Exam Review: Problem Set 5 & Red-Black Trees

Part 1: Priority Queues & Heaps

Question 3: Sorted Array Implementation

Scenario: Implementing a Priority Queue using a sorted array (largest to smallest).

- **Extract-Min:** The minimum element is at the end of the array (index $n-1$). Removing it takes constant time.
- **Insert:** We must use Binary Search to find the position ($O(\log n)$), but then we must shift all smaller elements to make space. In the worst case (inserting a new maximum), this takes linear time.

Answer: Insert is $\Theta(n)$ and Extract-Min is $\Theta(1)$.

Question 4: Unsorted Array Implementation

Scenario: Implementing a Priority Queue using an unsorted array.

- **Insert:** Order does not matter, so we can simply append the new element to the end.
- **Extract-Min:** The minimum could be anywhere. We must scan the entire array (n elements) to find it.

Answer: Insert is $\Theta(1)$ and Extract-Min is $\Theta(n)$.

Question 5: Heap Capabilities

Scenario: Binary Heap with n elements. What can be done in $O(\log n)$?

- **Find largest:** In a Min-Heap, the largest element is at a leaf. We might need to check all $\lceil n/2 \rceil$ leaves. Time: $\Theta(n)$.
- **Find median:** Heaps are not fully sorted. Finding the median generally takes $\Theta(n)$ or $\Theta(n \log n)$.
- **Find fifth-smallest:** The smallest is the root. The 2nd and 3rd are in level 2. The 4th and 5th are near the top. Since $k = 5$ is a constant, searching the top few levels takes constant time relative to n.

Answer: Find the fifth-smallest element.

Part 2: Shortest Paths (Graphs)

Question 1: Path Properties

Scenario: Directed graph with distinct, non-negative edge lengths.

- **True:** The shortest path might have $n - 1$ edges (e.g., a line graph).
- **True:** There is a shortest path with no repeated vertices (a simple path). Cycles with non-negative weights can be removed without increasing path length.
- **False:** It must include the minimum-length edge (it might not be on the path).
- **False:** It must exclude the maximum-length edge (it might be a necessary bridge).

Answer: Statements 1 and 2 are true.

Question 2: Modified Dijkstra

Scenario: Edges leaving source s can be negative. No edges enter s. All other edges non-negative.

- Dijkstra normally fails with negative edges because it assumes a "closed" node's distance is final.
- Here, negative edges only occur at the start. Since no edges return to s, we process neighbors of s first.
- We can view this as adding a constant M to all edges leaving s to make them positive. This increases all path lengths by exactly M, preserving the relative order of shortest paths.

Answer: Dijkstra always works in this specific case.

Part 3: Red-Black Trees

Re-implementation Requirements

To maintain Red-Black invariants (no double reds, equal black height):

- **Search:** No change needed (standard BST search).
- **Insert:** Requires recoloring and rotations to fix violations (e.g., Red parent + Red child).
- **Delete:** Removing a node can violate Black Height. Requires complex fix-up (rotations/recoloring).

Answer: Insert and Delete must be re-implemented.

Height Guarantee

The height of a Red-Black tree is $\Theta(\log n)$ because:

1. Every root-NULL path has $\leq \log_2(n+1)$ black nodes.
2. No red node has a red child, so at most 50% of nodes on a path are red.
3. Therefore, total height $\leq 2 \times$ (Black Height) $\leq 2\log_2(n+1)$.

Optional Theory Problems: Bottleneck Paths

Problem 1: Modified Dijkstra for Bottleneck Paths

Goal: Compute a path where the maximum edge weight (bottleneck) is minimized. Time: $O(m \log n)$.

- **Algorithm:** Use Dijkstra's algorithm but modify the relaxation step.
- **Relaxation:** Instead of $d[v] = \min(d[v], d[u] + w(u,v))$, use:

$$d[v] = \min(d[v], \max(d[u], w(u,v)))$$

- **Explanation:** $d[u]$ represents the minimum bottleneck capacity to reach u. To extend this path to v, the new bottleneck is the larger of the previous bottleneck ($d[u]$) or the new edge itself ($w(u,v)$).
- **Complexity:** We use the same priority queue structure as standard Dijkstra. The number of operations is identical. Thus, $O(m \log n)$.

Problem 2: Undirected Graphs in Linear Time

Goal: Compute minimum-bottleneck path in undirected graph in $O(m)$.

- **Insight:** We can use a deterministic median-finding algorithm.
- **Algorithm (Sketch):**
  1. Find the median edge weight w_{med} of the current edges ($O(m)$).
  2. Consider the subgraph G_{low} containing only edges with weight $\leq w_{med}$.
  3. Run BFS/DFS on G_{low} to see if s and t are in the same connected component ($O(m)$).
  4. **If Connected:** The optimal bottleneck is $\leq w_{med}$. Discard all edges $> w_{med}$ and recurse on G_{low}.
  5. **If Not Connected:** The optimal bottleneck is $> w_{med}$. Contract the connected components of G_{low} into super-nodes and recurse using the edges $> w_{med}$.
- **Complexity:** In each step, we reduce the number of edges by half or contract vertices significantly. The recurrence is $T(m) = T(m/2) + O(m)$, which solves to $O(m)$.

Problem 3: Directed Graphs

Goal: Can we do faster than $O(m \log n)$ for directed graphs?

- **Answer:** No (or at least, not using the simple contraction method).
- **Reasoning:** The linear-time strategy for undirected graphs relies on **contracting** connected components to reduce the problem size when the bottleneck is in the "upper half" of edge weights.
- In directed graphs, reachability is not symmetric. If s cannot reach t using only "light" edges, we cannot simply contract the components, because a "light" edge might still be required to bridge two "heavy" edges in a valid path.
- Therefore, we typically cannot beat the $O(m \log n)$ bound of the modified Dijkstra (or a bottleneck-sort based approach) easily.

Programming Assignment 5

Question: In this programming problem you'll code up Dijkstra's shortest-path algorithm.

Download the following text file (Right click and select "Save As..."): dijkstraData.txt

The file contains an adjacency list representation of an undirected weighted graph with 200 vertices labeled 1 to 200. Each row consists of the node tuples that are adjacent to that particular vertex along with the length of that edge. For example, the 6th row has 6 as the first entry indicating that this row corresponds to the vertex labeled 6. The next entry of this row "141,8200" indicates that there is an edge between vertex 6 and vertex 141 that has length 8200. The rest of the pairs of this row indicate the other vertices adjacent to vertex 6 and the lengths of the corresponding edges.

Your task is to run Dijkstra's shortest-path algorithm on this graph, using 1 (the first vertex) as the source vertex, and to compute the shortest-path distances between 1 and every other vertex of the graph. If there is no path between a vertex and vertex 1, we'll define the shortest-path distance between 1 and to be 1000000.

You should report the shortest-path distances to the following ten vertices, in order: 7,37,59,82,99,115,133,165,188,197. Enter the shortest-path distances using the fields below for each of the vertices.

IMPLEMENTATION NOTES: This graph is small enough that the straightforward time implementation of Dijkstra's algorithm should work fine. OPTIONAL: For those of you seeking an additional challenge, try implementing the heap-based version. Note this requires a heap that supports deletions, and you'll probably need to maintain some kind of mapping between vertices and their positions in the heap.

Solution:

```
"""
Parsing: Reading the file and storing the graph.

The Algorithm: Implementing the loop to find shortest paths.
Reporting: Outputting specific distances.
"""

from typing import Any


def load_graph(filename):
    """
    Loads the graph from a text file into an adjacency list.
    Returns: A dictionary where keys are vertex IDs (int) and values
             are lists of tuples (neighbor_id, weight).
    """
    adj_list = {}

    try:
        with open(filename, "r") as f:
            for line in f:  # read line by line
                # 1. Split the line into parts (the vertex and the rest)
                parts = line.split()

                # 2. The first part is the current vertex
                vertex = int(parts[0])

                # 3. Initialize the list for this vertex in the dictionary
                adj_list[vertex] = []

                # 4. Loop through the remaining parts (neighbors)
                for neighbour in parts[1:]:
                    neighbour_id, weight = neighbour.split(",")
                    neighbour_id = int(neighbour_id)
                    weight = int(weight)
                    adj_list[vertex].append((neighbour_id, weight))
                #    For each part:
                #      a. Split by comma to get neighbor_id and weight
                #      b. Convert both to integers
                #      c. Append tuple (neighbor_id, weight) to adj_list[vertex]

    except FileNotFoundError:
        print(f"Error: The file '{filename}' was not found.")
        return None

    return adj_list
```

```
def djikstra(graph) -> dict[Any, Any]:
    X = []  # vertices processed so far
    V = list(graph.keys())
    A = {}  # computed shortest path distances
    for v in graph:
        A[v] = 1000000
    A[1] = 0

    while len(X) != len(V):
        # while total number of processed vertices doesn't total up to all the vertices
        # After picking the current_vertex (the one with the smallest distance), you nee
        # Create a list of unprocessed nodes

        # 1. Pick the unprocessed node with the smallest distance
        unprocessed_nodes = list(set(V) - set(X))
        min_key = min(
            unprocessed_nodes, key=lambda x: A[x]
        )  # the way we compare the vertices is by their A[key] = value

        # 2. MARK IT AS PROCESSED!
        X.append(min_key)

        for neighbour_tuple in graph[min_key]:
            neighbor_id = neighbour_tuple[0]
            edge_weight = neighbour_tuple[1]
            A[neighbor_id] = min(
                A[neighbor_id], A[min_key] + edge_weight
            )  # Djikstra's Greedy Criterion

    return A


graph = load_graph(
    "/Users/shiva/Documents/Competitive-programming/edx/Stanford/data/dijkstraData.txt"
)
if graph:
    # Print the first vertex to verify
    first_vertex = 1
    # print(f"Vertex {first_vertex}: {graph.get(first_vertex)}")

    print(djikstra(graph=graph))
```