Project 2

Implementation of Dijkstra’s Algorithm

By: Kiefer, Potala, Yi Heng

A) Using Adjacency Matrix and Array for Priority Queue

```
def main():
  graph = generate_random_graph(10, 25)

  start_time = time.time()  # Start the timer.
  distances, comp = dijkstra(graph, 0)  # Run Dijkstra's algorithm.
  end_time = time.time() # Stop the timer.

  # Print the time it took to run the algorithm.
  print("Time to run Dijkstra's algorithm:", end_time - start_time)
  print("Number of key comparisons: ", comp)
```

Generates a random graph with 10 vertices and 25 edges

Record runtime for dijkstra, with input graph and starting vertex

Print runtime and number of key comparisons

A) Using Adjacency Matrix and Array for Priority Queue

```python
def generate_random_graph(n, m):

  graph = [[0 for i in range(n)] for j in range(n)]

  for i in range(m):
    while True:
        u = random.randint(0, n - 1)
        v = random.randint(0, n - 1)
        weight = random.randint(1, 100)
        if (graph[u][v] != 0):
            continue
        else:
            graph[u][v] = weight
            break

  return graph
```

Initialise graph

Ensures that no edges repeat (so that E accurate)

Assign if not repeated

A) Using Adjacency Matrix and Array for Priority Queue

```
[0, 10, 0, 0, 0, 56, 0, 99, 0, 53]
[0, 0, 53, 63, 0, 0, 7, 0, 25, 88]
[0, 2, 85, 23, 0, 29, 0, 16, 100, 31]
[72, 0, 29, 43, 0, 96, 95, 71, 41, 33]
[0, 0, 13, 0, 19, 0, 0, 75, 0, 92]
[45, 0, 0, 30, 44, 0, 0, 59, 0, 0]
[0, 4, 0, 81, 0, 0, 0, 0, 10, 0]
[43, 41, 0, 15, 0, 0, 0, 0, 3, 0]
[0, 29, 0, 4, 0, 0, 13, 20, 78, 0]
[89, 0, 0, 0, 0, 59, 22, 81, 22, 14]
```

Example of random graph

A) Using Adjacency Matrix and Array for Priority Queue

```
def dijkstra(graph, start):

    distance = [sys.maxsize] * len(graph)
    distance[start] = 0
    queue = [start]
    comparisons = 0
```

Initialise distance (shortest path), to infinity, except starting node

Initialise priority queue

Initialise key comparisons variable

A) Using Adjacency Matrix and Array for Priority Queue

```
while queue:
    # Find the node with the minimum distance in the queue
    min_dist = sys.maxsize
    min_node = None
    for node in queue:
        if distance[node] < min_dist:
            min_dist = distance[node]
            min_node = node

    # Remove the node with the minimum distance from the queue
    queue.remove(min_node)

    # Update the distances of the neighboring nodes
    for i in range(len(graph[min_node])):
        if graph[min_node][i] > 0:
            new_dist = distance[min_node] + graph[min_node][i]
            comparisons += 1
            if new_dist < distance[i]:
                distance[i] = new_dist
                queue.append(i)


return distance, comparisons
```

Set minimum dist to infinity, minimum vertex to -1 first

Find vertex with minimum distance

Go into the node and update neighbouring nodes, and add into queue if shortest is not already found

Key comparison being made

B) Using Adjacency Lists and Minimising Heap for Priority Queue

```
def main():

    adjacency_lists = generate_random_graph(10, 25)

    start_time = time.time()
    distances, comp = dijkstra(adjacency_lists, 0)
    end_time = time.time()

    print("Time to run Dijkstra's algorithm:", end_time - start_time)
    print("Number of key comparisons: ", comp)
```

Generate random graph with 10 vertices and 25 edges

Record time

Print results

B) Using Adjacency Lists and Minimising Heap for Priority Queue

```
def generate_random_graph(num_vertices, num_edges):
    graph = {}

    for vertex in range(num_vertices):
        graph[vertex] = []

    #keep track of number of edges added to graph
    edge_count = 0
    while edge_count != num_edges:
        vertex1 = random.randint(0, num_vertices - 1)
        vertex2 = random.randint(0, num_vertices - 1)

        if vertex1 != vertex2: #make sure not equal
            weight = random.randint(1, 10)

            #add into graph
            if (vertex2, weight) not in graph[vertex1]: #make sure no dupe
                graph[vertex1].append((vertex2, weight))
                edge_count += 1

    return graph
```

Assigns empty list as value for that vertex key in dict

Keep track of number of edges added to graph

Make sure that edges are not repeated

B) Using Adjacency Lists and Minimising Heap for Priority Queue

```
Original adjacency list:
Vertex 0: [(4, 9), (2, 4), (3, 4), (1, 4)]
Vertex 1: [(2, 5), (4, 2)]
Vertex 2: [(4, 6), (0, 3)]
Vertex 3: [(0, 5)]
Vertex 4: [(1, 9)]
```

Example of random graph

B) Using Adjacency Lists and Minimising Heap for Priority Queue

```python
def dijkstra(graph, start):
    key_comp = 0
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0
    visited = set()
    heap = [(0, start)]
```

Initialise distances, set all distances to infinity, except starting, = 0

B) Using Adjacency Lists and Minimising Heap for Priority Queue

```
while heap:
    current_distance, current_vertex = heapq.heappop(heap)

    if current_vertex in visited:
        continue

    visited.add(current_vertex)

    key_comp += 1
    if current_distance > distances[current_vertex]:
        continue

    for neighbor, weight in graph[current_vertex]:
        distance = current_distance + weight

        key_comp += 1
        if distance < distances[neighbor]:
            distances[neighbor] = distance
            heapq.heappush(heap, (distance, neighbor))

return distances, key_comp
```

Vertex with minimum distance is popped

If vertex already visited, skip it

If more than, shortest path already found, go next

For each neighbour, find new distance

If less than, update