Algorithms for Data Processing
Lecture V: Network Flow

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science
http://www.inf.unibz.it/˜artale
artale@inf.unibz.it

2019/20 – First Semester
MSc in Computational Data Science — UNIBZ

Some material (text, figures) displayed in these slides is courtesy of:
Alberto Montresor, Werner Nutt, Kevin Wayne, Jon Kleinberg, Eva Tardos.

Flow Networks

- Use of graphs to model transportation networks: networks whose edges carry some sort of traffic and whose nodes act as switches passing traffic between different edges.
  - Example. Fluid network in which edges are pipes that carry liquid, and the nodes are junctures where pipes are plugged together.

Flow Networks – Ingredients

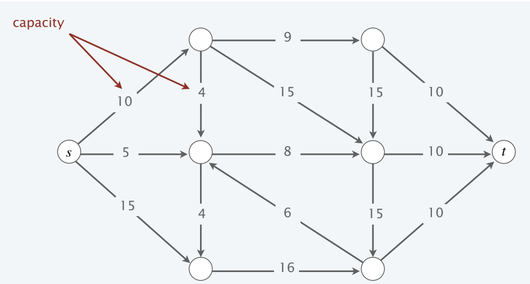
- Capacity on the edge, indicating how much “traffic” can carry;
- Source nodes in the graph, which generate traffic;
- sink (or destination) nodes in the graph, which can "absorb" traffic as it arrives;
- traffic or flow which is currently transmitted across an edge.

Flow Networks – Ingredients/2

A Flow Network is a tuple $G = (V, E, s, t, c)$, where:

- (V, E) is a directed graph, with
- a single source $s \in V$, and a single sink $t \in V$;
- Nodes other than s and t will be called internal nodes.
- Capacity $c(e) > 0$ for each $e \in E$.

Intuition. Material flowing through a transportation network, originating at source and sent to sink.



Flow Networks – Ingredients/3

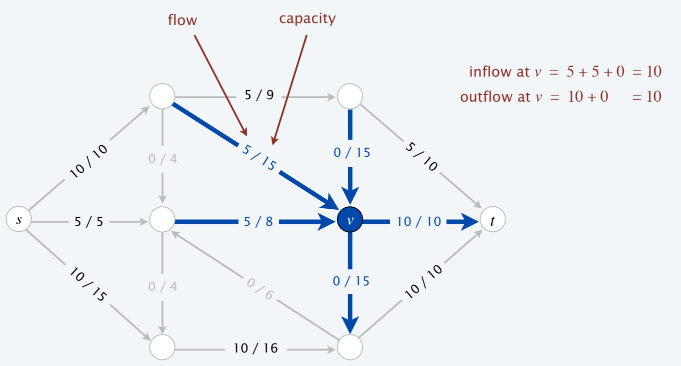
We make the following assumptions:

- No edge enters the source s, and no edge leaves the sink t;
- There is at least one edge incident to each node;
- Capacities are integers.

Flow Networks – Ingredients/4

We now define what it means for our network to carry traffic or flow.

- An *st*-flow is a function, $f : E \to \mathbf{R}^+$, that satisfies:
  Capacity Condition $\forall e \in E. 0 \leq f(e) \leq c(e)$
  Conservation Condition $\forall v \in V \setminus \{s, t\}. \sum_{e \ into \ v} f(e) = \sum_{e \ out \ of \ v} f(e)$

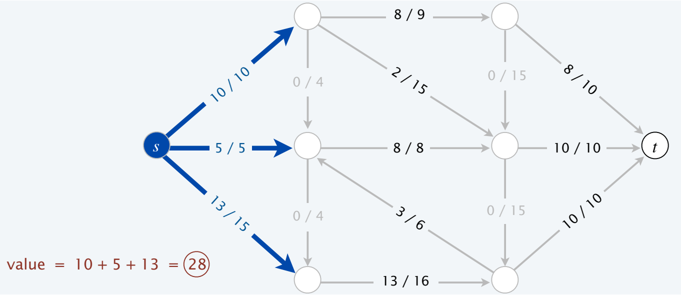


Flow Networks – Ingredients/5

Value of a flow. Amount of flow generated at the source:

$$val(f) = \sum_{e \text{ out of } s} f(e)$$

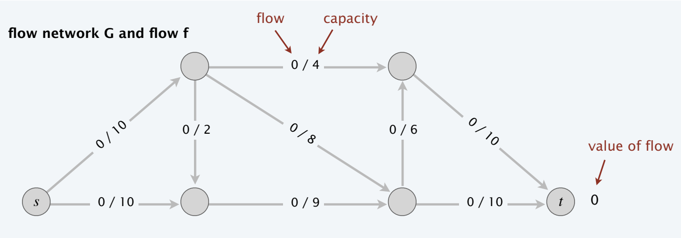
Max-Flow Problem. Find a flow of maximum value.



Toward a Max-Flow Algorithm

Greedy algorithm.

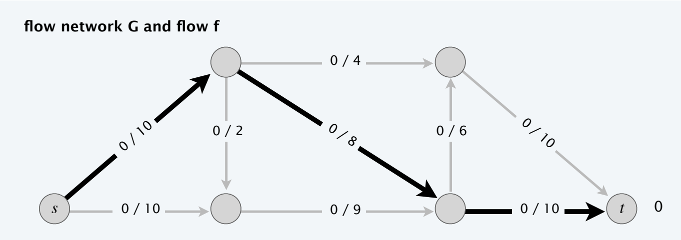
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an s-t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck!



Toward a Max-Flow Algorithm

Greedy algorithm.

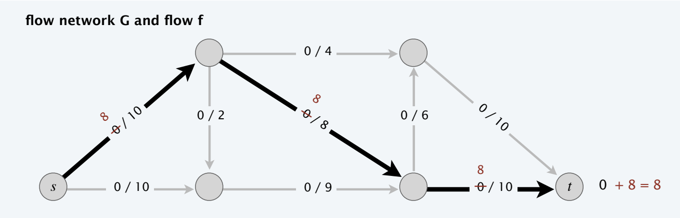
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an s–t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck!



Toward a Max-Flow Algorithm

Greedy algorithm.

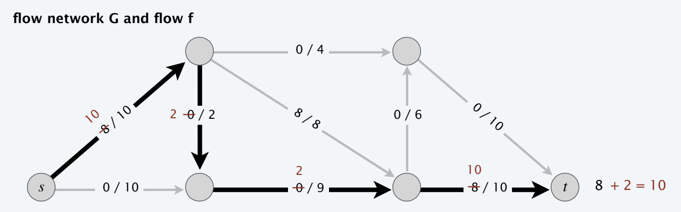
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an s-t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck!



Toward a Max-Flow Algorithm

Greedy algorithm.

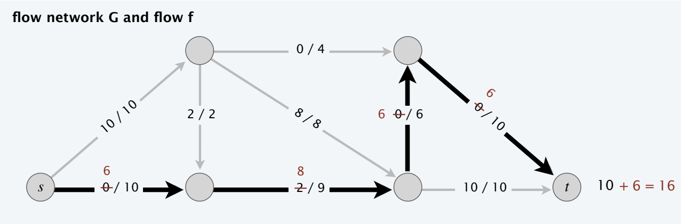
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an s-t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck!



Toward a Max-Flow Algorithm

Greedy algorithm.

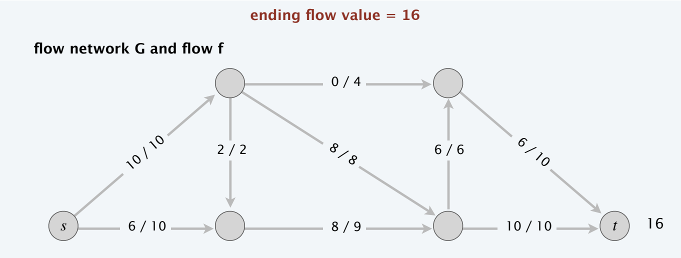
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an s-t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck!



Toward a Max-Flow Algorithm

Greedy algorithm.

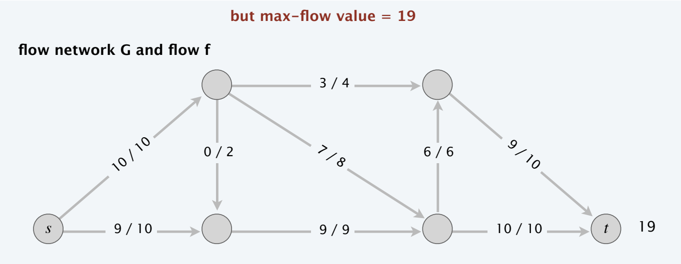
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an s-t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck!



Toward a Max-Flow Algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an s-t path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck!



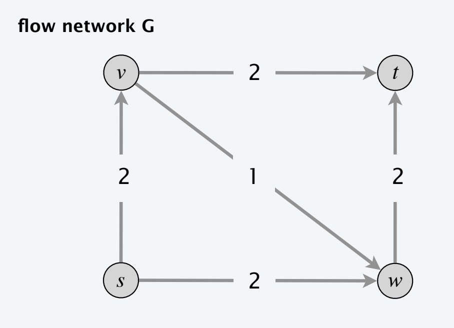
Why the greedy algorithm fails

Q. Why does the greedy algorithm fail?

A. Greedy algorithm never decreases the flow on an edge.

Example. Consider the flow network G below.

- The unique max flow has $f(v, w) = 0$.
- Greedy algorithm could choose $s \to v \to w \to t$ as first augmenting path.



Mechanism to Undo a Bad Decision

This is a more general way of pushing flow:

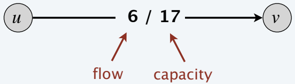
- We can push forward on edges with leftover capacity;
- We can push backward on edges that are already carrying flow, to divert it in a different direction.

We now define the residual graph, which provides a systematic way to search for *forward-backward* operations.

Residual network

Original edge. $e = (u, v) \in E$.

- Flow $f(e)$.
- Capacity $c(e)$.

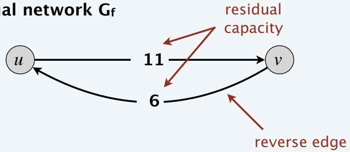


Reverse edge. $e^{\text{reverse}} = (v, u)$.

- "Undo" flow sent.

Residual capacity.

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^{\text{reverse}} \in E \end{cases}$$



edges with positive residual capacity

where flow on a reverse edge negates flow on corresponding forward edge

Residual network. $G_f = (V, E_f, s, t, c_f)$.

- $E_f = \{e : f(e) < c(e)\} \cup \{e^{\text{reverse}} : f(e) > 0\}$.
- Key property: f' is a flow in G_f iff $f + f'$ is a flow in G.

Augmenting Path

Augmenting Path. A simple $s - t$ path in the *residual network* G_f, for a given graph G and a flow f.

Bottleneck Capacity. Minimal residual capacity of any edge in an augmenting path.

Note. In the following a *reverse* edge will be denoted as a backward edge.

Augmenting Path/2

The following algorithm makes precise the way in which we push flow from s to t in G as a consequence of an augmenting path, P, in the residual network G_f.

```
AUGMENT(f,P)
b = BOTTLENECK(P, f);
for each edge (u, v) ∈ P do
    if e = (u, v) is a forward edge then
        f(e) = f(e) + b in G
    else
        f(e) = f(e) − b in G;        /* (u, v) is a backward edge, and e = (v, u) */
return f
```

Key Property. The result of `AUGMENT(f,P)` is a new flow f' in G such that $val(f') = val(f) + b$.

Ford-Fulkerson Algorithm (1956)

```
MAX-FLOW(G)
f(e) = 0 for all e ∈ G;
G_0 = G;                                  /* Initialize the residual graph G_0 */
while there is an s-t path, P, in the residual graph G_f do
    if P is a simple s-t path in G_f then
        f' = AUGMENT(f, P);
        G_f' = UPDATE(G_f,P,f');                        /* Update G_f to G_f' */
        f ← f';
return f
```

```
UPDATE(G_f,P,f')
compute the bottleneck b from f and f';
for each (u, v) in P do
    c(u,v) = c(u,v) - b;
    if c(u,v) =0 then
        delete edge (u, v) from G_f
    if (v, u) ∉ G_f then
        add edge (v, u) to G_f
    c(v,u) = c(v,u) + b;
return G_f
```

Correctness of the Ford-Fulkerson Algorithm

Property/1. Let f be a flow in G, and let P be a simple s-t path in G_f. Then, $val(f') = val(f) + \texttt{bottleneck}(P, f)$; and since $\texttt{bottleneck}(P, f) > 0$, we have $val(f') > val(f)$.

Property/2. Let $C = \texttt{max}_e\{c(e)\}$, then, the Ford-Fulkerson Algorithm terminates in at most nC iterations of the `While loop`.

Proof. Note that:

- $val(f_{max}) \leq \sum_{e \text{ out of } s} c(e) \leq nC$;
- By Property/1, the value of the flow increases at each iteration by at least 1 unit.

Note! The book has a different choice: $C = \sum_{e \text{ out of } s} c(e)$

Running Time of the Ford-Fulkerson Algorithm

Assuming that all nodes have at least one incident edge, then $m > n/2$, and so we can say that $O(m+n) = O(m)$.

Running Time. The Ford-Fulkerson Algorithm can be implemented to run in $O(mnC)$ time.

Proof. Complexity in one iteration of the `While` loop.

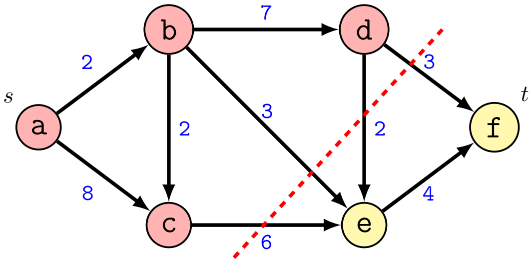
- The residual graph G_f has at most $2m$ edges;
- G_f is stored using an adjacency list;
- To find an s-t path in G_f, we can use BFS or DFS, which runs in $O(m+n)$ time which, by our assumption, is the same as $O(m)$;
- The procedure `AUGMENT`(f, P) takes time $O(n)$, as the path P has at most $n-1$ edges;
- The procedure `UPDATE`(G_f, P, b) takes also time $O(n)$.

Insight in the Max-Flow Problem

- We continue with the analysis of the Ford-Fulkerson Algorithm.
- Objective: to find considerable insights into the Maximum-Flow problem itself.
- We already saw that $val(f) \leq \sum_{e \text{ out of } s} c(e)$.
- Is there a better approximation of $val(f)$?

Cut in a Flow Network

Cut in a Flow Network. A cut, (S, T), in a Flow Network $G = (V, E, s, t, c)$ is called an s–t cut if it is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.

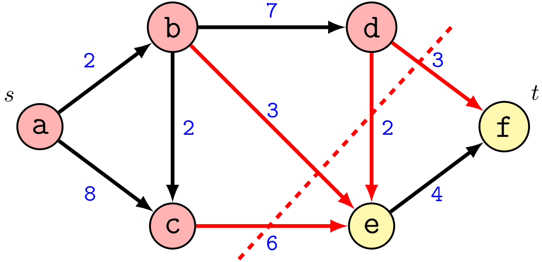


$S = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$
$T = \{\mathtt{e}, \mathtt{f}\}$

Cut in a Flow Network/2

Capacity across the cut. The capacity across a cut, $c(S, T)$, is given by the following formula:

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v) = \sum_{e \text{ out of } S} c(e)$$



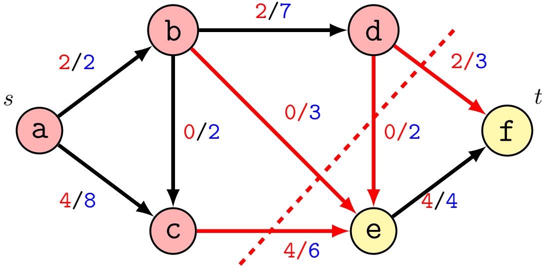
$S = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$
$T = \{\mathtt{e}, \mathtt{f}\}$

$C(S, T) = 14$

Cut in a Flow Network/3

Flow across the cut. The flow across a cut, $f(S,T)$, is given by the following formula:

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v) - \sum_{v \in T, u \in S} f(v,u) = \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e)$$



$S = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$
$T = \{\mathtt{e}, \mathtt{f}\}$

$C(S,T) = 14$

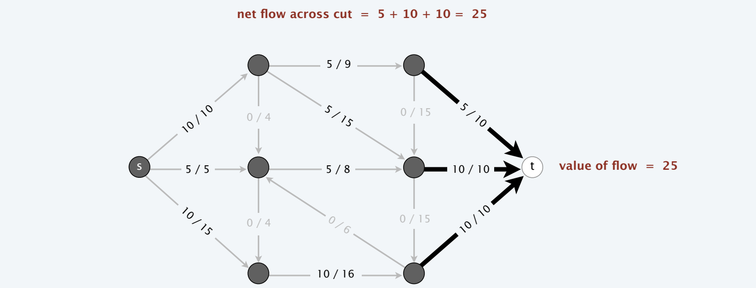
$F(S,T) = 6$

Flow value Lemma

Cuts provide very natural upper bounds on the values of flows.

Flow value Lemma. Let f be any s–t flow and (S, T) be any s–t cut. Then, the value of the flow f equals the flow across the cut (S, T):

$$val(f) = \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e)$$

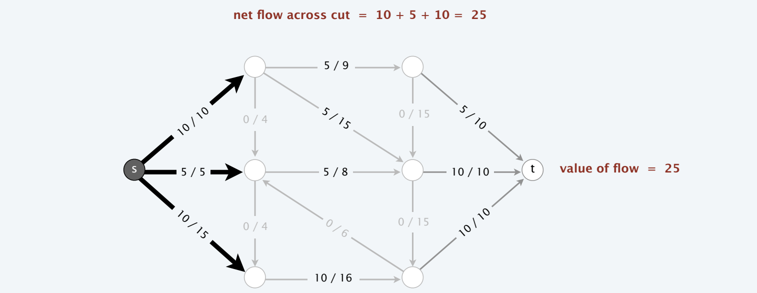


Flow value Lemma/2

Cuts provide very natural upper bounds on the values of flows.

Flow value Lemma. Let f be any s–t flow and (S, T) be any s–t cut. Then, the value of the flow f equals the flow across the cut (S, T):

$$val(f) = \sum_{e \ out\ of\ S} f(e) - \sum_{e\ into\ S} f(e)$$

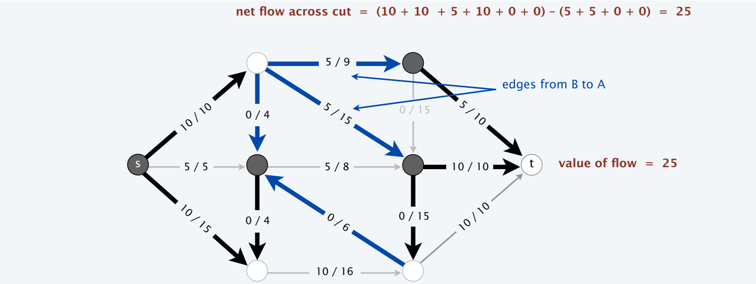


Flow value Lemma/3

Cuts provide very natural upper bounds on the values of flows.

Flow value Lemma. Let f be any s–t flow and (S, T) be any s–t cut. Then, the value of the flow f equals the flow across the cut (S, T):

$$val(f) = \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e)$$



Flow value Lemma – Proof

Flow value Lemma. Let f be any s–t flow and (S, T) be any s–t cut. Then, the value of the flow f equals the flow across the cut (S, T):

$$val(f) = \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e)$$

Proof.

$$val(f) = \sum_{e \text{ out of } s} f(e) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ into } s} f(e) =$$

$$= \sum_{v \in S} \Big(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \Big) \; [\textit{all are 0 except for } v{=}s]$$

$$= \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e)$$

Relationship between Flow and Capacity

Weak duality Lemma. Let f be any s–t flow, and (S, T) any s–t cut. Then,

$$val(f) \leq c(S, T).$$

Proof.

$$\begin{aligned} val(f) &= \sum_{e \ out \ of \ S} f(e) - \sum_{e \ into \ S} f(e) \leq \\ &\leq \sum_{e \ out \ of \ S} f(e) \leq \\ &\leq \sum_{e \ out \ of \ S} c(e) = c(S, T). \end{aligned}$$

Relationship between Flow and Capacity/2

- The Weak duality Lemma abstracts from a partcular flow;
- The value of every flow is upper-bounded by the capacity of any possible cut;
- Thus, if we exhibit an s-t cut in G of some value $c(S, T)$ we know that there cannot be an s-t flow in G of value greater than $c(S, T)$.
- Viceversa, if we exhibit any s-t flow, f^* in G, we know that there cannot be an s-t cut in G with s-t capacity less than f^*.

Certificate of optimality

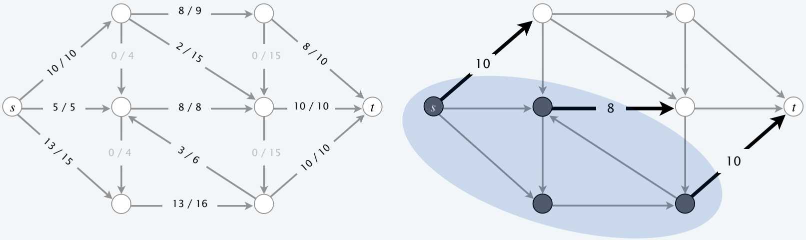
Corollary. Let f be a flow and let (A, B) be any cut.
If $val(f) = cap(A, B)$, then f is a max flow and (A, B) is a min cut.

Pf.

weak duality

- For any flow f': $val(f') \leq cap(A, B) = val(f)$.
- For any cut (A', B'): $cap(A', B') \geq val(f) = cap(A, B)$. ▪

weak duality



value of flow = 28 = capacity of cut = 28

Max-flow min-cut theorem

Max-flow min-cut theorem. Value of a max flow = capacity of a min cut.

strong duality

MAXIMAL FLOW THROUGH A NETWORK

L. R. FORD, JR. AND D. R. FULKERSON

Introduction. The problem discussed in this paper was formulated by T. Harris as follows:

"Consider a rail network connecting two cities by way of a number of intermediate cities, where each link of the network has a number assigned to it representing its capacity. Assuming a steady state condition, find a maximal flow from one given city to the other."

ON THE MAX FLOW MIN CUT THEOREM OF NETWORKS

G. B. Dantzig
D. R. Fulkerson

P-826

April 15, 1955

A Note on the Maximum Flow Through a Network*

P. ELIAS†, A. FEINSTEIN‡, AND C. E. SHANNON§

Summary—This note discusses the problem of maximizing the rate of flow from one terminal to another, through a network which consists of a number of branches, each of which has a limited capacity. The main result is a theorem: The maximum possible flow from left to right through a network is equal to the minimum value among all simple cut-sets. This theorem is applied to solve a more general problem, in which a number of input nodes and a number of output nodes are used.

from one terminal to the other in the original network passes through at least one branch in the cut-set. In the network above, some examples of cut-sets are (d, e, f), and (b, c, e, g, h), (d, g, h, i). By a *simple cut-set* we will mean a cut-set such that if any branch is omitted it is no longer a cut-set. Thus (d, e, f) and (b, c, e, g, h) are simple cut-sets while (d, g, h, i) is not. When a simple cut-set is

Analyzing the Ford-Fulkerson Algorithm

- Let $\overline{f}$ denote the flow that is returned by the Ford-Fulkerson Algorithm. We want to prove that $\overline{f}$ is the maximum-flow by showing that there is an (S, T) cut such that:

$$val(\overline{f}) = c(S, T)$$

- The Ford-Fulkerson Algorithm terminates when for the flow f there is no $s - t$ path in the residual graph G_f.
  This is the only property needed for proving its maximality.

Correctness the Ford-Fulkerson Algorithm/1

Theorem 1. If f is an s-t flow such that there is no s-t path in the residual graph G_f (Augmenting Path) then there is an s-t cut (A, B) in G for which $val(f) = c(A, B)$.

Proof.

- Let A be set of nodes reachable from s in the residual network G_f and $B = V \setminus A$. We show that (A, B) is an s-t cut:
  - (A, B) is a partition of V, and $s \in A$, and $t \notin A$ by the assumption that there is no s-t path in G_f, hence $t \in B$.
- Let $e = (u, v)$ be an edge in G for which $u \in A$ and $v \in B$, then $f(e) = c(e)$ – otherwise e would be a forward edge in G_f.
- Let $e' = (u', v')$ be an edge in G for which $u' \in B$ and $v' \in A$, then $f(e') = 0$ – otherwise e' would give rise to a backward edge $e'' = (v', u')$ in G_f.
- So all edges out of A are completely saturated with flow, while all edges into A are completely unused.

Correctness the Ford-Fulkerson Algorithm/2

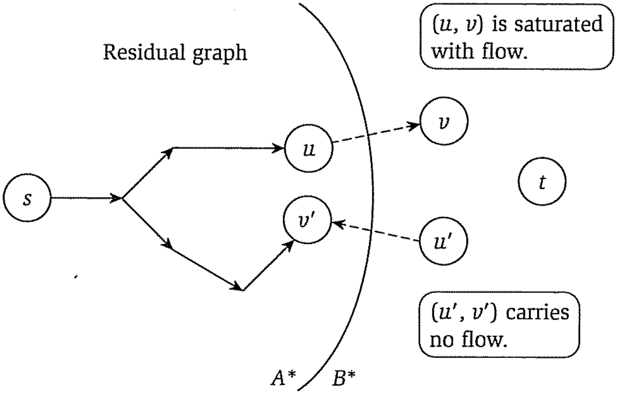


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Correctness the Ford-Fulkerson Algorithm/3

- We can now use the Flow value Lemma:

$$
\begin{aligned}
val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = \\
&= \sum_{e \text{ out of } A} c(e) - 0 \\
&= c(A, B)
\end{aligned}
$$

That proves the Theorem!

Theorem 2. The flow $\overline{f}$ returned by the Ford-Fulkerson Algorithm is a maximum flow.

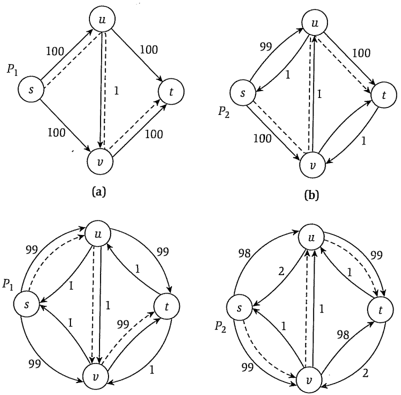
Integer-Value Flows

When all capacities are integer values we can guarantee the existence of a max-flow as expressed in the following Theorem.

Theorem [Integrality theorem.] If all capacities in the flow network are integers, then there is a maximum flow f for which every flow value $f(e)$ is an integer.

Choosing Good Augmenting Paths

- We already saw that $val(f) \leq \sum_{e \text{ out of } s} c(e)$ is an upper bound to the number of iterations.
- The Ford-Fulkerson Algorithm can perform vary badly when pathological augmenting paths are selected: Here we need 200 steps!!



Choosing Good Augmenting Paths

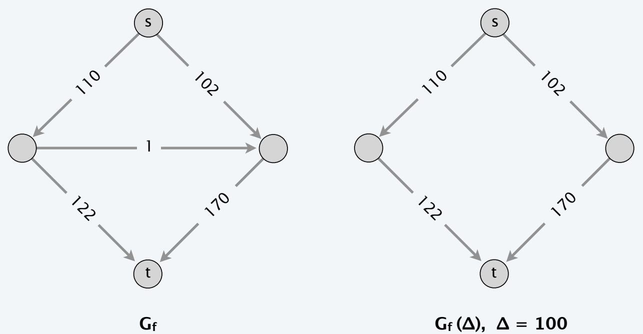
Use care when selecting augmenting paths.

- If we choose paths with large bottleneck capacity we will require less iterations.
- A natural idea is to select at each iteration the path that has the largest bottleneck capacity.
- Finding such paths can slow down each iteration.

Capacity-Scaling Algorithm

Main Idea: Choosing augmenting paths with large bottleneck capacity though not necessarily the largest.

- Maintain a scaling parameter Δ;
- Let $G_f(\Delta)$ be the sub-graph of the residual network G_f containing only those edges with residual capacity $\geq \Delta$;
- Any augmenting path in $G_f(\Delta)$ has bottleneck capacity $\geq \Delta$.



Capacity-Scaling Algorithm/2

```
CAPACITY-SCALING(G,s,t)
f(e) = 0 for all e ∈ G;
Δ = largest power of 2 ≤ max_{e out of s}{c(e)};
Compute G_0(Δ);
while Δ ≥ 1 do
    while there is an s-t path, P, in the residual graph G_f(Δ) do
        if P is a simple s-t path then
            f' = AUGMENT(f, P);
            G_f'(Δ) = UPDATE(G_f(Δ),P,f');          /* Update G_f(Δ) to G_f'(Δ) */
            f ← f';
    Δ = Δ/2
return f
```

Correctness of the Capacity-Scaling Algorithm

- The Capacity-Scaling Max-Flow Algorithm is just an optimized implementation of the original Ford-Fulkerson Algorithm.
- The search in the restricted residual graph $G_f(\Delta)$ is used to guide the selection of augmenting paths with large residual capacity.

Properties of Capacity-Scaling Algorithm. If the capacities are integer-valued, then throughout the Capacity-Scaling Max-Flow algorithm the flow and the residual capacities remain integer-valued. This implies that when $\Delta = 1$, $G_f(\Delta)$ is the same as G_f and hence when the algorithm terminates the flow, f, is of maximum value.

Capacity-Scaling Algorithm: Run Time Analysis

- We call an iteration of the outside `While` loop, with a fixed value of Δ, the Δ-scaling phase.
- We denote $C = \mathtt{max}_e\{c(e)\}$.

Lemma 1. There are $1 + \lfloor \log_2 C \rfloor$ Δ-scaling phases.
Proof. Initially $C/2 < \Delta \leq C$; Δ decreases by a factor of 2 in each iteration.

Lemma 2. During the Δ-scaling phase each augmentation increases the flow value by at least Δ.
Proof. During the Δ-scaling phase we only use edges with residual capacity of at least Δ.

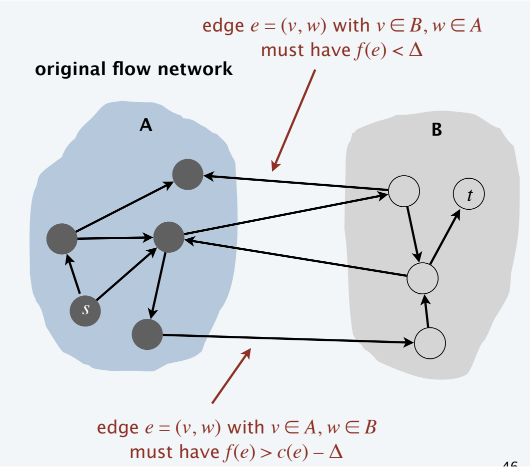
Capacity-Scaling Algorithm: Run Time Analysis/2// (Proof Not Required)

Lemma 3. Let f be the flow at the end of a Δ-scaling phase. Then, the max-flow value $\leq val(f) + m\Delta$ (where m is the number of edges).

Proof.

- We show there exists a cut (A, B) such that $c(A, B) \leq val(f) + m\Delta$.
- Let A be the set of nodes reachable from s in $G_f(\Delta)$ and $B = V \setminus A$. We show that (A, B) is an s–t cut:
  - (A, B) is a partition of V, and $s \in A$, and $t \notin A$ for otherwise there is an s–t path in $G_f(\Delta)$, hence $t \in B$.
- Let $e = (u, v)$ be an edge in G for which $u \in A$ and $v \in B$, then $c(e) - f(e) < \Delta$, otherwise e would be a forward edge in $G_f(\Delta)$, contradicting $v \in B$.
- Let $e' = (u', v')$ be an edge in G for which $u' \in B$ and $v' \in A$, then $f(e') < \Delta$, otherwise e' would give rise to a backward edge $e'' = (v', u')$ in $G_f(\Delta)$.
- So all edges out of A are almost saturated ($f(e) > c(e) - \Delta$), while all edges into A are almost empty ($f(e) < \Delta$).

Capacity-Scaling Algorithm: Run Time Analysis/3 (Proof Not Required)



Capacity-Scaling Algorithm: Run Time Analysis/4 (Proof Not Required)

Proof of Lemma 3 (cont.)

$$\begin{aligned} val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \geq \quad [\texttt{By the Flow Value Lemma}] \\ &\geq \sum_{e \text{ out of } A} \big(c(e) - \Delta\big) - \sum_{e \text{ into } A} \Delta = \\ &= \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ into } A} \Delta \geq c(A, B) - m\Delta. \end{aligned}$$

Thus, $c(A, B) \leq val(f) + m\Delta$, which proves Lemma 3.

Capacity-Scaling Algorithm: Run Time Analysis/5 (Proof Not Required)

Lemma 4. The number of augmentations in each scaling phase is $\leq 2m$.
Proof.

- True in the first scaling phase: we can have as many augmenting paths as many edges out of s (note that by Lemma 2, each augmenting phase increases the flow by at least Δ);
- In any later Δ-scaling phase, let f_p the flow at the end of the *previous* scaling phase;
- In the *previous* scaling phase we had $\Delta_p = 2\Delta$;
- By Lemma 3, $val(f_{max}) \leq val(f_p) + m\Delta_p = val(f_p) + 2m\Delta$;
- In the current Δ-scaling phase, by Lemma 2, each augmentation increases the flow by at least Δ, and hence there can be at most $2m$ augmentations.

Capacity-Scaling Algorithm: Run Time Analysis/6 (Proof Not Required)

Theorem. The Scaling Max-Flow algorithm in a graph with m edges and integer capacities finds a maximum flow in at most $2m(1 + \lfloor \log_2 C \rfloor)$ augmentations. It can be implemented to run in at most $O(m^2 \log_2 C)$ time.

Proof.

- By Lemmas 1 and 4, we can have at most $2m(1 + \lfloor \log_2 C \rfloor)$ augmentations, i.e., $O(m \log_2 C)$;
- Each augmentation takes $O(m)$ including the time to find a path (BFS/DFS) and to generate the new residual graph.

Considerations on Running Time

- When C is large, the scaling algorithm, $O(m^2 \log_2 C)$, outperforms the generic implementation of the Ford-Fulkerson Algorithm, $O(mnC)$.

Considerations on Running Time

- When C is large, the scaling algorithm, $O(m^2 \log_2 C)$, outperforms the generic implementation of the Ford-Fulkerson Algorithm, $O(mnC)$.
- The generic Ford-Fulkerson algorithm requires time proportional to the *magnitude* of the capacities, while the scaling algorithm only requires time proportional to the *number of bits* needed to specify the capacities in the input.

Considerations on Running Time

- When C is large, the scaling algorithm, $O(m^2 \log_2 C)$, outperforms the generic implementation of the Ford-Fulkerson Algorithm, $O(mnC)$.
- The generic Ford-Fulkerson algorithm requires time proportional to the *magnitude* of the capacities, while the scaling algorithm only requires time proportional to the *number of bits* needed to specify the capacities in the input.
- When the generic Ford-Fulkerson algorithm chooses patological paths could require C iterations, i.e., exponential time in the size of the bit representation of the input.

Considerations on Running Time

- When C is large, the scaling algorithm, $O(m^2 \log_2 C)$, outperforms the generic implementation of the Ford-Fulkerson Algorithm, $O(mnC)$.
- The generic Ford-Fulkerson algorithm requires time proportional to the *magnitude* of the capacities, while the scaling algorithm only requires time proportional to the *number of bits* needed to specify the capacities in the input.
- When the generic Ford-Fulkerson algorithm chooses patological paths could require C iterations, i.e., exponential time in the size of the bit representation of the input.
- The scaling algorithm is running in time polynomial in the size of the input, i.e., the number of edges and the bit representation of the capacities.

Augmenting Path Algorithms: Summary

year	method	# augmentations	running time
1955	**augmenting path**	$n\,C$	$O(m\,n\,C)$
1972	**fattest path**	$m \log(mC)$	$O(m^2 \log n \log(mC))$
1972	**capacity scaling**	$m \log C$	$O(m^2 \log C)$
1985	**improved capacity scaling**	$m \log C$	$O(m\,n \log C)$
1970	**shortest augmenting path**	$m\,n$	$O(m^2\,n)$
1970	**level graph**	$m\,n$	$O(m\,n^2)$
1983	**dynamic trees**	$m\,n$	$O(m\,n \log n)$

fat paths

shortest paths

augmenting-path algorithms with m edges, n nodes, and integer capacities between 1 and C

Max-Flow Algorithms: Summary

year	method	worst case	discovered by
1951	**simplex**	$O(m\, n^2\, C)$	Dantzig
1955	**augmenting paths**	$O(m\, n\, C)$	Ford–Fulkerson
1970	**shortest augmenting paths**	$O(m\, n^2)$	Edmonds–Karp, Dinitz
1974	**blocking flows**	$O(n^3)$	Karzanov
1983	**dynamic trees**	$O(m\, n \log n)$	Sleator–Tarjan
1985	**improved capacity scaling**	$O(m\, n \log C)$	Gabow
1988	**push-relabel**	$O(m\, n \log (n^2 / m))$	Goldberg–Tarjan
1998	**binary blocking flows**	$O(m^{3/2} \log (n^2 / m) \log C)$	Goldberg–Rao
2013	**compact networks**	$O(m\, n)$	Orlin
2014	**interior-point methods**	$\tilde{O}(m\, m^{1/2} \log C)$	Lee–Sidford
2016	**electrical flows**	$\tilde{O}(m^{10/7}\, C^{1/7})$	Mądry
20xx		???	

max-flow algorithms with m edges, n nodes, and integer capacities between 1 and C

Thank You!