Project 3 Report

Dist Node

```
class DistNode:  3 usages
    def __init__(self, key, distance):
        self.key = key
        self.distance = distance

    def __str__(self):
        return f"({self.key}: {self.distance})"

    def __lt__(self, other):
        return self.distance < other.distance
```

This class allows me to more conveniently store the distances in each implementation of the priority queue. It simply contains the distance and the key value for the vertex. All of these functions have a time complexity of O(1) and a space complexity of O(1) so this class doesn't make the implementations of the priority queues any more complex, in terms of space or time.

Linear Priority Queue

```
class LinearPriorityQueue:  1 usage
    def __init__(self):
        self.index = dict()
        self.dist = []
        self.prev = dict()

    def insert(self, key):  1 usage
        self.dist.append(DistNode(key, math.inf))
        self.index[key] = len(self.dist) - 1
        self.prev[key] = None

    def make_queue(self, nodes: dict):  1 usage
        for key in nodes.keys():
            self.insert(key)

    def delete_min(self):  1 usage
        least = min(self.dist)
        i = self.index[least.key]
        ret = self.dist.pop(i)
        if i < len(self.dist):
            self.index[least.key] = len(self.dist)
            self.dist.insert(i, self.dist.pop())
            self.index[self.dist[i].key] = i
        return ret

    def decrease_key(self, key, distance, previous):  2
        self.dist[self.index[key]].distance = distance
        self.prev[key] = previous
```

This implementation of a priority queue uses a simple array to keep track of the distances. The "insert" function has a time and space complexity of O(1) because it just has to insert another node at the end of the array. The "make_queue" function has a time and space complexity of O(|V|) because it does an insert for all vertices in the graph. The "delete_min" function has a space complexity of O(1) and a time complexity of O(|V|) because finding the minimum value in an array takes O(n) operations for the size of the array. The rest of the operations in "delete_min" are all constant or linear as well. The "decrease_key" function has a time and space complexity of O(1) because array indexing and variable reassignments are both constant time operations.

Array Implementation

```
def find_shortest_path_with_array(  2 usages
        graph: dict[int, dict[int, float]],
        source: int,
        target: int
) -> tuple[list[int], float]:
    pq = LinearPriorityQueue()
    pq.make_queue(graph)
    pq.decrease_key(source,  distance: 0,  previous: None)
    while len(pq.dist) > 1:
        least = pq.delete_min()
        for key in graph[least.key].keys():
            if pq.index[key] < len(pq.dist):
                old_dist = pq.dist[pq.index[key]].distance
                new_dist = least.distance + graph[least.key][key]
                if old_dist > new_dist:
                    pq.decrease_key(key, new_dist, least.key)
    cost = 0
    current = target
    path = [current]
    while current != source:
        cost += graph[pq.prev[current]][current]
        current = pq.prev[current]
        path.insert( _index: 0, current)
    return path, cost
```

In the "find_shortest_path_with_array" function, we use the Linear Priority Queue to implement Djisktra's Algorithm. The while loop will loop as many times as there are vertices in the graph. In the whole runtime of the function the max number of iterations in the for loop cannot exceed the number of edges in the graph. Because of this "delete_min" will run |V| times and "decrease_key" will run at most |E| times. Therefore, the entire while loop will have a space complexity of O(|V|) and a time complexity of O(|V|^2 + |E|) which can be simplified to O(|V|^2) because the graph can never have more edges than |V|^2. Finding the path and cost won't take any longer than O(|V|^2) so the total time complexity of the "find_shortest_path_with_array" function remains at O(|V|^2).

Heap Priority Queue

```
class HeapPriorityQueue:  1 usage

    def __init__(self):
        self.heap: [DistNode] = []
        self.index = dict()
        self.prev = dict()

    def swap(self, key1, key2):  3 usages
        temp_dist = self.heap[self.index[key1]]
        temp_index = self.index[key1]
        self.heap[self.index[key1]] = self.heap[self.index[key2]]
        self.index[key1] = self.index[key2]
        self.heap[self.index[key2]] = temp_dist
        self.index[key2] = temp_index
```

This implementation of the priority queue uses a heap to keep track of the distances and to keep track of the least distance in the heap. The heap is also an array so it won’t be any bigger than the dist array from the linear implementation. The “swap” function is able to switch to nodes in the heap and update their indices contained in the index dictionary. This function has a time and space complexity of O(1). The “min_child” function is able to inspect the possible children of a node and return the key of the smaller child, or itself if it has no children. This function also has a time and space complexity of O(1). The “sift_down” function is a recursive function that causes a node to swap with one of its children if it has a larger distance than it until it is in the right place. Similarly, “bubble_up” function does the same thing but in the other direction.

```
def min_child(self, key):  1 usage
    left_child_index = (self.index[key] * 2) + 1
    right_child_index = (self.index[key]* 2) + 2
    if left_child_index < len(self.heap):
        if right_child_index >= len(self.heap):
            return self.heap[left_child_index].key
        else:
            left_child = self.heap[left_child_index]
            right_child = self.heap[right_child_index]
            if left_child.distance < right_child.distance:
                return left_child.key
            return right_child.key
    else:
        return key

def sift_down(self, key):  2 usages
    min_child = self.min_child(key)
    if (self.heap[self.index[key]].distance >
            self.heap[self.index[min_child]].distance):
        self.swap(key, min_child)
        self.sift_down(key)

def bubble_up(self, key):  2 usages
    if self.index[key] > 0:
        parent = self.heap[((self.index[key] + 1) // 2) -1]
        if (parent.distance >
                self.heap[self.index[key]].distance):
            self.swap(key, parent.key)
            self.bubble_up(key)
```

Because both of these functions are recursive they have a worst case time and space complexity of O(log|V|). Since I'm currently only using the "insert" function in "make_queue", I know that each node will only ever have a value of infinity. Because of that, "insert" can be done in linear time. However, if I had to worry about inserts of nodes that didn't have value of infinity, then I would have to call "bubble_up" which would cause it have a time and space complexity of O(log|V|). The "make_queue" function is able to be performed in linear time so it has a time and space complexity of O(|V|). The "delete_min" function is able to find the least value in constant time but it then has to call "sift_down" on the value that replaced it. Because of that, "delete_min" has a time and space complexity of O(log|V|). The "decrease_key" function is able to change the distance of a given node, and the "bubble_up" to the right position on the heap. It also has a time and space complexity of O(log|V|).

```
def insert(self, key):  1 usage
    self.heap.append(DistNode(key, math.inf))
    self.index[key] = len(self.heap) - 1
    self.prev[key] = None

def make_queue(self, nodes: dict):  1 usage
    for key in nodes.keys():
        self.insert(key)

def delete_min(self):  1 usage
    least = self.heap[0]
    self.swap(least.key, self.heap[len(self.heap) - 1].key)
    self.heap.pop()
    self.sift_down(self.heap[0].key)
    return least

def decrease_key(self, key, distance, previous):  2 usages
    self.heap[self.index[key]].distance = distance
    self.prev[key] = previous
    self.bubble_up(key)
```

Heap Implementation

```
def find_shortest_path_with_heap(  2 usages
        graph: dict[int, dict[int, float]],
        source: int,
        target: int
) -> tuple[list[int], float]:
    pq = HeapPriorityQueue()
    pq.make_queue(graph)
    pq.decrease_key(source,  distance: 0,  previous: None)
    while len(pq.heap) > 1:
        least = pq.delete_min()
        for key in graph[least.key].keys():
            if pq.index[key] < len(pq.heap):
                old_dist = pq.heap[pq.index[key]].distance
                new_dist = least.distance + graph[least.key][key]
                if old_dist > new_dist:
                    pq.decrease_key(key, new_dist, least.key)
    cost = 0
    current = target
    path = [current]
    while current != source:
        cost += graph[pq.prev[current]][current]
        current = pq.prev[current]
        path.insert( __index: 0, current)
    return path, cost
```

In the "find_shortest_path_with_heap" function, we use the Heap Priority Queue to implement Djisktra's Algorithm. Besides using a different priority queue, the implementation is almost identical to the Array Implementation. Because of that, "delete_min" will also run |V| times and "decrease_key" will also run at most |E| times. Therefore the entire loop will have space complexity of O(|V|) and a time complexity of O(|V|(log|V|) + |E|(log|V|)) which can be simplified to O((|V|+|E|)log|V|). Again, because finding the path and cost can be done in linear time, the entire function will have a time complexity of O((|V|+|E|)log|V|).

Empirical Analysis

n	density	# edges	"heap" time	"linear" time
1000	0.01	10000	0.009	0.02589
5000	0.002	50000	0.06245	0.70568
10000	0.001	100000	0.134669	3.110585
50000	0.0002	500000	1.04429	89.33
100000	0.0001	1000000	2.51996	459.857

n	density	# edges	"heap" time	"linear" time
1000	1	999000	0.1351	0.15398
2000	1	3998000	0.571	0.7099
3000	1	8997000	1.5187	1.84
4000	1	15996000	2.85	3.397
5000	1	24995000	5.7275	6.033
6000	1	35994000	9.857	9.123

The heap implementation is way better at handling graphs with a lower density of edges than the linear implementation. However it does not handle more dense graphs as well as it does with less dense graphs as a more dense graph of 6000 nodes already takes more time than a less dense graph of 100000 nodes. The linear implementation is just as good as the heap implementation at a density of 1. This is because the time complexity of the linear implementation doesn't hinge on the number of edges whereas the time complexity of the heap implementation does ($O(|V|^2)$ vs $O((|V|+|E|)\log|V|)$).