# CS 381 HW 6

Ankur Dhoot

November 5, 2016

## Q1

The basic idea to find the shortest augmenting path in $G_f$ is as follows:

We assume that G has been constructed such that if (u,v) is an edge in G.E, then G.adj[u] contains v and G.adj[v] contains u (i.e u and v are in each others adjacency lists). If not, a simple scan through the edges can ensure this property.

We know that the shortest augmenting path will be found using breadth first search in $G_f$ from the source vertex, s. But what are the edges that exist from any vertex u in $G_f$? Well, if v $\in$ G.adj[u], then either (u,v) $\in$ G.E or (v,u) $\in$ G.E. If (u,v) $\in$ G.E, then the residual capacity from u to v is the capacity of edge (u,v) - the flow along (u,v) = (u,v).c - (u,v).f. Otherwise, (v, u) $\in$ G.E, and the residual capacity from u to v is the flow along (v, u) = (v, u).f. If the residual capacity from u to v is positive, then the edge (u, v) exists in $G_f$ and we can run BFS from vertex v if it has not been visited yet.

The algorithm initializes v.$\pi$ to be NIL for all vertices. Then we run BFS from the source vertex calculating residual capacities in $G_f$ as we go. At the end of the algorithm, if t.$\pi$ is not NIL, we've found a shortest path from s to t in $G_f$, meaning there exists an augmenting path. Moreover, this augmenting path can be retraced following $\pi$ pointers from t to s.

BFS runs in O(V + E) and we've only added constant time modifications to the inner for loop, so our modified version runs in time O(V + E) = O(E) in the flow network context. Thus, we can find the shortest augmenting path in O(E) time.

**Algorithm 1** Find augmenting path if one exists

```
1: function hasAugmentingPath(G, s, t)
2:     for each edge v in G.V do
3:         v.π = NIL
4:     end for
5:     let Q be a new queue
6:     Q.enqueue(s)
7:     while not Q.isEmpty() do
8:         u = Q.dequeue()
9:         for v in G.adj[u] do
10:            residualCapacity = 0
11:            if (u,v) in G.E then
12:                residualCapacity = (u,v).c - (u,v).f
13:            else
14:                residualCapacity = (v,u).f
15:            end if
16:            if residualCapacity > 0 && v.π == NIL  then
17:                v.π = u
18:                Q.enqueue(v)
19:            end if
20:        end for
21:    end while
22:    if t.π != NIL then
23:        return true
24:    end if
25: end function
```

# Q2

Given the network graph G, construct the flow network G' by giving each edge unit capacity. (Consider the network graph G as a directed graph with directed edges (u,v) and (v,u) if (u,v) $\in$ G.E) What is the minimum number of edges we must remove in order to disconnect two vertices, s and t?

Consider any cut (S,T) of the graph G', where S and T are some partition of G', s $\in$ S and t $\in$ T. If we remove all the edges going from S to T in G' (call this graph G"), then we have disconnected s and t. Why? Suppose not. Then there exists some path from s to t in G". At some point, the path must have an edge from a vertex in S to a vertex in T (since t $\in$ T). But all such edges have been deleted, a contradiction.

How many edges are we removing? Since all edges have unit capacity, the number of edges being removed equals the capacity of the cut (S,T).

It's clear then that if we want to find the minimum number of edges we must remove to disconnect s and t, we must compute an st min-cut of G', and the minimum number of edges then equals the capacity of this st min-cut.

And how should we find the capacity an st min-cut in G'? Well, we know that the maximum flow is equivalent to the capacity of an st min-cut, so we must compute the maximum flow. And how should we compute the maximum flow? Well, we can use a particular implementation of Ford-Fulkerson such as the Edmonds-Karp algorithm!

Moreover, if we want the actual edges needed to be removed, we can compute this information as well. Let S be the set of vertices reachable from s in the final residual network of G'. Let T = V - S. Then, as is shown in the book, we have c(S,T) = f(S,T) implying (S, T) is our desired st min-cut, and the edges from S to T are those which must be deleted.

The algorithm follows immediately:

**Algorithm 2** Find minimum number of connectivity edges

```
1: function findConnectivity(G, s, t)
2:     Initialize G'
3:     Run Edmonds-Karp on G' with source s and destination t
4:     return max-flow computed
5: end function
```

The runtime is dominated by the Edmonds-Karp algorithm. Initializing G' trivially takes O(V + E) time. Edmonds-Karp runs in O($VE^2$). Thus, the runtime is dominated by the Edmonds-Karp algorithm and we can compute the minimum number of edges required to disconnect two vertices in O($VE^2$).