HW8

Keerthana Golla

October 2023

1 Problem -1

Of course, we could just recompute the minimum spanning tree from scratch in O($|E| + |V|log|V|$) time, but we can do better by considering the following 4 scenarios . We can easily say that there will be only four scenarios - edge being in mst and edge not in mst , which is 2 cases and along with it edge decreased and edge increased - so 2*2=4 total 4 scenarios discussed below.

Part (a) - Decreasing Edge Weight and the edge is in T

In this case, we have a minimum spanning tree (MST) T, and the weight of an edge $e = (u, v)$ in T has been decreased by d units. The algorithm works as follows:

It calculates the weight of the new MST after the decrease, which is $W - d$, where W is the original weight of the MST.

The key insight here is that T remains an MST after the weight decrease. To prove this, we need to consider the properties of a minimum spanning tree. T is minimal, which means there is no other spanning tree with a lower weight. If there were another MST T_0 without edge e, it would have a weight less than $W - d$ because the weight of e has been decreased. This would contradict the minimality of T.

Therefore, T remains minimal even after decreasing the weight of edge e. Thus, it remains a minimum spanning tree of the updated graph. The algorithm's correctness is established by this observation.

Part (b) - Increasing Edge Weight and the edge is not in T

In this case, we have an MST T, and the weight of an edge e not in T has been increased. The algorithm efficiently updates the MST as follows:

It simply notes that T, which was produced by an algorithm like Kruskal's or Prim's, remains an MST.

The reason for this is that algorithms like Kruskal's and Prim's construct an MST by considering edges in ascending order of weight. If the edge e was not originally in T, it means it wasn't considered during the construction of T in the first place. When its weight increases, it still won't be considered, and T will remain an MST.

Hence, the correctness is straightforward in this case: the original MST T remains an MST even after the weight increase of edge e.

Part (c) - Increasing Edge Weight and the edge is in T

In this case, we have an MST T, and the weight of an edge $e = (u, v)$ in T has been increased. The algorithm to update the MST works as follows:

It determines the subtrees T_u and T_v by using a BFS/DFS from both u and v.

It identifies the minimum-weight edge e_0 connecting T_u and T_v in the original graph.

The correctness of this algorithm relies on two key observations:

Using BFS to determine T_u and T_v ensures that the partitioning of the vertices into two subtrees is accurate.

The minimum-weight edge e_0 connecting T_u and T_v in the original graph remains the minimum-weight edge in the updated graph.

These observations ensure that the new MST T_0, which replaces e with e_0, is still minimal. Since e_0 was the minimum-weight edge connecting the two subtrees before the weight increase, this choice maintains the minimal spanning property of the new MST. Thus, the algorithm produces a correct MST.

Part (d) - Decreasing Edge Weight and the edge is not in T

In this case, we have an MST T, and the weight of an edge $e = (u, v)$ in G has been decreased. The algorithm to update the MST works as follows:

It adds the edge e to the MST T, creating a unique cycle C.

It identifies the maximum-weight edge $e_{\max}$ on the cycle C.

It removes the edge $e_{\max}$ from T, resulting in the new MST T_0.

The correctness of this algorithm can be explained as follows:

When you add the edge e to T, you create a unique cycle. This cycle must contain the edge e because it is the edge that was changed.

Removing the maximum-weight edge $e_{\max}$ from the cycle preserves the minimality of the spanning tree. This is because the edge with the maximum weight on a cycle cannot be part of any MST (it forms a cycle and can be replaced with a lighter edge without breaking the spanning tree property).

Therefore, the new MST T_0 obtained after the weight decrease of e is still a minimum spanning tree of the updated graph, making the algorithm correct.

Time Complexity for Updating MST

Part a,b:

Since the MST structure remains unchanged in a,b scenarioes like explained above, the time complexity is very low. It's essentially $O(1)$, as no modifications to the MST are needed.

Part c:

The time complexity for this case depends on how you identify the minimum-weight edge connecting the two subtrees. If you use a standard Breadth-First Search (BFS) or Depth-First Search (DFS) to find the subtrees and then find the minimum-weight edge, the time complexity is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

Part d:

This case is also relatively efficient. Adding an edge and identifying the maximum-weight edge on a cycle are typically $O(V)$ operations.Note MST will have $|V|$-1 edges. Thus, the time complexity for this case is $O(V)$. Maximum time is taken here by part c - $O(V + E)$ for using the DFS/BFS

2 Problem -2

```
def modified_bellman_ford(G, w):
    # Initialize the distances to Infinity for each vertex
    for v in G.V:
        v.d = float('inf')

    # Main loop: Relax edges for |G.V| - 1 times
    for i in range(1, len(G.V)):
        for edge in G.E:
            u, v = edge
            # Calculate the minimum distance (Relaxing)
            if v.d > min(w(u, v), w(u, v) + u.d)
                v.d = min(w(u, v), w(u, v) + u.d)
                v.pi = u.pi
```

We use bellman-ford algorithm for this problem but only change is we initialize all the distances to infinity and a small change in relaxing i.e, To get to a vertex v, you have 2 options:

1. Take the minimum incoming edge from u to v, i.e., $w(u, v)$.
2. Get to vertex u and then take the edge from u to v, i.e., $u.d + w(u, v)$.

You have to find the minimum of these 2 options, which is given by $\min(w(u, v), u.d + w(u, v))$.

Correctness:

Initialization: The algorithm initializes all vertex distances to infinity, which is a valid starting point for finding shortest paths because it ensures that the initial distances are set to values that are greater than or equal to the actual shortest-path distances. This initialization step is consistent with the standard Bellman-Ford algorithm. Except in the standard Bellman-Ford algorithm, we will initialize zero for the start node. Here, since there is no particular start node, we assign it to infinity, but due to the change in the relaxation step after $|V| - 1$ steps, we will find the required solution.

Main Loop: The main loop runs for $|V| - 1$ iterations, where $|V|$ is the number of vertices. This is a fundamental characteristic of the Bellman-Ford algorithm and guarantees that all possible shortest paths are considered.

Relaxation Step Modification: In the relaxation step, the modification involves considering two options for updating distances: - The first option is to take the weight of the edge from u to v ($w(u, v)$). - The second option is to go through vertex u first, and then take the edge from u to v ($u.d + w(u, v)$). The minimum of these two options is calculated using the min function, which ensures that the algorithm correctly updates the distance to vertex v based on the shortest path found so far. This modification is a valid adaptation to the standard Bellman-Ford algorithm, as it still adheres to the fundamental principles of relaxation.

Overall: The correctness of this modified algorithm is established by its adherence to the fundamental principles of the Bellman-Ford algorithm, including proper initialization, a correct number of iterations in the main loop, and a valid relaxation step. Therefore, the provided algorithm is correct for finding the shortest paths from a single source vertex to all other vertices in the graph, with the modification ensuring that it considers both direct edges and paths through other vertices.

Running time analysis:

The time complexity of the provided pseudo-code for the modified Bellman-Ford algorithm is as follows:
1. Initializing distances for each vertex takes $O(|V|)$ time, where $|V|$ is the number of vertices.
2. The main loop runs for $|V|$ - 1 iterations, and within each iteration, it iterates through all edges in G.E.
a. For a graph with $|E|$ edges, the inner loop takes $O(|E|)$ time in each iteration.
3. The total time complexity of the algorithm is $O(|V| \cdot |E|)$, where $|V|$ is the number of vertices, and $|E|$ is the number of edges.

So, the time complexity of this algorithm is $\mathcal{O}(|V| \cdot |E|)$.