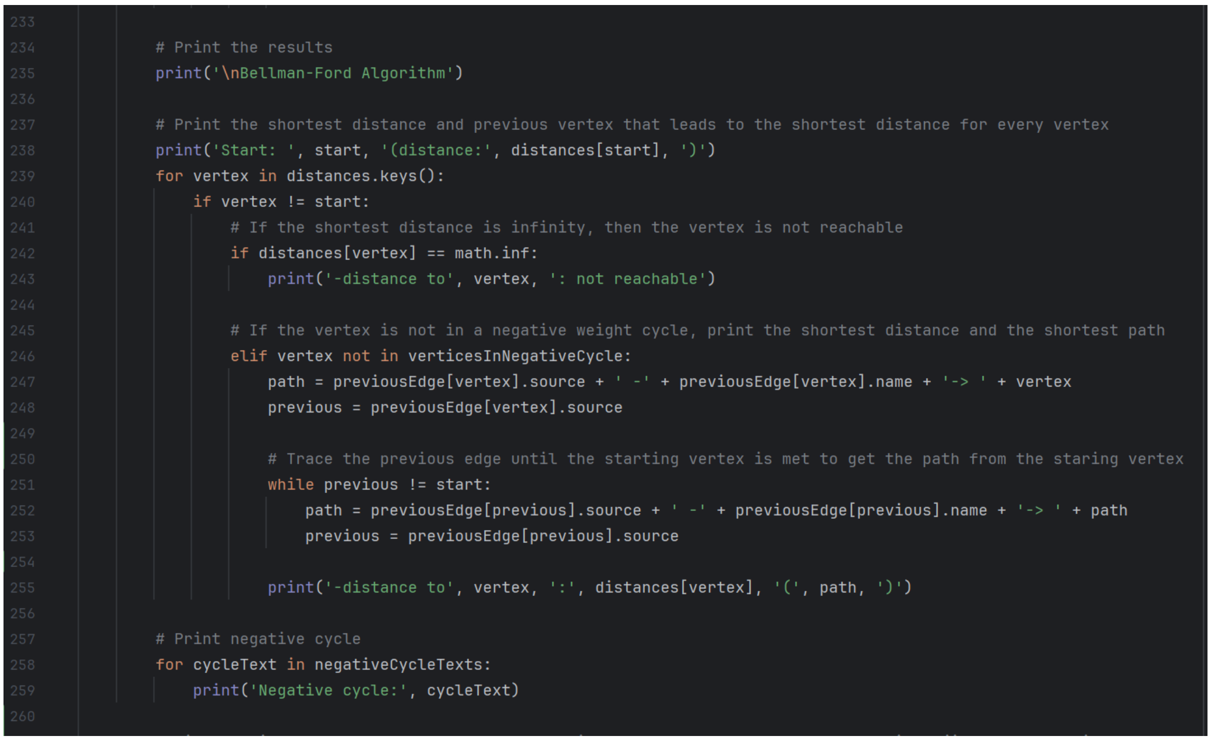
Whenever an edge is in a negative cycle, a variable named cycle is initialized as an empty list. After that, the source vertex of the edge, the destination vertex of the edge and the weight of the edge are recorded in a variable called cycleText. After that, the destination vertex is appended to the verticesInNegativeCycle list and the cycle list.

The programme then backtracks to trace the negative cycle. Firstly, the programme assigns a variable named previous to point to the source vertex of the edge. For the vertex previous, the source vertex of the previous edge of previous and the weight of the previous edge of previous are added to the current record of cycleText variable. Then, the vertex previous is appended to the cycle list and the verticesInNegativeCycle list. Finally, the variable previous is pointed to the source vertex of the previous edge of the vertex previous. This brings to the next iteration of tracing of negative cycle. This process is repeated until the variable previous points to the destination vertex of the first edge, which signifies that a cycle is formed.

After the tracing of the negative cycle finished, the cycle list is converted to set data type. If the cycle set is not equal to any set in the negativeCycles list, then the cycles set is appended to the negativeCycles list and the cycleText string is appended to the negativeCycleTexts list. This step ensures that there is no duplicate in the negative cycles recorded. The cycle list is converted to set data type because the order of the elements is important when comparing list, whereas the order of the element is not important when comparing set.

3.2.3 Print results

As the final step of Bellman-Ford algorithm, the shortest path from the start vertex to all other vertices (except the vertices in negative cycle) and the negative cycle (if exist) are printed. Figure 3.7 shows the code for printing the results.



```python

        # Print the results
        print('\nBellman-Ford Algorithm')

        # Print the shortest distance and previous vertex that leads to the shortest distance for every vertex
        print('Start: ', start, '(distance:', distances[start], ')')
        for vertex in distances.keys():
            if vertex != start:
                # If the shortest distance is infinity, then the vertex is not reachable
                if distances[vertex] == math.inf:
                    print('-distance to', vertex, ': not reachable')

                # If the vertex is not in a negative weight cycle, print the shortest distance and the shortest path
                elif vertex not in verticesInNegativeCycle:
                    path = previousEdge[vertex].source + ' -' + previousEdge[vertex].name + '-> ' + vertex
                    previous = previousEdge[vertex].source

                    # Trace the previous edge until the starting vertex is met to get the path from the staring vertex
                    while previous != start:
                        path = previousEdge[previous].source + ' -' + previousEdge[previous].name + '-> ' + path
                        previous = previousEdge[previous].source

                    print('-distance to', vertex, ':', distances[vertex], '(', path, ')')

        # Print negative cycle
        for cycleText in negativeCycleTexts:
            print('Negative cycle:', cycleText)

```

Figure 3.7 Print results (Bellman-Ford algorithm)

As shown in Figure 3.7, firstly, the programme prints the shortest distance from the start vertex to the start vertex. After that, the programme loops through the list of keys of the distances dictionary, which is equivalent to the list of vertices, to print the shortest distance of each vertex.

For each vertex, if the shortest distance to the vertex is infinity, then the programme will state that the vertex is not reachable. Otherwise, the programme will trace the shortest path from the start vertex to the vertex. The name of the vertex, the name of its previous edge and the source vertex of its previous edge are recorded in a variable named path. After that, a variable named previous is assigned to point to the source vertex of the previous edge of the currently examined vertex. Then, the name of the source vertex of the previous edge of the vertex previous and the name of the previous edge of the vertex previous are added to the variable path. After that, the variable previous is pointed to the source vertex of the previous edge of the vertex previous to go up to the previous level of the shortest path. This process is repeated until the variable previous points to the start vertex. Then, the shortest distance of the vertex and the shortest path from the variable path are printed.

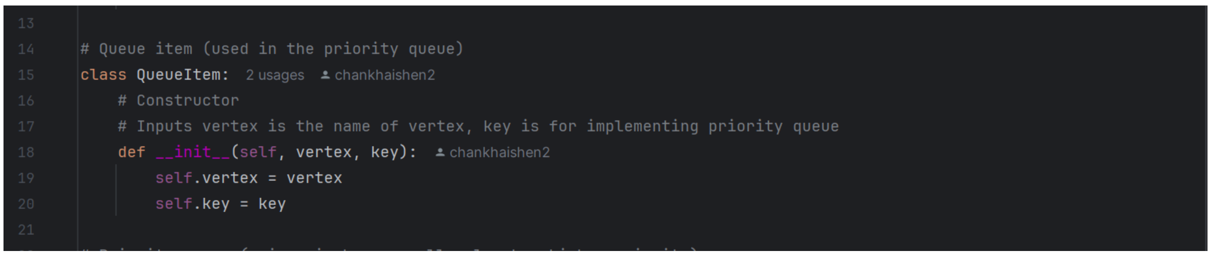
On the other hand, for the negative cycles, the programme loops through the negativeCycleTexts list to print the cycleText string of each negative cycle. If there is no negative cycle in the graph, then nothing will be printed from this portion of the programme.

3.3 Djikstra’s algorithm

In the source code, Djikstra’s algorithm is implemented as a function in the graph class. This function is the Djikstra() function.

3.3.1 Priority queue

Djikstra’s algorithm in the Djikstra() function uses a priority queue to store the vertices and get the vertex with the shortest distance. The priority queue is implemented using the priority queue class, which implements priority queue using min heap. The priority queue class holds a list of queue item object. Figure 3.8 shows code for the constructor of the queue item class.

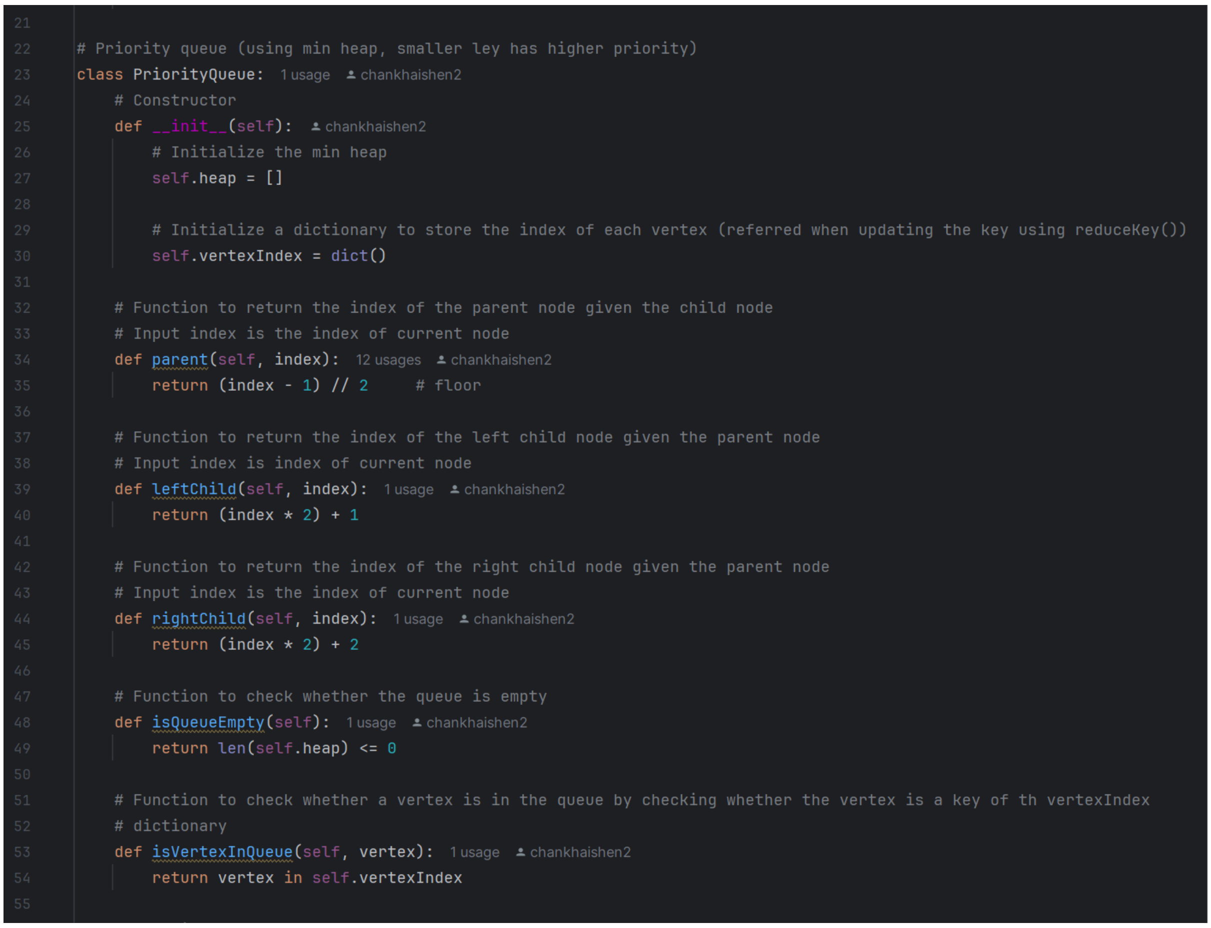


```
# Queue item (used in the priority queue)
class QueueItem:  2 usages  chankhaishen2
    # Constructor
    # Inputs vertex is the name of vertex, key is for implementing priority queue
    def __init__(self, vertex, key):  chankhaishen2
        self.vertex = vertex
        self.key = key
```

Figure 3.8 Queue item class

As shown in Figure 3.8, the queue item class is a simple class which its constructor takes two input arguments, namely vertex and key, and stores it inside the class. The vertex is the name of the vertex inserted to the queue and the key is the key of the vertex which is equivalent to its shortest distance from the start vertex.

Figure 3.9 depicts the code for implementing the constructor of the priority queue, the functions to return the index of parent, left child and right child, the function to check whether the queue is empty and the function to check whether a vertex is in the queue.



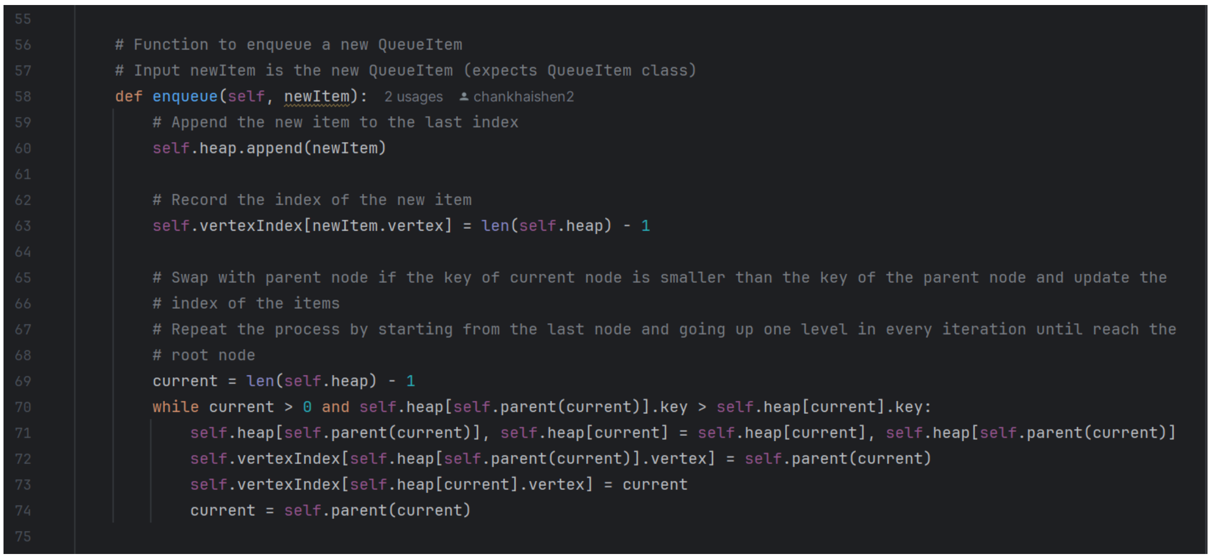
```
# Priority queue (using min heap, smaller ley has higher priority)
class PriorityQueue:
    # Constructor
    def __init__(self):
        # Initialize the min heap
        self.heap = []

        # Initialize a dictionary to store the index of each vertex (referred when updating the key using reduceKey())
        self.vertexIndex = dict()

    # Function to return the index of the parent node given the child node
    # Input index is the index of current node
    def parent(self, index):
        return (index - 1) // 2     # floor

    # Function to return the index of the left child node given the parent node
    # Input index is index of current node
    def leftChild(self, index):
        return (index * 2) + 1

    # Function to return the index of the right child node given the parent node
    # Input index is the index of current node
    def rightChild(self, index):
        return (index * 2) + 2

    # Function to check whether the queue is empty
    def isQueueEmpty(self):
        return len(self.heap) <= 0

    # Function to check whether a vertex is in the queue by checking whether the vertex is a key of th vertexIndex
    # dictionary
    def isVertexInQueue(self, vertex):
        return vertex in self.vertexIndex
```

Figure 3.9 Constructor and some functions (priority queue)

As shown in Figure 3.9, the constructor of the priority queue class initializes an empty list named heap and an empty dictionary named vertexIndex. The heap list holds a list of queue item objects, which contains the vertices in the queue with their keys, which is the shortest distance from the start vertex to the vertex. The vertexIndex dictionary holds the index of each vertex in the heap list, where the key is the name of the vertex and the value is the index of the vertex. The dictionary helps improve the efficiency by eliminating the need to parse through the whole list when enquiring the index of a vertex given its name. By using a dictionary, checking the index of a vertex can be done in time complexity of $O(1)$.

The parent() function, the leftChild() function and the rightChild() function takes the index of any vertex as the input parameter and computes the index of its parent, left child and right child respectively. The index of the parent is equal to the floor of $(index - 1) \div 2$, whereas the index of the left child is equal to $(index \times 2) + 1$, whereas the index of the right child is equal to $(index \times 2) + 2$. The isQueueEmpty() function checks whether the queue is empty by returning true if the length of the heap list is 0 and returning false if otherwise. The isVertexInQueue() function takes the name of a vertex as the input and return true if the vertex is in the queue and return false if otherwise. The function checks whether the name of the vertex exists in the list of keys of the vertexIndex dictionary, which takes time complexity of $O(1)$.

Figure 3.10 shows the code for the enqueue() function of the priority queue class.



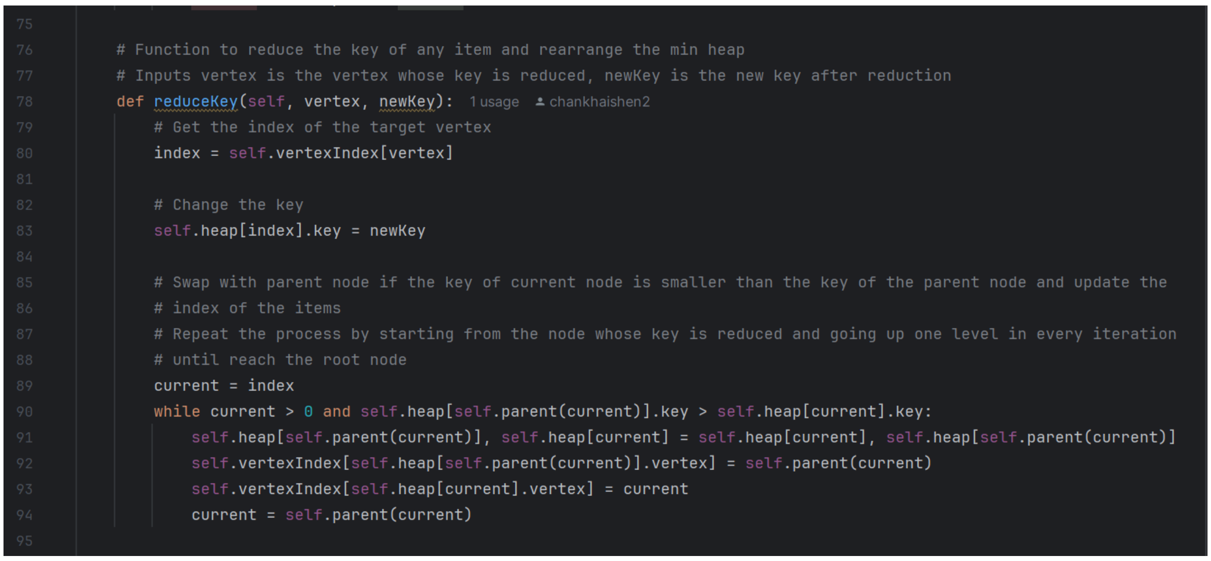
```python

    # Function to enqueue a new QueueItem
    # Input newItem is the new QueueItem (expects QueueItem class)
    def enqueue(self, newItem):  2 usages  chankhaishen2
        # Append the new item to the last index
        self.heap.append(newItem)

        # Record the index of the new item
        self.vertexIndex[newItem.vertex] = len(self.heap) - 1

        # Swap with parent node if the key of current node is smaller than the key of the parent node and update the
        # index of the items
        # Repeat the process by starting from the last node and going up one level in every iteration until reach the
        # root node
        current = len(self.heap) - 1
        while current > 0 and self.heap[self.parent(current)].key > self.heap[current].key:
            self.heap[self.parent(current)], self.heap[current] = self.heap[current], self.heap[self.parent(current)]
            self.vertexIndex[self.heap[self.parent(current)].vertex] = self.parent(current)
            self.vertexIndex[self.heap[current].vertex] = current
            current = self.parent(current)

```

Figure 3.10 Enqueue (priority queue)

As shown in Figure 3.10, the enqueue() function takes a queue item named newItem as its input parameter. The newItem parameter is the item to be inserted to the priority queue and is of queue item class. In the enqueue() function, firstly, the programme appends the new item to the heap list. The programme then creates an entry to the vertexIndex dictionary to record the index of the new item, which is the length of the heap list minus one.

After that, the programme goes through a series of steps to re-establish the min heap property. The programme sets a variable named current as the index of the new item in the heap list. If current is larger than 0 (current does not points to the first item, i.e. the root), the programme compares the key of the item pointed by current variable with the key of the parent of the item. If the key of the parent of the item is larger than the key of the item, then the item is swapped with its parent. If swapping happens, then the record in the vertexIndex dictionary is updated and the current variable is changed to the index of the parent of the currently examined item. This is repeated until either the root is reached or the key of the parent no longer larger than the key of the currently examined item.

Figure 3.11 depicts the code for the reduceKey() function of the priority queue class, which is used to reduce the key of a vertex in the queue when a shorter distance is found.

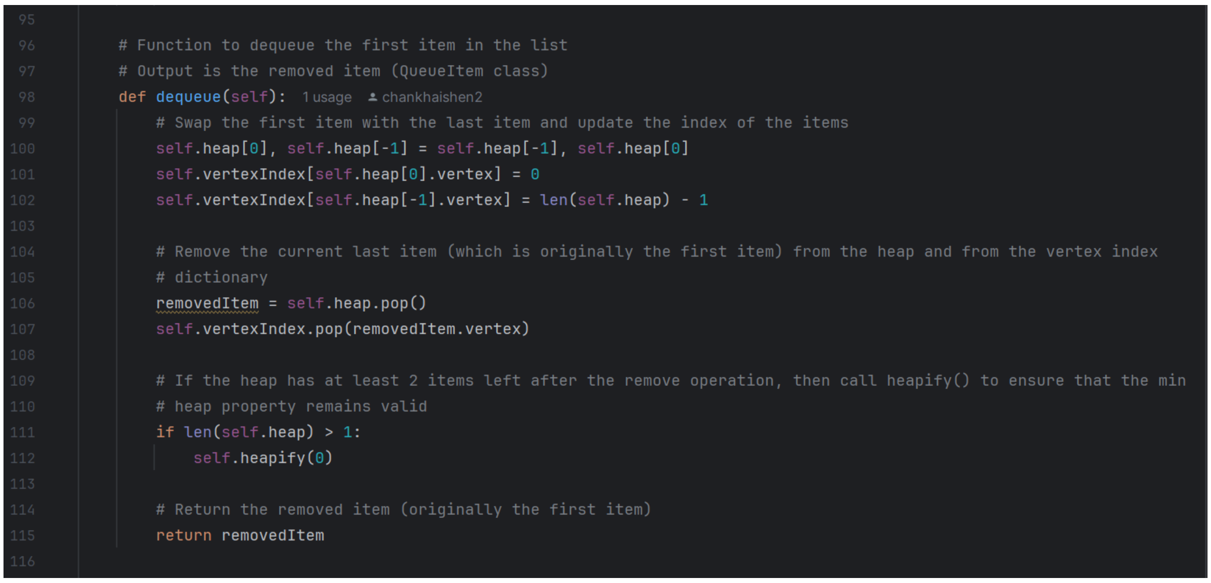


```
    # Function to reduce the key of any item and rearrange the min heap
    # Inputs vertex is the vertex whose key is reduced, newKey is the new key after reduction
    def reduceKey(self, vertex, newKey):  1 usage  chankhaishen2
        # Get the index of the target vertex
        index = self.vertexIndex[vertex]

        # Change the key
        self.heap[index].key = newKey

        # Swap with parent node if the key of current node is smaller than the key of the parent node and update the
        # index of the items
        # Repeat the process by starting from the node whose key is reduced and going up one level in every iteration
        # until reach the root node
        current = index
        while current > 0 and self.heap[self.parent(current)].key > self.heap[current].key:
            self.heap[self.parent(current)], self.heap[current] = self.heap[current], self.heap[self.parent(current)]
            self.vertexIndex[self.heap[self.parent(current)].vertex] = self.parent(current)
            self.vertexIndex[self.heap[current].vertex] = current
            current = self.parent(current)
```

Figure 3.11 Reduce key (priority queue)

As shown in Figure 3.11, the reduceKey() function takes two input parameters, namely vertex and newKey. The vertex parameter is the name of the vertex and the newKey parameter is the new smaller key of the vertex, which is equivalent to a smaller distance. Firstly, the programme retrieves the index of the vertex by referring to the vertexIndex dictionary by specifying the name of the vertex. Then, by specifying the index, the programme changes the key of the vertex. After that, the programme follows a similar workflow with that in enqueue() function to re-establish the min heap property (explained previously in enqueue() section). The only difference is that instead of the last index, the current variable in reduceKey() is initially set to the index of the vertex which the key is changed. This is because the process of re-establishing min heap property starts from the vertex which the key is changed.

Figure 3.12 shows the code for the dequeue() function of the priority queue class.

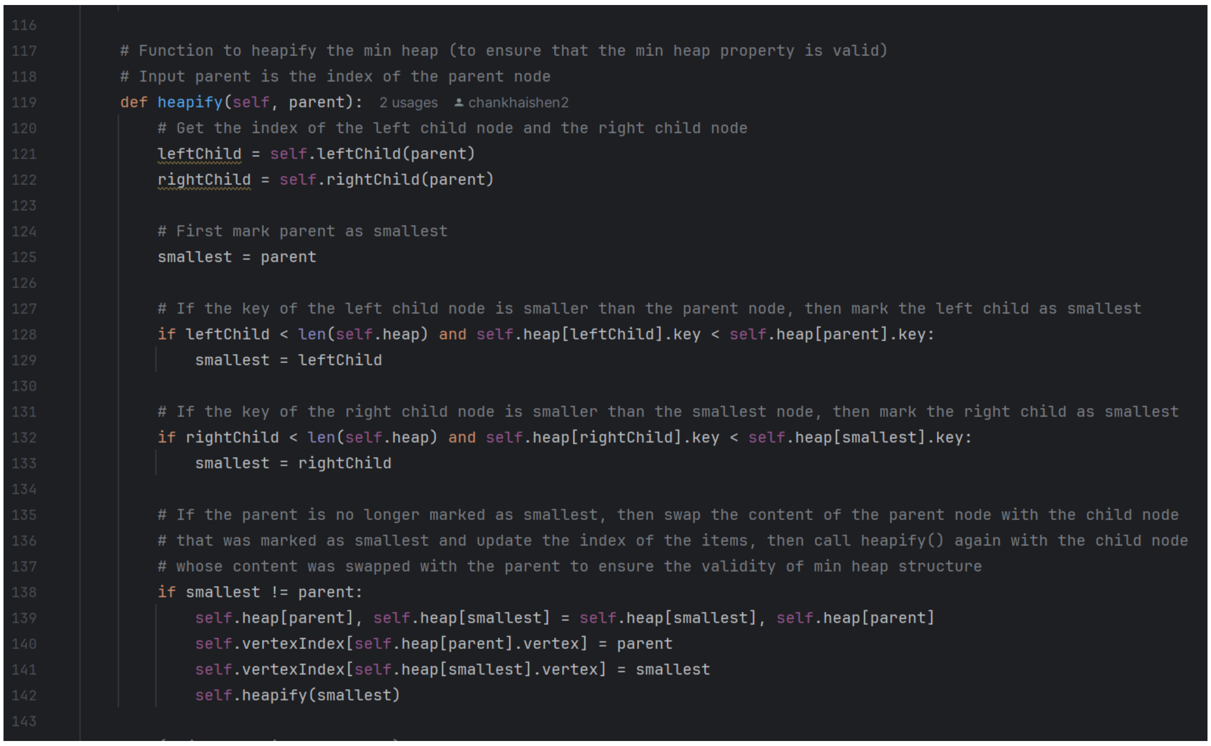


```python

    # Function to dequeue the first item in the list
    # Output is the removed item (QueueItem class)
    def dequeue(self):  1 usage  chankhaishen2
        # Swap the first item with the last item and update the index of the items
        self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]
        self.vertexIndex[self.heap[0].vertex] = 0
        self.vertexIndex[self.heap[-1].vertex] = len(self.heap) - 1

        # Remove the current last item (which is originally the first item) from the heap and from the vertex index
        # dictionary
        removedItem = self.heap.pop()
        self.vertexIndex.pop(removedItem.vertex)

        # If the heap has at least 2 items left after the remove operation, then call heapify() to ensure that the min
        # heap property remains valid
        if len(self.heap) > 1:
            self.heapify(0)

        # Return the removed item (originally the first item)
        return removedItem

```

Figure 3.12 Dequeue (priority queue)

As shown in Figure 3.12, firstly, the dequeue() function swaps the root of the heap (first item) with the bottom rightmost leaf (last item). The programme then updates the index of the swapped vertices in the vertexIndex dictionary. After that, the programme removes the bottom rightmost leaf of the heap, which is initially the root of the heap before swapping, from the heap. The record of the removed vertex in the vertexIndex dictionary is also removed. If the size of the heap is larger than 1, then the programme calls the heapify() function (will be explained later) by using index 0 as the input parameter to re-establish the min heap property of the heap. Finally, the dequeue() function returns the removed item, which is of the queue item class.

As mentioned previously, the dequeue() function calls the heapify() function. Figure 3.13 depicts the code for implementing the heapify() function.



```python
    # Function to heapify the min heap (to ensure that the min heap property is valid)
    # Input parent is the index of the parent node
    def heapify(self, parent):  2 usages  chankhaishen2
        # Get the index of the left child node and the right child node
        leftChild = self.leftChild(parent)
        rightChild = self.rightChild(parent)

        # First mark parent as smallest
        smallest = parent

        # If the key of the left child node is smaller than the parent node, then mark the left child as smallest
        if leftChild < len(self.heap) and self.heap[leftChild].key < self.heap[parent].key:
            smallest = leftChild

        # If the key of the right child node is smaller than the smallest node, then mark the right child as smallest
        if rightChild < len(self.heap) and self.heap[rightChild].key < self.heap[smallest].key:
            smallest = rightChild

        # If the parent is no longer marked as smallest, then swap the content of the parent node with the child node
        # that was marked as smallest and update the index of the items, then call heapify() again with the child node
        # whose content was swapped with the parent to ensure the validity of min heap structure
        if smallest != parent:
            self.heap[parent], self.heap[smallest] = self.heap[smallest], self.heap[parent]
            self.vertexIndex[self.heap[parent].vertex] = parent
            self.vertexIndex[self.heap[smallest].vertex] = smallest
            self.heapify(smallest)
```

Figure 3.13 Heapify (priority queue)

As shown in Figure 3.13, the heapify() function takes one input parameter named parent. The parent parameter is the index of the target node to be compared with its children node. Firstly, the programme gets the index of the left child and the right child of the node pointed by the parent parameter. Then, the programme assigns a variable named smallest to be equal to the parent parameter.

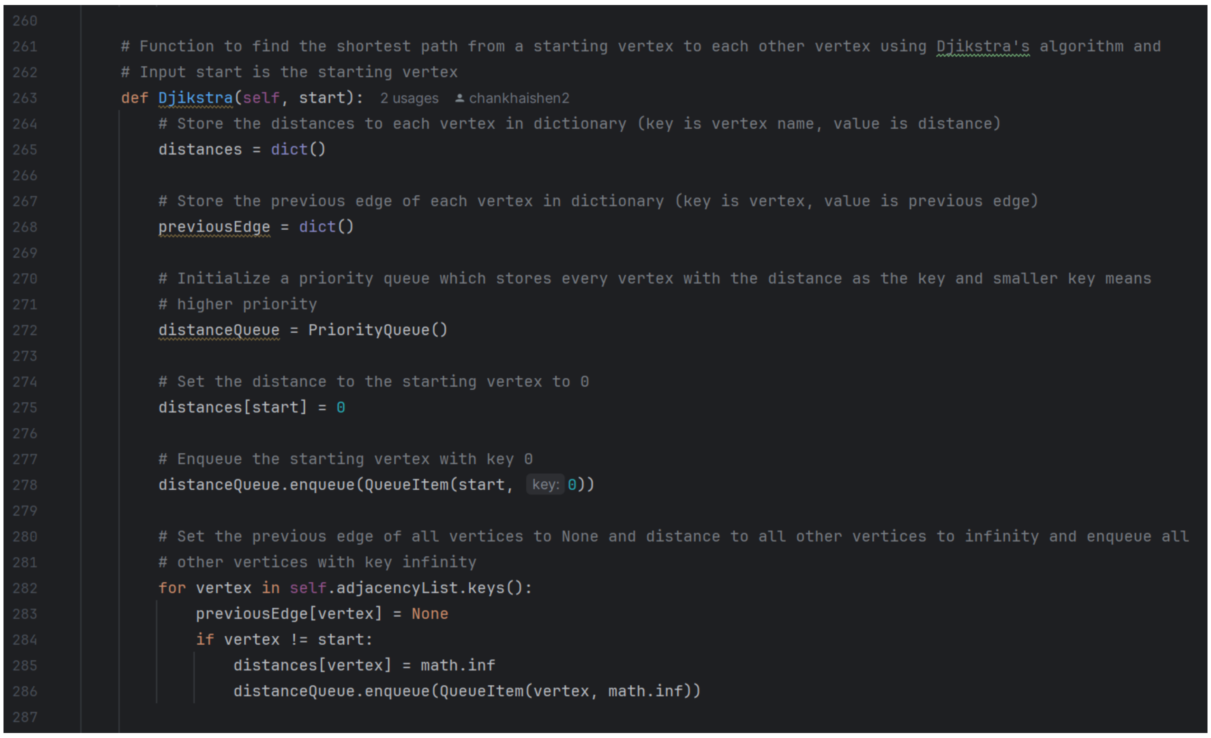
After that, the programme compares the key of the target node (parent) with the key of the left child node and the right child node of the target node. If the target node has a left child (the index of the left child is smaller than the length of the heap) and the key of the left child of the target node is smaller than the key of the target node, then the smallest variable is set to the index of the left child. After that, if the target node has a right child (the index of the right child is smaller than the length of the heap) and the key of the right child of the target node is smaller than the key of the node pointed by the smallest variable, then the smallest variable is set to the index of the right child.

After the crucial step of comparing the keys, if the smallest variable is no longer the index of the target node (not equal to parent parameter), then either the key of the left child or the key of the right child is smaller than the key of the target node (parent). In this case, the content at the target node (pointed by parent parameter) is swapped with the content at the node pointed by the smallest variable. After that, the index of the corresponding vertices in the vertexIndex dictionary are updated.

Finally, if the target node (parent) has been swapped with its one of its children, the heapify() function calls itself by using the index of the swapped child as the parent parameter. This is because the min heap property is not yet fully re-established, so the heapify process has to proceed to the next level of the heap tree as the position of vertex initially pointed by the parent parameter has been moved to its child. Else, if the target node (parent) has not been swapped, then the vertex pointed by the parent parameter is at the correct position and the heap property is re-established.

3.3.2 Initialization and finding the shortest path

The Djikstra's algorithm is implemented as a function in the graph class. Figure 3.14 shows the code for initialization for Djikstra's algorithm.



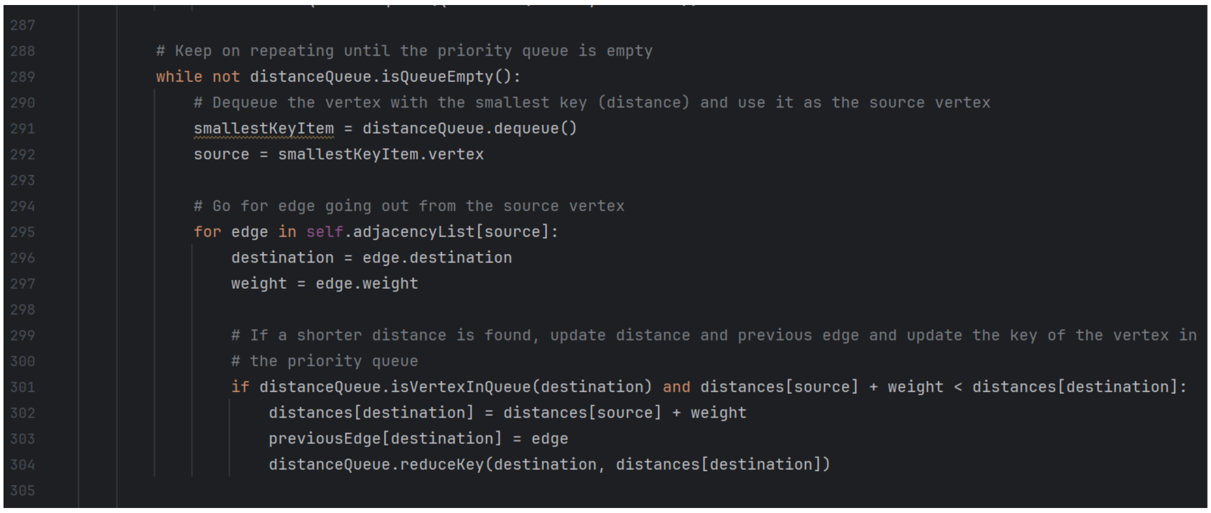
```

    # Function to find the shortest path from a starting vertex to each other vertex using Djikstra's algorithm and
    # Input start is the starting vertex
    def Djikstra(self, start):  2 usages  chankhaishen2
        # Store the distances to each vertex in dictionary (key is vertex name, value is distance)
        distances = dict()

        # Store the previous edge of each vertex in dictionary (key is vertex, value is previous edge)
        previousEdge = dict()

        # Initialize a priority queue which stores every vertex with the distance as the key and smaller key means
        # higher priority
        distanceQueue = PriorityQueue()

        # Set the distance to the starting vertex to 0
        distances[start] = 0

        # Enqueue the starting vertex with key 0
        distanceQueue.enqueue(QueueItem(start,  key: 0))

        # Set the previous edge of all vertices to None and distance to all other vertices to infinity and enqueue all
        # other vertices with key infinity
        for vertex in self.adjacencyList.keys():
            previousEdge[vertex] = None
            if vertex != start:
                distances[vertex] = math.inf
                distanceQueue.enqueue(QueueItem(vertex, math.inf))

```

Figure 3.14 Initialization (Djikstra's algorithm)

As shown in Figure 3.14, the Djikstra() function takes one input parameter named start, which is the starting vertex of Djikstra's algorithm. Firstly, two empty dictionaries, namely distances and previousEdge, are initialized. The distances dictionary stores the shortest distance from the start vertex to all other vertices, whereas the previousEdge dictionary stores the previous edge of each vertex in the shortest path. Besides that, a priority queue object named distanceQueue is initialized. The distance queue object is a priority queue that stores all vertices with their key and outputs the vertex with the smallest key every time when dequeue() is called.

As part of initialization of Djikstra’s algorithm, the shortest distance to the start vertex is set to 0 and the start vertex is enqueued to the priority queue with key 0. The shortest distance of all other vertices is set to positive infinity, which is represented by “math.inf” from the Math library. All other vertices are also enqueued into the priority queue with key infinity. Besides that, the previous edge of each vertex is set to None.

After initialization, the search for the shortest path starts. Figure 3.15 depicts the code for finding the shortest path using Djikstra’s algorithm.



```
        # Keep on repeating until the priority queue is empty
        while not distanceQueue.isQueueEmpty():
            # Dequeue the vertex with the smallest key (distance) and use it as the source vertex
            smallestKeyItem = distanceQueue.dequeue()
            source = smallestKeyItem.vertex

            # Go for edge going out from the source vertex
            for edge in self.adjacencyList[source]:
                destination = edge.destination
                weight = edge.weight

                # If a shorter distance is found, update distance and previous edge and update the key of the vertex in
                # the priority queue
                if distanceQueue.isVertexInQueue(destination) and distances[source] + weight < distances[destination]:
                    distances[destination] = distances[source] + weight
                    previousEdge[destination] = edge
                    distanceQueue.reduceKey(destination, distances[destination])
```

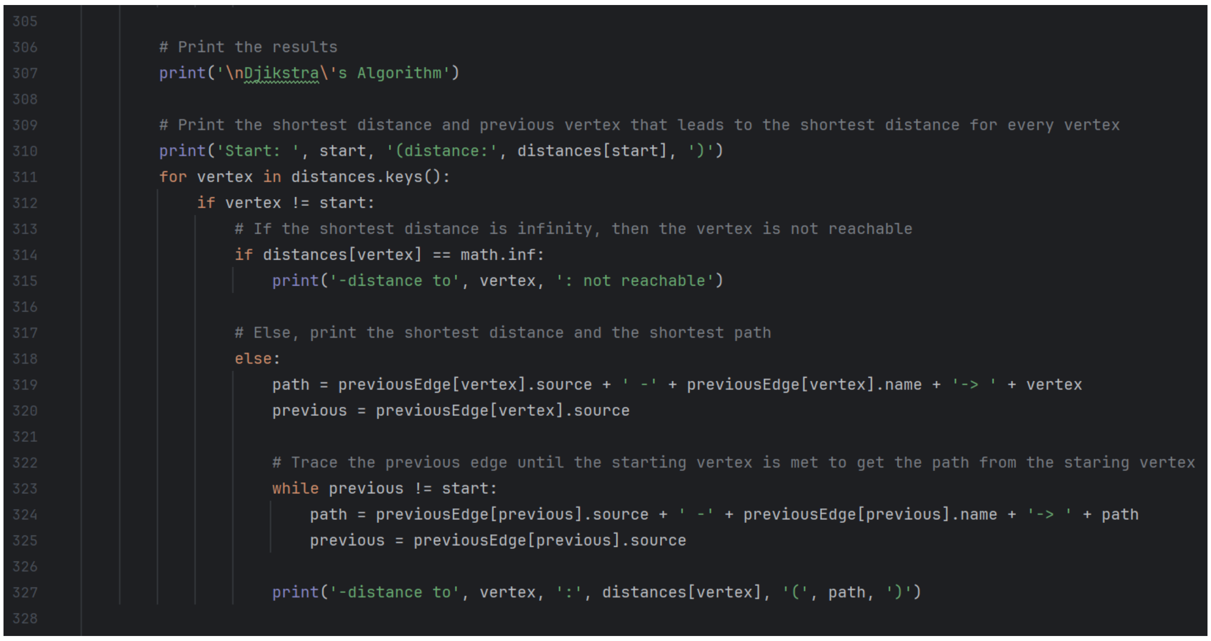
Figure 3.15 Find the shortest path (Djikstra's algorithm)

As shown in Figure 3.15, the programme runs as long as the priority queue is not empty. In ither words, the programme runs for as many iterations as the number of items in the priority queue. Since all vertices exists in the priority queue for one instance, the number of items in the priority queue is equal to the number of vertices.

In each iteration, the queue item with smallest key is dequeued and the vertex stored in that item is set as the source vertex in that iteration. Every outgoing edge from the source vertex is examined. If the destination vertex of the edge is still in the priority queue and the sum of the weight of the edge and the shortest distance of the source vertex of the edge is smaller than the shortest distance of the destination vertex, then the shortest distance of the destination vertex is updated to the sum of the weight of the edge and the shortest distance of the source vertex. The distances dictionary and the previousEdge dictionary are updated to reflect the changes. The recudeKey() function of the priority queue is called by specifying the name of the destination vertex and the updated distance.

3.3.3 Print results

As the final step of Djikstra’s algorithm, the shortest distance and the shortest path from the start vertex to all other vertices are printed. Figure 3.16 shows the code for printing results for Djikstra’s algorithm.



```python
        # Print the results
        print('\nDjikstra\'s Algorithm')

        # Print the shortest distance and previous vertex that leads to the shortest distance for every vertex
        print('Start: ', start, '(distance:', distances[start], ')')
        for vertex in distances.keys():
            if vertex != start:
                # If the shortest distance is infinity, then the vertex is not reachable
                if distances[vertex] == math.inf:
                    print('-distance to', vertex, ': not reachable')

                # Else, print the shortest distance and the shortest path
                else:
                    path = previousEdge[vertex].source + ' -' + previousEdge[vertex].name + '-> ' + vertex
                    previous = previousEdge[vertex].source

                    # Trace the previous edge until the starting vertex is met to get the path from the staring vertex
                    while previous != start:
                        path = previousEdge[previous].source + ' -' + previousEdge[previous].name + '-> ' + path
                        previous = previousEdge[previous].source

                    print('-distance to', vertex, ':', distances[vertex], '(', path, ')')
```

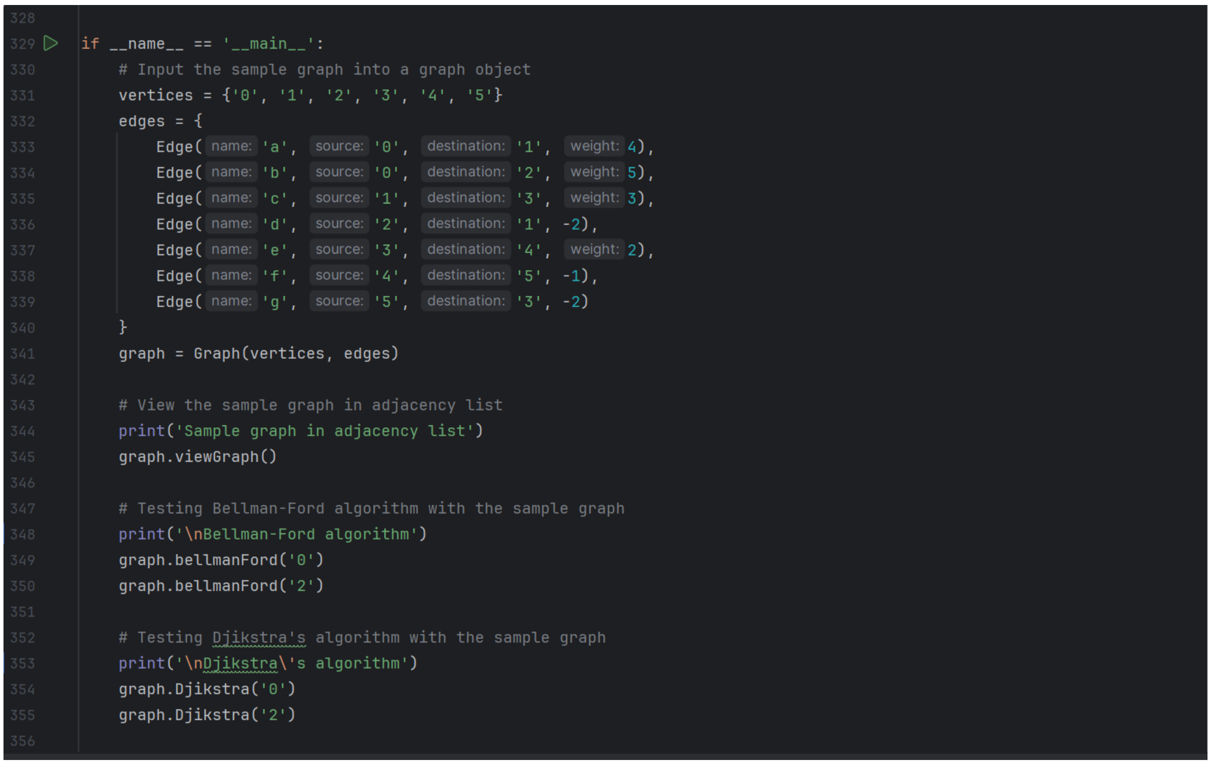
Figure 3.16 Print results (Djikstra's algorithm)

As shown in Figure 3.16, firstly, the programme prints the shortest distance from the start vertex to the start vertex. After that, the programme loops through the list of keys of the distances dictionary, which is equivalent to the list of vertices, to print the shortest distance of each vertex.

For each vertex, if the shortest distance to the vertex is infinity, then the programme will state that the vertex is not reachable. Otherwise, the programme will trace the shortest path from the start vertex to the vertex. The name of the vertex, the name of its previous edge and the source vertex of its previous edge are recorded in a variable named path. After that, a variable named previous is assigned to point to the source vertex of the previous edge of the currently examined vertex. Then, the name of the source vertex of the previous edge of the vertex previous and the name of the previous edge of the vertex previous are added to the variable path. After that, the variable previous is pointed to the source vertex of the previous edge of the vertex previous to go up to the previous level of the shortest path. This process is repeated until the variable previous points to the start vertex. Then, the shortest distance of the vertex and the shortest path from the variable path is printed.

4 Test results

The test results after running the programme are screenshotted and shown in this section. Figure 4.1 depicts the script to test run the display the sample graph, find the shortest path and detect negative cycle using Bellman-Ford algorithm and find the shortest path using Djikstra's algorithm.



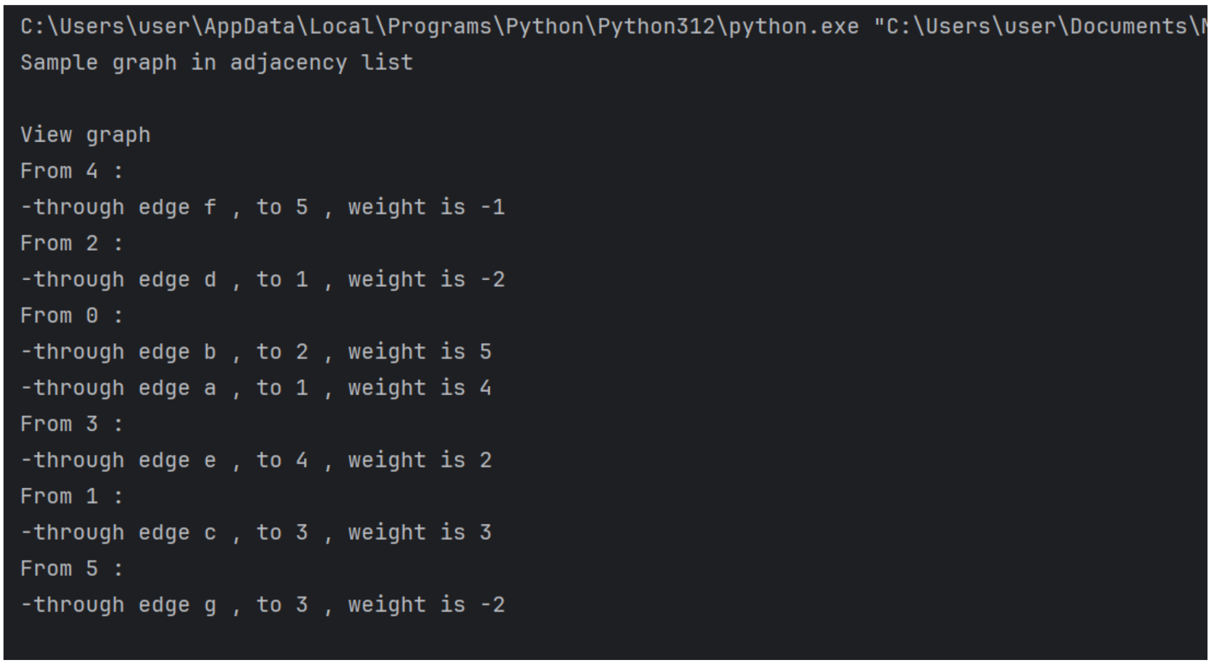
```python
if __name__ == '__main__':
    # Input the sample graph into a graph object
    vertices = {'0', '1', '2', '3', '4', '5'}
    edges = {
        Edge( name: 'a',  source: '0',  destination: '1',  weight: 4),
        Edge( name: 'b',  source: '0',  destination: '2',  weight: 5),
        Edge( name: 'c',  source: '1',  destination: '3',  weight: 3),
        Edge( name: 'd',  source: '2',  destination: '1', -2),
        Edge( name: 'e',  source: '3',  destination: '4',  weight: 2),
        Edge( name: 'f',  source: '4',  destination: '5', -1),
        Edge( name: 'g',  source: '5',  destination: '3', -2)
    }
    graph = Graph(vertices, edges)

    # View the sample graph in adjacency list
    print('Sample graph in adjacency list')
    graph.viewGraph()

    # Testing Bellman-Ford algorithm with the sample graph
    print('\nBellman-Ford algorithm')
    graph.bellmanFord('0')
    graph.bellmanFord('2')

    # Testing Djikstra's algorithm with the sample graph
    print('\nDjikstra\'s algorithm')
    graph.Djikstra('0')
    graph.Djikstra('2')
```

Figure 4.1 Test run script

As shown in Figure 4.1, the list of the six vertices in the sample graph, namely "0", "1", "2", "3", "4" and "5", and the seven edges in the sample graph are inputted into the constructor of a graph object. The seven edges in the sample graph does not have names, but in this report, for the ease of referring to these edges, the edges are named as “a”, ‘b”, “c”, “d”, “e”, “f” and “g” respectively. Then, the viewGraph() is called to display the sample graph in adjacency list.

After that, the bellmanFord() function is called twice, to find the shortest path to all vertices from vertex “0” and vertex “2” respectively, using Bellman-Ford algorithm. The bellmanFord() function also detects and traces the negative cycles using Bellman-Ford algorithm. Finally, the Djikstra() function is also called twice, to find the shortest path to all vertices from vertex “0” and vertex “2” respectively, using Djikstra’s algorithm.

Figure 4.2 is the test result of displaying the sample graph as adjacency list by calling the viewGraph() function in the graph class.

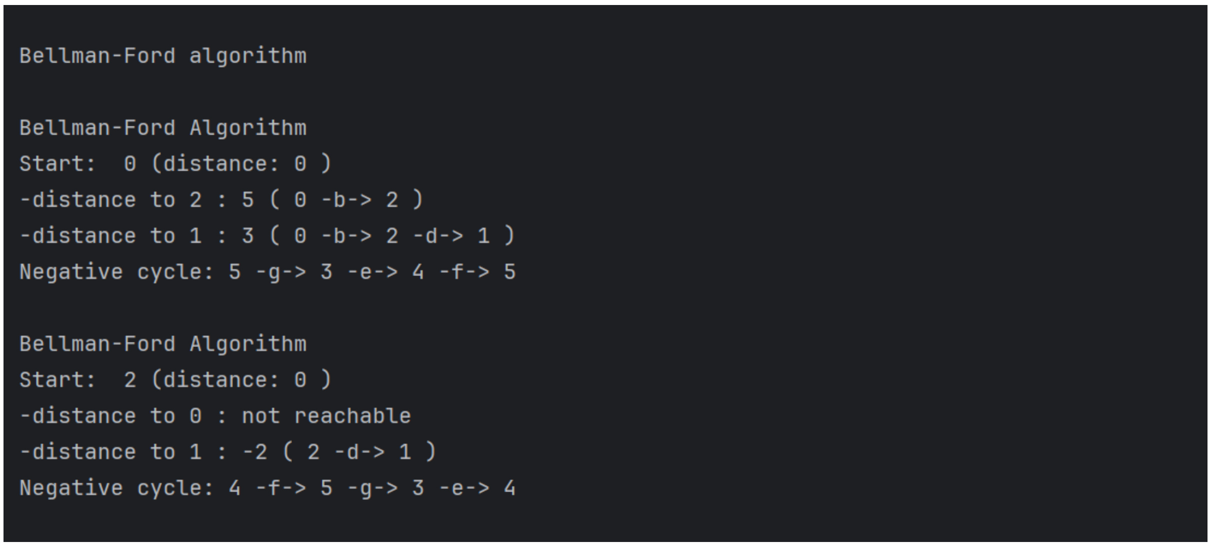


```
C:\Users\user\AppData\Local\Programs\Python\Python312\python.exe "C:\Users\user\Documents\M
Sample graph in adjacency list

View graph
From 4 :
-through edge f , to 5 , weight is -1
From 2 :
-through edge d , to 1 , weight is -2
From 0 :
-through edge b , to 2 , weight is 5
-through edge a , to 1 , weight is 4
From 3 :
-through edge e , to 4 , weight is 2
From 1 :
-through edge c , to 3 , weight is 3
From 5 :
-through edge g , to 3 , weight is -2
```

Figure 4.2 View graph (test results)

As shown in Figure 4.2, the sample graph is displayed as adjacency list. The print operation is performed by the code shown in Figure 3.3. For each vertex, all outgoing edges from that vertex are listed. The information displayed includes the name of the destination vertex of the outgoing edge, the name of the outgoing edge and the weight of the outgoing edge. After checking, it is confirmed that the graph structure printed is the same as the structure of the sample graph.

Figure 4.3 is the test result of finding the smallest path and detect negative cycle using Bellman-Ford algorithm. The algorithm is run two times with different start vertex, namely vertex ‘0’ and vertex ‘2’.



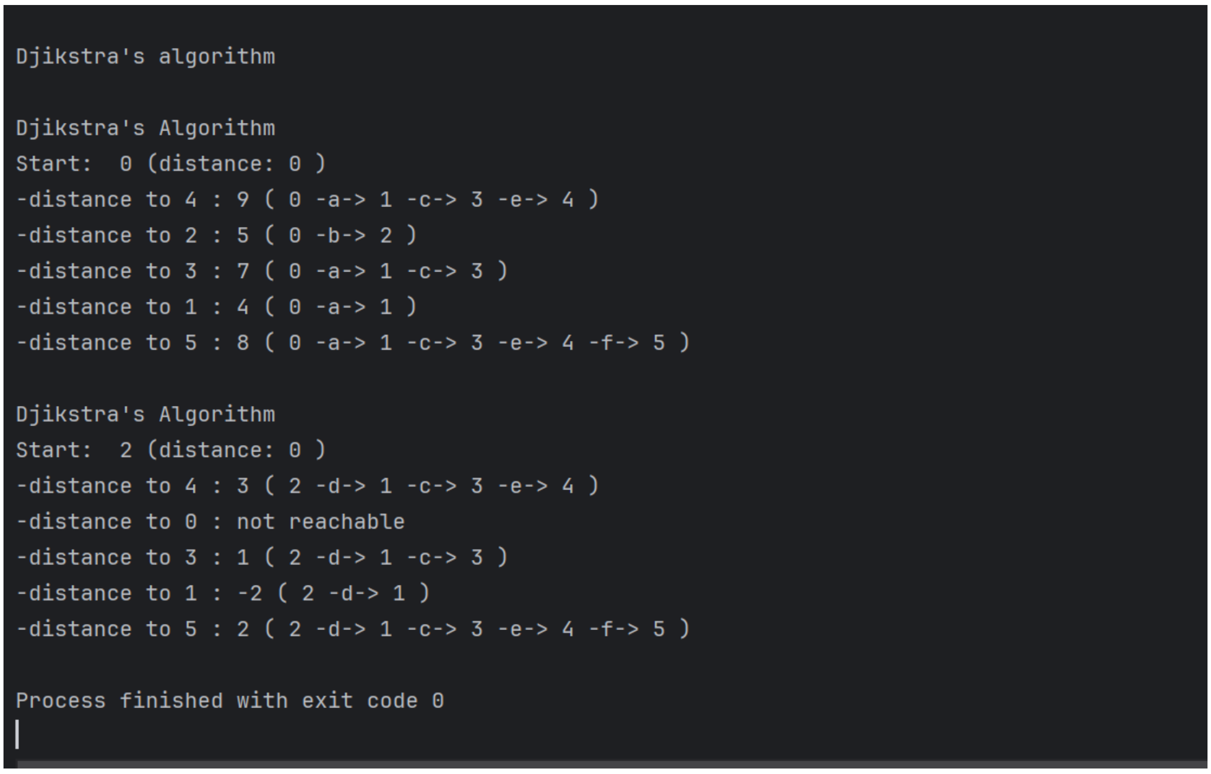
```
Bellman-Ford algorithm

Bellman-Ford Algorithm
Start:  0 (distance: 0 )
-distance to 2 : 5 ( 0 -b-> 2 )
-distance to 1 : 3 ( 0 -b-> 2 -d-> 1 )
Negative cycle: 5 -g-> 3 -e-> 4 -f-> 5

Bellman-Ford Algorithm
Start:  2 (distance: 0 )
-distance to 0 : not reachable
-distance to 1 : -2 ( 2 -d-> 1 )
Negative cycle: 4 -f-> 5 -g-> 3 -e-> 4
```

Figure 4.3 Bellman-Ford algorithm (test results)

As shown in Figure 4.3, the shortest distance and the shortest path to every other vertex from vertex ‘0’ and vertex ‘2’ respectively, are printed. By starting from vertex ‘0’, the algorithm found that the shortest distance to vertex ‘2’ is 5 through the edge from vertex ‘0’ to vertex ‘2’. The algorithm also found that the shortest distance to vertex ‘1’ is 3 via a path that goes from vertex ‘0’ to vertex ‘2’, then from vertex ‘2’ to vertex ‘1’. After manually checking the sample graph, it is found that the shortest distance and the shortest path for both vertex ‘2’ and vertex ‘1’ are correct. Notably, the path from vertex ‘0’ to vertex ‘1’ via vertex ‘2’ has shorter distance than the path from vertex ‘0’ to vertex ‘2’ because the edge from vertex ‘2’ to vertex ‘1’ has a negative weight of -2, which reduces the total distance. This shows that Bellman-Ford algorithm is able to handle negative weight correctly. The algorithm found a negative cycle, which is the cycle of vertex ‘5’, vertex ‘3’ and vertex ‘4’. After checking the sample graph, it is found that the cycle is a negative cycle, with the sum of weights in the cycle of -1. So, Bellman-Ford algorithm is correct at detecting and tracing negative cycle.

On the other hand, by starting from vertex ‘2’, the algorithm found that vertex ‘0’ is not reachable, whereas the distance to vertex ‘1’ is -2 through the edge from vertex ‘2’ to vertex ‘1’. After manually checking the sample graph, it is found that the results are correct. Notably, although vertex ‘2’ is reachable from vertex ‘0’ (refer to the previous run), but vertex ‘0’ is not reachable from vertex ‘2’ because the edge from vertex ‘0’ to vertex ‘2’ is in one direction only. The algorithm also found a negative cycle, which is the cycle of vertex ‘4’, vertex ‘5’ and vertex ‘3’. This is the correct negative cycle. So, the test results of both runs show that Bellman-Ford algorithm is able to detect shortest path and shortest distance correctly in a graph that contains negative weight and detect and trace the negative cycle correctly if the graph contains a negative cycle.

Figure 4.4 is the test result of running Djikstra's algorithm on the sample graph with two different start vertices, namely vertex '0' and vertex '2'. This is the same start vertices with the previous runs of Bellman-Ford algorithm.



```
Djikstra's algorithm

Djikstra's Algorithm
Start:  0 (distance: 0 )
-distance to 4 : 9 ( 0 -a-> 1 -c-> 3 -e-> 4 )
-distance to 2 : 5 ( 0 -b-> 2 )
-distance to 3 : 7 ( 0 -a-> 1 -c-> 3 )
-distance to 1 : 4 ( 0 -a-> 1 )
-distance to 5 : 8 ( 0 -a-> 1 -c-> 3 -e-> 4 -f-> 5 )

Djikstra's Algorithm
Start:  2 (distance: 0 )
-distance to 4 : 3 ( 2 -d-> 1 -c-> 3 -e-> 4 )
-distance to 0 : not reachable
-distance to 3 : 1 ( 2 -d-> 1 -c-> 3 )
-distance to 1 : -2 ( 2 -d-> 1 )
-distance to 5 : 2 ( 2 -d-> 1 -c-> 3 -e-> 4 -f-> 5 )

Process finished with exit code 0
```

Figure 4.4 Djikstra's algorithm (test results)

As shown in Figure 4.4, the shortest distance and the shortest path to every vertex from vertex '0' and vertex '2' are printed. By starting from vertex '0', the algorithm found that the shortest distance to vertex '4' is 7, whereas the shortest distance to vertex '2' is 5, whereas the shortest distance to vertex '3' is 7, whereas the shortest distance to vertex '1' is 4, whereas the shortest distance to vertex '5' is 8. After manually checking the sample graph, it is found that the shortest distance to vertex '1' is not correct because the correct shortest distance is 3 through the path from vertex '0' to vertex '2' (weight 5) and from vertex '2' to vertex '1' (weight -2). This shows that Djikstra's algorithm sometimes fail to handle negative weight edge. The shortest distance to other vertices is correct if the path is a simple path without cycle. However, the algorithm fails to detect the negative cycle of vertex ‘3’, vertex ‘4’ and vertex ‘5’. This shows that Djisktra’s algorithm is not able to detect negative cycle.

On the other hand, by starting from vertex ‘2’, the algorithm found that the shortest distance to vertex ‘4’ is 3, whereas the shortest distance to vertex ‘3’ is 1, whereas the shortest distance to vertex ‘1’ is -2, whereas the shortest distance to vertex ‘5’ is 2. The algorithm also found that vertex ‘0’ is not reachable. After manually checking the sample graph, it is found that the interpretation is correct if the path is a simple path without cycle. However, the algorithm fails to detect the negative cycle of vertex ‘3’, vertex ‘4’ and vertex ‘5’. So, considering the test results of both runs, it can be concluded that sometimes Djikstra’s algorithm can give wrong result when there are negative weights in the graph and Djikstra’s algorithm cannot detect negative cycle in the graph.

5 Comparison with Djikstra

In this section, Bellman-Ford algorithm is compared with Djikstra’s algorithm in two aspects, namely the ability to handle negative weight edge and the time complexity.

5.1 Ability to handle negative weight edge

From the test result, it is found that Bellman-Ford algorithm is able to handle negative weight correctly because Bellman-Ford algorithm successfully found the correct shortest distance and shortest path in both test runs on the sample graph, which contains some negative weight edges. Contrarily, from the test result, it is found that Djikstra’s algorithm is not able to handle negative weights correctly because Djikstra’s algorithm gave the wrong shortest distance and shortest path from vertex ‘0’ to vertex ‘1’. So, Djikstra’s algorithm is not suitable to be used for the sample graph because the sample graph contains negative weights.

Djikstra’s algorithm fails in finding the shortest path and the shortest distance from vertex ‘0’ to vertex ‘1’ because the actual shortest path contains negative weight. Djikstra’s algorithm is based on the greedy approach, which chooses the locally optimum solution in each step. Djikstra’s algorithm assumes that the vertices are added in ascending order of shortest distance. However, the existence of negative weight makes this assumption to be violated sometimes. Using the example of the shortest path from vertex ‘0’ to vertex ‘1’, by starting from vertex ‘0’, the algorithm firstly relaxes the outgoing edges from vertex ‘0’. So, the algorithm updates the shortest distance to vertex ‘1’ to 4 and the shortest distance to vertex ‘2’ to 5. Then, the algorithm chooses the vertex with the shortest distance to be included into the cloud of shortest path. So, vertex ‘1’ is chosen and the shortest distance to vertex ‘1’ is finalized at 4. Since vertex ‘1’ is no longer in the priority queue, the algorithm will never relax the edge whose destination is vertex ‘1’ in the future iterations. This causes the algorithm to miss the actual shortest path from vertex ‘0’ to vertex ‘1’, which is going from vertex ‘0’ to vertex ‘2’ (distance is 5) and going from vertex ‘2’ to vertex ‘1’ (distance is -2). The actual shortest path results in a distance of 3, which is shorter than the shortest distance finalized earlier by the algorithm. The algorithm missed the actual shortest path because the actual shortest path first goes through a path with longer distance (5) than the locally shortest distance (4) and undergoes a reduction of distance through a negative weight edge (-2). The greedy approach causes the algorithm to choose the locally shortest distance whenever possible. As shown in the example, this property sometimes causes Djikstra’s algorithm to miss the shortest path with a negative weight edge.

Besides that, Djikstra’s algorithm cannot be modified to detect negative cycle. When handling the sample graph, Djikstra’s algorithm searched for the shortest distance and the shortest path by assuming there is no negative cycle. Contrarily, Bellman-Ford algorithm can be modified to detect and trace negative cycle. When handling the sample graph, the modified Bellman-Ford algorithm detected, traced and reported the negative cycle, which is the cycle of vertex ‘3’, vertex ‘4’ and vertex ‘5’. The modified Bellman-Ford algorithm detects negative cycle by running another iteration after n-1 iterations (n is the number of vertices). After n-1 iterations, if there is an edge which after relaxed will result in shorter distance, then the edge is in a negative cycle. After detecting negative cycle, the modified Bellman-Ford algorithm traces the cycle by backtracking the previous edge until it meets with the destination vertex of the original edge again.

5.2 Time complexity

The time complexity of Bellman-Ford algorithm is $O(n \times m)$, where n is the number of vertices and m is the number of edges. Setting the shortest distance to the start vertex to 0 and all other vertices to infinity takes $O(n)$ time because there are n vertices. Setting the previous edge of all vertices to None takes another $O(n)$ time because there are n vertices. The search of the shortest edges and the shortest path is run for $n - 1$ iterations. In each iteration, all edges are examined once and there are m edges. For each edge, the name of the source vertex of the edge, the name of the destination vertex of the edge and the weight of the edge are retrieved from the object of edge class. Each of these operations takes $O(1)$ time. The distance to the source edge and the distance to the destination edge are retrieved from the distances dictionary using the name of the source vertex and the name of the destination vertex. Dictionary check takes $O(1)$ time. The sum of the distance to the source vertex and the weight of the edge is compared with the distance to the destination vertex. This comparison takes $O(1)$ time. If the comparison returns true, then the distance to the destination vertex and the previous edge of the destination vertex are updated in the distances dictionary and the previous edge dictionary respectively. Updating the value in a dictionary by referring the key, which is the name of the vertex, takes $O(1)$ time. So, in overall, the process of searching for the shortest distance and the shortest path from the start vertex to every other vertex takes $O(n \times m)$ time because it passes through $n - 1$ iterations and m edges for each iteration. As explained previously, all operations for processing each edge takes $O(1)$ time. For the negative cycle detection and tracing, it repeats one more iteration of the process of the search for the shortest distance and the shortest path. If a negative cycle is found, then the tracing of the negative cycle takes $O(size\ of\ negative\ cycle)$ time because the tracing process passes each vertex and each edge in the negative cycle. So, in the worst case, a negative cycle is found for each edge and the negative cycle includes every vertex in the graph. So, in the worst case, the negative cycle detection and tracing takes $O(m \times n)$ time because it passes through m edges and at each edge it finds a cycle with size n. So, considering both shortest distance and shortest path searching and negative cycle detection and tracing, the time complexity of Bellman-Ford algorithm is $O(n + n + (n \times m)) + O(m \times n)$, which can be simplified as $O(n \times m)$.

On the other hand, the time complexity of Djikstra's algorithm is $O((\mathrm{n}+\mathrm{m}) \times \log n)$, where n is the number of vertices and m is the number of edges. Setting the shortest distance to the start vertex to 0 and all other vertices to infinity takes $O(n)$ time because there are n vertices. Setting the previous edge of all vertices to None takes another $O(n)$ time because there are n vertices. The priority queue used in the source code is using min heap data structure. In the worst case, when enqueuing one vertex with its key into the priority queue, the vertex starts from the bottom rightmost leaf and continuingly swaps with its parent until it reaches the root. This time for this swapping is proportional to the height of the heap tree, which is equal to $\log n.$ So, enqueuing one vertex with its key into the priority queue takes $O(\log n)$ time and enqueuing all vertices into the priority queue takes $O(n \times \log n)$ time because there is n vertices and each vertex is enqueued once. When dequeuing a vertex, the root is swapped with the bottom rightmost leaf and removed. The new root then swaps with its child to re-establish the min heap property. In the worst case, the swapping is done from the root until the leaf, resulting a time of swapping which is proportional to the height of the tree, which is $\log n$. So, dequeuing one vertex takes $O(\log n)$ time and dequeuing every vertex takes $O(\mathrm{n} \times \log n)$ time because there is n vertices and each vertex is dequeued once. For every dequeued vertex, each of its outgoing edge is examined. For each outgoing edge, the name of the destination vertex of the edge and the weight of the edge are retrieved from the object of edge class. Each of these operations takes $O(1)$ time. The programme checks whether the destination vertex is in the priority queue by checking whether the name of the vertex is one of the leys of vertex index dictionary in the priority queue object. Checking whether a key exists in a dictionary using in operator takes $O(1)$ time. The shortest distance to the source vertex and the shortest distance to the destination vertex are retrieved from the distances dictionary using the name of the vertex as the key. Getting a value from dictionary by referring a key takes $O(1)$ time. The sum of the shortest distance to the source vertex of the edge and the weight of the edge is compared with the shortest distance of the destination vertex of the edge. This comparison takes $O(1)$ time. If the comparison returns true, then the shortest distance to the destination vertex is updated in the distances dictionary and the previous edge of the destination vertex is updated in the previous edge dictionary. Updating the value in a dictionary by specifying the key takes $O(1)$ time. The key of the destination vertex in the priority queue is also updated. When updating the key, the reduceKey() function checks the index of the vertex in the queue by checking the vertex index dictionary in the priority queue object by specifying the key, which is the name of the vertex. This takes $O(1)$ time. A reduction to the key of any vertex might involve swapping of the vertex with its parent if the key of the vertex becomes smaller than the key of its parent. In the worst case, the vertex is at the leaf and the vertex is continuingly swapped with its parent until the root. This takes a time which is proportional to the height of the tree, which is $\log n$. So, reducing the key of one vertex in the priority queue takes $O(\log n)$ time. For every vertex, the key is changed for at most $\deg(v)$ times, where $\deg(v)$ is the number of in-degrees of the vertex v. For a directed graph, the sum of in-degrees of all vertex, $\sum \deg(v)$, is equal to the number of edges, m. So, for the worst case, the time complexity for reducing the key of all vertices in the priority queue is $O(m \times \log n)$. So, considering everything, the time complexity of Djikstra's algorithm is $O(n + n + (n \times \log n) + (n \times \log n) + (m \times \log n))$, which can be simplified to $O((n \times \log n) + (m \times \log n)) = O((n + m) \times \log n)$.

In summary, the time complexity of Djikstra's algorithm ($O((n + m) \times \log n)$) is lower than that of Bellman-Ford algorithm ($O(n \times m)$). So, although Djikstra's algorithm faces problem in handling negative weight edges, but in terms of performance, Djikstra's algorithm is clearly more efficient than Bellman-Ford algorithm.

6 Conclusion

After analysing the simulation of Bellman-Ford algorithm and Djikstra's algorithm on a sample graph with a few negative weight edges and a negative cycle, it can be concluded that Djikstra's algorithm cannot handle negative weight edge, whereas Bellman-Ford algorithm can. This is backed by the result of the simulation, where Djikstra's algorithm gave a shortest distance and a shortest path that is not the shortest from vertex '0' to vertex '1' in the sample graph. After investigation, it is found that the reason is there is a negative edge weight in the actual shortest path. Djikstra's algorithm missed that path because it straight away takes the shorter positive weight edge, but the reality is the actual path goes with a longer positive weight edge first, then goes with a negative weight edge, which significantly decreases the distance. On the other hand, Bellman-Ford algorithm re-considers every edge for n-1 iterations, where n is the number of vertices. This ensures that all combinations of the edges are considered in the process of searching for the shortest distance and the shortest path. This enables Bellman-Ford algorithm to handle negative weight edges. This is proven by the fact that in the simulation, Bellman-Ford algorithm found all shortest distance and shortest path correctly.

Moreover, by running one more iteration, Bellman-Ford algorithm can detect any negative cycle in the graph, whereas Djikstra's algorithm cannot. In Bellman-Ford algorithm, if a shorter distance can still be found after n-1 iterations, then there is a negative cycle. This is supported by the observation in the simulation, where Bellman-Ford algorithm detected, traced and reported the negative cycle correctly in all two runs which used different vertices as the start vertex.

However, the superiority of Bellman-Ford algorithm comes with a cost in its higher time complexity. After careful analysis on the source code, it is found that the time complexity of Bellman-Ford algorithm is $O(n \times m)$, which is higher than Djikstra's algorithm's $O((n + m) \times \log n)$. In other words, for a dense and complex graph with many vertices and many edges, the processing time needed by Bellman-Ford algorithm will be significantly longer than that of Djikstra's algorithm.

As a closing note for this report, it can be concluded that among the two algorithms, Djikstra's algorithm has better performance than Bellman-Ford algorithm in finding the shortest distance and the shortest path from a start vertex to all other vertices, but Djikstra's algorithm can only handle graphs with all positive weight edge. So, if the graph has all positive weight, then it is recommended to use Djikstra's algorithm. On the other hand, Bellman-Ford algorithm still has its unique value, which lies on its ability to handle graph with negative weight edge and detect negative cycles in the graph. So, if the graph has some negative edge, then it is recommended to use Bellman-Ford algorithm.