CMPS 102: Homework #8

Due on Tuesday, June 2nd, 2015

John Allard 1437547

Problem 1

In the slides an algorithm was given for the following problem: Given a digraph G and a source node s and sink node t, find the maximum number of edge disjoint paths from s to t.
Give an algorithm that finds the maximum number of edge disjoint SIMPLE paths from s to t (i.e. paths with no loops). Hint: Start with the maximum number of disjoint paths from s to t and then delete loops.

Algorithm : This algorithm piggy-backs on the basic algorithm that is used to find the number of edge disjoint paths. It then takes the flow network that is output by the original algorithm, goes over all of the paths and uses an array and a stack to find and remove the cycles from the paths.

```
# Input : Set of paths P = {p1, p2, .., pn } returned
# from standard FF algorithm. p = {s, v, v, .. t} is a list of vertices
# that start at s, end at t, and visits internal vertices along the way
# Output : Set of paths P' = {p1', p2', .., pn'} that have no cycles

find_simple(P)
  while P not empty :
    s = empty stack
    a = empty map
    p = P.extract_path() # grab next path from P
    Q = empty set of paths

    for vertex v in p :
      if a.contains(v) : # if our map says we've seen this vertex
        do : # pop vertices off the stack
          q = s.pop()
        while q != v # until we have removed the loop
      else : # if we haven't seen this vertex yet
        a.insert(v) # insert the vertex into our map
        s.push(v) # push the vertex onto the stack
    q = empty list
    while s.size() : # while vertices in our stack
      q.insert(s.pop()) # put them in the new simple path
    Q.insert(q.reverse()) # put the new path in our set of simple paths
    # note a and s are emptied for the next iteration
  return Q
```

a) Reason that the solution produces the correct maximum number of edge disjoint simple paths.
Answer : In any flow network, the number maximum number of edge-disjoint paths is equal to the number of simple edge disjoint paths, this is because any non-simple edge disoint path can be made simple by removing any loops, and since we are just removing edges from the non-simple path, this process cannot turn an edge-disjoint path into one that is not edge-disjoint, we would have to add edges to the path to do such a thing. Since I am returning the same number of edge-disjoint paths that are given to my algorithm from the Ford-Fulkenson routine, I am returning the correct number of paths. Now I need to reason that the paths I return truly are simple.
To see this, we turn to the algorithm. I take a given not-necessarily simple e.d. path, and walk along all of its vertices. During each walk along the vertices, I use a stack and a standard map that takes a vertex as an argument and tells of if it has already been entered into the map. I also use a stack that has the vertices encountered pushed onto it in the order they are seen, this keeps the most recently seen vertices at the top of the stack. The map is key for finding the cycles, if we come across a vertex that is already in the map, this

Problem 1 continued on next page. . .

means we have already seen it on our path and thus we have hit a cycle. If this is the case, we continually pop vertices off the top of the stack until we get to the vertex we have already seen. This removes the cycle from the path, at which point we continue processing the vertices.
b) What is the running time of your algorithm?

Answer : My algorithm first needs to use the standard algorithm for finding edge-disjoint paths, which when using the Edmonds-Karp algorithm runs in time $O(VE^2)$. Once this is finished, my algorithm then proceeds.
My algorithm consists of one main outer loop that iterates for as many times as there are paths in the graph, lets call this number k. Inside of this outer loop, we have an innter for-loop that has to iterate as many times as there are edges in the given path, which can be upper-bounded by E, the total number of edges in the graph. Inserting and removing from both the stack and map can be done in constant time. Thus, composing the inner and outer-loops, my algorithm runs in time $O(kE)$. This is asymptotically less than the running time of the Ford-Fulkerson algorithm, so the total running time of the algorithms run together is still $O(VE^2)$.

Problem 2

In class we give a flow argument for showing that every k-regular bipartite graph has a perfect matching. In such graphs every women knows k men and every man knows k women.
Give an alternate proof using Hall's Theorem: That is show that for k-regular graphs the following holds: For every subset S of the women, $|N(S)| >= |S|$, where $N(S)$ is the total set of men they know together.

Problem 3

KT 12, p420.
Give a reason why your algorithm reduces the flow as much as possible.

Hint: Use Ford-Fulkerson to find a min cut. Somehow reduce the min cut.

Problem 4

KT 21, p427.
This uses bipartite matching.

Problem 5

5) KT 27, p431. Design a flow network where each driver p_j get flow $Delta_j$.
Then round up the capacities and use the integral flow theorem.

Problem 6

KT 45, p444. Think circulation!