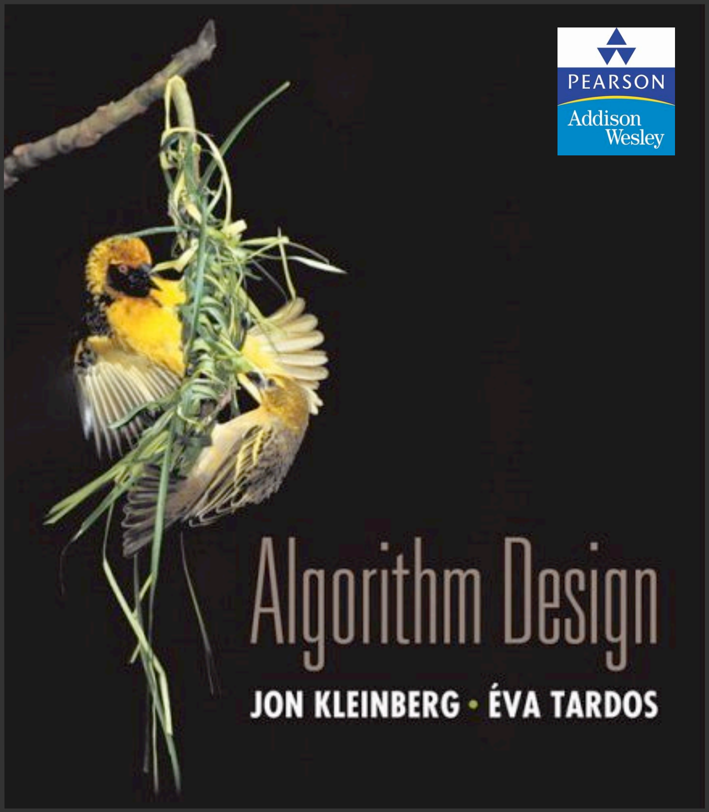


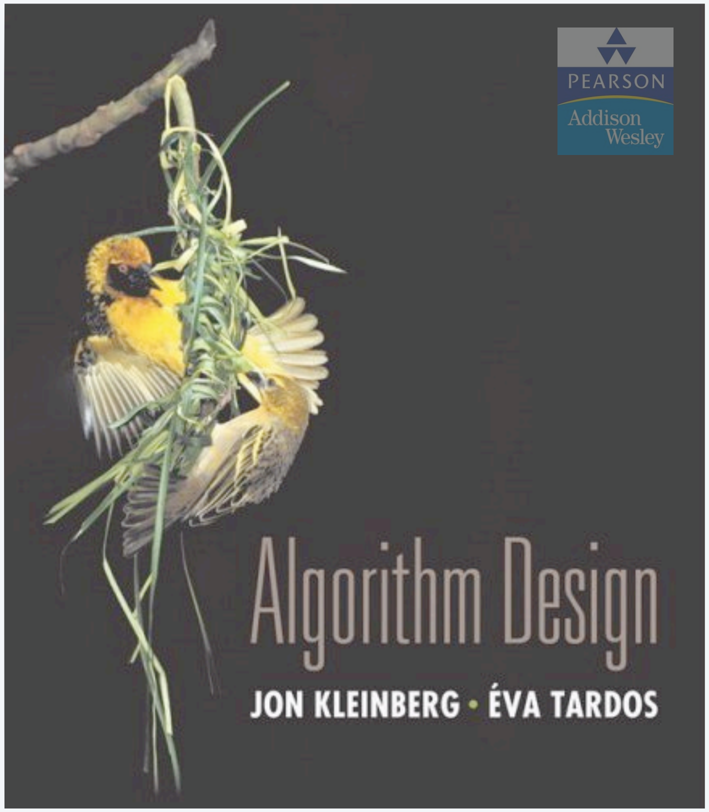
7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford-Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *blocking-flow algorithm*
- *unit-capacity simple networks*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on Sep 8, 2013 6:40 AM



SECTION 7.1

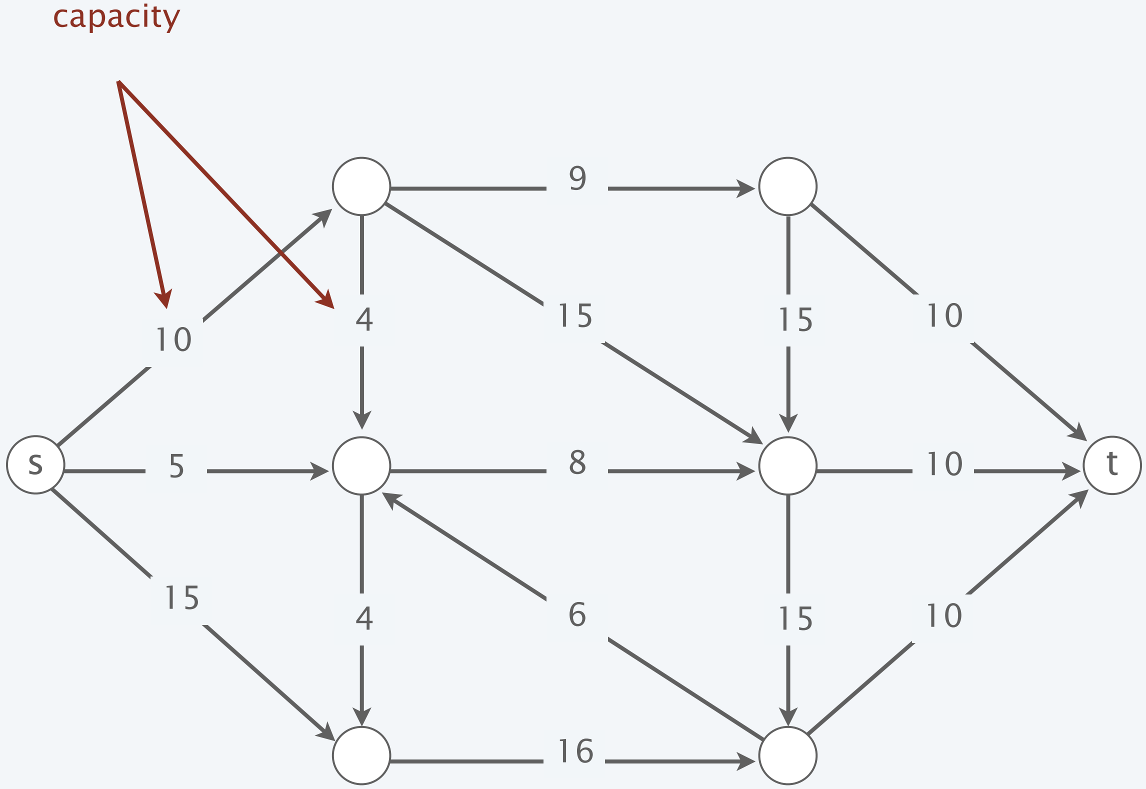
7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford-Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *blocking-flow algorithm*
- *unit-capacity simple networks*

Flow network

- Abstraction for material flowing through the edges.
- Digraph $G = (V, E)$ with source $s \in V$ and sink $t \in V$.
- Nonnegative integer capacity $c(e)$ for each $e \in E$.

no parallel edges
no edge enters s
no edge leaves t

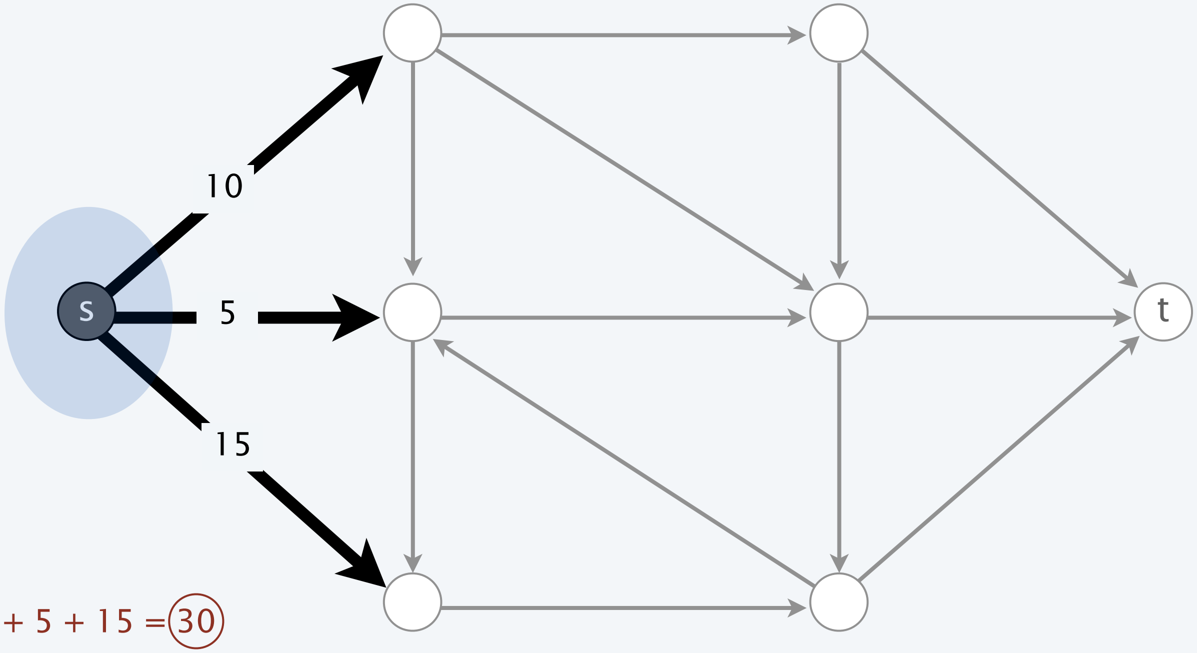


Minimum cut problem

Def. A *st*-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



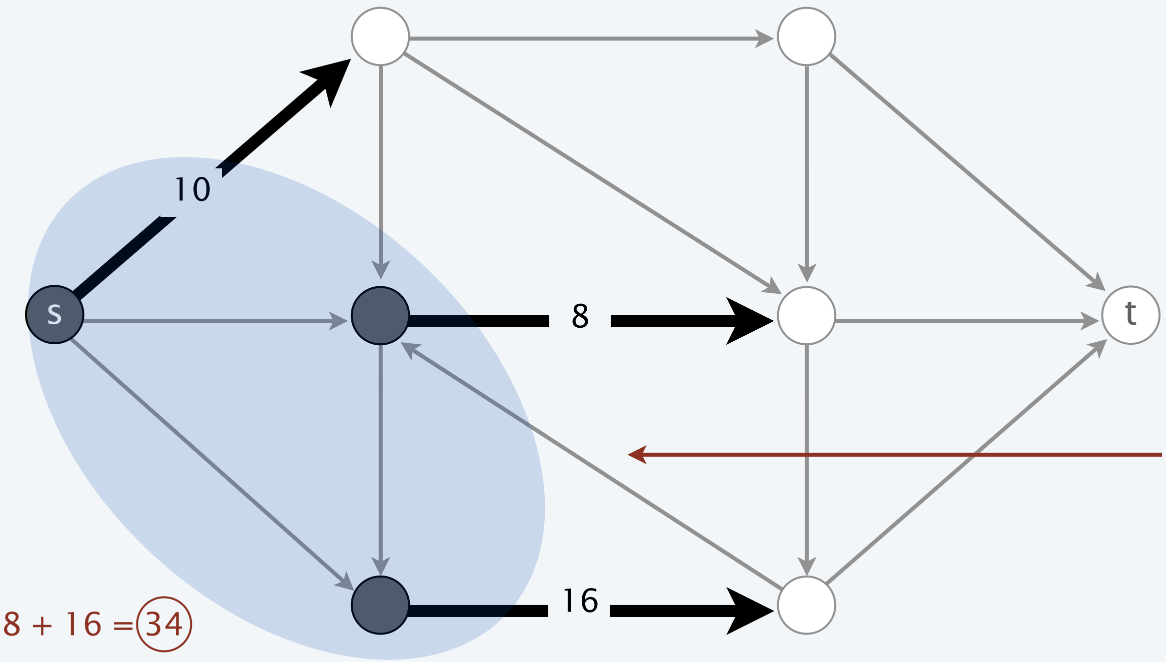
capacity = 10 + 5 + 15 = 30

Minimum cut problem

Def. A *st*-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



don't count edges from B to A

capacity = 10 + 8 + 16 = 34

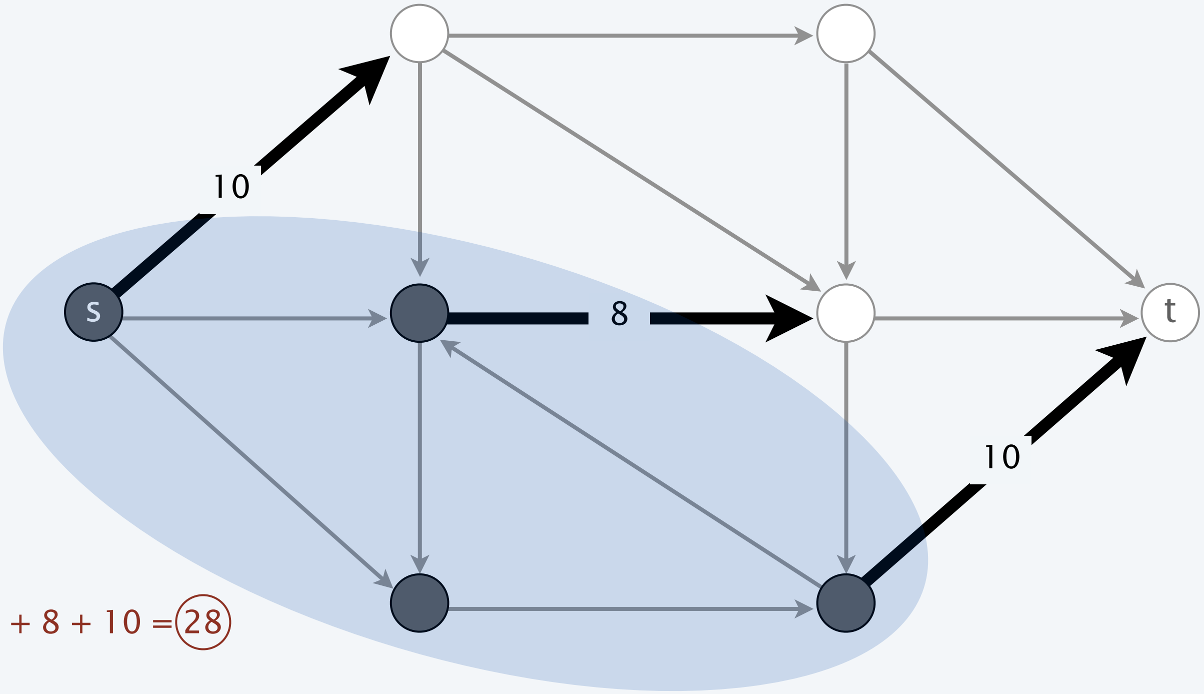
Minimum cut problem

Def. A st-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

Min-cut problem. Find a cut of minimum capacity.

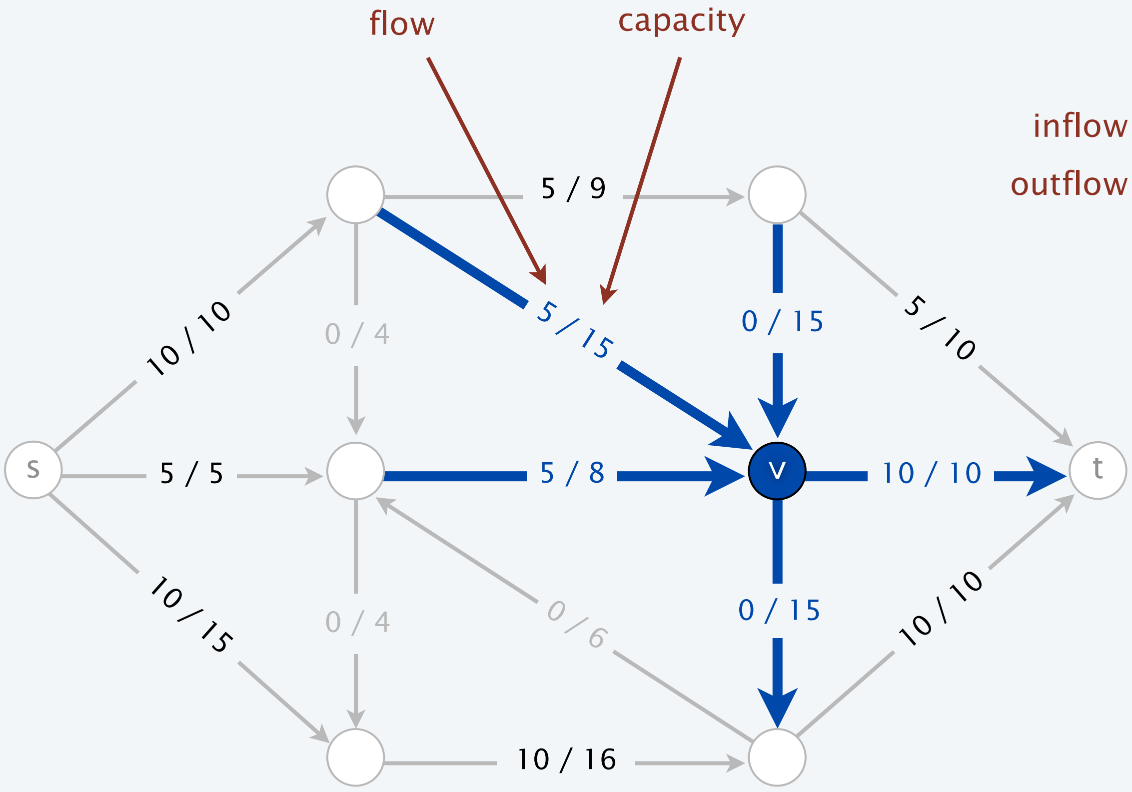


capacity = 10 + 8 + 10 = $\textcircled{28}$

Maximum flow problem

Def. An st-flow (flow) f is a function that satisfies:

- For each $e \in E$: $\quad 0 \le f(e) \le c(e) \quad$ [capacity]
- For each $v \in V - \{s, t\}$: $\quad \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e) \quad$ [flow conservation]



inflow at v = 5 + 5 + 0 = 10

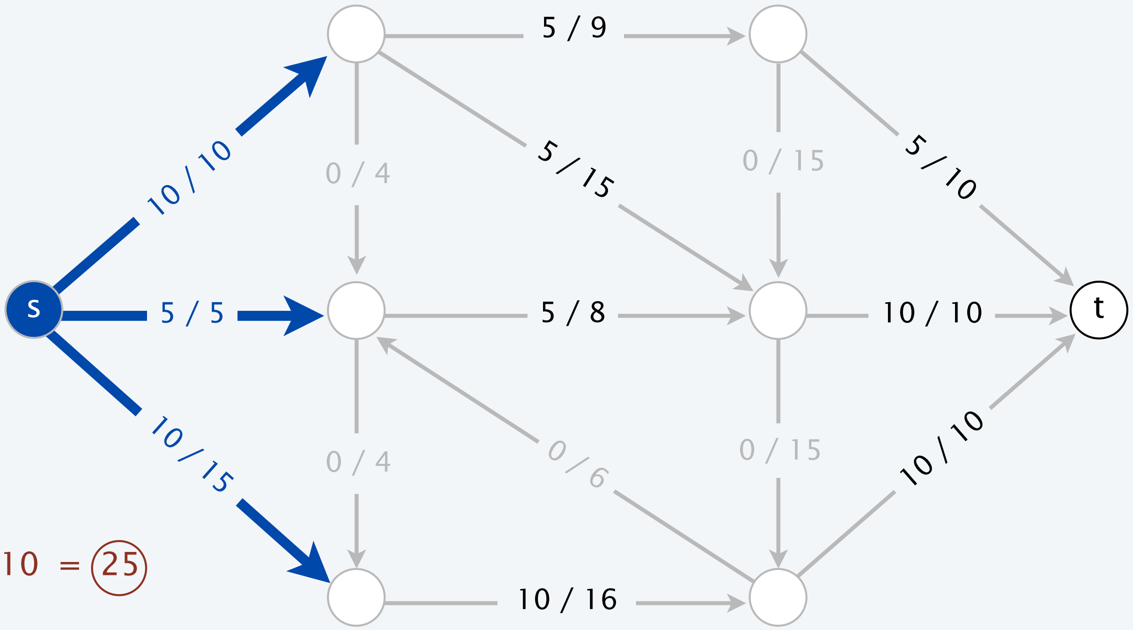
outflow at v = 10 + 0 = 10

Maximum flow problem

Def. An *st*-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e)$.



value = 5 + 10 + 10 = 25

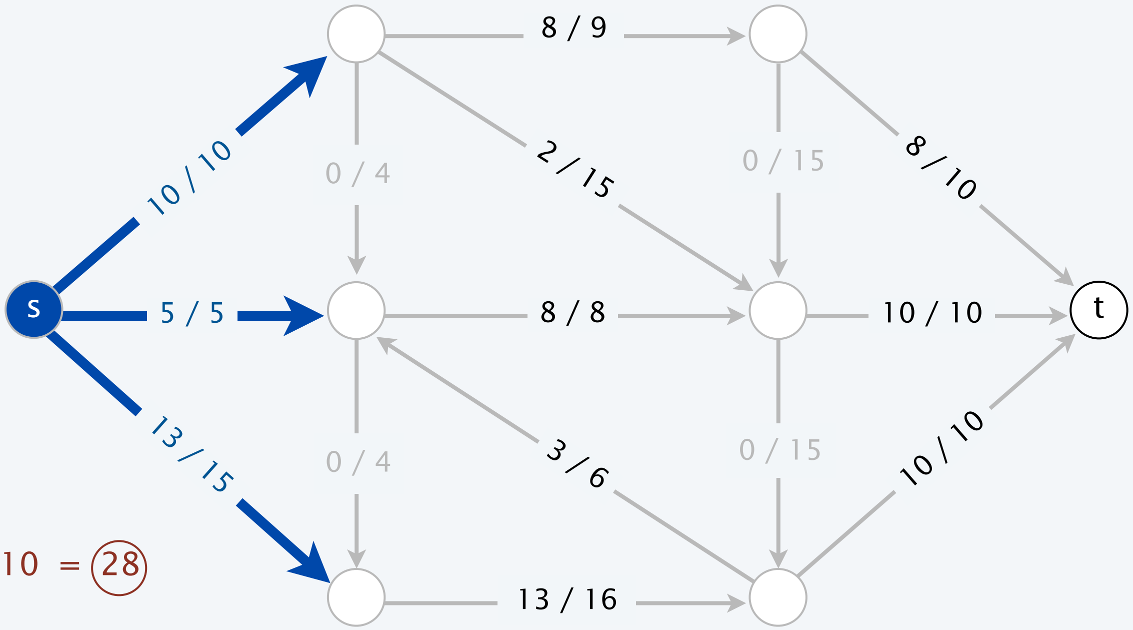
Maximum flow problem

Def. An *st*-flow (flow) f is a function that satisfies:

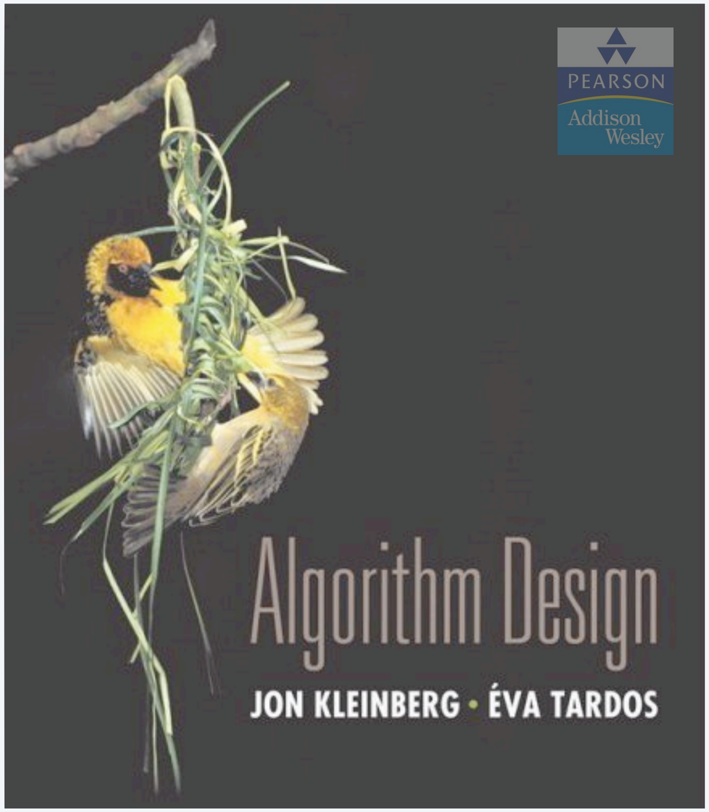
- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e)$.

Max-flow problem. Find a flow of maximum value.



value = 8 + 10 + 10 = 28



SECTION 7.1

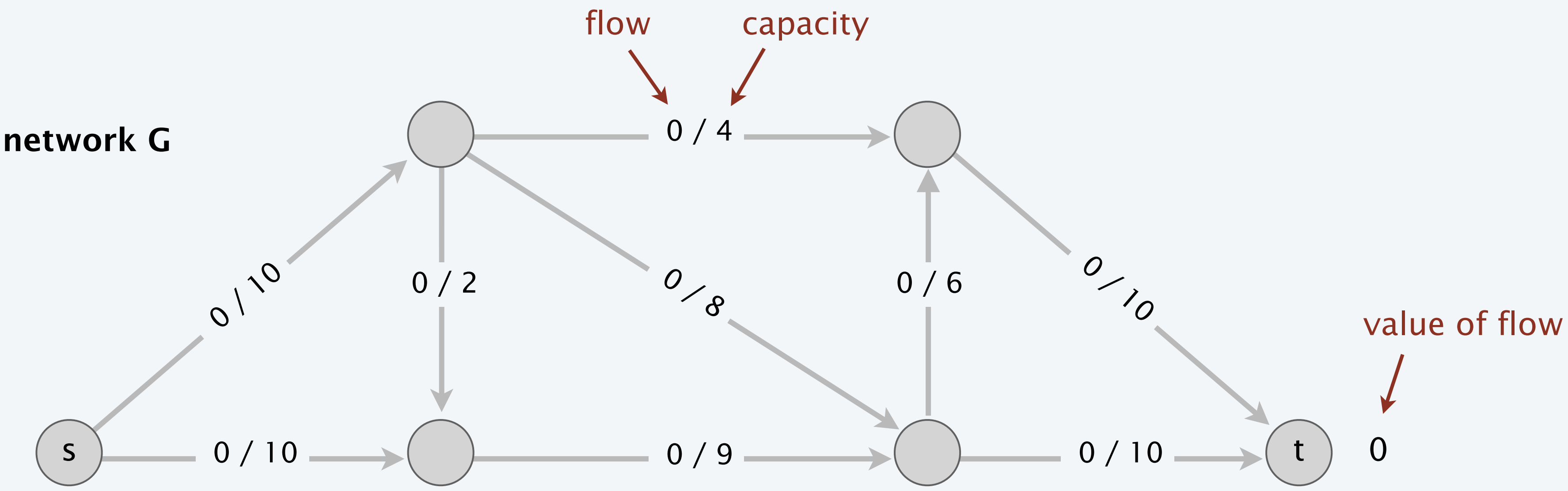
7. NETWORK FLOW I

- *max-flow and min-cut problems*
- ***Ford-Fulkerson algorithm***
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *blocking-flow algorithm*
- *unit-capacity simple networks*

Towards a max-flow algorithm

Greedy algorithm.

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

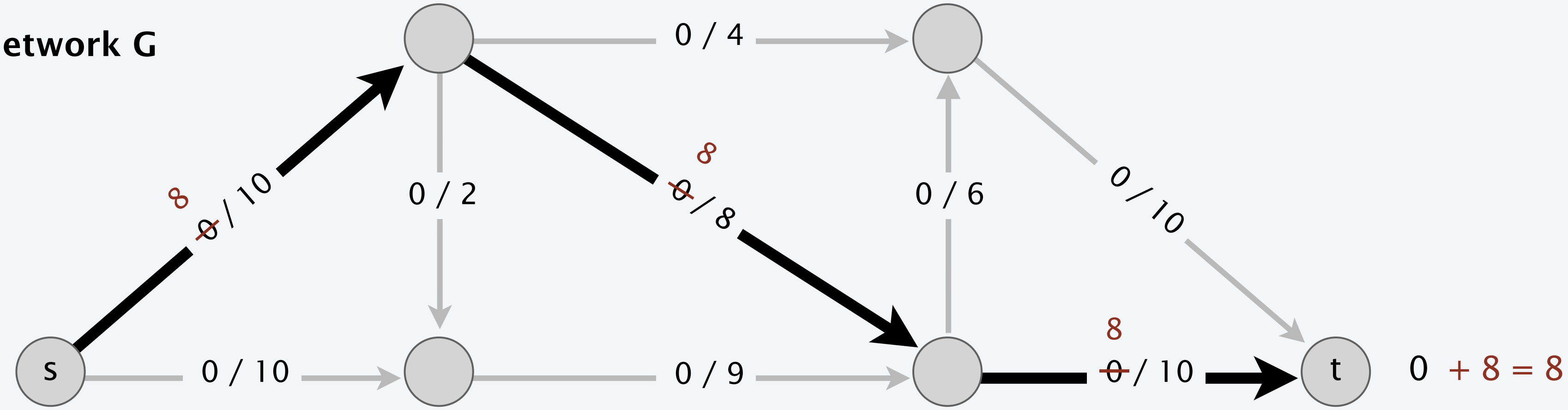


Towards a max-flow algorithm

Greedy algorithm.

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

network G

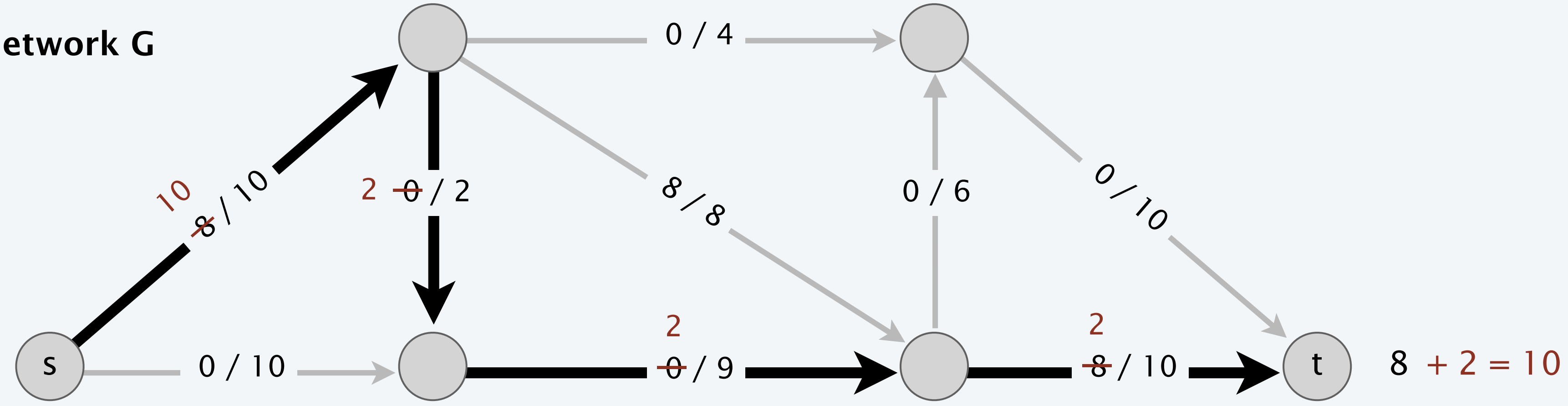


Towards a max-flow algorithm

Greedy algorithm.

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

network G

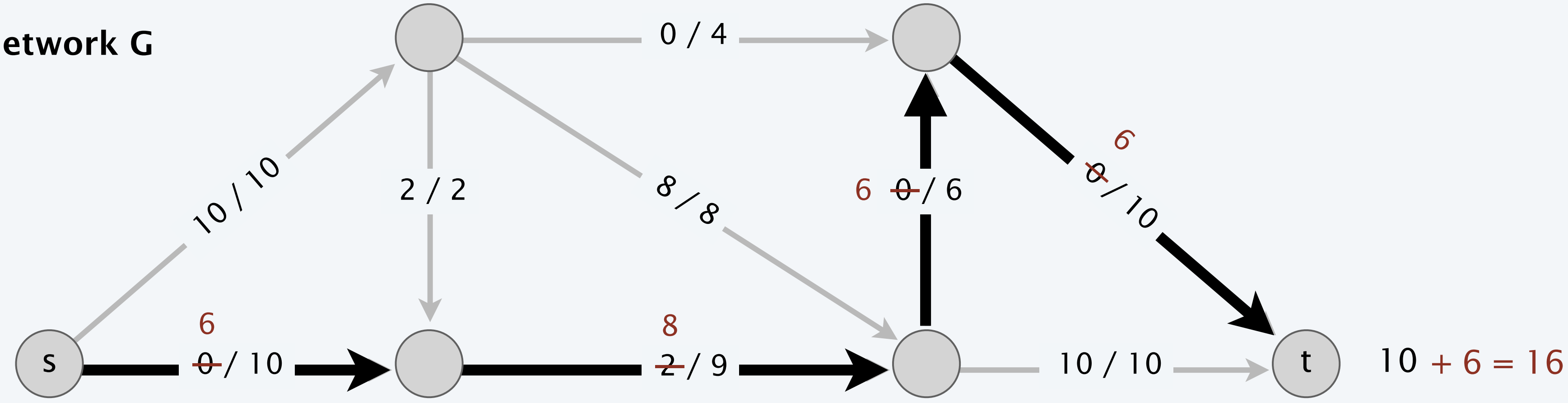


Towards a max-flow algorithm

Greedy algorithm.

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

network G



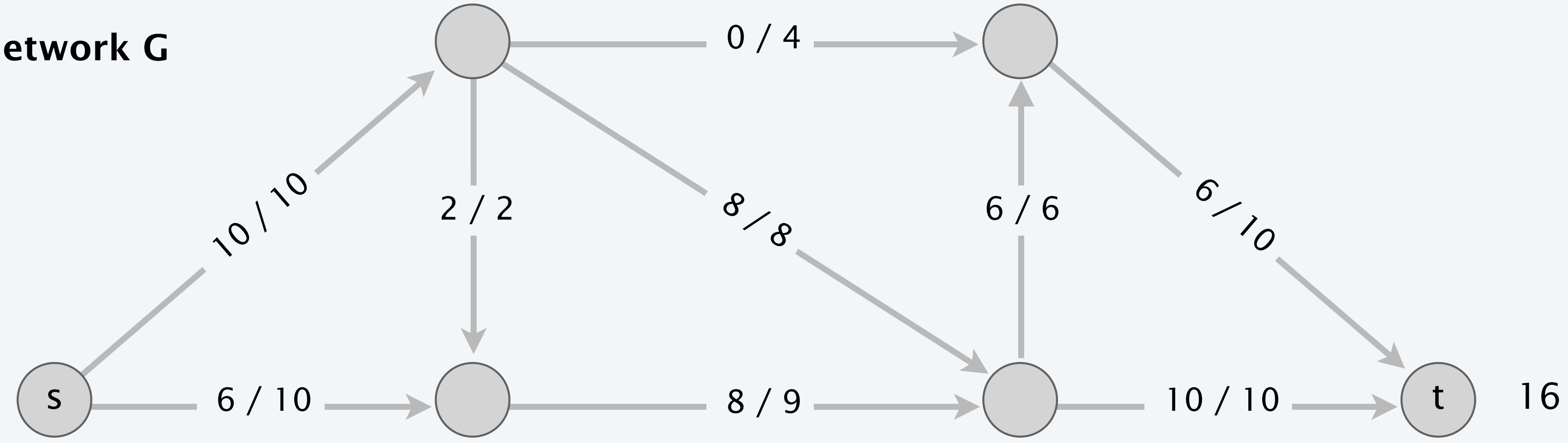
Towards a max-flow algorithm

Greedy algorithm.

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

ending flow value = 16

network G



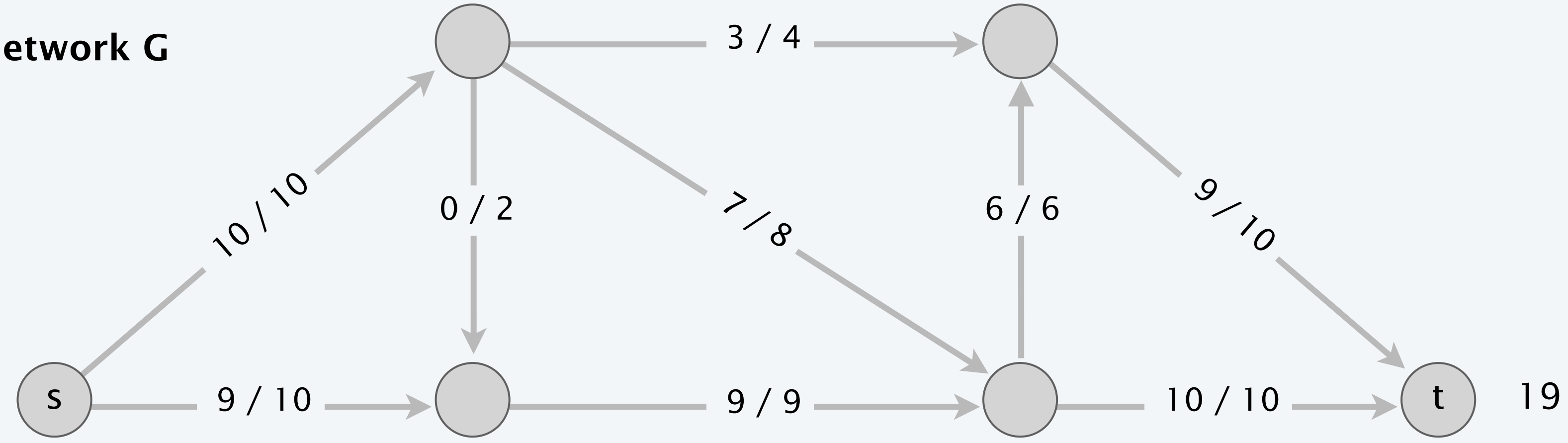
Towards a max-flow algorithm

Greedy algorithm.

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

but max-flow value = 19

network G

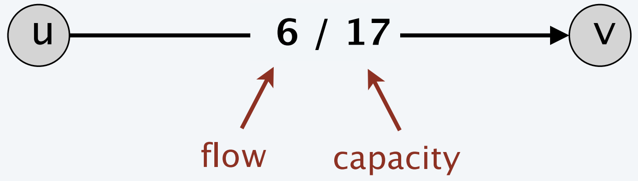


Residual graph

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

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

original graph G

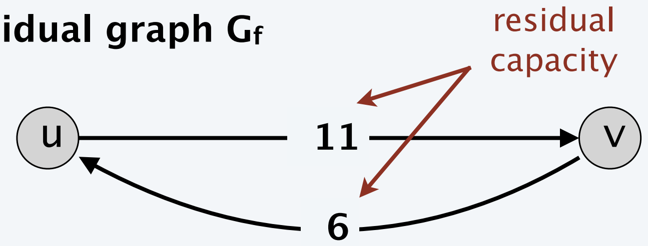


Residual edge.

- "Undo" flow sent.
- $e = (u, v)$ and $e^R = (v, u)$.
- Residual capacity:

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

residual graph G_f



Residual graph: $G_f = (V, E_f)$.

- Residual edges with positive residual capacity.
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$.
- Key property: f' is a flow in G_f iff $f + f'$ is a flow in G.

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

Augmenting path

Def. An augmenting path is a simple $s \rightsquigarrow t$ path P in the residual graph G_f.

Def. The bottleneck capacity of an augmenting P is the minimum residual capacity of any edge in P.

Key property. Let f be a flow and let P be an augmenting path in G_f. Then f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$.

```
AUGMENT ($f$, $c$, $P$)
$b$ ← bottleneck capacity of path $P$.
FOREACH edge $e \in P$
    IF ($e \in E$) $f(e) \leftarrow f(e) + b$.
    ELSE           $f(e^R) \leftarrow f(e^R) - b$.
RETURN $f$.
```

Ford-Fulkerson algorithm

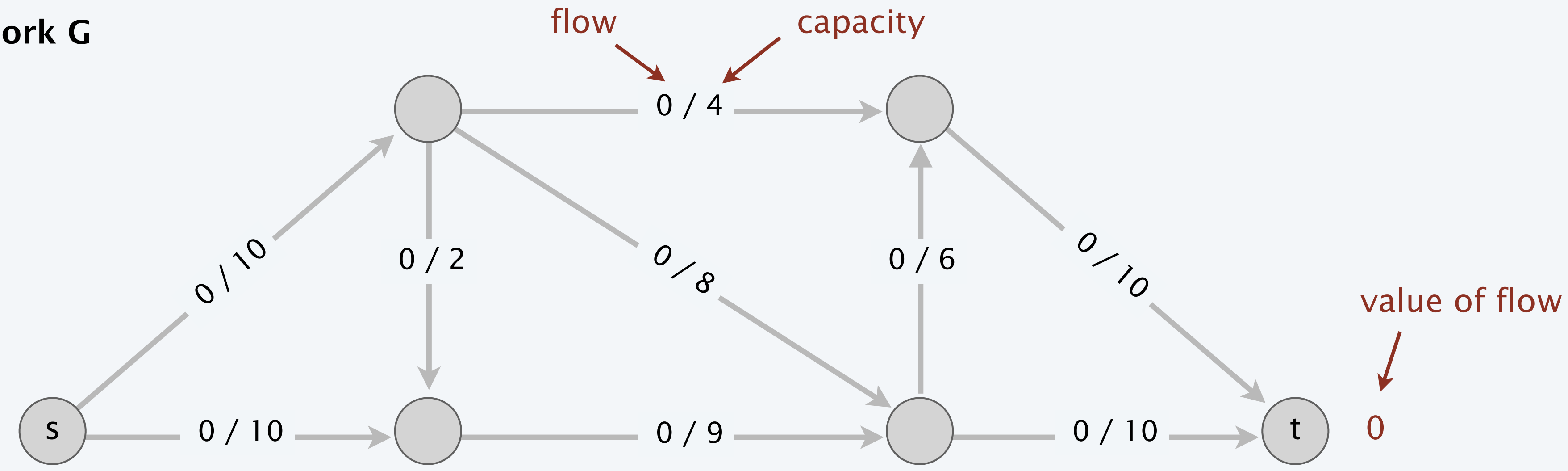
Ford-Fulkerson augmenting path algorithm.

- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an augmenting path P in the residual graph G_f.
- Augment flow along path P.
- Repeat until you get stuck.

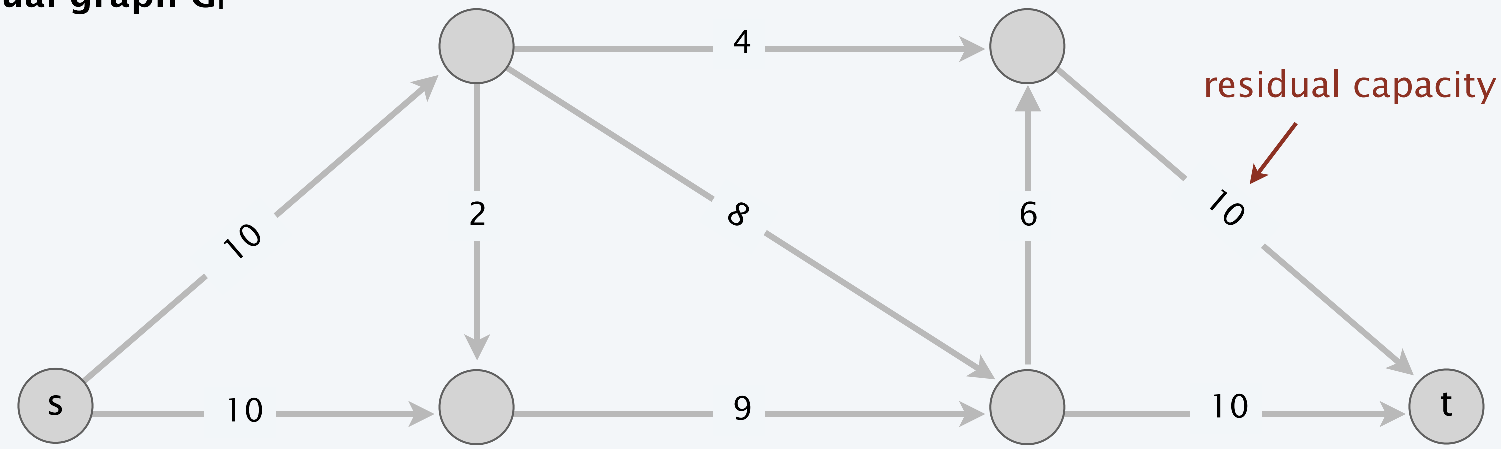
```
FORD-FULKERSON $(G, s, t, c)$
_________________________________
FOREACH edge $e \in E : f(e) \leftarrow 0$.
    $G_f \leftarrow$ residual graph.
    WHILE (there exists an augmenting path $P$ in $G_f$)
        $f \leftarrow$ AUGMENT $(f, c, P)$.
        Update $G_f$.
    RETURN $f$.
}
_________________________________
```

Ford-Fulkerson algorithm demo

network G

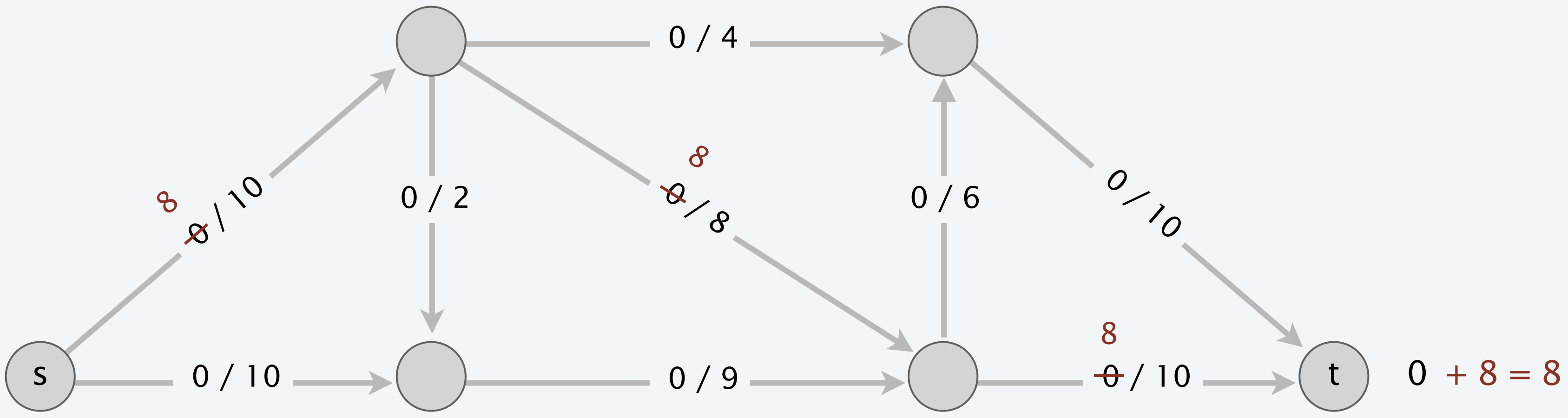


residual graph G_f

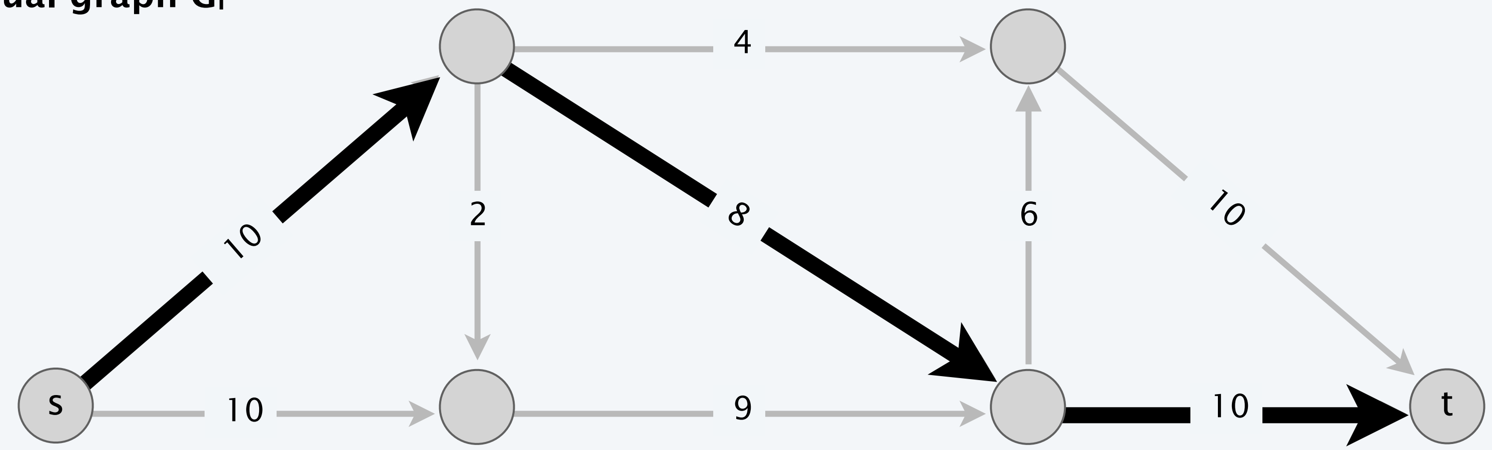


Ford-Fulkerson algorithm demo

network G

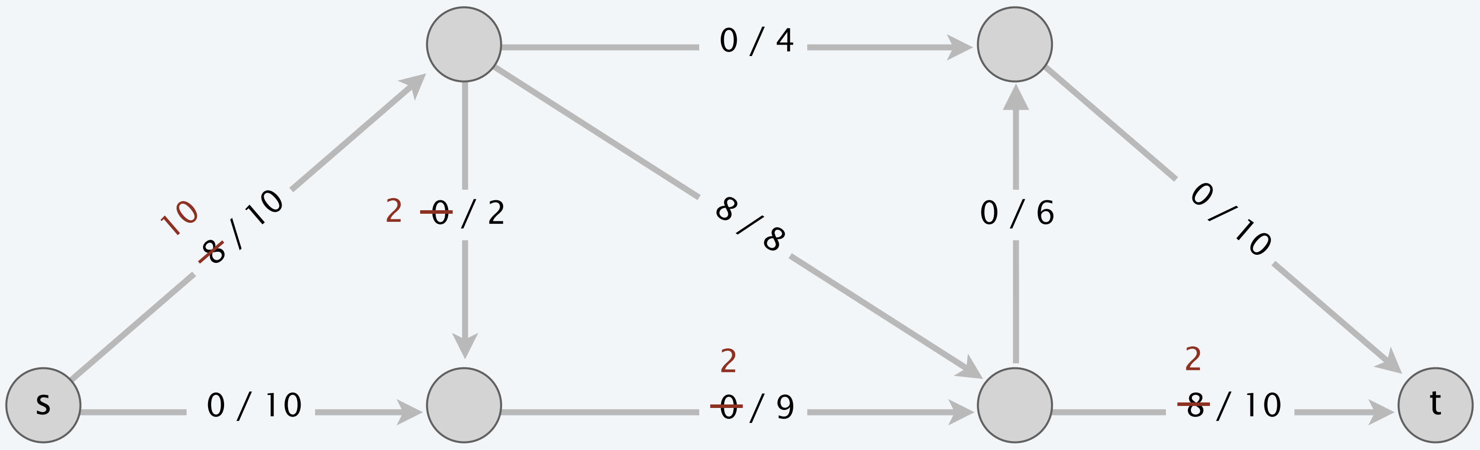


residual graph G_f



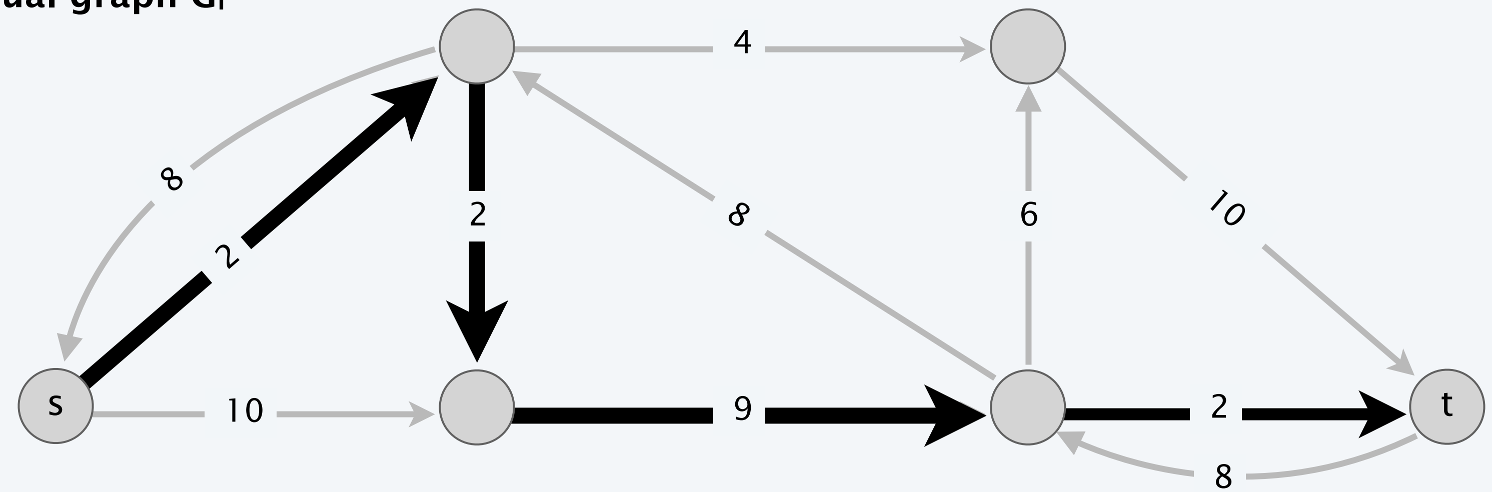
Ford-Fulkerson algorithm demo

network G



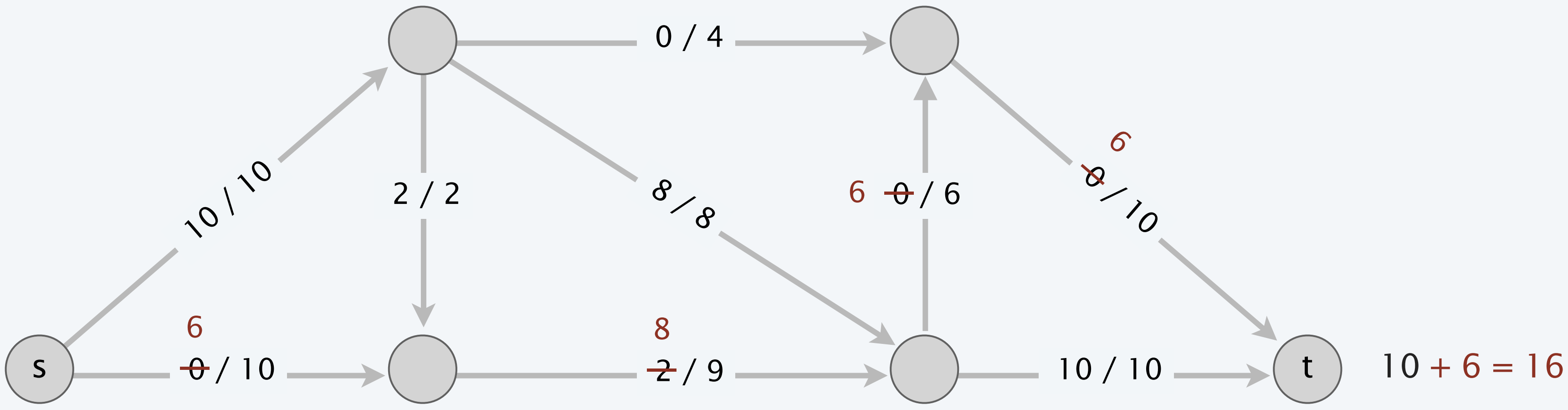
8 + 2 = 10

residual graph G_f

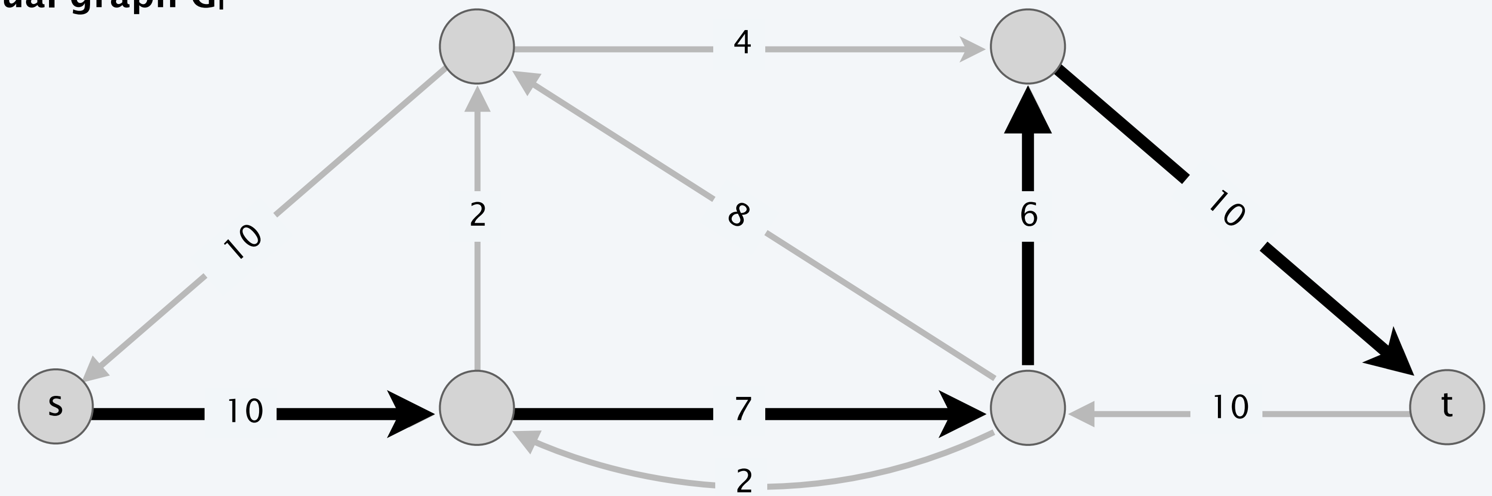


Ford-Fulkerson algorithm demo

network G

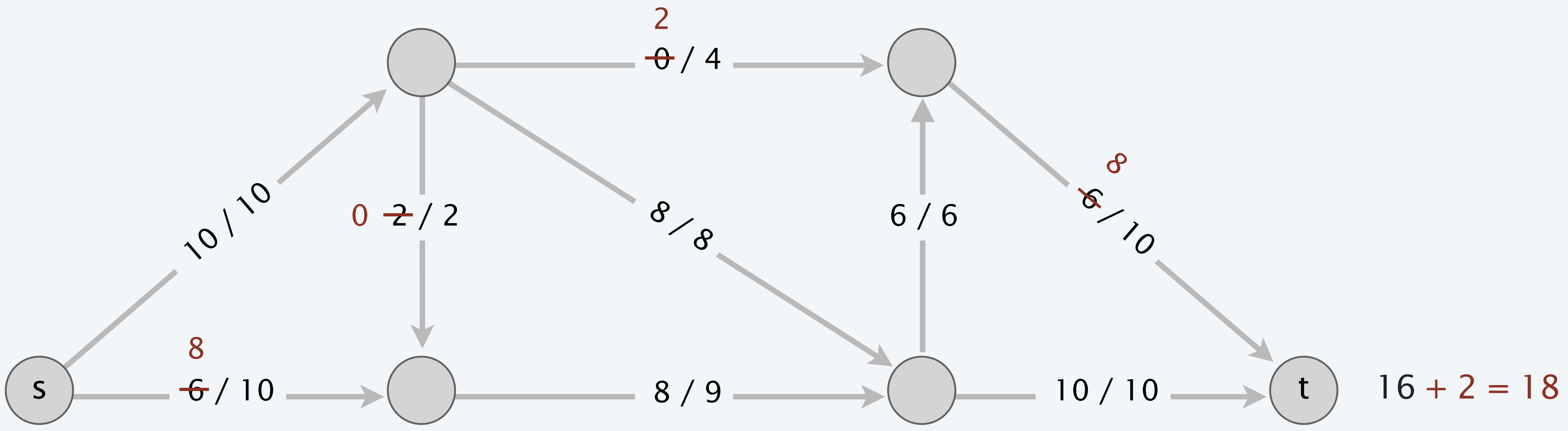


residual graph G$_f$

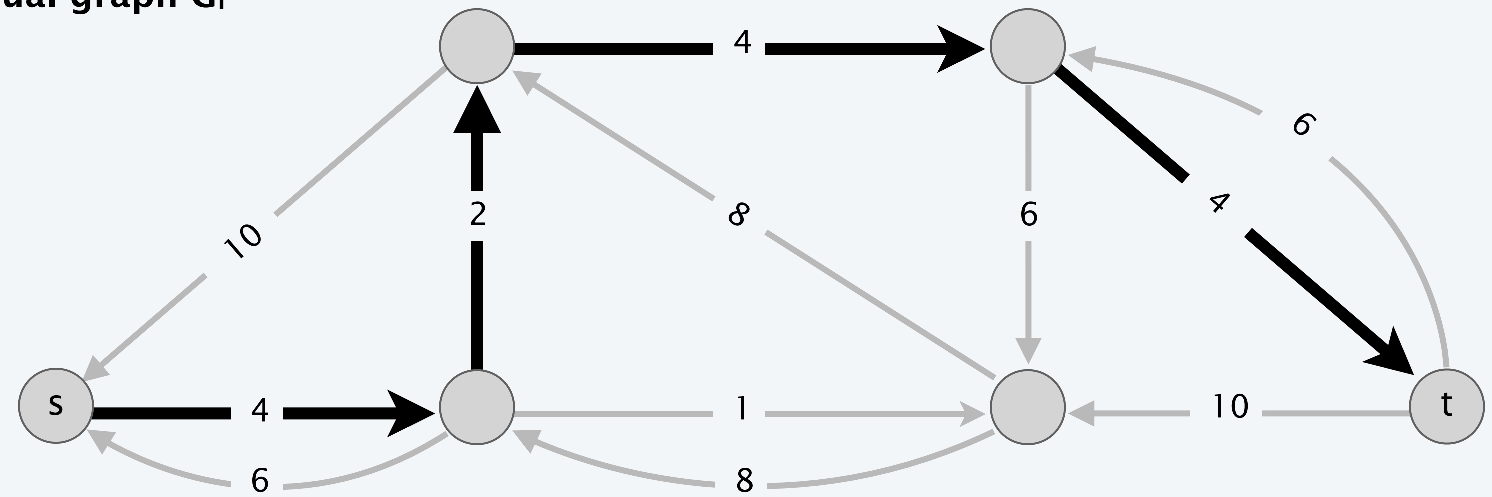


Ford-Fulkerson algorithm demo

network G

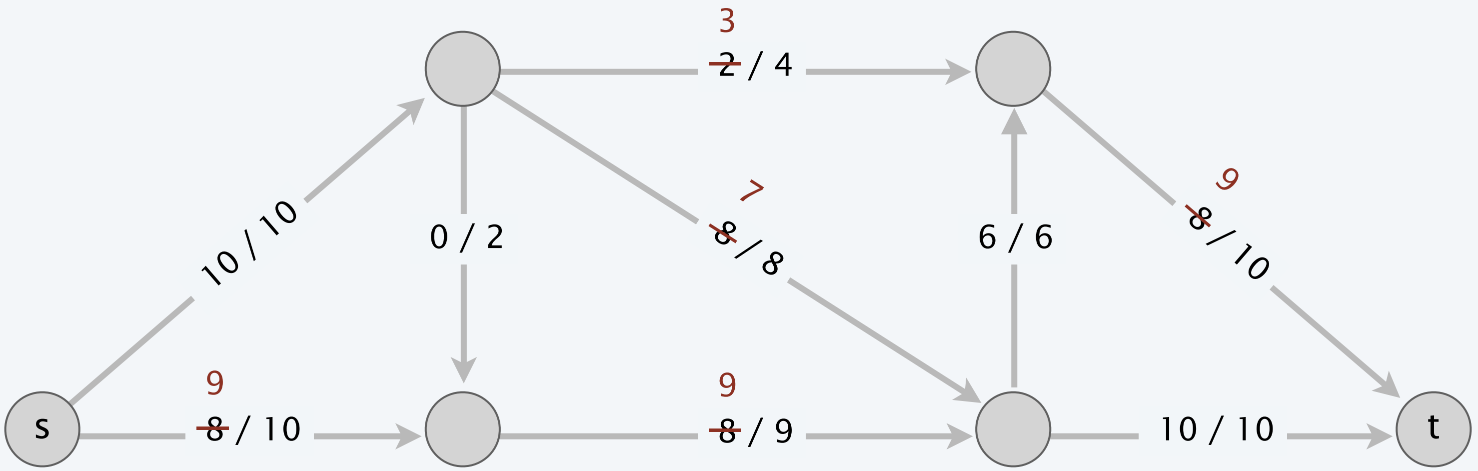


residual graph G_f



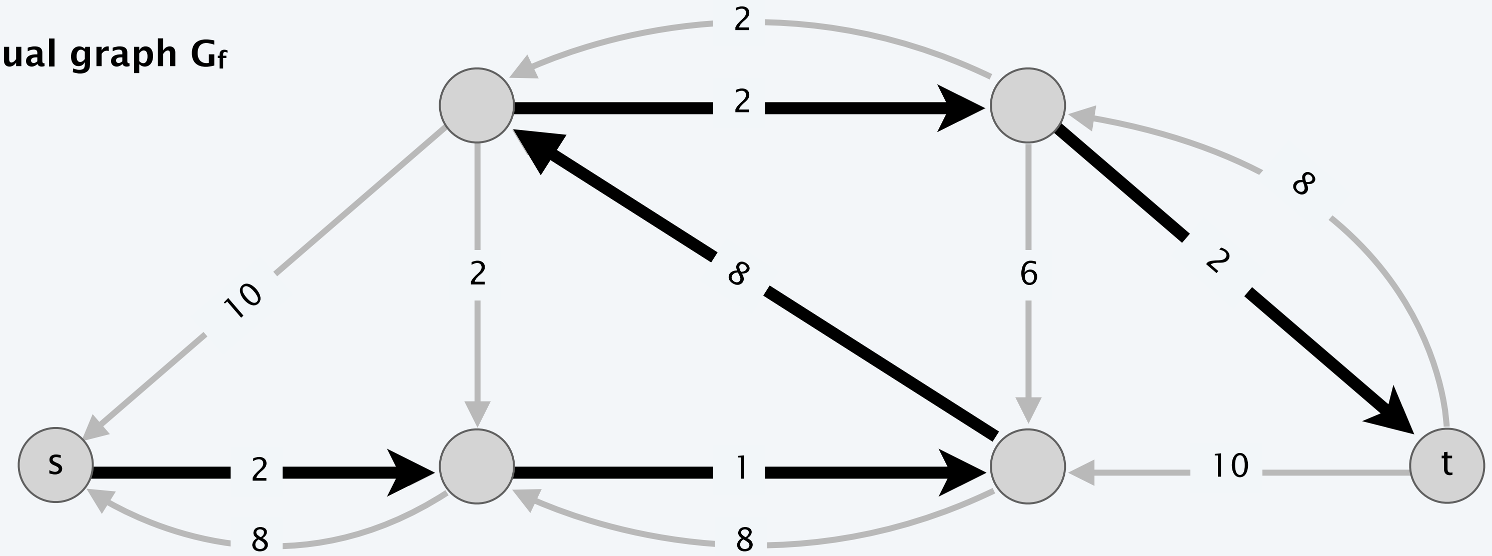
Ford-Fulkerson algorithm demo

network G



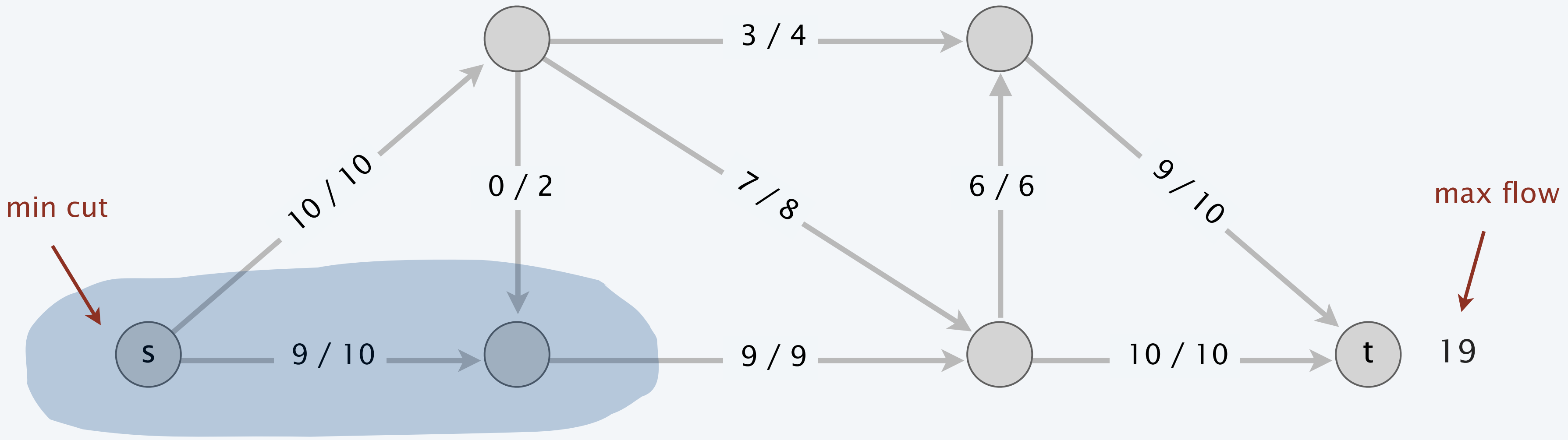
18 + 1 = 19

residual graph G_f

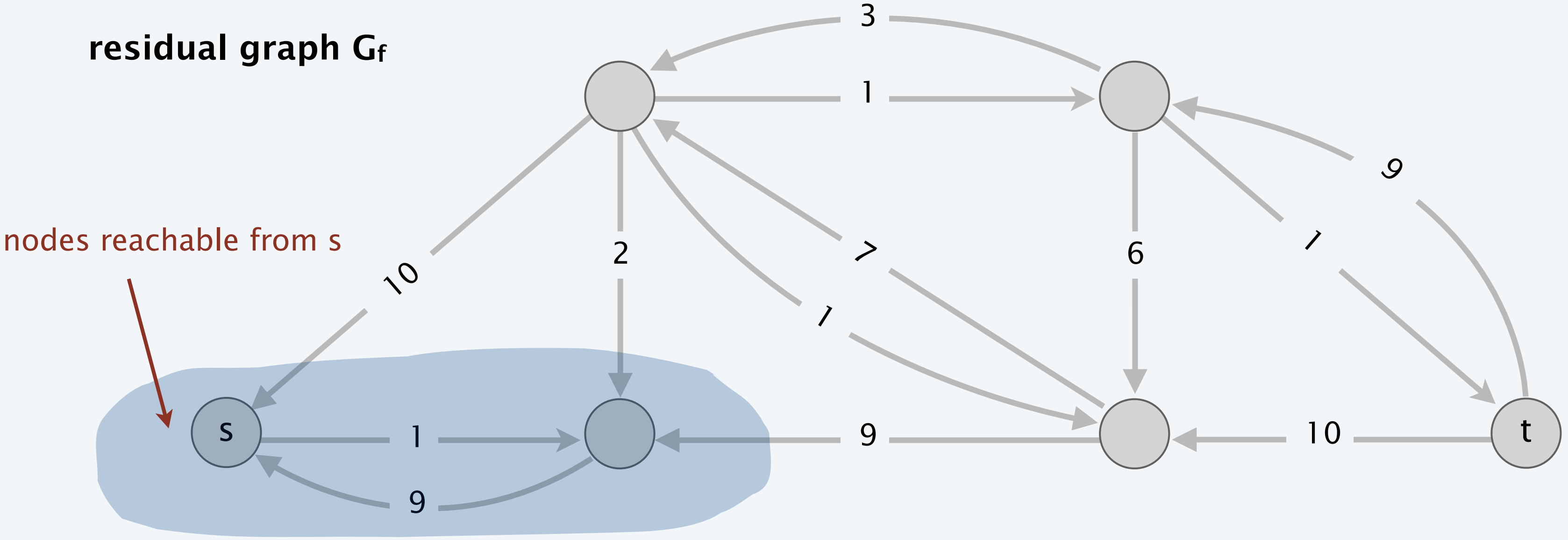


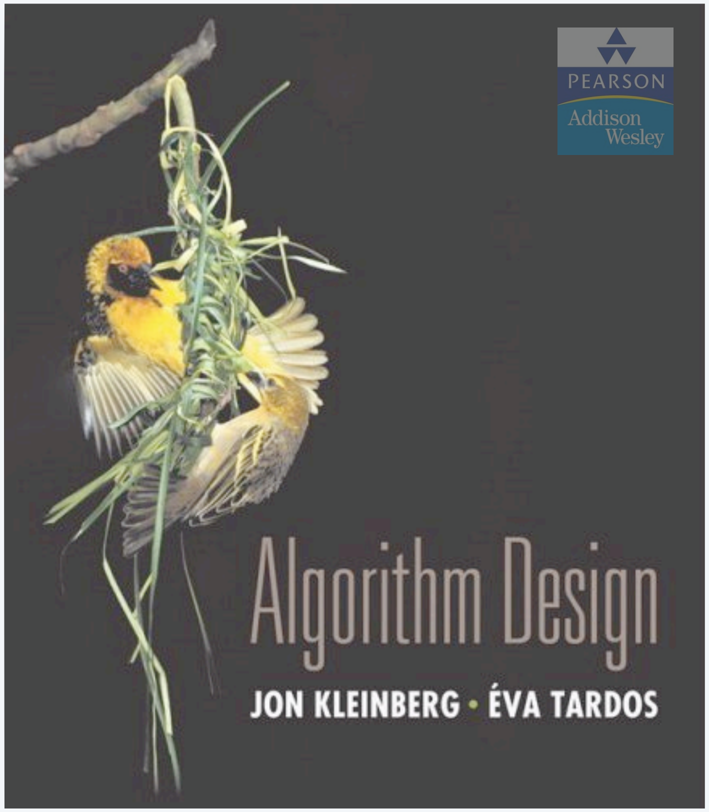
Ford-Fulkerson algorithm demo

network G



residual graph G$_f$





SECTION 7.2

7. NETWORK FLOW I

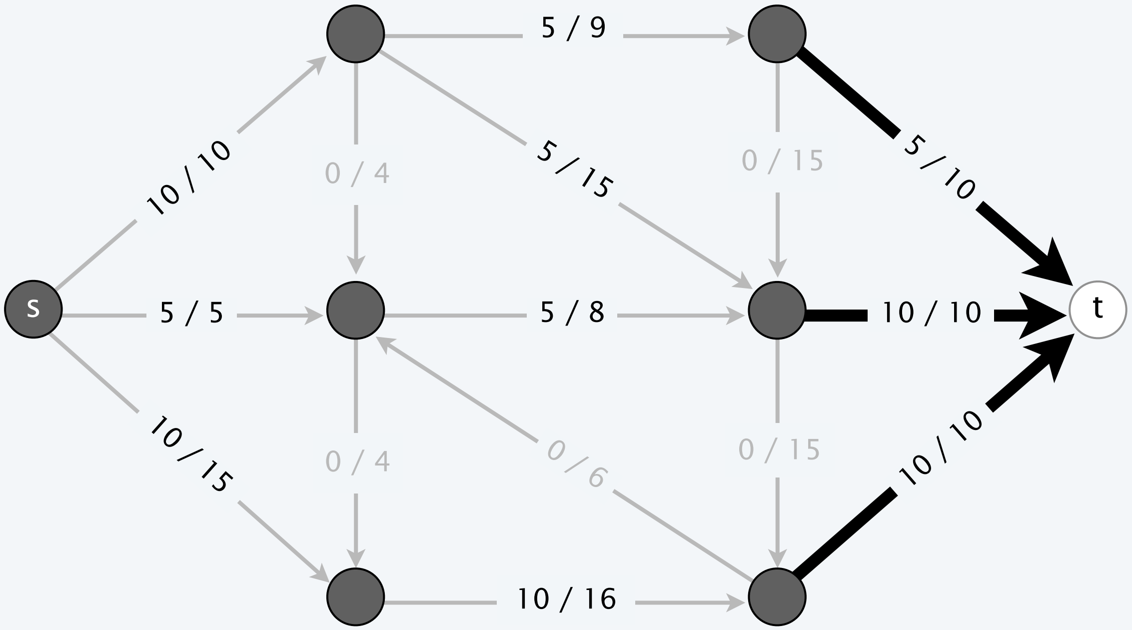
- max-flow and min-cut problems
- Ford-Fulkerson algorithm
- *max-flow min-cut theorem*
- capacity-scaling algorithm
- shortest augmenting paths
- blocking-flow algorithm
- unit-capacity simple networks

Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.

$$\sum_{e \text{ out of } A} f(e) \; - \sum_{e \text{ in to } A} f(e) \; = \; v(f)$$

net flow across cut = 5 + 10 + 10 = 25



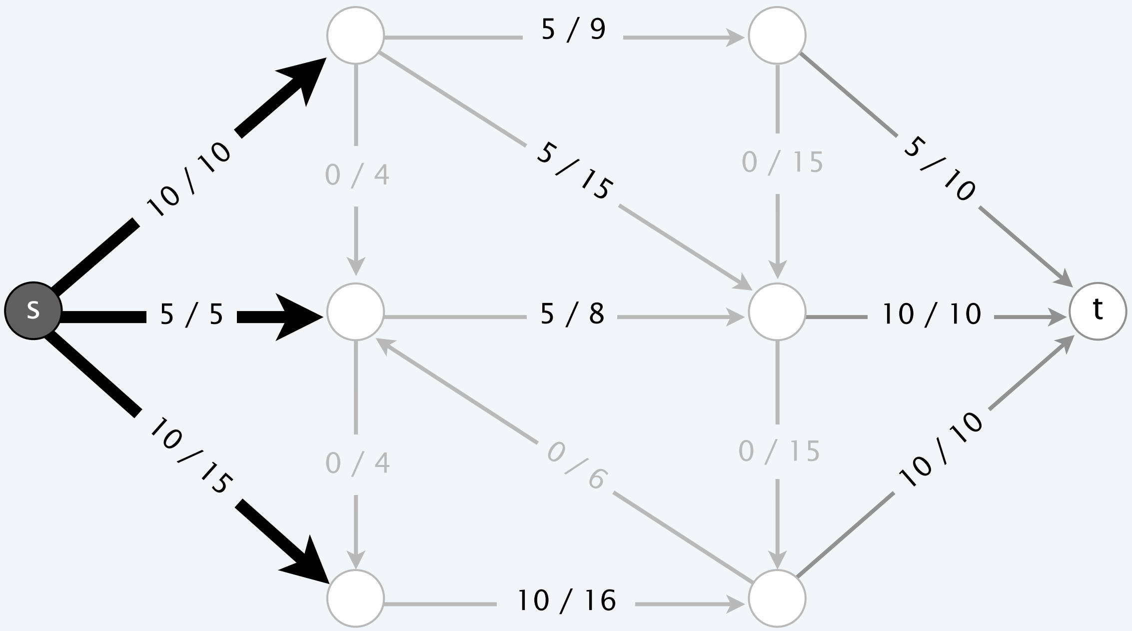
value of flow = 25

Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.

$$\sum_{e \text{ out of } A} f(e) \; - \sum_{e \text{ in to } A} f(e) \; = \; v(f)$$

net flow across cut = 10 + 5 + 10 = 25



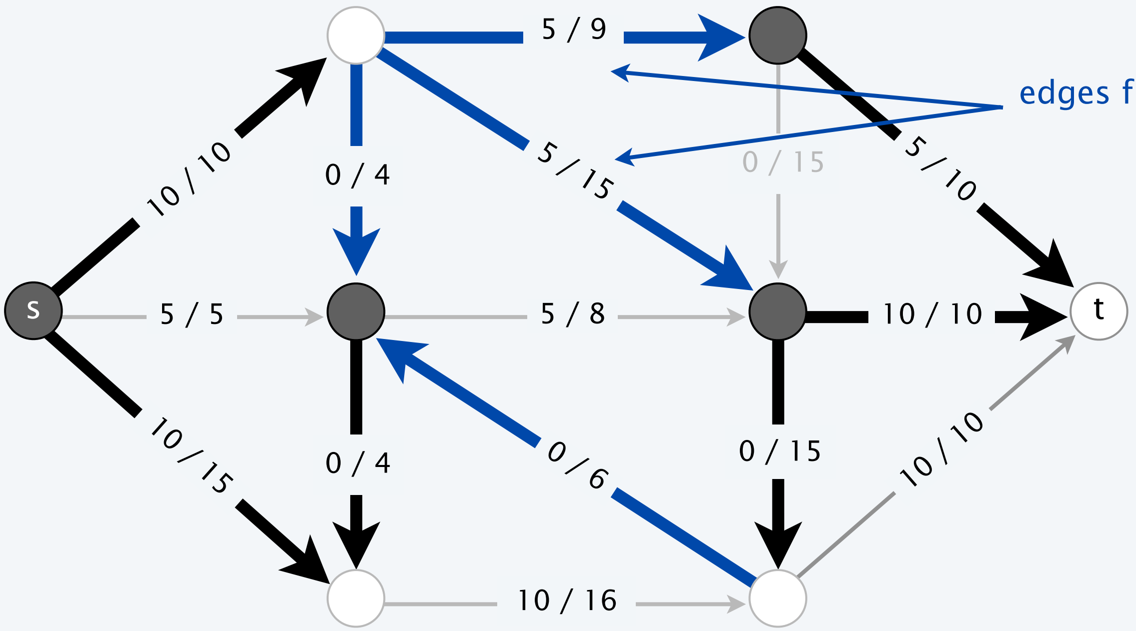
value of flow = 25

Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.

$$\sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to } A} f(e) \;=\; v(f)$$

net flow across cut = (10 + 10 + 5 + 10 + 0 + 0) – (5 + 5 + 0 + 0) = 25



value of flow = 25

Relationship between flows and cuts

Flow value lemma. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.

$$\sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to A}} f(e) \;=\; v(f)$$

Pf.

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

by flow conservation, all terms except v = s are 0 ⟶

$$= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to v}} f(e) \right)$$

$$= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to A}} f(e). \quad ▪$$

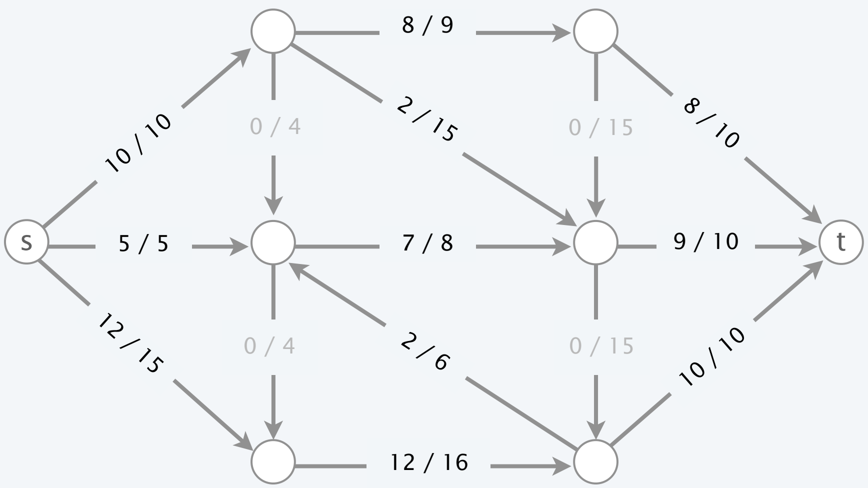
Relationship between flows and cuts

Weak duality. Let f be any flow and (A, B) be any cut. Then, $v(f) \leq cap(A, B)$.

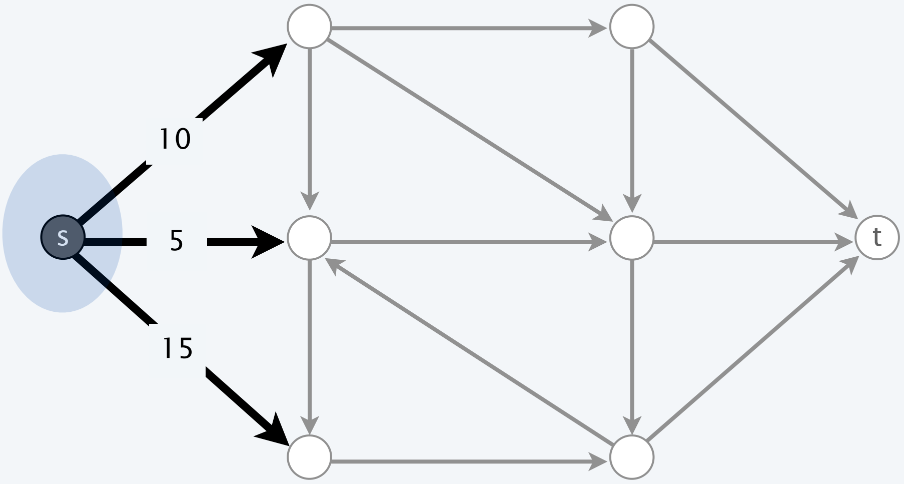
Pf.

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= \text{cap}(A,B) \quad \blacksquare \end{aligned}$$

flow-value lemma



value of flow = 27 ≤



capacity of cut = 30

Max-flow min-cut theorem

Augmenting path theorem. A flow f is a max-flow iff no augmenting paths.

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

Pf. The following three conditions are equivalent for any flow f:

i. There exists a cut (A, B) such that $cap(A, B) = val(f)$.
ii. f is a max-flow.
iii. There is no augmenting path with respect to f.

[i ⇒ ii]

- Suppose that (A, B) is a cut such that $cap(A, B) = val(f)$.
- Then, for any flow f', $val(f') \underset{\substack{\uparrow \\ \text{weak duality}}}{\leq} cap(A, B) \underset{\substack{\uparrow \\ \text{by assumption}}}{=} val(f)$.
- Thus, f is a max-flow. ▪

Max-flow min-cut theorem

Augmenting path theorem. A flow f is a max-flow iff no augmenting paths.

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

Pf. The following three conditions are equivalent for any flow f:

i. There exists a cut (A, B) such that $cap(A, B) = val(f)$.

ii. f is a max-flow.

iii. There is no augmenting path with respect to f.

[ii ⇒ iii] We prove contrapositive: ~iii ⇒ ~ii.

- Suppose that there is an augmenting path with respect to f.
- Can improve flow f by sending flow along this path.
- Thus, f is not a max-flow. ▪

Max-flow min-cut theorem

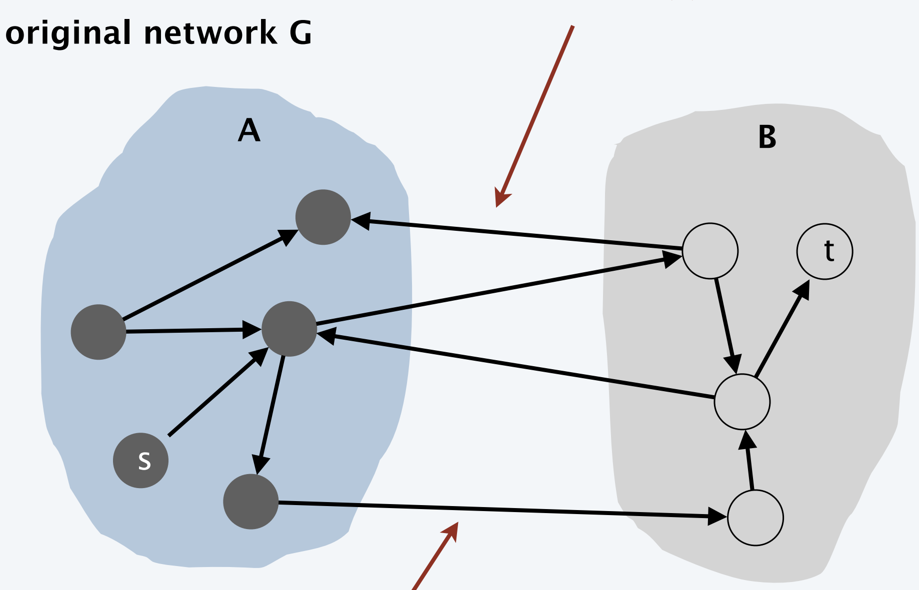
[iii ⇒ i]

- Let f be a flow with no augmenting paths.
- Let A be set of nodes reachable from s in residual graph G_f.
- By definition of cut A, $s \in A$.
- By definition of flow f, $t \notin A$.

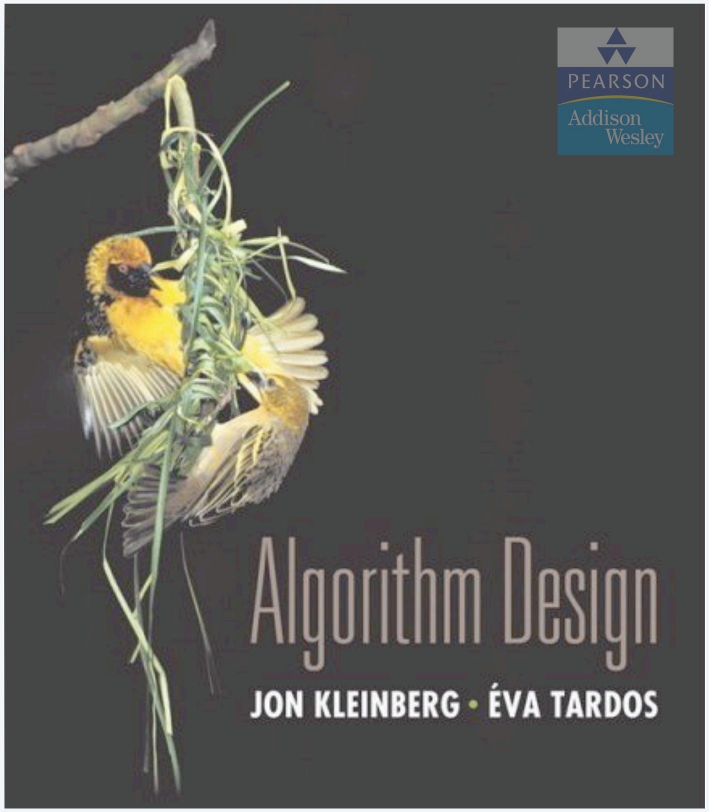
$$
\begin{aligned}
v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to A}} f(e) \\
&= \sum_{e \text{ out of } A} c(e) \\
&= cap(A, B) \quad ▪
\end{aligned}
$$

flow-value lemma

edge e = (v, w) with v ∈ B, w ∈ A
must have f(e) = 0



edge e = (v, w) with v ∈ A, w ∈ B
must have f(e) = c(e)



SECTION 7.3

7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford-Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *blocking-flow algorithm*
- *unit-capacity simple networks*

Running time

Assumption. Capacities are integers between 1 and C.

Integrality invariant. Throughout the algorithm, the flow values $f(e)$ and the residual capacities $c_f(e)$ are integers.

Theorem. The algorithm terminates in at most $val(f^*) \leq nC$ iterations.
Pf. Each augmentation increases the value by at least 1. ▪

Corollary. The running time of Ford-Fulkerson is $O(mnC)$.
Corollary. If $C = 1$, the running time of Ford-Fulkerson is $O(mn)$.

Integrality theorem. Then exists a max-flow f^* for which every flow value $f^*(e)$ is an integer.
Pf. Since algorithm terminates, theorem follows from invariant. ▪

Bad case for Ford-Fulkerson

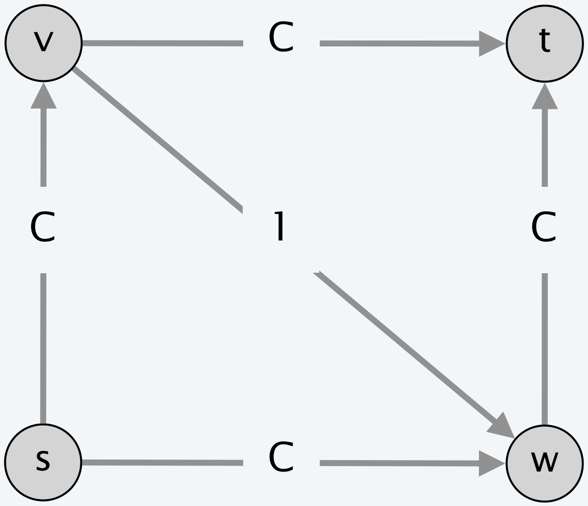
Q. Is generic Ford-Fulkerson algorithm poly-time in input size?

m, n, and log C

A. No. If max capacity is C, then algorithm can take ≥ C iterations.

- $s \to v \to w \to t$
- $s \to w \to v \to t$
- $s \to v \to w \to t$
- $s \to w \to v \to t$
- …
- $s \to v \to w \to t$
- $s \to w \to v \to t$

each augmenting path
sends only 1 unit of flow
(# augmenting paths = 2C)



Choosing good augmenting paths

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.
- If capacities are irrational, algorithm not guaranteed to terminate!

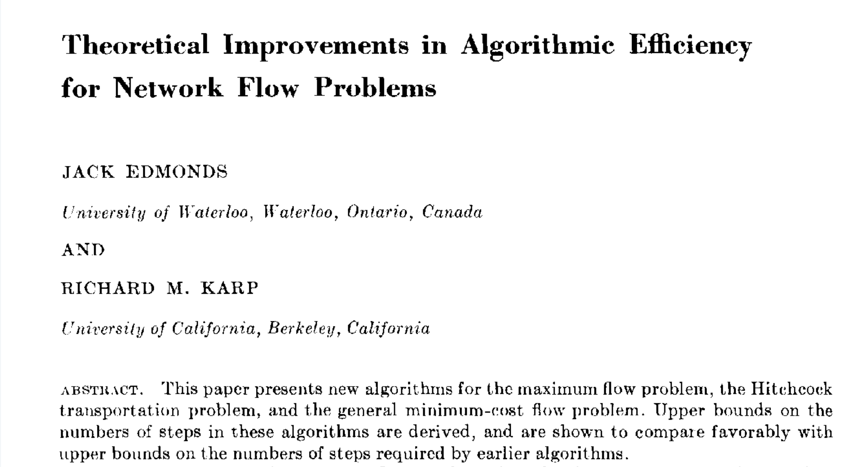
Goal. Choose augmenting paths so that:

- Can find augmenting paths efficiently.
- Few iterations.

Choosing good augmenting paths

Choose augmenting paths with:

- Max bottleneck capacity.
- Sufficiently large bottleneck capacity.
- Fewest number of edges.



Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems

JACK EDMONDS

University of Waterloo, Waterloo, Ontario, Canada

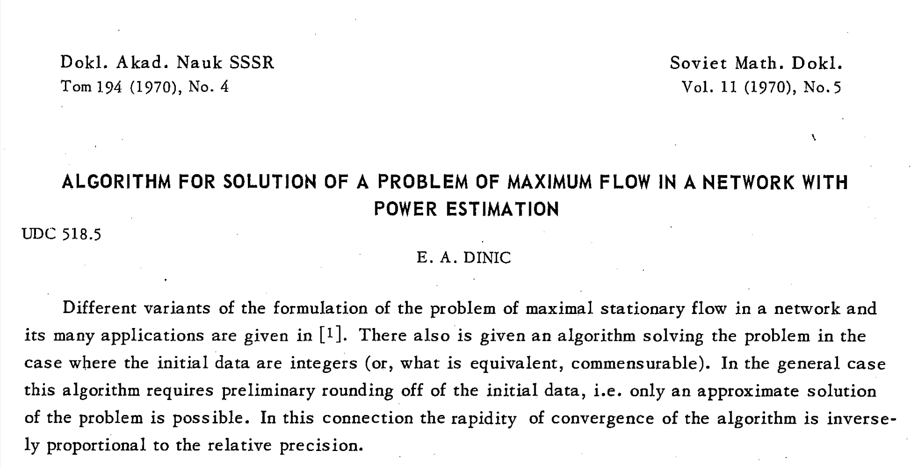
AND

RICHARD M. KARP

University of California, Berkeley, California

ABSTRACT. This paper presents new algorithms for the maximum flow problem, the Hitchcock transportation problem, and the general minimum-cost flow problem. Upper bounds on the numbers of steps in these algorithms are derived, and are shown to compare favorably with upper bounds on the numbers of steps required by earlier algorithms.

Edmonds–Karp 1972 (USA)



Dokl. Akad. Nauk SSSR
Tom 194 (1970), No. 4

Soviet Math. Dokl.
Vol. 11 (1970), No. 5

ALGORITHM FOR SOLUTION OF A PROBLEM OF MAXIMUM FLOW IN A NETWORK WITH POWER ESTIMATION

UDC 518.5

E. A. DINIC

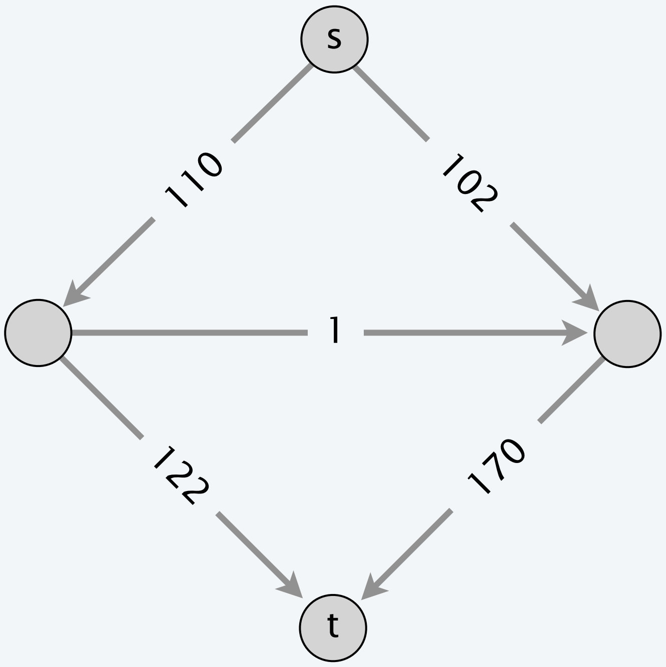
Different variants of the formulation of the problem of maximal stationary flow in a network and its many applications are given in [1]. There also is given an algorithm solving the problem in the case where the initial data are integers (or, what is equivalent, commensurable). In the general case this algorithm requires preliminary rounding off of the initial data, i.e. only an approximate solution of the problem is possible. In this connection the rapidity of convergence of the algorithm is inversely proportional to the relative precision.

Dinic 1970 (Soviet Union)

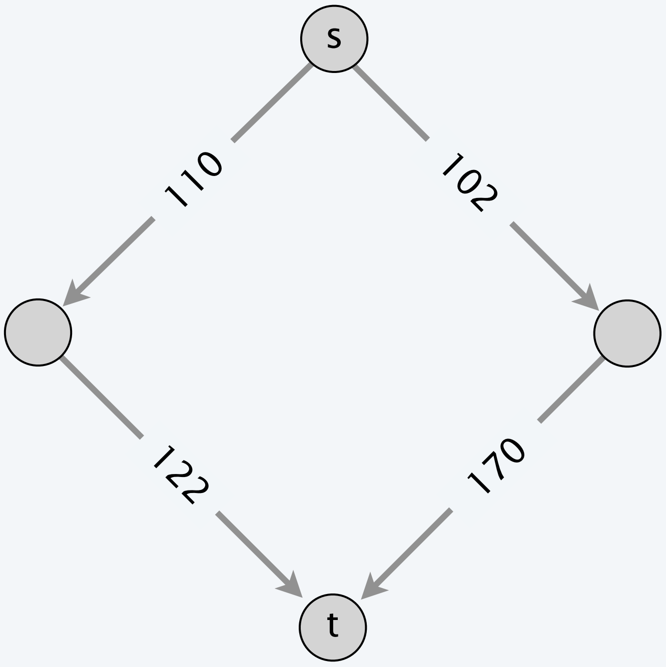
Capacity-scaling algorithm

Intuition. Choose augmenting path with highest bottleneck capacity: it increases flow by max possible amount in given iteration.

- Don't worry about finding exact highest bottleneck path.
- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting only of arcs with capacity ≥ Δ.



G_f



$G_f(\Delta)$, $\Delta = 100$

Capacity-scaling algorithm

```
CAPACITY-SCALING($G$, $s$, $t$, $c$)
FOREACH edge $e \in E$ : $f(e) \leftarrow 0$.
$\Delta \leftarrow$ largest power of $2 \leq C$.

WHILE ($\Delta \geq 1$)
    $G_f(\Delta) \leftarrow \Delta$-residual graph.
    WHILE (there exists an augmenting path $P$ in $G_f(\Delta)$)
        $f \leftarrow$ AUGMENT ($f$, $c$, $P$).
        Update $G_f(\Delta)$.
    $\Delta \leftarrow \Delta / 2$.

RETURN $f$.
```

Capacity-scaling algorithm: proof of correctness

Assumption. All edge capacities are integers between 1 and C.

Integrality invariant. All flow and residual capacity values are integral.

Theorem. If capacity-scaling algorithm terminates, then f is a max-flow.

Pf.

- By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths. ▪

Capacity-scaling algorithm: analysis of running time

Lemma 1. The outer while loop repeats $1 + \lceil \log_2 C \rceil$ times.

Pf. Initially $C/2 < \Delta \le C$; Δ decreases by a factor of 2 in each iteration. ▪

Lemma 2. Let f be the flow at the end of a Δ-scaling phase. Then, the value of the max-flow $\le$ $val(f) + m\,\Delta$. ⟵ proof on next slide

Lemma 3. There are at most $2m$ augmentations per scaling phase.

Pf.

- Let f be the flow at the end of the previous scaling phase.
- LEMMA 2 $\Rightarrow$ $val(f^*) \le val(f) + 2\,m\,\Delta$.
- Each augmentation in a Δ-phase increases $val(f)$ by at least Δ. ▪

Theorem. The scaling max-flow algorithm finds a max flow in $O(m \log C)$ augmentations. It can be implemented to run in $O(m^2 \log C)$ time.

Pf. Follows from LEMMA 1 and LEMMA 3. ▪

Capacity-scaling algorithm: analysis of running time

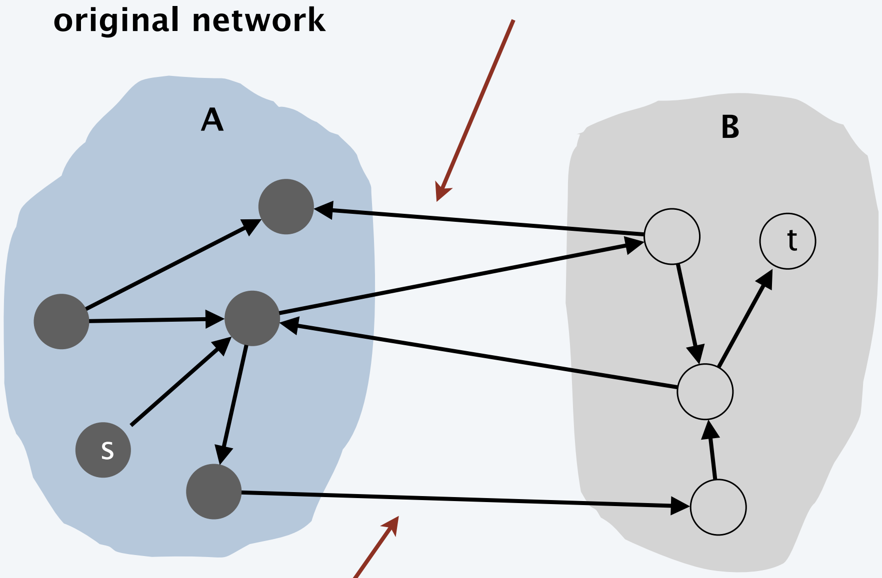
Lemma 2. Let f be the flow at the end of a Δ-scaling phase. Then, the value of the max-flow ≤ $val(f) + m\,\Delta$.

Pf.

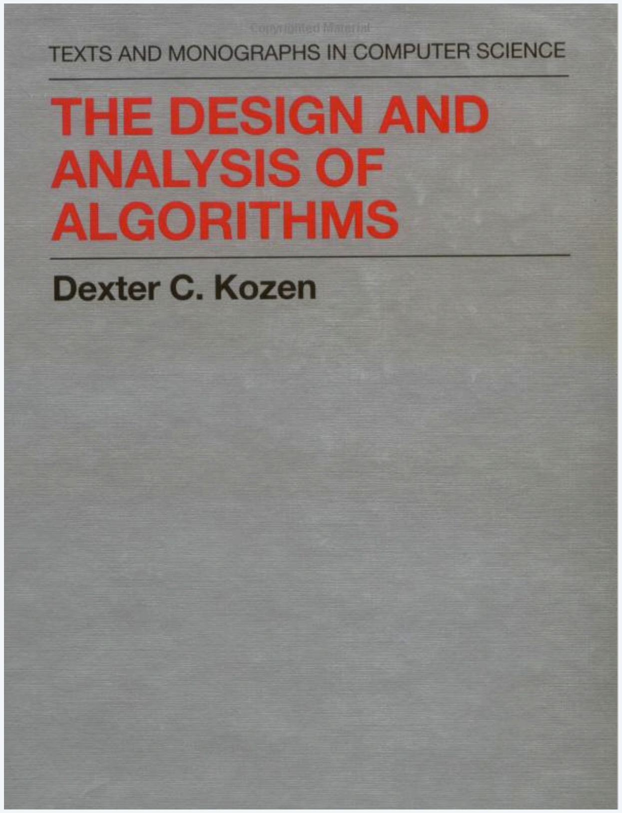
- We show there exists a cut (A, B) such that $cap(A, B) \leq val(f) + m\,\Delta$.
- Choose A to be the set of nodes reachable from s in $G_f(\Delta)$.
- By definition of cut A, $s \in A$.
- By definition of flow f, $t \notin A$.

$$\begin{aligned} val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to A}} f(e) \\ &\geq \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ in to } A} \Delta \\ &= \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\ &\geq cap(A, B) \ - \ m\Delta \ \blacksquare \end{aligned}$$

edge e = (v, w) with v ∈ B, w ∈ A
must have f(e) ≤ Δ



edge e = (v, w) with v ∈ A, w ∈ B
must have f(e) ≥ c(e) – Δ


TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

THE DESIGN AND ANALYSIS OF ALGORITHMS

Dexter C. Kozen

SECTION 17.2

7. NETWORK FLOW I

- max-flow and min-cut problems
- Ford-Fulkerson algorithm
- max-flow min-cut theorem
- capacity-scaling algorithm
- shortest augmenting paths
- blocking-flow algorithm
- unit-capacity simple networks

Shortest augmenting path

Q. Which augmenting path?

A. The one with the fewest number of edges.

can find via BFS

```
SHORTEST-AUGMENTING-PATH($G, s, t, c$)
FOREACH $e \in E : f(e) \leftarrow 0$.
    $G_f \leftarrow$ residual graph.
    WHILE (there exists an augmenting path in $G_f$)
        $P \leftarrow$ BREADTH-FIRST-SEARCH ($G_f, s, t$).
        $f \leftarrow$ AUGMENT ($f, c, P$).
        Update $G_f$.
RETURN $f$.
```

Shortest augmenting path: overview of analysis

L1. Throughout the algorithm, length of the shortest path never decreases.

L2. After at most m shortest path augmentations, the length of the shortest augmenting path strictly increases.

Theorem. The shortest augmenting path algorithm runs in $O(m^2 n)$ time.

Pf.

- $O(m + n)$ time to find shortest augmenting path via BFS.
- $O(m)$ augmentations for paths of length k.
- If there is an augmenting path, there is a simple one.

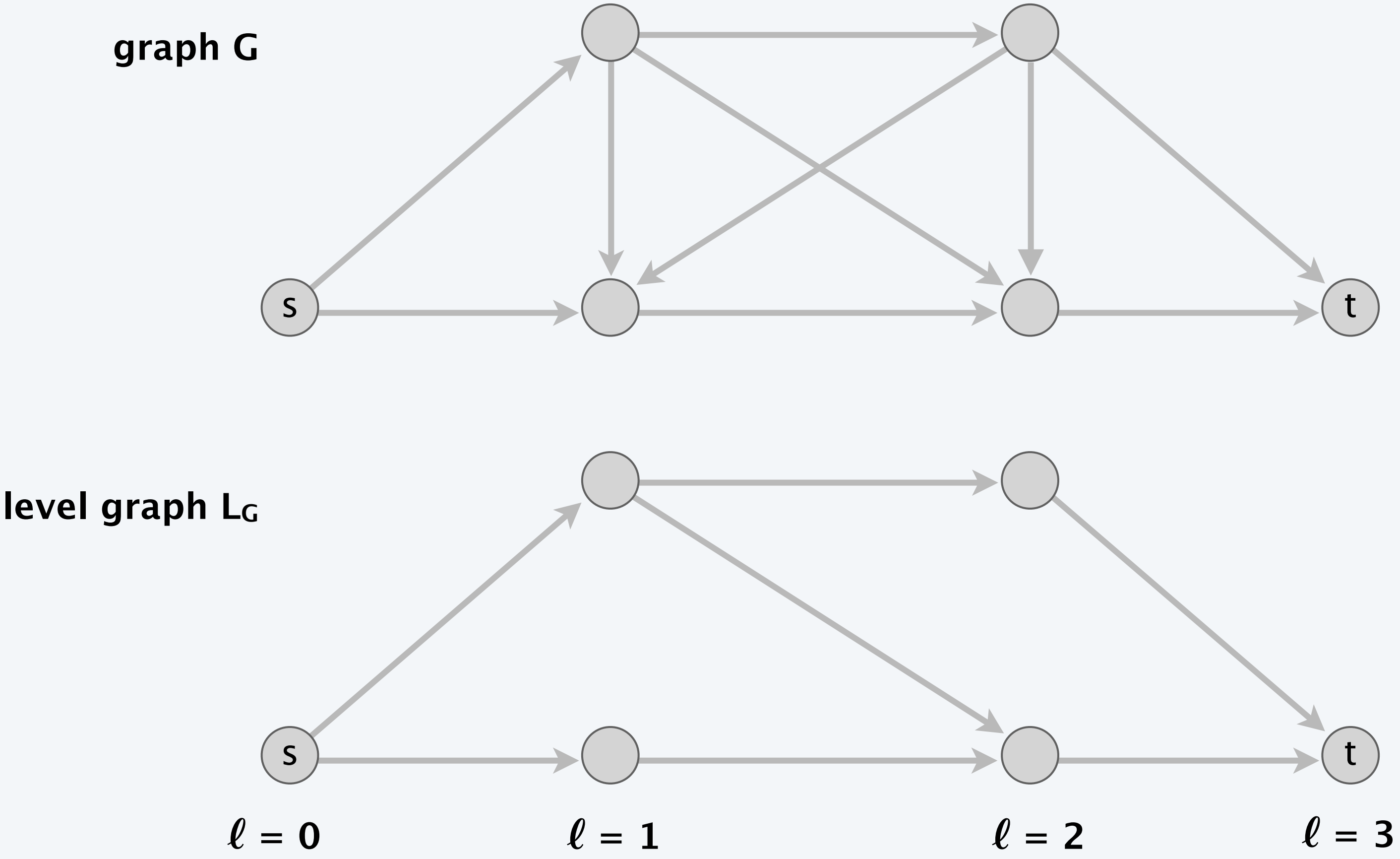
  $\Rightarrow \; 1 \le k < n$

  $\Rightarrow \; O(m\,n)$ augmentations. ▪

Shortest augmenting path: analysis

Def. Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell(v) =$ number of edges in shortest path from s to v.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v,w) \in E$ with $\ell(w) = \ell(v) + 1$.



Shortest augmenting path: analysis

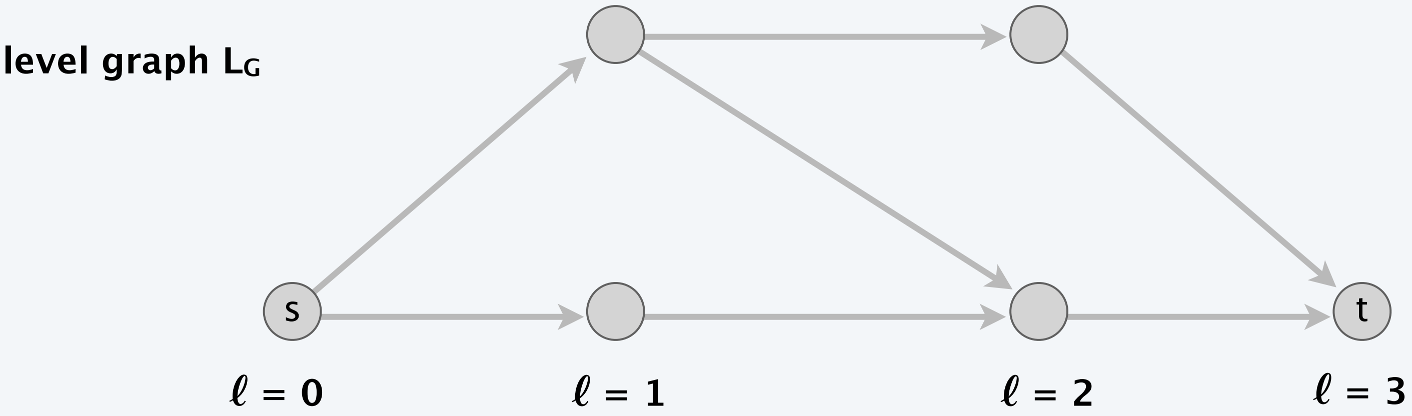
Def. Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell\,(v) =$ number of edges in shortest path from s to v.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v,w) \in E$ with $\ell\,(w) = \ell\,(v) + 1$.

Property. Can compute level graph in $O(m + n)$ time.

Pf. Run BFS; delete back and side edges.

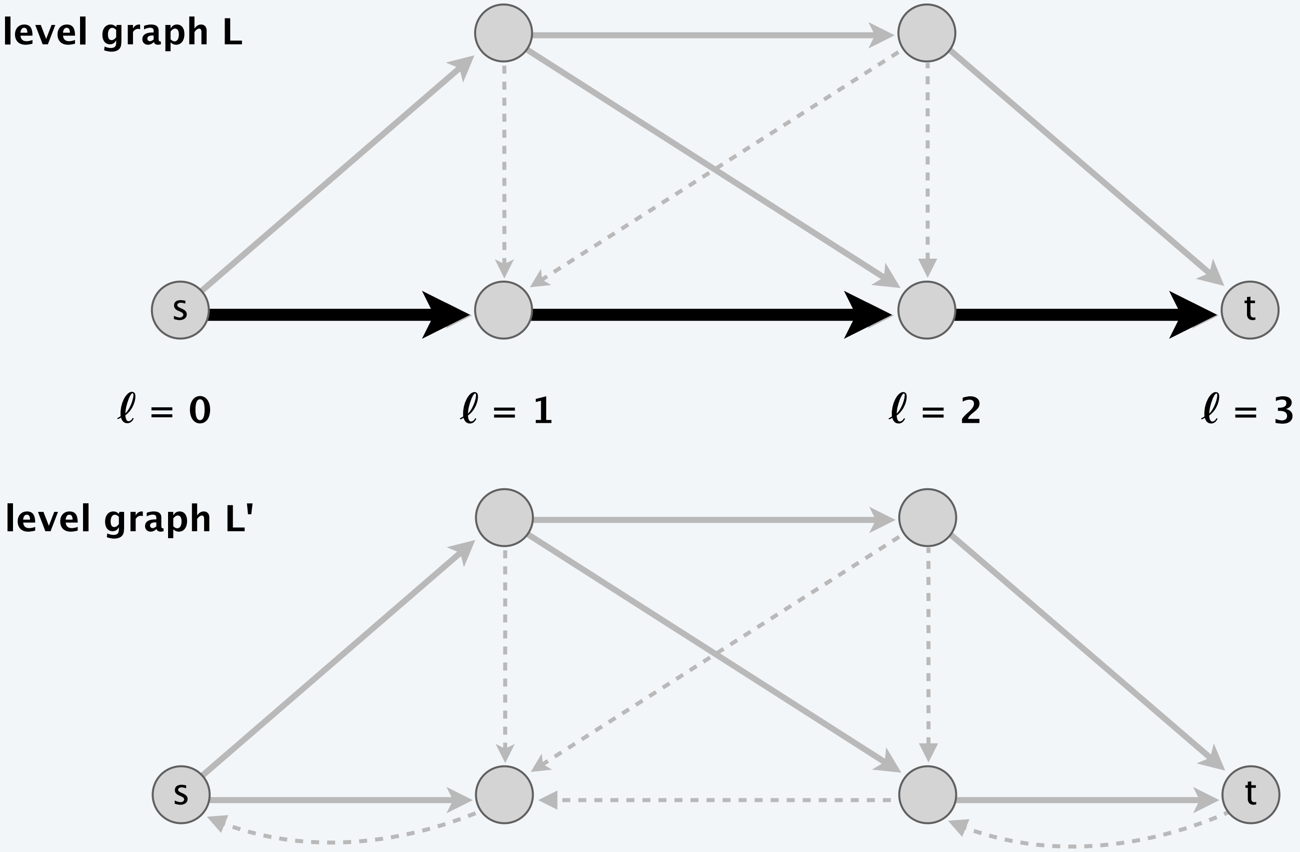
Key property. P is a shortest $s \rightsquigarrow v$ path in G iff P is an $s \rightsquigarrow v$ path L_G.



Shortest augmenting path: analysis

L1. Throughout the algorithm, length of the shortest path never decreases.

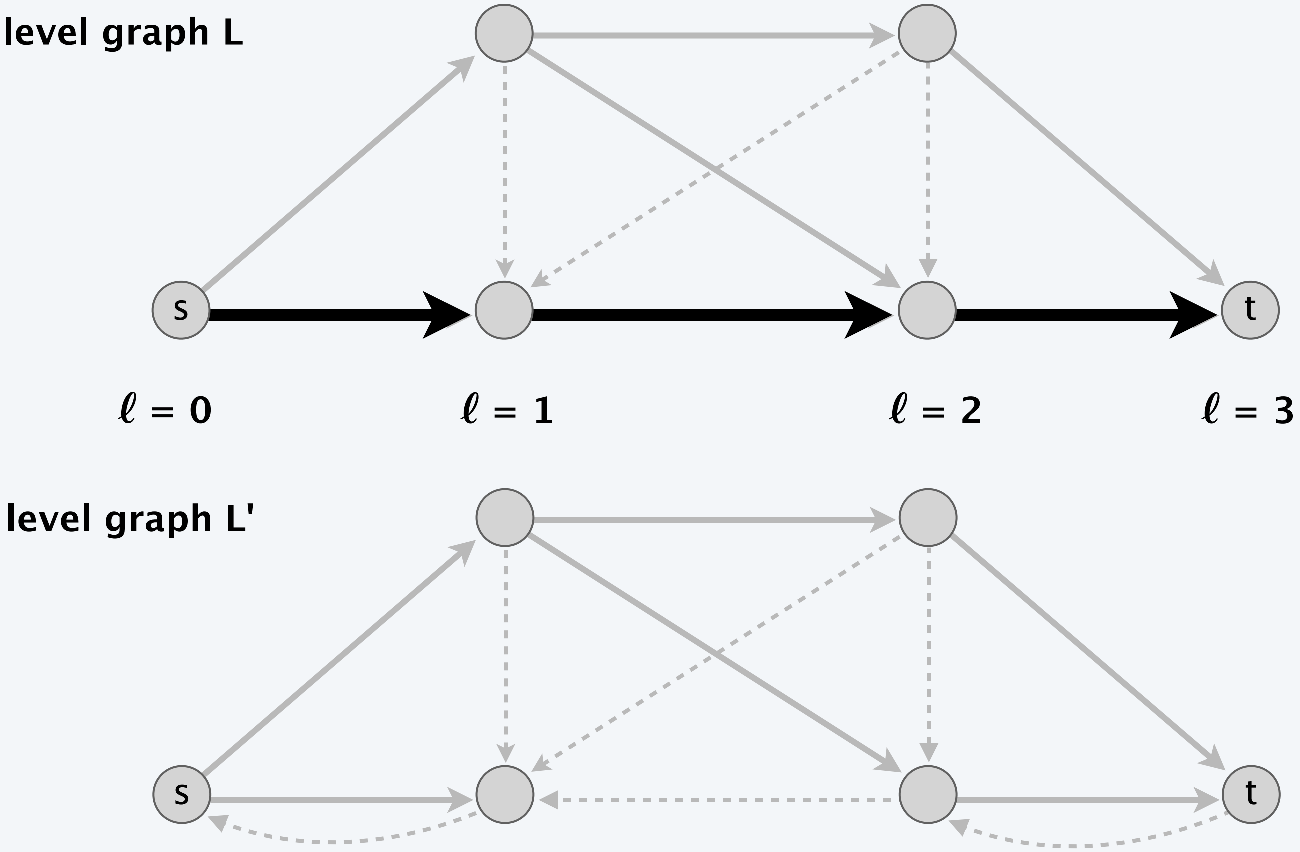
- Let f and f' be flow before and after a shortest path augmentation.
- Let L and L' be level graphs of G_f and $G_{f'}$.
- Only back edges added to $G_{f'}$
  (any path with a back edge is longer than previous length) ▪



Shortest augmenting path: analysis

L2. After at most m shortest path augmentations, the length of the shortest augmenting path strictly increases.

- The bottleneck edge(s) is deleted from L after each augmentation.
- No new edge added to L until length of shortest path strictly increases. ▪



Shortest augmenting path: review of analysis

L1. Throughout the algorithm, length of the shortest path never decreases.

L2. After at most m shortest path augmentations, the length of the shortest augmenting path strictly increases.

Theorem. The shortest augmenting path algorithm runs in $O(m^2 n)$ time.

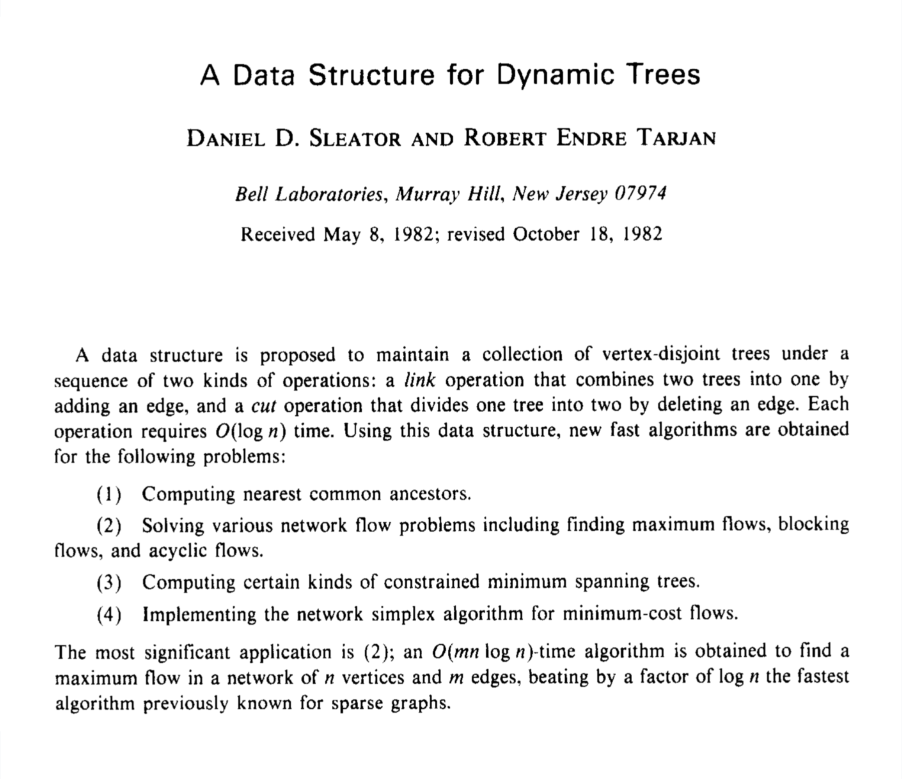
Pf.

- $O(m + n)$ time to find shortest augmenting path via BFS.
- $O(m)$ augmentations for paths of exactly k edges.
- $O(m\,n)$ augmentations. ▪

Shortest augmenting path: improving the running time

Note. $\Theta(m\,n)$ augmentations necessary on some networks.

- Try to decrease time per augmentation instead.
- Simple idea ⇒ $O(m\,n^2)$ [Dinic 1970]
- Dynamic trees ⇒ $O(m\,n \log n)$ [Sleator-Tarjan 1983]



A Data Structure for Dynamic Trees

DANIEL D. SLEATOR AND ROBERT ENDRE TARJAN

Bell Laboratories, Murray Hill, New Jersey 07974

Received May 8, 1982; revised October 18, 1982

A data structure is proposed to maintain a collection of vertex-disjoint trees under a sequence of two kinds of operations: a *link* operation that combines two trees into one by adding an edge, and a *cut* operation that divides one tree into two by deleting an edge. Each operation requires $O(\log n)$ time. Using this data structure, new fast algorithms are obtained for the following problems:

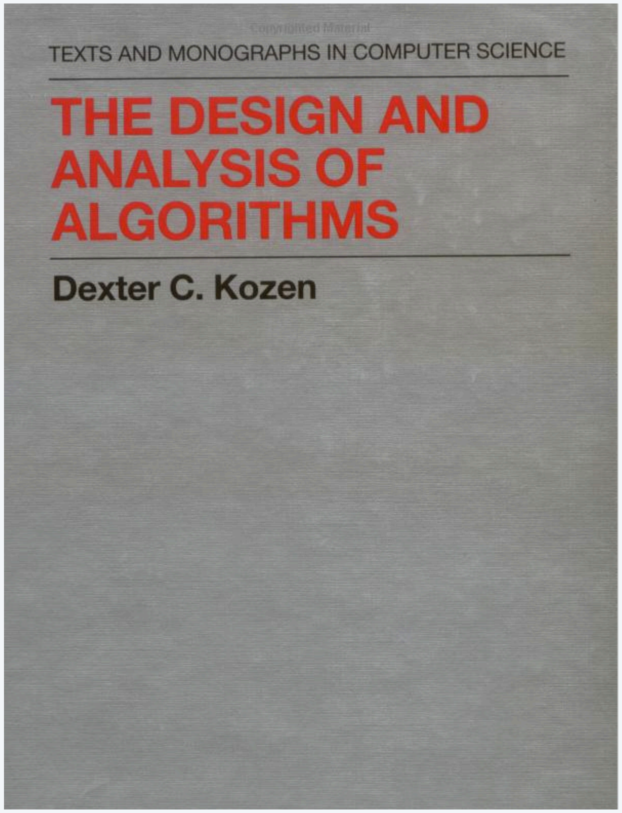
(1) Computing nearest common ancestors.

(2) Solving various network flow problems including finding maximum flows, blocking flows, and acyclic flows.

(3) Computing certain kinds of constrained minimum spanning trees.

(4) Implementing the network simplex algorithm for minimum-cost flows.

The most significant application is (2); an $O(mn \log n)$-time algorithm is obtained to find a maximum flow in a network of n vertices and m edges, beating by a factor of $\log n$ the fastest algorithm previously known for sparse graphs.


TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

THE DESIGN AND ANALYSIS OF ALGORITHMS

Dexter C. Kozen

SECTION 18.1

7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford-Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *blocking-flow algorithm*
- *unit-capacity simple networks*

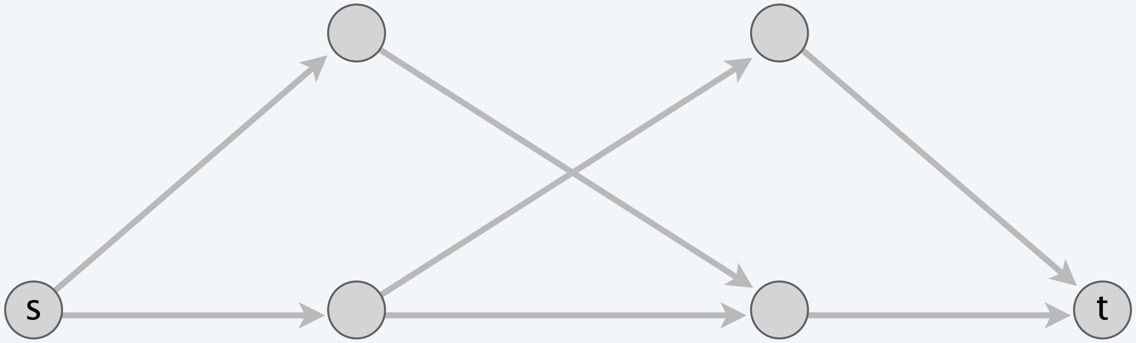
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



level graph L_G

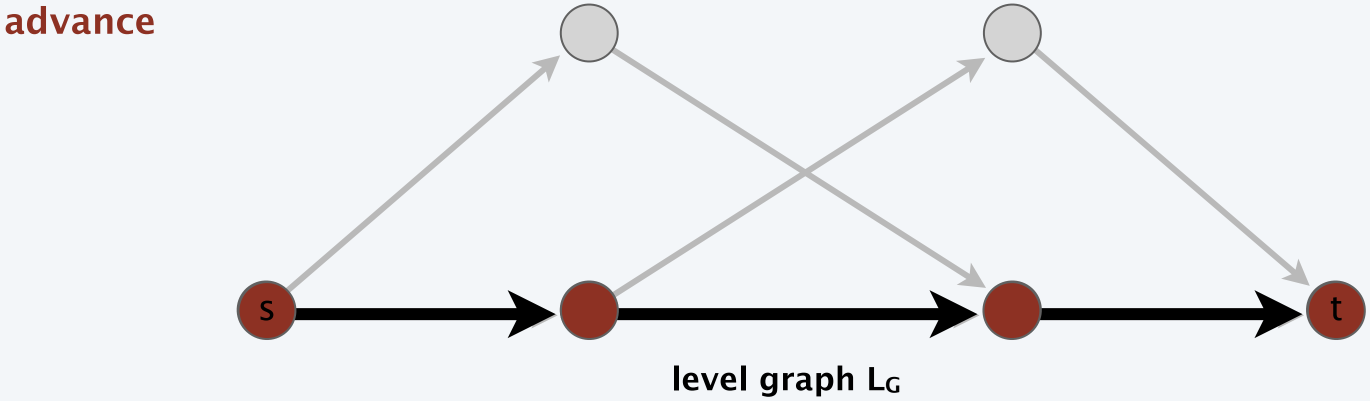
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



Blocking-flow algorithm

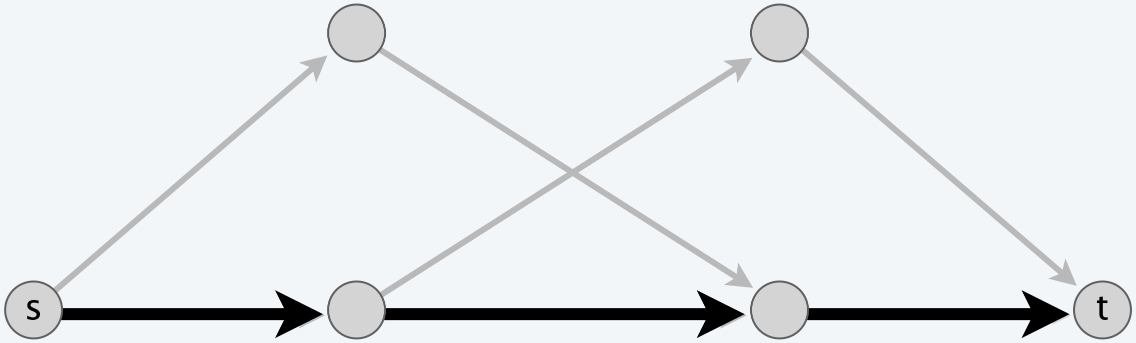
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

augment



level graph L_G

Blocking-flow algorithm

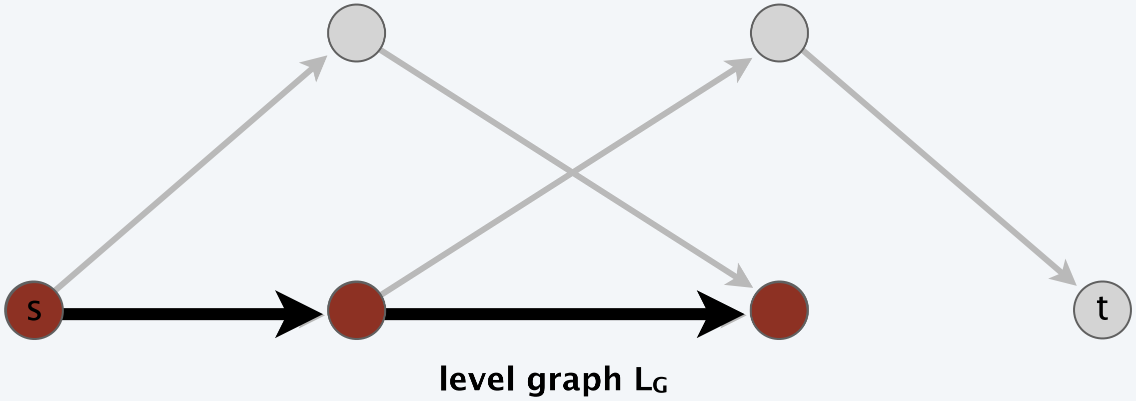
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

advance



Blocking-flow algorithm

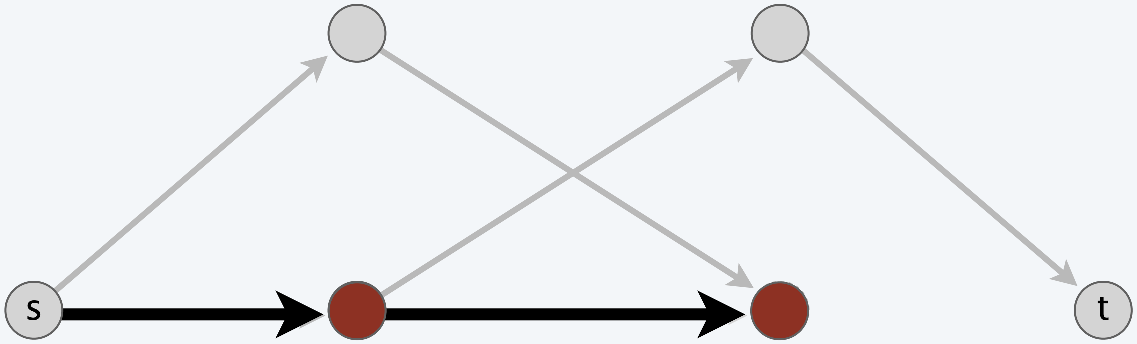
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

retreat



level graph L_G

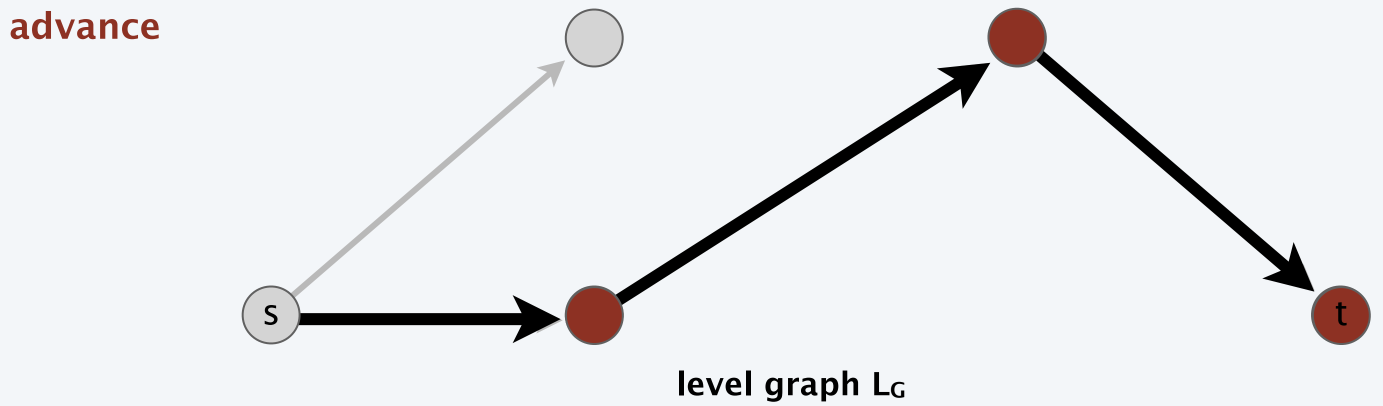
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



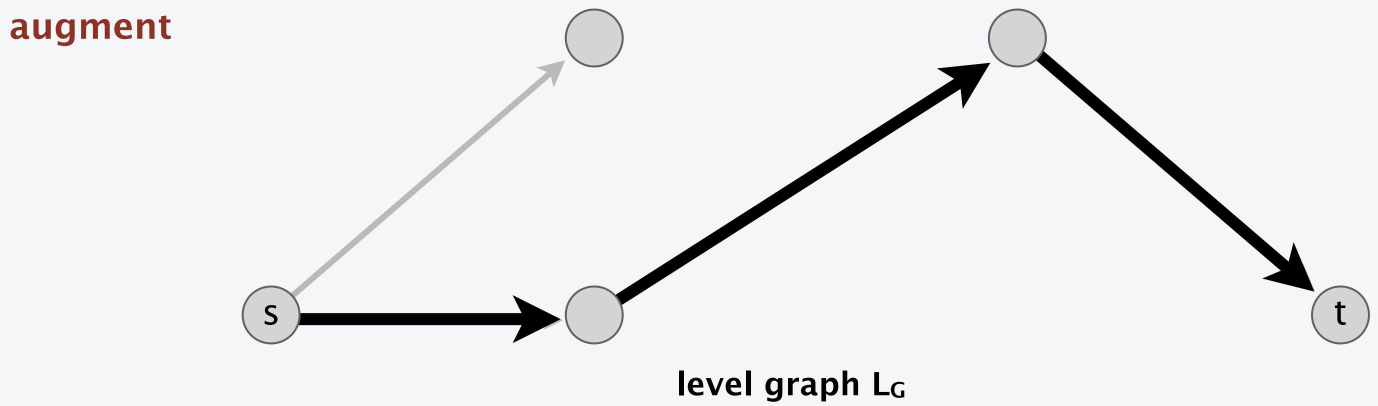
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



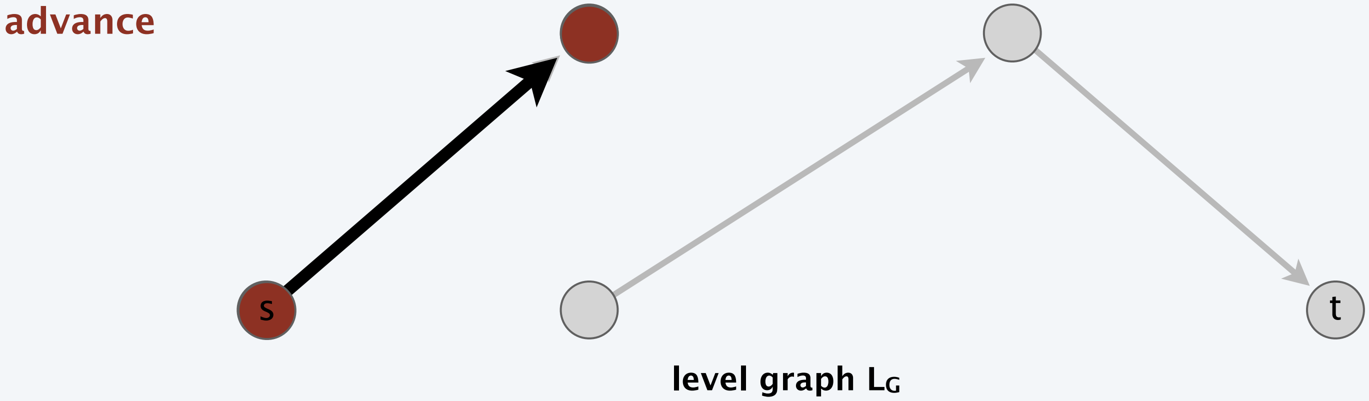
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



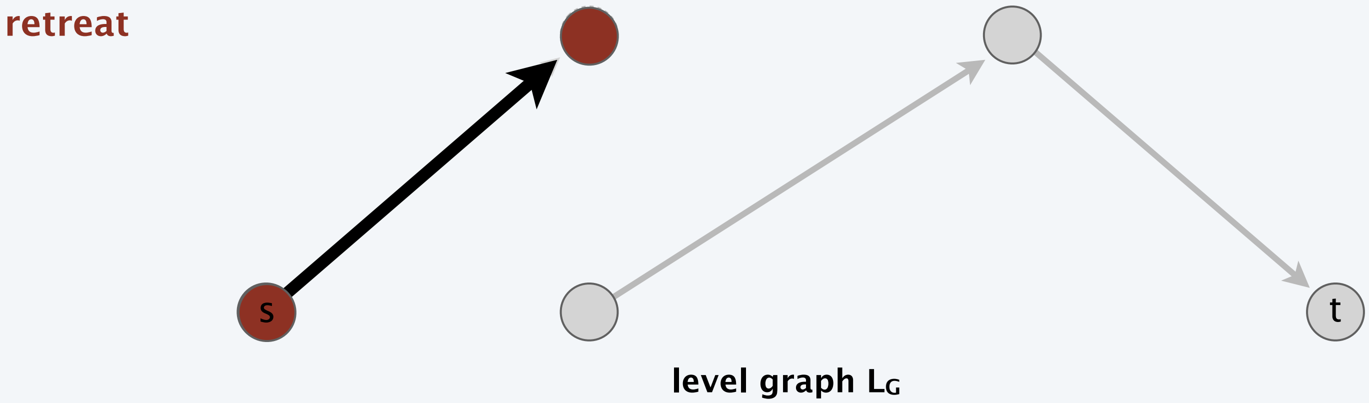
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



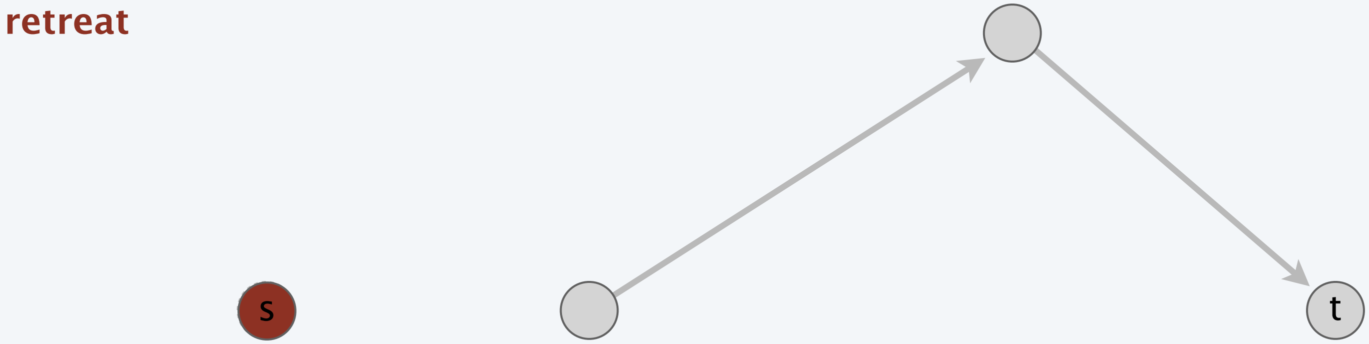
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



level graph L_G

Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

end of phase

level graph L_G

Blocking-flow algorithm

```
INITIALIZE($G, s, t, f, c$)

$L_G \leftarrow$ level-graph of $G_f$.
$P \leftarrow \varnothing$.
GOTO ADVANCE($s$).
```

```
RETREAT($v$)

IF ($v = s$) STOP.
ELSE
    Delete $v$ (and all incident edges) from $L_G$.
    Remove last edge ($u$, $v$) from $P$.
    GOTO ADVANCE($u$).
```

```
ADVANCE($v$)

IF ($v = t$)
    AUGMENT($P$).
    Remove saturated edges from $L_G$.
    $P \leftarrow \varnothing$.
    GOTO ADVANCE($s$).

IF (there exists edge $(v, w) \in L_G$)
    Add edge $(v, w)$ to $P$.
    GOTO ADVANCE($w$).

ELSE GOTO RETREAT($v$).
```

Blocking-flow algorithm: analysis

Lemma. A phase can be implemented in $O(m\,n)$ time.

Pf.

- Initialization happens once per phase. ← O(m) using BFS
- At most m augmentations per phase. ← O(mn) per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase. ← O(m + n) per phase
  (because a retreat deletes one node from L_G)
- At most $m\,n$ advances per phase. ← O(mn) per phase
  (because at most n advances before retreat or augmentation) ▪

Theorem. [Dinic 1970] The blocking-flow algorithm runs in $O(mn^2)$ time.

Pf.

- By lemma, $O(mn)$ time per phase.
- At most n phases (as in shortest augment path analysis). ▪

Choosing good augmenting paths: summary

Assumption. Integer capacities between 1 and C.

method	# augmentations	running time
augmenting path	$n\,C$	$O(m\,n\,C)$
fattest augmenting path	$m \log (mC)$	$O(m^2 \log n \log (mC))$
capacity scaling	$m \log C$	$O(m^2 \log C)$
improved capacity scaling	$m \log C$	$O(m\,n \log C)$
shortest augmenting path	$m\,n$	$O(m^2\,n)$
improved shortest augmenting path	$m\,n$	$O(m\,n^2)$
dynamic trees	$m\,n$	$O(m\,n \log n)$

Maximum flow algorithms: theory

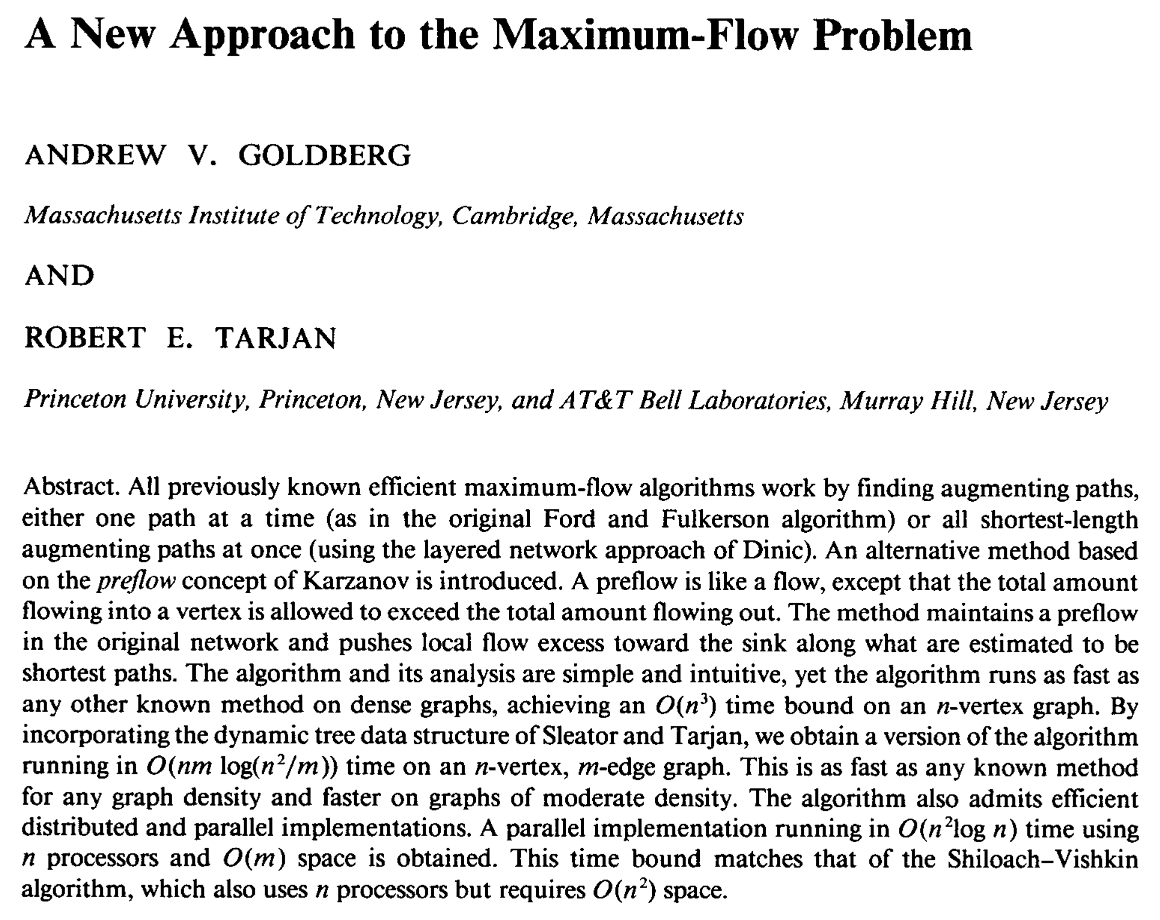
year	method	worst case	discovered by
1951	simplex	$O(m^3\ C)$	Dantzig
1955	augmenting path	$O(m^2\ C)$	Ford-Fulkerson
1970	shortest augmenting path	$O(m^3)$	Dinic, Edmonds-Karp
1970	fattest augmenting path	$O(m^2 \log m \log(m C))$	Dinic, Edmonds-Karp
1977	blocking flow	$O(m^{5/2})$	Cherkasky
1978	blocking flow	$O(m^{7/3})$	Galil
1983	dynamic trees	$O(m^2 \log m)$	Sleator-Tarjan
1985	capacity scaling	$O(m^2 \log C)$	Gabow
1997	length function	$O(m^{3/2} \log m \log C)$	Goldberg-Rao
2012	compact network	$O(m^2 / \log m)$	Orlin
?	?	$O(m)$	?

max-flow algorithms for sparse digraphs with m edges, integer capacities between 1 and C

Maximum flow algorithms: practice

Push-relabel algorithm (SECTION 7.4). [Goldberg-Tarjan 1988]

Increases flow one edge at a time instead of one augmenting path at a time.



A New Approach to the Maximum-Flow Problem

ANDREW V. GOLDBERG

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

ROBERT E. TARJAN

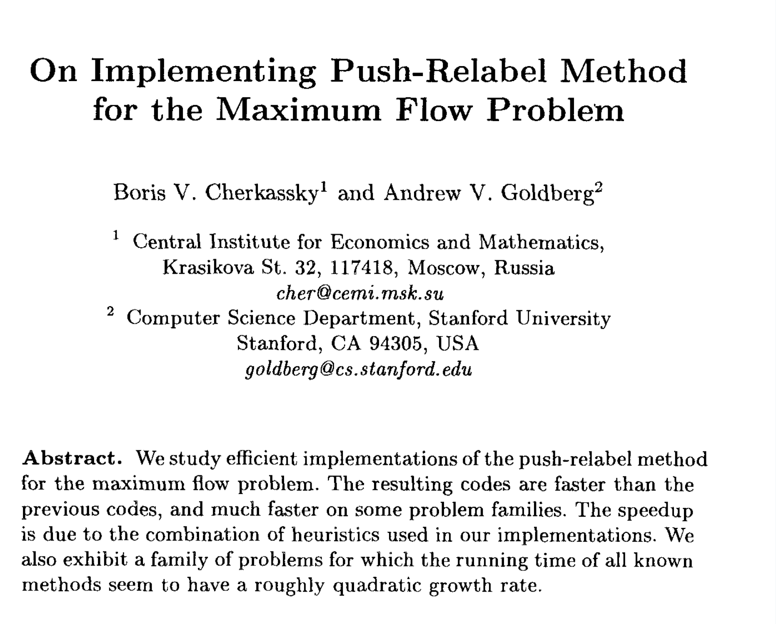
Princeton University, Princeton, New Jersey, and AT&T Bell Laboratories, Murray Hill, New Jersey

Abstract. All previously known efficient maximum-flow algorithms work by finding augmenting paths, either one path at a time (as in the original Ford and Fulkerson algorithm) or all shortest-length augmenting paths at once (using the layered network approach of Dinic). An alternative method based on the *preflow* concept of Karzanov is introduced. A preflow is like a flow, except that the total amount flowing into a vertex is allowed to exceed the total amount flowing out. The method maintains a preflow in the original network and pushes local flow excess toward the sink along what are estimated to be shortest paths. The algorithm and its analysis are simple and intuitive, yet the algorithm runs as fast as any other known method on dense graphs, achieving an $O(n^3)$ time bound on an n-vertex graph. By incorporating the dynamic tree data structure of Sleator and Tarjan, we obtain a version of the algorithm running in $O(nm \log(n^2/m))$ time on an n-vertex, m-edge graph. This is as fast as any known method for any graph density and faster on graphs of moderate density. The algorithm also admits efficient distributed and parallel implementations. A parallel implementation running in $O(n^2 \log n)$ time using n processors and $O(m)$ space is obtained. This time bound matches that of the Shiloach–Vishkin algorithm, which also uses n processors but requires $O(n^2)$ space.

Maximum flow algorithms: practice

Warning. Worst-case running time is generally not useful for predicting or comparing max-flow algorithm performance in practice.

Best in practice. Push-relabel method with gap relabeling: $O(m^{3/2})$.

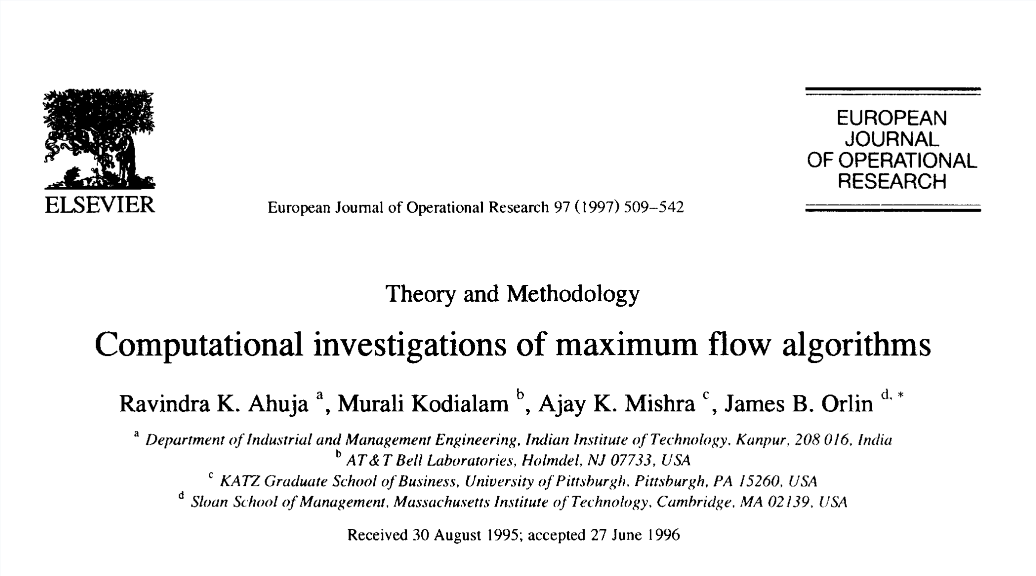


On Implementing Push-Relabel Method for the Maximum Flow Problem

Boris V. Cherkassky[1] and Andrew V. Goldberg[2]

[1] Central Institute for Economics and Mathematics,
Krasikova St. 32, 117418, Moscow, Russia
cher@cemi.msk.su
[2] Computer Science Department, Stanford University
Stanford, CA 94305, USA
goldberg@cs.stanford.edu

Abstract. We study efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous codes, and much faster on some problem families. The speedup is due to the combination of heuristics used in our implementations. We also exhibit a family of problems for which the running time of all known methods seem to have a roughly quadratic growth rate.



ELSEVIER

European Journal of Operational Research 97 (1997) 509–542

EUROPEAN
JOURNAL
OF OPERATIONAL
RESEARCH

Theory and Methodology

Computational investigations of maximum flow algorithms

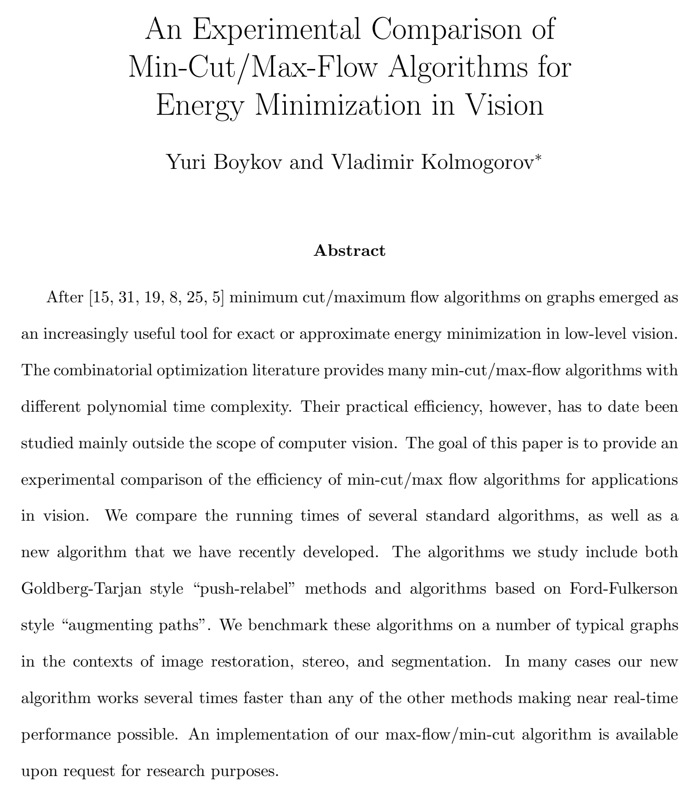
Ravindra K. Ahuja [a], Murali Kodialam [b], Ajay K. Mishra [c], James B. Orlin [d, *]

[a] *Department of Industrial and Management Engineering, Indian Institute of Technology, Kanpur, 208 016, India*
[b] *AT&T Bell Laboratories, Holmdel, NJ 07733, USA*
[c] *KATZ Graduate School of Business, University of Pittsburgh, Pittsburgh, PA 15260, USA*
[d] *Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

Received 30 August 1995; accepted 27 June 1996

Maximum flow algorithms: practice

Computer vision. Different algorithms work better for some dense problems that arise in applications to computer vision.

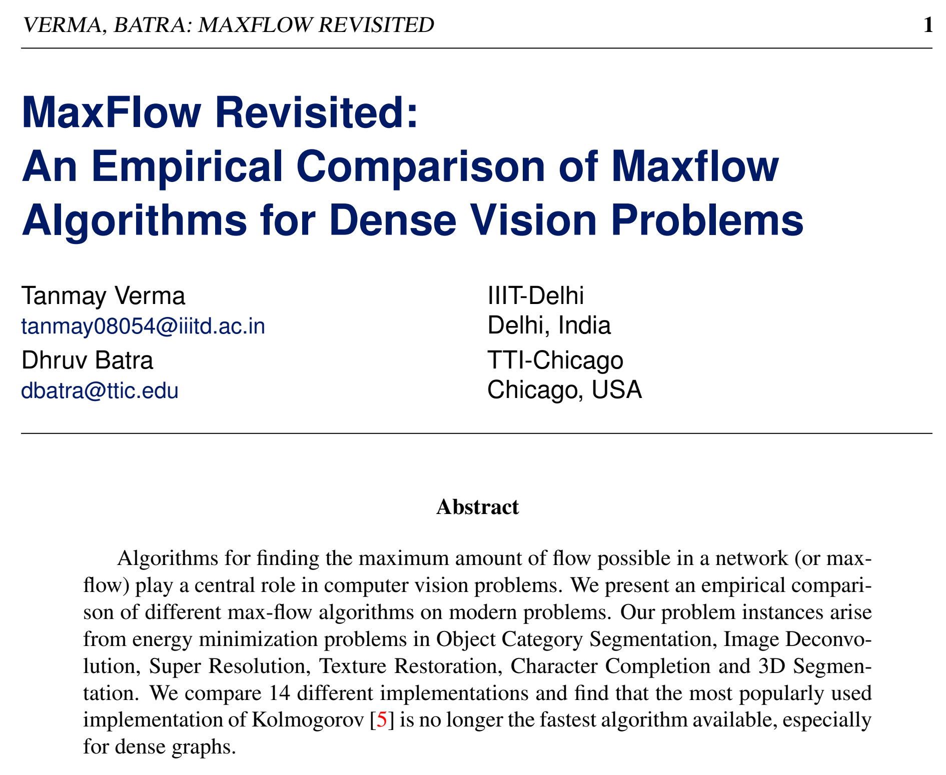


An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision

Yuri Boykov and Vladimir Kolmogorov*

Abstract

After [15, 31, 19, 8, 25, 5] minimum cut/maximum flow algorithms on graphs emerged as an increasingly useful tool for exact or approximate energy minimization in low-level vision. The combinatorial optimization literature provides many min-cut/max-flow algorithms with different polynomial time complexity. Their practical efficiency, however, has to date been studied mainly outside the scope of computer vision. The goal of this paper is to provide an experimental comparison of the efficiency of min-cut/max flow algorithms for applications in vision. We compare the running times of several standard algorithms, as well as a new algorithm that we have recently developed. The algorithms we study include both Goldberg-Tarjan style "push-relabel" methods and algorithms based on Ford-Fulkerson style "augmenting paths". We benchmark these algorithms on a number of typical graphs in the contexts of image restoration, stereo, and segmentation. In many cases our new algorithm works several times faster than any of the other methods making near real-time performance possible. An implementation of our max-flow/min-cut algorithm is available upon request for research purposes.



VERMA, BATRA: MAXFLOW REVISITED **1**

MaxFlow Revisited: An Empirical Comparison of Maxflow Algorithms for Dense Vision Problems

Tanmay Verma
tanmay08054@iiitd.ac.in
IIIT-Delhi
Delhi, India
Dhruv Batra
dbatra@ttic.edu
TTI-Chicago
Chicago, USA

Abstract

Algorithms for finding the maximum amount of flow possible in a network (or max-flow) play a central role in computer vision problems. We present an empirical comparison of different max-flow algorithms on modern problems. Our problem instances arise from energy minimization problems in Object Category Segmentation, Image Deconvolution, Super Resolution, Texture Restoration, Character Completion and 3D Segmentation. We compare 14 different implementations and find that the most popularly used implementation of Kolmogorov [5] is no longer the fastest algorithm available, especially for dense graphs.

7. NETWORK FLOW I

- *max-flow and min-cut problems*
- *Ford-Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *blocking-flow algorithm*
- *unit-capacity simple networks*

Bipartite matching

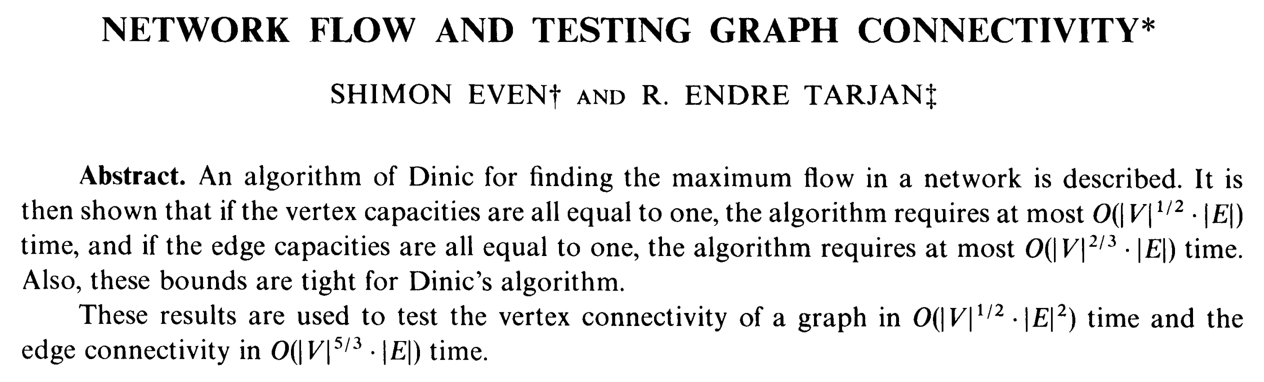
Q. Which max-flow algorithm to use for bipartite matching?

- Generic augmenting path: $O(m \mid f * \mid) = O(m n)$.
- Capacity scaling: $O(m^2 \log U) = \text{O}(m^2)$.
- Shortest augmenting path: $O(m n^2)$.

Q. Suggests "more clever" algorithms are not as good as we first thought?

A. No, just need more clever analysis!

Next. We prove that shortest augmenting path algorithm can be implemented in $O(m n^{1/2})$ time.



NETWORK FLOW AND TESTING GRAPH CONNECTIVITY*

SHIMON EVEN† AND R. ENDRE TARJAN‡

Abstract. An algorithm of Dinic for finding the maximum flow in a network is described. It is then shown that if the vertex capacities are all equal to one, the algorithm requires at most $O(|V|^{1/2} \cdot |E|)$ time, and if the edge capacities are all equal to one, the algorithm requires at most $O(|V|^{2/3} \cdot |E|)$ time. Also, these bounds are tight for Dinic's algorithm.

These results are used to test the vertex connectivity of a graph in $O(|V|^{1/2} \cdot |E|^2)$ time and the edge connectivity in $O(|V|^{5/3} \cdot |E|)$ time.

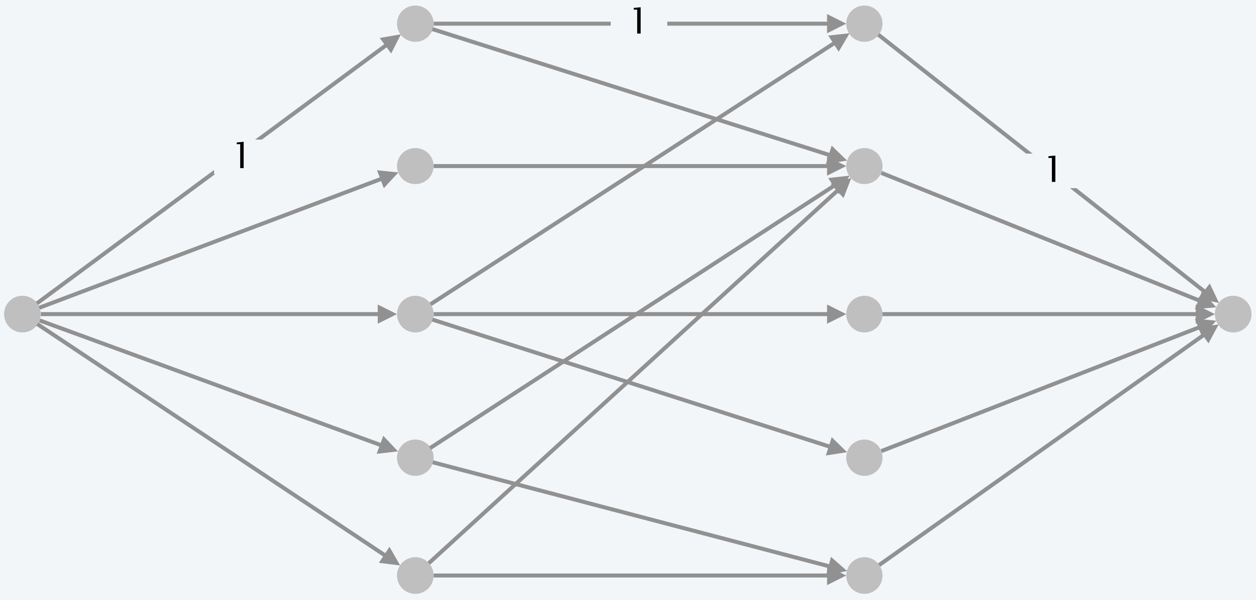
Unit-capacity simple networks

Def. A network is a unit-capacity simple network if:

- Every edge capacity is 1.
- Every node (other than s or t) has either (i) at most one entering edge or (ii) at most one leaving edge.

Property. Let G be a simple unit-capacity network and let f be a 0-1 flow, then G_f is a unit-capacity simple network.

Ex. Bipartite matching.



Unit-capacity simple networks

Shortest augmenting path algorithm.

- Normal augmentation: length of shortest path does not change.
- Special augmentation: length of shortest path strictly increases.

Theorem. [Even-Tarjan 1975] In unit-capacity simple networks, the shortest augmenting path algorithm computes a maximum flow in $O(m\, n^{1/2})$ time.

Pf.

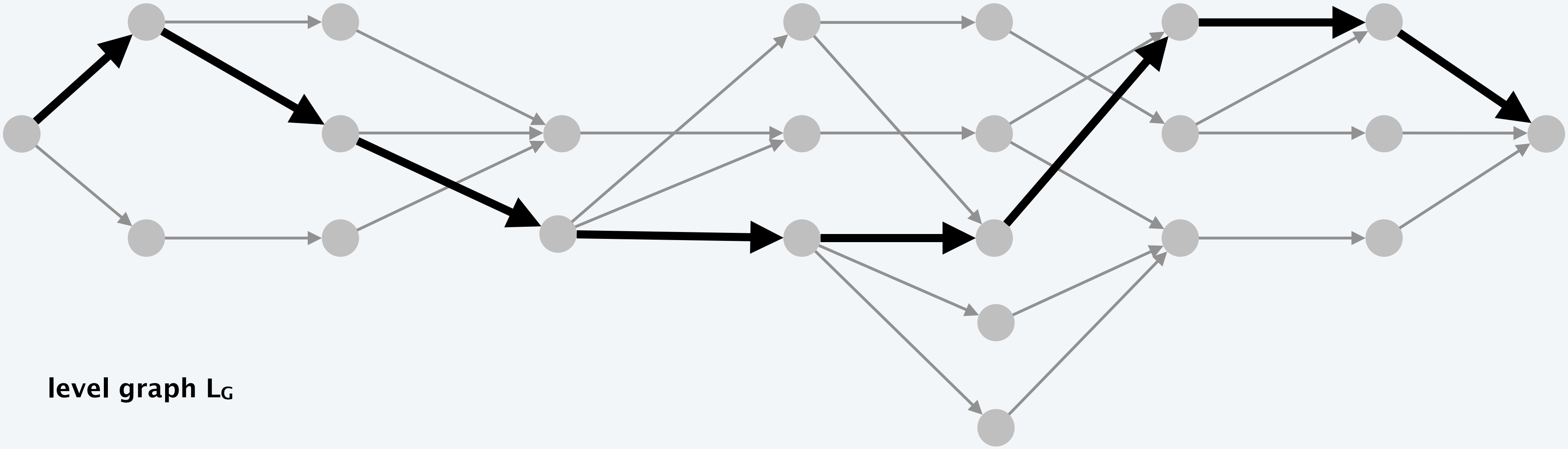
- L1. Each phase of normal augmentations takes $O(m)$ time.
- L2. After at most $n^{1/2}$ phases, $|f| \geq |f^*| - n^{1/2}$.
- L3. After at most $n^{1/2}$ additional augmentations, flow is optimal. ▪

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

advance



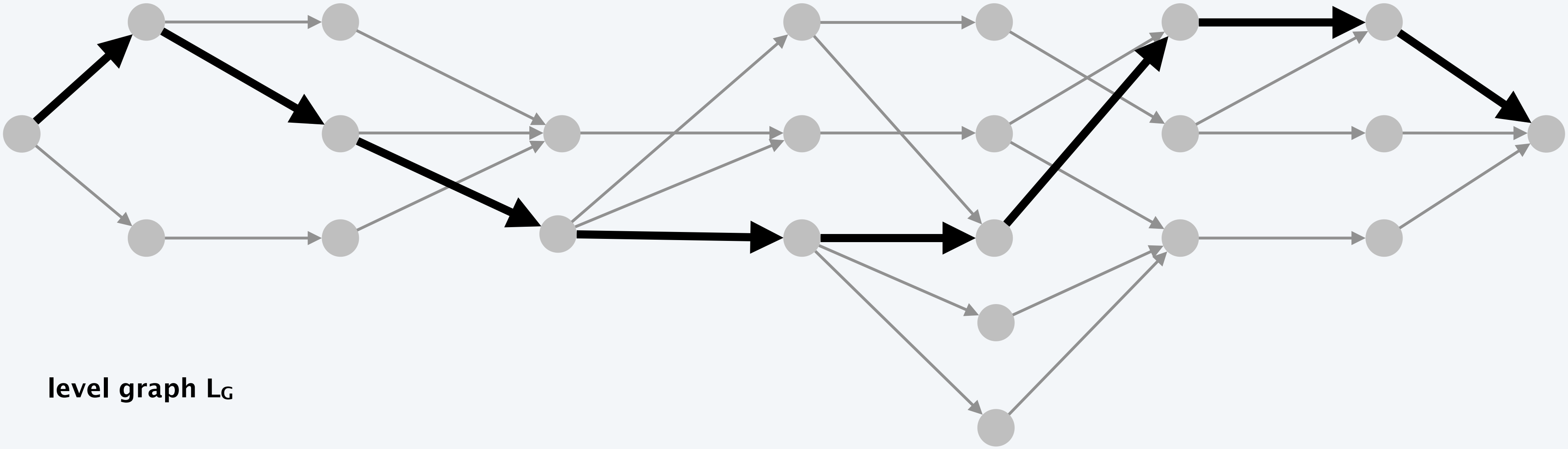
level graph L_G

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

augment



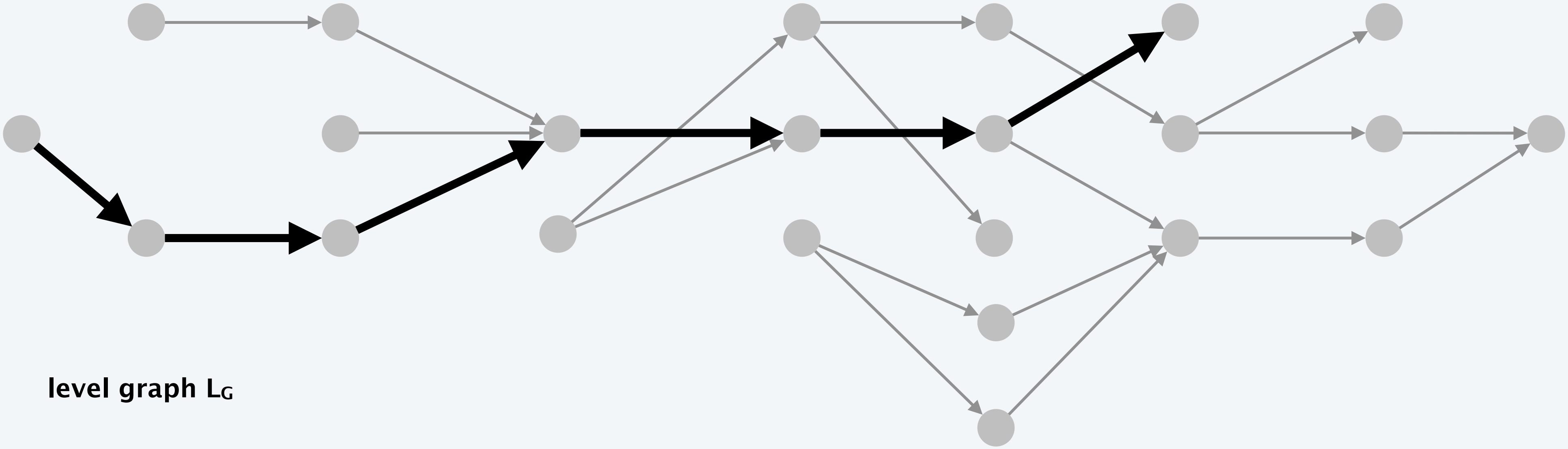
level graph L_G

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

advance



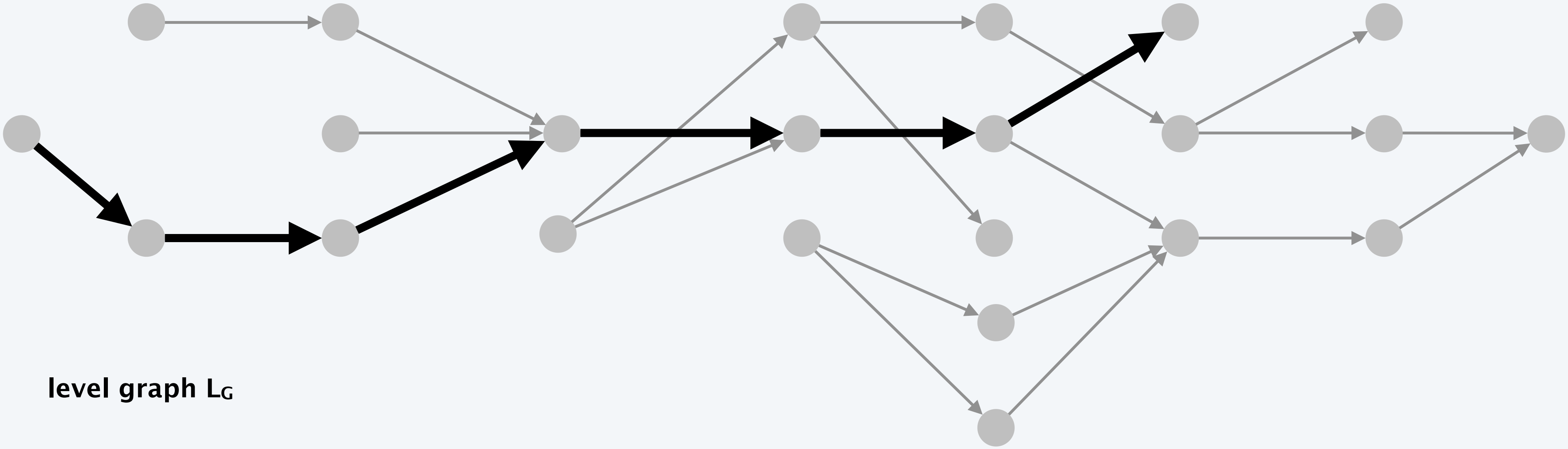
level graph L_G

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

retreat



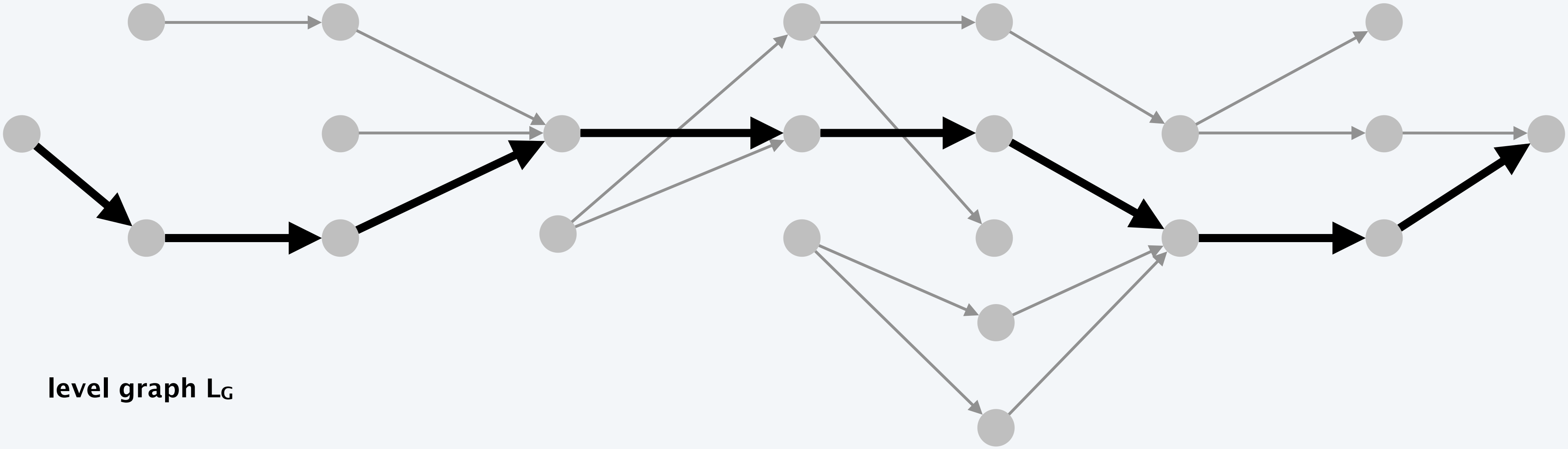
level graph L_G

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

advance



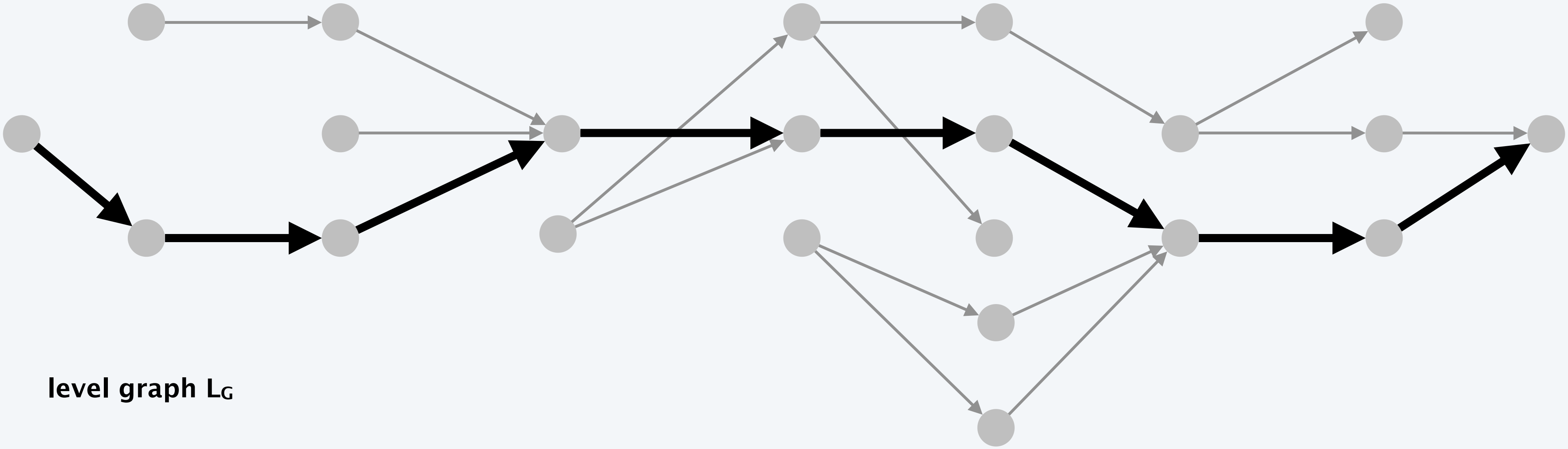
level graph L_G

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

augment



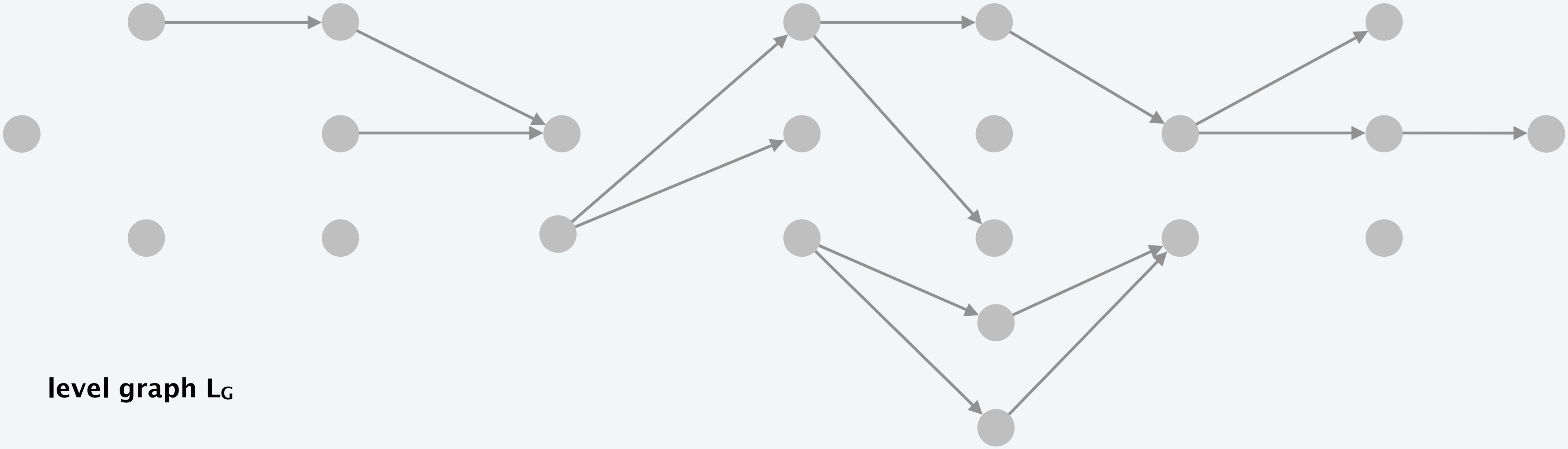
level graph L_G

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

end of phase



level graph L_G

Unit-capacity simple networks: analysis

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

LEMMA 1. A phase of normal augmentations takes $O(m)$ time.

Pf.

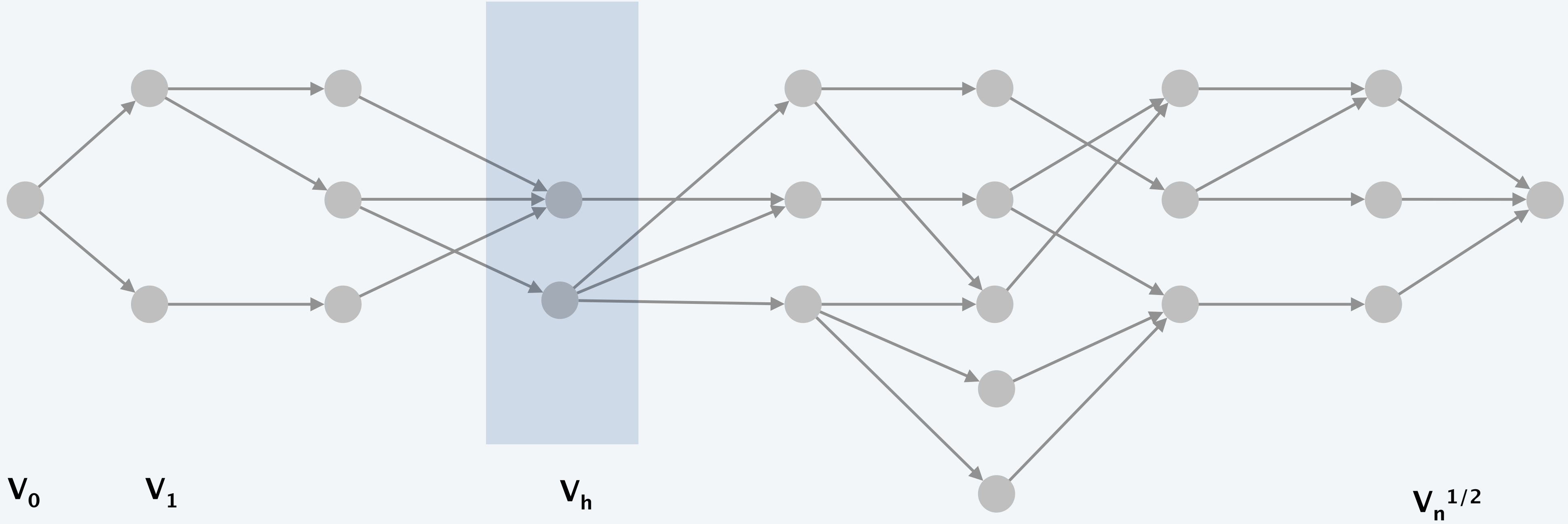
- $O(m)$ to create level graph L_G.
- $O(1)$ per edge since each edge traversed and deleted at most once.
- $O(1)$ per node since each node deleted at most once. ▪

Unit-capacity simple networks: analysis

Lemma 2. After at most $n^{1/2}$ phases, $|f| \geq |f^*| - n^{1/2}$.

- After $n^{1/2}$ phases, length of shortest augmenting path is $> n^{1/2}$.
- Level graph has more than $n^{1/2}$ levels.
- Let $1 \leq h \leq n^{1/2}$ be layer with min number of nodes: $|V_h| \leq n^{1/2}$.

level graph L_G for flow f



Unit-capacity simple networks: analysis

LEMMA 2. After at most $n^{1/2}$ phases, $|f| \geq |f^*| - n^{1/2}$.

- After $n^{1/2}$ phases, length of shortest augmenting path is $> n^{1/2}$.
- Level graph has more than $n^{1/2}$ levels.
- Let $1 \leq h \leq n^{1/2}$ be layer with min number of nodes: $|V_h| \leq n^{1/2}$.
- Let $A = \{v : \ell(v) < h\} \cup \{v : \ell(v) = h$ and v has ≤ 1 outgoing residual edge$\}$.
- $cap_f(A, B) \leq |V_h| \leq n^{1/2} \Rightarrow |f| \geq |f^*| - n^{1/2}$. ▪