1 Max-flow min-cut theorem

From Wikipedia, the free encyclopedia (October 2009)

In optimization theory, the **max-flow min-cut theorem** states that in a flow network, the maximum amount of flow passing from the *source* to the *sink* is equal to the minimum capacity that needs to be removed from the network so that no flow can pass from the source to the sink.

The **max-flow min-cut theorem** was proved by P. Elias, A. Feinstein, and C.E. Shannon in 1956, and independently also by L.R. Ford, Jr. and D.R. Fulkerson in the same year.

2 Definition

Let $N = (V, E)$ be a network with s and t being the source and the sink of N respectively.

The **capacity** of an edge is a mapping c: $E \rightarrow R^+$, denoted by c_{uv} or $c(u,v)$. It represents the maximum amount of flow that can pass through an edge.

A **flow** is a mapping f: $E \rightarrow R^+$, denoted by f_{uv} or $f(u,v)$, subject to the following two constraints:

1. $f_{uv} \leq c_{uv}$, for each $(u,v) \in E$ (capacity constraint)

2. $\Sigma_{v:(u,v) \in E} f_{uv} = \Sigma_{v:(v,u) \in E} f_{vu}$, for each $v \in V \setminus \{s,t\}$ (conservation of flows).

The **value of flow** is defined by $|f| = \Sigma_{v \in V} f_{sv}$, where s is the source of N. It represents the amount of flow passing from the source to the sink.

The *maximum flow problem* is to maximize $|f|$, that is, to route as much flow as possible from s to the t.

An **s-t cut** $C = (S,T)$ is a partition of V such that $s \in S$ and $t \in T$. The **cut-set** of C is the set $\{(u,v) \in E \mid u \in S, v \in T\}$. Note that if the edges in the cut-set of C are removed, $|f| = 0$.
The **capacity** of an *s-t cut* is defined by $c(S,T) = \Sigma_{(u,v) \in (S,T)} c_{uv}$.

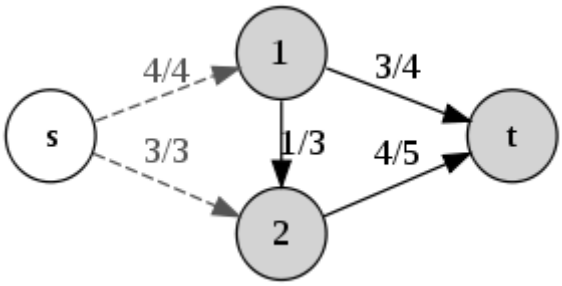
The *minimum cut problem* is to minimize $c(S,T)$, that is, to minimize the amount of capacity of an *s-t cut*.

3 Statement

The max-flow min-cut theorem states

The maximum value of an s-t flow is equal to the minimum capacity of an s-t cut.

4 Example



A network with the value of flow equal to the capacity of an s-t cut

The figure on the top is a network having a value of flow of 7. The vertex in white and the vertices in grey form the subsets *S* and *T* of an s-t cut, whose cut-set contains the dashed edges. Since the capacity of the s-t cut is 7, which is equal to the value of flow, the max-flow min-cut theorem tells us that the value of flow and the capacity of the s-t cut are both optimal in this network.

5 Ford–Fulkerson algorithm

The **Ford–Fulkerson algorithm** (named for L. R. Ford, Jr. and D. R. Fulkerson) computes the maximum flow in a flow network. It was published in 1956. The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson.

The idea behind the algorithm is very simple: As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

5.1 Algorithm

Given a graph $G(V,E)$, with capacity $c(u,v)$ and flow $f(u,v) = 0$ for the edge from u to v. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

- $f(u,v) \leq c(u,v)$
  The flow from u to v does not exceed the capacity.

- $f(u,v) = -f(v,u)$
  Maintain the net flow between u and v. If in reality a units are going from u to v, and b units from v to u, maintain $f(u,v) = a - b$ and $f(v,u) = b - a$.

- $\sum_v f(u,v) = 0 \iff f_{\text{in}}(u) = f_{\text{out}}(u)$
  for all nodes u, except s and t. The amount of flow into a node equals the flow out of the node.

This means that the flow through the network is a *legal flow* after each round in the algorithm. We define the **residual network** $G_f(V,E_f)$ to be the network with capacity $c_f(u,v) = c(u,v) - f(u,v)$ and no flow. Notice that it

can happen that a flow from v to u is allowed in the residual network, though disallowed in the original network: if $f(u,v) > 0$ and $c(v,u) = 0$ then $c_f(v,u) > 0$.

Algorithm Ford–Fulkerson

Inputs Graph G with flow capacity c, a source node s, and a sink node t

Output A flow f from s to t which is a maximum

1. $f(u,v) \leftarrow 0$ for all edges (u,v)
2. While there is a path p from s to t in G_f, such that $c_f(u,v) > 0$ for all edges $(u,v) \in p$:
    1. Find $c_f(p) = \min\{c_f(u,v) \mid (u,v) \in p\}$
    2. For each edge $(u,v) \in p$
        1. $f(u,v) \leftarrow f(u,v) + c_f(p)$ (*Send flow along the path*)
        2. $f(v,u) \leftarrow f(v,u) - c_f(p)$ (*The flow might be "returned" later*)

The path in step 2 can be found with for example a breadth-first search or a depth-first search in $G_f(V,E_f)$. If you use the former, the algorithm is called Edmonds–Karp.

When no more paths in step 2 can be found, s will not be able to reach t in the residual network. If S is the set of nodes reachable by s in the residual network, then the total capacity in the original network of edges from S to the remainder of V is on the one hand equal to the total flow we found from s to t, and on the other hand serves as an upper bound for all such flows. This proves that the flow we found is maximal.

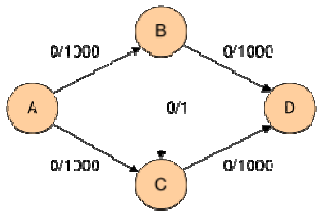
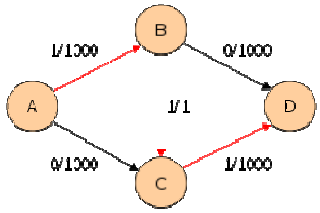
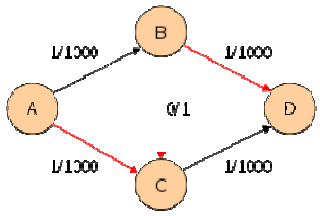
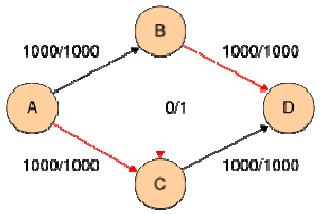
5.2 Complexity

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values. When the capacities are integers, the runtime of Ford-Fulkerson is bounded by O ($E*f$), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in O (E) time and increases the flow by an integer amount which is at least 1.

A variation of the Ford–Fulkerson algorithm with guaranteed termination and a runtime independent of the maximum flow value is the Edmonds–Karp algorithm, which runs in O (VE^2) time.

5.3 Example

The following example shows the first steps of Ford–Fulkerson in a flow network with 4 nodes, source A and sink D. This example shows the worst-case behaviour of the algorithm. In each step, only a flow of 1 is sent across the network. If you used breadth-first-search instead, you would only need two steps.

Path	**Capacity**	**Resulting flow network**
Initial flow network		
A,B,C,D	$\min(c_f(A,B),c_f(B,C),c_f(C,D)) =$ $\min(c(A,B)-f(A,B),c(B,C)-f(B,C),c(C,D)-f(C,D)) =$ $\min(1000-0,1-0,1000-0) = 1$	
A,C,B,D	$\min(c_f(A,C),c_f(C,B),c_f(B,D)) =$ $\min(c(A,C)-f(A,C),c(C,B)-f(C,B),c(B,D)-f(B,D)) =$ $\min(1000-0,0-(-1),1000-0) = 1$	
After 1998 more steps …		
Final flow network		

Notice how flow is "pushed back" from C to B when finding the path A,C,B,D.

5.4 Python implementation (using depth first search)

```
class FlowNetwork(object):
    def __init__(self):
        self.adj, self.flow, = {},{}

    def add_vertex(self, vertex):
        self.adj[vertex] = []

    def get_edges(self, v):
        return self.adj[v]

    def add_edge(self, u,v,w=0):
        self.adj[u].append((v,w))
        self.adj[v].append((u,0))
        self.flow[(u,v)] = self.flow[(v,u)] = 0

    def find_path(self, source, sink, path):
        if source == sink:
            return path
        for vertex, capacity in self.get_edges(source):
            residual = capacity - self.flow[(source,vertex)]
            edge = (source,vertex,residual)
            if residual > 0 and not edge in path:
                result = self.find_path(vertex, sink, path + [edge])
                if result != None:
                    return result

    def max_flow(self, source, sink):
        path = self.find_path(source, sink, [])
        while path != None:
            flow = min(r for u,v,r in path)
            for u,v,_ in path:
                self.flow[(u,v)] += flow
                self.flow[(v,u)] -= flow
            path = self.find_path(source, sink, [])
        return sum(self.flow[(source, vertex)] for vertex, capacity in
self.get_edges(source))
```

Usage example

For the example flow network in maximum flow problem we do:

```
g=FlowNetwork()
map(g.add_vertex, ['s','o','p','q','r','t'])
g.add_edge('s','o',3)
g.add_edge('s','p',3)
g.add_edge('o','p',2)
g.add_edge('o','q',3)
g.add_edge('p','r',2)
g.add_edge('r','t',3)
g.add_edge('q','r',4)
g.add_edge('q','t',2)
print g.max_flow('s','t')
Output: 5
```

6 Edmonds–Karp algorithm

In computer science and graph theory, the **Edmonds–Karp algorithm** is an implementation of the Ford–Fulkerson method for computing the maximum flow in a flow network in $\mathcal{O}(|V| \cdot |E|^2)$. The algorithm was first published by a Russian scientist, Dinic, in 1970, and independently by Jack Edmonds and Richard Karp in 1972 (discovered earlier). Dinic's algorithm includes additional techniques that reduce the running time to $\mathcal{O}(|V|^2 \cdot |E|)$.

6.1 Algorithm

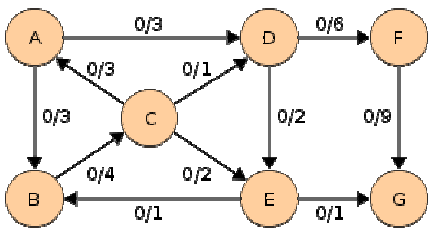
The algorithm is identical to the Ford–Fulkerson algorithm, except that the search order when finding the augmenting path is defined. The path found must be the shortest path which has available capacity. This can be found by a breadth-first search, as we let edges have unit length. The running time of $\mathcal{O}(|V| \cdot |E|^2)$ is found by showing that each augmenting path can be found in $\mathcal{O}(|E|)$ time, that every time at least one of the E edges becomes saturated, that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the distance is at most V long. Another property of this algorithm is that the length of the shortest augmenting path increases monotonically.

6.2 C Implementation

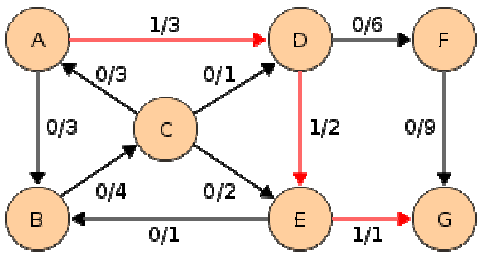
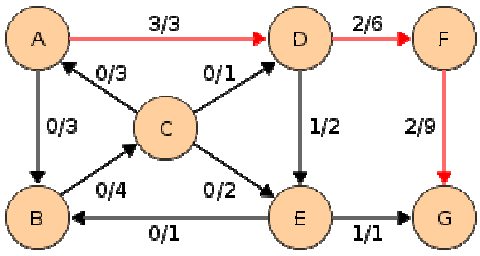
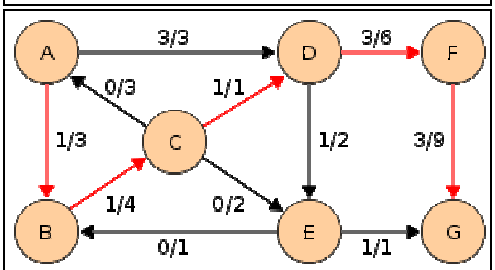
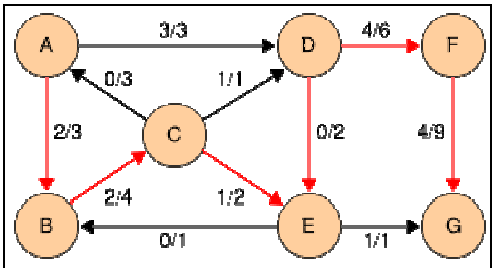
```
//Edmonds-Karp
//return the largest flow;flow[] will record every edge's flow
//n, the number of nodes in the graph;cap, the capacity
//O(VE^2)
#define N 100
#define inf 0x3f3f3f3f
int Edmonds_Karp(int n,int cap[][N],int source,int sink){
     int flow[N][N];
     int pre[N],que[N],d[N],p,q,t,i,j;
     if (source==sink) return inf;
     memset(flow,0,sizeof(flow));
     while (true){
             memset(pre,-1,sizeof(pre));
            d[source]=inf;p=q=0,que[q++]=source;
            while(p<q&&pre[sink]<0){
                   t=que[p++];
                   for (i=0;i<n;i++)
                             if (pre[i]<0&&(j=cap[t][i]-flow[t][i]))
                                  pre[que[q++]=i]=t,d[i]=min(d[t],j);
            }
            if (pre[sink]<0) break;
            for (i=sink;i!=source;i=pre[i])
                          flow[pre[i]][i]+=d[sink],flow[i][pre[i]]-=d[sink];
     }
     for (j=i=0;i<n;j+=flow[source][i++]);
        return j;
}
```

6.3 Example

Given a network of seven nodes, source A, sink G, and capacities as shown below:



In the pairs f / c written on the edges, f is the current flow, and c is the capacity. The residual capacity from u to v is $c_f(u,v) = c(u,v) - f(u,v)$, the total capacity, minus the flow that is already used. If the net flow from u to v is negative, it *contributes* to the residual capacity.

Capacity	**Path** **Resulting network**
$\min(c_f(A,D),c_f(D,E),c_f(E,G)) =$ $\min(3 - 0,2 - 0,1 - 0) =$ $\min(3,2,1) = 1$	*A,D,E,G* 
$\min(c_f(A,D),c_f(D,F),c_f(F,G)) =$ $\min(3 - 1,6 - 0,9 - 0) =$ $\min(2,6,9) = 2$	*A,D,F,G* 
$\min(c_f(A,B),c_f(B,C),c_f(C,D),c_f(D,F),c_f(F,G)) =$ $\min(3 - 0,4 - 0,1 - 0,6 - 2,9 - 2) =$ $\min(3,4,1,4,7) = 1$	*A,B,C,D,F,G* 
$\min(c_f(A,B),c_f(B,C),c_f(C,E),c_f(E,D),c_f(D,F),c_f(F,G)) =$ $\min(3 - 1,4 - 1,2 - 0,0 - - 1,6 - 3,9 - 3) =$ $\min(2,3,2,1,3,6) = 1$	*A,B,C,E,D,F,G* 

Notice how the length of the augmenting path found by the algorithm (in red) never decreases. The paths found are the shortest possible. The flow found is equal to the capacity across the minimum cut in the graph separating the source and the sink. There is only one minimal cut in this graph, partitioning the nodes into the sets $\{A,B,C,E\}$ and $\{D,F,G\}$, with the capacity $c(A,D) + c(C,D) + c(E,G) = 3 + 1 + 1 = 5$.