

CSE373 Homework 3 Solutions & Hints

In this solution, if the input is $G(V, E)$, we assume that $|V| = n, |E| = m$.

General principle: wrong algorithm but serious attempt: 50%; correct algorithm without a proof of correctness: 70%; slower algorithm with a proof: 60%.

Pay attention to the 'Common mistake' in this solution.

1 Exercise 5-4 (10 pts)

We prove such kind of edge could not appear in a BFS tree: an edge connecting a vertex y and its (indirect) ancestor x ; otherwise, y would have been explored when we expanded x , so in this case y would be x 's direct child.

Notes: we label each vertex of G by their distance to some node R . Then any two vertices connected by an edge cannot have their label gap larger than 1.

2 Exercise 5-7 (10 pts)

(1) Yes. We assume all the nodes are distinct in their values. We can find the root of the tree and deal with its left and right subtree recursively. The algorithm is like this:

```
BuildT(string preorder, string inorder, root){
    root -> value = preorder[0];
    tmp = inorder.Search(preorder[0]);
    BuildT(preorder.Substring(1, tmp), inorder.Substring(0, tmp - 1), root -> left);
    BuildT(preorder.Substring(tmp + 1, tail),
           inorder.Substring(tmp + 1, tail), root -> right);
}
```

Here boundary conditions are ignored.

(2) No. Counterexample:

Tree1: altogether two nodes. The root has a value 1 and its left child has a value 2;

Tree2: altogether two nodes. The root has a value 1 and its right child has a value 2.

3 Exercises 5-12 to 5-14 (3 pts + 4 pts + 3 pts = 10)

5-12:

Adjacency list: use brute-force method.

foreach vertex v in V

 foreach edge (v, u) in v 's adjacency list

 foreach edge (u, w) in u 's adjacency list

 add (v, w) into the result

For each vertex v , we are visiting no more than $2m$ edges since each edge could be visited at most twice. So the running time is $O(mn)$.

Matrix: Similar strategy works. However, we can present our algorithm in a simpler way. Suppose the matrix of G is M , then the matrix of G^2 is M^2 . Here the boolean matrix multiplication is defined based upon these rules: $a + b = a \vee b$, $a * b = a \wedge b$, while a, b are 0 or 1. Then we know $M_{ij}^2 = 1 \Leftrightarrow$ there exists some k , s.t. $M_{ik} = 1$ and $M_{kj} = 1$.

Simple matrix multiplication algorithm is in $O(n^3)$.

Notes: Asymptotically faster algorithms for matrix multiplication do exist. But they must be slightly modified in this boolean matrix case.

5-13:

(a) and (b) are special cases of (c). So we solve (c) directly. Here I will give an $O(n)$ algorithm, so it is also the asymptotically best solution for (a) and (b).

(c) Use the idea of dynamic programming. Let $Opt_{in}(x)$ and $Opt_{out}(x)$ be the value of the minimum weight vertex cover for the subtree with its root x , on the conditions that x is or isn't in the cover, respectively.

We define $children(x)$ as the set of all the children of vertex x in the tree, w_x as the value of vertex x . Then we have:

$Opt_{in}(x) = w_x, Opt_{out}(x) = 0$, if x is a leaf.

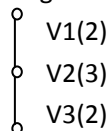
$Opt_{in}(x) = w_x + \sum_{y \in children(x)} \min\{Opt_{in}(y), Opt_{out}(y)\}$,

$Opt_{out}(x) = \sum_{y \in children(x)} Opt_{in}(y)$, if x is not a leaf. (If x is in the vertex cover, then it doesn't matter whether x 's children are in the vertex cover; if x is not, then all of x 's children must be in the cover.)

And our answer is $\min\{Opt_{in}(r), Opt_{out}(r)\}$, while r is the root of the original tree.

It's not difficult to see the running time is $O(n)$.

Common mistake: using some sort of greedy algorithm. This is a counter-example for most of your algorithms:



Most of your algorithms will choose $v1$ and $v3$, but the answer is $v2$.

Notes: maybe there exists greedy algorithms for (a) and (b), but probably greedy algorithms don't work for the general case (c). Hint for (a): Consider from the leaf. Hint for (b): sum of weights of any vertex set \geq # of edges covered by the set.

5-14:

Yes. There are three kinds of edges: the edges connecting two non-leaves, the edges connecting a non-leaf vertex and a leaf, and the edges connecting two leaves. The first two kinds are covered because all the non-leaf vertices are retained. The third kind doesn't exist. Suppose x and y are two leaves in the DFS tree, and x is explored earlier than y . If there is an edge connecting x and y , x would be y 's parent in the DFS tree, so x is not a leaf.

4 Exercise 5-19 (10 pts)

Solution1: When we do a BFS, we can label the depth of each vertex, or the distance from the root to that vertex, without increasing the complexity.

We start at arbitrary vertex s , do a BFS, and find the deepest vertex u from s , e.g. the last vertex discovered. (If there is more than one deepest vertex, pick arbitrary one). Then we start at u , do a BFS, and find the deepest vertex v from u . Then the path from u to v is the longest path, and its length is the diameter of T .

Now we prove the correctness of the algorithm. For any vertex x , we define $h(x) = \delta(s, x)$. For any vertex x and y , we define $p(x, y)$ as the unique path connecting x and y .

Suppose $p(a, b)$ is the longest path in the tree. We find the unique t in $p(a, b)$ with the smallest $h(t)$. If t is not u 's ancestor, we know $p(u, t)$ and $p(t, b)$ have only one common vertex t . If t is u 's ancestor, we know that it's impossible that $p(t, u)$ has common vertex other than t with both $p(t, a)$ and $p(t, b)$. In this case, without losing generality, we assume $p(t, u)$ and $p(t, b)$ have only one common vertex t .

In both cases, we have $\delta(u, t) + \delta(t, b) = \delta(u, b)$, so

$$\begin{aligned} \delta(a, b) &= \delta(a, t) + \delta(t, b) = h(a) - h(t) + \delta(t, b) \leq h(u) - h(t) + \delta(t, b) \leq \delta(u, t) + \delta(t, b) \\ &= \delta(u, b) \leq \delta(u, v) \end{aligned}$$

So $p(u, v)$ is also the longest path.

The running time is $O(m + n) = O(n)$.

Solution2: Dynamic programming. Let $h(v)$ be the height of the subtree whose root is v . Let $dia(v)$ be the diameter of the subtree whose root is v . Let $children(v)$ be the set of v 's children. We have:

$h(v) = 0, dia(v) = 0$, if v is a leaf.

$$h(v) = 1 + \max_{w \in children(v)} h(w)$$

$$kind1(v) = \max_{w \in children(v)} dia(w)$$

$$kind2(v) = \begin{cases} h(v), & \text{if } |children(v)| = 1 \\ \max_{\substack{x, y \in children(v) \\ x \neq y}} (h(x) + h(y) + 2), & \text{if } |children(v)| \geq 2 \end{cases}$$

$dia(v) = \max\{kind1(v), kind2(v)\}$, if v is not a leaf.

Here $kind1$ considers the paths that don't pass v ; $kind2$ considers the paths that pass v .

The running time is $O(n)$.

Common mistake: only consider $kind2$.

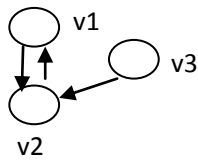
5 Exercise 5-25 (10 pts)

- Do a DFS from v .
- Compute the strongly connected components of G . Then construct a new graph G' , and use one vertex to present one component. We know G' is a DAG. If there's more than one vertex in G' whose in-degree is 0, G doesn't have a mother vertex; otherwise we have a vertex v in G' that can visit all other components, so any vertex in the component which v presents would be a mother vertex.

Common mistake: Arbitrarily pick up a vertex v . Do a DFS from v . If v can visit all the vertices, then v is the mother vertex. Otherwise, pick up another vertex u that has not been visited yet. We do a DFS from u , and find all the vertices that can be reached from u (of course we do not

expand vertices that have already been marked visited). If all the vertices can be reached, we return u ; otherwise we run the third round. We repeat this process, until it returns a mother vertex or all the vertices have been visited; in the second case it will return false.

This algorithm doesn't work since without using other techniques you cannot judge whether all the vertices can be reached from u efficiently. For instance:



If we first run DFS from $v1$, we will mark $v1$ and $v2$ as visited. Then we run a DFS from $v3$. How can we judge whether $v3$ can visit $v1$? (Actually $v3$ is the mother vertex)

6 Programming (10 pts)

C1: 3 components

C2: 10 components

C3: 25 components

C4: 2 components

Concrete results and sample program omitted.

7 Exercises 6-4 and 6-5 (5 pts + 5 pts = 10)

6-4:

Yes. Examples are easy to find.

Notes: If all edge weights are distinct, the MST is unique. If not, there may be more than one solution. But here's a useful conclusion: all the MSTs can be generated by Kruskal's algorithm if we can decide which edge to choose whenever there is more than one choice.

6-5:

Yes. Both P-Algo and K-Algo perform the same if the topology of the graph and the more/less relationship of edge weights remains the same. Suppose the algorithm doesn't work correctly. Assume the tree that the algorithm outputs is T_G , and the MST is T'_G , we know $w(T_G) > w(T'_G)$. We add some value x to the weight of each edge of G to make all the weights positive. We call this new graph G' . Suppose $T_{G'}$ is the spanning tree of G' which has the same topology as T_G , and $T'_{G'}$ has a similar definition. We know if we run the algorithm on G' , we will get $T_{G'}$, because the topology and more/less relationship are the same. Since all the edge weights of G' are positive, the algorithm would work. So $T_{G'}$ is the MST of G' . Thus $w(T_{G'}) \leq w(T'_{G'})$. But we know $w(T_{G'}) - w(T_G) = w(T'_{G'}) - w(T'_G) = (n - 1)x$. So this gives us $w(T_G) \leq w(T'_G)$. We get a contradiction.

Common mistake: just say 'the algorithm will perform the same' or 'it will choose the most negative edges first'. Consider why Dijkstra's algorithm doesn't work in negative cases? So you must give a formal proof, or specify it will perform the same with what. The key points here are both algorithms rely on comparisons only, and that the number of edges of any spanning tree is

fixed.

8 Exercise 6-9 (10 pts)

- (a) If there are negative edges, all of them should be included in the MWCS, even if they can form a cycle.
- (b) **Solution 1:** First the MWCS is the empty set. We choose all non-positive edges and add them to the MWCS. Then compute the connecting components of the graph formed by the edges we chose. Then build a union-find structure, and the vertices of the same component are also in the same component of the structure. Finally Run K-algo and add all the edges chosen by the algorithm to the MWCS. The running time is $O(m \log n)$.

Proof: Let T be the graph that this algorithm outputs. It's clear that T is a connecting set. Suppose M is any MWCS of the graph. We want to prove $w(T) \leq w(M)$. Split both T and M to positive and negative values, i.e., $w(T) = w(T^+) + w(T^-)$, $w(M) = w(M^+) + w(M^-)$. M has to connect the components which are computed after we choose all the negative edges. Because now all the remaining edges we can choose from are positive (negative edges cannot connect two components), by the correctness of Kruskal's algorithm, we know $w(T^+) \leq w(M^+)$. On the other hand, T contains all negative edges, so $w(T^-) \leq w(M^-)$. Thus we proved $w(T) \leq w(M)$.

Solution 2: First compute the MST using K-Algo. Then add all negative edges. The running time is $O(m \log n)$.

Proof: Continued from the proof of solution 1. Suppose this algorithm produces S . Then S is a connecting set. We want to prove $w(T) = w(S)$. Similarly, split S into 2 parts, i.e., $w(S) = w(S^+) + w(S^-)$. Consider the state when we have just processed all non-positive edges using Kruskal's algorithm. We claim the union-find structure is the same as the one when we pick all the non-positive edges in solution 1. This is obvious since all the negative edges that Kruskal's algorithm bypassed could only connect two vertices that are in the same components. Since both solutions use Kruskal's algorithm to pick positive edges, we know $w(S^+) = w(T^+)$. Moreover, $w(S^-) = w(T^-)$ since both solutions pick all the negative edges. So we have proved two solutions produce the same value.

Notes: solution1 is harder to implement, but easier to prove. Here Kruskal's algorithm is useful in the proof. In some problems we'll apply the conclusion in 6-4's notes first.

9 Exercise 6-15 (10 pts)

Not necessarily. Counter-example: $V = \{v1, v2, v3, v4\}$; $E = \{(v1, v2), (v2, v3), (v3, v4), (v4, v1)\}$; $W(v1, v2) = W(v2, v3) = W(v3, v4) = 1$, $W(v4, v1) = 4$; the root is $v1$.

10 Exercise 6-17 (10 pts)

The answers to both (a) and (b) are no. Counter-example:

$V = \{v1, v2, v3, v4\}$; $E = \{(v1, v2), (v2, v3), (v3, v4), (v4, v1)\}$;

$W(v1, v2) = W(v2, v3) = W(v3, v4) = 1$, $W(v4, v1) = 2$; the two vertices are $v1$ and $v4$.