Solutions to Homework-1

Sanath Kumar Ramesh, A50054305

12-April-2011

Utpal Kumar, Sanjukta Mitra and I discussed about the homework. The solution write-up is my own.

1 Problem-1

1.1

Let C be a cut that divides the vertex set into S and $V - S$ such that the heaviest edge e is a part of the cut. There are two possibilities

1. C has some more edges in it other than e
2. C has only the edge e ie. e is the only edge in the cut.

Consider the second case. Since e is the only edge in the cut, then any minimum spanning tree of the graph must include this edge e.
Proof: Since e is the only edge in the cut, removal of that edge will result in the graph getting split into two disconnected components. Let us assume that an MST did not include this edge e. Since this is the only edge in the cut, such an MST can span within either one one of the partitions created by the cut but can never span through the whole graph ie. the MST will not include all vertices of the graph. Therefore the obtained tree can never be the spanning tree of the graph. Initially we assumed that our tree is an MST which is a contradiction. Therefore, by contraction, the MST must include this edge e which is the unique heaviest edge in the graph.

The statement in the problem is FALSE.

1.2

The statement given in the problem is FALSE. The Find operation with path compression has the property that when a node's root has already been found, there is a direct path from the node to its root. Because of this feature, Find operation will take constant time. But when a new node/subgraph is inserted into the data structure, for nodes whose path has not been yet compressed, it'll take log^*n steps to find the root. Once the root is found, later queries of the node's parent will happen in constant time. Therefore, the upper bound of the Find operation is log^*n but not the asymptotic tight bound. $\Theta(\log n)$ is not the tight bound of Find operation as it can take constant time for some cases.

1.3

The statement given in the problem is FALSE. It is a well known theorem that the lightest edge in a cut will be a part of an MST. But it is not always true that the same lightest edge will be a part of all MSTs of the graph. This happens when there are two or more edges in the cut each with the same weight as the lightest edge in the cut. In other words, when the lightest edge in the cut is not unique, any of the edges could be chosen arbitrarily for the formation of an MST. Therefore, it is not necessary for one lightest edge in a cut to be a part of all MSTs.

1.4

The statement given in the problem is FALSE. If there is an unique heavy edge in a cycle, then an MST can be formed without this edge. But when there are more than one heaviest edges in a cycle, only one of them can be dropped to form an MST. After removing one of the heaviest edge from a cycle, any attempt to remove the other heaviest edges will disconnect the graph.
Proof: If there are n nodes, then a cycle would require a minimum of n edges. After removing one heaviest edge, there will be n nodes and $n-1$ edges and the graph will still be connected. Such a graph is a tree. Since a tree is the minimally connected graph, removal of any more edges will disconnect the graph. Therefore, removal of any more heaviest edges, will disconnect the graph, making it impossible to form an MST.

1.5

The statement given in the problem is TRUE. Prim's algorithm works by (1) Partitioning the graph using a cut; (2) Selecting the lightest edge across the cut. Forming a cut in the graph is solely dependent on the edge connections between two nodes and not on the weight on the edges. Therefore, a cut can be formed irrespective of the weights on the edges. The lightest edge across a cut is the edge with minimum weight. Since negative numbers have lesser value than positive numbers, a edge with negative weight can be chosen as the lightest edge across the cut. Since both steps of Prim's algorithm are not affected by the negative weights, Prim's algorithm works fine with edges having negative weights.

2 Problem-2

Let us denote the set of equality constraints using M_{eq} and the set of inequality constraints using M_{ineq}. M_{eq} has m_1 constraints and M_{ineq} has m_2 constraints. The idea of this algorithm is that we represent each x_i as a node in a graph and each constraint in M_{eq} as an edge in it. Let $G = (V, E)$ be our undirected graph. $V = x_1, x_2, \ldots, x_n$. For every equality constraint of the form $x_i = x_j$, make an edge in G from x_i to x_j. In order to find if both constraints can be satisfied simultaneously, for each constraint of the form $x_i \neq x_j$ in the set M_{ineq}, find if there is a path from x_i to x_j in the graph G. If there is one such path, then the constraint cannot be satisfied. If there exist no path, then repeat the steps with another constraint from M_{ineq}. Since G was constructed using the equality constraints, a path between two nodes mean that those two nodes are equal in value.

```
(1)  Algorithm check-constraints
(2)  Input: Sets M_eq and M_ineq
(3)  Output: true, if constraints can be satisfied, false, otherwise
(4)  begin
(5)     V = {x_1, x_2, ..., x_n}
(6)     E = {}
(7)     for each (x_i = x_j) ∈ M_eq do
(8)         E = E ∪ (x_i, x_j)
(9)     od
(10)    comment: The graph has been constructed
(12)    for each x_i ≠ x_j ∈ M_ineq do
(13)        if path exists between x_i and x_j then
(14)            return false          // The inequality condition violates equality condition
(15)        fi
(16)    od
(17)    return true                   // No false condition was generated
(18) end
```

Proof of Correctness:
Assuming that the sets M_{eq} and M_{ineq} have constraints that cannot be satisfied. Let there exist a set of constraints $(x_i = x_{p1}), (x_{p1} = x_{p2}), (x_{p2} = x_{p3}), \ldots, (x_{pk} = x_j) \in M_{eq}$. By transitive nature of the

constraints, $(x_i = x_j)$. This means that in the graph G constructed by the algorithm, there will be a path from x_i to x_j. Let us assume that $(x_i \neq x_j)$ is one of the constraints in M_{ineq}. On running the above algorithm, let us assume that the algorithm returned a $true$. Since the algorithm returned a true, it means that $\forall x_i \neq x_j \in M_{ineq}$, there is NO path from x_i to x_j in G. But according to the assumption, there is a path from x_i to x_j in G. Therefore by contradiction, the algorithm cannot return $true$. It must return $false$ under this conditions.

Complexity:

Let this algorithm be implemented using the Union-Find data structure with path compression. The algorithm is rewritten with the data structure as follows:

```
(1)  Algorithm check-constraints
(2)  Input: Sets M_eq and M_ineq
(3)  Output: true, if constraints can be satisfied, false, otherwise
(4)  begin
(5)     V = {x_1, x_2, ..., x_n}
(6)     E = {}
(8)     for all u ∈ V do
(9)        makeset(u)
(10)    od
(12)    for each (x_i = x_j) ∈ M_eq do
(13)       E = E ∪ (x_i, x_j)
(14)       union(x_i, x_j)
(15)    od
(17)    for each x_i ≠ x_j ∈ M_ineq do
(18)       if find(x_i) = find(x_j) then
(19)          return false          // The inequality condition violates equality condition
(20)       fi
(21)    od
(22)    return true                 // No false condition was generated
(23) end
```

The functions for implementing Union-Find data structure is as follows. It is exactly as the ones used for Kruskal's algorithm in the book. Therefore, I'm not explaining them here.

```
(1)  function makeset(x)
(2)  begin
(3)     π(x) = x
(4)     rank(x) = 0
(5)  end
```

```
(1)  function union(x,y)
(2)  begin
(3)     r_x = find(x)
(4)     r_y = find(y)
(5)     if r_x = r_y then
(6)        return
(7)     fi
(8)     if rank(r_x) > rank(r_y) then
(9)        π(r_y) = r_x
(10) else
(11)       π(r_x) = r_y
(12)       if rank(r_x) = rank(r_y) then
```

```
(13)          rank(r_y) = rank(r_y) + 1
(14)       fi
(15)    fi
(16) end
```

```
(1) function find(x)
(2) begin
(3)    while x ≠ π(x) do
(4)          π(x) = find(π(x))
(5)    end
(6)    return π(x)
(7) end
```

For the construction of the graph G, it takes m_1 union operations. Each union operation takes $log(n)$ steps. Therefore the graph construction ie. the first loop in the algorithm takes utmost $m_1 * log(n)$ steps. With path compression, each find operation takes constant time. Therefore, in the second loop, m_2 constraints have to be evaluated with each evaluation taking constant time. Therefore the second loop takes $m_2 * 1$ steps.

Complexity of the algorithm $= O(m_1 * log(n) + m_2)$

3 Problem-3

Each person that Alice knows is represented as a node of a graph. If one person knows another, then an edge is created between the nodes representing those persons. Each node is annotated with the number of people the person representing that node knows. This number is exactly equal to the degree of that node. The algorithm iteratively removes nodes whose degree is less than 5 or nodes that have $(n - degree - of - node) < 5$. The first condition enforces the property that a given person should know 5 or more people. The second condition enforces the property that for a given guest, there should be atleast 5 other guests who they don't know. When nodes are removed, all the incident edges are removed and the scores of neighboring nodes are decreased to reflect the removal of this node. The algorithm proceeds until all nodes in graph satisfy the constraint or there is no more node in the graph.

```
(1) Algorithm party-invite:
(2) Input: A graph G = (V, E) representing the persons and their relationships
(3) Output: A set of nodes representing the persons who can be invited to party
(4) begin
(5)    for v ∈ V such that (score(v) < 5) ∨ ((n − score(v)) < 5) do
(6)       V = V − {v}
(7)       E = E − {edges incident on v}
(9)       for u ∈ neighbors(v) do
(10)         score(u) = score(u) − 1;
(11)      od
(12)   od
(13) end
```

The function $score(v)$ represents the number of neighbors of the node v.

Proof of Correctness: The algorithm terminates after all the nodes obey the condition or when the graph has no more nodes. In both the terminating cases, the output obeys the constraints given in the problem. For a graph with n nodes, each node can have a maximum of $n - 1$ neighbors. Therefore the inner loop will definitely finish in utmost $n - 1$ iterations. The outer loop will definitely terminate in utmost n iterations. Therefore the algorithm definitely terminates and when it terminates it produces the correct output.

Complexity The worst case for this algorithm is when the input is a completely connected graph. For n nodes, each node will have $n - 1$ neighbors. Since in a completely connected graph none of the nodes satisfy

the problem constraint, in each iteration of the outer loop, one node will be removed from the graph. The inner loop runs for $m-1$ iterations if there are m nodes in a graph. Therefore,

Total number of steps taken by whole program $= n*(n-1)+(n-1)*(n-2)+\ldots+2*1+1*0 = O(n^2)$

The algorithm can be implemented using Adjacency Lists for storing the graph. The score of each node is stored in the nodes of the graph.