CSC425 Assignment #2

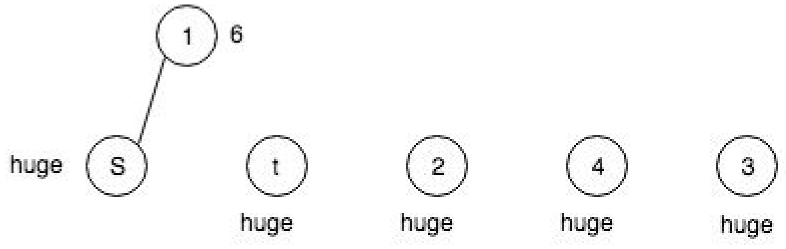
Instructor: Valerie King | Fall Term | UVic

Zhaocheng Li (V00832770)

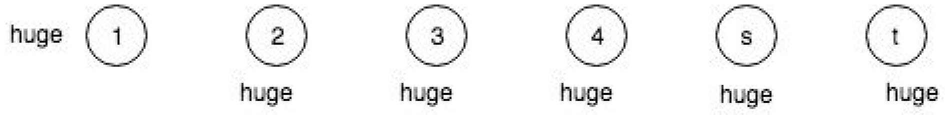
Question 1:

PART(a): from Dinic's algorithm first example in Wikipedia;

- Forest after first flow {s, 1, 3, t} :



- Forest after second flow (s, 1, 4, t):



PART(b):

For the first flow:

Initial :

- maketree(v) for all vertex in G
- addcost(v,huge) for all vertex(root)

advance: link(s,1), link(1,3), link(3,t)

argument: addcost(s,-4)

delete: (1). cut(t), addcost(t,huge) (2). cut(3), addcost(3,huge)

For second flow:

advance: link(1,4), link(4,t)

argument: addcost(s,-6)

delete: (1). cut(t), addcost(t,huge); (2). cut(4), addcost(4,huge);
(3). cut(1), addcost(1,huge)

Question 2:

PART(a):

Prove by contradiction.

Assumption:

(1). $W_J \in T$ for a finite -capacity cut (S, T) of G, and also

(2). $\exists\ I_k \in S, I_k \in R_j$.

But since the capacity of edge (I_K, W_j) is infinite, thus the capacity cut (S, T) is infinite. While in the assumption, the cut, so the assumption contradicts.
Therefore, if $W_j \in T$, $\forall I_k \in R_j \in T$.

PART(b):
We are implementing S and T as disjoint sets, suppose the algorithm is denoted as ***Net-revenue***. Which gives:

```
Net-revenue(cur(S, T), W, I, c, p):
    rev = 0 \\revenue
    for i in range(1,m+1):
        if $W_i$ in T:
            rev = rev + $p_i$
    for i in range(1,n+1):
        if $I_i$ in T:
            rev = rev - $c_i$
    return rev
```

PART(c):
We implement algorithm for it, denoted as ***Packages***. Similarly, in the network G, we implement S and T with disjoint set.

```
Package(G, W, I, s, t, c, p):
    Wc = ∅
    Ic = ∅
    Ford-Fulkerson(G, s, t) // get minimum cut C(S, T)
    rev = Net-revenue(cut(S, T), W, I, c, p) // get net revenue
    for i in range(1,m+1)
        if $W_i$ in T:
            Wc = Wc ∪ {$W_i$}
    for i in range(1, n+1):
        if $I_i$ in T:
            Ic = Ic ∪ {$I_i$}
    return Wc, Ic, rev
```

Analyzing the running time, given running time of Edmonds-Karp as $O(VW^2)$. In this algorithm, $|V| = n+m+2$. $|W|=n+m+r$.
Then its running time is the sum of running time of Edmonds-Karp, running time of Net-revenue, and O(n+m), which is $O((n+m+2)(n+m+r)^{2.})+O(n+m) = O((n+m)(n+m+r)^2)$.

Question 3:
PART(a):

Given the fact that we partition the set of total edges E into two sets E_1, E_2 such that $E=E_1 \cup E_2$. Notice that this is slightly different from what we perform "cut" in the **Blue Rule,** which separates vertices into two sets each time.

Prove; For such graph G=(V, E), MST(G) = MST(MST(V, E_1) $\cup$ MST(V, E_2)).
Proof:

By the hint, we are performing the **Blue Rule** on the graph. Notice that for each subgraph G_1=(V, E_1) and G_2=(V, E_2), the graph is likely to be disconnected due to the incomplete set of edges. In this case for each component of each graph, we are perform the same procedure by the Blue Rule as we do for one connected graph, and thus for disconnected graph, the minimum spanning forest is the set of all MST for each component.

A. In G_1, we perform the Blue Rule for each connected component. In each component, we make a cut to separate V into two set of vertices V_1, V_2. Choose the least-weight, uncolored edge that crosses the cut, and color it blue. Thus we can find the MST of each connected component of G_1. And we can simply inference that:
    a. If the component contains only one vertex, then there is not edge and MST naturally;
    b. If the component contains only one edge for some pair of vertices. Then such edges are certainly in the MST.
    c. If there are more than one edges between a pair of vertices, then the least-weight uncolored edge that crosses the cut will be in MST. It means no matter in G_1 or G_2, we are always ignore the non-least-weight, crossing-cut edges each time after performing a "cut", given the premise that we find a MST connecting all vertices of each connected component.

B. Similarly, we perform the exactly same thing in G_2, as we did in G_1.

Therefore, we can form the minimum spanning tree or forest for G_1 and G_2 respectively.

Notice that for G_1 and G_2, they share the same set of vertices V, but contain the disjoint set of edges, each set of edges is the complement of another.

After concatenating MST(G_1) and MST(G_2) (G'=MST(V, E_1) $\cup$ MST(V, E_2)). No two edges are overlapping. Comparing G' with G, the missed edges in G' are the eliminated edges from either E_1 or E_2 because they failed to form the MSTs since they are not the least-weight, uncolored edges that crosses any "cut".

Thus while we perform Blue Rule and make any cut, the uncolored edges crossing the cut are all "least-weight" edges selected from either G_1 or G_2. Then for arbitrary cur, we can always choose the least-weight edges just like we will do for the whole G. Thus MST(G) = MST(MST(V, E_1) $\cup$ MST(V, E_2)), which MST is minimum spanning tree is G is connected; MST is the minimum spanning forest if G is disconnected.

PART(b)

A full binary tree is a tree in which every node other than the leaves has two children. Then we arbitrary partition the edges E into n set of size <= n so that we very efficiently make use every set of edges, avoiding some extreme cases such as one or two edges in a set (too small) or too many edges in one set in terms of capacity of E. Those cases could possibly lead to waste of space or of time for processing.

Similarly we apply the **Blue Rule** in each set and on each level of the tree. For each set, it contains full V and subset e of E, |e|<=n. We perform Blue Rule on each subset e_i such that there are always the least-weight uncolored edges, which cross random cuts, that are selected as part of MST. Therefore, after each time of cutting, the edges can ONLY be ignored if they are not the least-weight edges among cross-cut edges. And the edges can be finally ruled out of the MST if they are not the least-weight uncolored edges in **any sort of cuts**.

When the MSTs of children are passes up to their parents, each parent node concatenates two children's MSTs such that the new graph contains all vertices and edges from either MST of left child or MST of right child, and no two edges are overlapping. Now we can treat the new graph as the ones with new subsets of edges contained in children node, in terms of the upper level of the tree. We do not need to consider the eliminated edges since they are already not the least-weight uncolored edges crossing any possible cut to connected two partition of vertices. And by the Blue Rule, we consider the least-weight uncolored edge for each cut only.

This iterative will continue and eliminate more unnecessary edges in order to form a MST of the graph. When we arrive at the root. Similarly, each subtree has contributed a MST for their subset of edges. Then the root will eventually find the least-weight edges of any kind of cuts in the whole set of edges, except ones that have been ruled out because of the certain unqualification.

Therefore, the root contains the MST of the whole graph.

PART(c):

In the full binary tree, each node *v* should contain the set of edges it contains in the subtree rooted at *v*. We denote it as ***edges(v) = E_v*.** Thus the running time of looking up a edge takes O(logn), it means we can find the locate this edge in a specific leaf in the logarithmic time. Moreover, we have operations:

1. ***Edges(v)***: return the set of edges it contains for the subtree rooted at v.
2. ***MST-child(v)***: contains the set of edges which form the MSTs of its children nodes. Notice MST-child(leaf) = Edges(leaf).

Give the fact that computing MST from scratch will take O(mlogn), with m edges and n vertices. Each insertion or deletions of edge i consists of:

(1). Search edge i;

- For insertion, we make sure ***i not in Edges(root)***. If i is not in E, for convenience, we add the edges on the left-most leaf and add i to each Edge(v_i) for all v_i along the way. It takes O(logn) to locate the leaf and O(1) for each addition. Totally, in the worst case, Search takes O(logn).
- If i is in Edges(root), insertion terminates.
- For deletion, we do not act if i not in Edges(root); otherwise, we use logarithmic time to locate leaf containing i and delete it from all Edges(v_i) along the way including leaf, we delete it from MST-child(leaf) as well.

(2). Delete or Insert edge i at leaf;

- For insertion, after i not in Edges(root), we add i to each Edge(v_i) for all v_i along the way of search. We add it to both Edges(leaf) and MST-child(leaf) after encountering leaf. It takes O(1) for each addition.
- For deletion, in reverse, we delete i in every Edges(v_i) for each node v_i along the way including Edges(leaf) and MST-child(leaf). It takes O(1) for each deletion.

(3). Dynamic MST generation: Generating new MST with new set of edges: it computing MST from the bottom level of the tree, and update upper MST-child(v) level by level. Since by hint, computing MST from scratch in each tree node takes O(mlogn), where m, n are the number of edges and node contained in that node. The tree node at the same level will generate MST simultaneously. There are n leaves (n sets). So in total, it will take O(nlogn($m_i$logn)), given the fact the constant m_i in each tree node may differ.

Therefore, to sum up, It cost O(nlogn($m_i$logn)+logn)=O(n$\log^2$n)for deletion or insertion.

Question 4:
PART(a):

Recall the BFS traversal to a tree, it traverse the nodes at the same level prior to moving to next level.

For a graph or a tree, the Bellman-Ford algorithm choose a vertex as a root with d=0, denoting it as d_0. Given the property that each node in the graph has label initially showing the distance to the root.

By the algorithm, it send d+1 value to all its neighbors. And all its neighbor receive the value and change their value of label d to y only if y < d (let y be the received value, in this step, y=d_0+1). And iteratively, each neighbor send value y=d+1 to their neighbors again except the node which it receives message from. And so on.

Therefore, suppose we set the root label value as d=0, and other label values of other nodes as infinity. So, starting at root, the Bellman algorithm iteratively sends value d+1 to **ALL its neighbor simultaneously**. We can regard this as a distribution of levels of a tree. Because for a node v, if the value it received is greater than the label value of itself, it will reject and it means the node u which sends this message goes a longer path (more nodes in-between)to the root than node v. Then the edge {u, v} is discarded (avoiding a circle potentially). Thus by changing the label values of each node by Bellman-Ford algorithm. It generates a BFS tree.

PART(b):

Since we observe that during the process of constructing a BFS tree by Bellman-Ford algorithm, we specify “levels” by evaluating the label values of all nodes. So it is possible that the node sends the message to its neighbor(s) at the same level with itself (gets rejected).

In the asynchronous network , in order to terminating Bellman-Ford algorithm right after the BFS is constructed. We can apply convergecast on each node --- the node will stop searching and sending messages when it received message from their neighbors (or potential children on a tree) “echoing” back, either “accept” or “reject” its value y.

In this case, the nodes will terminate level by level and the leaves will terminate as well when it receives all “reject” messages from its neighbors. This method can save the increasing cost significantly, more than a constant value, as the radius of root getting larger as the communication between nodes expands.

Question 5:

D is the distance from the source node to the furthest node.

Moreover, since we need to design an algorithm such that it finds BFS tree in an asynchronous network in $O(D^2)$ time and in $O(nD)$ massages. Thus the algorithm we build BFS “layer by layer” such that it could have the properties:

(1). Starting from the source (or root), each time after nodes sending messages to all their neighbors (front layer), the algorithm forward the information about the front layer nodes(their IDs, etc.) to the source.

(2) Then source computes the BFS edges to connect the nodes in the next layer, and then forward the information about BFS to the nodes in the new front layer, iteratively.

In this case, also since messages can hold up to n IDs, we can conclude that thie algorithm will cost $O(D^2)$ in time and $O(nD)$ messages.