Mira Final Project: Flow Networks

FLOW NETWORKS

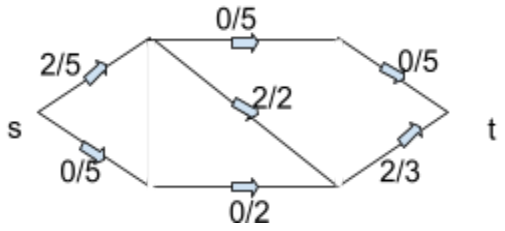
A flow network is a (non-negative) directed weighted graph $G = (V, E)$with a defined source and sink such that no edge leads to the source and no edge leads to the sink. The source $s \in V$and the sink $t \in V$must not have $s = t$. The weight of each edge $e \in E$ is called its capacity $c(e) > 0$. In addition, each edge $e \in E$ has a flow $f(e)$. There are 2 rules a flow network must follow. First, the flow along any edge $e \in E$ cannot exceed its predetermined capacity, that is, $0 \leq f(e) \leq c(e)$. Secondly, for each vertex $v \in V$, the sum of the flows going into the vertex must exactly equate the sum of the flows leaving the vertex. The exception to this second law is the source $s$and the sink$t$.

MAX FLOW PROBLEM

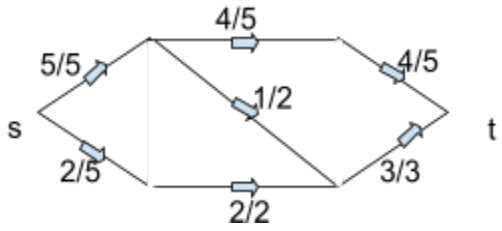
The Max Flow Problem is a well-known problem within the domain of flow networks. It asks the following question: what is the maximal flow that can be sent from source to sink on any given flow network? To reword the problem, let us call $S \subseteq E$the set of edges that go from s to some other $v \in V$, and $T \subseteq E$the set of edges from some other $v \in V$to t. Then, the max flow is the maximal sum of $f(e) \forall e \in S$, which is equivalent to $f(e) \forall e \in T$. This problem can be solved by a numerous number of algorithms, such as Edmonds-Karp, Dinic's Algorithm, and a variety of Push Relabel algorithms. This write-up will discuss the Edmonds-Karp application of Ford-Fulkerson.

FORD FULKERSON

The Ford-Fulkerson algorithm is defined as follows: in each step of the algorithm, we search for an augmenting path from $s$to t. An augmenting path is a path that we can send more flow along. Each edge $e$in the augmenting path must have $c(e) - f(e) > 0$. By Ford-Fulkerson, we choose an augmenting path, send as much flow as possible along it, and repeat until there exist no more augmenting paths from $s$to t. Note that an augmenting path can send flow backwards along an edge. That is, let us define a residual edge as (v, u)given some$(u, v) \in E$. Each edge in Ghas a residual edge, and we can send flow on it in a step of Ford-Fulkerson given the following rule: $f(v, u) \leq f(u, v)$where $(v, u)$is the residual edge. This ensures we do not send negative flow along an edge. Sending flow along residual edges allows us to "undo" an "erroneous step". For example, consider the following situation where $f(u, v)$in the max flow is less than $f(u, v)$currently in the algorithm. Here's an example where we require a residual edge. Notice this is the first augmenting path we send flow along:



Here, we have the 2/3 edge blocking flow for the 0/5 to 0/2 to 2/3 path. The max flow is actually as follows:



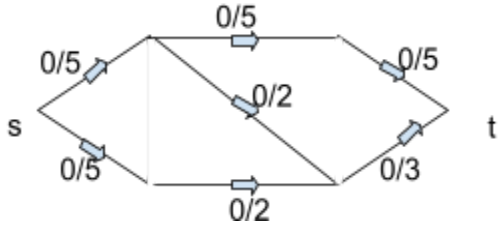
Note the same edge that previously had $f(e) = 2,\ c(e) = 2$has $f(e) = 1$in the max flow. We need to send flow backwards on it using a residual edge at some point.

EDMONDS KARP

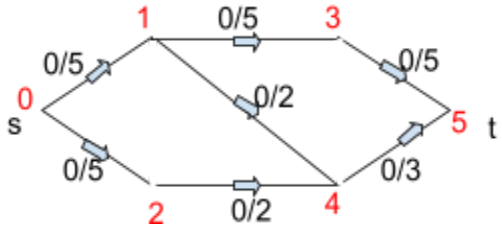
Edmonds-Karp is an implementation of the Ford-Fulkerson Algorithm. The reason Ford-Fulkerson requires an implementation is because it does not specify how to find an augmenting path, so the program might randomly guess using DFS which could lead to a program that does not terminate. Edmonds-Karp is using Ford-Fulkerson by using BFS to find the augmenting paths. This creates a runtime of $O(|E|^2|V|)$.

IMPLEMENTATION DETAILS

When coding Edmonds-Karp, I decided to represent a graph using an edge matrix and an adjacency matrix. Consider the following graph:



We would assign each vertex a number as shown below:



And then, the edge matrix is as follows: each vertex has a row E[v] such that E[v] is a list of all the vertices it is connected to. For example, with our example graph we have

E =[[1,2],
[3,4],
[4],
[5],
[5],

[]]

The capacity matrix is defined as an $|V| \cdot |V|$matrix so C[u,v] has the capacity of (u, v)in the graph, which is 0 if the edge does not exist. In our example,

C=[[0,5,5,0,0,0],
[0,0,0,5,2,0],
[0,0,0,0,2,0],
[0,0,0,0,0,5],
[0,0,0,0,0,3],
[0,0,0,0,0,0]]

We can see, for example, C[0][1] is 5, and we know $c(0, 1) = 5$. The benefit for doing this was that I could have instant access to an edge's capacity and, more importantly, the reverse edge of any edge. Note that the reverse edge is simply the backwards edge of any edge, it need not be a residual edge. For example, if I am working with a residual edge, I also need access to its reverse edge. I could also create a loop for each edge in E, and then have instant access to the capacity, which is very useful in implementation. However, if I had used the graph representation from class, I would have had to create a complicated set-up for reverse edges and accessing any edge would take much more time. The built-in reverse edge access and instant access simplified the more technical details of implementation setup so I could focus on the algorithm.