The Edmonds Karp Algorithm

Hello, welcome to competitive programming! Today we are going to outline the Edmonds Karp algorithm.

Objectives

You objective is to be able to implement the Edmonds Karp algorithm.

Simple Example

Here's a simple example to illustrate the problem of network flow. We start with this graph. The edge weights have two numbers; the first is the capacity of the edge, and the second is the amount of flow that we have assigned it. Node A has only outgoing edges, so it is the source node. Node D has only incoming edges, so it is the sink node. We can assume without loss of generality that there is only one of each.

Now, what we want is to know the total flow that this graph can support.

There are two common algorithms that will get this information for us; they are Ford Fulkerson and Edmonds Karp. We'll get into some details in a minute, but for now know that both algorithms work by selecting a path from the source to the sink that has some flow capacity left.

So let's pick the path A-B-D. (next)

A-B has capacity 10, and B-D has capacity 5. The total capacity of this path is therefore five. So, we perform an operation called augmenting, where we increase

the flow of any forward edges. Augmenting can also do things to backward edges, but we'll talk about that in a minute.

For this path, the result is the flow increases by five for both edges.

Now let's pick another path. Let's try A-C-D. (next)

Again, the minimum edge capacity of this path is 5, so we augment this path by five as well. (next)

You'll notice that source A still has some capacity, and sink D still has some capacity. So we pick another path. This time we pick A-B-C-D. (next)

This has capacity of 5, so we augment the graph. (next)

At this point, we have saturated all our edges, so we are done.

A second example

Now lets talk about reverse edges. Here's another example, the same graph, but the capacities are a bit different.

Suppose we pick path A-B-C-D first. (next)

The minimum edge weight is 5, so we augment the path as before. (next)

Now look at the edge A-C. It has capacity 10, but the path B-D is already saturated, and edge B-C flows in the wrong direction.

The problem is that path A-B-C-D shouldn't have been picked first. It would have been better to pick A-B-D instead. Fortunately, Edmonds Karp has our back. We are also allowed to send flow down a *reverse* edge and cancel out any flow that was there. This allows the algorithm to correct sub-optimal choices.

So, we select A-C-B-D. (next)

To augment this path, we add to the flow of A-C and B-D, and subtract from the flow of B-C. (next)

There. At this point, there are no more paths from A to D that have any leftover capacity. A-C has 5 left over, but B-C and B-D have no more capacity. We are done.

The flow capacity of this graph is 10; note that the sum of the outflow of the source and the sum of the inflow of the sync are both 10.

Let's talk about implementation now. Usually you will hear a term called "residual graph". In that implementation, we model the possible back-flow by making a reverse edge for every forward edge. The forward edges start with the capacity of the weight, and the backward edges start with capacity zero. When you augment a path, you increase the flow of the forward facing edge and decrease the flow of the corresponding backward edge.

The source code in the text uses this method. It is also possible to store the capacity and current flow, like in these diagrams just now. The residual is therefore implicit. The difficulty in that approach is that both vertices need a way of accessing that edge.

Implementation

Here is the textbook implementation of Edmonds Karp. The basic difference between Edmonds Karp and Ford Fulkerson is that Ford Fulkerson used depth first search to find paths, and Edmonds Karp uses breadth first search.

It is beyond the scope of this video to explain why, but it turns out there are some weird cases where Ford Fulkerson will take a very long time to solve certain graphs by picking sub-optimal edges. And the problem setters will be sure to make tests that reveal this behavior just so your problem gets TLE.

Edmonds Karp gets out of that by always picking shorter paths first.

So here's the implementation.

We have some global variables; `res` is the residual graph as before. `mf` is the maximum flow, `f` will hold the value of the minimum edge when a path is augmented. Finally `s` and `t` contain the source and sink nodes.

For augment, we call this from the sink node and have it traverse to the source node by following the parents in the `p` array.

We calculate the minimum edge as we go, and when we hit the source we set `f` to be the final minimum. I am not sure why they do this rather than returning the minimum instead of making the routine void.

Notice how the forward edge of the residual is decreased by the flow, and the backward edge is increased.

Now lets look at the main code.

Implementation, 2

To use the algorithm, you set up residual matrix `res`. Again, the forward edges will start with the capacities, and the backward edges will start with 0.

We initialize `mf` to zero, and then enter our loop.

We initialize `f` to zero, set our distances to infinite, except for the source node. And then we put the source node into the queue.

Next we initialize our parent array to -1.

For the most part this is standard BFS so far.

Now we enter the loop. We pop off the front element into u. If we hit the sink node we are done with the BFS. Otherwise, we go through the `res` array to see if there are any edges that have residual capacity. If we find anything we update the distance array and enqueue the neighbor. Note that looping through the array like this is slow. If you have a lot of edges this could be okay, but as a way of understanding what this is doing, I encourage you to think of a way to speed this up.

Anyway, once we are finished with the BFS, we augment the shortest path we found. If the return value `f` is zero, we are done, otherwise we increase `mf` accordingly and loop again.

That is it for the algorithm. This was a quick rundown, and I am assuming you have seen this before. But I would encourage you to take the example graphs and this code and run the algorithm by hand to see what would happen.

Next
Transcript — Euclid's Algorithms →