Math 182 Homework #7

Nathan Solomon

March 10, 2025

Problem 0.1.

My method is to find a min-cut, such that removing any edge across the cut decreases the maximum flow by 1 and removing any other edge will decrease the maximum flow by 0 or 1.

```
Use Ford-Fulkerson to create a residual graph G
Use BFS to visit every vertex in G that can be reached from the source s
Let MinCut = []
For each edge (u, v) in G:
    If u has been visited but v hasn't or vice versa:
        Append (u, v) to MinCut
If MinCut has at least k elements:
    Remove any k edges in MinCut from G
Otherwise:
    Remove every edge in MinCut from G
    Remove any other edges until you've removed k edges in total
```

The run time of this algorithm is $O(|V| \cdot |E|^2)$, because that's how long Ford-Fulkerson takes, BFS takes $O(|V| + |E|)$ time, the for-loop takes $O(|E|)$ time, and the if-otherwise statement takes $O(k) \subset O(|E|)$ time.

The algorithm is correct because every time you remove an edge, the maximum flow of the network decreases by at most one (since each edge has capacity one), so we need to either decrease the maximum flow by k or decrease the maximum flow to zero. We proved in class that the subgraph A containing all vertices which we mark as "visited" in the algorithm above and its complement B for a minimum cut of G. We also showed that $s \in A$ and $t \in B$, and since every augmenting path connects s to t, removing any edge between A and B will decrease the maximum flow of the graph by one. Once we have found all edges between A and B, we simply need to either remove k of them (to decrease the max flow by k) or remove all of them (to decrease the max flow to zero).

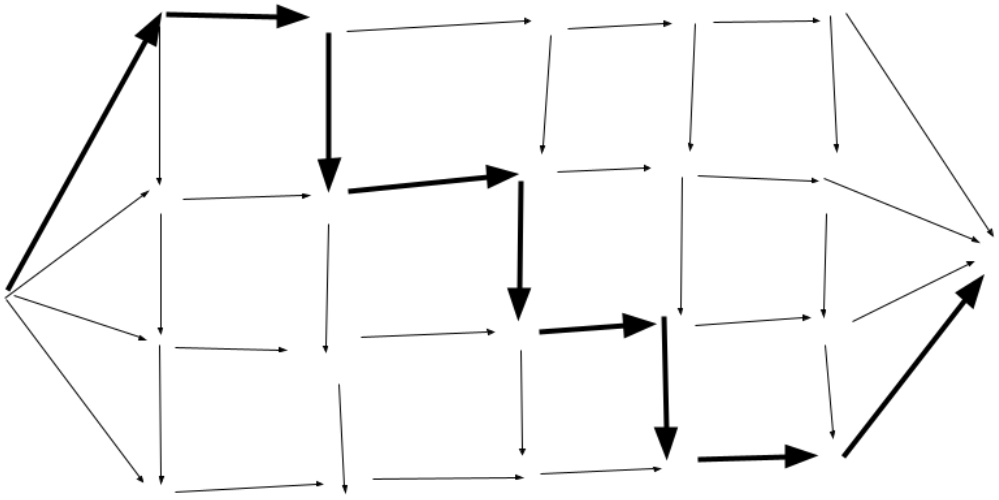
Problem 0.2.

The claim is not true.

Consider an $n \times n$ square grid graph made by taking the Cartesian product of two path graphs which each have n vertices. Connect all n nodes on the left side of the square to the source s, and all nodes on the right side of the triangle to the sink t. Make the edges all have weight 1 and point either to the right or down. The maximum flow of this graph is clearly n, since one unit of flow can go through each row of the grid.

With the greedy algorithm, if the first path found from s to t is a zig-zagging path like the one highlighted in the figure below, there will be no more paths from s to t found aftwerwards, so the max flow found by the greedy algorithm is only 1.

The ratio of max flow found by the greedy algorithm to actual max flow is $1/n$, which can be made arbitrarily small by increasing n.



Problem 0.3.

This is similar to the "circulations with demands" problem from the week 8 lecture notes. My approach will be to create a directed graph from a single source to each base station, then from each base station to the clients within range, then from every cliet to a single sink. By giving each edge the right weight, we can find a max flow network, and the max flow will be the max number of clients that can be connected. Here is some pseudocode:

```
Create a directed weighted graph G with two vertices, s and t, and no edges
For i from 1 to n:
    Add a vertex to G representing the ith client
    Add an edge to G with weight 1 from the ith client to t
For j from 1 to k:
    Add a vertex to G representing the jth base station
    Add an edge to G with weight l from s to the jth base station
    (^ That's a lowercase L, not the number one)
    For i from 1 to n:
        If the ith client is within distance r of the jth base station:
            Add an edge to G with weight 1 from the jth base station to the ith client

Run Edmonds–Karp on G to find the max flow f from s to t
If f is equal to n, return true, otherwise return false
```

The first for-loop runs in $O(n)$ time. The second for-loop runs in $O(nk)$ time because it contains a nested for-loop which takes $O(n)$ time, and the if-statement inside that for-loop runs in $O(1)$ time. Edmonds-Karp runs in $O(|V| \cdot |E|^2)$. The number of vertices in G when Edmonds-Karp is ran will be $n+k$, and the number of edges will be at most $n + nk + k$. I will assume that I do not need to take the magnitude of l into consideration, so the run time for the whole algorithm will be

$$O(n) + O(nk) + O((n+k)(n+nk+k)^2) = O((n+k)n^2k^2).$$

This algorithm is correct because every path from s to t in the max flow network contains exactly one edge from a base station to a client. Therefore there will be no more than l edges going out of each base station, and the number of connections between clients and base stations will be equal to the maximum flow, f. That implies every client can be connected to a base station iff $f = n$.

Problem 0.4.

My approach for this problem will be to take an existing residual graph and update the capacitites, then perform a single step of the Ford-Fulkerson algorithm.

```
Assume we already have the residual graph G=(V,E,s,t,w) from yesterday's flow
Let (u,v) represent the directed edge e
Increase the weight of the edge (v,u) by one
If BFS identifies a path of non-zero weight from s to t in G:
    Decrease the weight of each edge in that path by E
Delete any edges in G with weight zero
```

The runtime of this algorithm is $O(|V| + |E|)$ because that's how long BFS takes. Decreasing the weights of any edges and deleting them if their weights have changed to zero cannot take more than $O(|E|)$ time.

The algorithm is correct because when you increase the capacity of a single edge, either the maximum flow does not change, or the maximum flow increases by one because one more unit of flow can be sent through an augmenting path containing e. If such an augmeting path exists, doing BFS on on the updated residual graph (ignoring the edges with weight zero) will find it, and increment the maximum flow by one.

Alternatively, we could use the fact that when edge capacities are integers, the maximum flow found by Ford-Fulkerson increases by at least one at each step of the algorithm, which we proved in class. We also proved that each step of Ford-Fulkerson takes $O(|V| + |E|)$ time and will increase the maximum flow found if it is possible to do so, which means that it is sufficient for us to note that increasing the weight of one edge in the residual graph will not disturb the existing maximum flow network nor will it increase the maximum flow by more than one.

Homework 7

Math 182, Winter 2025

When designing an algorithm, you must include the following:

1. A short (1-2 sentence) sketch of the main idea of the algorithm.
2. The algorithm itself, written in pseudo-code.
3. The runtime of the algorithm.
4. A proof that the algorithm is correct.
5. A proof that the runtime is correct.

If you wish to sort a list, assume that you can do so in $O(n \log n)$ time; you do not need to write out the details of the sorting algorithm. If you wish to use an algorithm we have discussed in class (eg BFS/DFS, Interval Scheduling, Dijkstra's Algorithm, Ford-Fulkerson, etc) you may do so without elaboration; if you need to modify the algorithm, you should provide sufficient details for the modification in your pseudocode. As mentioned in class, the runtime of the Ford-Fulkerson algorithm depend on implementation. You can use the Edmonds-Karp $O(|V||E|^2)$ bound without justification.

Question 1:

Suppose (G, w) is a network such that $w(e) = 1$ for all edges of G. Let k be a positive integer with $k \leq |E|$. Design an efficient algorithm to select k edges so that when these edges are removed the maximum flow in the new graph is as small as possible.

Question 2:

Recall our initial flawed attempt to write a greedy algorithm to solve the max flow problem:

1. Find an $s - t$ path.
2. Push as much flow through as possible.
3. Reduce the edge-weights along the path by the amount of flow we used, deleting an edge when the weight hits zero.
4. Repeat until there are no more $s - t$ paths.

We showed that this will not always produce an optimal graph. But sometimes greedy algorithms that aren't correct will nonetheless always produce a good approximation.

Prove or disprove the following claim: There is a fixed constant $A > 1$ (independent of the network and the method used to determine the path) so that the greedy algorithm will always produce a flow with value at least 1/A times the actual max-flow value.

Question 3:

Consider a set of computing clients in a town who need to be connected to one of several possible base stations. Clients can only be connected to base stations within a certain distance of them, and the base stations have a maximum capacity.

Formally, we have a list of n clients, with the location of the i-th client given by coordinates (x_i, y_i). Similarly, we have k base stations, with the coordinate of the j-th station given by (x'_j, y'_j). We are also given two parameters: a range parameter r and a load parameter l. Clients must connect only to a single base station, and can only connect to a station if the distance between the client and the base station is at most r. (This is a midwestern town, so there is no elevation changes to worry about; you

can compute the distance using the standard 2D distance formula.) Each base station can connect to at most l clients.

Design an efficient algorithm that takes as input the coordinates of each client and base station and the load and range parameters, and determines whether every client can be connected to a base station.

Question 4:

Suppose you work for the LA department of water and power managing the water mains for the city of LA. Each day, water is drawn from the Lake Mathews reservoir[1] in Riverside and delivered through a network of pipes and pumping stations to a storage tank in Westwood to supply water for UCLA's campus[2]. You are in charge of figuring out how to get as much water as possible from the reservoir to the storage tank each day.

Usually, the amount of water delivered each day is the same as the previous day. However, occasionally a new pipe is added or an old pipe is replaced with a larger pipe and more water can be sent than the previous day. You would like to come up with an algorithm to efficiently update the amount of water that can be delivered in such situations without having to redo the entire calculation from scratch.

You decide to formalize the problem as follows. You are given a network $G = (V, E, s, t, w)$ representing the pipes and pumping stations throughout the city (s represents the reservoir, t the storage tank in Westwood and the edge weights represent the pipe capacities). You are also given a maximum flow f on G (representing the amount of water sent through each pipe yesterday). You need to design an efficient algorithm to solve the following problem: given an edge $e \in E$, find a maximum flow on the network that is formed by increasing the weight of $w(e)$ by 1. For full credit, your solution should run in $\Theta(|V| + |E|)$ time. You may assume that all weights and flow values are integers.

Hint: Use the residual graph for yesterday's flow as a starting point.

Question 5:

Optional: Given an undirected graph $G = (V, E)$ and a set $A \subseteq V$, define the *set of neighbors* of A, written $N(A)$, to be the set of vertices which are the neighbor of at least one vertex in A. In other words,

$$N(A) = \{u \mid \text{for some } v \in A,\ (v, u) \in E\}.$$

Define a perfect matching of G to be a matching which touches every vertex—i.e. a matching $M \subseteq E$ such that for all $v \in V$, there is some $e \in M$ such that v is an endpoint of e.

Prove the following statement: if $G = (V, E)$ is a bipartite graph with bipartition (V_1, V_2) then G has a perfect matching if and only if $|V_1| = |V_2|$ and for every subset $A \subseteq V_1$, $|N(A)| \geq |A|$.

Hint: The forward direction is straightforward. For the other direction, use the max-flow/min-cut theorem.

[1] Fed by the Colorado River Aqueduct, which diverts water from the Colorado river 240 miles across the Mojave desert to supply water to Southern California.

[2] Actually I'm pretty sure the water at UCLA does not come from the Lake Mathews reservoir, but from one of the many other reservoirs in LA.