Sample Solution for Problem Set 9

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

Note $n \equiv |V|$, $m \equiv |E|$.

- Range from 1 to n:
  - Use counting sort to sort edges, $O(n + m)$.
  - $\Theta(n)$ times Make-Set, $O(n)$.
  - $\Theta(2m)$ times Find-Set, $O(m\alpha(n))$.
  - $O(m)$ times Union, $O(m\alpha(n))$.
  - In total, $O(m\alpha(n))$.
- Range from 1 to W:
  - If $W = O(m \lg m)$, the answer is $O(m\alpha(n) + W)$ (using counting sort).
  - Otherwise, the answer is $O(m \lg m)$ (using merge sort).

2 Problem 2

(a) There is nothing needed to be updated.

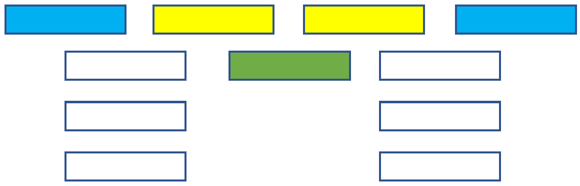
(b) Suppose that the edge e is from u to v, and the graph T' is T with the extra edge e. Since T is a tree, T' has a cycle which contains e. We use tree traversal to find this cycle and remove the edge e' with the maximum weight in the cycle. Then we get a new MST.

(c) There is nothing needed to be updated.

(d) Just run Prim algorithm to calculate the new MST. It costs $O((|V| + |E|) \lg |V|)$ time.
Bonus.
Remove e from T, we will get two tree $T_1(V_1, E_1), T_2(V_2, E_2)$.
Find the minimum weight edge e' across T_1, T_2 which means one vertex is in V_1 and the other is in V_2, If $\hat{w}(e') < \hat{w}(e)$, the new MST is $(V, E_1 \cup E_1 \cup \{e'\})$, otherwise the MST is not changed.

3 Problem 3

- Least duration:
  - Counterexample: $\{[0,3), [2,4), [3,6)\}$.
  - Greedy: $\{[2,4)\}$
  - Truth: $\{[0,3), [3,6)\}$
- Fewest overlaps:



  - Counterexample:
  - Greedy: blue and green
  - Truth: blue and yellow
- Earliest start time:
  - Counterexample: $\{[0,6), [1,2), [3,4)\}$.
  - Greedy: $\{[0,6)\}$
  - Truth: $\{[1,2), [3,4)\}$

4 Problem 4

4.1 Algorithm

Choose the most valuable item in the remaining ones repeatedly until knapsack is full.

4.2 Correctness

Correctness holds once we spot the following fact.

- Given items I with least weight and most value, there exists one optimal solution containing this item.

5 Problem 5

Actually, our Huffman code for all character contains 8 bits. To see this, we should notice that sum of frequencies of K elements will never exceed that of $2K$ elements.

6 Problem 6

Algorthm: For convenience assume endpoints are distinct.

Create a new array $K[1...2n]$ where $K = L+R$, i.e., $K[i] = L[i]$ and $K[n+i] = R[i]$ for $1 \leq i \leq n$. Each element $K[i]$ maintain an attribute $K[i].key$ where $K[i].key = L$ if $i \leq n$ and $K[i].key = R$ if $i > n$. Sort the elements in K increasingly in $O(n \log n)$ time (swap two elements will also swap their attribute). Then do the following loop. cnt is an interger initialied to 0, M is an integer initialized to 0.

- For $i = 1$ to $2n$ do
  - If $K[i].key = L$, $cnt \leftarrow cnt + 1$. Else, $cnt \leftarrow cnt - 1$.
  - $M \leftarrow \max(M, cnt)$.

The answer is M.

Complexity: Each loop cost $O(1)$ time and the total time complexity is $O(n \log n)$.

Correctness: Intuitively, cnt is always the number of intervals that cover the point $K[i]$. M will be the max number of cnt. Suppose the optimal coloring uses opt colors. Obviously, we have $opt \geq M$, since we cannot use less than M colors to color the intervals where M of them interates at the same point.

Now we prove $opt \leq M$, i.e., there is a legal coloring way using M colors. We create M colors $C[1...M]$. Create a color set $colors$ intialized to $\emptyset$. In each loop while $K[i].key = L$, we color the interval with left point $K[i]$ with a new color in $C[1...M] \backslash colors$ and add the new color to $colors$. We maintain the following loop invariance:

loop invariance: After the i-the loop, intervals with endpoints in $K[1...i]$ are colored different colors if they interated with each other, and the intervals that cover point $K[i]$ only use colors in set $colors$.

The proof of the loop invariance can be trivially varified.

7 Problem 7

Algorithm: ans is an integer initialied to 0. Suppose the nodes in tree is represented in BFS order and stored in $B[1...n]$ (Which can be done in $O(n)$ time). For each node in the tree, initialize an attribute vis as 0. We do the following loop to find paths and update ans:

- For $i = n$ to 1 do
  - If $B[i].vis = 1$, skip the loop. Otherwise do the following procedure.
  - Let p be the k-th parent of $B[i]$ ($B[i].p$ is the first parent, i.e., $B[i]$ perfom k-moves to the loop and get the k-th parent of $B[i]$). If p do not exists, end the procedure.
  - $ans \leftarrow ans + 1$, mark the attribute vis of all the nodes in the subtree rooted at p as 1 (The procedure can be achieved by running DFS on p).

We define our $DFS(p)$ in the third step of the loop as follows:

- For each child c of p, if $c.vis = 0$ then $c.vis \leftarrow 1$ and $DFS(c)$.

Intuitively, the algorithm find a path in each loop. The path goes from the deepest possible nodes of the tree.

Complexity: The cost for loop is $O(n)$. We need to analysis the cost for the third step(DFS) of the loop. Since each node will be access at most once (change its vis from 0 to 1), the total complexity for DFS is $O(n)$. The total complexity is $O(n)$ where n is the number of nodes.

Correctness: We claim that the path goes from the deepest nodes in a tree must exists in an optimal solution. Denote the deepest node as c and its k-th parent as p. Suppose there is a optimal solution that do not contain the path from c to p. Say a node is occupied by node a if it is on the path started from a in the optimal solution. If p is not occupied, then any node in the subtree of p can not be occupied (Otherwise p must be occupied since c is the deepest node in the tree), in which case we can add a path from c to p, contradicting the fact that it is a optimal solution. If p is occupied by node a, a must exist in the subtree rooted at p. Due to the same reason (c is the deepest node in the tree), other nodes in the subtree of p must not be occupied. We delete the path started at a, and add the path from c to p, then we get the optimal solution with the same number of path containint the path from c to p.

By marking the vis to 1 for each node in the subtree of p, we delete the subtree from the tree. We claim that the optimal solution on the remaining tree adding the path from c to p is the optimal solution. Otherwise, if there exists a better solution, since we have proved that the path from c to p must exists in the solution, we can delete the subtree of p and also get a better solution in the deleted tree.