CS 445

Flow Networks

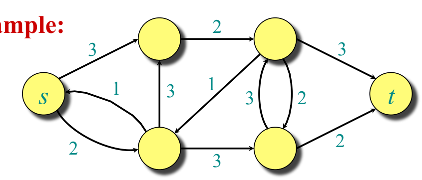
Alon Efrat

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

Flow networks

Definition. A ***flow network*** is a directed graph $G = (V, E)$ with two distinguished vertices: a ***source*** s and a ***sink*** t. Each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v)$. If $(u, v) \notin E$, then $c(u, v) = 0$.

Example:



Flow networks

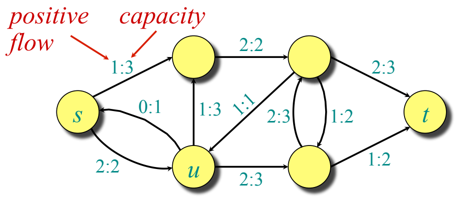
Definition. A ***positive flow*** on G is a function $p : V \times V \to \mathbb{R}$ satisfying the following:

- ***Capacity constraint:*** For all $u, v \in V$,
  $$0 \le p(u, v) \le c(u, v).$$
- ***Flow conservation:*** For all $u \in V - \{s, t\}$,
  $$\sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u) = 0.$$

The ***value*** of a flow is the net flow out of the source:

$$\sum_{v \in V} p(s,v) - \sum_{v \in V} p(v,s).$$

A flow on a network

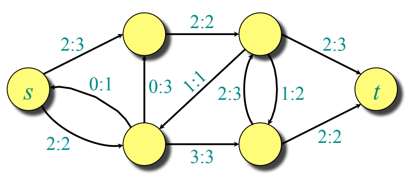


Flow conservation
- Flow into u is $2 + 1 = 3$.
- Flow out of u is $0 + 1 + 2 = 3$.

The value of this flow is $1 - 0 + 2 = 3$.

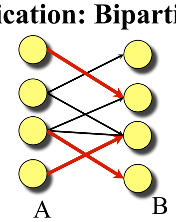
The maximum-flow problem

Maximum-flow problem: Given a flow network G, find a flow of maximum value on G.



The value of the maximum flow is 4.

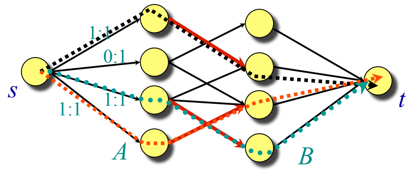
Application: Bipartite Matching.



A graph $G(V,E)$ is called **bipartite** if V can be partitioned into two sets $V=A\cup B$, and each edge of E connects a vertex of A to a vertex of B.

A **matching** is a set of edges M of E, where each vertex of A is adjacent to at most one vertex of *B, and vice versa.*

Matching and flow problem



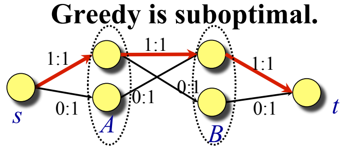
Add a vertex *s*, and connect it to each vertex of *A*.
Add a vertex *t*, and connect each vertex of *B* to *t*.
The capacity of all edges is 1.

Find max flow. Assume it is an **integer** flow, so the flow of each edge is either 0 or 1.

Each edge of *G* that carries flow is in the matching.
Each edge of *G* that **does not** carry flow is **not in** the matching.

Claim: The edge between *A* and *B* that carry flow form a matching.

Greedy is suboptimal.



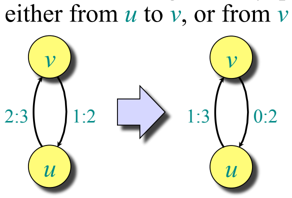
Assume we have already some edges in a (partial) matching *M*.

In order to increase the cardinality of the matching we might need to first remove from *M* some edges (somehow counterintuitive ?)

Thinking again about the matching as flow problem, it means that we might need to remove flow from edges that currently curry flow.

Flow cancellation

Without loss of generality, positive flow goes either from *u* to *v*, or from *v* to *u*, but not both.



Net flow from *u* to *v* in both cases is 1.

The capacity constraint and flow conservation are preserved by this transformation.

A notational simplification

IDEA: Work with the net flow between two vertices, rather than with the positive flow.

Definition. A ***(net) flow*** on G is a function $f : V \times V \to \mathbb{R}$ satisfying the following:

- ***Capacity constraint:*** For all $u, v \in V$,
$$f(u, v) \le c(u, v).$$
- ***Flow conservation:*** For all $u \in V - \{s, t\}$,
$$\sum_{v \in V} f(u,v) = 0.$$
- ***Skew symmetry:*** For all $u, v \in V$,
$$f(u, v) = -f(v, u).$$

Equivalence of definitions

Net flow *vs.* positive flow.

Theorem. The two definitions are equivalent.

Proof. ($\Rightarrow$) Let $f(u, v) = p(u, v) - p(v, u)$.

- ***Capacity constraint:*** Since $p(u, v) \le c(u, v)$ and $p(v, u) \ge 0$, we have $f(u, v) \le c(u, v)$.
- ***Flow conservation:***
$$\sum_{v \in V} f(u,v) = \sum_{v \in V} \big(p(u,v) - p(v,u)\big) = \sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u)$$

- In particular, if $u \in V - \{s, t\}$, then $\sum_{v \in V} f(u,v) = 0$

- ***Skew symmetry:***
$$\begin{aligned} f(u, v) &= p(u, v) - p(v, u) \\ &= -\big(p(v, u) - p(u, v)\big) \\ &= -f(v, u). \end{aligned}$$

Proof (continued)

Obtaining the positive flow from the net flow

($\Leftarrow$) Define
$$p(u, v) = \begin{cases} f(u, v) & \text{if } f(u, v) > 0, \\ 0 & \text{if } f(u, v) \le 0. \end{cases}$$

- ***Capacity constraint:*** By definition, $p(u, v) \ge 0$. Since $f(u, v) \le c(u, v)$, it follows that $p(u, v) \le c(u, v)$.
- ***Flow conservation:*** If $f(u, v) > 0$, then $f(v, u) < 0$ so $p(v,u)=0$.
  $p(u, v) - p(v, u) = f(u, v)$.
  If $f(u, v) \le 0$, then
  $p(u, v) - p(v, u) = -f(v, u) = f(u, v)$ by skew symmetry.
  Therefore,

$$\sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u) = \sum_{v \in V} f(u,v)$$

Residual network

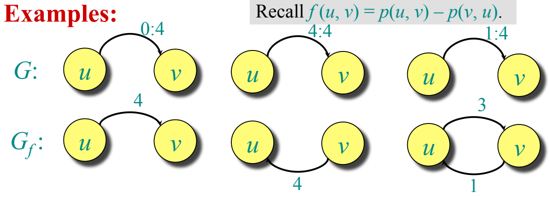
Definition. Let f be a flow on $G = (V, E)$.

residual capacities $c_f(u, v) = c(u, v) - f(u, v)$

The ***residual network*** – $G_f(V, E_f)$, and, E_f are all edges whose residual capacity **>0**

Examples:

Recall $f(u, v) = p(u, v) - p(v, u)$.



Lemma. $|E| \le |E_f| \le 2|E|$.

Augmenting paths

•**Definition.** Any path from s to t in G_f is an ***augmenting path*** in G with respect to f. It has a **bottleneck** $c_f(p):=\min\{ c_f(u,v) \mid (u,v) \text{ on } p \}$

•The flow value can be **increased** along an augmenting path p by adding $c_f(p)$ to the net flow of each edge along p.

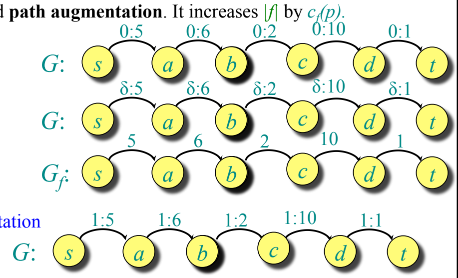
•This is called **path augmentation**. It increases $|f|$ by $c_f(p)$.

Start with δ=0, slowly increase

when δ=0,

$c_f(p) = 1$

After augmentation



Augmenting paths-more accurately

•**Definition.** Any path p from s to t in G_f is an ***augmenting path*** in G with respect to f.

•$c_f(p):=\min\{ c_f(u,v) \mid (u,v) \in p \}$

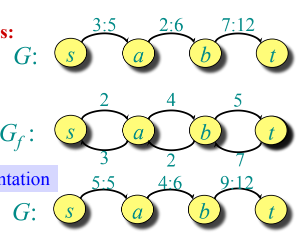
• $\forall (u,v) \in p$ set $f(u,v) \mathrel{+}= c_f(p)$; $f(v,u) \mathrel{-}= c_f(p)$

• This is called **path augmentation**.

Examples:

$c_f(p) = 2$

After augmentation



Note – flow conservation is preserved.

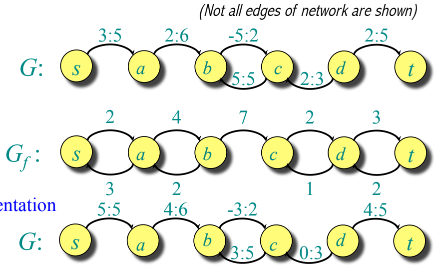
Augmenting paths – Another example

- **Definition.** Any path p from s to t in G_f is an ***augmenting path*** in G with respect to f.
- $c_f(p) := \min\{\, c_f(u,v) \mid (u,v) \in p \,\}$
- $\forall\ (u,v) \in p$ set $f(u,v) \mathrel{+}= c_f(p)$; $f(v,u) \mathrel{-}= c_f(p)$
- This is called **path augmentation**.

(Not all edges of network are shown)

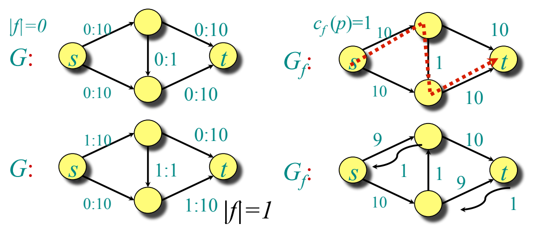
$c_f(p) = 2$

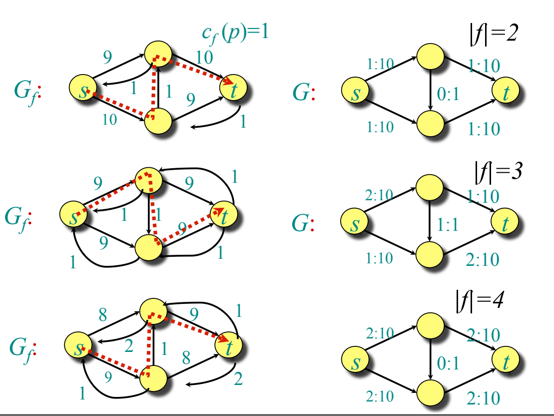
After augmentation



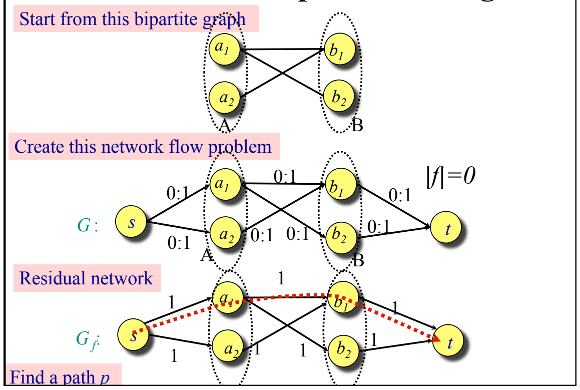
Ford-Fulkerson max-flow algorithm

- Start: $f[u, v] \leftarrow 0$ for all $u, v \in V$
- While **(1)** {
  - construct G_f
  - if an augmenting path p in G_f exists then
    augment f by $c_f(p)$ //Any path would do
  - else exit }

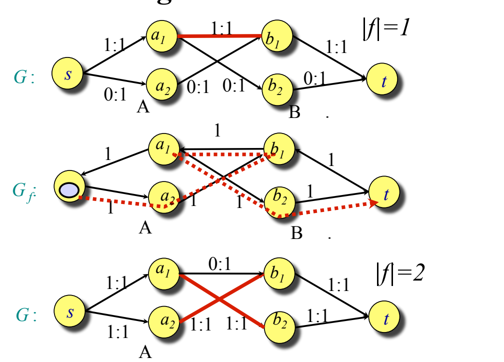




Another example - Matching



Matching Cont

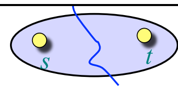


Correctness of FF algorithm

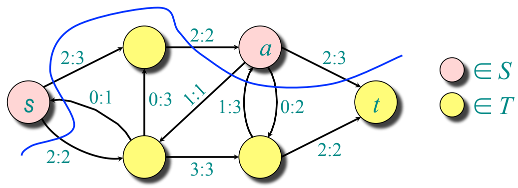
We will next show that when FF-algorithm terminates, it is because it has found a maximum flow.

The proof also provides additional information about the network (wait for it)

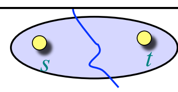
Cuts



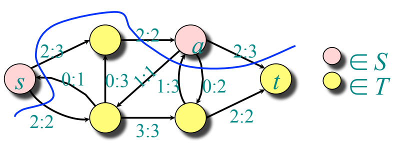
Definitions. A ***cut*** (S, T) of a flow network $G = (V, E)$ is a partition of V into two sets S, T, such that $s \in S$ and $t \in T$.



Cuts



Definitions. A ***cut*** (S, T) of a flow network $G = (V, E)$ is a partition of V into two sets S, T, such that $s \in S$ and $t \in T$.

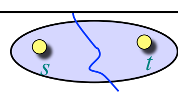


More exampls of cuts.

$(s, V\text{-}s)$

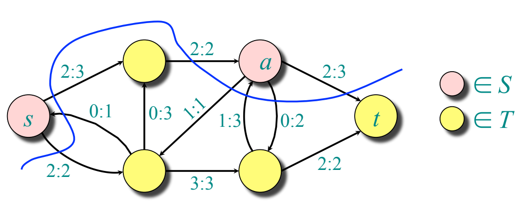
$(V\text{-}t, t)$

Flow across a cut



Definitions.

If f is a flow on G, then the ***flow across the cut***, denoted $f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u,v)$



$S=\{s,a\}$
$f(S, T) = (2 + 2) + (-2 + 1 - 1 + 2) = 4$

Another characterization of flow value

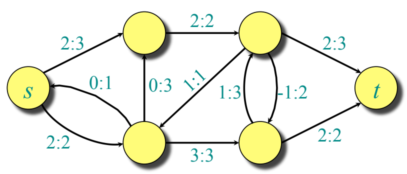
Recall: $|f|=f(s,V)= \sum_{v \in V} f(s,v)$

Lemma. For any flow f and any cut (S, T), we have $|f| = f(S, T)$.

Proof. Omitted
Meaning that you don't need to know the proof, but you do need to know this lemma

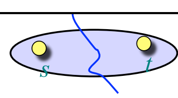
Conclusion: The flow into the sink equals the flow from the source $f(s,V)=f(V.t)$

Flow into the sink

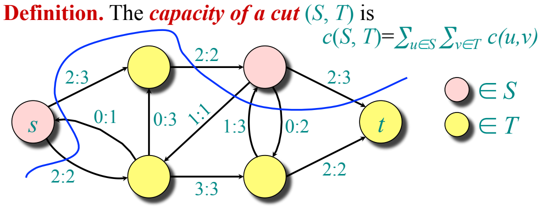


$|f| = f(s, V) = 4$ $\quad$ $f(V, t) = 4$

Capacity of a cut



Definition. The ***capacity of a cut*** (S, T) is
$c(S, T)=\sum_{u \in S} \sum_{v \in T} c(u,v)$



$c(S, T) = (3 + 2) + (1 + 2 + 3)$
$= 11$

Upper bound on the maximum flow value

Theorem. The value of any flow no larger than the capacity of any cut: $|f| \le c(S,T)$.

Proof: Recall that for the flow f to be "legit", we must have $f(u,v) \le c(u,v)$ for every pair of vertice (u,v) . Hence

$$\begin{aligned}|f| &= f(S,T)\\ &= \sum_{u\in S}\sum_{v\in T} f(u,v)\\ &\le \sum_{u\in S}\sum_{v\in T} c(u,v)\\ &= c(S,T) \quad \square\end{aligned}$$

Max-flow, min-cut theorem

Theorem. The following are equivalent:

1. $|f| = c(S, T)$ for some cut (S, T).
2. f is a maximum flow.
3. f admits no augmenting paths.

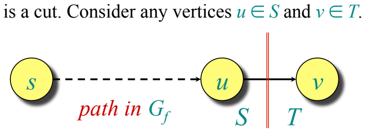
Proof.
(1) $\Rightarrow$ (2): Since $|f| \le c(S, T)$ for any cut (S, T) (by the theorem from a few slides back), the assumption that $|f| = c(S, T)$ implies that f is a maximum flow.

(2) $\Rightarrow$ (3): If there were an augmenting path, the flow value could be increased, contradicting the maximality of f.

(3) $\Rightarrow$ (1): Define $S = \{v \in V \mid \text{there exists a path in } G_f \text{ from } s \text{ to } v\}$,

Let $T = V - S$. Since f admits no augmenting paths, there is no path from s to t in G_f.
Hence, $s \in S$ and $t \notin S$, So $t \in T$.

Thus (S, T) is a cut. Consider any vertices $u \in S$ and $v \in T$.



Consider $u \in S$, $v \in T$. We must have $c_f(u, v) = 0$, since if $c_f(u, v) > 0$, then $v \in S$, not $v \in T$ as assumed.

Thus, $f(u, v) = c(u, v)$, since $c_f(u, v) = c(u, v) - f(u, v)$.

Summing over all $u \in S$ and $v \in T$ yields $f(S, T) = c(S, T)$, and since $|f| = f(S, T)$, the theorem follows. $\square$

Note that this proof is constructive

- It find A cut which forms a bottleneck
- (note that there might be others bottlenecks cuts)
- If we decrease the capacity of ANY edge along this cut, it would decrease the max flow.
- The Theorem also implies that FF algorithm is results in a max flow.

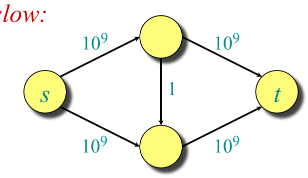
Ford-Fulkerson max-flow algorithm

Algorithm:

$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G_f wrt f exists
do augment f by $c_f(p)$

Can be slow:

G:



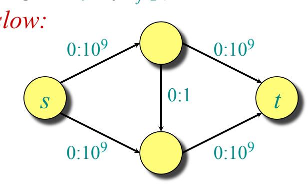
Ford-Fulkerson max-flow algorithm

Algorithm:

$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G wrt f exists
do augment f by $c_f(p)$

Can be slow:

G:

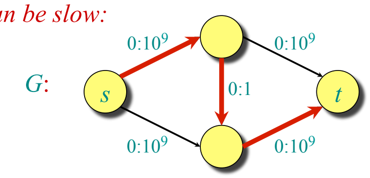


Ford-Fulkerson max-flow algorithm

Algorithm:

$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G wrt f exists
  do augment f by $c_f(p)$

Can be slow:

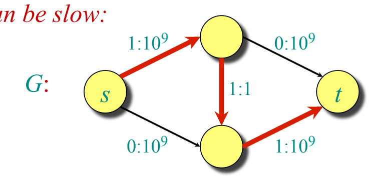


Ford-Fulkerson max-flow algorithm

Algorithm:

$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G wrt f exists
  do augment f by $c_f(p)$

Can be slow:

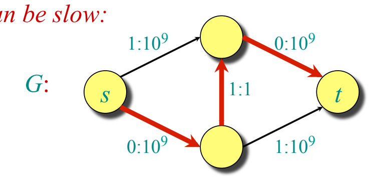


Ford-Fulkerson max-flow algorithm

Algorithm:

$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G wrt f exists
  do augment f by $c_f(p)$

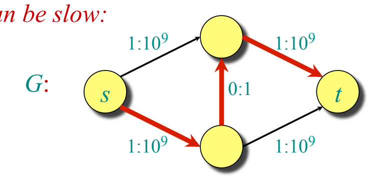
Can be slow:



Ford-Fulkerson max-flow algorithm

Algorithm:
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G wrt f exists
do augment f by $c_f(p)$

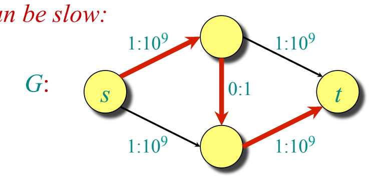
Can be slow:



Ford-Fulkerson max-flow algorithm

Algorithm:
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G wrt f exists
do augment f by $c_f(p)$

Can be slow:



Ford-Fulkerson max-flow algorithm

Algorithm:
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path p in G wrt f exists
do augment f by $c_f(p)$

Runtime:
- Let $|f^*|$ be the value of a maximum flow, and assume it is an **integral** value.
- The initialization takes $O(|E|)$
- There are at most $|f^*|$ iterations of the loop
- Find an augmenting path with DFS in $O(|V|+|E|)$ time
- Each augmentation takes $O(|V|)$ time

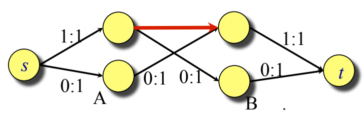
$\Rightarrow O(|E| \cdot |f^*|)$ in total

Ford-Fulkerson and matching

Recall – we expressed the maximum matching problem as a network flow, but we can express the max flow as a matching, only if the flow is an **integer** flow.

However, this is always the case once using F&F algorithm: The flow along each edge is either 0 or 1.

G :



Runtime analysis of F&F-algorithm applied to bipartite matching

- We saw that in each iteration of F&F algorithm, $|f|$ increases by at least 1.
- Let $|f^*|$ be the maximum value.
- How large can $|f^*|$ be ?

- <u>Claim</u>: $|f^*| \le min\{|A|, |B|\}$ (why ?)
- Runtime is $O(|E| \cdot min\{|A|, |B|\}) = O(|E||V|)$
- Can be done in $O(|E|^{1/2} \cdot |V|)$ (Dinic Algorithm)

Edmonds-Karp algorithm

Edmonds and Karp noticed that many people's implementations of Ford-Fulkerson augment along a ***breadth-first augmenting path***: a path with smallest number of edges in G_f from s to t.

These implementations would always run relatively fast.

Since a breadth-first augmenting path can be found in $O(|E|)$ time, their analysis, focuses on bounding the number of flow augmentations.

(In independent work, Dinic also gave polynomial-time bounds.)

Running time of Edmonds-Karp

- One can show that the number of flow augmentations (i.e., the number of iterations of the while loop) is $O(|V|\ |E|)$.
- Breadth-first search runs in $O(|E|)$ time
- All other bookkeeping is $O(|V|)$ per augmentation.

⇒ The Edmonds-Karp maximum-flow algorithm runs in $O(|V||\ E|^{2})$ time.

Best to date

- The asymptotically fastest algorithm to date for maximum flow, due to King, Rao, and Tarjan, runs in $O(V\,E \log_{E/(V \lg V)} V)$ time.
- If we allow running times as a function of edge weights, the fastest algorithm for maximum flow, due to Goldberg and Rao, runs in time
$$O(\min\{V^{2/3}, E^{1/2}\} \cdot E \lg (V^{2}/E + 2) \cdot \lg C),$$
where C is the maximum capacity of any edge in the graph.