ASSIGNMENT – 6

Introduction to Algorithms (CS 430)

1.

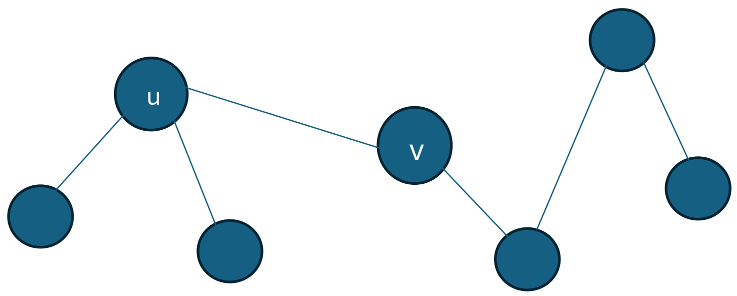
a) Let the MST for graph G and largest weight edge of the MST is $W(u, v), b = a$ with edge $W(u, v)$, which is not the part of the bottleneck spanning tree of the same graph G.

Case 1: Assume, the largest edge $W^1(u^1, v^1) = b$, in the bottleneck spanning tree is larger than edge $W(u,v), b > a$.

- In this case, it's clear that the edge $W1(u1, v1)$ can't be the return value of the bottleneck spanning tree of graph G. Otherwise there exists another tree structure that provides a smaller largest edge. It's a conflict of the definition of bottleneck spanning tree.

Case 2: Assume the largest edge $W^1(u^1,v^1) = b$ in the bottleneck spanning tree is smaller than edge $W(u,v), b < a$.

- In this case, based on the definition of the tree structure, consider that the edge $W(u, v)$, in MST separates all vertices into two parts, left sub-tree T_l and right sub-tree T_r. Here, no cycles exist in tree structures, and all edges connect two sub-parts of the tree graph.
- tree graph.



According to the bottleneck spanning tree, the relationship of $e(x, y) = c$ and $W^1(u^1,v^1) = b$ should be $c <= b$. At the same time, according to the MST definition, the relationship of $e(x, y) = c$ and $W (u,v) = a$ should be $c > a$.

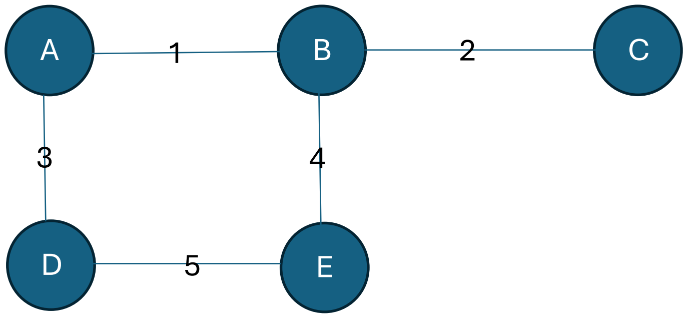
Conclusion:

Therefore, $a < b$, because $c <= b$ and $c > a$. It indicates the length of $W(u, v)$ is smaller than the length of $W^1(u^1,v^1)$. This is a conflict. The largest edge $W^1(u^1,v^1) = b$ in the bottleneck spanning tree is smaller than edge $W(u, v), b < a$. Therefore we proved that $W(u,v)$ is also in the bottleneck spanning tree and is the largest edge of it. Therefore, MST is also a bottleneck spanning tree.

b)

To prove that a Bottleneck Spanning Tree might not be a Minimum Spanning Tree, we can provide a counterexample.

Consider a graph G with the following edge weights:



If we construct a minimum spanning tree for G, it will consist of the edges (A-B), (B-C), (A-D), and (D-E), with a total weight of 3+2+5=10.

However, if we consider the bottleneck spanning tree with the minimum largest edge weight, it will consist of the edges (A-D), (D-E), and (B-C), with the largest edge weight being 5.

In this case, the bottleneck spanning tree is not the same as the minimum spanning tree since the edge (A-B) is excluded. The minimum spanning tree has a smaller total weight than the bottleneck spanning tree.

Hence, we have proved that a bottleneck spanning tree might not be a minimum spanning tree.

2.

Given, a weighted directed graph *G = (V, E)* where *(u, v)* $\in E$ is the only arc with negative weight *w(u, v) < 0,* and *s, u, v, t* are all different vertices, we can use the following algorithm to find the shortest path from *s* to *t:*

```
function ShortestPath(G, s, t, u, v):
    // G is the original graph with all edges
    // s is the start vertex, t is the end vertex
    // (u,v) is the only edge with negative weight

    // Create G' by removing edge (u,v) from G
    G' = G - {(u,v)}

    // Case 1: Path not including (u,v)
    d1 = Dijkstra(G', s, t)

```

```
    // Case 2: Path including (u,v)
    d_su = Dijkstra(G', s, u)
    d_vt = Dijkstra(G', v, t)

    // Check if a path exists through (u,v)
    if d_su != infinity and d_vt != infinity:
        d2 = d_su + w(u,v) + d_vt
    else:
        d2 = infinity

    // Check for negative cycle
    if d_su + w(u,v) + Dijkstra(G', v, u) < 0:
        return "Negative cycle detected"

    // Return the shorter of the two paths
    return min(d1, d2)

function Dijkstra(G, start, end):
    // Standard Dijkstra's algorithm implementation
    // Returns the shortest distance from start to end in G
    // Returns infinity if no path exists
    // Assumes all edge weights in G are non-negative
    // Implementation details omitted for brevity
```

This algorithm is efficient because:

- It considers both possible cases: the shortest path either includes or doesn't include the negative edge (u,v).
- It uses Dijkstra's algorithm only on G', which has no negative edges, ensuring its correct application.
- It handles the case where no path exists through (u,v).
- It detects and reports negative cycles, which is crucial as the concept of a "shortest path" is not well-defined in graphs with negative cycles.
- The time complexity is dominated by Dijkstra’s algorithm calls, making it efficient for graphs without negative cycles.

3.

Given,

- A firm trades shares in n different companies.
- For each pair i != j, there's a trade ratio r_{ij} (1 share of company i trades for r_{ij} shares of company j).
- An opportunity cycle is a trading cycle that increases the number of shares (product of ratios > 1).
- We have an n × n chart R of trade ratios where R(i,j) = r_{ij}.

Pseudo-code to check whether such an opportunity cycle exists:

```
function FindOpportunityCycle(R):
    n = length(R)  // number of companies

    // Construct a graph G
    G = new Graph(n)
    for i = 1 to n:
        for j = 1 to n:
            if i != j:
                G.addEdge(i, j, -log(R[i][j]))

    // Run Bellman-Ford algorithm from any vertex (e.g., vertex 1)
    return BellmanFordNegativeCycle(G, 1)

function BellmanFordNegativeCycle(G, start):
    n = G.numberOfVertices()
    distance = new Array(n, infinity)
    distance[start] = 0

    // Relax edges n-1 times
    for i = 1 to n-1:
        for each edge (u, v) with weight w in G.edges:
            if distance[u] != infinity and distance[u] + w < distance[v]:
                distance[v] = distance[u] + w

    // Check for negative weight cycle
    for each edge (u, v) with weight w in G.edges:
        if distance[u] != infinity and distance[u] + w < distance[v]:
            return true  // Negative cycle found (opportunity cycle exists)

    return false  // No negative cycle (no opportunity cycle)
```