

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.

```

5  # 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
10  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
15              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
20              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
25  # 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, his is because any non-simple edge disjoint 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-Fulkerson 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

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, let's call this number k . Inside of this outer loop, we have an inner 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 woman 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, $|N(S)| \geq |S|$, where $N(S)$ is the total set of men they know together.

Answer : We are given k -regular bipartite graph with vertex set $V = L \cup R$ (left and right vertices, women on the left and men on the right) and edge set E . Note that $|L| = |R|$ necessarily otherwise no perfect matching could exist. I need to show that Hall's theorem holds, aka that for every subset of vertices S in either L or R , if $|N(S)| \geq |S|$. If I show that this is true, then Hall's theorem says there must exist a perfect matching.

To start, choose an arbitrary subset of women $S \subseteq L$. Let E be the number of edges that leave this group, and let $N(S)$ be the vertices in R that are connected to S via these edges. Because we are given a k -regular graph, each vertex $v \in V$ has $k \geq 1$ edges incident to it, and no parallel edges are allowed, so each edge leaving a specific vertex v must end at a unique vertex (no two edges start and end at the same vertex). There are also no self-loops allowed. Thus $|E| \leq k \times |N(S)|$. Also note that $|S| = \frac{|E|}{k}$, because E exactly consists of k edges for each of the vertices in S . Thus, if we divide both sides of the before-given inequality by k , we get :

$$|E|/k \leq |N(S)|$$

$$|S| \leq |N(S)|$$

The last line is what we wished to show, completing the proof.

Problem 3

Consider the following problem. You are given a flow network with unit-capacity edges: It consists of a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$; and $ce = 1$ for every $e \in E$. You are also given a parameter k . The goal is to delete k edges so as to reduce the maximum $s - t$ flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum $s - t$ flow in $G' = (V, E \setminus F)$ is as small as possible subject to this. Give a polynomial-time algorithm to solve this problem.

(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.)

Answer : My algorithm will piggy-back on the typical Ford-Fulkerson algorithm. It will start by running the FF method, then looking at the final residual graph. With this graph, we find all vertices which are reachable from the source vertex s using BFS. We find these vertices because any edges that travels from one of these vertices to a vertex not reachable from s is an edge that will be included in the minimum cut. At this point, we remove k of these edges, which because this is a min-cut (which is equal to the max-flow), will reduce the max-flow by a value of k . This is true because the min-cut is a bottle-neck, there are no other means of directing current around these edges to get to the sink from the source, so every edge that is removed here will reduce the flow by the 1, since each edge in this cut has a flow of one.

Algorithm :

Note, I am assuming I don't have to write out the BFS algorithm or the FF algorithm as we have already had to implement those for previous assignments or previous classes in the case of BFS

```

# input : Flow network G with vertices G.V and edges G.E
# k, the number of edges to remove
# Output : Flow network G' such that the max-flow has been reduced as much as possible
5 shrink_flow(G, k) :
    residual_graph = Ford_Fulkerson(G) # run FF and retrieve the final residual graph
    V_p = BFS(residual_graph, G.s) # V_p is a list of vertices reachable from the source
                                   # vertex in the residual graph
    E_p = empty list # this will hold the min-cut edges
10 for v in V_p : # cycle through all reachable vertices
    for e in v.edges # cycle through all incident edges to the current vertex
        if e.target not in V_p : # if the edge touches a vertex not reachable from source vertex s
            E_p.insert(e)

15 while k >= 0 : # while we haven't removed k edges
    E_p.remove_front() # remove edge from min-cut edges

G_p = Graph(V, E_p) # new graph with same vertices but k edges removed from min-cut
return G_p

```

In a flow-network, the minimum-cut is equal to the maximum-flow. This means that every edge in the min-cut must be completely saturated with flow, and since each edge has a capacity of 1, each flow on the edges in the min-cut must be 1. This means that by removing k edges from the min-cut, we will necessarily have removed k units of flow from the graph. Because we are only allowed to remove k edges, and each edge has at most 1 unit of flow going through it, this is the optimal solution, there is no way to remove more than k units of flow. My algorithm uses the FF method, which can run in time $O(VE^2)$, it then uses BFS, which runs in time $O(V + E)$. It then runs a double for-loop, with the outer loop iterating over all reachable vertices from s , and the inner loop iterating over all edges that emanate from those vertices.

Because a residual graph returned from FF must not contain an $s - t$ path (this is a termination condition for FF), the number of reachable vertices from s is bounded above by $V - 1$. The number of edges incident to these $V - 1$ vertices is at most $E - 1$, since we aren't able to reach vertex t and there must exist at least one edge incident to vertex t . The inside of the inner-loop simply does a comparison and an insert into a list, which is constant time. Thus, by composing the two loops, the rest of my algorithm runs in time $O((V - 1)(E - 1)) = O(VE)$. Combining the FF method, BFS, and the rest of my algorithm gives a running time of $O(VE^2) + O(V + E) + O(VE) = O(VE^2)$, which is indeed polynomial in the number of vertices and edges.

Problem 4

You're helping to organize a class on campus that has decided to give all its students wireless laptops for the semester. Thus there is a collection of n wireless laptops; there is also have a collection of n wireless access points, to which a laptop can connect when it is in range. The laptops are currently scattered across campus; laptop l is within range of a set S_l of access points. We will assume that each laptop is within range of at least one access point (so the sets S_l are nonempty); we will also assume that every access point p has at least one laptop within range of it. To make sure that all the wireless connectivity software is working correctly, you need to try having laptops make contact with access points in such a way that each laptop and each access point is involved in at least one connection. Thus we will say that a test set T is a collection of ordered pairs of the form (l, p) , for a laptop l and access point p , with the properties that

1. if $(l, p) \in T$, then l is within range of p
2. Each laptop appears in at least one ordered-pair in T .
3. Each access point appears in at least one pair in T

This way, by trying out all the connections specified by the pairs in T , we can be sure that each laptop and each access point have correctly functioning software. The problem is: Given the sets S_l for each laptop (i.e., which laptops are within range of which access points), and a number k , decide whether there is a test set of size at most k .

a.) Give an example of an instance of this problem for which there is no test set of size n . (Recall that we assume each laptop is within range of at least one access point, and each access point p has at least one laptop within range of it.)

Take $n = 3$, with the following properties : Laptop 1 is in range of access points 2 and 3. Laptop 2 is only in range of access point 1, and laptop 3 is only in range of access point 1. All access points have a close laptop, and all laptops are within range of an access point. However, since laptops 2 and 3 can only access point 1, the two pairs $(2, 1)$ and $(3, 1)$ must be in the test set. However, since each access point must also be tested, this means that the sets $(1, 2)$ and $(1, 3)$ must be included. This however leads to a test-set of size 4 while our number of laptops and access points (n) is 3.

b.) Give a polynomial-time algorithm that takes the input to an instance of this problem (including the parameter k) and decides whether there is a test set of size at most k .

we have to turn this into a circulation problem (thanks for the hint). We start by labeling all of the laptops as having supplies equal to the number of access points that are in range of the laptop. The

Problem 5

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

Problem 6

KT 45, p444. Think circulation!