REPORT PROJECT 3 Zach McWhorter

TIME AND SPACE COMPLEXITY:

Linear Priority Queue Implementation -

LinearPriorityQueueDict.py > LinearPQDict > extract_min

```python
import math

class LinearPQDict:
    def __init__(self):
        self.dict = {}

    def insert1(self, node, priority):
        self.dict[node] = priority

    def extract_min(self):
        if not self.dict:
            return None

        min = math.inf
        min_node = ""

        for node, priority in self.dict.items():
            if priority < min:
                min = priority
                min_node = node

        if min_node == "":
            return None

        del self.dict[min_node]

        return (min_node, min)

    def decrease_key(self, node, new_priority):
        self.dict[node] = new_priority
```

The space complexity of the dictionary will be O(v), because each node will be added. Instantiation of the dictionary will be O(1) time.

insert() is o(1) operation because it is just adding to a dictionary implemented with a hash table.

decrease_key() will also be O(1) because it is just updating the dictionary.

extract_min() checks if there is anything in the dictionary in O(1) time. Then it iterates over each key value pair in the dictionary O(v), does a simple comparison O(1), and a simple variable reassignment O(1). Finally it deletes a key value pair from the dictionary in O(1).

Heap Priority Queue Implementation

```
class HeapPQ:
    def __init__(self):
        self.heap = []
        self.position_map = {}

```

The heap implementation uses an array and a map to store the heap and the indexes of each node. Space requirements for these will grow linearly O(v). They require constant time for instantiation.

```python
    def swap(self, index1, index2):
        self.position_map[self.heap[index1][0]] = index2
        self.position_map[self.heap[index2][0]] = index1

        temp = self.heap[index1]
        self.heap[index1] = self.heap[index2]
        self.heap[index2] = temp

    def bubble_up(self, index):
        while index > 0:
            parent = self.get_parent_index(index)
            if self.heap[index][1] < self.heap[parent][1]:
                self.swap(index, self.get_parent_index(index))
                index = parent
            else:
                break

    def bubble_down(self):
        index = 0
        size = len(self.heap)
        while index < size:
            left = self.get_left_child_index(index)
            right = self.get_right_child_index(index)
            smallest = index

            if left < size and self.heap[smallest][1] > self.heap[left][1]:
                smallest = left
            if right < size and self.heap[smallest][1] > self.heap[right][1]:
                smallest = right

            if smallest != index:
                self.swap(index, smallest)
                index = smallest
            else:
                break
```

The swap method will operate in O(1) time because it is simply updating the dictionary.

Bubble up will occur in O(log(v)) time because it must perform a maximum log(v), with n being nodes, swaps in order to correct the heap.

Bubble down is also O(log(v)) time for the same reason. All comparisons are O(1) operations.

```python
    def insert1(self, node, priority):
        self.heap.append((node, priority))
        self.position_map[node] = len(self.heap) - 1
        self.bubble_up(len(self.heap) - 1)

    def extract_min(self):
        if (len(self.heap) == 1):
            return self.heap.pop()
        min_node = self.heap[0]
        self.heap[0] = self.heap.pop()
        del self.position_map[min_node[0]]
        if self.heap:
            self.position_map[self.heap[0][0]] = 0
            self.bubble_down()
        return min_node

    def decrease_key(self, node, new_priority):
        #update and bubble up
        index = self.position_map[node]
        self.heap[index] = (node, new_priority)
        self.bubble_up(index)
```

insert() is an O(log(v)) operation. Appending to the heap is O(1). Adding a key, value pair to the map is O(1). Bubble_up() is O(log(v)).

extract_min() is O(log(v)) operation because we have to move the last element in the heap to the root and call bubble down. Popping the last element, replacing the first, and deleting from the map are all O(1) operations. But we must bubble_down() which is O(log(v)).

decrease_key() is also O(log(v)) because we are simple grabbing the index of the node from the map, O(1), and reassigning the node and new priority to that index, O(1), and then doing bubble_up(), O(log(v)).

Djikstras

```python
def djikstras(graph: dict[int, dict[int, float]],
        source: int,
        target: int,
        pq_implementation
) -> tuple[list[int], float]:

    pq = pq_implementation()
    dist = defaultdict(lambda: math.inf)
    visited = set()
    prev = {}
    dist[source] = 0
    pq.insert1(source, 0)

    while pq:
        u, u_dist = pq.extract_min()

        if u in visited:
            continue
        visited.add(u)

        # Early exit if we reached the target node
        if u == target:
            path = []
            while u is not None:
                path.insert(0,u)
                u = prev.get(u, None)
            return path, dist[target]

        for neighbor, weight in graph[u].items():
            if (neighbor not in visited) and (dist[u] + weight < dist[neighbor]):
                dist[neighbor] = dist[u] + weight
                prev[neighbor] = u

                if neighbor not in visited:
                    pq.insert1(neighbor, dist[neighbor])
                else:
                    pq.decrease_key(neighbor, dist[neighbor])

    return [], -1
```

The dictionaries dist, and prev both have worst case space complexity, O(v). The set visited also has O(v) worst case space complexity. Populating the dictionary with infinity values will take O(v) time.

The initial call to insert will happen only once, so O(1).

The while loop will iterate maximum over each node v, O(v) In the loop we check each edge O(e). The other comparisons are each O(1). Once we find the target node we also assemble the path list which will be worst case O(v) time and space. The overall complexity for djikstras before accounting for the implementation of the priority queue is O(e*v).

Djikstras with Heap PQ:

Because the Heap PQ operations are each log(v) the overall complexity of djikstras will be O((V*E) log(v)).

Djikstras with Linear PQ:

Because the Linear PQ has operation O(v) the overall complexity of djikstras will be O(V * E)(V)), which simplifies to O(V^2).

Empirical and Theoretical Data Analysis:

These are the results from the averages of 5 runs

Sparse Data

n	density	# edges	"heap" time	"linear" time
1000	0.01	10000.0	0.00944056510925293	0.013390445709228515
5000	0.002	50000.0	0.03916950225830078	0.25856833457946776
10000	0.001	100000.0	0.04231538772583008	0.4276014804840088
50000	0.0002	500000.0	0.2459270477294922	8.846491479873658
100000	0.0001	1000000.0	0.7791440963745118	65.52773480415344

Dense Data

n	density	# edges	"heap" time	"linear" time
1000	1	999000.0	0.2064115047454834	0.0994659423828125
2000	1	3998000.0	0.7667218685150147	0.39360713958740234
3000	1	8997000.0	1.6733731269836425	0.9945423603057861
4000	1	15996000.0	3.1063557624816895	1.958125352859497
5000	1	24995000.0	4.373342704772949	3.028878021240234
6000	1	35994000.0	6.031571578979492	4.427707290649414

The empirical data for the Sparse graph follows closely the theoretical complexity of V^2 and V*E log(V). The results appear slightly better than the theoretical big O complexity. This is due to the fact that big O is a worst case scenario. The average case may be better.

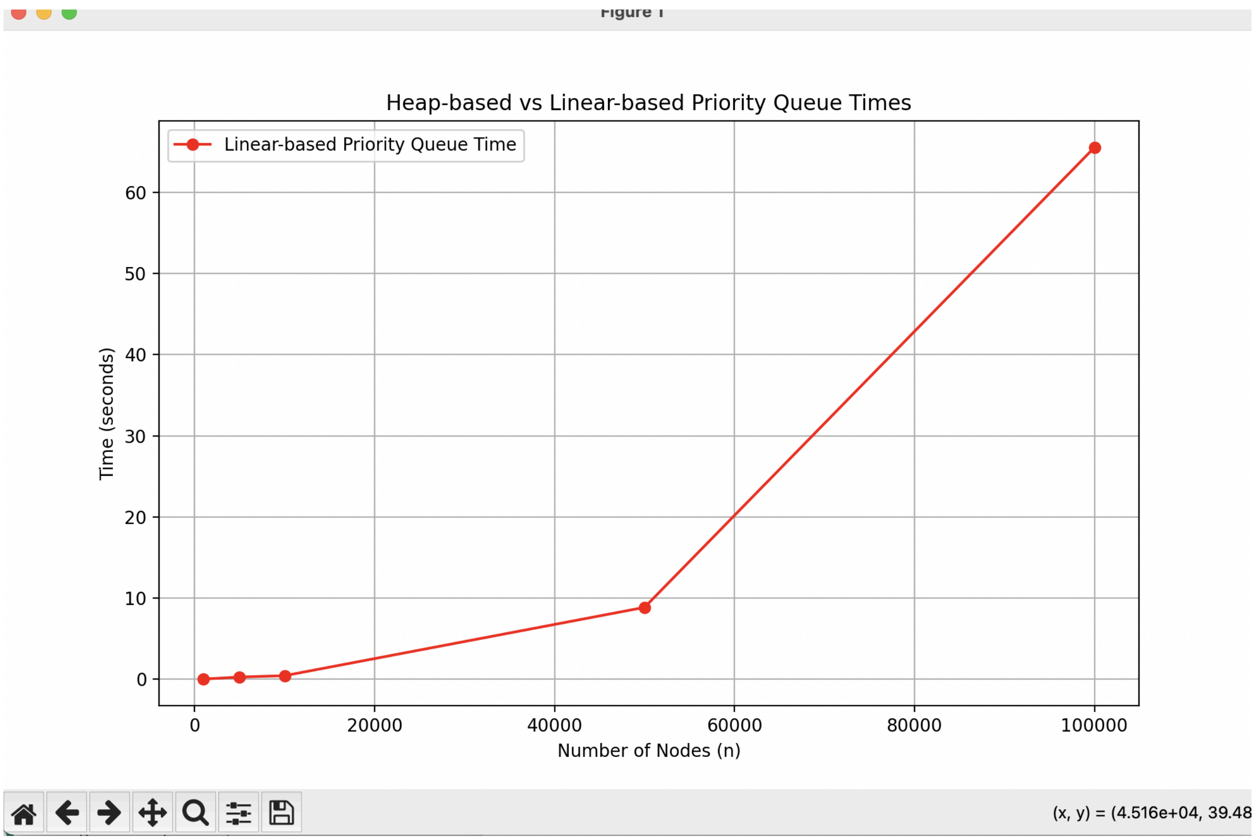
Figure 1
Heap-based vs Linear-based Priority Queue Times
Linear-based Priority Queue Time
Time (seconds)
Number of Nodes (n)
0
10
20
30
40
50
60
0
20000
40000
60000
80000
100000
(x, y) = (4.516e+04, 39.48

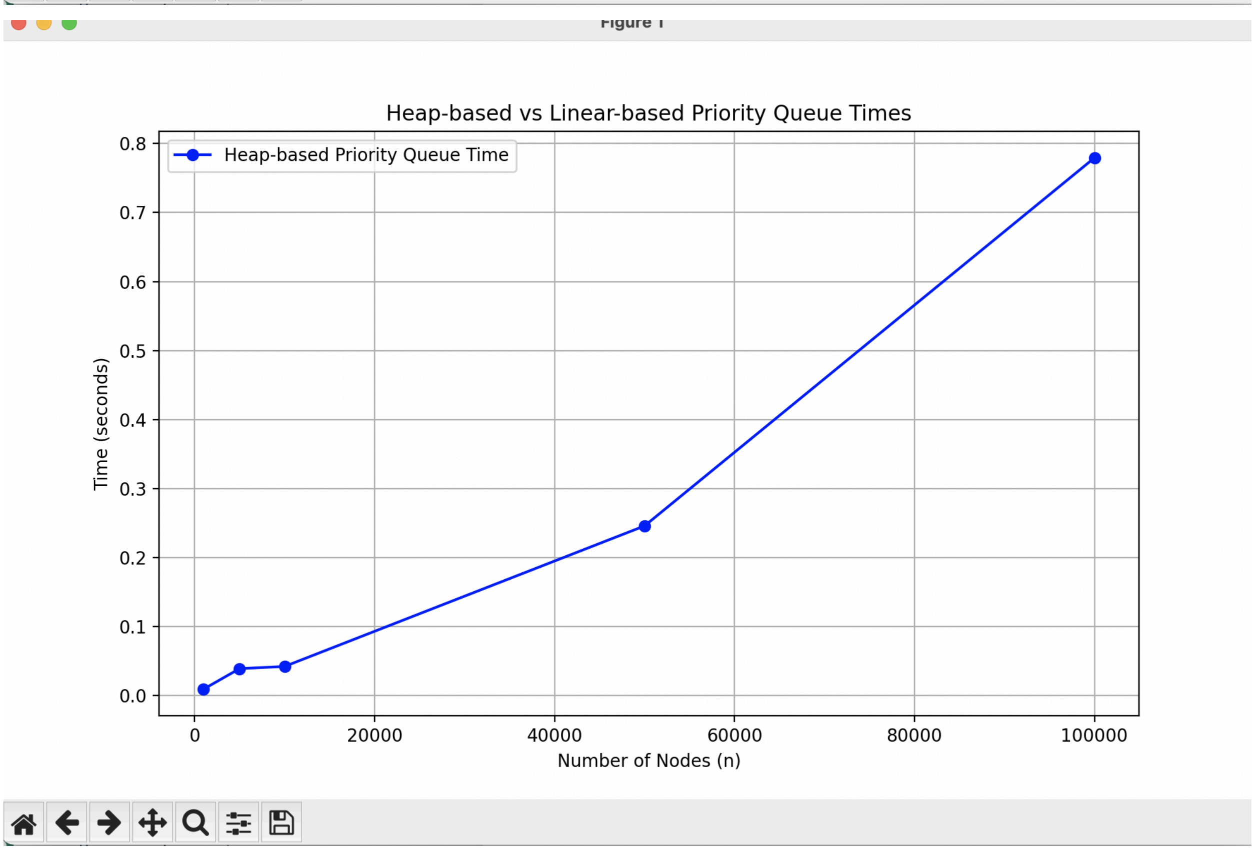
Figure 1
Heap-based vs Linear-based Priority Queue Times
Heap-based Priority Queue Time
Time (seconds)
Number of Nodes (n)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
20000
40000
60000
80000
100000

For the dense graph, the linear PQ performs better than the heap PQ. This is due to the fact that when the algorithm runs on a dense graph, it must check many more edges per node, and inevitably will run decrease_key() much more often than on a sparse graph. Since decrease_key() is O(logV) for the heap PQ, it will perform worse than the linear PQ which has O(1) decrease_key().