CS5800 Algorithms *Out: October 08, 2019*

Solutions for Problem Set 4

Name: Instructor's solutions *Due: October 15, 2019*

Problem 1 (The MST, the MST, and the Shortest Path) 5+5+5+5+5

Given a connected undirected graph $G = (V, E)$ such that $|E| > n$, and $w : E \to (1, \infty)$ be a *one-to-one* weight function on the edges of G. (i.e., $w(e) > 1$ for every $e \in E$; and $w(e) \neq w(e')$, for $e \neq e' \in E$). In this framework, the weights of the edges are determined by querying w on the respective edges. By defining different weight functions, we can ask questions about G with respect to these different weight functions. Let $w' = w(e) - 1$ and let $w''(e) = w(e)/2$, for every $e \in E$.

1. Prove that there exists exactly one MST of $G(V, E)$ with respect to w; exactly one MST of $G(V, E)$ with respect to w'; and exactly one MST of $G(V, E)$ with respect to w''.

   Solution:

   Proof by contradiction:

   Let there be another MST of $G(V, E)$, called $T_{w2}(V, E) \neq T_w(V, E)$. Since both $MSTs$ are distinct, there must be an edge that is present only in T_w and not in T_{w2}. Since e_w is not present in T_{w2}, adding it must create a cycle. In the created cycle, there should be another edge e_{w2}, that would be present only in T_{w2} and not in T_w. Now since the weights of the edges are distinct, i.e., $w(e_w) < w(e_{w2})$ or $w(e_w) > w(e_{w2})$. Replacing the larger weight with the smaller weight in the respective MST would give a new MST, (in this case, assuming $w(e_w) < w(e_{w2})$), we get, $T = T_{w2} + e_w - e_{w2}$. Which would have the total weight, $w(T) < w(T_{w2})$. Which would mean $w(T_{w2})$ is not a MST, which is a contradiction. Similarly this can be extended for $w(e)' = w(e) - 1$ and let $w(e)'' = w(e)/2$ as the both weight functions will result in distinct weights for every edge in $w(e)'$ and $w(e)''$. ■

2. Let $T_w, T_{w'}, T_{w''} \subseteq E$ be the MSTs of $G(V, E)$ with respect to w, w' and w''. Decide whether each of the following statements are correct. Give a proof if it is true, or a counter example if it is not.

   (a) $T_{w'} = T_{w''}$.

   Solution:

   Yes, since the MSTs are unique (from previous subpart), and since all of w, w', w'' preserve the same ordering on the edges, any MST algorithm will have the same preference for picking edges. So the same set of edges will get picked. Note that this is true for any strictly monotonically increasing function of w.[1] ■

   (b) The minimum cost edge e_{min} belongs to T_w and the maximum cost edge e_{max} does not belong to T_w .

[1] Informally, any function whose output preserves the order of its inputs.

Solution:

False. The first part is true, the e_{min} edge will always belong to T_w but e_{max} may also belong to T_w. Consider the graph A — 4 — B — 100 — C. In here, $e_{min} = 4$ and $e_{max} = 100$ will both be picked up in the MST. ■

3. We are given two nodes $u, v \in V$. Assume that there exists exactly one shortest path from u to v in G with respect to w; exactly one shortest path from u to v in G with respect to w'; and exactly one shortest path from u to v in G with respect to w''. Let $P_w(u,v), P_{w'}(u,v), P_{w''}(u,v) \subseteq E$ be these shortest paths from u to v with respect to w, w' and w''. Decide whether each of the following statements are correct. Give a proof if it is true, or a counter example if it is not.

   (a) $P_w(u,v) = P_{w'}(u,v)$.

   Solution:

   False, consider:
   $A - 1.01 - B - 1.01 - C$
   |_____2_____|

   Shortest path from $A \to C$ is AC, with weight 2, but in w':
   $A - 0.01 - B - 0.01 - C$
   |_____1_____|
   Shortest path is now $A - B - C$, with weight 0.02

   ■

   (b) $P_w(u,v) = P_{w''}(u,v)$.

   Solution:

   True. For a path P, define $w(P)$ as the sum of the edge weights in w, $w''(P)$ as the sum of the edge weights in w''.
   Let P be the shortest path, we have for all other P':

$$\begin{aligned} w(P) &< w(P') \\ \sum_{e \in P} w(e) &< \sum_{e \in P'} w(e) \\ \sum_{e \in P} w(e)/2 &< \sum_{e \in P'} w(e)/2 \\ \sum_{e \in P} w''(e) &< \sum_{e \in P'} w''(e) \\ w''(P) &< w''(P') \end{aligned}$$

   Thus P is also the shortest path in w''. This proof does not work for part i because there is no transformation we can do to both sides simultaneously to get sum w(e) to be of the form $w'(e)$. ■

Problem 2 (Broad Diameters) 5 + 5 + 5

For a given connected graph $G = (V, E)$. The *distance* $d_G(v, u)$ between v and u is the length in hops of the shortest path between v and u. The *diameter* D_G of a graph G is the maximum distance among all pairs (of nodes) in V, i.e., $D_G = \max\{d_G(u, v) \mid v, u \in V\}$.
Let $a, b \in V$ be two nodes such that $d_G(a, b) = D_G$. Decide whether each of the following statements are correct and give a proof for each part.

1. For every node $r \in V$ either a or b is a leaf in a BFS tree of G rooted at r.

   Solution:

   False

   Consider a cycle with 5 nodes, labeled a, c, b, d, e in order. The diameter would be 2 and the nodes labeled a, b would be nodes that are endpoints of the diameter. For any BFS rooted at c, we will have both a and b be internal nodes with the BFS tree having the edges $(c, a), (c, b), (a, e), (b, d)$. ■

2. Node a is a leaf in any BFS tree of G rooted at b.

   Solution:

   True
   We have know $d_G(a, b) = D_G$ which means $d_G(a, b)$ is the longest path of G. Run BFS on node b in the graph, a leaf of BFS tree is the last found node. Moreover, in BFS nodes are found in distance order. Closer to the root a node is, sooner it can be found, and vice verse. Since we know a is the furthest node from b, it must be discovered last. Thus it is a leaf. On the contrary, If a is not a leaf, it must have children nodes. Then the last discovered node cannot be a but its children. So a is not the furthest node from b, which means $d_G(a, b)$ is not the longest path. The conclusion contradicts with its condition. Overall, the statement holds.

   ■

3. The depth of every BFS tree of G is at least $D_G/2$. (The depth of a tree is the depth of its deepest leaf).

   Solution:

   True
   Assume there exists one tree root on node r with depth less than $D_G/2$. The distance for any two node (a, b) will be: $d_G(a, b) \leq d_G(a, r) + d_G(r, b) \leq depth + depth < D_G/2 + D_G/2 = D_G$
   Which means the distance between every two nodes is less than D_G. Therefore the diameter of this graph should be less than D_G as well. The conclusion contradicts with its assumption. This tree does not exist.

   ■

Problem 3 (Break-ups among trees) 10

Prove that in any tree T, there exists a node u which if removed from T breaks the tree into connected components such that no connected component contains more than half the original nodes. That is, if the T had n nodes, then after removing the node u each component will have at most $\lceil \frac{n}{2} \rceil$ nodes in it.

Solution:

Algorithm:

1. Separate the graph using any node u. If the new connected components are all with less than or equal to half of the nodes, then terminate.
2. If not, go to the component with size larger than $n/2$ (there can be only one, component A). Using the node v connected with u in A as the new separation node.
3. If the new connected components are all with less than or equal to half of the nodes, then terminate. Otherwise go to step 2.
4. By moving to the node in the largest component as separation node, the components other than A (the largest component) will possibly merge together and add the separation node into its weight (add one more). However, even after the merge, because component A takes more than a half weight, all the other nodes all together will have weights less than or equal to a half. Therefore the merged tree will not have weights larger than a half. However, by using the node u in component A will decrease the weight of A by at least 1.
5. Repeat until you find a u.

By continuing finding the node connected with the separation node in the largest connected component, the size of the largest component will monotonically decrease while keep all the other connected components with weights less than or equal to one. The loop will also terminate because in the tree structure, the node in the component A connected with the current separation node will eventually reach the leave, considering no cycles in the structure. Therefore, the solution that with all connected components with less or equal than a half nodes will be reached. ■

Problem 4 (Programming: Contract Corruption) 10 + 10 + 30

Part 1: The one with the Proof

Suppose we have a graph G, and T is a spanning tree (not necessarily with minimum total weight) of G. Consider the following operation $SWAP(T, e_1, e_2)$, where we remove e_1 from T (conditioned that it is already in T) and add e_2 to T. If the resulting graph $T' = SWAP(T, e_1, e_2)$ is also a spanning tree of G, then we will call this a *valid* swapping. Show that for any pair of spanning trees T, T' of G, it is possible to transform T into T' by a sequence of valid swapping operations.

Hint: Show by mathematical induction that this is possible if T and T' differ by k edges

Solution:

Induct on k, the number of edges which are in T_1 but not T_2. When $k = 1$, then by the definition of our $SWAP$ operation we can use a single $SWAP$ to transform T_1 into T_2.

Suppose it is true for some value k. We will show it is true when T_1 and T_2 differ by $k+1$ edges. Let e_1 be some edge which is in T_1 but not in T_2. Consider the result T_2' of adding e_1 to T_2. Since

T_2 is a spanning tree, $T_2 + e_1$ now has a cycle. This cycle must contain an edge which is not in T_1 (we know this because if all the edges where in T_1, T_1 would have a cycle, which it doesn't). Call this edge e_2. If we now delete e_2, T_2 will be a spanning tree again, while being one edge closer to T. Then define $T' = SWAP(T_2, e_2, e_1)$. T' now differs from T_1 by k edges. Thus by our inductive hypothesis we can use k swap operations to get from T' to T_1. Thus we can use $k+1$ operations to get from T_2 to T_1.

■

Part 2: The one with the Algorithm

Describe and give a proof of correctness for the algorithm you used to solve the programming assignment

Hint: Use the results from the solution to the previous problem

Solution:

We first find the spanning tree that uses the least number of contracts from company A (let that number be l_A) and also find the spanning tree that uses the most number of contracts from company A (let that number be h_A). Note that we can find h_A by finding the l_B. Also note that in both these cases the edge weights are all 1, we can make it into an MST problem by assigning edge weights of 1 for edges of company A and 0 for those of B. This will give us l_A, and to find h_A we can flip the edge weights. However, we can also solve this problem by finding these values using BFS since all the weights are 1 (BFS can find a spanning tree in unweighte graphs).

If $l_A \leq (n-1)/2$ and $h_A \geq (n-1)/2$ then using the result from the previous subpart we know that given these spanning trees we can find a spanning tree that uses any number of contracts between these two. So we know that the best bias is 1 if $(n-1)/2$ is not divisible by 2 and 0 otherwise. If that is not the case but $l_A > (n-1)/2$, then we know that the bias has to be $|l_A - (n-1-l_A)|$. If on the other hand, $l_A < (n-1)/2$, then we use h_A and the best bias possible is going to be $h_A - (n-1-h_A)$. Note that in both these cases whether to use l_A or h_A is dictated by which is closer to $(n-1)/2$ as that is the choice to minimize the bias. ■

Part 3: The one with the Coding

You are on a committee for figuring out the best way to install power grid infrastructure within your community. Currently, there are N distribution centers that have been constructed. It has been decided that all of these distribution centers must be connected in a network. In order to construct the power line connections, the work must be outsourced to private companies.

There are two main power mainatanance companies that will construct such connections: Maverick and Desperado[2]. You have approached each one with information regarding your distribution centers, and an amount you are willing to pay for each one, which is constant independent of what distribution centers you are trying to connect. In response, both companies have responded with a list of connections that they are willing to build given your proposed price. A given connection connects two of the distribution centers.

Since each contract has the same price, you will minimize your costs by minimizing the number of contracts you give out. A set of contracts is valid if and only if it satisfies the following:

- All distribution centers are connected within the power grid. There can't be isolated centers; that is, should exist a path between every pair of centers.
- The total number of contracts given out is minimized.

[2]Iceman would have been a better name here but I don't want to change old test cases.

Given their responses, you believe it should be relatively straight forward to figure out how to connect all the distribution centers for minimum cost. However, there is a problem, you have shares in both companies and do not want to look biased towards either. This means you cannot give away too many of the contracts to a single company or else you will be charged with contract corruption.

Suppose that you give A contracts out to Maverick, and B contracts out to Desperado. Then the bias of your assignments will be the absolute value difference of A and B. Your updated goal is to figure out which subset of contracts to give out, so that all of the distribution centers are connected, costs are minimized, and the bias of your contract selection is minimized.

Write a program to take the contract information, and determine how small of a bias you can achieve for a valid set of contracts. You do not need to determine the actual set of contracts to be outputted.

Input Format:

Line 1 : N C.

The first line indicates there are N distribution centers labeled 1 through N, and there are C contracts to consider

Next C lines: i j c

Each of the following lines indicates a contract where i and j are distribution center labels, and c is either "MAVERICK" or "DESPERADO", the name of the company willing to purchase this contract.

The set of contracts will always be enough to yield at least one valid set of contracts.

Output Format:

Line 1: V

Indicating V is the minimum possible bias for a valid set of contracts.

Sample Input:

3 3
1 2 MAVERICK
2 3 MAVERICK
1 3 DESPERADO

Sample Output:

0

Explanation:

For 3 nodes, you need at least 2 contracts to connect them all. It is possible to select the following contracts:
1 2 MAVERICK
1 3 DESPERADO

Then all of the distribution centers will be connected, while the bias of the contract set is $|1 - 1| = 0$.

Solution:

```
#include <bits/stdc++.h>

using namespace std;

int runBFS(vector <vector <int>> graph[], int n, int ty) {
    int total = 0;
    vector<int> visited(n+1, 0);

    for (int i = 1;i<=n;i++) {
        if (visited[i] == 1) {
```

```
            continue;
        }
        vector <int> q;
        q.push_back(i);
        visited[i] = 1;
        while (q.size() != 0) {
            int temp = q.back();
            q.pop_back();
            for (int j = 0; j < graph[temp].size();j++) {
                vector <int> z = graph[temp][j];
                if (z[1] == ty && visited[z[0]] == 0) {
                    visited[z[0]] = 1;
                    q.push_back(z[0]);
                }
            }
        }
        total++;
    }
    return total;
}

int main () {
    int n, m;
    cin>>n>>m;
    vector <vector <int>> graph[n+1];
    for (int i = 0;i<m;i++) {
        int a, b, c;
        string s;
        cin>>a>>b>>s;
        if (s == "MAVERICK") {
            c = 0;
        }
        else {
            c = 1;
        }
        vector <int> z;
        z.push_back(b);
        z.push_back(c);
        vector <int> y;
        y.push_back(a);
        y.push_back(c);
        graph[a].push_back(z);
        graph[b].push_back(y);
    }

    int A = runBFS(graph, n, 1) - 1;
    int B = runBFS(graph, n, 0) - 1;
    int notA = (n-1) - B;

    int mi = min(notA, A);
    int mx = max(notA, A);
    int best_bias = (n-1)/2;
    if (mi <= best_bias and mx >= best_bias) {
        cout<<(n-1) % 2;
    }
    else {
        if (mi > best_bias) {
            int C = n - 1 - mi;
            cout<<max(mi - C, C - mi);
        }
        else{
```

```
            int C = n - 1 - mx;
            cout<<max(mx - C, C - mx);
        }
    }



    return 0;
}
```

Listing 1: C/C++ solution

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;

public class SolutionContractCorruption {

  private static final String A = "DESPERADO";
  private static final String B = "MAVERICK";

  static int bfs_cc_count(
      Map<Integer, List<Pair>> graph, String label) {
    int count = 0;
    List<Integer> queue = new ArrayList<>();
    Set<Integer> seen = new HashSet<>();
    for (Integer node : graph.keySet()) {
      if (seen.contains(node)) {
        continue;
      }
      queue.add(node);
      seen.add(node);
      while (!queue.isEmpty()) {
        int x = queue.remove(0);
        for (Pair pair : graph.get(x)) {
          int neighbour = pair.getFirst();
          if (pair.getContracter().equals(label) &&
                !seen.contains(neighbour)) {
            seen.add(neighbour);
            queue.add(neighbour);
          }
        }
      }
      count++;
    }
    return count;
  }

  public static void main(String[] args) {

    int N, M;
    Scanner input = new Scanner(System.in);
    N = input.nextInt();
    M = input.nextInt();
    input.nextLine();
    Map<Integer, List<Pair>> graph = new HashMap<>();
    for (int i = 0; i < M; i++) {
```

```java
      String[] next = input.nextLine().split(" ");
      int edge_0 = Integer.parseInt(next[0]);
      int edge_1 = Integer.parseInt(next[1]);
      String edge_2 = next[2];
      graph.computeIfAbsent(edge_0, k -> new ArrayList<>());
      graph.computeIfPresent(edge_0, (k, v) -> {
        v.add(new Pair(edge_1, edge_2));
        return v;
      });

      graph.computeIfAbsent(edge_1, k -> new ArrayList<>());
      graph.computeIfPresent(edge_1, (k, v) -> {
        v.add(new Pair(edge_0, edge_2));
        return v;
      });
    }

    int A_max = bfs_cc_count(graph, A) - 1;
    int A_min = N - bfs_cc_count(graph, B);
    int mst_val = Math.min(A_min, A_max);
    int max_st_val = Math.max(A_min, A_max);
    int best_bias = (N - 1) / 2;
    if (mst_val <= best_bias && max_st_val >= best_bias) {
      System.out.println((N - 1) % 2);
    } else {
      if (mst_val > best_bias) {
        int tmp = N - 1 - mst_val;
        System.out.println(Math.abs(mst_val - tmp));
      } else {
        int tmp = N - 1 - max_st_val;
        System.out.println(Math.abs(max_st_val - tmp));
      }
    }
  }

  static class Pair {

    int first;
    String contracter;

    Pair(int first, String contracter) {
      this.first = first;
      this.contracter = contracter;
    }

    int getFirst() {
      return first;
    }

    String getContracter() {
      return contracter;
    }
  }

}
```

Listing 2: Java solution

```python
A = "DESPERADO"
B = "MAVERICK"

def leader(i,parent):
```

```
    if parent[i] != i:
        parent[i] = leader(parent[i],parent)
    return parent[i]

def union(i,j,par,size):
    i = leader(i,par)
    j = leader(j,par)
    if i != j:
        (low_size, low_index), (max_size, max_index) = sorted([(size[i],i), (size[j], j)
    ])
        new_size = max_size + low_size
        par[low_index] = max_index
        size[low_index] = new_size
        size[max_index] = new_size
    return

def mst(N, edges, modifier):
    if modifier:
        edges = sorted(map(lambda e: (-1 * e[0], e[1], e[2]), edges))
    else:
        edges = sorted(edges)
    parents = range(N+2)
    sizes = [1]*(N+2)

    components = N
    MST_val = 0
    for e in edges:
        if components == 1:
            break
        val, i, j = e
        if leader(i, parents) != leader(j, parents):
            union(i, j, parents, sizes)
            components -= 1
            MST_val += val
    if modifier:
        return MST_val * -1
    else:
        return MST_val

if __name__=="__main__":
    edges = []
    N, M = map(int, raw_input().split())
    assert M <= 250000, M
    d = {}
    for _ in range(M):
        s = raw_input().split()
        edge = [int(s[0]), int(s[1])]
        if str(s[2]) == A:
            edge = [1] + edge
        else:
            edge = [0] + edge
        edges.append(edge)
        d[edge[1]] = 1
        d[edge[2]] = 1
    assert len(d) == N

    mst_val = mst(N, edges, False)

    max_st_val = mst(N, edges, True)
    best_bias = (N-1)/2

```

```
    if mst_val <= (N-1)/2 and max_st_val >= (N-1)/2:
        print (N-1) % 2
    else:
        if mst_val > (N-1)/2:
            B = N - 1 - mst_val
            print max(mst_val - B, B - mst_val)
        else:
            B = N - 1 - max_st_val
            print max(max_st_val - B, B - max_st_val)
```

Listing 3: Python solution

■