INTRODUCTION TO MAXIMUM FLOW ALGORITHMS
By Alei Reyes

PROBLEM STATEMENT

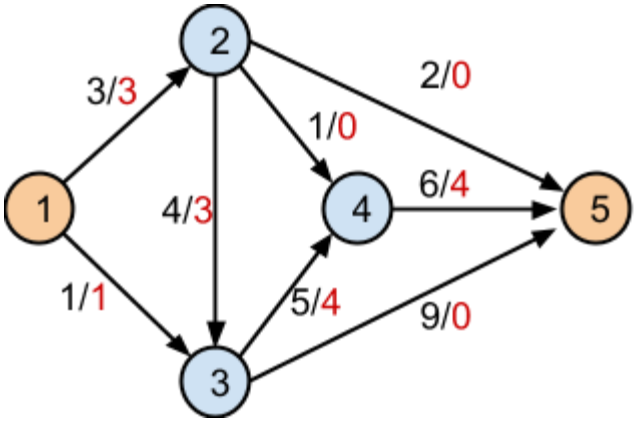
Given a network of pipes we are interested in the maximum volume of water per unit of time that we can deliver from a source to a sink, with the constraint that each pipe supports certain maximum rate of water through it.
So we have a directed graph G(V,E), of |V|=n vertices and |E|=m edges, in which each edge has one number called **capacity,** and we want to assign another number called **flow** less than or equal to the capacity, in such a way that the flow is **conserved**, and that the flow that goes out of one special vertex called **source** is maximum.
The flow that goes inside a vertex u is defined by the sum of the flow of all the edges that ends at u. Similarly, the flow that goes outside u is the sum of the flow of all the edges that starts at u. A flow is conserved when for each vertex different from the source or **sink**, the flow that goes inside it is equal to the flow that goes outside it.
Let's denote the capacity and the flow of an edge (u,v) by c(u,v), and f(u,v) respectively. Then a feasible flow from source **s** to sink **t** must satisfy the following constraints:

$$f(u,v) \leq c(u,v) \ldots (\ capacity\ constraint)$$

$$\sum_{(v,u)\in E} f(v,u) = \sum_{(u,v)\in E} f(u,v)\ ,\ for\ every\ u \in \{s,t\} \ldots (conservation\ constraint)$$

The first constraint tell us that we can't send more flow than the allowed given by the capacity. And the second says that if a vertex receives certain amount of flow, it has to deliver the same amount of flow. We want to maximize the flow that go out of the source: $f = \sum_{(s,u)\in E} f(s,u)$.

The following picture shows a graph with capacities in black, and flows in red. As you can see, the flow that goes outside vertex 1 is 4.



PROBLEM DISSECTION

1. **The structure of a feasible flow**

Probably the easiest way to create a non-zero flow that satisfies the capacity and conservation constraint, is by choosing a path from s to t, and sending a flow equal to the minimum capacity of that path (this is called an augmenting path).
It turns out that any feasible flow can be completely decomposed into simple paths and cycles carrying flow. That is because if there is still a non-zero flow, then there should be an edge with non-zero flow that goes out of s, however that edge is delivering flow to another vertex, and in order to keep flow conservation, that vertex should deliver also a non-zero flow to another vertex, and so on until we reach a previously visited vertex or when we reach t. In the first case, we have found a cycle (cycles does not contribute to the flow, so we can ignore cycles). If we reach t, then we have a simple path that have a non-zero flow, and we can decrease the flow of each edge of that path by an amount equal to the minimum flow in that path (note that after doing this, the flow is still a feasible flow). We can perform this algorithm until there is no edge that goes out of s and have positive flow. At the end there can't be free vertices that are unreachable from s (through a set of non-zero flow edges) and that deliver flow to t, because in that case there would exist nodes that violate the conservation constraint, since they are sending flow without receiving flow.
Note that according to our previous algorithm (let's call it path decomposition algorithm) we are decomposing the flow in at most m paths, since after each augmentation at least one edge is saturated. Another important property is that in our decomposition there exists at least one path that carries a flow of at least floor(f,m) , that is because if all paths are carrying a flow less than floor(f/m), then they could not add up to f.

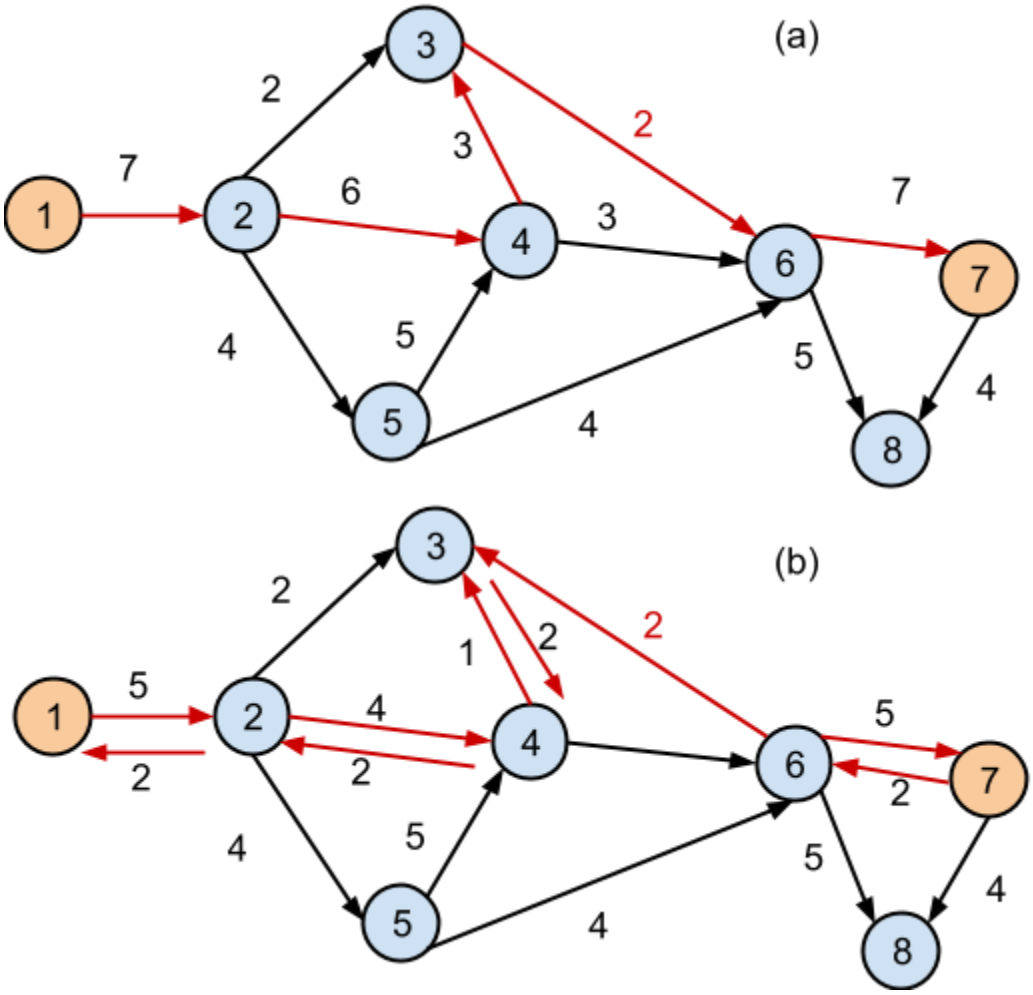
2. Checking if a flow is maximum

Let's suppose that we have already a feasible flow. How can we determine if our flow is the maximum?
First let's find an upper bound for the value of a flow. It's obvious that any flow must be less than or equal to the sum of the capacities of the edges that go outside of the source. In order to generalize the former idea let's define the concept of a **cut**: a s-t cut is a partition of the vertices of a graph into two sets X,Y in such a way that s belongs to X and t belongs to Y. The capacity of a cut is the sum of the capacities of all the edges that starts at X and ends at Y.
We can claim that every flow must be less than or equal to the capacity of an s-t cut. The intuition behind this is that since we can decompose a flow in many paths and cycles carrying flow, then every path that delivers flow to t must cross the cut. Note that this observation also implies that the flow that goes out of s is equal to the flow that goes to t.
Now we know that every flow must be less than or equal to the capacity of any s-t cut. A natural next step would be to calculate the minimum value of a cut (this problem is called min-cut). However we are going to move to another question: if we have a feasible flow, then, can we improve it?
One way of improving a flow is finding an augmenting path from s to t, and sending flow through it. The problem with this idea is that we can increase the flow along an edge, but we can't decrease it (maybe decreasing the flow through one edge and increasing the flow of another we

can get a better flow). That suggests to create another graph that tell us how much flow can we send (adding or removing) through each edge (residual capacity). Let's call this graph the **residual graph**. If the original graph have an edge (u,v), and we send a flow f(u,v) through it, then at most we can send c(u,v) - f(u,v) more flow through (u,v), and we can return back an amount of f(u,v) through (v,u). Let's denote the residual capacity of an edge (u,v) by c'(u,v). Therefore given a feasible flow, we can calculate the residual capacities using the following equations:

c'(u,v)=c(u,v)-f(u,v)

c'(v,u)=f(u,v)



In the above picture, we can observe that after augmenting the red path, in the residual graph new edges are created, and one is removed.

So if we can find an s-t path (with non-zero capacity) in the residual graph, then we can improve the flow. But if we can't find an s-t path, can we be sure that the flow can not be improved (i.e that the flow is maximum)?

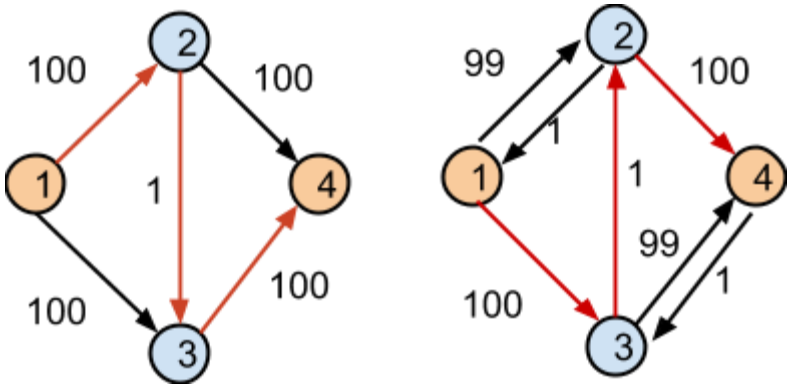
Note that if there is no s-t path in the residual graph, then s and t are disconnected, and that determines an s-t cut, moreover all the edges that cross the cut are saturated by the flow. Since every flow must be less than or equal to any cut, that means that we have found a maximum flow, and also a minimum cut.
The previous argument implies that the **maximum flow is equal to the minimum cut**. Note that this theorem also implies that if the capacities of the graph are integers, then the value of the maximum flow is also an integer.

3. Devising algorithms

We already have an algorithm to calculate the maximum flow:

- ➔ While there is a s-t augmenting path in the residual graph
- ➔ Send a flow equal to the minimum residual capacity of that path

This generic algorithm is called Ford-Fulkerson method. Note that in each step, we are increasing the value of the flow by at least by one unit, so the time complexity of the algorithm is O(m*f) where f is the value of the maximum flow. The following picture shows a graph in which after one augmentation, the flow increases by just one unit.



Our algorithm is exponential because it depends on the value of the flow, moreover its complexity is a bit annoying since it depends on the value of the answer of the problem, and at the start of the algorithm we don't know that answer!

The Ford-Fulkerson method does not specify how to choose a path, maybe we can improve our algorithm choosing an augmenting path in a more clever way.

3.1. Maximum capacity augmenting paths

A natural idea for choosing a path in the Ford-Fulkerson method is by choosing the path with the maximum capacity (the capacity of a path is the minimum capacity of its edges), the intuition is that is better to increase the flow in bigger steps. Remember that according to our path decomposition algorithm, the maximum capacity of a s-t path is at least f/m, so after k augmentations, the value of the flow is $f \cdot (1 - 1/m)^k = O(m \cdot log\, f)$. Now the problem is how to

find a s-t path with maximum capacity, one way is to add the edges of the graph in order from maximum capacity to minimum capacity in a union-find data structure until s and t connects. So the overall complexity is $O(m^2 \cdot log f \cdot log\, m)$. That is a big improvement respect to the raw ford-fulkerson method, because now the algorithm is not exponential, however our complexity still depends on the value of the flow.

3.2. Edmond-Karp algorithm

The idea of the Edmond-Karp algorithm is to choose augmenting paths with the minimum number of edges in the residual graph. There is a dual relationship between the value of the flow, and the distances of the vertices to the source. It turns out that if we chose s-t paths with the minimum number of edges, then after each augmentation the distance from any vertex to the source does not decrease. We can easily prove that in this way:
Suppose that after augmenting a s-t shortest path, exists some vertices whose distance have decreased, let u be the nearest vertex from the source whose distance have decreased, and v the previous vertex in a shortest path (after the augmentation) from s to u. Then the edge (v,u) must be a new edge created by the augmentation, because otherwise the distance from s to v will also have decreased, and v would be closer to s than u. Since (v,u) is a new edge, then the previous augmenting path contained (u,v) as one of its edges. So before the augmentation existed a shortest path of the form $s \rightarrow ... \rightarrow u \rightarrow v \rightarrow ... \rightarrow t$, and after the augmentation there exists a shortest path of the form $s \rightarrow ... \rightarrow v \rightarrow u \rightarrow ... \rightarrow t$. Note that the distance from s to u must be less in the path after the augmentation that in the path before the augmentation, but that implies that the distance from s to v have also decreased, contradicting again our assumption that u is the nearest vertex from the source whose distance have decreased.
So the distance from the vertices to the source after each shortest-path augmentation remains the same or increases. Using the previous argument, it is possible to prove (task for the reader) that we can do at most $(n \cdot m)/2$ augmentations. We can calculate each shortest-augmenting path using a standard bfs, so the complexity of the Edmond-Karp algorithm is $O(n \cdot m^2)$
In despite of the analysis of the Edmond-Karp is a bit tricky, implementing it is very easy:

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int uli;

const int mx=105;
const uli inf=1e15;
int s,t; //s=source, t=sink
vector<int>g[mx];//adjacency list representation of graph

//if we send a flow through (u,v), we must reduce the
//capacity in (u,v) and increase the capacity in (v,u)
//if v=g[u][i], then u=g[v][inv[u][i]]
vector<int>inv[mx];
```

```
vector<uli>w[mx];//the residual capacity
//p helps getting an s-t path after bfs execution
//p[u]=i if g[u][i]=v, and v->u belongs to a shortest s-t path
int p[mx];

//create an edge from u to v with residual capacity c
//and from v to u with capacity 0
void addEdge(int u,int v,uli c){
   int vs=g[v].size();
   int us=g[u].size();
   g[u].push_back(v);
   w[u].push_back(c);
   inv[u].push_back(vs);

   g[v].push_back(u);
   w[v].push_back(0);
   inv[v].push_back(us);
}
//calculates shortest s-t path
bool bfs(){
   queue<int>q;
   q.push(s);
   memset(p,-1,sizeof p);
   while(!q.empty()){
      int u=q.front();
      q.pop();
      if(u==t)return true;
      for(int i=0;i<int(g[u].size());i++){
         int v=g[u][i];
         if(p[v]==-1 && w[u][i]>0){
            p[v]=inv[u][i];
            q.push(v);
         }
      }
   }
   return false;
}
//t<-...<-u<-v<-...<-s
uli augment(int u,uli q){
   if(u==s)return q;
   int iuv=p[u];
   int ivu=inv[u][iuv];
   int v=g[u][iuv];

   q=min(q,w[v][ivu]);
   q=augment(v,q);
```

```
    w[v][ivu]-=q;
    w[u][iuv]+=q;
    return q;
}
uli maxflow(){
    uli f=0;
    while(bfs())f+=augment(t,inf);
    return f;
}
```

RELATED ONLINE JUDGE PROBLEMS

Internet Bandwidth
Monkeys in the Emei Mountain
The Cool Monkeys
Toll