Problem 1 Present full pseudocode of a variant of Prim's algorithm that runs in time $O(|V|^2)$ for a graph $G = (V, E)$ given by an adjacency matrix A. Analyze the running time.

Note: do not use any procedure or data structures operation.

```
"""
      Supporting Class for the prim's algorithm to work.
"""

class Graph:
   def __init__(self, vertices):
       self.vertices = vertices
       self.adjacency_matrix = [[0 for _ in range(0, vertices)]
                                for _ in range(0, vertices)]

"""
      Following prims_mst algorithm prints the minimum edge between
two nodes and weight of the minimum spanning tree.

   1. from line 2 to 4, 9 to 10 are constant time c1.
   2. from line 5 to 8 we have for loop and inside we have constant
      time assignments c2 so time complexity of the for loop will
      be O(V * c2), which is equal to O(V).
   3. from line 11 to 23 we have nested for loop in which time
      complexity is as follows:-
         a.  at line 13  we have a for loop inside which retrieves
            minimum distance edge vertex in time complexity of O(V).
         b. at line 20 we have another for loop which contains
            constant time operations and runs for V time so it's
            time complexity is O(V).
         c. Other operations than two for loops are constant time.
      So time complexity of the nested for loop is O(V * V) which
is O(V^2).

So the total time complexity of the algorithm will be O(c1 + V +
V^2). Which is equivalent to O(V^2).

Hence the total time complexity of the algorithm is O(c1 + V + c2 + V^2),
which is equal to O(V^2).

"""
1     def prim_mst(graph: Graph, source: int):
2        key = []
3        parent = []
4        mst_set = []
5        for vertex in range(0, graph.vertices):
6            key.append(sys.maxsize)
7            parent.append(None)
8            mst_set.append(False)
```

```
9       key[source] = 0

10      print("Edge     ", "Weight")
11      for vertex in range(graph.vertices):

/** ----------- Start Equivalent to Extract Min
-----------------**/
12          minimum = sys.maxsize
13          for i in range(0, graph.vertices):
14              if mst_set[i] is False and key[i] < minimum:
15                  minimum = key[i]
16                  u = i

/** ----------- End Equivalent to Extract Min -----------------**/

17          mst_set[u] = True

18          if parent[u] is not None:
19              print(parent[u], " - ", u, "    ",
                          graph.adjacency_matrix[u][parent[u]])

20          for v in range(graph.vertices):
21              if 0 < graph.adjacency_matrix[u][v] < key[v] and
                                        mst_set[v] is False:
22                  key[v] = graph.adjacency_matrix[u][v]
23                  parent[v] = u
```

Problem 2 Give an example of a weighted directed graph $\vec{G}$ with negative-weight edges, but no negative-weight cycle, such that Dijkstra's algorithm incorrectly computes the shortest-path distances from some start vertex v. Use the algorithm version from the handout.

A four-vertex example is possible. Draw the graph, mention the start vertex, show the result of Dijkstra's algorithm, and point out for which vertex the result is incorrect.

(This was on a previous final exam)

Solution

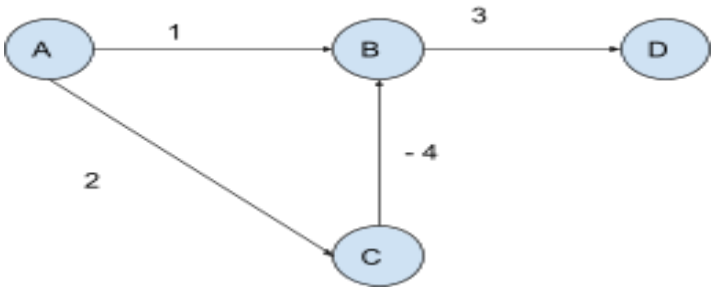
Dijkstra's algorithm given is as following:-

Dijkstra's algorithm for single-source shortest paths in directed graphs with non-negative weight

DIJKSTRA(G,w,r)

1. **for** each vertex $u \in V[G]$
2. **do** $d[u] \longleftarrow \infty$
3. $\pi[u] \longleftarrow NIL$
4. $d[r] \longleftarrow 0$
5. $Q \longleftarrow V[G]$
6. **while** $Q \neq \emptyset$
7. **do** $u \longleftarrow$ EXTRACT-MIN(Q)
8. **for** each $v \in Adj[u]$
9. **do if** $d[u] + w(u,v) < d[v]$ // This is called "Relax($u, v, w()$)"
10. **then** $\pi[v] \longleftarrow u$
11. $d[v] \longleftarrow d[u] + w(u,v)$
12. DECREASE-KEY($Q, v, d[v]$)

Following is Graph with negative - weight edge, but no negative - weight cycle. Here Dijkstra's algorithm as above will incorrectly compute the shortest - path distances from start vertex a.



The above Algorithm will run as following:-

Initialization of priority queue and distances to every vertex from source vertex A.

Priority Queue	(A, 0), (B, infinite), (C, infinite), (D, infinite)			
Distances to vertices	A	B	C	D
	0	infinite	infinite	infinite

1st Iteration of while loop:-

Priority Queue	(B, 1), (C, 2), (D, infinite)			
u	A			
Distances from A vertices	A	B	C	D
	0	1	2	infinite

2nd Iteration of while loop:-

Priority Queue	(C, 2), (D,4)			
u	B			
Distances from A vertices	A	B	C	D
	0	1	2	4

3rd Iteration of while loop:-

Priority Queue	(D,4)			
u	C			
Distances from A to vertices	A	B	C	D
	0	-2	2	4

4th Iteration of while loop:-

Priority Queue				
u	D			
Distances from A to vertices	A	B	C	D
	0	-2	2	4

Now Priority Queue is empty so last distance from A to every vertex using Dijkstra's algorithm will be as following:-

Distances from A to vertices	A	B	C	D
	0	-2	2	4

Here Distance A to D is computed incorrectly as d(A, D) should be -2 + 3 = 1. But as the traversing of the weight update is done as follows:-

1. A, D
2. A, C
3. B, D
4. C, B

D was not able to get updated as B was already removed from the priority queue. So D can not be updated to it's real weight which is 1.

Problem 3 A *complete digraph* has exactly one directed edge (also called *arc*) from every vertex u to every vertex v other than itself. Let G be a complete digraph with non-negative arc weights. Let the *capacity* of a path be the *minimum* arc weight along it, and let the *capacity* of a pair of nodes (u, v) be the *maximum* capacity of a path from u to v. Find a Dijkstra-like algorithm to find, for all $v \neq s$, the capacity of (s, v). (Node s is a fixed source.)

Present the pseudocode, analyze the running time, and prove correctness.

Solution

```
"""
      This is helping class used for storing adjacent vertex of a
vertex in the graph and weight of the edge
"""
class Adjacent:
   def __init__(self, vertex: int, weight: int):
       self.vertex = vertex
       self.weight = weight

"""
      This is helping class used for storing graph and its contents
"""
class Graph:

   def __init__(self, vertices):
       self.vertices = vertices
       self.adjacent_vertices = defaultdict(list)

   def add_edge(self, u: int, v: int, weight: int):
       self.adjacent_vertices[u].append(Adjacent(v, weight))

"""
     maximum_minimum_capacity algorithm will give maximum capacity
from source to every other node. Where capacity is minimum arc
weight in the path.
```

Analysis of Running Time:-

1. capacity initialization line 2 time complexity O(V).
2. line 3, 5, 6 constant time complexity O(c1)
3. line 4 parent initialization time complexity O(V).
4. while loop at line 7
    a. line 8 O(logV) time complexity as there can be max V vertices in the priority queue.
    b. line 9 is constant time O(c2).
    c. inner for loop at line 10
        i. line 11 to 17 except 16 is constant time O(c3).
        ii. line 16 time complexity is O(logV) as there can be max E vertices in the priority queue in a time. In complete digraph E = V * (V - 1). As we know that insertion operation in priority queue leads to O(log n) where n is number of nodes in priority

queue. So here time complexity to put and get will be O(log V^2) which is equivalent to O(log V).

Total Time complexity of the for loop is O(len(adjacent_edges) * logV). The number of adjacent nodes in complete digraph will be V - 1. So time complexity will be O((V - 1)* (c2 + logV)), which is equal to O(V logV).

Time complexity of the while loop will be as following:- O((log V + c2 + V logV) * V). Multiplication of V because outer while loop will run for V times. So time complexity is O(V^2 log V) for the while loop.

Now total algorithm time complexity is O(V) + O(c1) + O(V) + O(V^2 log V). which is equal to O(V^2 log V).

This time complexity is equal to dijkstra's algorithm as we studied time complexity of E log V using binary heap. Here in complete digraph E = V * V - 1 as every vertex u is connected to every vertex v. So time complexity can still be shown as O(E log V).

```
"""
1     def maximum_minimum_arc_capacity(graph: Graph, source: int):
2        capacity = [-sys.maxsize for _ in range(graph.vertices)]
3        pq = PriorityQueue()
4        parent = [None for _ in range(graph.vertices)]

5        pq.put((0, source))
6        capacity[source] = sys.maxsize
7        while pq.empty() is not True:
8            vertex_tuple = pq.get()
9            u = vertex_tuple[1]

10           for adjacent in graph.adjacent_vertices[u]:
11               adjacent_vertex = adjacent.vertex
12               weight = adjacent.weight

13               max_capacity = max(capacity[adjacent_vertex], min(weight,
                                                                   capacity[u]))
14               if max_capacity > capacity[adjacent_vertex]:
15                   capacity[adjacent_vertex] = max_capacity
16                   pq.put((capacity[adjacent_vertex], adjacent_vertex))
17                   parent[adjacent_vertex] = u

18       print("Vertex   ", "Capacity")
19       for vertex in range(graph.vertices):
20           print(vertex, "     ", capacity[vertex])

21       return capacity
```

Correctness:-

Above algorithm first gets the minimum capacity vertex which is not traversed yet. To start the source is the maximum capacity vertex. Now it discovers the adjacent nodes and update the capacity for adjacent nodes as following:-

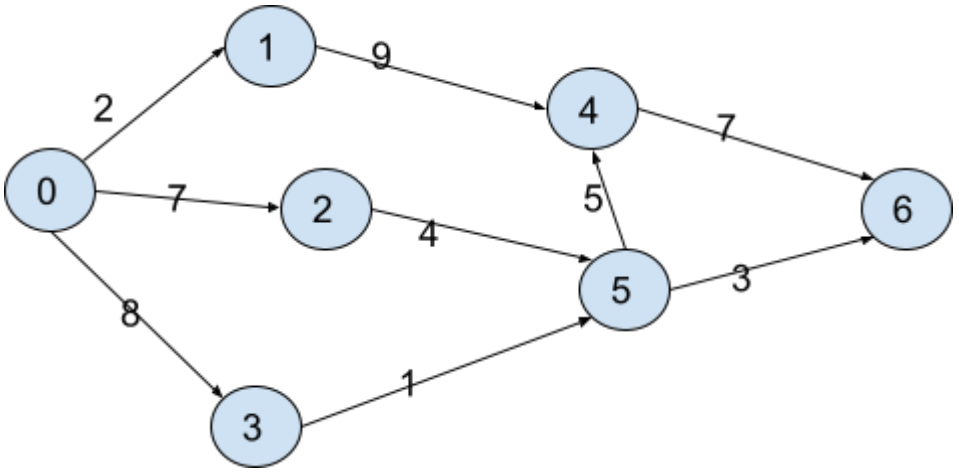
max(capacity[adjacent_vertex], min(weight, capacity[u]))

which is described as below:-

1. where capacity[adajacent_vertex] is previously calculated capacity.
2. As we want minimum arc weight along the path we will compare the current weight of edge to adjacent_vertex and capacity of u(vertex from which we are taking look at adjacent vertex).
3. max is used because we want to take the maximum of minimum arc weight between different paths we calculated. So we will update the capacity of vertex only if we find a capacity which is greater than previously calculated capacity of a vertex. This part is managed in line number 14 and 15.

Here instead of adding distance to the priority queue we are adding capacity. Using priority queue we first retrieve the minimum capacity vertex and calculate capacity of adjacent vertex as above method which always leads us to max capacity of minimum arc weight between different paths.

Example of Directional Graph **(Here is Graph is not complete Graph but will help to understand the correctness of algorithm but algorithm will still work on complete graph also)**:-



Let's say we want to calculate capacity between Source 0 and destination 6. Now path's from 0 to 6 is as following:-

	Path	Minimum Arc Weight
1.	0 -> 1 -> 4 -> 6	2
2.	0 -> 2 -> 5 -> 4 -> 6	4
3.	0 -> 2 -> 5 -> 6	3
4.	0 -> 3 -> 5 -> 6	1

So the maximum capacity of minimum arch weight between different paths from source 0 to 6 will be 4. So the path is maximum capacity path is as following :-
0 -> 2 -> 5 -> 4 -> 6 with maximum capacity = 4

How the above algorithm will calculate it.

Initialization
(a, b) in tuple a represent maximum capacity to the vertex and b is the vertex

Priority Q	(0, 0) # 0 is source here source can be different as required.						
	0	1	2	3	4	5	6
capacity	inf	-inf	-inf	-inf	-inf	-inf	-inf

1 iteration

u	0						
Priority Q	(2, 1) (7, 2) (8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	-inf	-inf	-inf

2 iteration

u	1						
Priority Q	(2, 4) (7, 2) (8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	2	-inf	-inf

3 iteration

u	4						
Priority Q	(2, 6) (7, 2) (8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	2	-inf	2

4 iteration

u	6						
Priority Q	(7, 2) (8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	2	-inf	2

5 iteration

u	2						
Priority Q	(4, 5) (8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	2	4	2

6 iteration

u	5						
Priority Q	(8, 3)						
	0	1	2	3	4	5	6

capacity	inf	2	7	8	2	4	2
7 iteration							
u	5						
Priority Q	(3, 6)(4, 4)(8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	4	4	3

8 iteration							
u	6						
Priority Q	(4, 4)(8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	4	4	3

9 iteration							
u	4						
Priority Q	(4, 6) (8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	4	4	4

10 iteration							
u	6						
Priority Q	(8, 3)						
	0	1	2	3	4	5	6
capacity	inf	2	7	8	4	4	4

10 iteration							
u	3						
Priority Q							
	0	1	2	3	4	5	6
capacity	inf	2	7	8	4	4	4

So maximum capacity among the minimum arc weight path calculated by the algorithm is 4 which is required by the problem. So that prove the correctness of the above algorithm.

Problem 4 Assume that the weights on the edges of directed graph G are integers in the range from 1 to K. Give a new method for implementing EXTRACT-MIN(Q) and DECREASE-KEY($Q, v, d[v]$) so that Dijkstra's algorithm's running time becomes $O(|E| + K|V|)$.

Describe the data structure and present the pseudocode for the two operations used by Dijkstra's algorithm, together with correctness and running-time analysis.

Solution

The Code of the algorithm is given as follows:-

```
"""
     This is helping class used for storing adjacent vertex of a
vertex in the graph and weight of the edge
"""
class Adjacent:
   def __init__(self, vertex: int, weight: int):
       self.vertex = vertex
       self.weight = weight

"""
     This is helping class used for storing graph and its contents
"""

class Graph:

   def __init__(self, vertices):
       self.vertices = vertices
       self.adjacent_vertices = defaultdict(list)

   def add_edge(self, u: int, v: int, weight: int):
       self.adjacent_vertices[u].append(Adjacent(v, weight))

"""
     K = max_weight
     r = source_vertex

     bucket_vertex_index and buckets are lists which are used here
as data structure to get time complexity of single source shortest
path algorithm in O(E + K * V).

Running Time Analysis:
   1. Line 2, 3 is constant time complexity O(c1).
```

2. Line 4, 6 is for a loop inside which is constant time operations and for loop runs for O(V) times so it's time complexity is O(V).
3. Line 7 have for loop for O(K * V) time so time complexity is O(K * V).
4. Line 8 to 11 is constant time complexity O(c2).
5. Line 12 while loop combined with inner while loop at line 13 will run for O(K * V) times as both depends on the length of the buckets which is O(K * V).
6. Line 12 while loop combined with inner for loop at line 13 to 27 will run for O(E) time complexity. The outer while loop will run for O(V) and every time the inner loop will run for adjacent edges. The total of adjacent edges is E. So the time complexity of both together will be O(E).
7. For loops at line 13 to 27 have all operations of constant time inside it.
8. Last for loop is O(V) for printing the distances from the source to each vertex.

Total time complexity = O(c1) + O(V) + O(K * V) + O(c2) + O(K*V) + O(E) + O(V).
= O(2 * V + 2 * (K * V) + E + c1 + c2)
= O(E + K * V)

"""

```
1    def single_source_shortest_path(graph: Graph, source_vertex:
int, max_weight: int):
2        distance = []

--   Used instead of binary heap (Start) --
3        bucket_vertex_index = []
4        for vertex in range(graph.vertices):
5            distance.append(sys.maxsize)
6            bucket_vertex_index.append(-1)

7        buckets = [[] for _ in range(graph.vertices * max_weight)]
--   Used instead of binary heap (End) --

8        distance[source_vertex] = 0
9        buckets[0].append(source_vertex)
10       bucket_vertex_index[source_vertex] = len(buckets[0]) - 1

11       bucket_index = 0
12       while True:

-- Equivalent to Extract Min Operation (Start) --
13           while bucket_index < graph.vertices * max_weight and
                                len(buckets[bucket_index]) == 0:
```

```
14              bucket_index += 1

15          if bucket_index == graph.vertices * max_weight:
16              break

17          min_distance_vertex = buckets[bucket_index].pop(0)
-- Equivalent to Extract Min Operation (End) --

18          for adjacent_vertex_edge in
                    graph.adjacent_vertices[min_distance_vertex]:
19              adjacent_vertex = adjacent_vertex_edge.vertex
20              adjacent_edge_weight = adjacent_vertex_edge.weight

21              if distance[min_distance_vertex] +
                adjacent_edge_weight < distance[adjacent_vertex]:

# we are doing this step from line 22 to 24 because if there was
#some value in linked list inside bucket of any vertex then that
is #going to be updated to different location new location and we
have #to remove it from the previous location.
22                  if distance[adjacent_vertex] != sys.maxsize:
#pop operation will be considered as O(1) time because we have
#location from where to pop value and also we are popping from
#linked list. Which gives us O(1) time of removal when we have
#location of the node.
23                      buckets[distance[adjacent_vertex]].
                          pop(bucket_vertex_index[adjacent_vertex]
                    )
24                      bucket_vertex_index[adjacent_vertex] = -1

25                  distance[adjacent_vertex] =
                distance[min_distance_vertex] +
            adjacent_edge_weight

-- Equivalent to Decrease Key Operation (Start) --
# appending the adjacent vertex at the end of linked list in
bucket # because priority of the vertex inserted before will be
more than #inserted later
26                  buckets[distance[adjacent_vertex]].
                                        append(adjacent_vertex)
27                  bucket_vertex_index[adjacent_vertex] =
                     len(buckets[distance[adjacent_vertex]]) - 1
-- Equivalent to Decrease Key Operation (End) --

28      for vertex in range(graph.vertices):
29          print(vertex, "     ", distance[vertex])
```

Correctness

Above algorithm after replacing extract min operation and decrease key operation is all similar to dijkstra's algorithm hence if replaced if operations are able to behave in similar fashion then it will be correct. So equivalency is proved as follows:-

1. Why extract min operation will be equivalent in above algorithm. In buckets every index of bucket represents distance from the source node and we are going from 0th index to (K * V) - 1 index. This helps us to find the index in which the linked list is present or say the bucket is not empty there. Now if the bucket is not empty then the linked list contains the minimum distance vertex from the source which is required by the extract min operation so both are equivalent.

2. Why decrease key operation is equivalent in the above algorithm. In the decrease key we update the new distance of vertex from the source vertex. Here if we find a new path to the vertex which has a lesser distance than we update its distance on the bucket index which is equivalent to the distance and remove previous distance from the bucket.

How Bucket Works Here and What Kind of Data Structure we are using here:-

The weights of the directed graph is bounded by K, so the maximum weight of a directed edge in the graph can be K. There is a V number of vertices in the graph. The upper bound for longest path from source to other node with these constraints can be K * (V - 1). So if we use array of size K * V to store the distance of the vertices. It will take O(K * V) time complexity for us to traverse all the weights of paths in the graph. So instead of using priority queue we can use two arrays as following:-

1. array of buckets:-
    a. Inside one bucket there will be a linked list of vertices. Here we are storing a linked list inside a bucket so that we can attain insertion and deletion operation in O(1) time.
    b. To attain O(1) time for insertion and deletion we must know where we put a vertex in the linked list which is inside the bucket. To know these locations we will use another array which is covered in the second point.
2. array of vertices location:-
    a. These locations represent locations inside the bucket in the linked list.