CSE 373 Analysis of Algorithms Date: Nov. 1, 2016

Solution to HW3

Total Points: 100

Steven Skiena

Problem 1

5-4 *[5]*

Let T be the BFS-tree of the graph G. For any e in G and $e \notin T$, we have to show that e is a cross edge. Prove it by contradiction: suppose $e = (x, y)$ is not a cross edge. Also, say x is an ancestor of y. This means that x was discovered before y in the BFS traversal. The way BFS works, all of x's children would have been discovered at the next level. This means that $e = (x, y) \in T$, which leads to a contradiction.

Remark: This is only a sample answer. We give credits for other reasonable answers.

Problem 2

5-7 *[10]*

Given pre-order and in-order traversals of a binary tree, is it possible to reconstruct the tree? *[5]*

Yes. We can do it as follows:

1. The first element of the pre-order traversal is the root.
2. Find the root in the in-order traversal, the elements to its left are in the left subtree and to its right are in the right subtree.
3. Recursively follows the above steps to construct the entire tree.

Given the pre-order and post-order traversals of a binary tree, is it possible to reconstruct the tree? *[5]*

No. See the following example:

Figure 1: Example.

Both tree have the same pre-order and post-order traversal.

Remark: For each question, 1 point for Yes/No. 4 points for algorithm, reason or counterexample.

Problem 3

5-12 *[5]*

For adjacency matrix:
for $i = 1$ to V do
  for $j = 1$ to V do
    $G^2(i, j) = 0$
    for $k = 1$ to V do
      if $G(i, k) == 1$ and $G(k, j) == 1$ then
        $G^2(i, j) = 1$
Thus, the total time complexity is $O(V^3)$. *[2.5]*

For adjacency list:
for $u \in V$ do
  for $v \in Adj(u)$ do
    for $w \in Adj(v)$ do
      Add edge (u, w) in G^2
The total time complexity is $O(VE)$. *[2.5]*

Remark: 2.5 points for each data structure.

5-13 *[15]*

(a). In a vertex cover we need to have at least one vertex for each edge. Every tree has at least two leaves, meaning that there is always an vertex v which is adjacent to a leaf u. Apparently it's always better to choose v than u, since it is the only one which can also cover other edges! After trimming othe covered edges, we have a smaller tree. We can repeat the process until the tree has 0 or 1 edges. When the tree has only one isolated edge, pick either vertex. All leaves can be identied and trimmed in $O(n)$ time during a DFS, so it takes $O(n)$ in total. *[5]*

(b). Actually this is a dynamic programming problem on tree. Let's define $score[u][include]$ as the minimum weight vertex cover for the subtree rooted at u. $include \in \{0, 1\}$ and indicates whether we include u itself in the vertex cover or not. Then, run DFS in the tree, and during the DFS traversal:

(1) If current vertex u is a leaf node, then simply set $score[u][0] = 0$, $score[u][1] = w_u$, where w_u is the weight of vertex.

(2) Else, update the score of current node u using its children's scores: $score[u][0] = \sum_{c \in child(u)} score[c][1]$, because when you don't

include current root, you have to include each of its child. $score[u][1] = w_u + \sum_{c \in child(u)} \min(score[c][0], score[c][1])$ because now you can choose to include each child or not. Finally, get the vertex cover by running DFS again from root node, and check which vertices we select. The total time complexity is the same as DFS, which is $O(V + E)$.

[5]

(c). The algorithm is the same as (b). *[5]*

Remark: 5 points for each sub-question.

5-14 *[5]*

If the tree has more than one vertex, then yes. The remaining vertices are still the vertex cover because for every edge $e \in E$ incident on the leaves, their other end-point is still in the remaining tree.

Problem 4

5-19 *[10]*

Do a BFS of the tree by starting at any node s and marking the distance of the nodes from the root s. Let's find a leaf u with maximum $d(s, u)$, and we do a BFS with root $= u$. Pick the leaf v such that $d(u, v)$ is the maximum among all leaves. The diameter of the tree is then $d(u, v)$. Apparently, the time complexity is the same as BFS, which is $O(V + E)$. *[4]*

Now let's prove the correctness of the algorithm.

For simplicity, let us assume that the diameter of the graph is unique. That is, there exists exactly one pair of vertices (u, v) which have path length $d(u, v)$ between them, which is the highest path length among any pair of vertices in the graph. It is easy to see that both u and v are leaf nodes.

Take any node w and find the vertex which is furthest from it. We will show that the vertex found will be either u or v. Suppose that the vertex found is different, say z. We will consider 2 cases.

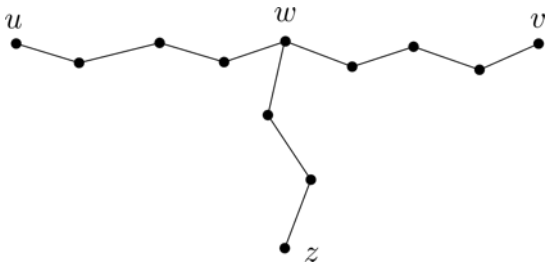


Figure 2: w lies on the path from u to v.

First suppose that w lies on the path from u to v. Without loss of generality, let the $w-u$ path have no edges overlapping with the $w-z$ path. We have $d(w,z) \geq d(w,v)$. But we know that $d(w,z) \geq d(w,u)$. Thus $d(u,z) = d(u,w) + d(w,z) \geq d(u,w) + d(w,v) = d(u,v)$. This contradicts the assumption that $d(u,v)$ is the unique diameter of the tree. *[2]*

So let w not lie on the path from u to v. Now, either the $w-z$ path either overlaps with the $u-v$ path or is disjoint.

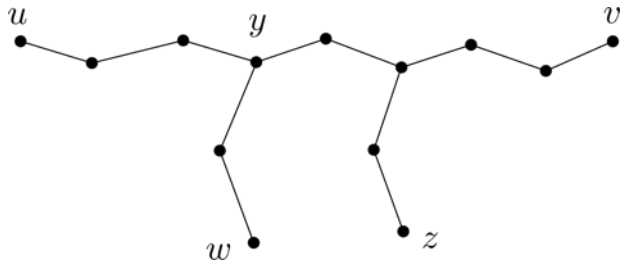


Figure 3: w doesn't lie on the path from u to v and the $w-z$ path overlaps with the $u-v$ path.

If there is overlap, consider the vertex y which is the vertex closest to w among the vertices are part of the overlap. Without loss of generality, let the $y-u$ path have no edges overlapping with the $y-z$ path. Now, $d(y,z) \geq d(y,v)$. Hence $d(u,z) = d(u,y) + d(y,z) \geq d(u,y) + d(y,v) = d(u,v)$. This once again contradicts the assumption of (u,v) being the unique diameter of the tree. *[2]*

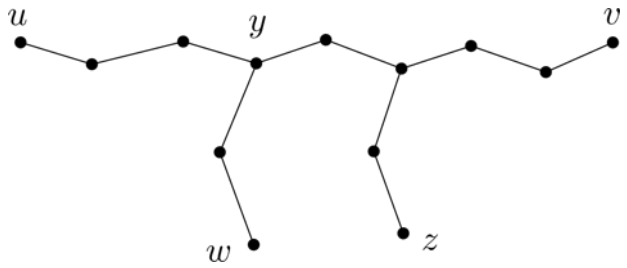


Figure 4: w doesn't lie on the path from u to v and the $w-z$ path and the $u-v$ path don't overlap.

If the paths do not overlap, there are vertices x and y on the $u-v$ and $w-z$ paths respectively which are closest to each other. $d(y,z) \geq d(y,v)$. Hence $d(u,z) = d(u,y) + d(y,z) \geq d(u,y) + d(y,v) \geq d(u,x) + d(x,v) = d(u,v)$. Hence the assumption that $d(u,v)$ is the diameter is contradicted. *[2]*

In each case, we have seen that there is a contradiction if z is not one of u or v. Hence it follows that z, the furthest vertex from w, is either u or v. Now it is easy to see that the furthest vertex from u (or v) is v (or u) and hence the distance we calculate in the second dfs is actually the diameter of the tree.

Remark: 3 points for giving the algorithm. 1 point for giving the running time. 6 points are for proving the correctness of the algorithm (2 points for each case). Other correct answers are also accepted.

Problem 5

5-25 *[5]*

Do a DFS for each vertex, if one of the DFS tree contains all the vertices of the graph, return true. Running time $O(n^2)$.

Problem 6

[Connected Components] (20pts)

The pseudocode is as follows:

```
dfs (node u)
  for each node v connected to u, do
    visited[v] = true
    dfs(v)

for each node u, do
 if u is not visited, then
   visited [u] = true
   component += 1
   dfs (u)
```

The code is as follows (written in C++):

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;

const int N = 105, L = 50;
bool tag[N];
vector<int> adj_list[N];
vector<int> component[N];
int component_count;

void init(int node_count)
{
    for (int i = 1; i <= node_count; ++i)
```

```
    {
        adj_list[i].clear();
        component[i].clear();
    }
    memset( tag, 0, sizeof(tag) );
    component_count = 0;
}

void dfs(int u)
{
    component[component_count-1].push_back(u);
    int adj_count = adj_list[u].size();
    for (int i = 0; i < adj_count; ++i)
    {
        int v = adj_list[u][i];
        if (!tag[v])
        {
            tag[v] = true;
            dfs(v);
        }
    }
}

void print_ans()
{
    for (int i = 0; i < component_count; ++i)
    {
        int component_size = component[i].size();
        printf("Connected Component # %d:\n", i + 1);
        for (int j = 0; j < component_size; ++j)
        {
            if (j)
                printf(" ");
            printf("\%d", component[i][j]);
        }
        printf("\n");
    }
}

int main()
{
    char fname[L];
    int node_count, edge_count, u, v;
    printf("Please enter the graph file's name:\n");
    scanf("%s", fname);
    freopen(fname, "r", stdin);
```

```
    scanf("%d%d", &edge_count, &node_count);
    init(node_count);
    for (int i = 0; i < edge_count; ++i)
    {
        scanf("%d%d", &u, &v);
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }
    for (int i = 1; i <= node_count; ++i)
    {
        if (!tag[i])
        {
            ++component_count;
            tag[i] = true;
            dfs(i);
        }
    }
    print_ans();
}
```

The result for test file 1:

```
Connected Component #1:
1 2 4 8 5 10 9 6
Connected Component #2:
3
Connected Component #3:
7
```

The result for test file 2:

```
Connected Component #1:
1 15 5 7 27 26 14 13 12 21 25 16 6 19 9 17 2 10 18 28 29
30 23 24 20 3
Connected Component #2:
4 8
Connected Component #3:
11
Connected Component #4:
22
Connected Component #5:
31 42 46 32 37 41 47 39 40 49
Connected Component #6:
33 48 45 44
Connected Component #7:
34 43 36
Connected Component #8:
35
```

```
Connected Component #9:
38
Connected Component #10:
50
```

The result for test file 3:

```
Connected Component #1:
1 11 37 62 23 46 63 4 55 85 52 75 12 87 67 70 97 30 43
77 34 96 47 33 79
Connected Component #2:
2 10
Connected Component #3:
3 15 44 25 26 54 22 82 27 59 29 39 24 51 35 89 9 72 68
74 99 76 93 71 90 80 84 64 92 6
Connected Component #4:
5 21
Connected Component #5:
7
Connected Component #6:
8 73 40 61 16 83 100 19 45 48 57 81 50 32
Connected Component #7:
13
Connected Component #8:
14
Connected Component #9:
17
Connected Component #10:
18
Connected Component #11:
20
Connected Component #12:
28 60 42 65 53 66 88
Connected Component #13:
31
Connected Component #14:
36
Connected Component #15:
38
Connected Component #16:
41
Connected Component #17:
49
Connected Component #18:
56
Connected Component #19:
58
```

```
Connected Component #20:
69 98
Connected Component #21:
78
Connected Component #22:
86
Connected Component #23:
91
Connected Component #24:
94
Connected Component #25:
95
```

The result for test file 4:

```
Connected Component #1:
1 4 7 8 9 12 13 14 16 20 22 23 25 26 30 32 34 38 40 41 42
44 45 46 48 49 50 52 53 54 55 56 59 62 65 66 67 70 73 75
77 81 82 87 88 89 90 91 93 100
Connected Component #2:
2 3 5 6 10 11 15 17 18 19 21 24 27 28 29 31 33 35 36 37 39
43 47 51 57 58 60 61 63 64 68 69 71 72 74 76 78 79 80 83
84 85 86 92 94 95 96 97 98 99
```

Remark: 3.5 points for providing the source code. 1.5 points in total for test file 1 (0.5 points for each case). 4 points in total for test file 2 (0.4 points for each case). 10 points in total for test file 3 (0.4 points for each case). 1 point in total for test file 4 (0.5 points for each case).

Problem 7

6-4 *[2.5]*

If the graph has distinct edge weights then it has a unique spanning tree and both Prim's and Kruskal's algorithm will output the same tree. But if there exists two edges with the same weight, and the tie is broken arbitrarily, then both the algorithms may output different trees.

6-5 *[2.5]*

Both Prim's and Kruskal's algorithm work with negative edge weights. It is because the correctness of the algorithm does not depend on the weights being positive. Kruskal's algorithm sorts the edges according to weights (which doesn't change when negative weights are present) and chooses the best edge from it each time. In Prim's algorithm the tree evolves by adding the least weight edge that connects a tree vertex to a non-tree vertex. The edge selection is not effected by negative weights.

Problem 8

6-9 *[10]*

Note that you have to include all vertices of G and T must be connected (but it does not necessarily have to be a spanning tree).

This problem is different from minimum spanning tree - consider the graph on $\{a, b, c\}$ which is a triangle and each edge has weight -1. A minimum spanning tree would include any two edges like $\{ab, ac\}$ with total weight -2, while minimum connected subset would include all three edges with total weight -3. *[5]*

An efficient algorithm to compute minimum weight connected subset T is:

1. Find the minimum spanning tree T of G using Kruskal's algorithm.
2. For each edge $e \in G$ with $w_e < 0$, add e to T.

This takes $O(E \log V + E) = O(E \log V)$ in total. *[5]*

Remark: 5 points for describing why this problem is different from minimum spanning tree problem. Note that it is only a sample answer. 5 points for the efficient algorithm.

Problem 9

6-15 *[5]*

No, it is not necessary. Consider the following example. Let G be the graph below and T be the shortest path spanning tree rooted at the leftmost vertex as shown. If we add $k = 2$ to all the edge weights and get graph G'. Then the shortest path spanning tree will change to T'. See Figure 5.

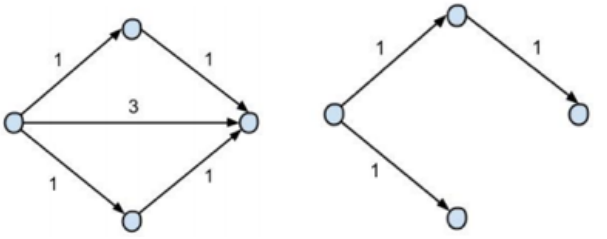
Problem 10

6-17 *[5]*

Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path? - No, see the example given in Figure 6. *[2.5]*

Suppose that the minimum spanning tree of the graph is unique. Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path? - No, see the example given in Figure 7. *[2.5]*

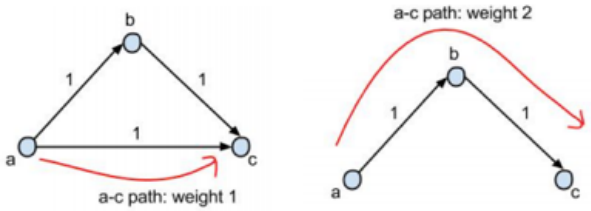
Remark: For each sub-question, 1 point for Yes/No. 1.5 points for reasons.



(a) Original graph G (b) T

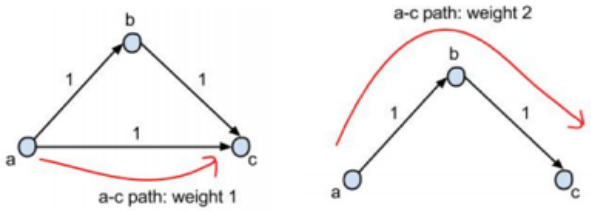
(c) New graph G' (d) T'

Figure 5: Illustration for Problem 9.



(a) Shortest path in G (b) Shortest path in T

Figure 6: Illustration for Problem 6-17 (a).



(a) Shortest path in G' (b) Shortest path in T'

Figure 7: Illustration for Problem 6-17 (b).