CS430 – FINAL

(A20561894, cnynavarapu)

1.

Greedy Algorithm 1: Choosing the Item with the Largest Value
Problem Instance:

- Knapsack Capacity C=2
- Items:
  - *Item 1:* Size s_1=1, Value v_1 = 1
  - *Item 1:* Size s_2=1, Value v_2 = 1
  - *Item 1:* Size s_3=2, Value v_3 = 2B

Explanation:

- **Greedy Choice:** The algorithm will choose Item 3 because it has the highest value 2B.
- **Optimal Solution:** The best solution is indeed to choose Item 3, filling the knapsack completely and maximizing the total value 2B.

In this case, the greedy solution aligns with the optimal solution, so the ratio is 1. However, to make the greedy solution arbitrarily worse, we can modify the example

Modified Example for Greedy Algorithm 1:

Problem Instance:

- Knapsack Capacity C=2
- Items:
  - *Item 1:* Size s_1=1, Value v_1 = B (Unit value 1)
  - *Item 1:* Size s_2=1, Value v_2 = B (Unit value 1)
  - *Item 1:* Size s_3=2, Value v_3 = 1 (Unit value B)

Explanation:

- **Greedy Choice:** The algorithm will choose Item 1 and Item 2 because they have higher individual values (both B)
- **Optimal Solution:** The optimal solution would be to choose Items 1 and 2, resulting in a total value of 2B. However, if the value of Item 3 were larger than 2B (e.g.,2B+1), the optimal choice would be Item 3, resulting in a value of 2B+1.

In this scenario, the greedy solution would be 2B while the optimal solution could be arbitrarily large depending on the value of B.

Greedy Algorithm 3: Choosing the Item with the Largest Unit Value
Problem Instance:

- Knapsack Capacity C=2

- Items:
    - *Item 1:* Size s_1=1, Value v_1 = 1 (Unit value 1)
    - *Item 1:* Size s_2=1, Value v_2 = 1 (Unit value 1)
    - *Item 1:* Size s_3=2, Value v_3 = 2B (Unit value B)

Explanation:

- **Greedy Choice:** The algorithm will choose Item 3 because it has the highest unit value B.
- **Optimal Solution:** The best solution is to choose Item 3, with a total value of 2B.

Modified Example for Greedy Algorithm 3:
Problem Instance:

- Knapsack Capacity C=2
- Items:
    - *Item 1:* Size s_1=1, Value v_1 = B (Unit value B)
    - *Item 1:* Size s_2=1, Value v_2 = B (Unit value B)
    - *Item 1:* Size s_3=2, Value v_3 = 1 (Unit value 0.5)

Explanation:

- **Greedy Choice:** The algorithm will choose Item 1 and Item 2 because they have the highest unit value B.
- **Optimal Solution:** The optimal solution, however, could involve choosing an item that has a lower unit value but fits better into the knapsack size constraint, resulting in a higher total value.

These illustrate the potential downfall of using greedy algorithms for the Knapsack problem, where the heuristic choice does not always lead to an optimal solution, especially when values can be made arbitrarily large (as indicated by B in the examples).

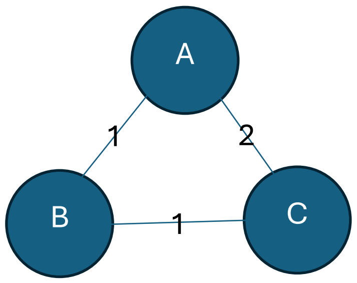
2.
Conjecture:
If G contains exactly two edges e_1 and e_2 with the same minimum weight among all edges of G, then every Minimum Spanning Tree (MST) of G contains both e_1 and e_2

Analysis:

To disprove the conjecture, we need to find a counterexample where a graph G has exactly two edges with the same minimum weight, but not every Minimum Spanning Tree (MST) of G contains both of these edges.

In this example, both MST1 and MST2 include both e1 and e2. Thus, this example does not disprove the conjecture.

Let's consider another example to find a valid counterexample:



Vertices: V = { A , B , C }
Edges and weights:

- e1 =(A,B) with weight 1
- e2 =(B,C) with weight 1 (A,C) with weight 2

This graph satisfies the conditions:

- It is a weighted, connected, undirected simple graph.
- It has |V|=3|V|=3, which satisfies |V|>=3|V|>=3.
- It contains exactly two edges e1=(A,B)$e1=(A,B)$ and e2=(B,C)$e2=(B,C)$ with the same minimum weight of 1.

Minimum Spanning Trees (MST)

Let's identify the MSTs for this graph:

MST1: Include edges e1=(A,B) and e2=(B,C)

- Total weight: 1+1=2
- Edges: {(A,B),(B,C)}

MST2: Include edges (A,C) and e2=(B,C)

- Total weight: 2+1=3
- Edges: {(A,C),(B,C)}

Conclusion

Therefore, we can say that the conjecture is false. And the counterexample provided demonstrates that there exist graphs where not every MST contains both of the two edges with the same minimum weight.

3.

Algorithm to solve Jim's problem of finding a minimum weight simple cycle containing a given vertex s in an undirected graph

Simple algorithm(stepwise solution):

1. Run Dijkstra's algorithm from vertex s to find shortest paths to all other vertices. This takes O(|E| + |V| log |V|) time using a Fibonacci heap.
2. For each edge (u,v) not in the shortest path tree:
   - Calculate the cycle weight: dist[u] + w(u,v) + dist[v]
   - Keep track of the minimum weight cycle found
3. Return the minimum weight cycle

Time complexity:

- Dijkstra's: O(|E| + |V| log |V|)
- Checking each non-tree edge: O(|E|)
- Total: O(|E| + |V| log |V|)

This meets the required time bound of O(|V||E| + |V|^2 log |V|).

This approach guarantees finding a simple cycle (no repeated vertices) containing s with minimum total weight.

PSEUDO-CODE:

```
function FindMinimumWeightCycle(G, s):
    minCycle = null, minWeight = ∞
    for each v in G.V:
        dist, pred = DijkstraFibonacciHeap(G, v)
        for each (u, w) in G.E:
            if pred[w] ≠ u and pred[u] ≠ w:
                cycleWeight = dist[u] + weight(u, w) + dist[w]
                    if cycleWeight < minWeight and (s in {u, w, v} or s in path(v to u) or s in path(v to w)):
                        minWeight = cycleWeight
                        minCycle = ReconstructCycle(v, u, w, pred)
    return minCycle, minWeight if minCycle else "No cycle found"

function DijkstraFibonacciHeap(G, start):
    Initialize dist[v] = ∞, pred[v] = null for all v in G.V
    dist[start] = 0
    Q = FibonacciHeap(G.V)
    while Q is not empty:
        u = Q.extractMin()
        for each neighbor v of u:
            if dist[u] + weight(u, v) < dist[v]:
```

```
20.                     dist[v] = dist[u] + weight(u, v)
21.                     pred[v] = u
22.                     Q.decreaseKey(v, dist[v])
23.         return dist, pred

24. function ReconstructCycle(v, u, w, pred):
25.         cycle = [w, u]
26.         while pred[u] ≠ v:
27.             cycle.prepend(pred[u])
28.             u = pred[u]
29.         cycle.prepend(v)
30.         while pred[w] ≠ v:
31.             cycle.append(pred[w])
32.             w = pred[w]
33.         return cycle
```

Time Complexity: O(|V||E| + |V|^2 log |V|)

- Dijkstra's algorithm runs |V| times: O(|V| * (|E| + |V| log |V|))
- Inner loop checks |E| edges for each vertex: O(|V| * |E|)Total: O(|V||E| + |V|^2 log |V|)

4.

To solve the problem of finding the path with the maximum sum from the peak to the base of the pyramid using dynamic programming, we need to define the recursive function b(i,j) which represents the maximum sum of numbers on the optimal path from the spot (i,j) to the base of the pyramid.

Here is the step-by-step approach to define b(i,j)b(i,j)b(i,j):

1. **Base Case**: When i=n(i.e., at the base of the pyramid), b(i,j) is simply the value at that spot because there are no further levels to traverse.

   - b(n,j)=value(n,j)for 1<=j<=n

2. **Recursive Case**: For other positions, b(i,j)depends on the values at the next level down, either directly below (i+1,j) or diagonally below to the right (i+1,j+1). Thus, the recursive formula is:
   - b(i,j)=value(i,j)+max((b(i+1,j),b(i+1,j+1))
   - Given these definitions, the optimal path sum starting from the top of the pyramid at (1,1) can be found using: **b(1,1)**

Pseudo-Code:

```
funtion findMaxSum(T)
    n ← length(T)
    maxSum ← T[0][0]

    for i  from 1 to  n-1 do
        for j from 0 to i do
            if j == 0 then
                T[i][j] += T[i-1][j]
            else if j == i then
                T[i][j] += T[i-1][j-1]
            else
                T[i][j] += MAX(T[i-1][j-1], T[i-1][j])
            End if
            maxSum ← MAX(maxSum, T[i][j])
        end for
    end for
    return maxSum
end funtion
```

5.

The graph consists of vertices A, B, C, D. With the following directed edges

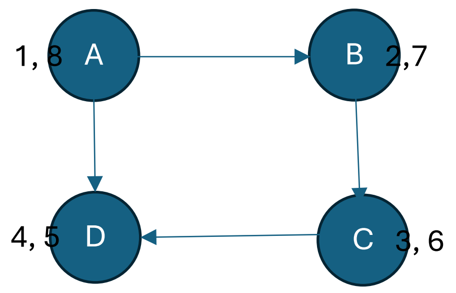
- A→B
- B→C
- C→D
- D→A
- B→D

Conjecture: In a directed graph, if there is a path from vertex u to v, and the discover time of u is earlier than the discover time of v in some depth-first search, then u is an ancestor of v in the depth-first forest produced.

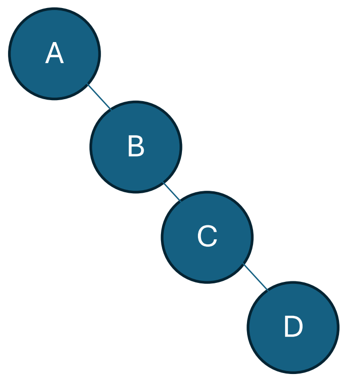
Counter Example:

1. **Vertices and edges**
   - Vertices: A, B, C, D
   - Edges: A→B, B→C, C→D, D→A, B→D
2. **DFS Procedure and Finish Times:**
   - A: Discover = 1, Finish = 8
   - B: Discover = 2, Finish = 7
   - C: Discover = 3, Finish = 6
   - D: Discover = 4, Finish = 5

3. **DFS Traversal** (discovery time, finish time)



4. **DFS Tree**



5. **Traversal Order: A → B → C → D**

Analysis:
Consider the path from B to D (via edge B→D). The discover time of B (2) is earlier than the discover time of D (4).
However, in the DFS tree, B is not an ancestor of D. Instead, D is a child of C in the tree. The direct edge B→D leads to a situation where B can reach D without B being an ancestor of D in the DFS forest.

Conclusion:
This demonstrates that the conjecture does not hold in all cases. The presence of the direct edge B→D creates a situation where B is discovered before D, but B is not an ancestor of D in the depth-first forest. Thus, the counterexample using the provided graph disproves the conjecture.