# CS6150 - Homework/Assignment-4

Arnab Das(u1014840)

November 4, 2016

## 1: Minimum Spanning Tree

**(a)** Let G be weighted directed graph with all edge weights being distinct. **Prove:** G has an unique spanning tree:
**Proof:** Consider G has two spanning trees, $T_1$ and $T_2$. Let $e_1$ be the edge with the minimum weight in the graph that is contained in $T_1$ and not in $T_2$. Then $T_2 \cup e_1$ results in a cycle, say C. Then one of the other edges of C, say $e_2$ is not in $T_1$, else $T_1$ would have had a cycle. Then, if we replace $e_2$ by $e_1$ in $T_2$, that would result in a spanning tree with a smaller weight than $T_2$. Then $T_2$ is not a spanning tree. Hence $T_1$ is a unique spanning tree. (Proved)

**(b)** Even if the weights of the graph are not distinct, it is still possible to get a unique spanning tree. We describe below an algorithm , that is a slight modification of the Kruskal algorithm. Given a weighted graph $G = (V, E, W)$, where V denotes the set of vertices, E denotes the set of edges and W denotes the edge weights. Consider initially a setting of all the nodes without any of the edges, each is a separate tree. Then we gradually build up the minimum spanning tree(mst) by union of these disjoint trees. We always start with the edge of minimal weight at every step, and if the end points of that edge lies in two different trees, then that edge acts as the union of these two trees resulting in a bigger tree. If , the end-points of the edge lie in the same tree, that edge, if added, will contribute to a cycle in that tree. In Kruskal's we ignore this edge. However, we can use this edge to detect non-uniqueness. Once a cycle is detected, then in that tree, we look for another edge that has the same weight as out current edge. If there exists another such an edge, that leads to the conclusion that it is not unique, since the other edge could have been replaced with the current edge, still yielding the minimum spanning tree.
**PseudoCode:**

```
KRUSKAL(G(V,E,W)):
    // Graph represented as triples(u,v,w), where u,v are the edge end-points and w the edge weight w(u,v).
    H=BuildHeap(G(V,E,W))
    foreach u ∈ V
        Make-Set(u)
    T = NULL // This corresponds to the spanning set.
    while (|T| = n-1) // if still spanning
        (u,v,w) = ExtractMin(H)
        Uset = Find(u)
        Vset = Find(v)
        if (Uset ≠ Vset)
            T = T ∪ (u, v)
            Merge(Uset, Vset)
        else // indicates part of same tree, hence contributes to a cycle
            bool x = search(w, Uset) // search for the same weight value in the Uset
            if (x=true) return MST_NOT_UNIQUE
    return MST_UNIQUE
```

**Correctness:** The correctness is same as Kruskal's algorithm. Consider that Kruskal's algorithm gives a minimum spanning tree, M=(V,F), where F is the set of the edges in the MST. Then, if there exists an edge, e, in E-F, such that adding e to the M yields a cycle, C, and if weight(e)=m is the lowest edge weight in $F \cap C$, then we can swap that edge of $F \cap C$ with e, yoielding another spanning tree. This is exactly the part executed by 'search' and hence guaranteed correctness. We searches for same weight edges in the Uset corresponding to the edge of the current edge when a cycle is detected, and if found we say the MST is not unique, else the MST is unique. Thus, we have modified the Kruskal's algorithm to dynamically find the MST uniqueness.

**Running time:** We use the heap structure to store the data, and use the weight of the edges as the keys for creating the heap. The heap creation takes time $O(|E|)$. The Kruskal's algorithm takes $(|E| + |V|)log|V|)$. Additionally , for the search operation, we need to perform a breadth first search on the subtree where a cycle is found. Breadth first search is $O(|E| + |V|)$, and atmost we do that for $(V - 1)$ times. So the overall bound comes to be polynomial.

---

**2: Max Flow Basics . . .**

---

**(a)** Given G is a directed graph with edges having **integer** capacity.
**To Prove:** For any s,t there exists a maximum flow between s,t in which every edge has an integeral flow on it.
**Proof:** The proof can be broken down into two parts. First we show that at every iteration iteration , the flow value on every edge f(u,v) and hence the flow of the augmenting paths are integers, and second that the iteration terminates.
**First:** Goal: At every intermediate step of the Ford-Fulkerson algorithm, the values f(u,v) and the residual capacities are integers.

At iteration 0, this is true by construction of the graph that the given flow capacities of every edge are integer values and the intialized flow capacities are zero. Now, suppose it is true after 'j' iterations. Then, if $G_f$ is the residual graph after the j'th iteration, the bottleneck evaluated in the (j+1) iteration will also be integer since all the residual capacities in $G_f$ and the flow values are integers. Now, since the capacities and the residual capacities are integers, then the flow on every edge must also be an integer as for a forward edge e,Residual Capacity = $c_e$ - f(e), and for a backward edge of e, Residual Capacity = f(e), where $c_e$ is the capacity of the edge e and f(e) is the flow value of edge e in j'th iteration. Then, in the (j+1) iteration, in the new flow over an augmenting path, we either increase f(e) for forward edges by the bottleneck amount which is an integer addition resulting in an integer, and decrease f(e) for backedges by the bottleneck amount which is an integer subtraction resulting in an integer. Thus the new flow and all the flow values on the edges will have integer values in the (j+1) iteration. This also leads to the fact, that the residual capacities we evaluate to build the residual graph will also be integer values since they are evaluated as the difference of the current capacities and the flow values which we proved to be integers. Hence, by induction, at every intermediate step, the flow values and the residual capacities are integers.
**Second:** Goal: The algorithm terminates.
Since, the flow value at every step is increased by the bottleneck amount, which is a positive integer quantity, and it cannot go beyond the out flow from the source, this leads to the fact that the while loop is guaranteed to terminate in atmost C steps, where C is the total outflow from the source.
Combining the **first** and **second** proofs, we can say, that the step at which it terminates, the result would have yielded an integer maximum flow and the flows on every edge will have integer values.

**(b) Case-1: Increase Capacity of an edge by 1**
**Algorithm:** As input, we are given the maximum feasible flow f in G, v(f) the value of f,an edge 'e' whose capacity is to be increased by 1, and we try to find a single augmenting path in $G'$ which contains the edge 'e' whose capacity is increased by 1. We increase the capacity of the edge 'e' in the residual graph and find an augmenting path in the residual graph. If an augmenting path is found, it will have a bottleneck of '1' due to this increased capacity of 'e' and hence the flow value inctreases by 1. If an augmenting path is not found, then the flow value remains the same.

**Correctness:** Since, we have already reached the Maximum flow in G, thus there exists a s-t cut(A,B), such that all the paths crossing the cut are saturated and cannot carry any flow. Note that the maximum flow in $G'$ is atleast v(f), since f is still a valid flow of the network as we haven't decreased the value of any capacity. All the edges across the cut have the same capacity in $G'$ as in G with only exception of 'e'(if and only if e crosses the cut). In that case 'e' can provide a unit path across (A,B) and the flow can increase

atmost by 1, else the flow remains the same. This guarantees the correctness of the algorithm.

**Running time:** We only need to find an augmenting path in the residual graph, which is $O(m+n)$, where m is the number of edges and n is the number of nodes in the network. This is significantly smaller than restarting the maximum flow evaluation from scratch.

**Case-2: Decrease Capacity of an edge by 1**
**Algorithm:** As input, we are given the maximum feasible flow f in G, v(f) the value of f, an edge 'e' whose capacity is to be decreased by 1. If the current capacity of 'e' is strictly greater than its flow value, return the current maxflow. Else,in the residual graph, we decrease the capacity of 'e' by 1 and increase the capacity of the backedge of 'e' by 1. Then attempt to find an augmenting path from sink(t) to the source(s). If an augmenting path is found, return 1 unit of flow from sink to source to balance the reduction in capacity. Next, run the augmenting path subroutine again from source to sink, to find a new augmentng path and recompute the flow. This new flow value is the maximum flow with the the capacity of 'e' reduced.

**Correctness:** If the edge 'e' had not reached maximum capacity, that is, its capacity value was still greater than its flow value when the termination happended, then by decreasing its capacity it might atmost reach its full capacity in which case the current flow still remains valid. So, in that case we can say the maxflow in the modified capacity is still the same v(f). However, if the flow value of 'e' had reached its maximum capacity, then if we reduce its capacity, it means that we have transferred one additional unit from the source to the sink with respect to the new graph of modified capacities. Hence, somehow we need to take out this 1 unit of flow from the sink to the source to balance out and then recompute. Let e=(u,v) and source(s) and sink(t). Note that, since the edge 'e' had reached its capacity(assuming positive, else there would be no meaning of reducing it), that flow must have travelled through from v to t through some path, and hence there might be backedges that can provide a path from t to v that can flow atleast 1 unit. Similarly, since there was a non zero flow across 'e', that flow must have reached 'e' through some path from s to u. Then, there will be some backedges connecting from u to s that can atleast carry one unit of flow. If we can find such augmenting paths from t to v and from u to s, then we can tranfer this one unit of flow on 'e' back using the path t-v-u-s. The unit flow we aim to return can be modeled as an additional unit on the backedge of e, that (v,u). Finding the augmenting path from t to s, exactly serves this purpose. Once , we do this and return the flow back, we have established the network to a situation where the capacity of 'e' is reduced by 1, and the flow would have been v(f)-1. Now we try finding a forward augmenting path from s to t, and if found, that will increase the flow by atmost 1, or will remain the same(since flow never decreases over iterations and cannot exceed v(f), as v(f) was the max flow of a graph with highre capacity). Hence, the max-flow of this graph will be either v(f) or v(f)-1. This, guarantees the correctness of our algorithm.