import heapq

def find\_shortest\_path\_with\_heap(

        graph: dict[int, dict[int, float]],

        source: int,

        target: int

) -> tuple[list[int], float]:

    """

    Find the shortest (least-cost) path from `source` to `target` in `graph`

    using the heap-based algorithm.

    Return:

        - the list of nodes (including `source` and `target`)

        - the cost of the path

    """

    priority\_queue = [(0, source)]

    distance = {node: float('inf') for node in graph}

    distance[source] = 0

    paths = {source: [source]}

    visited = set()

    while priority\_queue:

        current\_distance, current\_node = heapq.heappop(priority\_queue)

        if current\_node in visited:

            continue

        visited.add(current\_node)

        if current\_node == target:

            return paths[current\_node], current\_distance

        for neighbor, weight in graph.get(current\_node, {}).items():

            if neighbor in visited:

                continue

            new\_distance = current\_distance + weight

            if new\_distance < distance[neighbor]:

                distance[neighbor] = new\_distance

                heapq.heappush(priority\_queue, (new\_distance, neighbor))

                paths[neighbor] = paths[current\_node] + [neighbor]

    return [], float('inf')

**Time Complexity**

Priority\_queue = [(0, source)] **O(1)** --- inserting the source (s) node and distance 0

Distance = {node: float(‘inf’) for node in graph} **O(V)** --- assigning the distance of all the node to infinity

O(V)

Paths = {source: [source]} **O(V)** ----assigning the path of the source

Visited = set() **O(1)** --- initializing the visted set

While loop

While priority\_queue:

Current\_distance, current\_node = heapq.heappop(priority\_queue) **O(logV)** --- the heappop is use to remove the smallest node from the priority queue. The heap is O(logV) because its implementation is the binary tree where all the levels are filled except for the last level. In the heap, the parent is always smaller or equal to the children. The system runs until the tree is filled with the right positions. Its takes log v to perform this function.

Current\_node in visited: **O(1)** ---- this checks if the current node has been visited.

Visited.add(current\_node) **O(1)** --- this adds the current node to the visited. This ensures that the node is not visited again.

Current\_node == target **O(1)**

Return paths[current\_node], current\_distance --- this checks if the current node is the target node in question. If it is then it has found the shortest path.

O(V)

For neighbor, weight in graph.get(current\_node, {}).items():

If neighbor in visited:

Continue **O(E/V)** ---- the for loop iterates over the neigbors of the current node and work around them when they have not been visited.

New\_distance = current\_distance + wight O(1) --- for each neighbor of the current node, it computes the new distance.

If new\_distance < distance[neighbor]:

Distance[neigbor] = new\_distance

Heapq.heappush(priority\_queue, (new\_distance , neighbor)) **O( logV)** --- when a shorter distance is found the heap push pushes/insert the new distance and neighbor into the priority queue and that takes (logV) times. This is because when a node is inserted, it will need to travel by swapping from the bottom to the top of the tree. It takes log V to perform this function.

Paths[neigbor] = path[current\_node] + [neighbor] **O(E)** --- iterating over the edges takes E times.

Therefore: **time complexity** is **(O(E + V)logV)**

**Space complexity**

Priority\_queue O(V) – this stores the nodes that will be used.

Distance O(V) --- the distance is a dictionary that stores the shortest distances from the source node.

Paths O(V2) --- this is a dictionary that stores the length of the source node to each node. This is because each path can be of length V, and there are V nodes. The space for storing the path could be V\*V = V^2

Visited set O(V) --- stores the nodes that have been used.

**The space complexity** is V2

V^2 of the space complexity is due to the paths

O(logV) of the time complexity is due to the heap implementation

def find\_shortest\_path\_with\_array(

        graph: dict[int, dict[int, float]],

        source: int,

        target: int

) -> tuple[list[int], float]:

    """

    Find the shortest (least-cost) path from `source` to `target` in `graph`

    using the array-based (linear lookup) algorithm.

    Return:

        - the list of nodes (including `source` and `target`)

        - the cost of the path

    """

    unvisited\_nodes = [(0, source)]

    distance = {node: float('inf') for node in graph}

    distance[source] = 0

    paths = {source: [source]}

    visited = set()

    while unvisited\_nodes:

        current\_distance, current\_node = min(unvisited\_nodes, key=lambda x: x[0])

        unvisited\_nodes.remove((current\_distance, current\_node))

        if current\_node in visited:

            continue

        visited.add(current\_node)

        if current\_node == target:

            return paths[current\_node], current\_distance

        for neighbor, weight in graph.get(current\_node, {}).items():

            if neighbor in visited:

                continue

            new\_distance = current\_distance + weight

            if new\_distance < distance[neighbor]:

                distance[neighbor] = new\_distance

                unvisited\_nodes.append((new\_distance, neighbor))

                paths[neighbor] = paths[current\_node] + [neighbor]

    return [], float('inf')

Time Complexity

unvisited\_nodes **O(1)** --- it takes constant amount of time to initialize the list

Distance = {node: float(‘inf’) for node in graph} **O(V)** --- this creates a dictionary where the nodes in the graph are assigned a value of inifinity. The time it takes to iterate through the node is O(V)  
distance[source] = 0 **O(1**) --- this updates the dictionary

Paths = {source: [source]} **O(1)** --- assigning the paths dictionary to the source node

Visited = set() = initializing the visited set

While loop

current\_distance, current\_node = min(unvisited\_nodes, key=lambda x: x[0]) **O(V2)** ---- this finds the node with the smalles distance in unvisited\_nodes. unvisited\_nodes list takes O(V) times to find the smallest node. The function is performed V times.

unvisited\_nodes.remove((current\_distance, current\_node)) **O(V2)** --- this removes nodes from the list and it takes O(V) times performing this function, worse case senario. Its also takes O(V) to shift nodes after the first node is removed.

if current\_node in visited:

continue **O(V)** --- checks nodes in the set which take O(1) times and this is performed V times.

if current\_node == target:

return paths[current\_node], current\_distance **O(V)** --- this checks if the current node is the target node which takes O(1) and this is done for every node which takes O(V) times.

for neighbor, weight in graph.get(current\_node, {}).items(): **O(E)** --- the for loop iterates over the neighbors(edges) of the nodes and this takes O(E) times.

if neighbor in visited:

continue **O(E)** --- checks the nodes in the set which takes O(1) times and this is performed on every edge and it takes O(E) times.

new\_distance = current\_distance + weight **O(1)** --- this line adds the current\_distance and the weight and it takes a constant time

if new\_distance < distance[neighbor]: **O(1)** --- this comparison takes a constant time

distance[neighbor] = new\_distance **O(1) ---** updating the dictionary takes a constant

unvisited\_nodes.append((new\_distance, neighbor)) **O(E)** --- appending a new nodes takes a constant time and the total time appending on every node takes O(E) times.

paths[neighbor] = paths[current\_node] + [neighbor] **O(V2**) --- this line creates a new list for the paths by coping the current path and appending its neighbors. Its takes O(V) worse case and a complexity of O(V2)

Therefore: **Time complexity** of the array is **O(V2 + E)**

Space complexity of the array take **O(V2)** due to the path dictionary which could store every node that contains V nodes.

1. [20] Complete the tables below. Then discuss how your empirical results compare to your theoretical analysis. How do your two implementations of the priority queue perform on different graphs (low vs high density)? Discuss why this might be.

Use seed = 312, noise = 0.02, source = 2, and target = 9 when generating the data for these tables.

**Empirical results**

| **n** | **density** | **# edges** | **"heap" time** | **"linear" time** |
| --- | --- | --- | --- | --- |
| 1000 | 0.01 | 10000 | 0.0082 | 0.1149 |
| 5000 | 0.002 | 500000 | 0.0320 | 2.8851 |
| 10000 | 0.001 | 100000 | 0.0519 | 3.3869 |
| 50000 | 0.0002 | 500000 | 0.5224 | 84.8670 |
| 100000 | 0.0001 | 1000000 | 1.4204 | 560.3124 |

| **n** | **density** | **# edges** | **"heap" time** | **"linear" time** |
| --- | --- | --- | --- | --- |
| 1000 | 1 | 999000 | 0.2286 | 26.7984 |
| 2000 | 1 | 3998000 | 1.1019 | 220.5582 |
| 3000 | 1 | 8997000 | 2.3155 | 680.0490 |
| 4000 | 1 | 15996000 | 5.1854 | 2120.2525 |
| 5000 | 1 | 24995000 | 4.1851 | 1143.1553 |
| 6000 | 1 | 35994000 | 6.2184 |  |

**Heap**

example

(O(E+V)logV)

(O(10000 + 1000)log1000)

(11000)(3)

(33,000)

Array

O(V^2 + E)

(1000^2 + 10000)

(1,000,000 = 1,0000)

(1,010,000)

**Heap**: With smaller graphs the heap runs in seconds and for larger graphs the completion time increases but its manageable.

**Array**: with smaller graphs the time is takes to run is fast but not as fast as the heap and as the time of completion increases the array become much slower.

**Theoretical analysis**

Heap priority queue

Time complexity = O(V+E)logV) --- this is efficient because both inserting the node and popping it takes O(logV) times. This is useful for both sparse and dense graph

Array priority queue

Time complexity = O(V^2 + E) – this performs poorly because finding the smallest node require liner search through a list which takes O(V) times.

**Low and High Density**

Low density graph

The heap performance is excellent because its time complexity is efficient even when the edge is small.

The array run time is worse because of it use of O(V^2) and since it has to perform a linear search through each iteration.

High density graph

The heap still perform well in the high density graph and it is efficient while the number of edges increases.

The array performs poorly because it struggles to search through all the vertices linearly as it iterate through the nodes.