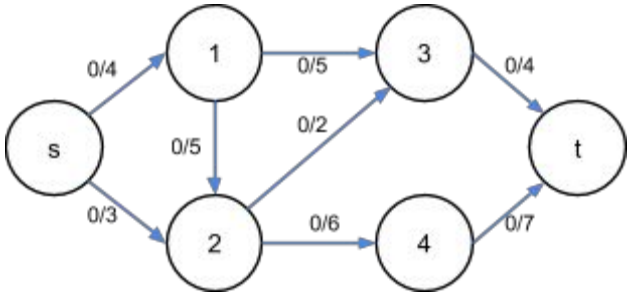
Maximum Flow Algorithms and Applications I

Alexander Shieh
INFOR, Taipei Chien Kuo High School

Maximum flow algorithms has been my recent research interest after reading some aspects of this topic in *Introduction to Algorithms*. These algorithms have a great variety of applications, including scheduling, demand and allocation problems…etc, mainly solving combinatorial optimization problems. In recent years it has been used to solve problems in machine learning and computer vision (which is a field in artificial intelligence that allowed computers to recognize patterns and identify objects in images). Though it was considered the more advanced problems in competitive programming (seldom appear in high school level contests), but the algorithms itself and the problems they aim to solve are fascinating. Here I want to elaborate on the basic concepts of maximum flow algorithms and its applications.

First, before starting to solve the problem, we have to carefully define the problem itself. The rigorous definition and proofs may take a while and sometimes obscure, so here is a simplified version:



Flow network $G = (V, E)$, V denotes the set of vertices and E denotes the set of edges, is a directed graph with each edge $(u, v) \in E$ has a **capacity** $c(u, v)$. There are two vertices that are distinguished from others, the **source** s and **sink** t, and every vertex $v \in V$ exist a path in the flow network that goes $s \to v \to t$.

A **flow** between $(u, v) \in V$, denote as $f(u, v)$ should satisfy $0 \le f(u, v) \le c(u, v)$. And $\forall u \in V - \{s, t\}$, should satisfy $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$, which is called **flow conservation**, the sum of flow came in vertex u should equal to the amount that came out.

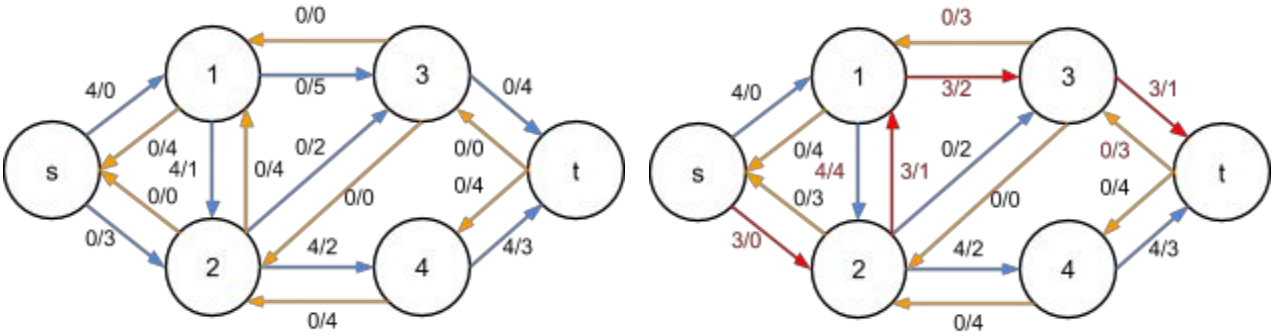
In order to solve the maximum flow problem, we have to introduce the following notions (here we introduce the augmenting flow scheme):

Reverse Edges: For every edge (u, v), we add a reverse edge (v, u) with an initial capacity zero. Every time we add a flow $f(u, v)$, we subtract the capacity of edge (u, v) by $f(u, v)$ and add the capacity of the reverse edge $c(v, u)$ by $f(u, v)$, so we can infer the amount of flow that can be cancelled, it is useful when finding augmenting paths.

Cancellation of Flow: With the reverse edges in place, we can regard the cancellation of the original flow $f(u, v)$ as a flow through the reverse edge (v, u), and of course, the capacity of the original edge will be added back.

Residual Network: The residual network is the original network plus a active flow, denote as G_f.

Augmenting Path: A new path of flow from s to t in the residual network (Includes both original edges and reverse edges). Finding an augmenting path can be done by using depth-first search of breadth-first search.



Left: An example of a residual network, blue edges are the original edges in the network, and orange ones are the reverse edges.
Right: An example of augmenting path of the previous residual network, the red lines and numbers indicates the path and the flow/capacity after augmenting.

In the above example, we can observe the augmenting and cancellation process, if we cancel the flow $1 \rightarrow 2$ by 3, the flow that used to flow into 2 is redirected to 3 and gets to sink t eventually. As for the augmenting flow $s \rightarrow 2$ replaced the flow that used to flow from $1 \rightarrow 2 \rightarrow 4$.

One of the most basic algorithm to solve maximum flow problem is the Ford-Fulkerson Algorithm, first introduced in 1956. The algorithm is simple:

Ford-Fulkerson Algorithm
While there exists a augmentation path in G_f **Do** find augment path f' in G_f and augment $f \leftarrow f+f'$

If we use depth-first search to find augment path, the algorithm has a time complexity of $O(F|E|)$ (F denotes maximum flow. Because $f' \geq 1$, there are at most F augmenting flows, and finding an augmenting flow takes approximately $|E|$ steps), and if implemented using breadth-first search then the complexity is $O(|V||E|^2)$ (usually called the Edmonds-Karp Algorithm), below is my implementation in C++.

Ford-Fulkerson Algorithm Using DFS in C++

```
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;
const int maxV = 101;
struct edge{int to, cap, rev;};
vector<edge> G[maxV];
bool used[maxV];
int V, E, s, t, INF = 2147483647;
void add_edge(int u, int v, int cap){
    G[u].push_back((edge){v, cap, G[v].size()}); //original edge
    G[v].push_back((edge){u, 0, G[u].size()-1}); //reverse edge
}
int dfs(int v, int f){
    if(v == t) return f; // returns if a path from s to t was found
    used[v] = true;
    for(int i = 0; i < int(G[v].size()); ++i){
        edge &e = G[v][i];
        if(!used[e.to] && e.cap > 0){ //selects a residual edge
            int d = dfs(e.to, min(f, e.cap));//search recursively
            if(d > 0){ //if an augment flow was found
                e.cap -= d; //subtract the capacity of this edge
                G[e.to][e.rev].cap += d; //then add to reverse edge
                return d;
            }
        }
    }
    return 0;
}
int max_flow(){
    int flow = 0;
    while(true){ //search until no more paths is found
        memset(used, 0, sizeof(used));
        int f = dfs(s, INF);
        if(f == 0) return flow;
        flow += f;
    }
}
```

References:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*.
2. 秋葉 拓哉, 岩田 陽一, 北川 宜稔. *プログラミングコンテストチャレンジブック*