Fall '15 CS 5800 Algorithms | Hackerrank Group: akashsingh245

Problem Set 3

Surekha Jadhwani, Akash Singh, and Ayush K. Singh

1 Part A

Answers

1. Given a connected undirected graph $G = (V, E)$ such that $|E| > n$, and $w : E(1, \infty)$ 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 every $e, e' \in E$ such that $e \neq e'$). 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 $e \in E$.

   (a) 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:
   Given: $w : E(1, \infty)$ is a $one - to - one$ weight function on the edges of graph G.
   $w' = w(e) - 1$, w'
   $w''(e) = w(e)/2$
   To Prove: There exists a unique MST of $G(V, E)$ with respect to w, one MST of $G(V, E)$ with respect to w' and one MST of $G(V, E)$ with respect to w''.
   Proof: Proof by contradiction:

      i. For edge function w:
      Assume that there exists two distinct minimum spanning trees m_1 and m_2 for graph G with respect to weight function w.
      Let e_1 be a minimum weight edge that is present only in m_1 and not in m_2 as both are distinct and have equal number of edges.
      If edge e_1 is added to m_2, then m_2 will contain a cycle.
      Let e_2 be one of the edges present only in m_2 which is forming a cycle with the addition of e_1.
      Since, w is a one-to-one function, we have, $w(e_1) < w(e_2)$.
      Let m be the new spanning tree which is created when e_1 is added to m_2 and e_2 is removed from it.
      This implies that the total weight of m is less than the total weight of m_2.
      This is a contradiction as m_2 was supposed to be a minimum spanning tree.
      Hence, there exists exactly one MST of $G(V, E)$ with respect to w.

ii. For edge function w':
Since, $w : E(1, \infty)$ is a $one - to - one$ weight function on the edges of graph G.
$w' = w(e) - 1$ for every $e \in E$, w' is also a $one - to - one$ weight function on the edges of graph G as subtracting 1 from a set of distinct numbers results in another set of distinct numbers.
Hence, from point (i) above, we can conclude that, there exists exactly one MST of $G(V, E)$ with respect to w'.

iii. For edge function w'':
Similarly, $w''(e) = w(e)/2$ is also a $one - to - one$ weight function as w is a $one - to - one$ weight function and division by 2 will also result in another $one - to - one$ weight function.
Hence, from point (i) above, we can conclude that, there exists exactly one MST of $G(V, E)$ with respect to w''.

(b) 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.

i. $T_{w'} = T_{w''}$.
Solution: The given statement is true.
Proof by contradiction:
Since, w is a one-to-one function, w' and w'' are also $one - to - one$ weight functions on the edges of graph G.
As, $w' = w(e) - 1$ and $w''(e) = w(e)/2$, the sequence of edges in increasing order of weight for graph G with respect to w' and w'' is same.
Let there exists a minimum weighted edge e of graph G.
Now, if e is minimum weighted edge in graph G w.r.t. w', it should be a minimum weighted edge in graph G w.r.t. w''.
Assuming, $T_{w'}$ and $T_{w''}$ are distinct $MSTs$ of graph G and edge e is part of $T_{w'}$ but not of $T_{w''}$.
If we add edge e in $T_{w''}$ by removing another edge which forms the cycle, a new spanning tree is created whose total cost is less than $T_{w''}$.
This is a contradiction as $T_{w''}$ is a minimum spanning tree.
Hence, $T_{w'} = T_{w''}$.

ii. The minimum cost edge e_{min} belongs to T_w and the maximum cost edge e_{max} does not belong to T_w.
Solution: The given statement is partially correct.
For "The minimum cost edge e_{min} belongs to T_w":
This statement is always true.
Proof by contradiction:
Suppose that e_{min} does not belong to T_w.
If we add e_{min} to T_w and remove other edge, say e_1 which forms a cycle, a new spanning tree is created.
The total weight of this new tree is less than that of T_w.
This is a contradiction as T_w is a minimum spanning tree.
Hence, the minimum cost edge e_{min} belongs to T_w.
For "The maximum cost edge e_{max} does not belong to T_w."
This statement can be true in some cases and false in some cases depending on the graph.
If the graph is already a tree, then all of its edges will be part of MST.
Hence, if e_{max} does not belong to T_w, the resulting tree would be disconnected.
In this case, e_{max} belongs to T_w.
Thus, the given statement is not correct always.

(c) 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.

i. $P_w(u,v) = P_{w'}(u,v)$.
Solution: The given statement is true.
Proof by contradiction:
Since, $w' = w(e) - 1$ and w is a one-to-one function, w' is also $one - to - one$ weight function on the edges of graph G.
As, $w' = w(e) - 1$, the sequence of edges in increasing order of weight for graph G with respect to w and w' is same.
Let there exists a minimum weighted edge e of graph G on the path from u to v.
Now, if e is minimum weighted edge in graph G w.r.t. w, it should be a minimum weighted edge in graph G w.r.t. w'.
Assuming, $P_w(u,v)$ and $P_{w'}(u,v)$ are not equal for graph G and edge e is included in $P_w(u,v)$ but not in $P_{w'}(u,v)$.
If we add edge e in $P_{w'}(u,v)$ by removing another edge which forms the cycle, a path is created from u to v whose total cost is less than $P_{w'}(u,v)$.
This is a contradiction as $P_{w'}(u,v)$ is the shortest path from u to v and there exists exactly one shortest path from u to v in graph G w.r.t. w'.
Hence, $P_w(u,v) = P_{w'}(u,v)$.

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

Solution: The given statement is true.

Proof by contradiction:

Since, $w'' = w(e)/2$ and w is a one-to-one function, w'' is also $one - to - one$ weight function on the edges of graph G.

As, $w'' = w(e)/2$, the sequence of edges in increasing order of weight for graph G with respect to w and w'' is same.

Let there exists a minimum weighted edge e of graph G on the path from u to v.

Now, if e is minimum weighted edge in graph G w.r.t. w, it should be a minimum weighted edge in graph G w.r.t. w''.

Assuming, $P_w(u,v)$ and $P_{w''}(u,v)$ are not equal for graph G and edge e is included in $P_w(u,v)$ but not in $P_{w''}(u,v)$.

If we add edge e in $P_{w''}(u,v)$ by removing another edge which forms the cycle, a path is created from u to v whose total cost is less than $P_{w''}(u,v)$.

This is a contradiction as $P_{w''}(u,v)$ is the shortest path from u to v and there exists exactly one shortest path from u to v in graph G w.r.t. w''.

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

2. Let $G = (V, E)$ be a connected undirected graph, with a weight function $w : E \rightarrow \{1, 2\}$. (In other words, all edges of G have weight 1 or 2) Give an efficient $O(|E|)$ algorithm that computes the shortest paths from a single source vertex $s \in V$ to all of the other vertices in the graph $G(V, E)$.

Solution: The given graph has edges of weights either 1 or 2. It is possible to split the edge of weight 2 into two edges, each having its weight = 1. If we do this for all the edges of weight 2, we will have a graph that contains edges having weight = 1. On this updated version of the graph, we can implement modified BFS to find the shortest path from source s to all vertices in the graph G.

Psuedocode:

```
for e in E do
 if e.weight = 2
  Split e into two edges by creating intermediate vertex in V
 end if
end for
Create an integer array Distance[V.size]
for v in V do
 Distance[v] ← ∞
 Mark vertice v as unvisited
end for
frontier = new Queue() //initialize an empty queue frontier
// Starting BFS for source vertex s
mark s as visited
set Distance[s] = 0
frontier.push(s)
while frontier not empty do
 Vertex v' = frontier.pop()
 for each successor u of v'
  if u unvisited
   frontier.push(u)
   mark u as visited
   Distance[u] = Distance[v'] + 1
  end if
 end for
end while
output Distance[]
```

Complexity: Worst-case analysis:

In the worst case scenario, the graph will have all the edges of weight 2. We would need to do $O(|E|)$ operations to split all these edges. Also, the complexity of BFS is $O(|V|+|E|)$. Hence the total time complexity will become $O(|E|)+O(|V|+|E|)$ which is equivalent to $O(|V| + |E|)$.

3. Given a set of numbers, its *median*, informally, is the halfway point of the set. When the sets size n is odd, the median is unique, occurring at the i^{th} value, where $i = (n+1)/2$. When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2 + 1$, which are called the lower median and upper median, respectively. For simplicity in this question, we use the phrase the median to refer to the lower median. Design and implement a data structure $\mathcal{D}$ to maintain a set of positive integers that supports $Build, Insert, Extract_X$ and $Plot_X$ (for $X \in \{min, max, median\}$) operations, defined as follows:

   - $Build_{\mathcal{D}}(S)$: Produces, in linear time, a data structure $\mathcal{D}$ for the set S from an unordered input array. (For implementing $Build_{\mathcal{D}}(S)$, you can use Procedure $FindMed(S)$, which finds the median of S in linear time.)
   - $Insert(\mathcal{D}, x)$: insert element x into $\mathcal{D}$ in $O(logn)$ time.
   - $Plot_{min}(\mathcal{D}), Plot_{max}(\mathcal{D}), Plot_{median}(\mathcal{D})$: Returns, in $O(1)$ time, the value of the minimum, maximum and the median of $\mathcal{D}$, respectively.
   - $Extract_{min}(\mathcal{D}), Extract_{max}(\mathcal{D}), Extract_{median}(\mathcal{D})$: Remove and return, in $O(logn)$ time, the value of the minimum, maximum and the median of $\mathcal{D}$, respectively. (Note that you will be removing and returning exactly one of these, depending on which of the three parameters $Extract$ was called with.)

     Solution:

     Design: In order to implement the given functionality, we can use 'min-max-median' heap variant.

     In this heap,

     (a) The root element represents the median value.

     (b) All the elements smaller than median value are stored in left sub-tree in the form of a min-heap

     (c) All the elements larger than median value are stored in right sub-tree in the form of a max-heap

     The maximum size of left subtree(H_{min}) is $\lfloor (n/2) \rfloor$ and that of right subtree(H_{max}) is $\lceil (n/2) \rceil$.

     Implementation of data structure and its operations:

     Initialization: This structure consists of an array which stores the median value in the first element, min value in the second element and max value in the third element.

     structure $\mathcal{D}$:
       Create an integer array $H[n]$
     end structure

$Build_{\mathcal{D}}(S)$**:** Here, we first partition the elements into two sets, one for each subtree and then create heaps using those sets.

function $Build_{\mathcal{D}}(S)$:
$median = FindMed(S)$
Create two empty arrays $S_{min}[\lfloor n/2 \rfloor]$ and $S_{max}[\lceil n/2 \rceil]$
//Partitioning
for x in S:
if $x \geq median$:
Add x to S_{max}
else
Add x to S_{min}
end if
end for
//Creating min-max-median heap
$H[1] \leftarrow median$
if $S_{min}.size > 0$
Create a min-heap from elements of S_{min}
Add the min-heap in H starting at location $H[2]$
end if
if $S_{max}.size > 0$
Create a max-heap from elements of S_{max}
Add the max-heap in H starting at location $H[3]$
end if
end function

Complexity: Since, we are iterating through all the numbers once for partitioning, and then creating heap in linear time, the running time complexity is $O(n)$.

$Insert(\mathcal{D}, x)$: After each insertion, we calculate the new median value and try to maintain equal size of both subtrees.

function $Insert(\mathcal{D}, x)$
if $x > H[1]$:
Insert x in right subtree
else
Insert x in left subtree
end if
Reheapify tree and update median value
end function

Complexity: Heap is a binary tree and in worst case, for reheapify, we start from root and come till leaf node. Hence, the running time complexity is $O(logn)$.

$Plot_{min}(\mathcal{D})$**:** As per the design, second element of array contains the min-value

function $Plot_{min}(\mathcal{D})$:
if $n = 0$:
return -1
else if $n = 1$:
return $H[1]$
else
return $H[2]$
end if

Complexity: Since, there are no loops or recursions, running time complexity is $O(1)$.

$Plot_{max}(\mathcal{D})$**:** As per the design, third element of array contains the max-value
function $Plot_{max}(\mathcal{D})$:
if $n = 0$:
return -1
else if $n = 1$:
return $H[1]$
else
return $H[3]$
end if

Complexity: Since, there are no loops or recursions, running time complexity is $O(1)$.

$Plot_{median}(\mathcal{D})$**:** As per the design, first element of array contains the median-value
function $Plot_{median}(\mathcal{D})$:
if $n = 0$:
return -1
else
return $H[1]$
end if

Complexity: Since, there are no loops or recursions, running time complexity is $O(1)$.

$Extract_{min}(\mathcal{D})$**:** Here, we extract the minimum value first and then reheapify the tree to balance it.
function $Extract_{min}(\mathcal{D})$:
Delete $H[2]$
Move $H[n]$ to $H[2]$
Reheapify the tree and update the median value.
end function

Complexity: Heap is a binary tree and in worst case, for reheapify, we start from root and come till leaf node. Hence, the running time complexity is $O(logn)$.

$Extract_{max}(\mathcal{D})$**:** Here, we extract the maximum value first and then reheapify the tree to balance it.

function $Extract_{max}(\mathcal{D})$:
Delete $H[3]$
Move $H[n]$ to $H[3]$
Reheapify the tree and update the median value.
end function

Complexity: Heap is a binary tree and in worst case, for reheapify, we start from root and come till leaf node. Hence, the running time complexity is $O(logn)$.

$Extract_{median}(\mathcal{D})$**:** Here, we extract the minimum or maximum value depending on the size and then reheapify the tree to balance it.

function $Extract_{median}(\mathcal{D})$:
if $H_{max}.size > H_{min}.size$
Delete $H[3]$
Move $H[n]$ to $H[3]$
else
Delete $H[2]$
Move $H[n]$ to $H[2]$
end if
Reheapify the tree and update the median value.
end function

Complexity: Heap is a binary tree and in worst case, for reheapify, we start from root and come till leaf node. Hence, the running time complexity is $O(logn)$.

2 Part B (programming)

1. **Preliminary Problem**
   Suppose we have a graph G, and T is a spanning tree (not necessarily with minimum total weight) of G. Consider the following operation $SWAP(T, e1, e2)$, where we remove e_1 from T (conditioned that it is already in T) and add e_2 to T . If the resulting graph $T' = SWAP(T, e1, e2)$ 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)
   Proof: By Mathematical Induction:
   Let P(n) : It is possible to transform a spanning tree T into another spanning tree T' for a graph G if they differ by n edges.
   Base case: For $P(1)$, we have two spanning trees T and T' which differ by 1 edge.
   Let e' be that edge which is present in T' but not in T.
   If we add e' in T, a cycle will be formed as T was already a spanning tree.
   Let e be the edge which forms the cycle after addition of e'.
   We can remove e from the new graph as it is a valid swapping.
   After removal of e from the new graph, a spanning tree is created which is equivalent to T'.
   Hence, $P(1)$ is true.
   Inductive Step: Since $P(1)$ is true, we assume $P(n)$ is true.
   For $P(n+1)$: we have two spanning trees T and T' which differ by $(n+1)$ edges.
   Let T_1 be a sub-tree of T which has 1 edge, say e, less than T.
   Let T_1' be a sub-tree of T' which has 1 edge, say e', less than T'.
   Since, T and T' differ by $(n+1)$ edges, T_1 and T_1' differ by n edges.
   By induction hypothesis, we can transform T_1 into T_1' using a set of valid swapping operations.
   If we add edge e back to T_1', the resulting tree would differ T' by only that edge.
   This edge can be replaced with e' which is the cycle forming edge.
   Hence, $P(n+1)$ is true.
   Program: Below is a sample program which determines whether edge swapping operation for given spanning tree is valid or not.

```
import java.util.ArrayList;

public class SpanTreeSWAP {

   public static boolean SWAP(ArrayList<VertexList> spanning_tree, Edge e1,
       Edge e2)
   {
      // Re-initializing subsets
      init(1000);
```

```
   // Connecting all edges of Spanning Tree except e1 and e2.
   for (VertexList curr_ver_list : spanning_tree)
   {
      for (Integer adjacent_vertex : curr_ver_list.adjacent_vertices)
      {
         if (!(edgeContainsVertices(e1, curr_ver_list.vertex,
             adjacent_vertex) ||
            edgeContainsVertices(e2, curr_ver_list.vertex,
                adjacent_vertex)))
         {
            union(curr_ver_list.vertex, adjacent_vertex);
         }
      }
   }

   // Try to add e2
   return (!(find(e2.vertex1) == find(e2.vertex2)));
}

public static boolean edgeContainsVertices (Edge e, int vertex1, int
    vertex2)
{
   return ((vertex1 == e.vertex1 && vertex2 == e.vertex2) ||
         (vertex2 == e.vertex1 && vertex1 == e.vertex2));
}

public static class VertexList
{
   public VertexList(int vertex, ArrayList<Integer> adjacent_vertices)
   {
      this.vertex = vertex;
      this.adjacent_vertices = adjacent_vertices;
   }

   int vertex;
   ArrayList<Integer> adjacent_vertices = new ArrayList<Integer>();
}

public static class Edge
{
   int vertex1;
   int vertex2;

   Edge(int vertex1, int vertex2)
   {
```

```
      this.vertex1 = vertex1;
      this.vertex2 = vertex2;
   }
}

// Data structure Disjoint Data Set

static Subset[] subsets = new Subset[1000];

public static void init(int n)
{
   for (int i = 0; i < n; i++)
   {
      subsets[i] = new Subset();
   }
}

public static void makeSet(int x)
{
   subsets[x].parent = x;
   subsets[x].rank = 0;
}

public static int find(int x)
{
   if (subsets[x].parent != x)
      subsets[x].parent = find(subsets[x].parent);

   return subsets[x].parent;
}

public static void union(int x, int y)
{
   int x_root = find(x);
   int y_root = find(y);

   if (x_root == 0)
   {
      makeSet(x);
      x_root = x;
   }

   if (y_root == 0)
   {
      makeSet(y);
```

```
      y_root = y;
    }

    if (x_root == y_root)
       return;

      // x and y are not already in same set. Merge them.
      if (subsets[x_root].rank < subsets[y_root].rank)
          subsets[x_root].parent = y_root;
      else if (subsets[x_root].rank > subsets[y_root].rank)
          subsets[y_root].parent = x_root;
      else
      {
        subsets[y_root].parent = x_root;
        subsets[x_root].rank = subsets[x_root].rank + 1;
      }
}

 static class Subset{
   int parent;
   int rank;
 }

 public static void main(String[] args)
 {
   init(1000);

   // Test data
   ArrayList<VertexList> st = new ArrayList<VertexList>();

   ArrayList<Integer> adj_v = new ArrayList<Integer>();
   adj_v.add(3);
   st.add(new VertexList(1, adj_v));
   st.add(new VertexList(2, adj_v));
   adj_v = new ArrayList<Integer>();
   adj_v.add(1);
   adj_v.add(2);
   adj_v.add(4);
   adj_v.add(6);
   st.add(new VertexList(3, adj_v));
   adj_v = new ArrayList<Integer>();
   adj_v.add(3);
   adj_v.add(5);
   st.add(new VertexList(4, adj_v));
```

```
adj_v = new ArrayList<Integer>();
adj_v.add(4);
st.add(new VertexList(5, adj_v));

adj_v = new ArrayList<Integer>();
adj_v.add(3);
st.add(new VertexList(6, adj_v));

// Case 1
System.out.println("Case 1");
if (SWAP(st, new Edge(3, 6), new Edge(2, 6)))
{
   System.out.println("Edge Swap is Valid");
}
else
{
   System.out.println("Edge swap is Invalid");
}
// Case 2
System.out.println("Case 2");
if (SWAP(st, new Edge(3, 6), new Edge(3, 5)))
{
   System.out.println("Edge Swap is Valid");
}
else
{
   System.out.println("Edge swap is Invalid");
}
// Case 3
System.out.println("Case 3");
if (SWAP(st, new Edge(4, 3), new Edge(1, 5)))
{
   System.out.println("Edge Swap is Valid");
}
else
{
   System.out.println("Edge swap is Invalid");
}
// Case 4
System.out.println("Case 4");
if (SWAP(st, new Edge(3, 1), new Edge(2, 1)))
{
   System.out.println("Edge Swap is Valid");
}
else
```

```
        {
            System.out.println("Edge swap is Invalid");
        }
        // Case 5
        System.out.println("Case 5");
        if (SWAP(st, new Edge(2, 3), new Edge(5, 6)))
        {
            System.out.println("Edge Swap is Valid");
        }
        else
        {
            System.out.println("Edge swap is Invalid");
        }
    }
}
```

Output:

```
Case 1
Edge Swap is Valid
Case 2
Edge swap is Invalid
Case 3
Edge Swap is Valid
Case 4
Edge Swap is Valid
Case 5
Edge swap is Invalid
```

2. Programming Problem:

Group Hackerrank Name: akashsingh245

(a) Surekha Jadhwani : surekha@ccs.neu.edu / jadhwani.s@husky.neu.edu

(b) Akash Singh : singhaka@ccs.neu.edu / singh.aka@husky.neu.edu

(c) Ayush K. Singh : singhay@ccs.neu.edu / singh.ay@husky.neu.edu

3. **Analysis of programming problem algorithm:** Describe and give a proof of correctness for the algorithm you used to solve the programming assignment. (Hint: Use the results from the first problem of Part B)

   Correctness:

   (a) Termination: As the number of nodes are limited and we are visiting each node just once, the algorithm terminates successfully.

   (b) Correctness: The algorithm gives correct value can be proved as follows:
   Proof by Induction: Let $P(k)$: For a k-node graph, it is possible to minimize *bias* and it can be calculated using the formulae: $(bias = req\_connections \ mod \ 2)$ when both maverick and desperado can connect more than half the nodes without forming a cycle independently or $(bias = Math.abs(2 * (req\_conns\_by\_each - desperado\_connectible\_count)))$ for other cases.
   Base Case: For P(3):

```
3 3
1 2 MAVERICK
2 3 MAVERICK
1 3 DESPERADO
```

   In the above case, we can connect 1 Desperado edge without forming a cycle and 2 maverick edges without forming a cycle. As we know that we can swap the edges to transform one spanning tree to another and that a spanning tree always exists such that it connects all distribution centers, we can use the formula $(bias = req\_connections \ mod \ 2)$ to compute the minimum *bias*.
   Both these values are greater or equal to 1 (which is $(n-1)/2$).
   Thus *bias* is $2 \ mod \ 2 = 0$ using the formula used in the program $(bias = req\_connections \ mod \ 2)$.
   Hence this base case is true.
   Considering another case as an extension to the above case:$P(4)$

```
4 4
1 2 MAVERICK
2 3 MAVERICK
4 3 MAVERICK
1 3 DESPERADO
```

   In above case, we can connect 3 maverick edges without a cycle but only 1 desperado edge without a cycle and hence bias by using the formula in the program $(bias = Math.abs(2 * (req\_conns\_by\_each - desperado\_connectible\_count)))$ is 1, which is yet again true. In this case, we can connect the firm which can cater lesser contracts and form the remainder of the tree using the contracts of the other firm with higher bias. As we know, we can join any two vertices of disconnected subtrees of the same graph without creating a cycle, the above is true.
   Induction step:

Given that the scenarios like above are true, we will assume $P(k)$ is true.
Now we need to prove this for $P(k+1)$.
As k-node graph is a spanning tree and if we add a new node, only one more edge is needed to connect to it.
If we have more than or equal to half of the edges needed to connect the centers and the new node can be connected using a desperado edge only, then it is possible to swap some other edge of desperado with maverick if bias is going in favor of desperado and hence the formula $((n-1)mod2)$ will still give the right answer.
Else, if one of the edges that the firm can connect without forming a cycle in its own contracts are less than the half of the required connections, and if the new edge is say of the firm having lower contracts, then the formula used in the program will reduce the bias correctly.
This will happen because it uses the number of contracts the firm can furnish without forming a cycle, and as the new node is also a contract without a cycle, it will be added to the previous value of contracts the firm had.
And if the firm with the higher contracts is again the only option to connect to the new center, then the formula will keep the contracts of firm with lower contracts as is and the bias will get increased.
Since we previously proved that it is true for $P(k)$ and adding a new edge, updates the bias correctly, we know that we are getting the right output for $P(k+1)$.
Hence this ensures the correctness of the program.

Complexity:
We are iterating through all the contracts and using $union-find$ data structure within it, and hence the complexity is of the form $O(mlogn)$. Hence the worst case complexity is $O(mlogn)$.

4. **Finding the tree:** For the programming assignment, we only asked that you compute the value of the minimum possible bias, instead of finding a set of contracts that achieves it. Give and analyze an algorithm which finds and returns a set of valid contracts which achieve the minimum possible bias.

Algorithm:

//Develop a data structure link cut tree which has the following operations:
$link(v, u) \leftarrow$ Link vertices v and u to show that they are connected by an edge
$cut(v, u) \leftarrow$ Remove the edge between the vertices v and u
$areConnected?(v, u) \leftarrow$ True iff u is reachable from v via a connected path
//Once the data structure is developed, use it to find the tree as follows:
Begin
Create 4 arrays to store the $maverick\_result\_edges[], desperado\_result\_edges[]$, $maverick\_non\_result - edges[]$ and $desperado\_non\_result\_edges[]$
For all the c contracts
If $maverick\_edge$
Add to $maverick\_non\_result\_edges[]$
Else
Add to $desperado\_non\_result\_edges[]$
end if
End for
Initialize rem_edges to $n(no\_of\_distribution\_centers) - 1$
Init Flags $maverick\_flag$ and $desperado\_flag$ to $true$ which will indicate whether a $maverick\_non\_result\_edge$ or $desperado\_non\_result\_edge$ can be connected without forming a cycle
While ($remaining\_edges\_counter > 0$)
// Try to add an edge of desperado and an edge of maverick at a time to keep $bias$ min
// if connecting edge of one without a cycle is not possible, form tree using edges of other
If (($bias >= 0$ or $!maverick\_flag$) and $desperado\_flag$)
$Desperado\_flag = false$
For all $non\_result\_desperado\_edges$
If ($!areConnected$(vertices of the current desperado edge))
$Link$(Vertices of current edge)
Add current edge to $desperado\_result\_edges$;
Remove current edge from $desperado\_non\_result\_edges$
$bias - -; rem\_edges - -; desperado\_flag = true$;
Break;
End if
End for
End if
Else If (($bias < 0$ or $!desperado\_flag$) and $maverick_flag$)

$Maverick\_flag = false$;
For all $non\_result\_maverick\_edges$
If (!$areConnected$(vertices of the current maverick edge))
$Link$(Vertices of current edge)
Add current edge to $maverick\_result\_edges$;
Remove current edge from $maverick\_non\_result\_edges$
$bias++; rem\_edges--; maverick\_flag = true$;
Break;
End if
End for
End if
End while
// Tree formed, Reduce bias
If ($current\_bias > 0$)
// try swapping currently connected maverick edge with an unconnected desperado edge
For all $Maverick\_result\_edges$
For all $desperado\_non\_result\_edges$
If $SWAP(current\_maverick\_edge, current\_desperado\_edge)$
Cut (vertices of $current\_maverick\_edge$)
Link (vertices of $current\_desperado\_edge$)
Add current desperado edge to $desperado\_result\_edges$;
Remove current desperado edge from $desperado\_non\_result\_edges$
Add current maverick edge to $maverick\_non\_result\_edges$;
Remove current maverick edge from $maverick\_result\_edges$
$bias-- = 2$
If ($bias <= 0$)
Break outer
End If
End If
End for
End for
Else If ($current\_bias < 0$)
// try swapping currently connected desperado edge with an unconnected maverick edge
For all $Desperado\_result\_edges$
For all $maverick\_non\_result\_edges$
If $SWAP(current\_maverick\_edge, current\_desperado\_edge)$
Cut (vertices of $current\_desperado\_edge$)
Link (vertices of $current\_maverick\_edge$)
Add current maverick edge to $maverick\_result\_edges$;
Remove current maverick edge from $maverick\_non\_result\_edges$

Add current desperado edge to *desperado_non_result_edges*
Remove current desperado edge from *desperado_result_edges*
$bias+ = 2$
If $(bias >= 0)$
Break outer
End If
End If
End for
End for
end if

Correctness:

(a) Termination: The algorithm terminates as we have a limited number of contracts and the loops are based upon:

i. The count of contracts where at max we have two loops, which are somehow based upon the contract count and we dont revisit any edge in the same loop again. Hence we can say that the program terminates for this case.

ii. Or on the condition that MST is formed and it is given that the input is such that at least one MST is present for the given graph and hence the program will terminate.

(b) Correctness: We first form the minimum spanning tree using a modified version of Kruskals algorithm and then try to reduce the bias by trying to $SWAP$ edges where the condition that the cycle is not formed in the graph. We try to replace all the possible edges of the company with higher *bias* with the ones currently having lesser contracts till we get a minimum *bias* and hence we know that the final result is correct.

Complexity:
The largest order of computation in the program is when we are trying to compute the swapping operations. In worst case, number of connected edges is $O(n)$ and the edges of the other firm which are not connected of the $O(m)$. Link cut tree is also being used within the loop of $(m * n)$ which adds the complexity of $O(logn)$. Hence the total running time complexity in the worst case will be $O(mnlogn)$.

Program:
Below mentioned is a sample program for the given requirement.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
```

```
public class SolutionLinkCutNew {

   public static int no_of_distribution_centers;

   public static final int _1_MAVERICK = 1;
   public static final int _0_DESPERADO = 0;

   public static final String MAVERICK = "MAVERICK";
   public static final String DESPERADO = "DESPERADO";

   public static BufferedReader br = new BufferedReader(new
       InputStreamReader (System.in));

   public static ArrayList<Edge> maverick_edges = new
       ArrayList<Edge>(250000);
   public static ArrayList<Edge> desperado_edges = new
       ArrayList<Edge>(250000);
   public static ArrayList<Edge> non_res_maverick_edges = new
       ArrayList<Edge>(250000);
   public static ArrayList<Edge> non_res_desperado_edges = new
       ArrayList<Edge>(250000);

   public static void makeGraph(int no_of_contracts) throws IOException
   {
      for (int i = 0; i < no_of_contracts; i++)
      {
         String[] s = br.readLine().split(" ");
         int v = Integer.parseInt(s[0]);
         int u = Integer.parseInt(s[1]);
         String contractor = s[2];

         int contractor_ind;

         if (MAVERICK.equals(contractor))
         {
            contractor_ind = _1_MAVERICK;
            non_res_maverick_edges.add(new Edge(v, u, contractor_ind));
         }
         else
         {
            contractor_ind = _0_DESPERADO;
            non_res_desperado_edges.add(new Edge(v, u, contractor_ind));
         }
      }
   }
```

```
public static int kruskalMod()
{
   int current_bias = 0;

   int rem_edges = no_of_distribution_centers - 1;

   boolean maverick_flag = true;
   boolean desperado_flag = true;

   while (rem_edges > 0)
   {
      if ((current_bias >= 0 || !maverick_flag) &&
         desperado_flag)
      {
         desperado_flag = false;
         for (int i = 0; i < non_res_desperado_edges.size(); i++)
         {
            Edge e = non_res_desperado_edges.get(i);
            if (!areConnected(nodes[e.vertex1], nodes[e.vertex2]))
            {
               link (nodes[e.vertex1], nodes[e.vertex2]);
               desperado_edges.add(e);
               non_res_desperado_edges.remove(i);
               current_bias--;
               rem_edges--;
               desperado_flag = true;
               break;
            }
         }
      }
      else if ((current_bias < 0 || !desperado_flag) &&
            maverick_flag)
      {
         maverick_flag = false;
         for (int i = 0; i < non_res_maverick_edges.size(); i++)
         {
            Edge e = non_res_maverick_edges.get(i);
            if (!areConnected(nodes[e.vertex1], nodes[e.vertex2]))
            {
               link (nodes[e.vertex1], nodes[e.vertex2]);
               maverick_edges.add(e);
               non_res_maverick_edges.remove(i);
               current_bias++;
               rem_edges--;
```

```
            maverick_flag = true;
            break;
         }
      }
   }
}

// Spanning trees constructed

// Remove bias
if ((current_bias > 0))
{
   for (int i = 0; i < maverick_edges.size(); i++)
   {
      Edge e1 = maverick_edges.get(i);
      for (int j = 0; j < non_res_desperado_edges.size(); j++)
      {
         Edge e2 = non_res_desperado_edges.get(j);
         if (SWAP(e1, e2))
         {
            current_bias -= 2;
            if (current_bias <= 0)
            {
               i = maverick_edges.size() + 1;
               break;
            }
         }
      }
   }
}
else if ((current_bias < 0))
{

   for (int i = 0; i < desperado_edges.size(); i++)
   {
      Edge e1 = desperado_edges.get(i);
      for (int j = 0; j < non_res_maverick_edges.size(); j++)
      {
         Edge e2 = non_res_maverick_edges.get(j);
         if (SWAP(e1, e2))
         {
            current_bias += 2;

            if (current_bias >= 0)
            {
```

```
                i = desperado_edges.size() + 1;
                break;
            }
          }
        }
      }
   }

return Math.abs(current_bias);
}

public static boolean SWAP(Edge e1, Edge e2)
{
   // Remove link edge 1
   cut (nodes[e1.vertex1], nodes[e1.vertex2]);

   // Try to add e2
   if (!areConnected(nodes[e2.vertex1], nodes[e2.vertex2]))
   {
      // Add edge 2
      link (nodes[e2.vertex1], nodes[e2.vertex2]);

      if (e1.contractor_indicator == _1_MAVERICK)
      {
         non_res_maverick_edges.add(e1);
         maverick_edges.remove(e1);
      }
      else
      {
         non_res_desperado_edges.add(e1);
         desperado_edges.remove(e1);
      }

      if (e2.contractor_indicator == _1_MAVERICK)
      {
         maverick_edges.add(e2);
         non_res_maverick_edges.remove(e2);
      }
      else
      {
         desperado_edges.add(e2);
         non_res_desperado_edges.remove(e2);
      }
      return true;
   }
```

```
    else
    {
        // If swap fails, add link back
        link (nodes[e1.vertex1], nodes[e1.vertex2]);
    }
    return false;
}

public static class Edge
{
    int vertex1;
    int vertex2;
    int contractor_indicator;

    Edge(int vertex1, int vertex2, int contractor_indicator)
    {
        this.vertex1 = vertex1;
        this.vertex2 = vertex2;
        this.contractor_indicator = contractor_indicator;
    }

    @Override
    public boolean equals(Object obj) {
        Edge e = (Edge) obj;
        return this.vertex1 == e.vertex1 &&
               this.vertex2 == e.vertex2 &&
               this.contractor_indicator == e.contractor_indicator;
    }
}

// Link Cut Tree Data Structure

static Node[] nodes = new Node[100001];

static
{
    for (int i = 0; i < 100001; i++)
    {
        nodes[i] = new Node();
    }
}

public static class Node
{
    Node parent;
```

```
   Node left;
   Node right;
   boolean revert;

   boolean isRoot()
   {
      return (parent == null ||
            (parent.left != this && parent.right != this));
   }

   void push()
   {
      if (revert)
      {
         revert = false;
         Node temp = left;
         left = right;
         right = temp;
         if (left != null)
         {
            left.revert = !left.revert;
         }
         if (right != null)
         {
            right.revert = !right.revert;
         }
      }
   }
}

public static void link(Node v, Node u)
{
   if (areConnected(v, u))
   {
      return;
   }

   makeRoot(v);
   v.parent = u;
}

public static void cut(Node v, Node u)
{
   makeRoot(v);
   expose(u);
```

```
   if (v != u.right ||
      v.left != null ||
      v.right != null)
   {
      return;
   }
   u.right.parent = null;
   u.right = null;
}

public static void rotate(Node n)
{
   Node parent = n.parent;
   Node grand_parent = parent.parent;
   boolean is_root_p = parent.isRoot();
   boolean left_child_n = (n == parent.left);

   // Connecting the edges
   connect((left_child_n ? n.right : n.left), parent, left_child_n);
   connect(parent, n, !left_child_n);
   connect(n, grand_parent, !is_root_p ? parent == grand_parent.left :
       null);
}

public static void connect(Node child, Node parent, Boolean
    is_left_child)
{
   if (child != null)
   {
      child.parent = parent;
   }
   if (is_left_child != null)
   {
      if (is_left_child)
      {
         parent.left = child;
      }
      else
      {
         parent.right = child;
      }
   }
}
```

```
public static void splay(Node n)
{
   while (!n.isRoot())
   {
      Node parent = n.parent;
      Node grand_parent = parent.parent;
      if (!parent.isRoot())
      {
         grand_parent.push();
      }

      parent.push();
      n.push();

      if (!parent.isRoot())
      {
         rotate((parent.left == n) == (parent == grand_parent.left) ?
             parent : n);
      }

      rotate(n);
   }

   n.push();
}

public static Node expose(Node n)
{
   Node last_node = null;
   for (Node m = n; m != null; m = m.parent)
   {
      splay(m);
      m.left = last_node;
      last_node = m;
   }
   splay(n);
   return last_node;
}

public static void makeRoot(Node n)
{
   expose(n);
   n.revert = !n.revert;
}
```

```
   public static boolean areConnected(Node v, Node u)
   {
      if (v == u)
      {
         return true;
      }

      expose(v);
      expose(u);
      return v.parent != null;
   }

   public static void main(String[] args) throws IOException
   {
      String[] s = br.readLine().split(" ");

      no_of_distribution_centers = Integer.parseInt(s[0]);
      int no_of_contracts = Integer.parseInt(s[1]);
      makeGraph(no_of_contracts);
      // Printing Bias
      System.out.println(kruskalMod());

      // Prints the tree
       displayTree();
   }

    // Extra marks portion to display Tree edges

   public static void displayTree() throws IOException
   {
      for (Edge e : maverick_edges)
      {
         System.out.println(e.vertex1 + " " + e.vertex2 + " " + MAVERICK);
      }
      for (Edge e : desperado_edges)
      {
         System.out.println(e.vertex1 + " " + e.vertex2 + " " + DESPERADO);
      }
    }
}
```

Input:

```
7 9
1 2 MAVERICK
1 3 MAVERICK
```

2 3 DESPERADO
2 4 MAVERICK
3 5 MAVERICK
4 5 DESPERADO
4 6 MAVERICK
5 7 MAVERICK
6 7 MAVERICK

Output:

2
1 2 MAVERICK
2 4 MAVERICK
4 6 MAVERICK
5 7 MAVERICK
2 3 DESPERADO
4 5 DESPERADO