Sample Solution for Problem Set 11

Data Structures and Algorithms, Fall 2020

December 29, 2020

Contents

1 Problem 1

(a) $k = 1$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

$k = 2$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 3$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 4$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 5$

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 6$

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

(b) If there was a negative-weight cycle, there would be a negative number occurring on the diagonal upon termination of the Floyd-Warshall algorithm.

2 Problem 2

(a) Here is the values of h and $\hat{w}$ computed by the algorithm to simplify the solution.

v	$h(v)$
1	-5
2	-3
3	0
4	-1
5	-6
6	-8

u	v	$\hat{w}(u,v)$	u	v	$\hat{w}(u,v)$
1	2	NIL	4	1	0
1	3	NIL	4	2	NIL
1	4	NIL	4	3	NIL
1	5	0	4	5	8
1	6	NIL	4	6	NIL
2	1	3	5	1	NIL
2	3	NIL	5	2	4
2	4	0	5	3	NIL
2	5	NIL	5	4	NIL
2	6	NIL	5	6	NIL
3	1	NIL	6	1	NIL
3	2	5	6	2	0
3	4	NIL	6	3	2
3	5	NIL	6	4	NIL
3	6	0	6	5	NIL

So, the d_{ij} values that we get are

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

.

(b) By lemma 25.1, we have that the total weight of any cycles is unchanged as a result of the reweighting procedure. This can be seen in a way similar to how the last claim of lemma 25.1 was proven. Namely, we consider the cycle c as a path that has the same starting and ending vertices, so, by the first half od lemma 25.1, we have that

$$\hat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c) = 0$$

This means that in the reweighted graph, we still have that the same cycle as before had a total weight of zero. Since there are no longer any negative weight edges after we reweight, this is precicely the second property of the reweighting procedure shown in the section. Since we have that the sum of all the edge weights in c is still equal to zero, but each of them individually has a nonnegative weight, it must be the case that each of them individually is equal to 0.

3 Problem 3

3.1 (a)

Update(u,v): For u and its ancestors, they can reach v and its descendants.

Algorithm 1: UPDATE(u, v)

```
1 for $i \in V$ do
2     if $T[i][u]$ or $i == u$ then
3         for $j \in V$ do
4             if $T[v][j]$ or $j == v$ then
5                 $T[i][j] = true$;
6             end
7         end
8     end
9 end
```

3.2 (b)

Assume $G =< V, E >$, $V = \{v_1, v_2, \dots, v_n\}$, $E = \{(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)\}$. Now add $e = (v_n, v_1)$. We need update $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ terms. Therefore, $\Omega(|V|^2)$ time is required for any algorithm.

3.3 (c)

Update2(u,v): Modify the algorithm in (a) by adding a condition.

Correctness:
If $T[i][v]$ is true, no transitive closure need to update.
If $T[i][v]$ is false, the same as the algorithm in (a).

Time complexity:
Use aggregate analysis. If $T[i][v]$ is **false**, the inner loop can make it **true**. Since T has $\Theta(n^2)$ elements, the inner loop (line 3-5) executes $O(n^2)$ times and costs $O(n^3)$. Since there are $x \in O(n^2)$ insertions, the outer loop (line 1-2) executes x times and cost $\Theta(x * n) = O(n^3)$. Therefore, this algorithm is $O(n^3)$.

Algorithm 2: UPDATE2(u, v)

```
1 for $i \in V$ do
2     if $(T[i][u]$ or $i == u)$ and $\neg T[i][v]$ then
3         for $j \in V$ do
4             if $T[v][j]$ or $j == v$ then
5                 $T[i][j] = true$;
6             end
7         end
8     end
9 end
```

4 Problem 4

4.1 (a)

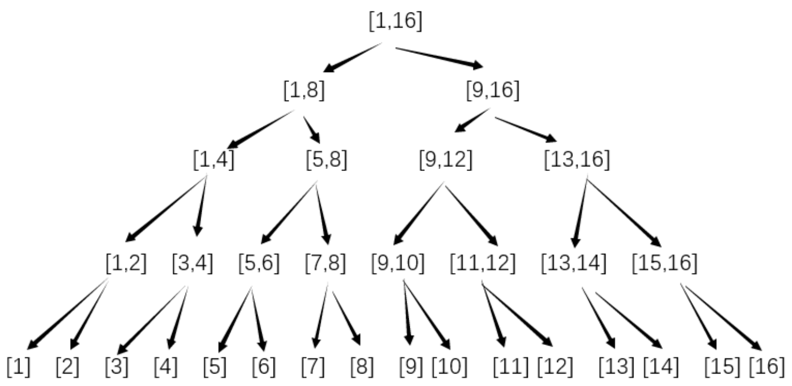
Denote the length of the subproblem $[l, r]$ as $k \equiv (r - l + 1)$. There are $(n + 1 - k)$ subproblems of length k:

$$|V| = \sum_{k=1}^{n} (n + 1 - k) = \frac{n(n+1)}{2}$$

For each problem of length k, there are $2(k - 1)$ subproblems:

$$|E| = \sum_{k=1}^{n} (n + 1 - k) * 2(k - 1) = \frac{n(n+1)(n-1)}{3}$$

4.2 (b)



There are no overlapping subproblems.

4.3 (c)

Counterexample: $n = 4, d = 10, r_{12} = 2, r_{13} = 3, r_{14} = 5, r_{23} = 2, r_{24} = 1, r_{34} = 2, c_1 = 0, c_2 = 0, c_3 = 1000$.

The optimal solution is $1 \to 3 \to 4$, which is equal to $dr_{13}r_{34} - c_1 - c_2 = 60$. However, if you only need to exchange 1 to 3, the optimal solution is $1 \to 2 \to 3$. Therefore, when c_k is arbitrary, there is no optimal substructure for this problem.

5 Problem 5

(a)

Consider $S = \{1, 9, 10\}$ and $target = 18$.

(b)

Let x_i the number of coins with value c_i. The statement follows from the following fact.

- $0 \leq x_i < p$ for all $1 \leq i < k$ if $\mathbf{x} = (x_1, x_2, \ldots, x_k)$ is optimal. (Note that $\mathbf{x}$ is determined with that constraint, and is exactly the same with the one produced in greedy algorithm)

(c)

Let dp_u the number of coins for value u, then $dp_u = 1 + \min_{\substack{1 \leq i \leq k, \\ u \geq c_k}} dp_{u-c_k}$

6 Problem 6

6.1 Dynamic Programming

(a) Suppose the given string is $A[1...n]$.

Overview: We use $dp[i][j]$ to denote the longest subsequence of $A[i...j]$ that is a palindrome. We have the following transition function (omit the base case)

$$dp[i][j] = \begin{cases} dp[i+1][j-1] + 2 & A[i] = A[j] \\ \max(dp[i+1][j], dp[i, j-1]) & A[i] \neq A[j] \end{cases} \tag{1}$$

Top-down implementation: Initialize $dp[i][j]$ to -1 for all $0 \leq i, j \leq 1$. Then call $FindAns(1, n)$. $FindAns(l, r)$ is defined as following:

- **(Base case)**: If $l == r$, return 1. If $l > r$, return 0.
- **(Memorize)**: If $dp[i][j] \geq 0$, return $dp[i][j]$.
- If $A[l] == A[r]$, let $dp[l][r] \leftarrow FindAns(l+1, r-1) + 2$.
- Else, let $dp[l][r] \leftarrow \max(FindAns(i+1, j), FindAns(i, j-1))$.
- Return $dp[l][r]$.

Down-top implementation: Initialize $dp[i][i] = 1$ $dp[i][i-1] = 0$ for all $0 \leq i \leq n+1$, then do the following loop.

```
• For $k$ from 1 to $n-1$
    – For $i$ from 1 to $n-k$
        ∗ Let $j \leftarrow i+k$.
        ∗ If $A[l] == A[r]$, let $dp[l][r] \leftarrow dp[l+1, r-1] + 2$.
        ∗ Else, let $dp[l][r] \leftarrow \max(dp[i+1][j], dp[i, j-1])$.
```

Then the ans is $dp[1][n]$.

Complexity: Remember that the complexity for dynamic programming is (number of states)$\times$(complexity for transition). The number of states is $O(n^2)$ and the complexity for transition is $O(1)$, the total time complexity is $O(n^2)$.

(b) Suppose the given is string is $A[1...n]$.

Overview: We use $dp[i][j]$ to denote the shortest palindrome supersequence of $A[i][j]$, then we have the following transition function:

$$dp[i][j] = \begin{cases} dp[i+1][j-1] + 2 & A[i] = A[j] \\ \max(dp[i+1][j], dp[i, j-1]) + 2 & A[i] \neq A[j] \end{cases} \tag{2}$$

The base case is $dp[i][i] = 1$ and $dp[i][i-1] = 0$ for all $0 \leq i \leq n+1$. The implementation of dynamic programming is described in (a).

6.2 Other algorithms

(a) Let $A[1...n]$ be the given string and $A'[1...n] = A[n...1]$, i.e., $A'[i] = A[n+1-i]$ for $1 \leq i \leq n$. Let the LCS (longest common subsequence) of A, A' be $LCS(A, A')$. We claim that the ans is $LCS(A, A')$.

Remark: The conclusion is **non-trivial**. Suppose the longest palindrome subsequence of $A[i...j]$ is sol. It is easy to prove $LCS(A, A') \geq sol$, since any palindrome subsequence form an equal length LCS between A and A'. But is **HARD** to prove $sol \geq LCS(A, A')$, i.e., any LCS of A, A' can form a palindrome subsequence with good length. it is highly recommended to consider proving it. If you want to get the proof, feel free to talk to me.

(b) Let $A[1...n]$ be the given string. Let sol be the longest palindrome subsequence of $A[i...j]$. The answer is $2n - sol$.

Similarly, to prove the correctness, you need to prove both sides: $ans \geq 2n{-}sol$ and $ans \leq 2n{-}sol$.

7 Problem 7

First choose an arbitrary vertex $r \in T$ as the root of T and use DFS to find the parent of u $u.p$ and the children of u for each $u \in T$. Suppose the parent if the root $r.p = -1$. This problem uses recursion on tree. Let the weight of edge (u, v) be $w(u, v)$. Let the weight of vertex u be $w(u)$.

(a) Algorithm overview: we let $dp[u]$ for each u in T be the path with the max weight from u down to the leaf (not required to get to one of the leaves). The algorithm first complute $dp[u]$ us recursion on tree and find $\max(dp[v_1] + w(u, v_1), 0) + \max(0, dp[v_2] + w(u, v_2))$ for each $u \in T$ and v_1, v_2 are two of u's children with the max value of $dp[v_2] + w(u, v_2)$.

Algorithm: Run $DFS(r)$ to compute dp. $DFS(u)$ for $u \in T$ is defined as following:

- If u has no children (u is a leaf), then let $dp[u] = 0$.
- Else, for each child v of u, call $DFS(v)$. Let v_m be the child of u with the max $dp[v_m]+w(u, v_m)$. Let $dp[u] = \max(0, dp[v_m] + w(u, v_m))$.

For each $u \in T$, compute $dp'[u]$ as following:

- $max_1 = -\infty, max_2 = -\infty$.
- For each child v of u,
  - If $dp[v] + w(u, v) > max_1$, then $max_2 \leftarrow max_1$ and $max_1 \leftarrow dp[v] + w(u, v)$.
  - Else, if $dp[v] + w(u, v) > max_2$, then $max_2 \leftarrow dp[v] + w(u, v)$.
- Let $dp'[u]$ be $\max(max_1, 0) + \max(max_2, 0)$.

The final answer is the following value

$$\max_{u \in T} dp'[u]$$

Complexity: DFS takes $O(n)$ time, compute $dp'[u]$ use $O(n)$ time, find answer use $O(n)$ time. The total complexity is $O(n)$.

(b) Algorithm overview: we let $dp[u]$ for each u in T be the weight of the max weight subtree of T that rooted at u. Then $dp[u] = w(u) + \sum_{v \in C(u), dp[v] > 0} dp[v]$ where $C(u)$ is the set of chilren of u.

Algorithm: Run $DFS(r)$ to compute dp. $DFS(u)$ for $u \in T$ is defined as following:

- Let $dp[u] \leftarrow w(u)$.
- For each child v of u, call $DFS(v)$. If $dp[v] > 0$, let $dp[u] \leftarrow dp[u] + dp[v]$.

The final answer is the following value

$$\max_{u \in T} dp[u]$$

Complexity: Trivially $O(n)$.