CS-GY 6033 : Design and Analysis of Algorithms , Section C : Assignment-5

Khwaab Thareja — N15911999 — kt3180

November 2024

Problem 5-1: MST Excercises

1. Prove or disprove (by counterexample): if the edge weights are distinct, then there is a unique minimum spanning tree.

2. Given an undirected graph $G = (V, E)$ and a minimum spanning tree T, suppose that we decrease the weight of one of the edges $(u, v) \in E$ that is not in T. Give an algorithm for finding the minimum spanning tree in the modified graph. Your algorithm should run in $O(V + E)$ time, i.e., not computing an MST from scratch. Hint: can you find a cycle in $T \cup \{(u, v)\}$?

3. Suppose that a graph $G = (V, E)$ has a minimum spanning tree T already computed. How quickly can we update the minimum spanning tree if we modify G by adding a new vertex and its incident edges? In particular, can you come up with an algorithm that does not look at every edge in the graph?

Answer a

We will prove this by showing that when edge weights are unique, any algorithm (such as Kruskal's or Prim's) that constructs the MST will always choose the same edges.

The *cut property* states that for any partition of the graph (cut), the smallest edge that connects the two parts of the graph must be included in the MST.

Reasoning:

1. **Unique Edge Weights:** Since no two edges have the same weight, every cut in the graph will have exactly one edge with the smallest weight. This ensures there is no ambiguity in choosing the edge for the MST.

2. **Same Edges Chosen:** MST algorithms, such as Kruskal's and Prim's, select edges based on their weights. With unique weights, the algorithms will always select the same edges in the same order.

3. **Only One MST:** If there were multiple MSTs, they would differ by at least one edge. However, the uniqueness of edge weights ensures that the edges selected are fixed and consistent, making multiple MSTs impossible.

Example of Non-Unique Edge Weights:

If edge weights are not unique, multiple MSTs can exist. Consider the following graph:

$$\text{Vertices: } \{A, B, C\}$$

$$\text{Edges: } (A, B, \text{weight } 1), (B, C, \text{weight } 1), (A, C, \text{weight } 1)$$

Here, all edges have the same weight. Any two edges can form an MST, resulting in multiple valid MSTs. This shows that when edge weights are not unique, the MST is not guaranteed to be unique.

If all edge weights are distinct, the MST is always unique because the selection of edges is deterministic and consistent. However, if edge weights are not distinct, multiple MSTs may exist.

Answer b

If the weight of an edge that was not part of the MST decreases, the new MST can be efficiently computed without recomputing it from scratch. Here's the step-by-step process:

1. Add the Modified Edge to the MST

- Take the edge whose weight has decreased, let's call it (u, v).
- Add this edge to the MST. This creates a temporary tree with one extra edge.
- Since the MST already connects all the nodes, adding another edge creates a cycle (a loop).

2. Find the Cycle

- The new structure now contains exactly one cycle due to the added edge.
- Use a graph traversal method like Depth First Search (DFS) or Breadth First Search (BFS) to identify all edges involved in the cycle.

3. Remove the Heaviest Edge in the Cycle

- Among all the edges in the cycle, locate the edge with the maximum weight.
- Remove the heaviest edge from the cycle to restore the tree structure while ensuring it remains a valid MST.

4. The Resulting MST

- After adding the new edge and removing the heaviest edge in the cycle, the resulting structure is the updated MST.
- This approach ensures the MST remains valid while incorporating the modified edge.

Example

Imagine a graph with the following roads between cities:

- Road A-B: Weight 1
- Road B-C: Weight 2
- Road C-D: Weight 3

These roads form the original MST. Now, suppose there is an additional road A-D with a weight of 4, which was not part of the MST. If the weight of road A-D decreases to 1.5, follow these steps:

1. Add road A-D to the MST. This creates a cycle involving cities A, B, C, and D.
2. The roads in the cycle are A-B (weight 1), B-C (weight 2), C-D (weight 3), and A-D (weight 1.5).
3. The heaviest road in the cycle is C-D (weight 3). Remove road C-D from the MST.
4. The new MST consists of A-B (weight 1), B-C (weight 2), and A-D (weight 1.5).

Efficiency

- This method avoids recalculating the MST from scratch, as it only adjusts the tree by adding one edge and removing one edge.
- The algorithm runs in $O(V + E)$ time because finding the cycle and identifying the heaviest edge can be done using simple graph traversal techniques.

Answer c

Given a graph $G = (V, E)$ with an existing Minimum Spanning Tree (MST) T, and a new vertex v_{new} along with its incident edges E_{new}, efficiently update the MST without examining all the edges in the graph.

When adding v_{new}, the only edges relevant for the MST update are:

- The edges in the existing MST (T).
- The edges directly connecting v_{new} to the graph (E_{new}).

This localized approach avoids processing unrelated edges.

Algorithm

1. **Input:**
   - T: The current MST of the graph.
   - v_{new}: The new vertex being added.
   - E_{new}: The set of edges incident to v_{new}.
2. **Find the Lightest Edge from v_{new}:**
   - Iterate through E_{new} to identify the edge with the smallest weight.
   - Let this edge be $e_{\min} = (v_{\text{new}}, u_{\min})$.
3. **Add $e_{\min}$ to the MST:**
   - Temporarily add $e_{\min}$ to T. This may create a cycle.
4. **Detect and Resolve the Cycle:**
   - Use DFS or BFS starting from v_{new} to trace the cycle created by adding $e_{\min}$.
   - During the traversal, track the edges within the cycle and identify the heaviest edge, $e_{\max}$.
   - Remove $e_{\max}$ to break the cycle and restore the MST property.
5. **Output:**
   - The updated tree T is the new MST.

Avoiding Full Graph Traversal

- The algorithm limits its operations to the edges in E_{new} and the MST T.
- The cycle created by adding $e_{\min}$ is localized to these edges, ensuring efficient processing.

Time Complexity

1. **Finding the Lightest Edge:**
   - Scanning E_{new} takes $O(d_{\text{new}})$, where $d_{\text{new}} = |E_{\text{new}}|$, the degree of v_{new}.
2. **Cycle Detection and Resolution:**
   - Detecting the cycle involves traversing part of the MST T, which takes $O(V)$, where V is the number of vertices in the graph.
3. **Total Complexity:**
$$O(d_{\text{new}} + V).$$

Reasoning

1. **Localized Updates:** The algorithm processes only the edges connecting v_{new} and the existing MST edges, avoiding unrelated edges.
2. **Cycle Property:** Removing the heaviest edge in a cycle ensures that the MST property is preserved.
3. **Efficiency:** The algorithm ensures $O(V + d_{\text{new}})$ complexity, meeting the requirement for efficient MST updates.

This refined algorithm efficiently updates the MST by focusing on v_{new}, its incident edges E_{new}, and the existing MST T. It avoids examining all edges and guarantees a time complexity of $O(V + d_{\text{new}})$.

Problem 5-2: MSTs on Very Sparse Graphs

Suppose you are given as input a connected undirected graph $G = (V, E)$ with distinct real edge weights $w(u, v)$ for each edge $(u, v) \in E$. By distinct weights, I mean that no two edges have the same weight. Suppose also that the graph is extremely sparse, specifically that $|E| \leq n + 3$, where $n = |V|$. This level of sparsity means that the graph is very close to (only 4 edges away from) being a tree already.

1. What are the worst-case asymptotic running times of Prim's and Kruskal's algorithms on this very sparse graph type as a function of $n = |V|$? Briefly justify your answer. I'm really asking: given the restricted graph type, do these algorithms perform better than the general bounds we learned in class?
2. Prove the cycle property (Lemma 1) as stated, i.e., with the assumption of distinctness.

3. Design a new MST algorithm for this very sparse graph type that runs in $O(n)$ time. The algorithm should be fairly simple and make use of the previous problem parts.

Answer a

Given a sparse graph $G = (V, E)$ with $|E| \leq n + 3$, the running times of Prim's and Kruskal's algorithms improve due to the reduced number of edges.

Prim's Algorithm

- Prim's algorithm with a binary heap typically runs in $O(E + V \log V)$.
- For this graph, where $|E| \leq n + 3$, the runtime becomes:

$$O((n + 3) + V \log V).$$

- Simplifying further, the time complexity is:

$$O(n + n \log n) = O(n \log n).$$

- The sparse nature of the graph reduces the overall runtime compared to denser graphs.

Kruskal's Algorithm

- Kruskal's algorithm has a general runtime of $O(E \log E + V)$.
- Sorting the edges dominates the runtime. Since $|E| \leq n + 3$:

$$O(E \log E) = O((n + 3) \log(n + 3)).$$

- This simplifies to:

$$O(n \log n).$$

- The union-find operations add $O(E \cdot \alpha(V))$, where $\alpha(V)$ is the inverse Ackermann function, which is nearly constant. Therefore, the total runtime remains:

$$O(n \log n).$$

Conclusion

For a sparse graph with $|E| \leq n + 3$, both Prim's and Kruskal's algorithms perform better:

- **Prim's Algorithm:** $O(n \log n)$.
- **Kruskal's Algorithm:** $O(n \log n)$.

The limited number of edges ensures these algorithms run more efficiently than their general-case bounds.

Answer b

Statement: In a connected graph $G = (V, E)$ with distinct edge weights, the edge with the highest weight in any cycle C cannot belong to any Minimum Spanning Tree (MST).

1. **Assume for Contradiction:** Suppose the edge (u, v), which has the highest weight in cycle C, is part of an MST T.
2. **Remove (u, v) from the MST:** Removing (u, v) from T splits the tree into two disjoint connected components, say A and B.
3. **Reconnect the Components Using Cycle C:** The cycle C, which includes (u, v), contains other edges that connect A and B. Since all edge weights are distinct, there exists at least one edge $(x, y) \in C \setminus \{(u, v)\}$ such that $w(x, y) < w(u, v)$.
4. **Construct a New Tree T':** Replace (u, v) in T with (x, y). This new tree T' is connected and spans all vertices of G.
5. **Compare Weights:** The total weight of T' is less than the total weight of T, as $w(x, y) < w(u, v)$. This contradicts the assumption that T was an MST, which must have the minimum possible total weight.
6. **Conclusion:** The edge (u, v), being the heaviest in cycle C, cannot belong to any MST. This completes the proof.

This proof leverages the fact that removing the heaviest edge from a cycle and replacing it with a lighter edge reduces the total weight while preserving connectivity, which directly contradicts the definition of an MST.

Answer c

Algorithm

Step 1: Initialize

- Let $T = \emptyset$, the set to store edges of the MST.

Step 2: Construct an Initial Spanning Tree

- Use Depth-First Search (DFS) or Breadth-First Search (BFS) to construct a spanning tree of G.
- Add the edges of this spanning tree to T, ensuring T contains $n-1$ edges and spans all vertices.
- When adding an edge check if it is visited in the visited table and mark the nodes added as True in the table. This ensures that there are no cycles in this initial spanning tree.

Step 3: Process Extra Edges

- Identify the $|E| - n + 1$ additional edges in G that are not part of the initial spanning tree. Since $|E| \leq n + 3$, there are at most 3 such edges.
- For each extra edge (u, v):
  1. Add (u, v) to T, creating a temporary cycle.
  2. Detect the cycle in T using a simple traversal (e.g., DFS or BFS).
  3. Identify the edge in the cycle with the highest weight.
  4. Remove the heaviest edge from the cycle to restore the tree structure.

Step 4: Return the MST

- The resulting edge set T is the MST of G.

Time Complexity Analysis

1. **Constructing the Initial Spanning Tree:** Using DFS or BFS takes $O(n)$, as it processes all vertices and $n-1$ edges.
2. **Processing Extra Edges:** There are at most 3 extra edges ($|E|-n+1$). For each edge:
   - Detecting a cycle and identifying the heaviest edge using DFS or BFS takes $O(n)$.

   Total time for processing all extra edges: $O(3 \cdot n) = O(n)$.

Overall Complexity:

$$O(n) + O(n) = O(n).$$

Reasoning

1. **Cycle Property:** The algorithm uses the cycle property of MSTs: the heaviest edge in any cycle can be removed while maintaining the MST.
2. **Efficient Handling of Sparse Graphs:** By focusing only on the extra edges ($|E| - n + 1$), the algorithm avoids global operations like sorting.
3. **Localized Updates:** Adding and processing one edge at a time ensures minimal computation while preserving the MST property.

Example

Graph: Vertices: A, B, C, D, E Edges and weights:

- $(A, B) : 1$,
- $(B, C) : 2$,
- $(C, D) : 3$,
- $(D, A) : 4$,
- $(A, E) : 5$.

Steps:

1. **Initial Spanning Tree:** Use DFS to find a spanning tree: $T = \{(A, B), (B, C), (C, D)\}$.
2. **Process Extra Edges:**
   - Extra edges: (D, A), (A, E).
   - Add (D, A): Creates a cycle $A \to B \to C \to D \to A$. Remove (C, D) (heaviest edge).
   - Add (A, E): No cycle is formed.
3. **Final MST:** $T = \{(A, B), (B, C), (D, A), (A, E)\}$.

Conclusion

This algorithm efficiently computes the MST for very sparse graphs in $O(n)$ time by leveraging the graph's structure and focusing only on the extra edges. It avoids expensive operations like sorting and ensures correctness using the cycle property.

Problem 5-3: Shortest Paths

The following exercises are unrelated.

1. Suppose that a weighted, directed graph (V, E) has a negative-weight cycle that is reachable from some designated source vertex $s \in V$. Give an efficient algorithm to list the vertices of one such cycle and analyze the running time. If you want to use any algorithms you already know, such as Bellman-Ford or DFS, you should apply them as a black box (i.e., don't explain how they work). The algorithm I'm looking for should be quite short given appropriate subroutines. But it's not obvious. I'm not asking you to argue correctness, but you should think about how you would do so. If you are unable to come up with even a hand-wavy correctness argument, there's a good chance your algorithm doesn't work.

2. We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associate value $r(u, v)$, which is a real number in the range $0 < r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will NOT fail. The reliability of the path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the probability that none of the edges along the path fails. We assume that edges failing is independent, so the reliability of the path is given by

$$r(p) = \prod_{i=1}^{k} r(v_{i-1}, v_i).$$

   Conversely, the probability that some edge on the path fails is

$$1 - r(p) = 1 - \prod_{i=1}^{k} r(v_{i-1}, v_i).$$

   Give an efficient algorithm to find the most reliable path between two given vertices, i.e., the path with lowest failure probability. Model this as a shortest path problem. You have to specify (1) the graph (which should be trivial), (2) the edge weights, and (3) which shortest path algorithm you should use.

3. Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

Answer a

To identify the vertices of a negative-weight cycle in a directed graph $G = (V, E)$ that is reachable from a given source vertex s, we use the **Bellman-Ford algorithm** to detect the cycle and predecessor pointers to trace its vertices.

Approach

1. **Execute the Bellman-Ford Algorithm:**
   - Run the Bellman-Ford algorithm to calculate the shortest paths from the source s to all vertices.
   - If the algorithm detects a negative-weight cycle during the $|V|$-th iteration, proceed to trace the cycle. Otherwise, no negative-weight cycle exists that is reachable from s.
2. **Identify a Vertex in the Cycle:**
   - Select a vertex v that was updated in the $|V|$-th iteration of Bellman-Ford. This vertex is guaranteed to belong to a negative-weight cycle.
3. **Locate the Cycle:**
   - Follow the predecessor pointers from v for $|V|$ steps to ensure you are within the cycle. This accounts for any additional paths leading to the cycle.
4. **Trace the Cycle Vertices:**
   - Starting from v, follow the predecessor pointers until v is revisited. Record all the vertices encountered during this traversal, as they constitute the negative-weight cycle.

Algorithm Steps

[label=0.]Run Bellman-Ford from source vertex s:

1. • If no negative-weight cycle is detected, return "No cycle detected."
2. Identify a vertex v updated during the $|V|$-th iteration.
3. Backtrack using predecessor pointers for $|V|$ steps to ensure the vertex is part of the cycle.
4. From v, follow predecessor pointers until v is reached again, recording all visited vertices.
5. Return the list of vertices in the cycle.

Analysis

1. **Time Complexity:**
   - The Bellman-Ford algorithm runs in $O(VE)$.
   - Tracing the cycle vertices using predecessor pointers takes $O(V)$.
   - Overall time complexity: $O(VE)$.
2. **Correctness:**
   - If a vertex v is updated during the $|V|$-th iteration of Bellman-Ford, it is guaranteed to belong to a negative-weight cycle.
   - Tracing the cycle with predecessor pointers ensures that all vertices in the cycle are accurately identified.

Example

Graph:

$$V = \{A, B, C, D\}, \quad E = \{(A, B, 1), (B, C, -2), (C, D, 1), (D, B, 1)\}$$

Source: $s = A$

1. **Bellman-Ford Execution:**
   - The algorithm identifies a negative-weight cycle involving B, C, and D.
2. **Cycle Detection:**
   - Vertex B is flagged during the $|V|$-th iteration.
3. **Trace the Cycle:**
   - Using predecessor pointers, backtracking reveals the cycle: $B \to C \to D \to B$.
4. **Output:**
   - The negative-weight cycle is B, C, D.

This method leverages the Bellman-Ford algorithm to detect negative-weight cycles and efficiently traces the cycle's vertices using predecessor pointers. The algorithm has a time complexity of $O(VE)$ and guarantees accurate identification of the cycle.

Answer b

We are given a directed graph $G = (V, E)$, where each edge $(u, v) \in E$ has a reliability value $r(u, v)$. This value represents the probability that the connection from u to v will not fail, with $0 < r(u, v) \leq 1$.

The goal is to find the path between two vertices s (source) and t (destination) that has the highest reliability:

$$r(p) = \prod_{i=1}^{k} r(v_{i-1}, v_i).$$

Alternatively, this is the path where the failure probability, $1 - r(p)$, is minimized:

$$1 - r(p) = 1 - \prod_{i=1}^{k} r(v_{i-1}, v_i).$$

To solve this, we transform the problem into a shortest path problem using logarithms.

1. Transform the Problem

To maximize the reliability of a path:

$$r(p) = \prod_{i=1}^{k} r(v_{i-1}, v_i),$$

we take the logarithm of the probabilities to convert the product into a sum:

$$\log \left(\prod_{i=1}^{k} r(v_{i-1}, v_i) \right) = \sum_{i=1}^{k} \log(r(v_{i-1}, v_i)).$$

To simplify the problem, we use negative logarithms for the edge weights:

$$w(u, v) = -\log(r(u, v)).$$

This transformation ensures that maximizing $r(p)$ is equivalent to minimizing:

$$w(p) = \sum_{(u,v) \in p} w(u, v).$$

2. Solve as a Shortest Path Problem

Finding the most reliable path becomes finding the shortest path with weights $w(u, v) = -\log(r(u, v))$. Since the weights $w(u, v)$ are non-negative, we can use **Dijkstra's algorithm** to solve the problem.

3. Compute Path Reliability

After finding the shortest path p, the reliability of the path can be calculated as:

$$r(p) = \exp\left(-\sum_{(u,v)\in p} w(u,v)\right).$$

Steps

1. **Input:** The graph $G = (V, E)$ with reliability values $r(u, v)$, and the source s and destination t.
2. **Transform Edge Weights:** For each edge (u, v), compute:
$$w(u, v) = -\log(r(u, v)).$$
3. **Run Dijkstra's Algorithm:** Use the transformed weights $w(u, v)$ to find the shortest path from s to t.
4. **Calculate Path Reliability:** Let p be the shortest path found. Compute:
$$r(p) = \exp\left(-\sum_{(u,v)\in p} w(u,v)\right).$$

Time Complexity

- **Transformation:** Calculating $w(u, v) = -\log(r(u, v))$ for all edges takes $O(E)$.
- **Shortest Path Calculation:** Using Dijkstra's algorithm with a priority queue has a time complexity of $O((V + E)\log V)$.
- **Reliability Calculation:** Summing the weights of the edges in the path takes $O(k)$, where k is the number of edges in the path.

Total Complexity: $O((V + E)\log V)$.

Example

Graph:

$$V = \{A, B, C, D\}, \quad E = \{(A, B), (B, C), (A, C), (C, D)\}.$$

Reliabilities:

$$r(A, B) = 0.9, \quad r(B, C) = 0.8, \quad r(A, C) = 0.7, \quad r(C, D) = 0.95.$$

Source: $s = A$, **Destination:** $t = D$.

1. **Transform Edge Weights:**

$$w(A, B) = -\log(0.9), \quad w(B, C) = -\log(0.8),$$
$$w(A, C) = -\log(0.7), \quad w(C, D) = -\log(0.95).$$

2. **Run Dijkstra's Algorithm:** Find the shortest path from A to D using the transformed weights.

3. **Output:** The path is $A \to B \to C \to D$, and the reliability is:

$$r(p) = \exp\left(-\sum_{(u,v)\in p} w(u, v)\right).$$

Answer c

We are given a weighted, directed graph $G = (V, E)$ with the following properties:

- Edges leaving the source vertex s may have **negative weights**.
- All other edges in the graph have **nonnegative weights**.
- The graph does not contain any **negative-weight cycles**.

The goal is to prove that Dijkstra's algorithm correctly computes the shortest paths from s to all vertices in V under these conditions.

Dijkstra's Algorithm

Dijkstra's algorithm works as follows:

1. Initialize $d[s] = 0$, $d[v] = \infty$ for all $v \neq s$, and set $S = \emptyset$ (the set of finalized vertices).
2. Repeatedly select the vertex $u \notin S$ with the smallest tentative distance $d[u]$.
3. Add u to S, and relax all outgoing edges (u, v) by updating $d[v]$ if a shorter path is found via u.

Key principles that ensure correctness:

1. **Loop Invariant:** At the start of each iteration, the shortest path distance $d[u] = \delta(s, u)$ (the true shortest distance from s) for all vertices $u \in S$.

2. **Relaxation Property:** The relaxation step ensures that $d[v] \geq \delta(s, v)$ for all vertices v.

3. **Greedy Selection:** The algorithm selects the vertex u with the smallest tentative distance, guaranteeing that $d[u] = \delta(s, u)$ once u is finalized.

Addressing Negative Weights

- **Impact of Negative Weights:** Negative weights are restricted to edges (s, v). These edges are relaxed immediately after s is processed, ensuring the shortest path estimates for the direct neighbors of s are correct.
- **Nonnegative Weights Elsewhere:** All other edges in the graph have nonnegative weights. This ensures that once a vertex u is finalized, no shorter path can exist to u through any other vertex.

Proof of Correctness

Initialization

Initially, $d[s] = 0$, $d[v] = \infty$ for all $v \neq s$, and $S = \emptyset$. The loop invariant holds trivially since no vertices have been processed yet.

Maintenance

Assume the invariant holds at the start of an iteration: for all $u \in S$, $d[u] = \delta(s, u)$. Let u be the vertex selected with the smallest $d[u]$ among $V - S$. By the greedy property, u has the correct shortest distance $\delta(s, u)$, because:

- Any shorter path to u would have to pass through a vertex not in S, which contradicts the minimality of $d[u]$.

After adding u to S, all outgoing edges (u, v) are relaxed. This ensures $d[v] \geq \delta(s, v)$ for all v.

Handling Negative Weights

Negative weights are limited to edges (s, v). When s is processed, these edges are relaxed, correctly updating $d[v]$ for neighbors of s. For $u \neq s$, all outgoing edges satisfy $w(u, v) \geq 0$. Relaxing these edges maintains the correctness of $d[v]$.

Termination

The algorithm terminates when all vertices are added to S. At this point, $d[u] = \delta(s, u)$ for all u.

Contradiction Argument

If Dijkstra's algorithm were incorrect, there would exist a vertex u for which $d[u] \neq \delta(s, u)$ when u is added to S. Let u be the first such vertex. Consider the shortest path p from s to u, which can be decomposed as:

$$p = s \to x \to u, \quad \text{where } x \in S \text{ and } u \notin S.$$

When x was added to S, the edge (x, u) was relaxed, ensuring:

$$d[u] \leq d[x] + w(x, u) = \delta(s, x) + w(x, u) = \delta(s, u).$$

This contradicts $d[u] \neq \delta(s, u)$. Thus, $d[u] = \delta(s, u)$ for all u when added to S.

Conclusion

Dijkstra's algorithm correctly computes shortest paths from s under the given conditions because:

1. Negative weights are restricted to edges leaving s, and they are relaxed immediately during the first iteration.
2. All other edges have nonnegative weights, ensuring no shorter paths are introduced after vertices are finalized.
3. The absence of negative-weight cycles ensures that the algorithm terminates without inconsistencies.

Thus, the algorithm works correctly in $O((V+E) \log V)$ time using a priority queue.

Problem 5-4: Balancing Trees by Rebuilding

1. A 1/2-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes 1/2-balanced. Your algorithm should run in time $\Theta(x.\text{size})$, and it can use $O(x.\text{size})$ auxiliary storage. Hint: you'll probably want to use a size-$(x.\text{size})$ array here as auxiliary storage. You'll also probably want to start your algorithm with an in-order tree walk.
2. Show that an n-node α-balanced binary search tree has $O(\log n)$ height. (Recall α is a constant.) It would be equivalent to show that a search in an n-node α-balanced binary search tree takes $O(\log n)$ time in the worst case, so you may do that if you prefer. However, it's more convenient to refer to height in a later problem part.

3. Argue that any binary search tree has nonnegative potential and that a 1/2-balanced tree has potential 0.

4. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take 0 amortized time to rebuild a subtree that is not α-balanced? Don't use big-O notation here. In other words, how large must c be in order for the change in potential caused by rebuilding to be sufficient to pay for the rebuild? Note the assumption that the tree is not α-balanced before the rebuild — that's important.

5. Show that inserting a node into an n-node α-balanced tree costs $O(\log n)$ amortized time. Hint: it's easier if you break up the insertion into steps: perform the regular BST insert, pay for any changes to potential, and pay for any subtree rebuilding.

Answer a

Goal: Rebuild the subtree rooted at x so that it becomes 1/2-balanced, using $\Theta(x.\text{size})$ time and $O(x.\text{size})$ auxiliary storage.

Algorithm:

1. **Perform an In-Order Traversal:**
   - Traverse the subtree rooted at x in in-order and store the elements in an auxiliary array A of size $x.\text{size}$.
   - This takes $O(x.\text{size})$ time since every node is visited once.
2. **Rebuild the Tree:**
   - Select the median element of A as the root of the subtree.
   - Recursively apply the same process to the left and right halves of A to construct the left and right subtrees.
3. **Output:**
   - The resulting subtree is 1/2-balanced because the median ensures that the left and right subtrees have sizes at most $x.\text{size}/2$.

Time Complexity:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1),$$

which solves to $T(n) = O(n)$.

Auxiliary Space: The array A uses $O(x.\text{size})$ space.

Answer b

Statement: An n-node α-balanced binary search tree has height $O(\log n)$.

Proof:

1. **Node Relationship:** For any node x, the left and right subtree sizes satisfy:

$$(x.\text{left}).\text{size} \leq \alpha \cdot x.\text{size}, \quad (x.\text{right}).\text{size} > (1-\alpha) \cdot x.\text{size}.$$

2. **Recursive Subtree Size:** At depth d, the subtree size decreases by at least a factor of α, so the size at depth h is:

$$n \cdot \alpha^h \geq 1 \quad \Longrightarrow \quad h \leq \log_{1/\alpha} n.$$

3. **Conclusion:** The height of the tree is $O(\log n)$.

Answer c

Nonnegative Potential: The potential function is defined as:

$$\Phi(T) = \sum_{x \in T : \Delta(x) \geq 2} c \cdot \Delta(x),$$

where $\Delta(x) = |(x.\text{left}).\text{size} - (x.\text{right}).\text{size}|$.

Since $\Delta(x) \geq 0$ for all nodes x, the potential $\Phi(T)$ is a sum of nonnegative terms. Therefore:

$$\Phi(T) \geq 0 \quad \text{for any binary search tree } T.$$

Potential of a $1/2$-Balanced Tree:

- For a $1/2$-balanced tree, every node x satisfies:

$$(x.\text{left}).\text{size} \leq \frac{1}{2} \cdot x.\text{size}, \quad (x.\text{right}).\text{size} \leq \frac{1}{2} \cdot x.\text{size}.$$

  This implies:

$$\Delta(x) = |(x.\text{left}).\text{size} - (x.\text{right}).\text{size}| \leq 1.$$

- Since $\Delta(x) < 2$ for all nodes in a $1/2$-balanced tree, no nodes contribute to the sum in $\Phi(T)$. Thus:

$$\Phi(T) = 0.$$

Equation for Potential Change: The potential change during a rebuild is given by:

$$c_i = c + \Phi(D_i) - \Phi(D_{i-1}),$$

where:

- c_i: Cost of the operation.
- $\Phi(D_i)$: Potential after the rebuild.
- $\Phi(D_{i-1})$: Potential before the rebuild.

For a 1/2-balanced tree, $\Phi(T) = 0$, ensuring that the change in potential $\Delta\Phi$ offsets the cost of rebuilding.

Answer d

Rebuilding Cost: Rebuilding an m-node subtree takes m units of work.

Imbalance and Potential Contribution: For a node x in the subtree, the imbalance $\Delta(x)$ satisfies:

$$\Delta(x) \geq \alpha \cdot m - ((1-\alpha)m - 1) = (2\alpha - 1)m + 1.$$

Condition for c: The potential decrease must be sufficient to pay for the rebuilding:

$$m \leq c \cdot ((2\alpha - 1)m + 1)\,.$$

Rearranging:

$$c \geq \frac{1}{2\alpha}.$$

Answer e

Steps:

1. **Perform the Standard BST Operation:** Search for the location to insert the new element or delete an existing one. This takes $O(\log n)$ time.
2. **Update Potential:** Only the ancestors of the modified node have their $\Delta(x)$ values affected. At most $O(\log n)$ nodes are affected in an α-balanced tree.
3. **Rebuild Subtrees:** If a subtree becomes α-imbalanced, rebuild it to make it 1/2-balanced. Each rebuild is paid for by the potential decrease, as shown in part (d).

Amortized Cost: The total cost for insertion or deletion is:

$$O(\log n) + O(\log n) = O(\log n).$$

Summary

1. Rebuilding a subtree to be 1/2-balanced takes $O(x.\text{size})$ time and $O(x.\text{size})$ space.
2. The height of an α-balanced tree is $O(\log n)$.
3. $\Phi(T) \geq 0$ for all trees, and $\Phi(T) = 0$ for 1/2-balanced trees.
4. $c \geq \frac{1}{2\alpha}$ ensures zero amortized cost for rebuilding.
5. The amortized cost of insertion or deletion is $O(\log n)$.