CS-GY 6033: Homework #7

Due on December 13, 2023

Professor Yi-Jen Chiang

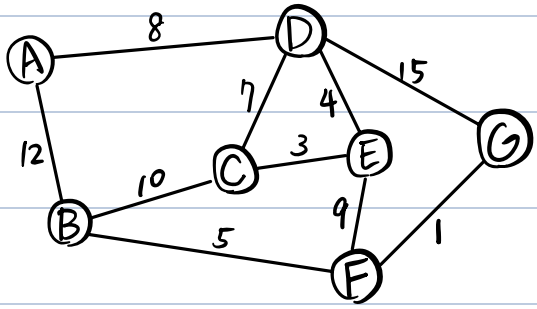
Runze Li
Nxxxxxxxx
rl50xx@nyu.edu

Tzu-Yi Chang
Nxxxxxxxx
tc39xx@nyu.edu

Jiayi Li
Nxxxxxxxx
jl156xx@nyu.edu

December 12, 2023

1.

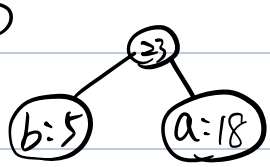


(a) FG(1), CE(3), DE(4), BF(5), AD(8)

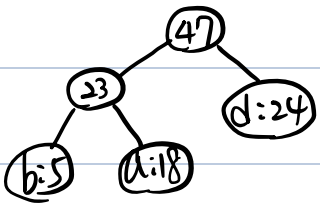
(b) AD(8), DE(4), EC(3), EF(9), FG(1)

2. a:18, b:5, c:30, d:24, e:42

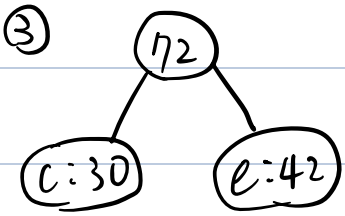
①



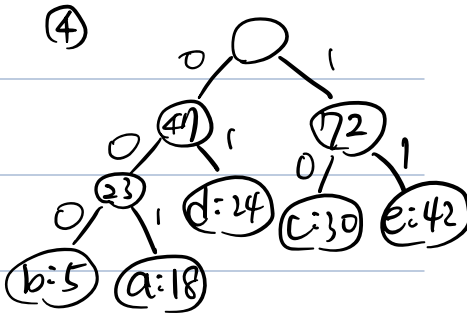
②



③



④



Ans:

a: 001
b: 000
c: 10
d: 01
e: 11

3. (a) We can prove that the fractional knapsack problem has the greedy-choice property by contradiction. Suppose in the provided n items, item a has the maximum value-to-weight ratio $(\frac{v_a}{w_a})$. Assume that there exists an optimal solution where at some point, the algorithm chooses an item b instead to maximize the total value. Now, consider swapping item b of weight x from the knapsack with item a of the same weight. The total value in the knapsack increases as the change in value is $x \cdot \frac{v_a}{w_a} - x \cdot \frac{v_b}{w_b} > 0$ (because $\frac{v_a}{w_a} > \frac{v_b}{w_b}$) and the total weight remains the same. This contradicts the assumption because the so-called optimal solution can be improved by selecting the greedy choice, which is, in this case, item with higher value-to-weight ratio.

(b)(1) Create an empty array dp of size W+1, where dp[i] represents the maximum value that we can get using all items and i capacity of knapsack. Recursively compute dp where

$$dp[i] = \begin{cases} 0, & \text{if } i = 0 \\ dp[i - w_j] + v_j, & \text{if taking item } j \text{ improves the total value} \\ dp[i-1], & \text{else (the maximum value for } i\text{-1 capacity)} \end{cases}$$

The dynamic programming algorithm may be as follow:

```
for i from 0 to W+1 :
    dp[i] = dp[i-1]
    for j from 0 to n :
        if i - w_j ≥ 0 :
            dp[i] = max(dp[i], dp[i - w_j] + v_j)
```

O(nW)

Return dp[W] as our result since dp[W] is the maximum value which can be achieved within the W capacity of knapsack. Compared to the classic 0-1 knapsack problem, instead of computing 2D array, this algorithm uses only 1D array because of the variant that all items are always available at anytime. As the algorithm iterates through the n items each time to compute the results for total weight ranging from 0 to W, it runs in O(nW) worst-case time.

(2) Modify the problem into 0-1 knapsack problem by creating a new set of 2n items, where each item duplicates once. Then we can apply the classic knapsack algorithm to solve the problem. Create a 2D array dp of size $(2n+1) \times (W+1)$ where dp[i][j] represents the maximum value we can get using first i items and having a total weight of j. Denote dp as follow:

$$dp[i][j] = \begin{cases} 0 & \text{, if } i=0 \text{ or } j=0 \\ dp[i-1][j] & \text{, if consider first } i-1 \text{ items only} \\ dp[i-1][j-w_i] + v_i & \text{, if taking the } i\text{th item improves the maximum value for } j \text{ capacity} \end{cases}$$

The algorithm may be as follow:

```
for i from 0 to 2n+1 :
    for j from 0 to W+1 :
        if i = 0 or j = 0 :
            dp[i][j] = 0
        else if j ≥ w_{i-1} :
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-w_{i-1}] + v_{i-1})
        else :
            dp[i][j] = dp[i-1][j]
```

$O(2n \cdot W) = O(nW)$

Return $dp[2n][W]$ as our final answer for it represents the maximum value that can be obtained using all $2n$ items, at most twice for each item in the provided set, within the total capacity W of the knapsack.

The running time is $O(nW)$ worst-case time since we iterate through n multiply a constant of items and compute the results of the total weight W times for each item.

4. ① Sort the provided set X in increasing order such that $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$.

② Initialize a real number last_guard_pos to $-\infty$, where last_guard_pos will record the position of the last guard we set up to protect the first i-1 paintings.

Initialize guard_count = 0.

③ Iterate through each painting i and check if x_i is within the distance that the last guard is giving protection to. If not, then set up a new guard at the position of $x_i + 1$, so that painting i is within its protection area. Update the last_guard_pos to the new guard's position and increment guard_count by 1.

The algorithm may be as the following:

```
for i from 0 to n :
    if x_i > last_guard_pos + 1 :
        last_guard_pos = x_i + 1
        guard_cound ++
```

O(n)

④ Return guard_count as it represents the total number of guards to protect all the paintings.

* Proving the greedy-choice property:

Prove that the optimal solution holds the greedy-choice property by contradiction. Assume that there exists an optimal solution that after the first i-1 paintings have been protected by some guards,

a new guard chooses position p_i to protect the i-th painting, where $p_i \neq x_i + 1$ (our greedy-choice). To protect painting i at x_i, p_i has the property that $x_i - 1 \leq p_i < x_i + 1$. Also, the guard can protect other paintings j where $p_i - 1 \leq x_j \leq p_i + 1$ and $x_i \leq x_j$. If we replace p_i with $x_i + 1$, in fact we can get a better solution because painting i is under protection and it provides a wider range of protection to other paintings ($[x_i, x_i + 2]$ is better than $[p_i - 1, p_i + 1]$ for $p_i < x_i + 1$. Note that the paintings before x_i are all protected by some guards.) Since there is a contradiction, we proved that the optimal solution holds the greedy-choice property.

* Prove the optimal substructure:

Consider the subproblem of guarding paintings from 0 to i. The optimal solution to this subproblem is the same as the optimal solution to the subproblem of guarding paintings from 0 to $i-1$ and consider placing a new guard or not to protect the painting i. Since the optimal solution to the entire problem can be constructed from optimal solutions to its subproblems, we proved the optimal substructure of the problem.

5. ① If the edge with new weight is not in T, then the given minimum spanning tree T is still the MST of G'. This is because if we choose the increased edge to form the new MST, it can only increase the total weight, while the weight of T remains the same, i.e., remains minimum. So if the edge with increased weight is not in T, we do not need to update T.

② If the edge with a new increased weight is in T, then we need to update T since T may no longer serve as the MST of G'.

a) Remove the updated edge $e = (u, v)$ from T.

b) Run DFS on u (or v) to compute A and B where A and B are the connected components of T obtained after deleting edge e. O(V+E)

c) Iterate through all the edges to find the crossing edge e' from A to B with the minimum weight. O(E)

d) Add the edge e' to T. Since A and B are both MST in T, adding an edge with the minimum weight to connect A and B can form an updated MST of G'.

The running time is O(V+E) since DFS runs in O(V+E) worst-case time and it costs only O(E) to find the replacement edge to update T.

6. ① Choose vertex s as the starting point and perform DFS on s to determine the finish time of each vertex, i.e. the topological sorting order.

$O(V+E)$ Note that there may or may not be some vertices that cannot be reached from s, we can exclude them from the below algorithm.

② Let $d(v)$ denote the length of the min-bottleneck path from s to v.

The recursive solution for $d(v)$ can be defined as follow:

$$d(v) = \min_{(u,v)\in E} \{\max(d(u), w(u,v))\}$$

Initialize $d(s) = 0$ and $d(v) = +\infty$ for all other vertices.

③ Iterate through each vertex u in topologically sorted order, and update $d(v)$ where vertex v has an incoming edge from u.

The algorithm may be as follow:

for each vertex u in topological order:

$O(V+E)$ [for each vertex v with an incoming edge from u:

[$d(v) = \min(d(v), \max(d(u), w(u,v)))$

④ Return array d as $d(v)$ represents the length of the min-bottleneck path from s to v for all vertices v.

The total running time is $O(V+E)$ since the DFS (topological sort) and the recursive computation to get $d(v)$ both runs in $O(V+E)$ worst-case time.