DAA HW 7

Pranav Tushar Pradhan N18401944 pp3051@nyu.edu

Udit Milind Gavasane N16545381 umg215@nyu.edu

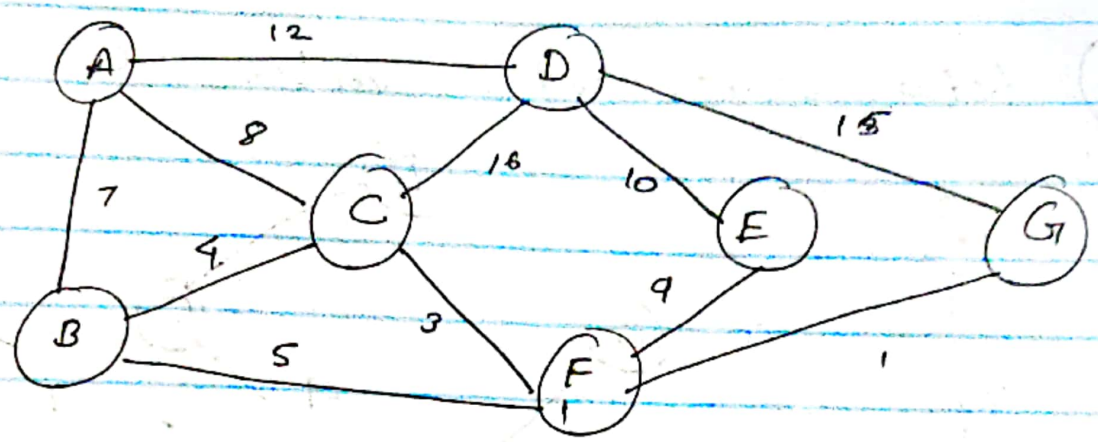
Rohan Mahesh Gore N19332535 rmg9725@nyu.edu

Saavy Singh N16140420 ss19170@nyu.edu

Sashank Badri Narayan N14628839 sb10192@nyu.edu

Homework 7

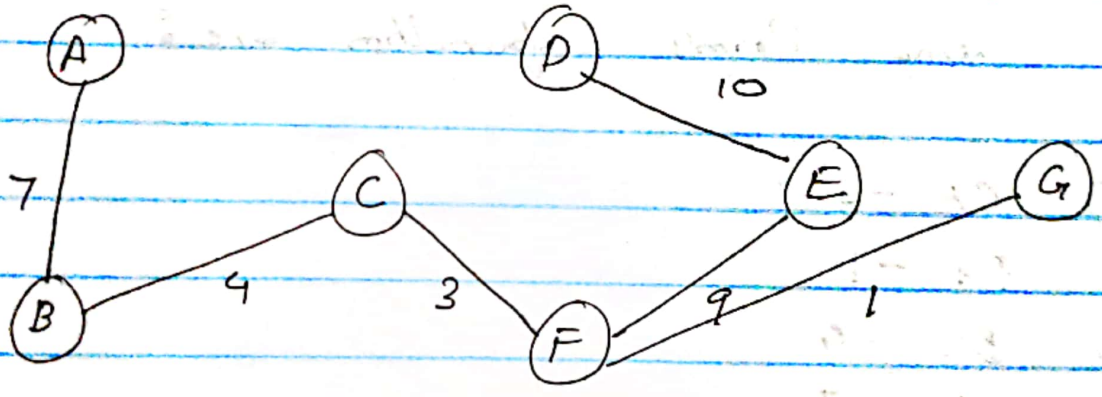
Q.1.



A) Sort edges in ascending order of their weights.

→ 1, 3, 4, 5, 7, 8, 9, 10, 12, 15, 16.

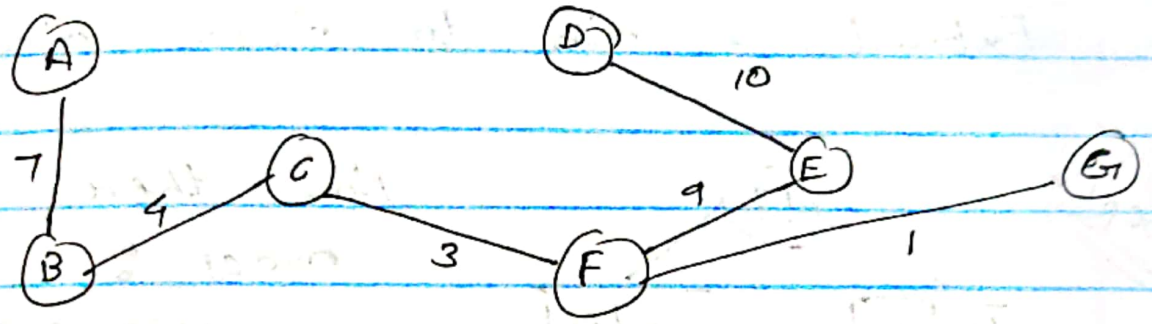
Below is the MST by Kruskal's Algorithm



The first 5 edges added to MST are:

1) FG – 1
2) CF – 3
3) BC – 4
4) AB – 7
5) FE – 9

B.) Lets start at vertex D.
Below is the MST by Prim's Algorithm.



The first five vertices added to MS using Prim's algorithm are:

1) D E – 10
2) E F – 9
3) F G – 1
4) F C – 3
5) C B – 4

Q. 2.

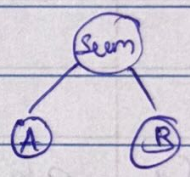
a: 10 , b: 7 , c: 22, d: 46, e: 32, f: 38.

— Push all the nodes in an "implicit binary heap" → implemented by a priority queue.

— At every ~~step~~ we:

① perform "2" extract_min()

② combine as tree → where children are the extracted nodes & parent is sum



③ Push again in p-queue.

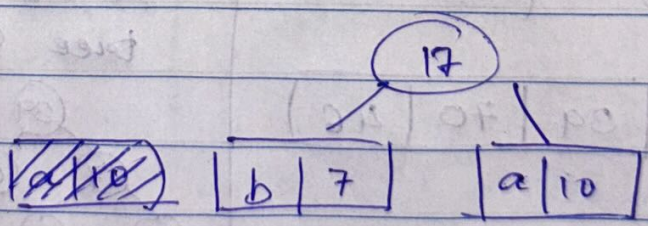
— Step 1:

① Extract_min()

b: 7

a: 10

② Combine



③ Push tree in Q with key = 17.

- Queue becomes.

| 17 | 22 | 32 | ~~38~~ | 46 |

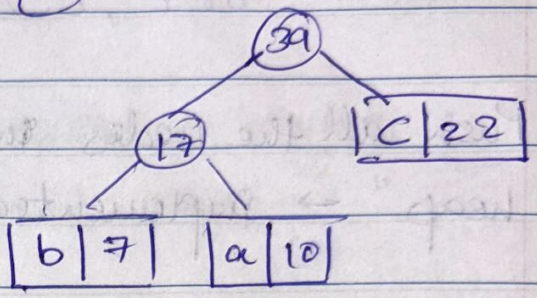
Step 2:

① Extract_min()

{b, a} = 17

c = 22

② Combine



③ P_Queue becomes.

| 39 | 32 | 38 | 46 |

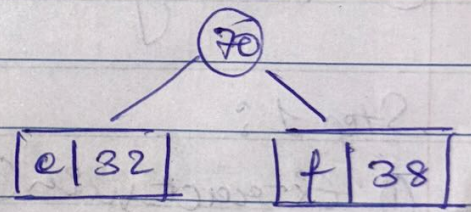
Step 3:

① Extract_min()

e = 32

f = 38

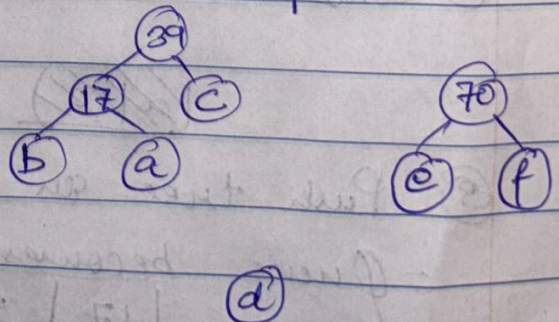
② Combine.



③ P_Queue.

| 39 | 70 | 46 |

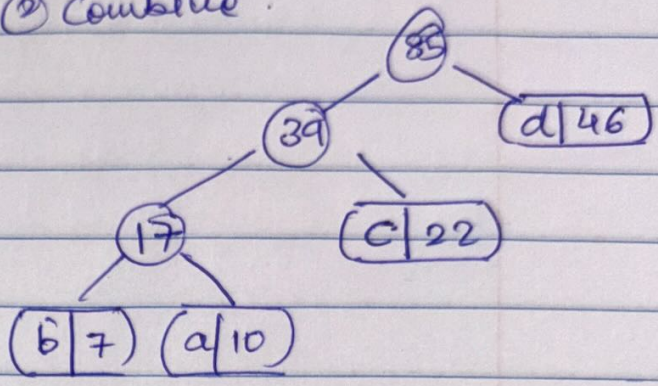
★. Current available tree composition.



Step 4:

① Extract_min()
{b, a, c} = 39
d = 46

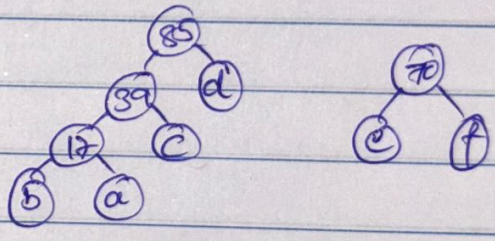
② Combine.



③ P_Queue becomes.

| 70 | 85 |

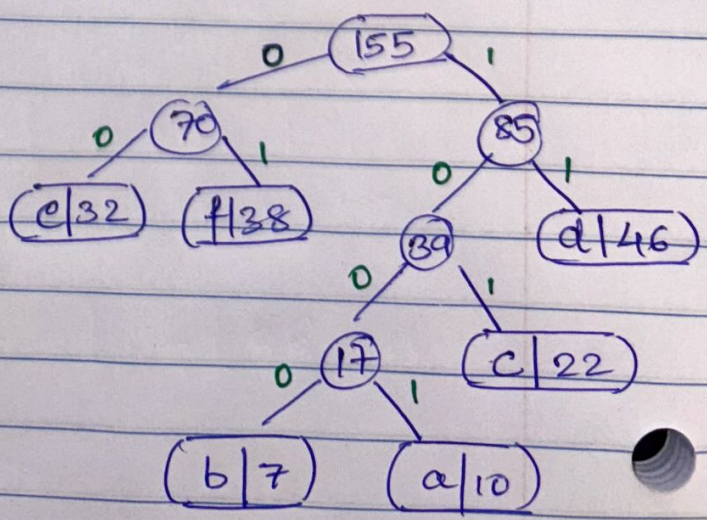
* Current available tree composition



Step 5:

① Extract_min()
{e, f} = 70
{b, a, c, d} = 85

② Combine & we get final tree



∴ Fixed Final code is

Symbol	Frequency	Code
a	10	1001
b	7	1000
c	22	101
d	46	11
e	32	00
f	38	01

Q3)a) Proving by induction -

Greedy choice property means that at every step, a globally optimal solution can be arrived at by choosing a greedy option - selecting the item with highest value to weight ratio v_i/w_i

Basis case -

Only one item, GCP is satisfied (where GCP is Greedy choice Property) as knapsack can either accomodate whole item or can only ~~take~~ it accomodate partially (based on its weight). ~~Fater~~

Induction hypothesis -

Assume greedy choice property, GCP, holds for k-1 items selected by greedy algorithm (in decreasing order of v_i/w_i), we obtain optimal solution.

Induction step -

To show - ~~GCP~~ GCP holds when the k^{th} item is considered.

1) The greedy algorithm selects item in decreasing order of their value to weight ratio v_i/w_i
2) Let x_i represent the fraction of item i that is taken into the knapsack, where $x_i = 1$ if item wholely fits, 0 if item is skipped, $0 < x_i < 1$ if partially taken.
3) After adding first k-1 items to the knapsack,
   - remaining capacity w' is sufficient to accomodate k^{th} item completely ($x_k = 1$) or
   - k^{th} item is added partially ($x_k \in (0,1)$)

4) Since v_k/w_k is largest among remaining items adding the k^{th} ~~par~~ item (or part of it) maximizes the value gained for the weight used.

Inference-

Hence, proved by induction.

b)(i) Cost function-

Let dp[w] represent maximum value that can be obtained with a knapsack capacity of w using unlimited quantities of items.

Recurrence relation-

For each weight w (from $0 \rightarrow W$):

$dp[w] = max(dp[w], dp[w-w_i] + v_i)$

Base case - $dp[0] = 0$ (no capacity, no value)

Pseudocode-

```
def knapsack_with_repetition(n, W, values, weights):
    #initialize dp array
    dp = [0] * (W+1)
    # Fill dp array
    for w in range(W+1):
        for i in range(n):
            if weights[i] <= w:
                dp[w] = max(dp[w], dp[w - weights[i]]
                            + values[i])
    return dp[W]
```

Time complexity -

Outer loop over w runs $O(W)$ inner loop over items runs $O(n)$. Total $O(nW)$

(2) Cost function -

Let $dp[i][w]$ be the cost function

$$dp[i][w] = \begin{cases} \text{True} & \text{if it is possible to achieve weight } w \text{ using first } i \text{ items;} \\ \text{False} & \text{otherwise} \end{cases}$$

Recurrence relation

$$dp[i][w] = dp[i-1][w] \vee (w \geq w_i \wedge dp[i-1][w-w_i])$$

Base case - $dp = true$, $dp = false$ for $w > 0$

Psaudo code:

```
def knapsack( weights, W, n):
    dp= [[False] * (W+1) for _ in range (n+1)]
    dp[0] [0] = True
    # For each item
    for i in range (1, n+1):
        dp[i][0] = True  # empty subset always possible
        # for each target weight
        for w in range (1, W+1):
            # Do not take item i
```

```
dp[i][w] = dp[i-1][w]
#Take item i if possible
if w >= weights[i-1]:
    dp[i][w] |= dp[i-1][w-weights[i-1]]
#return result & reconstruct soln if true
if dp[n][W]:
    solution = []
    w = W
    for i in range(n, 0, -1):
        ~~if w >= weights~~
        if w >= weights[i-1] and dp[i-1][w
                - weights[i-1]]:
            solution.append(i-1)
            w -= weights[i-1]
    return True, solution
return False, []
```

Time complexity - O(nW)
2 nested loops: n iteration x W iterations
each operation takes O(1)
solution reconstruction takes O(n) additional time.

DAA assignment 7

4. In this problem, we are given a hallway (represented as a line) with a set of positions where paintings are placed. The task is to place guards along the hallway such that each guard can cover all paintings within a distance d on both sides. Our goal is to determine the minimum number of guards needed to cover all the paintings.

Solution:

To solve this problem, we can use a greedy approach.

1. Sort the positions of the paintings in increasing order. This ensures that we are considering the paintings from left to right, making it easier to place guards efficiently.

2. Start at the leftmost uncovered painting & place the first guard as far as possible, but still within the range of the first uncovered painting. This ensures that the guard will cover the maximum number of paintings.

3. Once we place a guard, cover all the paintings within the guard's range, & then move on to the next uncovered painting.

4. Repeat the process until all paintings are covered.

Algorithm Steps:

1. Sort the positions of the paintings $x_0, x_1, x_2 \ldots, x_{n-1}$ in increasing order.

2. Start from the first unmarked painting
3. Place a guard at the farthest position possible within the reach of the current unmarked painting.
4. Continue to place guards for all remaining unmarked paintings.

Algorithm:

```
def min_guards (x, d):
    x . sort ()
    guards = 0
    i = 0
    n = len (x)

    while i<n:

        guards += 1

        s = x[i]
        while i < n and x[i] <= s + d:
            i += 1
        p = x[i-1]
        while i < n and x[i] <= p + d:
            i += 1

    return guards
```

Working:

1. Greedy choice property: By always placing a guard at the farthest possible position within reach, we maximize the number of paintings covered by each guard. This helps reduce the total number of guards needed

5.

2. Optimal Substructure: After placing a guard at a certain position, the problem of covering the remaining paintings is the same as the original problem, but with fewer paintings to cover. Thus, we can solve the subproblem optimally & build up to the overall solution.

5. In this problem we are given a connected, undirected, weighted graph and we need to prove that the minimum spanning tree (MST) is also a bottleneck spanning tree. A bottleneck spanning tree is one where the largest edge weight in the tree is minimized compared to all other spanning trees.

Let us prove this by contradiction:

1. Assume the opposite: Suppose the MST is not a bottleneck spanning tree. This means there exists another spanning tree T' where the largest edge weight in T' is strictly smaller than the largest edge weight in the MST T.

2. Consider a cut: Now lets consider a cut in the graph that seperates the MST and T'. According to the cut property of MST's, the MST contains the lightest edge crossing the cut. If T' has a ~~few~~ larger edge crossing the cut, then this suggests a contradiction.

3. Swapping argument: If the edge in T' has an larger weight than the edge in the MST, we can swap them. This would form a new spanning tree that has a

smaller total weight, which contradicts the assumption that T was the MST.

4. Conclusion: Since our assumption leads to a contradiction, it means the MST must also be the bottleneck spanning tree. Therefore, the MST is indeed a bottleneck ~~spanning~~ spanning tree.

working:
contradiction: The assumption that the MST is not a bottleneck spanning tree leads to a contradiction because swapping edges would result in a spanning tree with a smaller total weight, which is impossible for an MST.

cut property: The cut property of MST's is crucial here as it guarantees that the MST contains ~~the lightest~~ lightest edge across any cut. This helps prove that the MST is optimal even when considering ~~the~~ the largest edge weight.

Q6 Given: (DAG) $G = (V, E)$, $w: E \to R - \{0\}$

vertex $s \in V$, int $K \in \{1, 2, \ldots |V|-1\}$

To find: $\forall v \in V$, find length of longest path from s to v that uses atleast K edges.

Let $d(i, v)$ = max. length of any path from s to v using i edges

Let $\ell = |V|-1 \Rightarrow$ max. possible no. of edges in any simple path in DAG.

① ∵ G is DAG, we first find topological ordering of vertices $(v_1, v_2 \ldots v_{|V|})$; $v_1 = s$

② DP base case : $\to d(0, s) = 0$ ∵ path with 0 edges from s has 0 length.

$\to d(0, v) = -\infty$ for all $v \neq s$

(∵ no 0-edge path from s to v exists)

Recursive rel^n:

for $i \geq 1$, consider all predecessors u of v s.t $(u, v) \in E$.

Best i-edge path to v comes from best $(i-1)$ edge path to some u + weight of edge (u, v)

$\Rightarrow d(i, v) = \max_{(u,v) \in E} \{ d(i-1, u) + w(u, v) \}$

If no predecessor u gives finite value, $d(i, v) = -\infty$ (no incoming edges)

$\Rightarrow$ For each $i = 1, 2, \ldots \ell$

for each edge $(u, v) \in E$: $d(i, v) = \max \{ d(i, v), d(i-1, u) + w(u, v) \}$

③ Length of longest path $= \max_{i \geq K} d(i, v)$

If this value is non finite, return "No such path".

Time complexity:

① Topological sort : $O(V+E)$

② DP computation:

$O(V)$ iterations for i ($1 \to |V|-1$)

$\forall i$, iterating all edges $(u,v) \in E$

↳ $O(V*E)$

③ Result:

$\forall v \in V$, compute $\max_{i \geq k} \{d(i,v)\}$ $\to O(V^2)$

Overall: $O(VE+V^2) = O(V(V+E))$