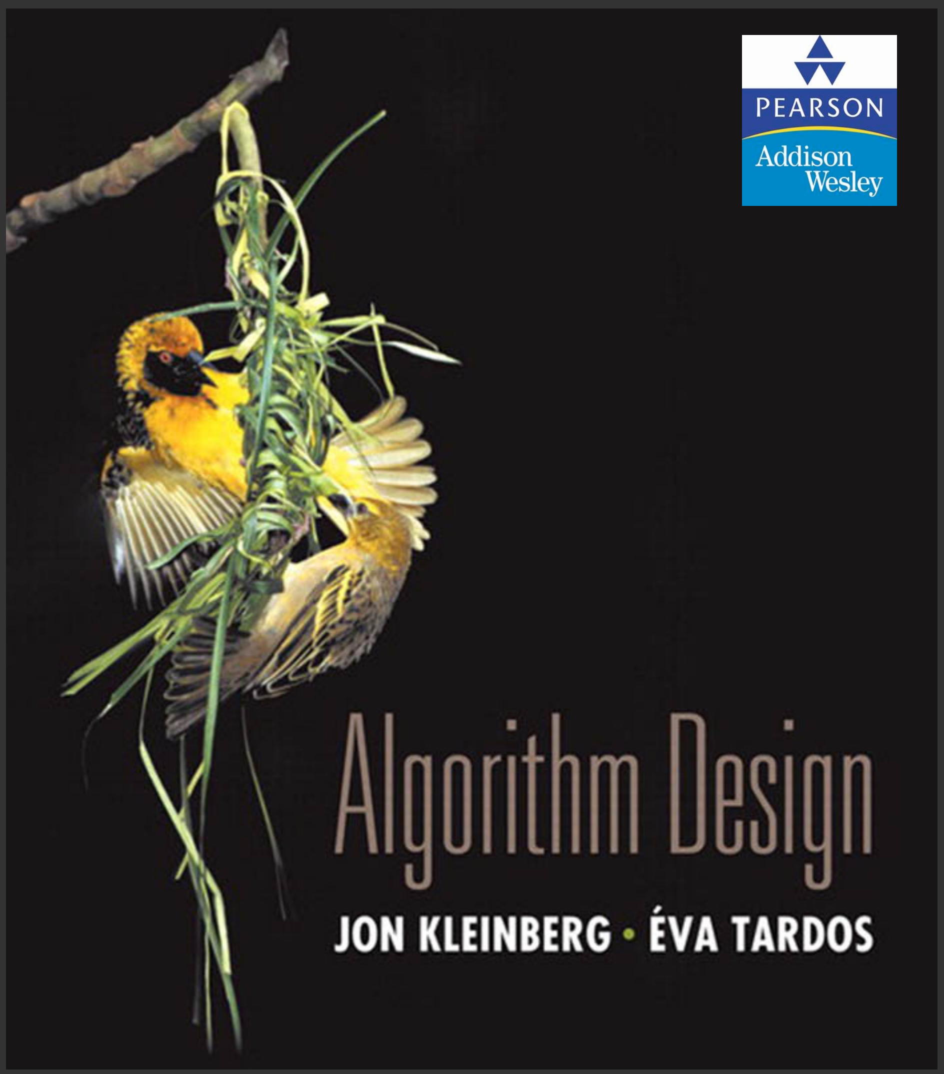


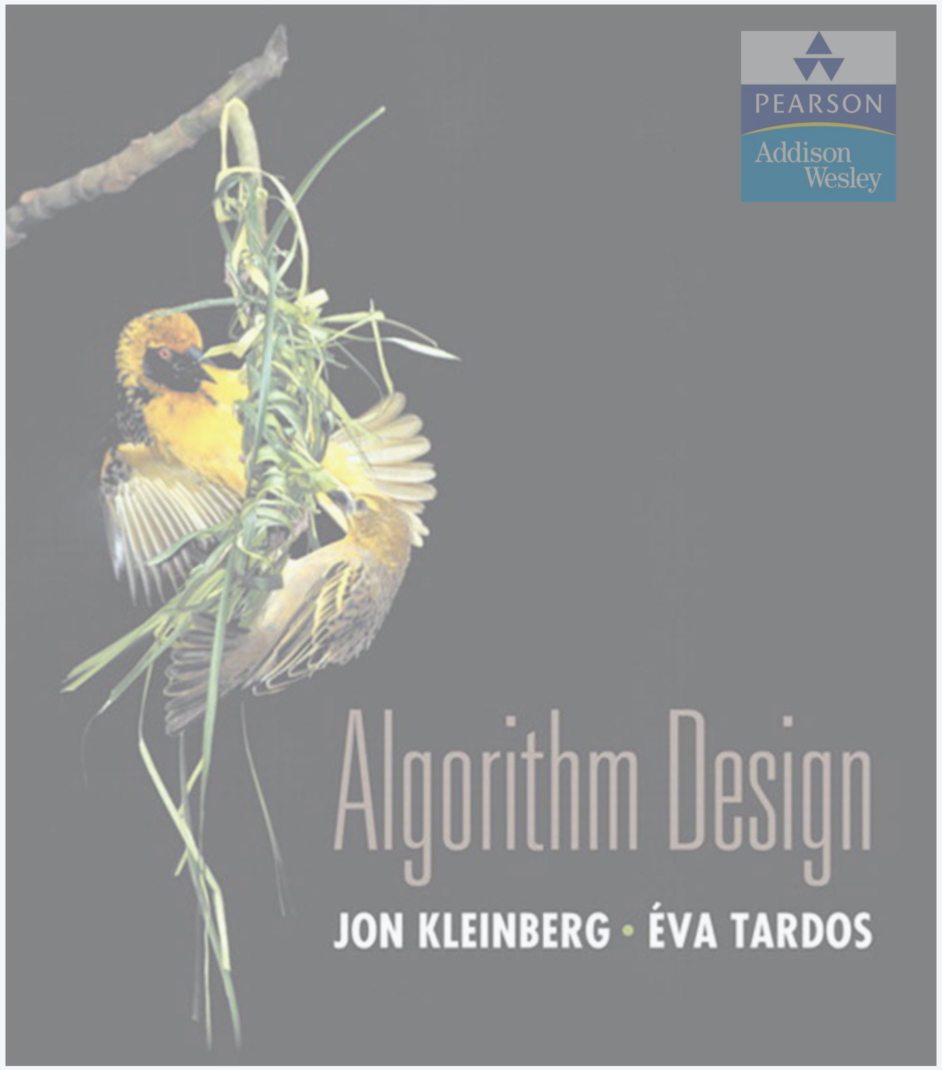
7. Network Flow I

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

Lecture slides by Kevin Wayne

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

Last updated on 4/27/18 5:51 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*
- *Dinitz' algorithm*
- *simple unit-capacity networks*

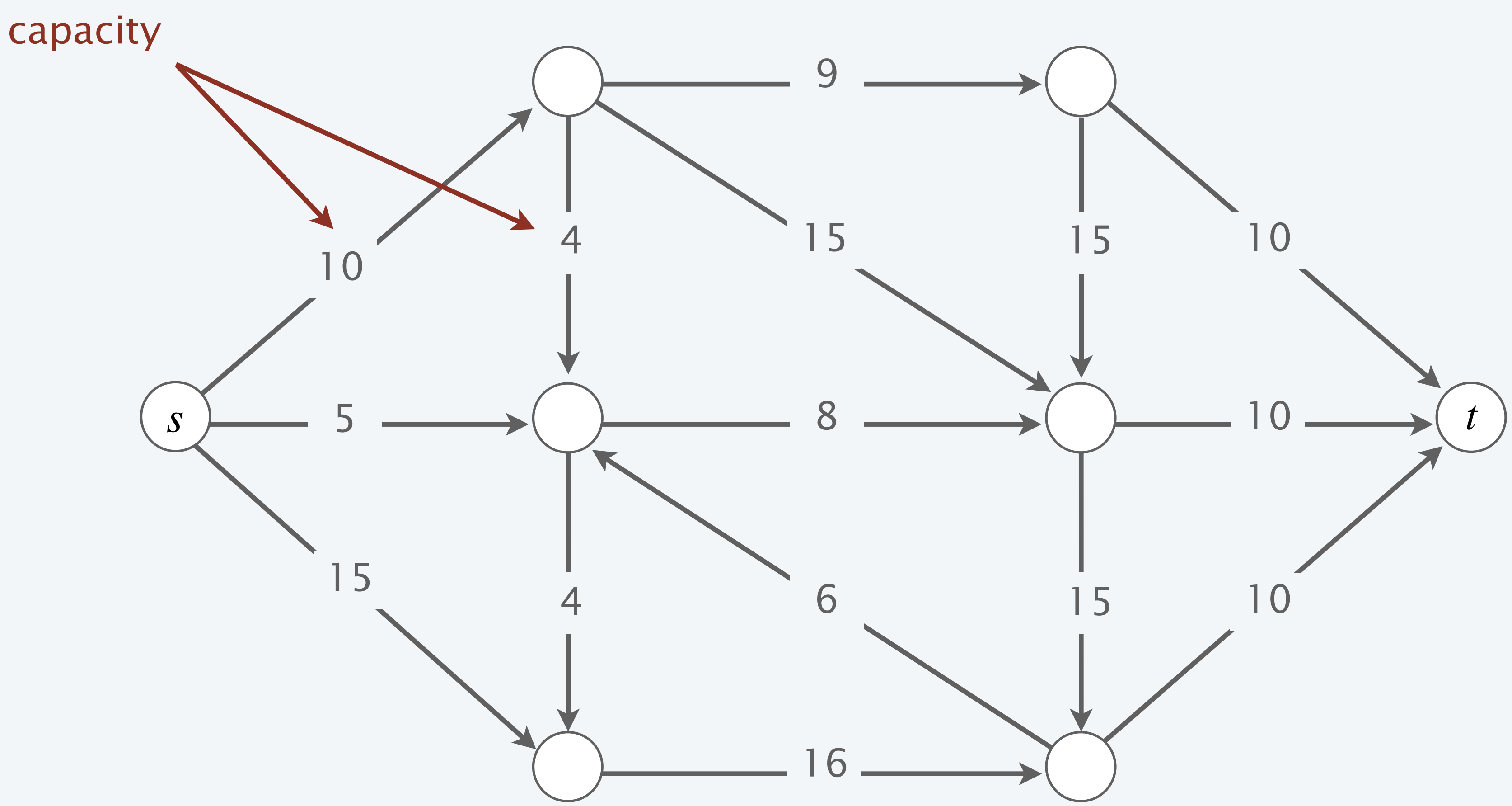
Flow network

A flow network is a tuple $G = (V, E, s, t, c)$.

- Digraph (V, E) with source $s \in V$ and sink $t \in V$.
- Capacity $c(e) > 0$ for each $e \in E$.

assume all nodes are reachable from s

Intuition. Material flowing through a transportation network; material originates at source and is sent to sink.

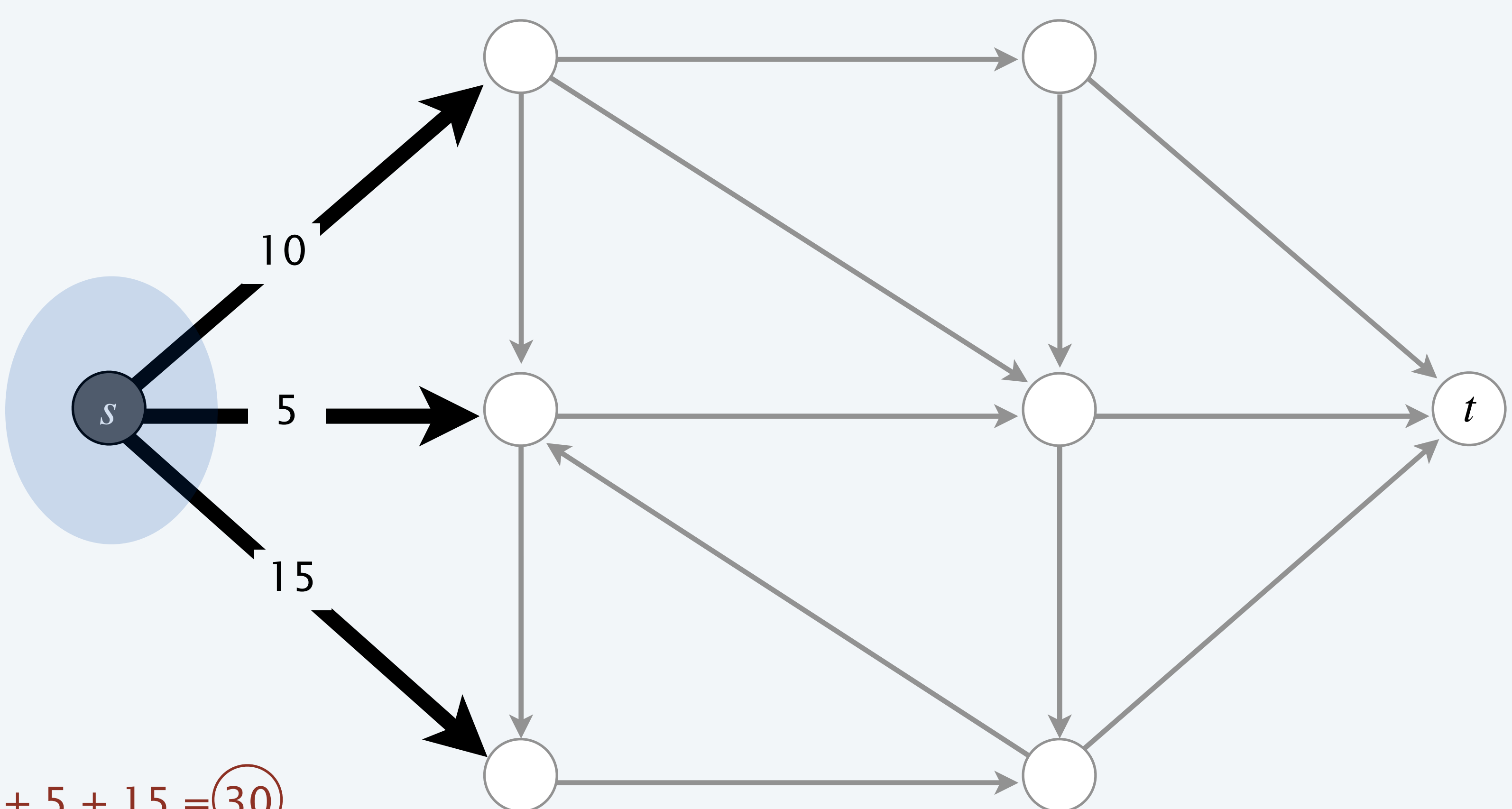


Minimum-cut problem

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

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

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



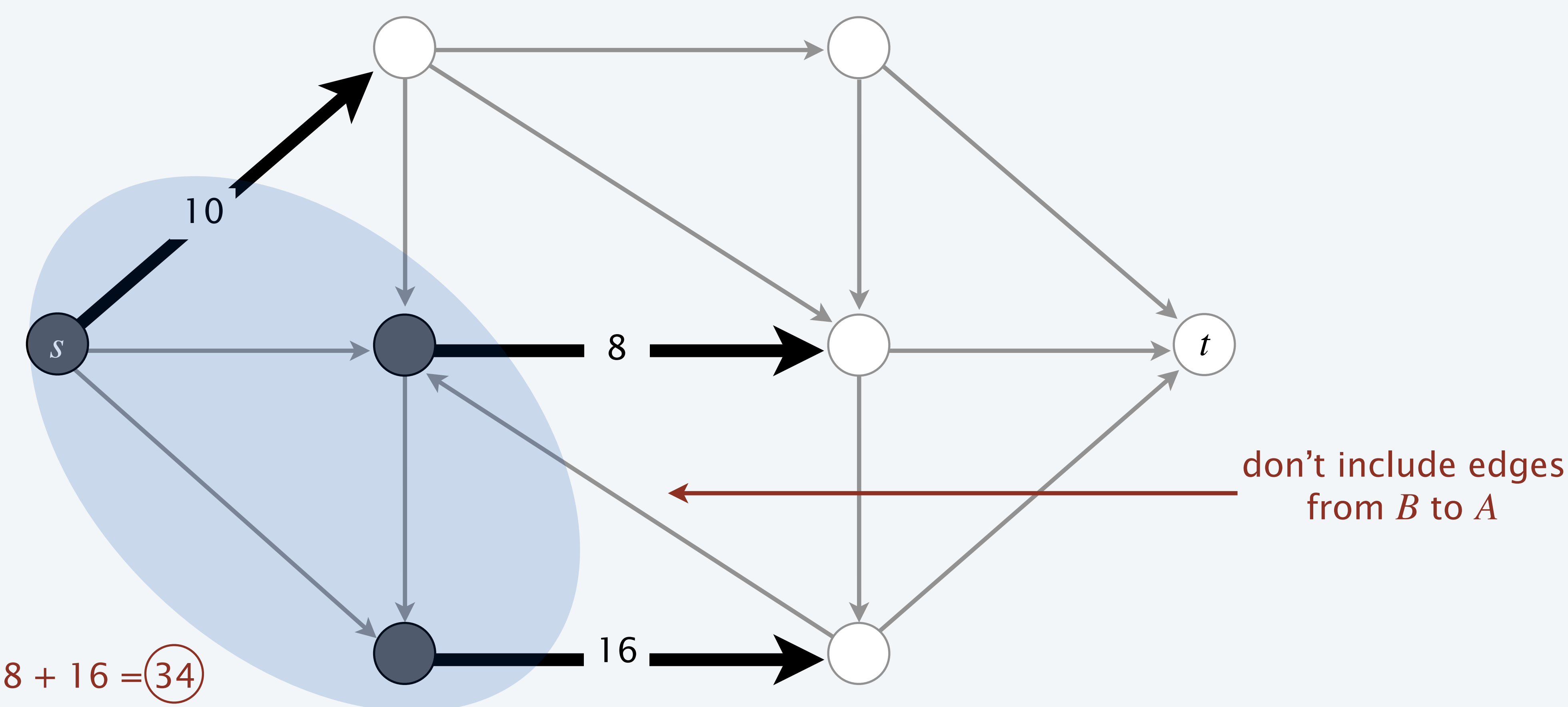
capacity = 10 + 5 + 15 = 30

Minimum-cut problem

Def. An *st*-cut (cut) is a partition (A, B) of the nodes 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 + 8 + 16 = 34

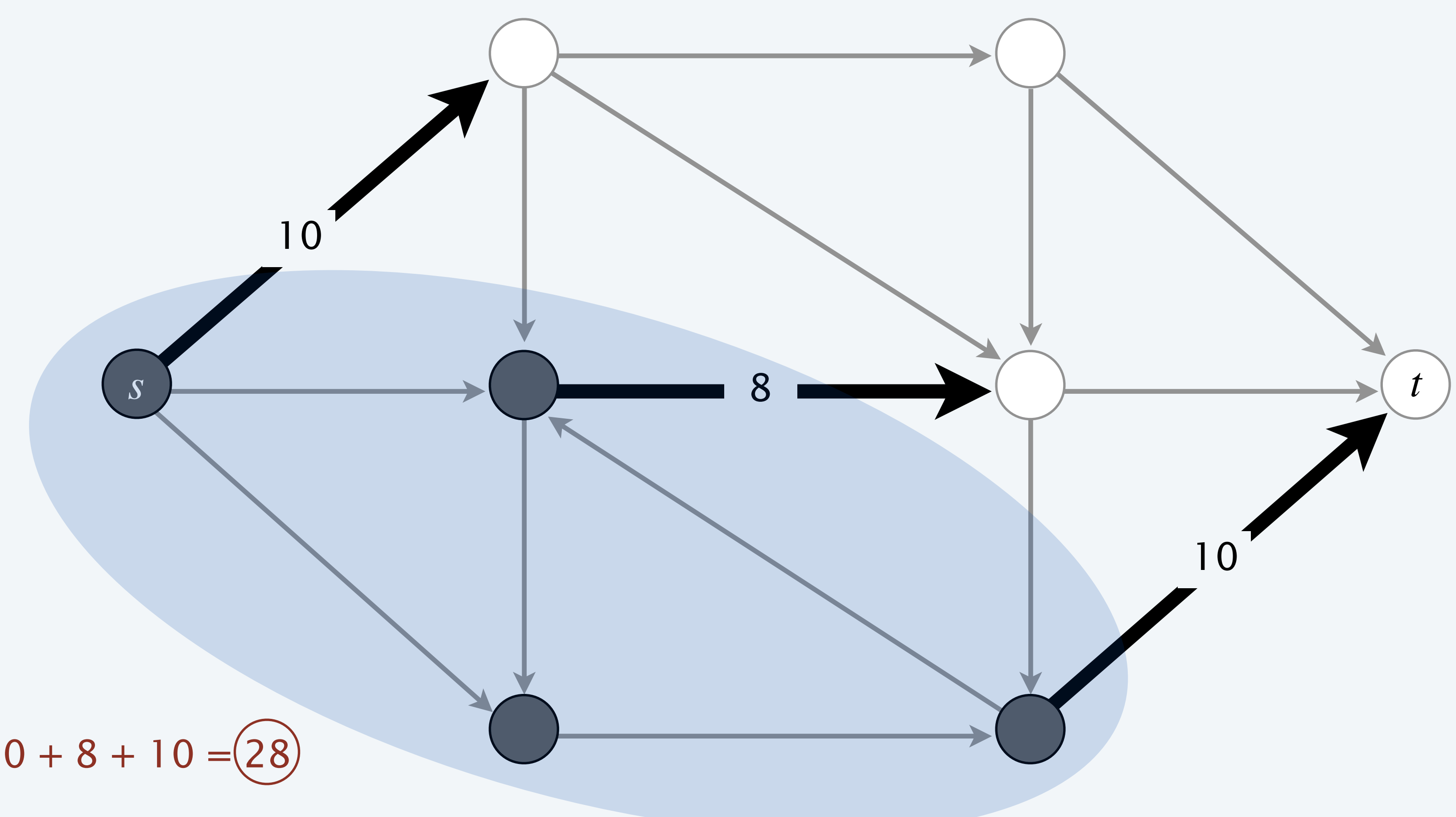
Minimum-cut problem

Def. An *st*-cut (cut) is a partition (A, B) of the nodes 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 = 28

Network flow: quiz 1

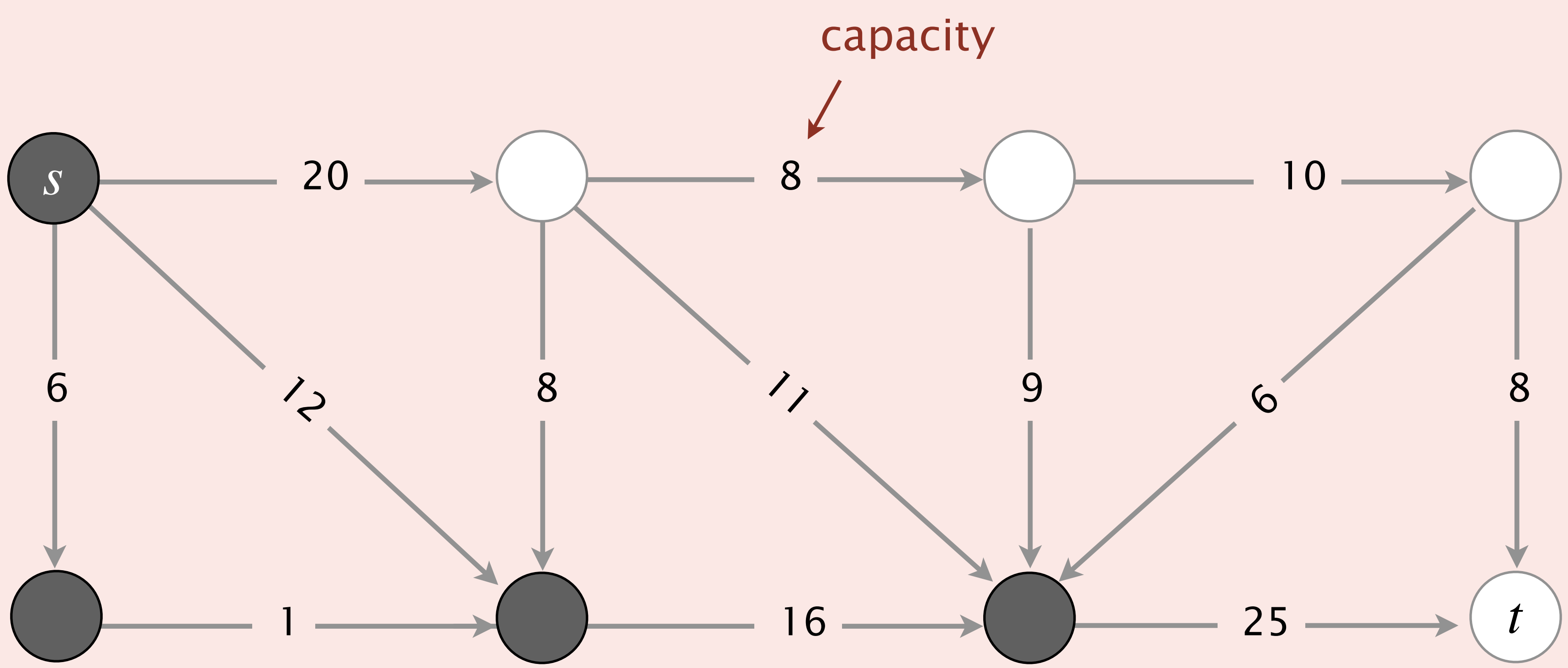
Which is the capacity of the given *st*-cut?

A. 11 $(20 + 25 - 8 - 11 - 9 - 6)$

B. 34 $(8 + 11 + 9 + 6)$

C. 45 $(20 + 25)$

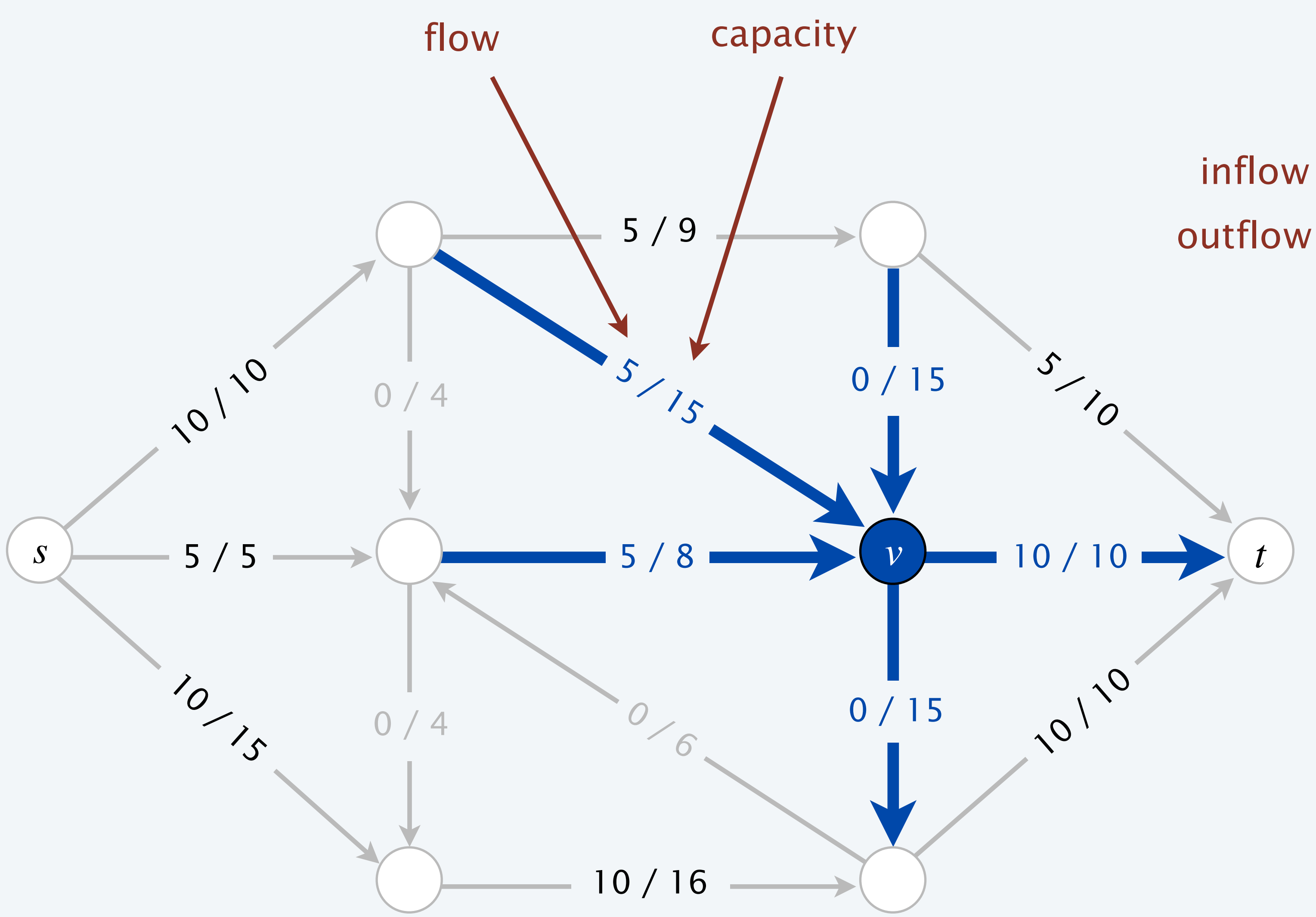
D. 79 $(20 + 25 + 8 + 11 + 9 + 6)$



Maximum-flow problem

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

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



inflow at v = $5 + 5 + 0$ = 10

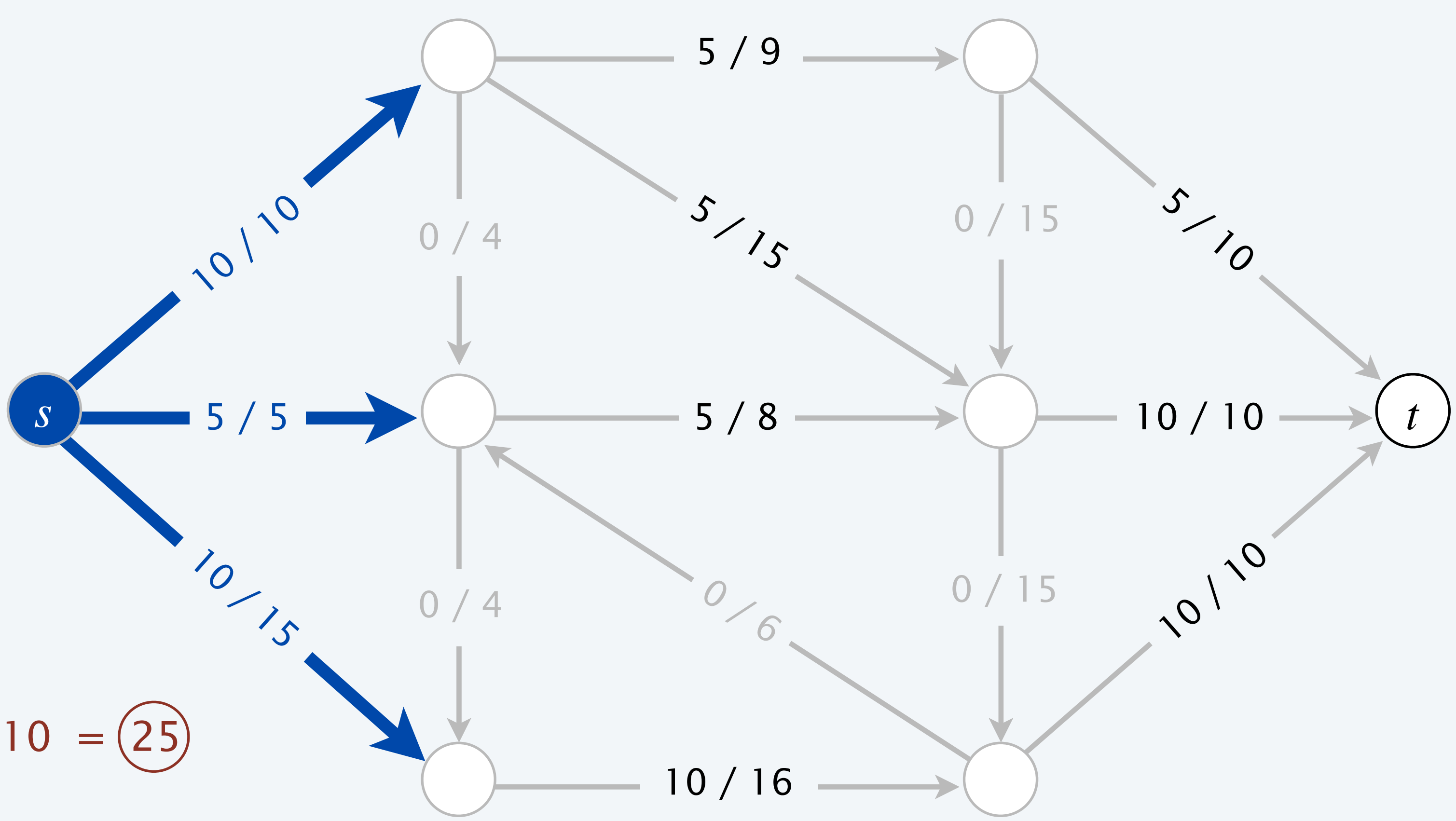
outflow at v = $10 + 0$ = 10

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 $v \in V - \{s, t\}$: $\displaystyle\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

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



value = 5 + 10 + 10 = 25

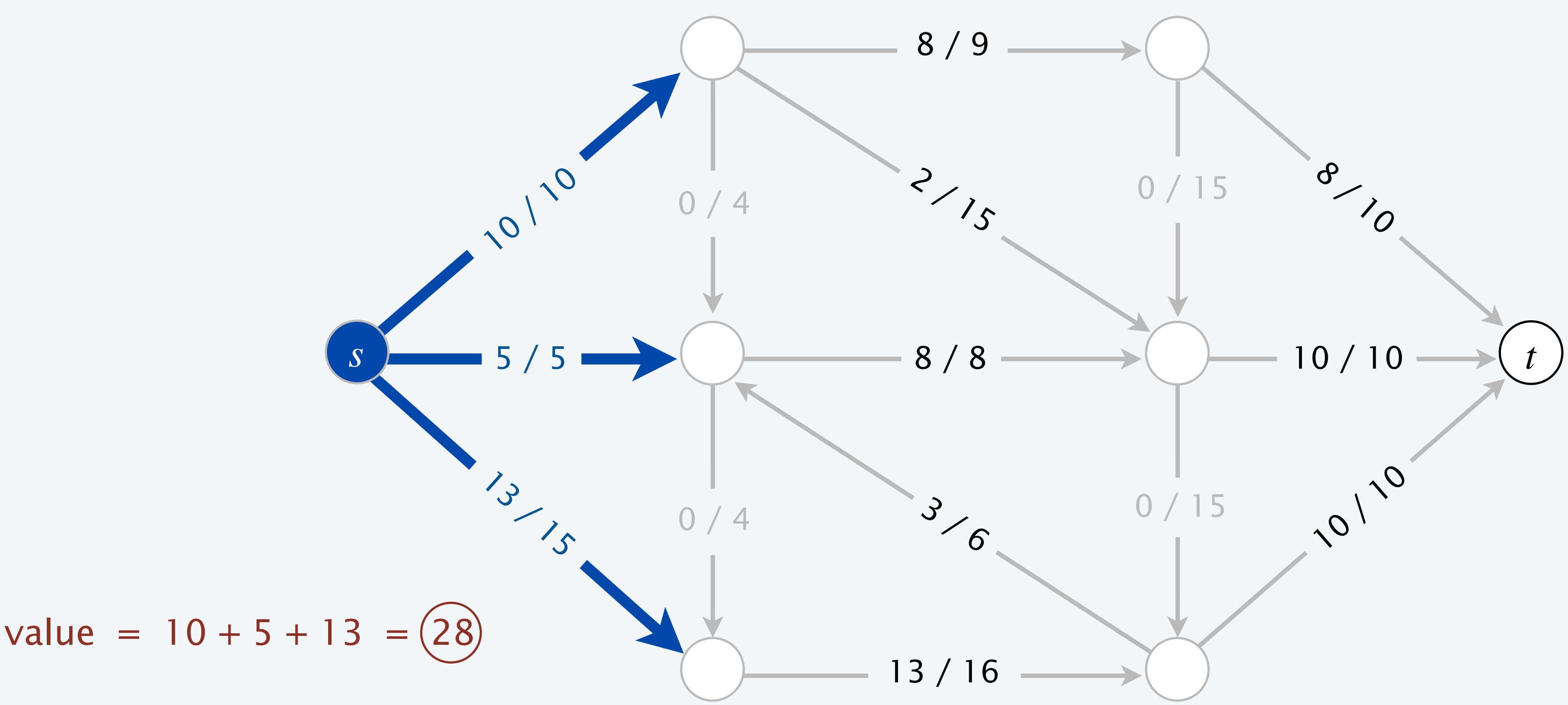
Maximum-flow problem

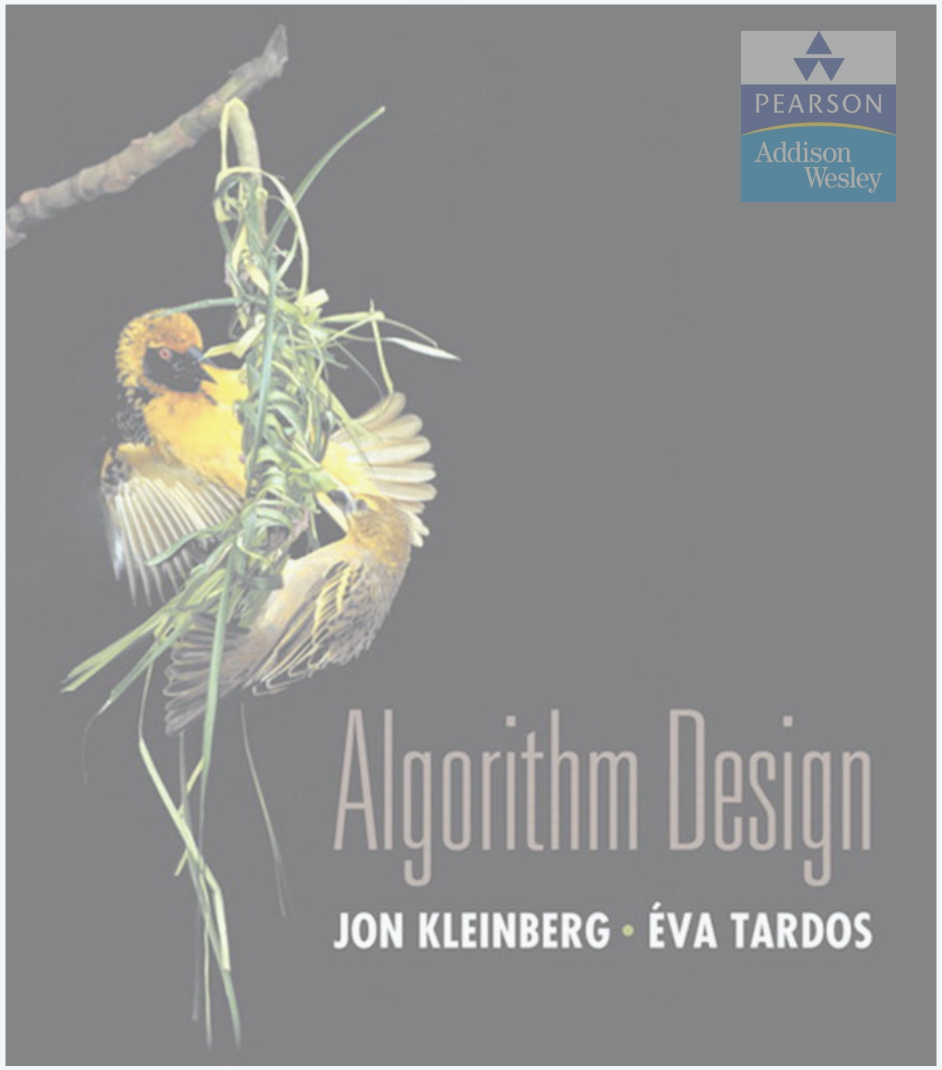
Def. An *st*-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [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: $val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$

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





SECTION 7.1

7. NETWORK FLOW I

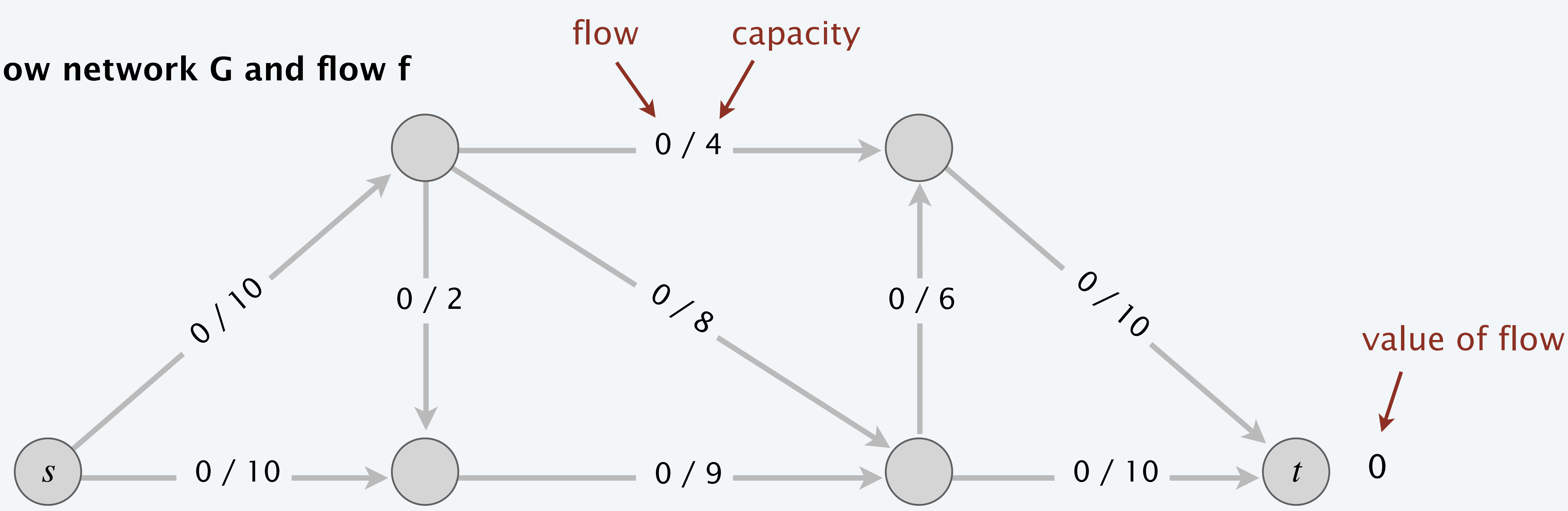
- *max-flow and min-cut problems*
- *Ford–Fulkerson algorithm*
- *max-flow min-cut theorem*
- *capacity-scaling algorithm*
- *shortest augmenting paths*
- *Dinitz' algorithm*
- *simple unit-capacity networks*

Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each 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.

flow network G and flow f

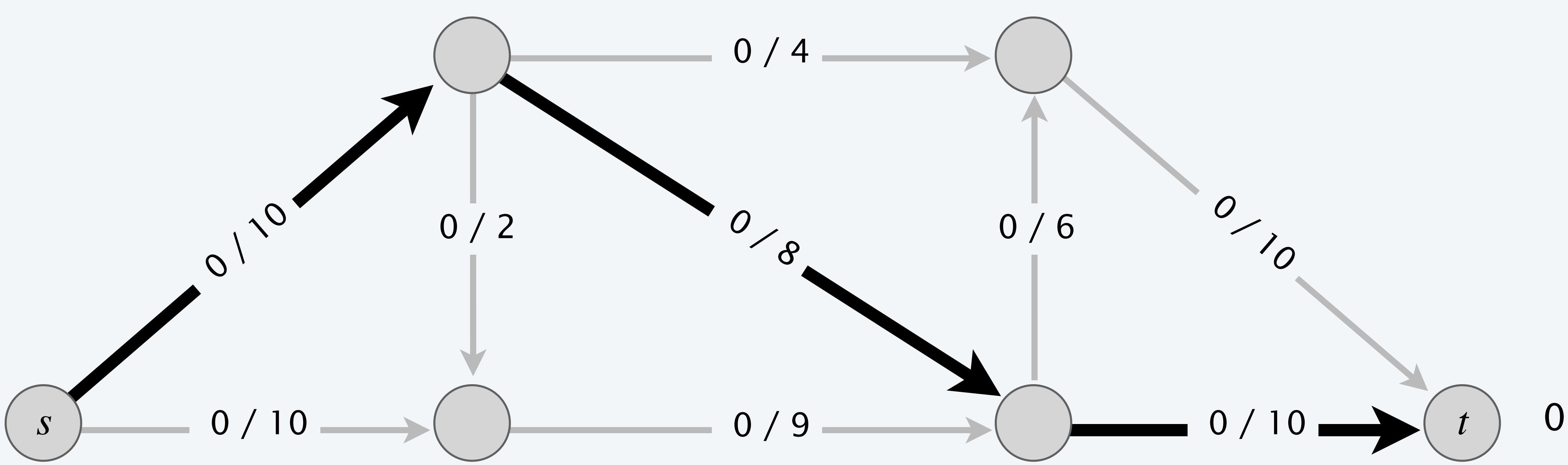


Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each 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.

flow network G and flow f

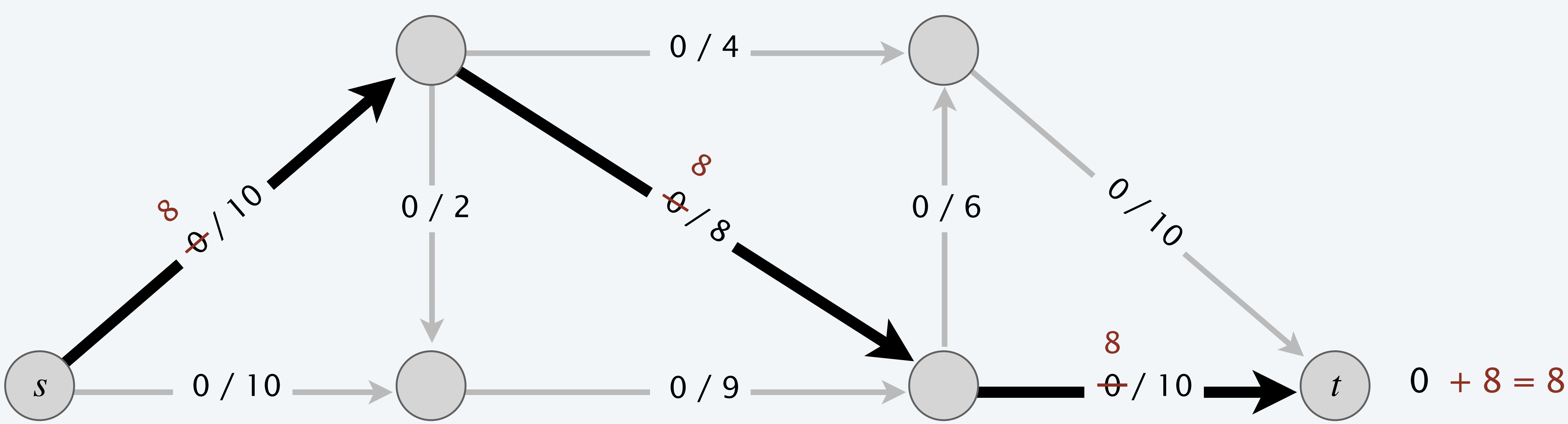


Toward a max-flow algorithm

Greedy algorithm.

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

flow network G and flow f

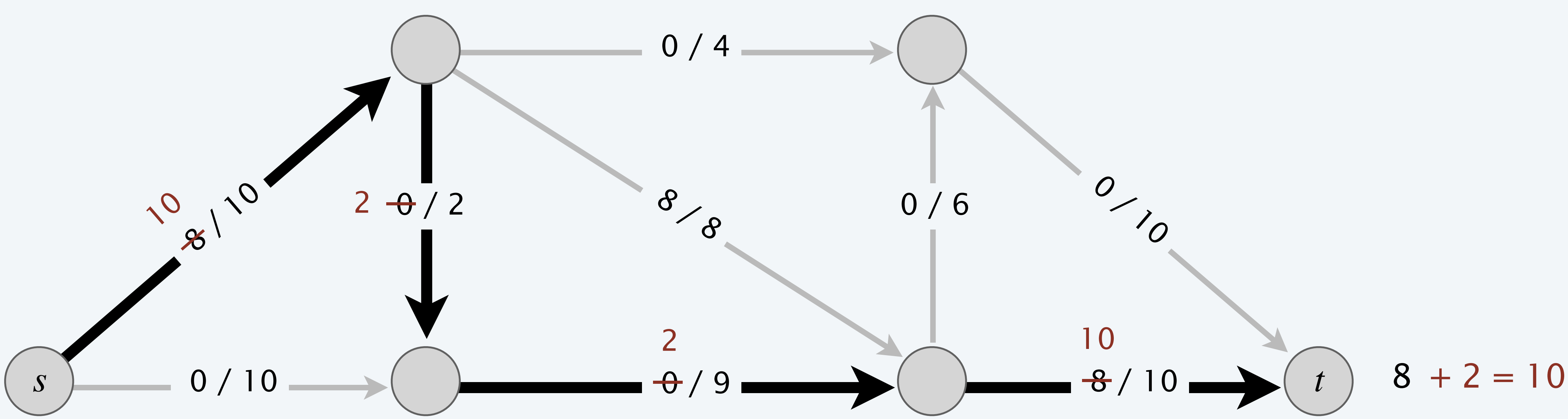


Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each 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.

flow network G and flow f

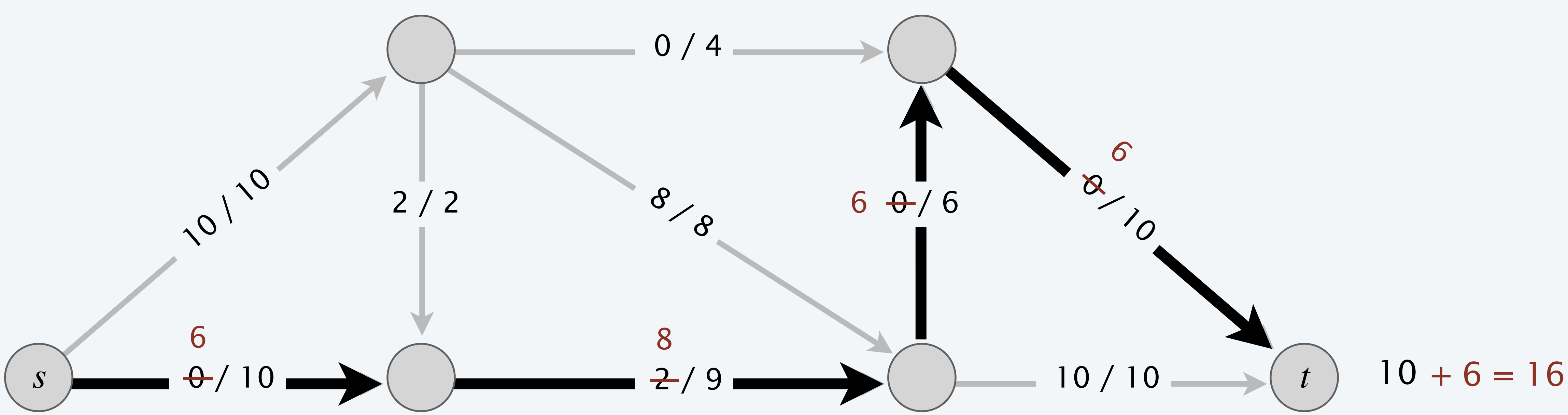


Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each 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.

flow network G and flow f



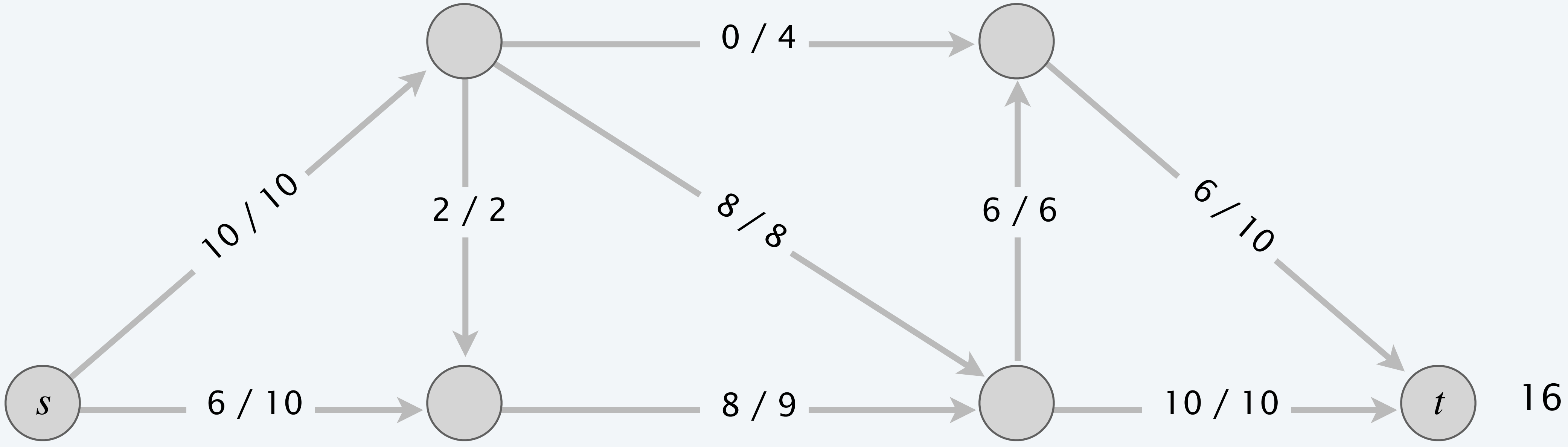
Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each 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

flow network G and flow f



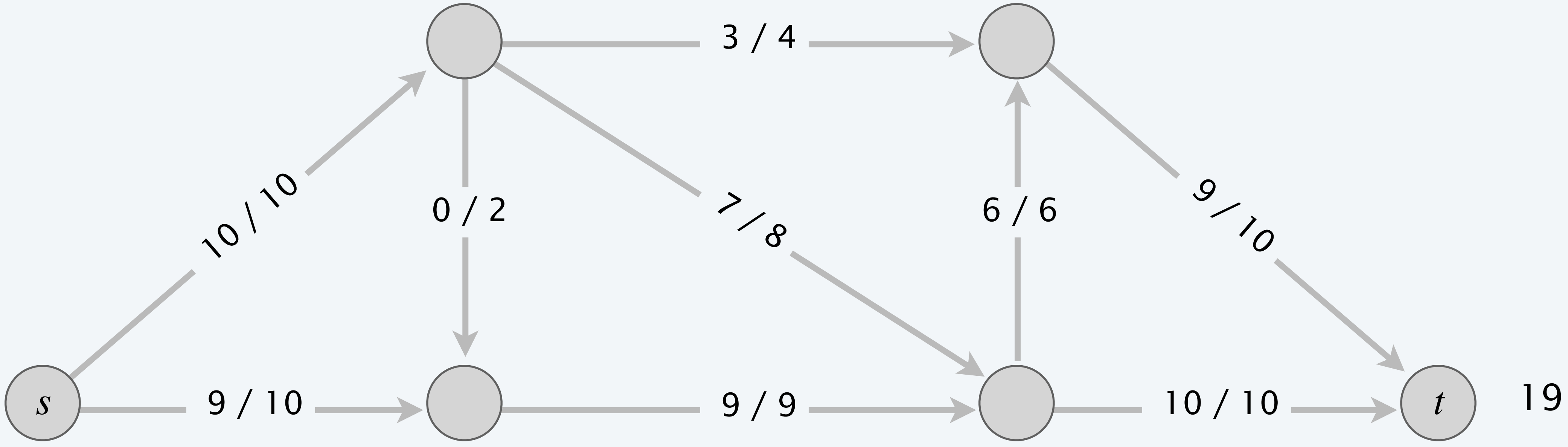
Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each 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

flow network G and flow f



Why the greedy algorithm fails

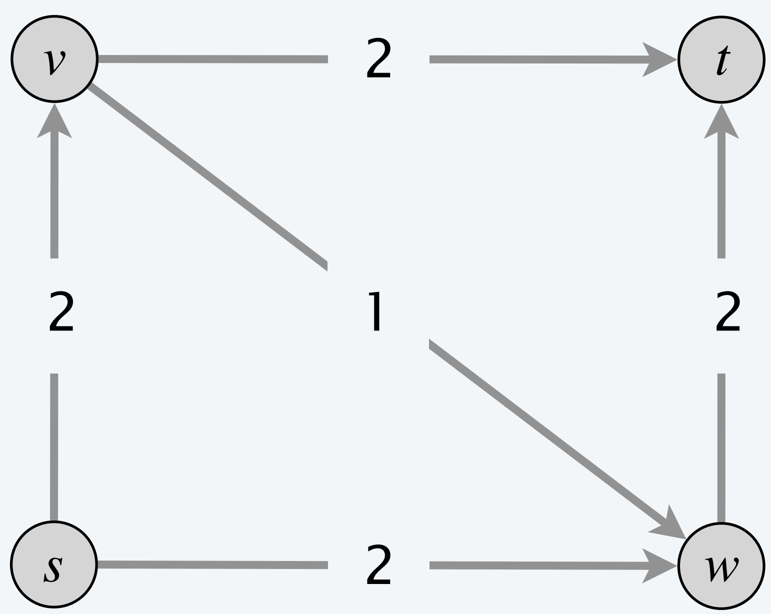
Q. Why does the greedy algorithm fail?

A. Once greedy algorithm increases flow on an edge, it never decreases it.

Ex. Consider flow network G.

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

flow network G



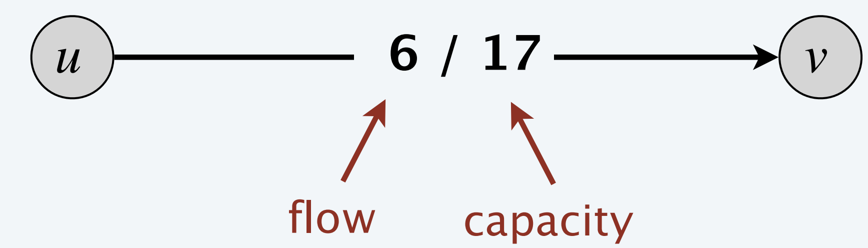
Bottom line. Need some mechanism to “undo” a bad decision.

Residual network

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

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

original flow network G



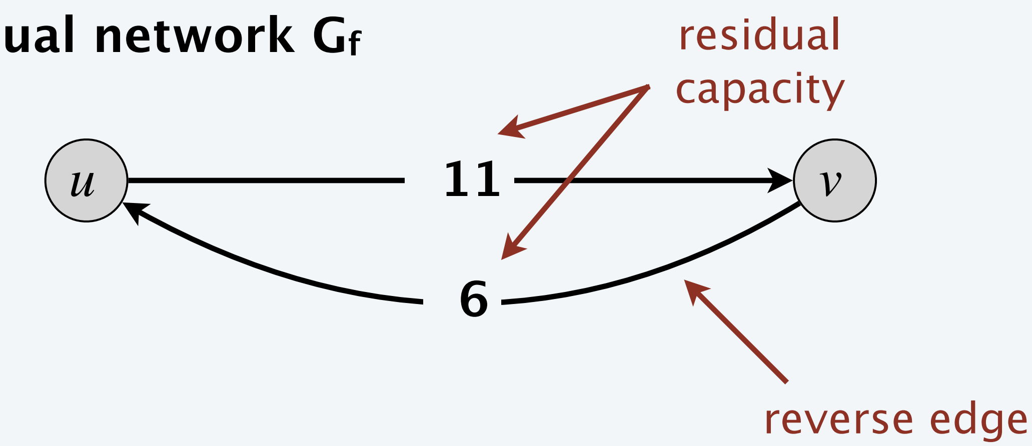
Reverse edge. $e^{\text{reverse}} = (v, u)$.

- "Undo" flow sent.

Residual capacity.

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

residual network G_f



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

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

Def. The bottleneck capacity of an augmenting path 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, after calling $f' \leftarrow$ AUGMENT(f, c, P), the resulting f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$.

AUGMENT(f, c, P)

$\delta \leftarrow$ bottleneck capacity of augmenting path P.

FOREACH edge $e \in P$:

  IF ($e \in E$) $f(e) \leftarrow f(e) + \delta$.

  ELSE $f(e^{\text{reverse}}) \leftarrow f(e^{\text{reverse}}) - \delta$.

RETURN f.

Network flow: quiz 2

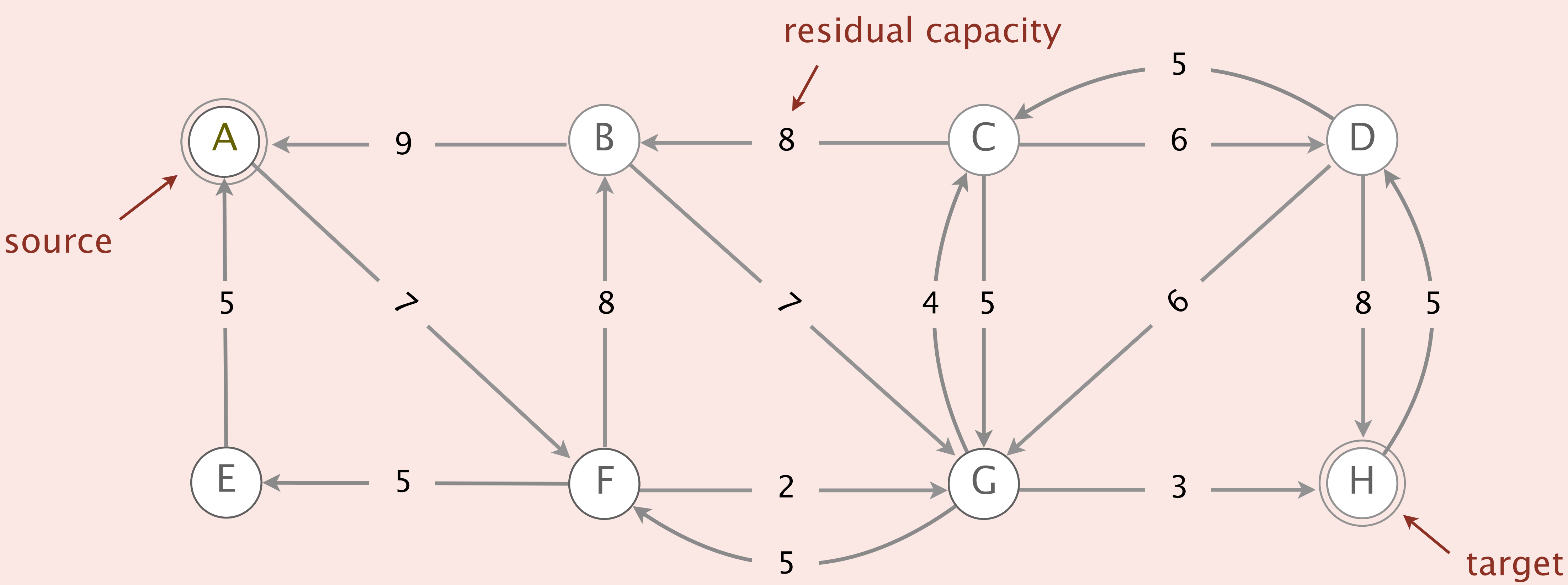
Which is the augmenting path of highest bottleneck capacity?

A. $A \rightarrow F \rightarrow G \rightarrow H$

B. $A \rightarrow B \rightarrow C \rightarrow D \rightarrow H$

C. $A \rightarrow F \rightarrow B \rightarrow G \rightarrow H$

D. $A \rightarrow F \rightarrow B \rightarrow G \rightarrow C \rightarrow D \rightarrow H$



Ford–Fulkerson algorithm

Ford–Fulkerson augmenting path algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P in the residual network G_f.
- Augment flow along path P.
- Repeat until you get stuck.

FORD–FULKERSON(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

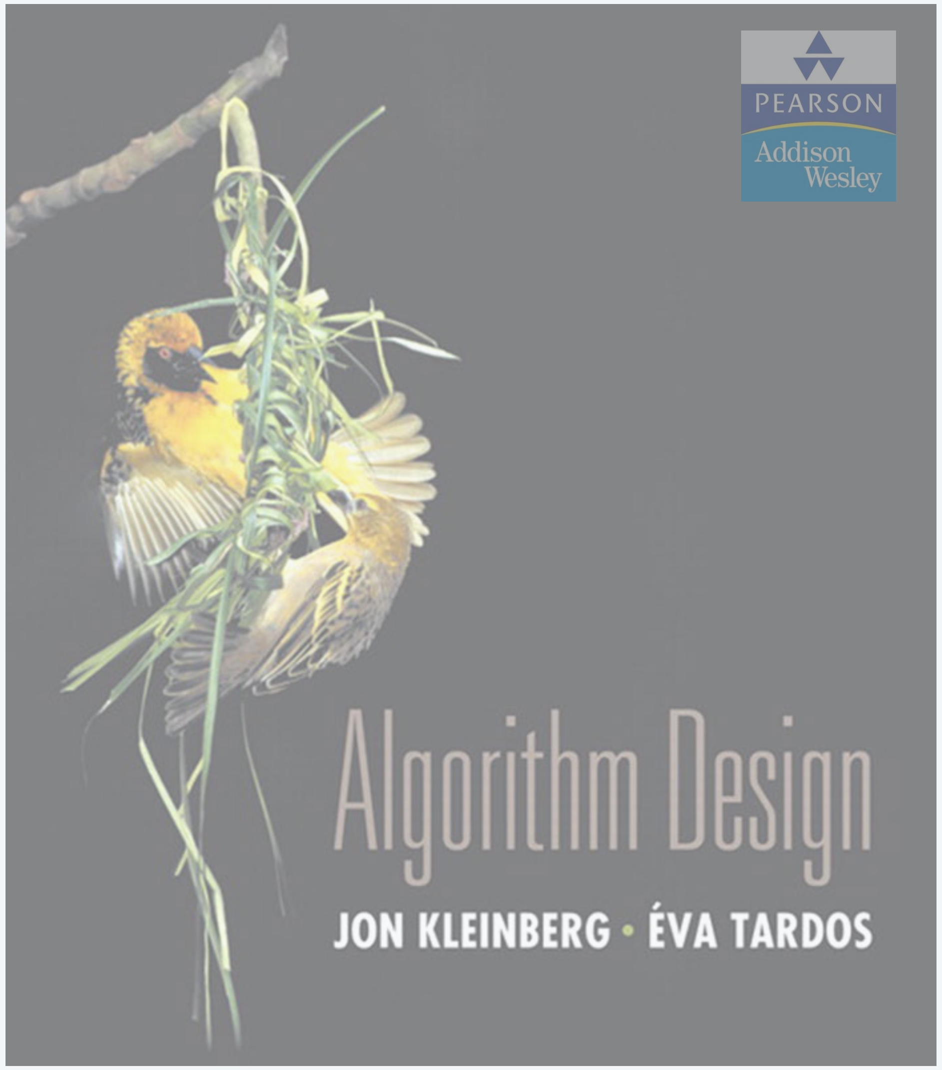
WHILE (there exists an s⇝t path P in G_f)

  $f \leftarrow$ AUGMENT(f, c, P).

  Update G_f.

RETURN f.

augmenting path



SECTION 7.2

7. NETWORK FLOW I

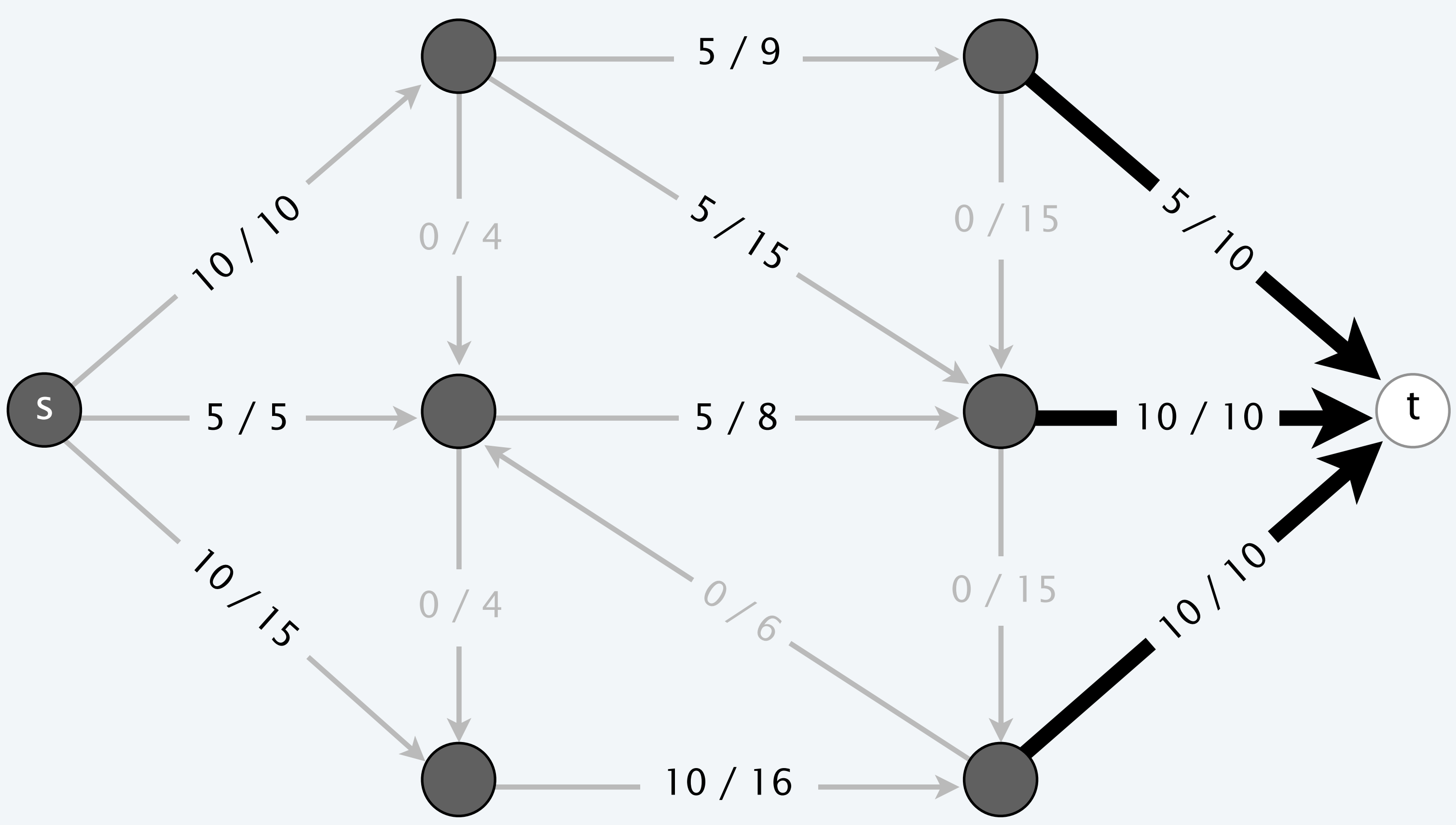
- max-flow and min-cut problems
- Ford–Fulkerson algorithm
- **max-flow min-cut theorem**
- capacity-scaling algorithm
- shortest augmenting paths
- Dinitz' algorithm
- simple unit-capacity networks

Relationship between flows and cuts

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

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

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



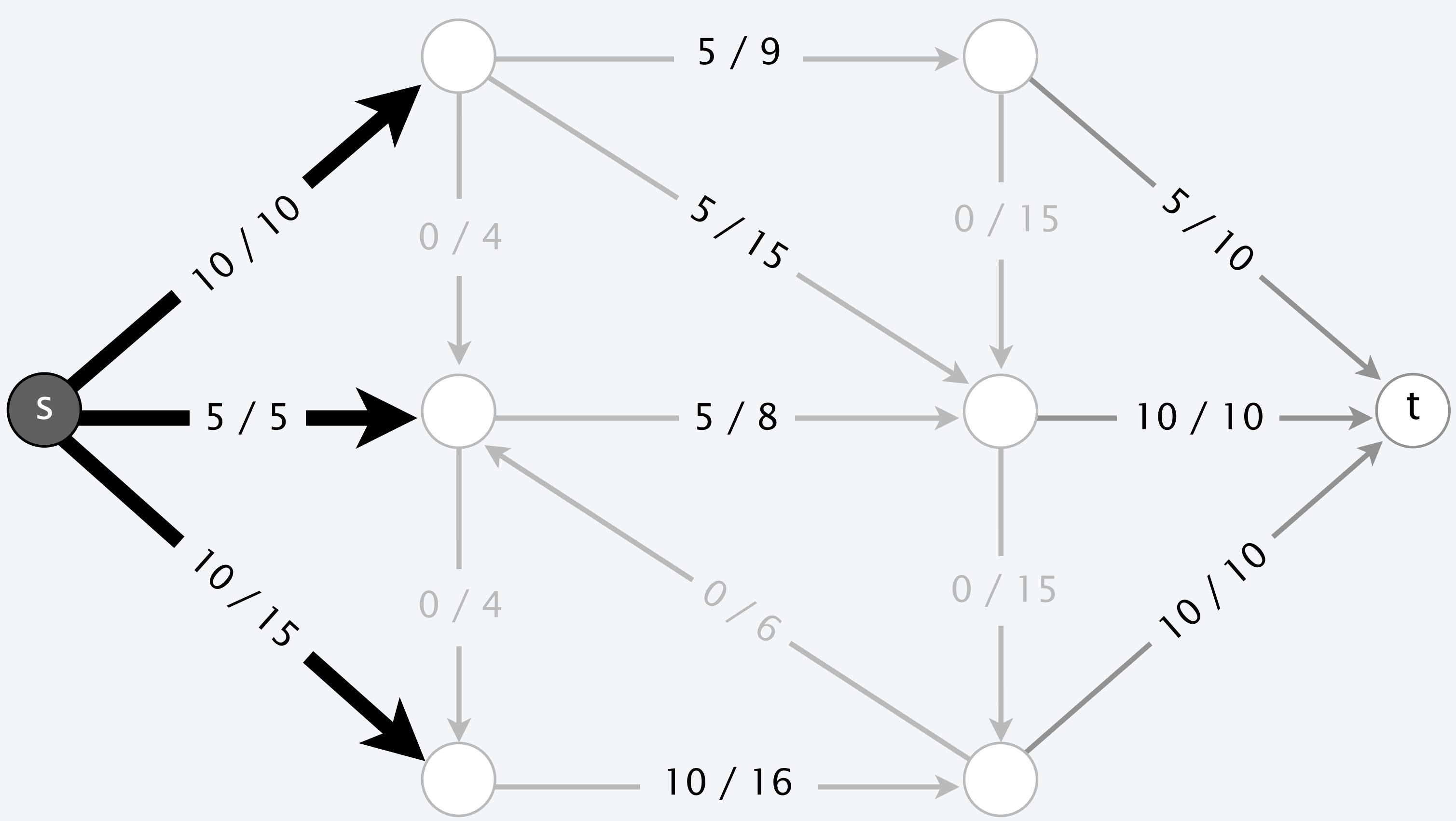
value of flow = 25

Relationship between flows and cuts

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

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

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



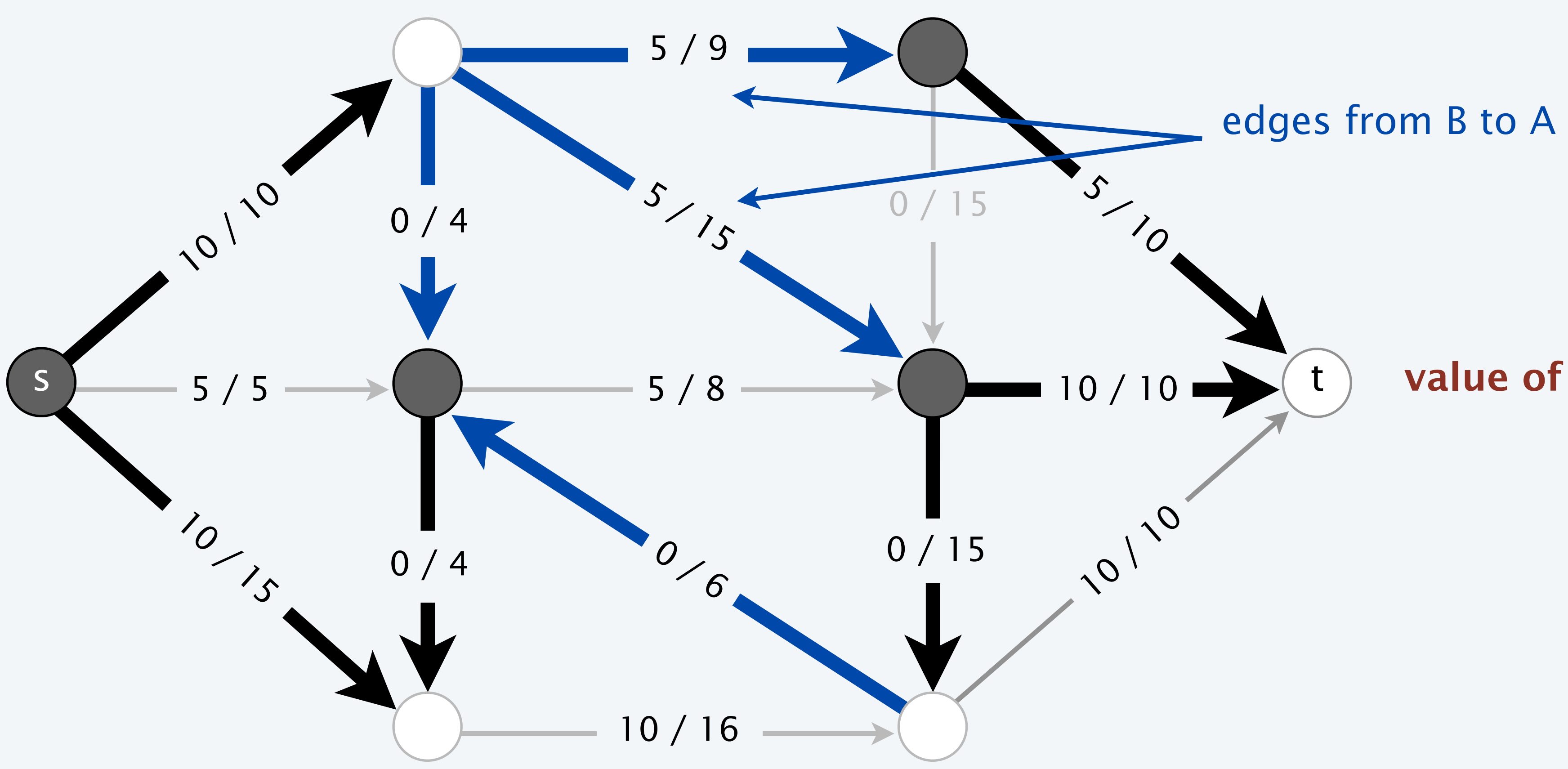
value of flow = 25

Relationship between flows and cuts

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

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

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



value of flow = 25

Network flow: quiz 3

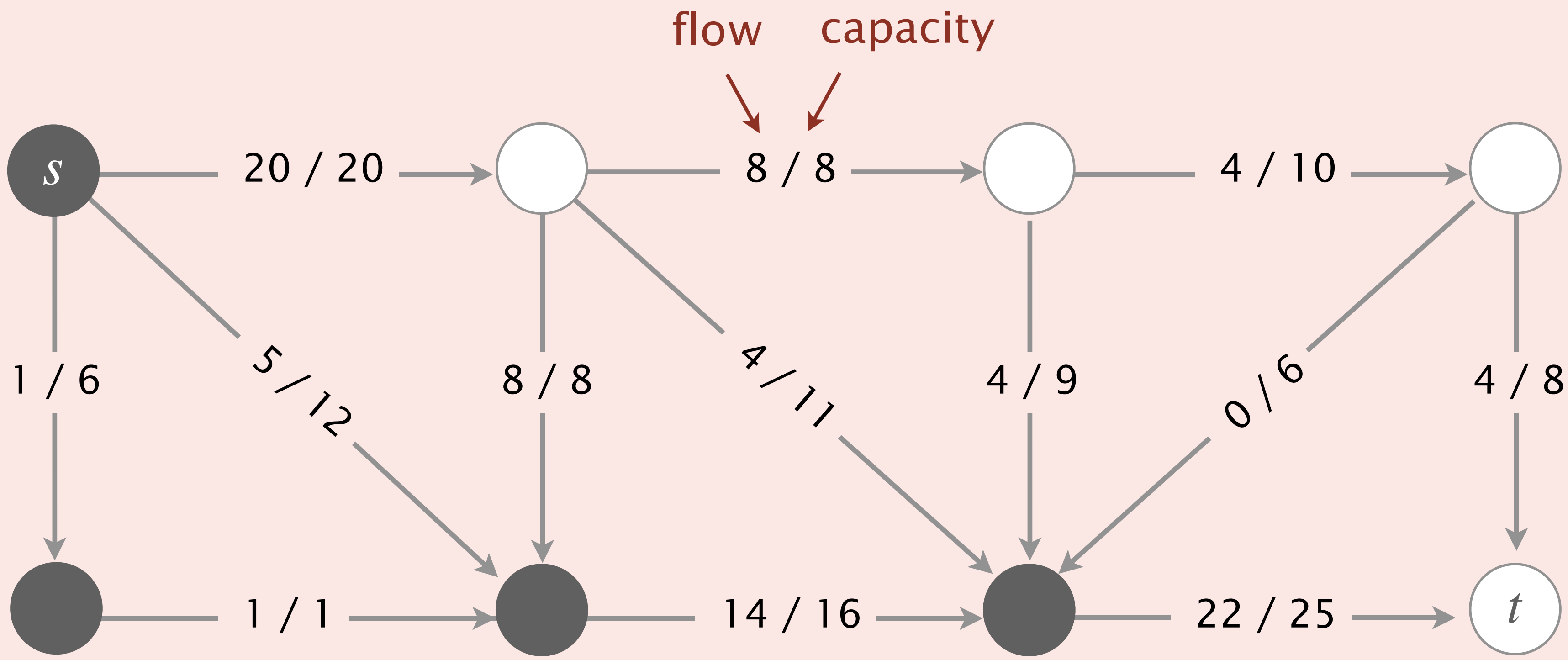
Which is the net flow across the given cut?

A. 11 (20 + 25 – 8 – 11 – 9 – 6)

B. 26 (20 + 22 – 8 – 4 – 4)

C. 42 (20 + 22)

D. 45 (20 + 25)



Relationship between flows and cuts

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

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

Pf.

$$\begin{aligned} val(f) &= \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e) \\ &= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right) \\ &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \quad \blacksquare \end{aligned}$$

by flow conservation, all terms except for $v = s$ are 0 →

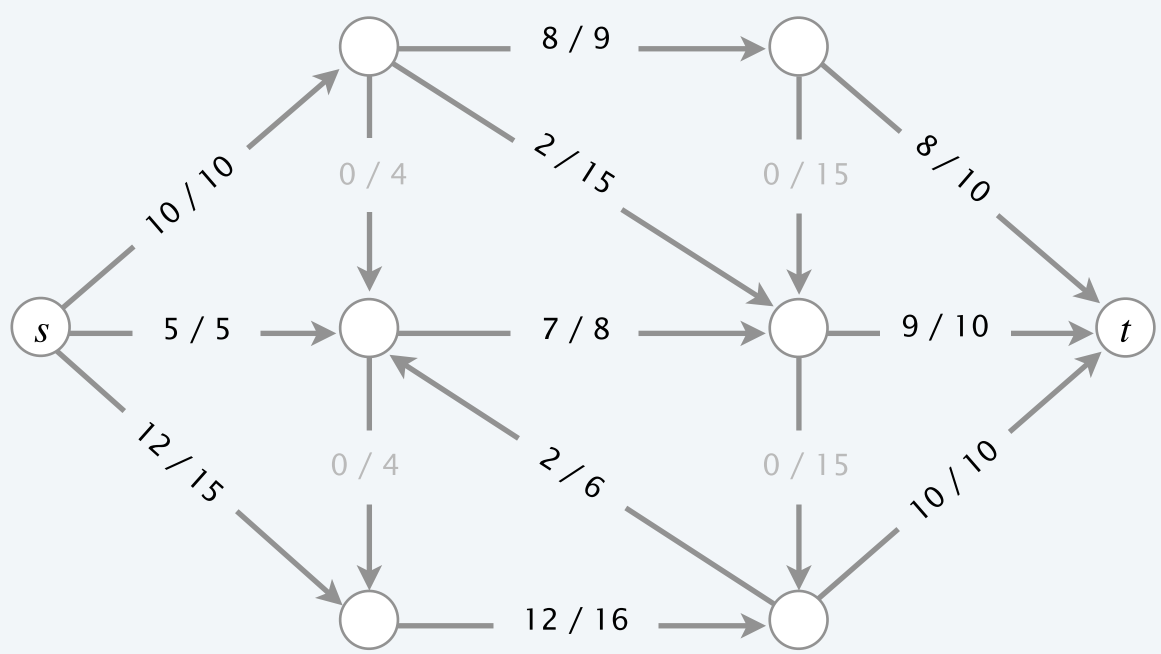
Relationship between flows and cuts

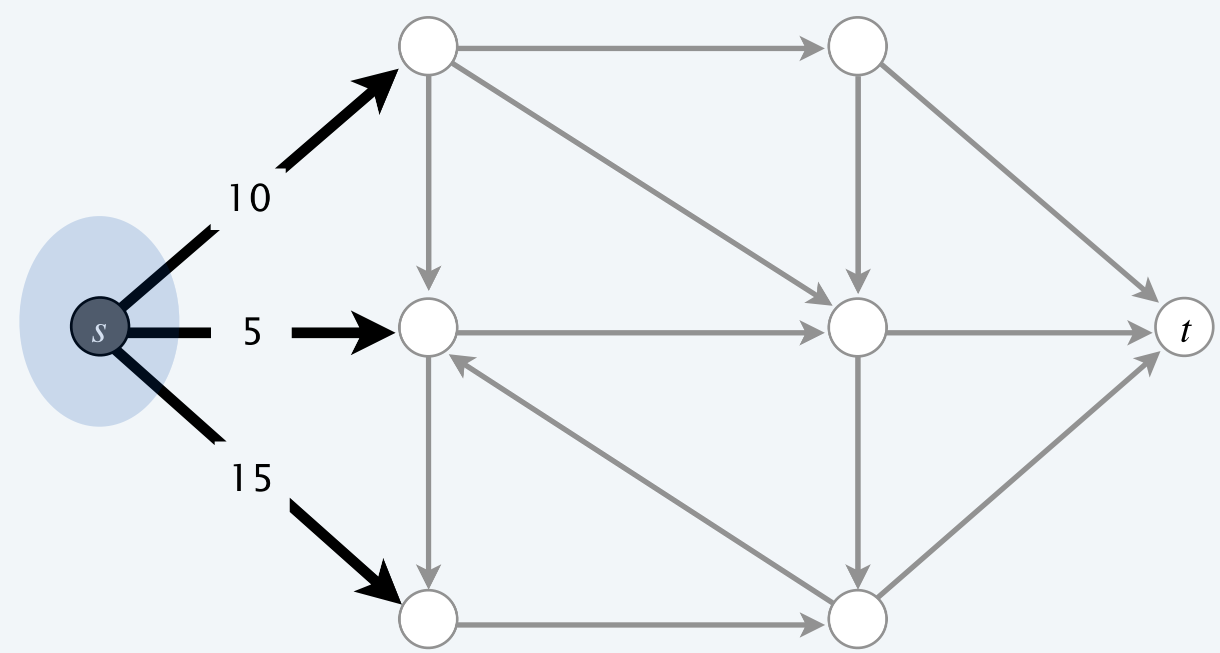
Weak duality. Let f be any flow and (A, B) be any cut. Then, $val(f) \leq cap(A, B)$.

Pf.

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

flow value lemma





value of flow = 27 $\leq$ **capacity of cut = 30**

Certificate of optimality

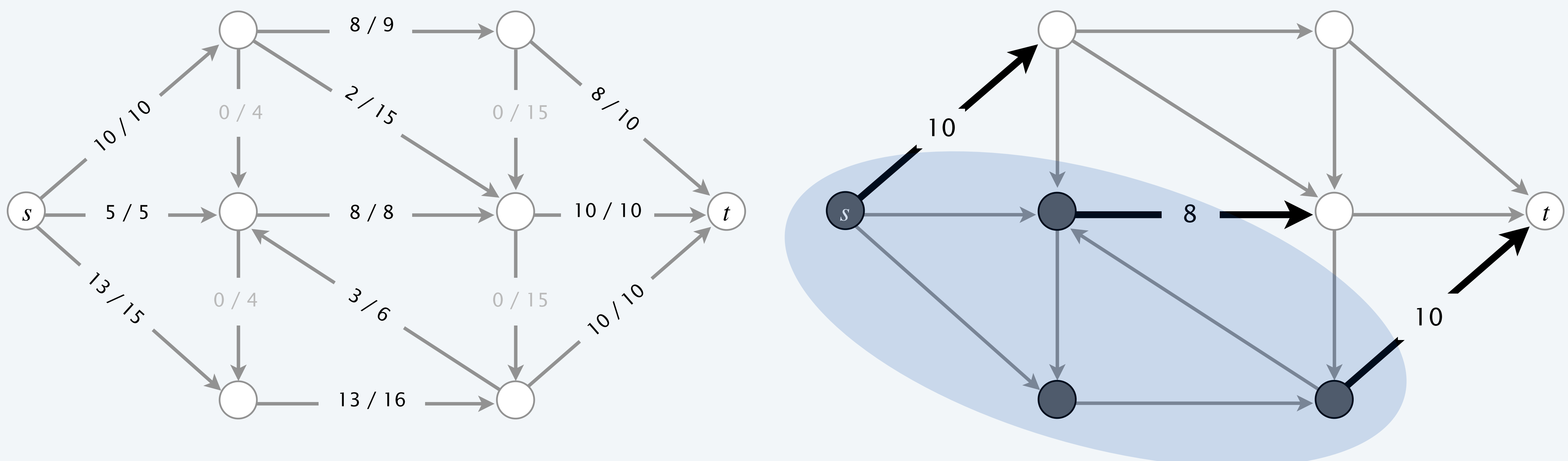
Corollary. Let f be a flow and let (A, B) be any cut.
If $val(f) = cap(A, B)$, then f is a max flow and (A, B) is a min cut.

Pf.

- 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

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

Max-flow min-cut theorem

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

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

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. ← if Ford–Fulkerson terminates, then f is max flow

[i ⇒ ii]

- This is the weak duality corollary. ▪

Max-flow min-cut theorem

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

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

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 network G_f.
- By definition of A: $s \in A$.
- By definition of flow f: $t \notin A$.

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

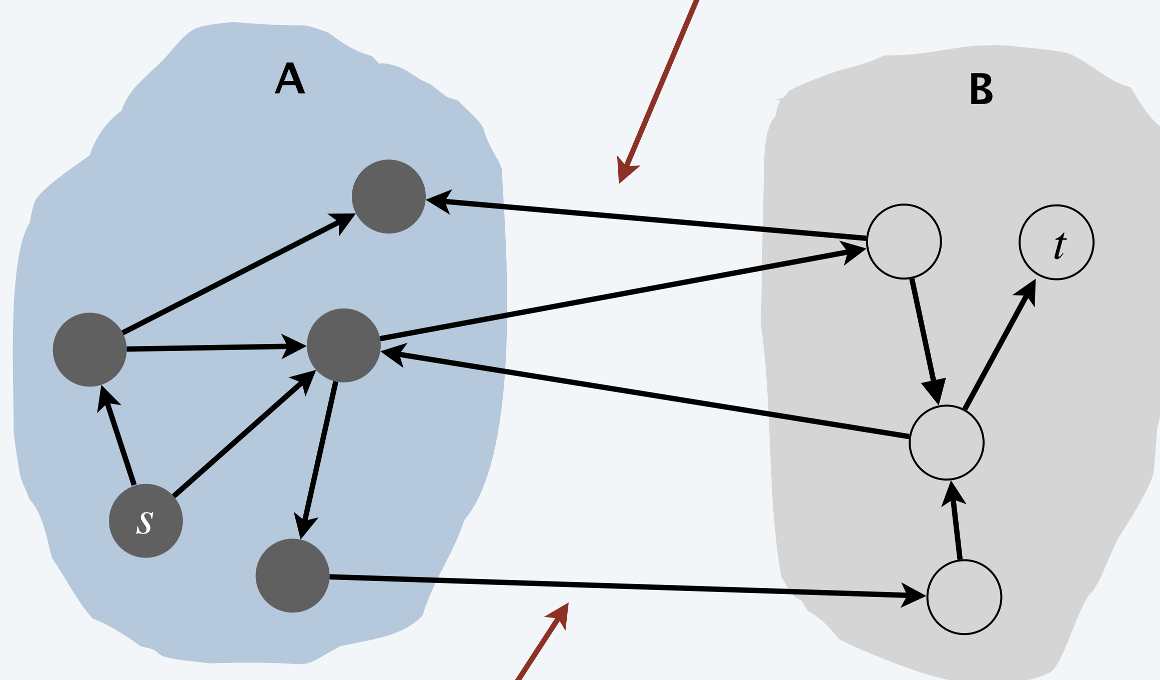
flow value lemma

$$= \sum_{e \text{ out of } A} c(e) - 0$$

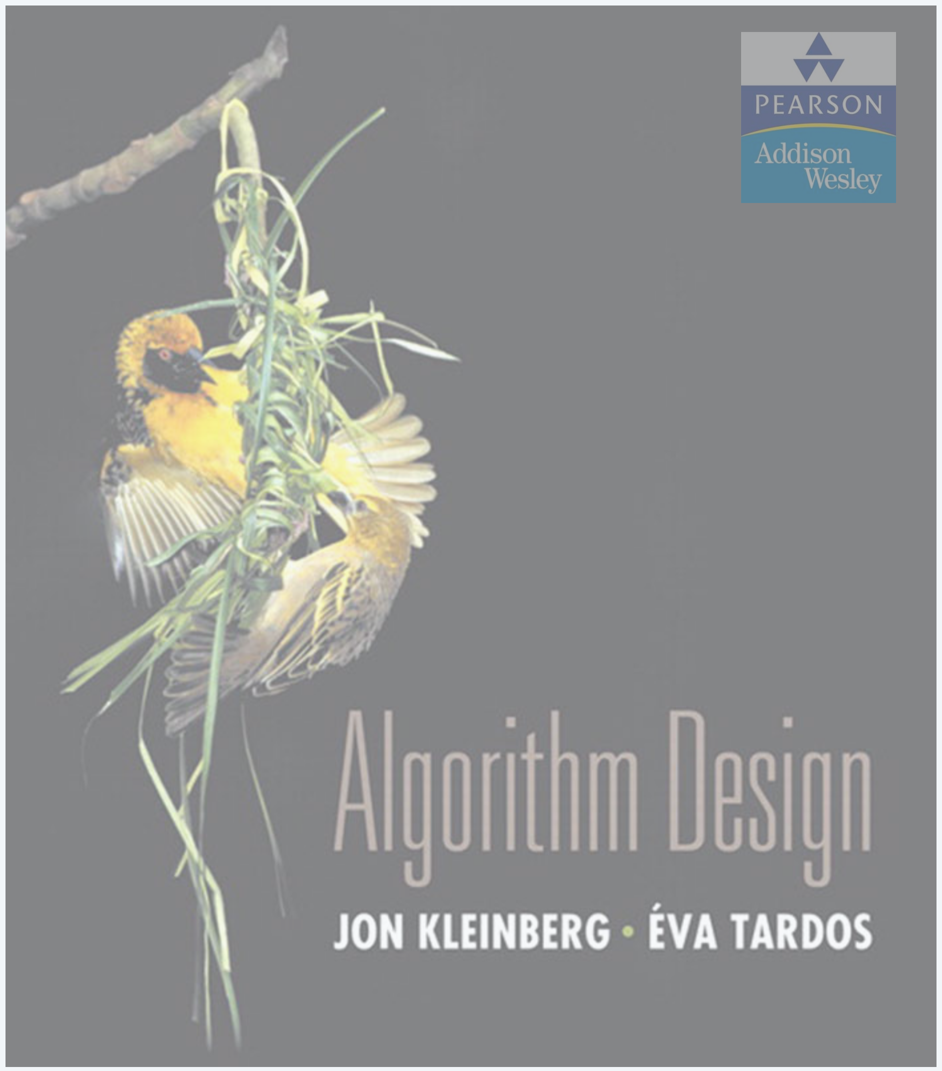
$$= cap(A, B) \quad ▪$$

edge $e = (v, w)$ with $v \in B, w \in A$ must have $f(e) = 0$

original flow network G



edge $e = (v, w)$ with $v \in A, w \in B$ 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*
- *Dinitz' algorithm*
- *simple unit-capacity networks*

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Assumption. Every edge capacity $c(e)$ is an integer between 1 and C.

Integrality invariant. Throughout Ford–Fulkerson, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

Pf. By induction on the number of augmenting paths. ▪

consider cut $A = \{ s \}$
(assumes no parallel edges)

Theorem. Ford–Fulkerson terminates after at most $val(f^*) \le nC$ augmenting paths, where f^* is a max flow.

Pf. Each augmentation increases the value of the flow by at least 1. ▪

Corollary. The running time of Ford–Fulkerson is $O(mnC)$.

Pf. Can use either BFS or DFS to find an augmenting path in $O(m)$ time. ▪

$f(e)$ is an integer for every e

Integrality theorem. There exists an integral max flow f^*.

Pf. Since Ford–Fulkerson terminates, theorem follows from integrality invariant (and augmenting path theorem). ▪

Ford–Fulkerson: exponential example

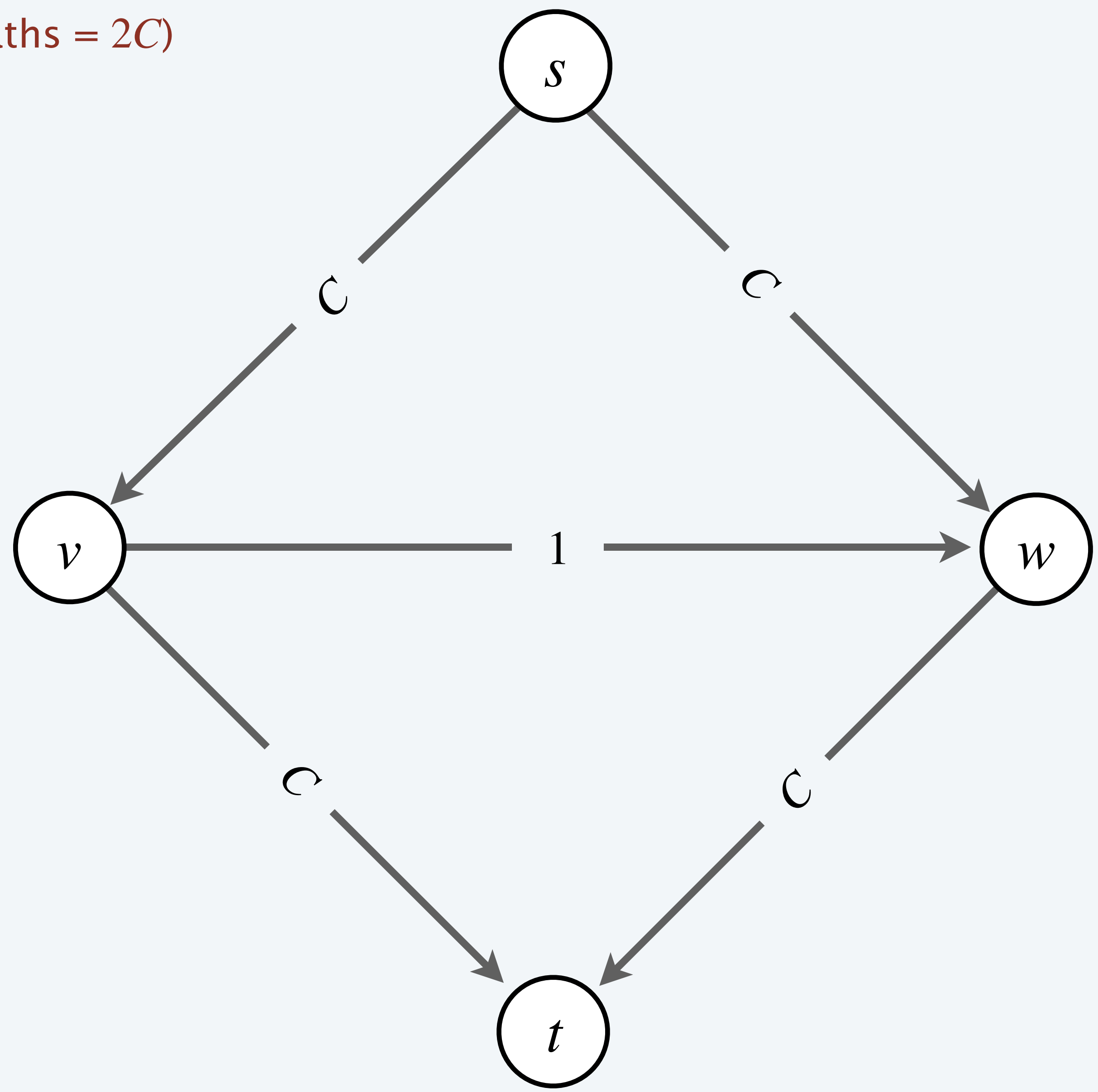
Q. Is generic Ford–Fulkerson algorithm poly-time in input size?

m, n, and $\log C$

A. No. If max capacity is C, then algorithm can take $\geq C$ 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 = $2C$)



The Ford–Fulkerson algorithm is guaranteed to terminate if the edge capacities are …

A. Rational numbers.

B. Real numbers.

C. Both A and B.

D. Neither A nor B.

Choosing good augmenting paths

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

Pathology. When edge capacities can be irrational, no guarantee that Ford–Fulkerson terminates (or converges to a maximum flow)!

Goal. Choose augmenting paths so that:

- Can find augmenting paths efficiently.
- Few iterations.

Choosing good augmenting paths

Choose augmenting paths with:

- Max bottleneck capacity ("fattest"). ← how to find?
- Sufficiently large bottleneck capacity. ← next
- Fewest edges. ← ahead

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.

Dinitz 1970 (Soviet Union)

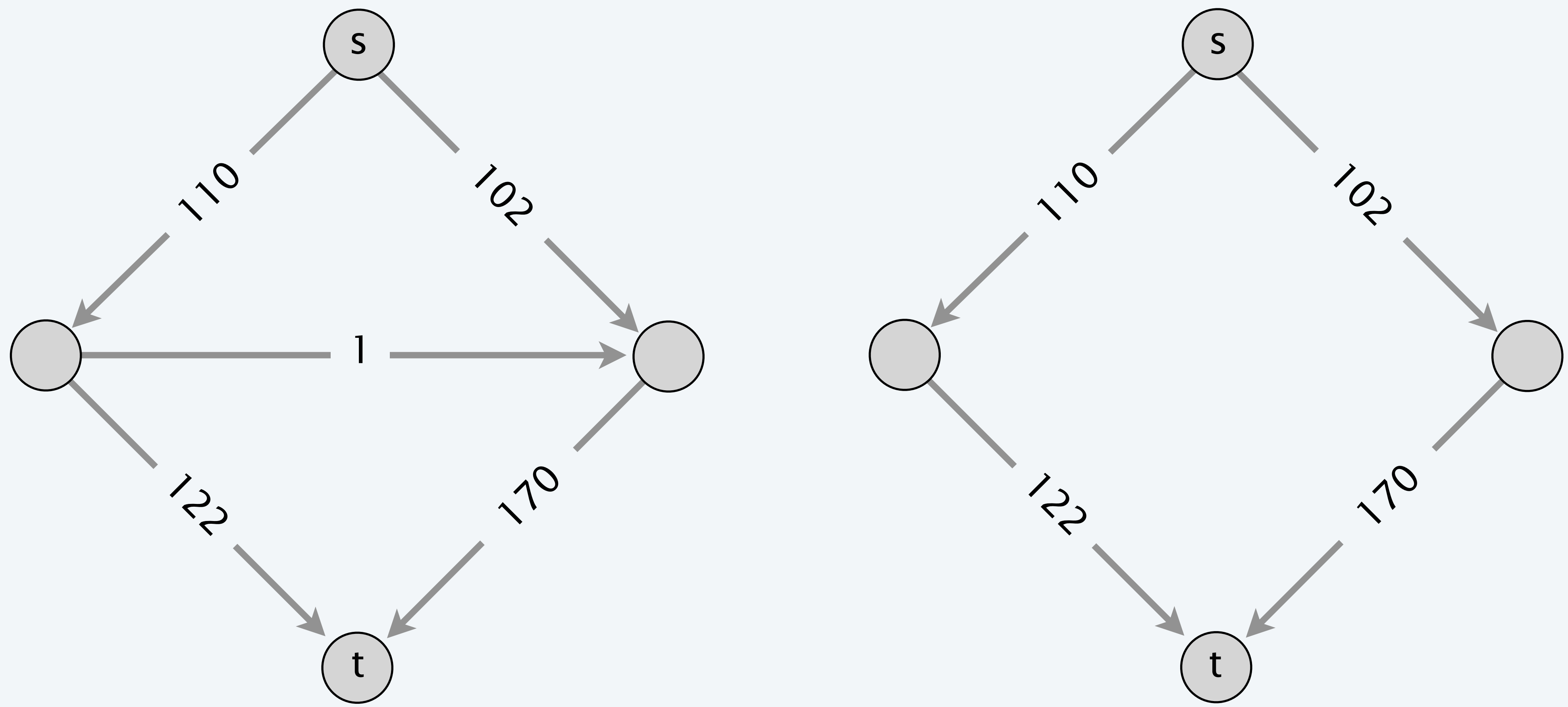
invented in response to a class exercises by Adel'son-Vel'skiĭ

Capacity-scaling algorithm

Overview. Choosing augmenting paths with "large" bottleneck capacity.

though not necessarily largest

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



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

Capacity-scaling algorithm

CAPACITY-SCALING(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$\Delta \leftarrow$ largest power of $2 \le C$.

WHILE $(\Delta \ge 1)$

  $G_f(\Delta) \leftarrow \Delta$-residual network of G with respect to flow f.

  WHILE (there exists an $s \to t$ path P in $G_f(\Delta)$)

    $f \leftarrow$ AUGMENT(f, c, P).

    Update $G_f(\Delta)$.

  $\Delta \leftarrow \Delta / 2$.

Δ-scaling phase

RETURN f.

Capacity-scaling algorithm: proof of correctness

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

Invariant. The scaling parameter Δ is a power of 2.
Pf. Initially a power of 2; each phase divides Δ by exactly 2. ▪

Integrality invariant. Throughout the algorithm, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.
Pf. Same as for generic Ford–Fulkerson. ▪

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.
- Result follows augmenting path theorem ▪

Capacity-scaling algorithm: analysis of running time

Lemma 1. There are $1 + \lfloor \log_2 C \rfloor$ scaling phases.

Pf. Initially $C / 2 < \Delta \leq C$; 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 max-flow value $\leq$ $val(f) + m\,\Delta$.

Pf. Next slide.

Lemma 3. There are $\leq 2m$ augmentations per scaling phase.

Pf.

- Let f be the flow at the beginning of a Δ-scaling phase.
- Lemma 2 $\Rightarrow$ max-flow value $\leq$ $val(f) + m\,(2\,\Delta)$.
- Each augmentation in a Δ-phase increases $val(f)$ by at least Δ. ▪

or equivalently,
at the end
of a 2Δ-scaling phase

Theorem. The capacity-scaling algorithm takes $O(m^2 \log C)$ time.

Pf.

- Lemma 1 + Lemma 3 $\Rightarrow$ $O(m \log C)$ augmentations.
- Finding an augmenting path takes $O(m)$ time. ▪

Capacity-scaling algorithm: analysis of running time

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

Pf.

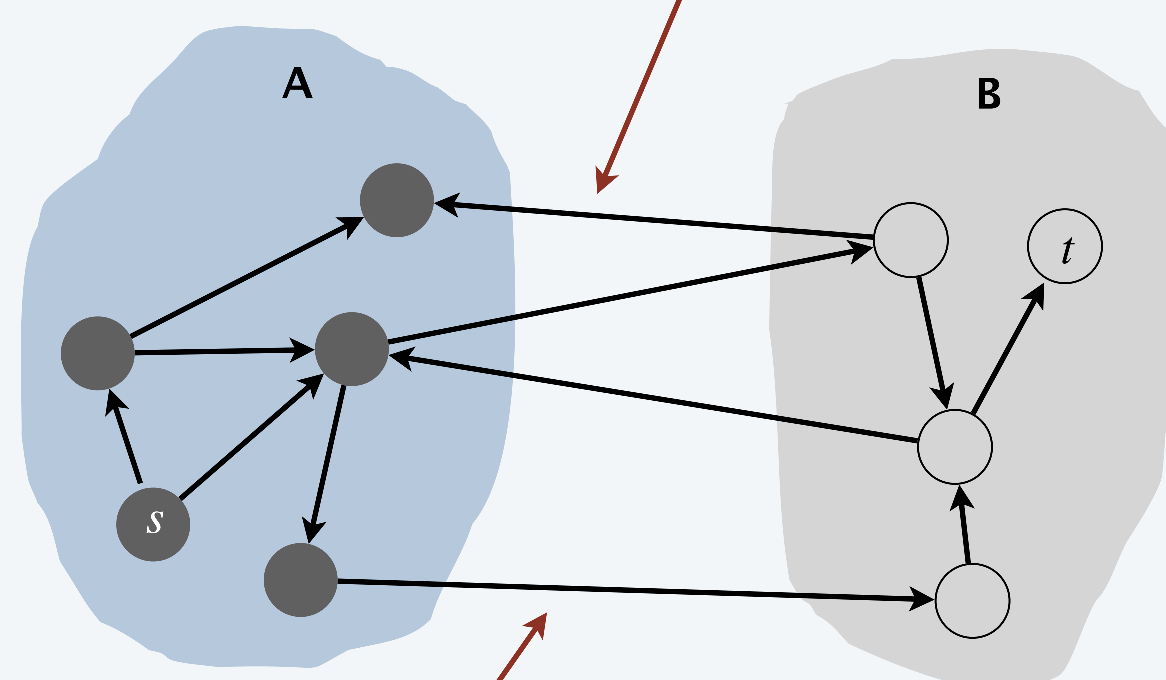
- We show there exists a cut (A, B) such that $cap(A, B) \le val(f) + m\,\Delta$.
- Choose A to be the set of nodes reachable from s in $G_f(\Delta)$.
- By definition of A: $s \in A$.
- By definition of flow f: $t \notin A$.

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

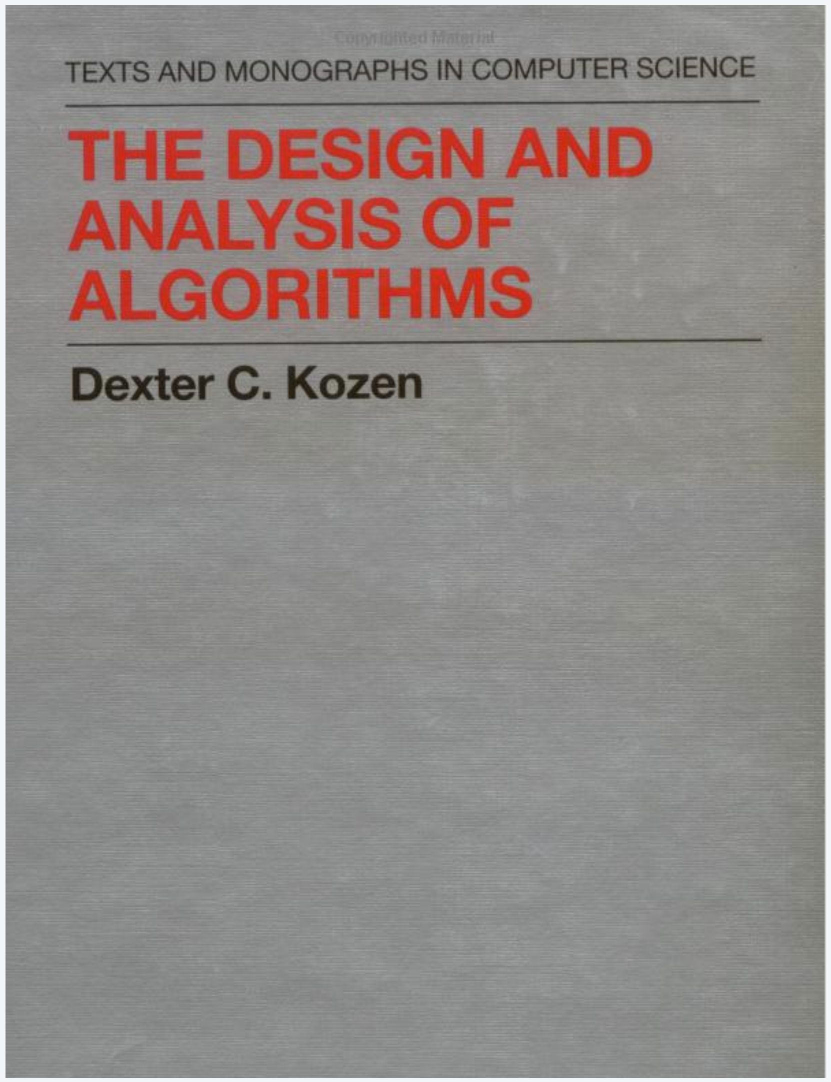
flow value lemma

edge $e = (v, w)$ with $v \in B, w \in A$ must have $f(e) < \Delta$

original flow network



edge $e = (v, w)$ with $v \in A, w \in B$ must have $f(e) > c(e) - \Delta$



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**
- Dinitz' algorithm
- simple unit-capacity networks

Shortest augmenting path

Q. How to choose next augmenting path in Ford–Fulkerson?

A. Pick one that uses the fewest edges.

can find via BFS

SHORTEST-AUGMENTING-PATH(G)

FOREACH $e \in E$: $f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

WHILE (there exists an $s \rightsquigarrow t$ path in G_f)

$P \leftarrow$ BREADTH-FIRST-SEARCH(G_f).

$f \leftarrow$ AUGMENT(f, c, P).

Update G_f.

RETURN f.

Shortest augmenting path: overview of analysis

Lemma 1. The length of a shortest augmenting path never decreases.
Pf. Ahead.

number of edges

Lemma 2. After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.
Pf. Ahead.

Theorem. The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.
Pf.

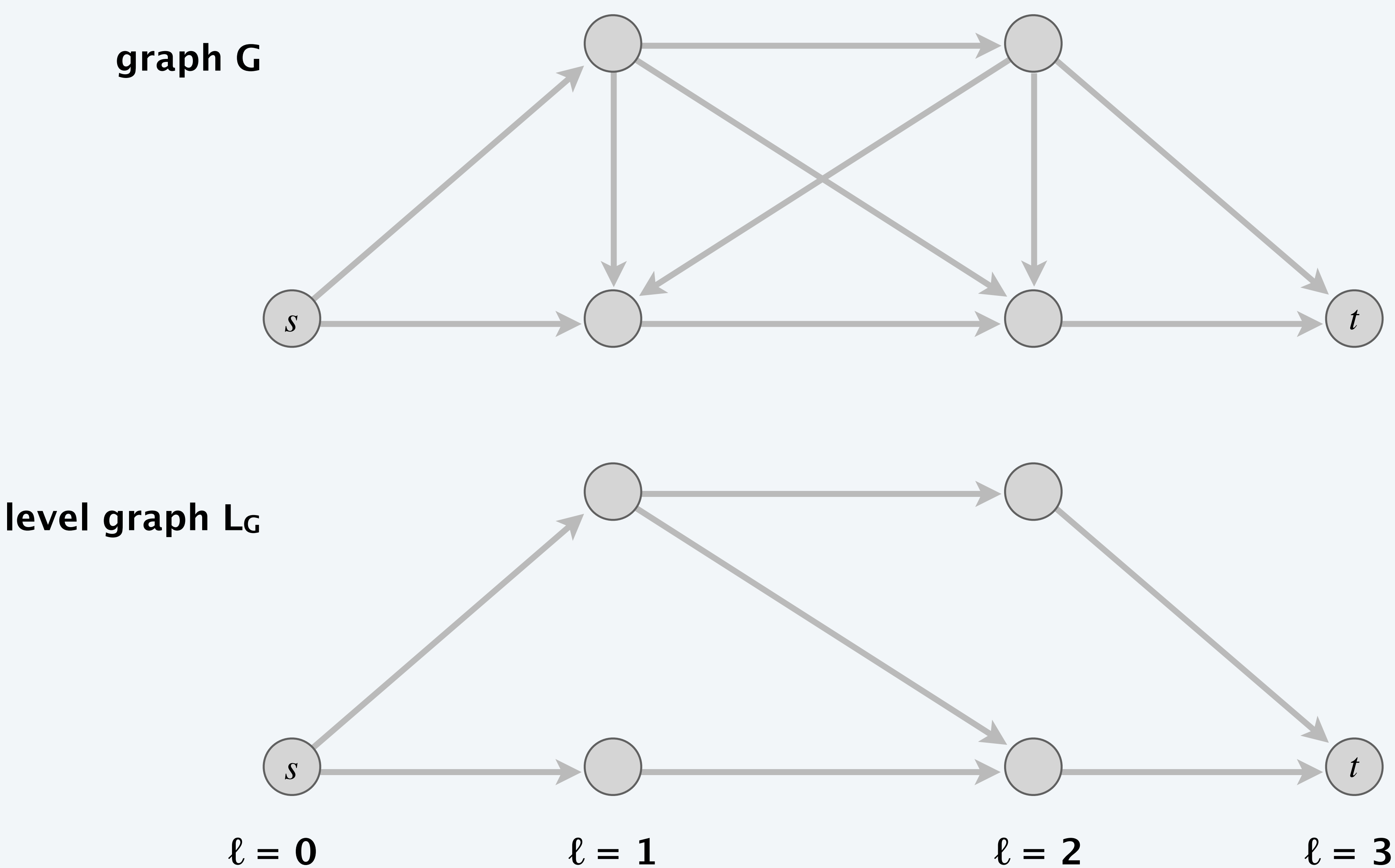
- $O(m)$ time to find a shortest augmenting path via BFS.
- There are $\leq m n$ augmentations.
  - at most m augmenting paths of length k ← Lemma 1 + Lemma 2
  - at most $n-1$ different lengths ▪

augmenting paths are simple paths

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



Network flow: quiz 5

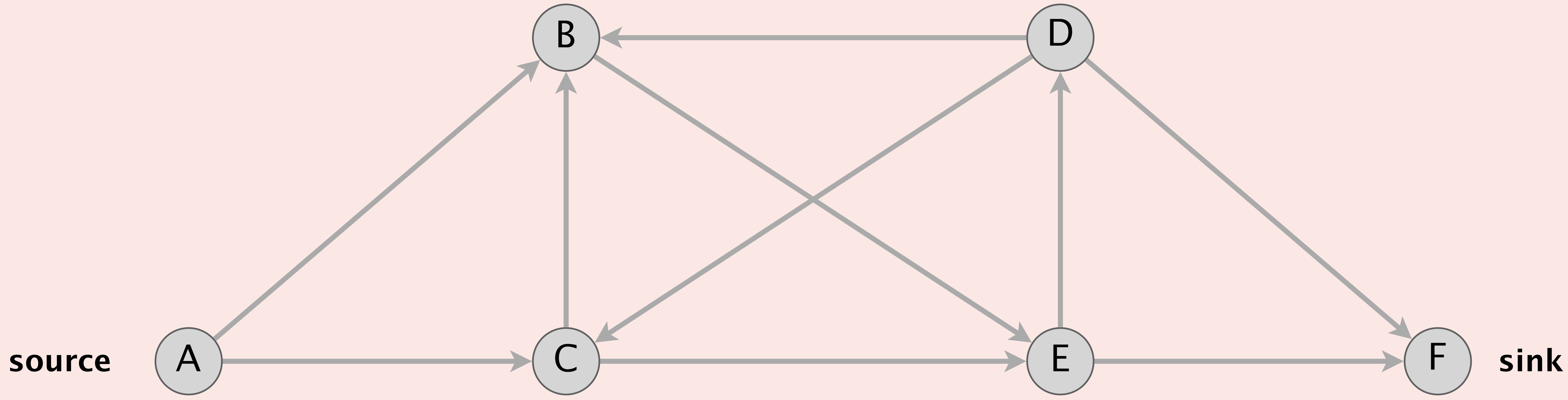
Which edges are in the level graph of the following digraph?

A. D→F.

B. E→F.

C. Both A and B.

D. Neither A nor B.

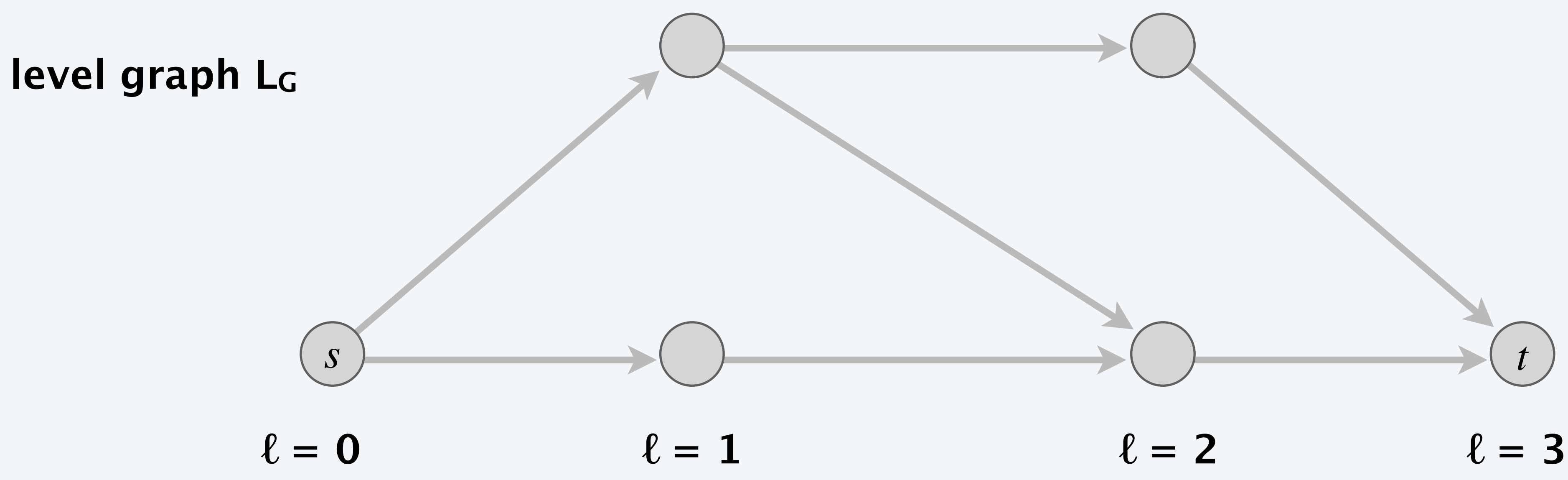


Shortest augmenting path: analysis

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

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

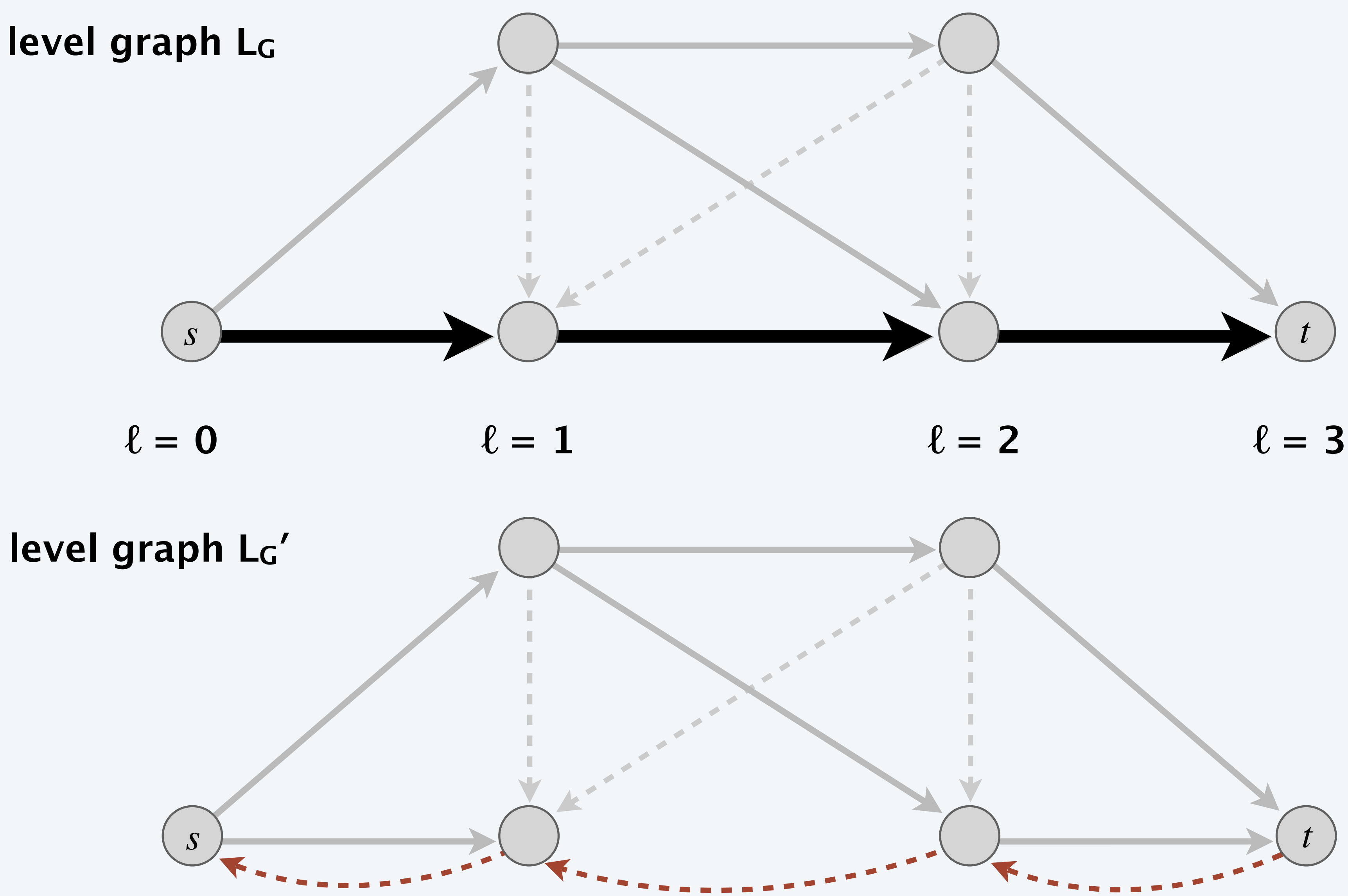
Key property. P is a shortest $s \rightsquigarrow v$ path in G iff P is an $s \rightsquigarrow v$ path in L_G.



Shortest augmenting path: analysis

Lemma 1. The length of a shortest augmenting path never decreases.

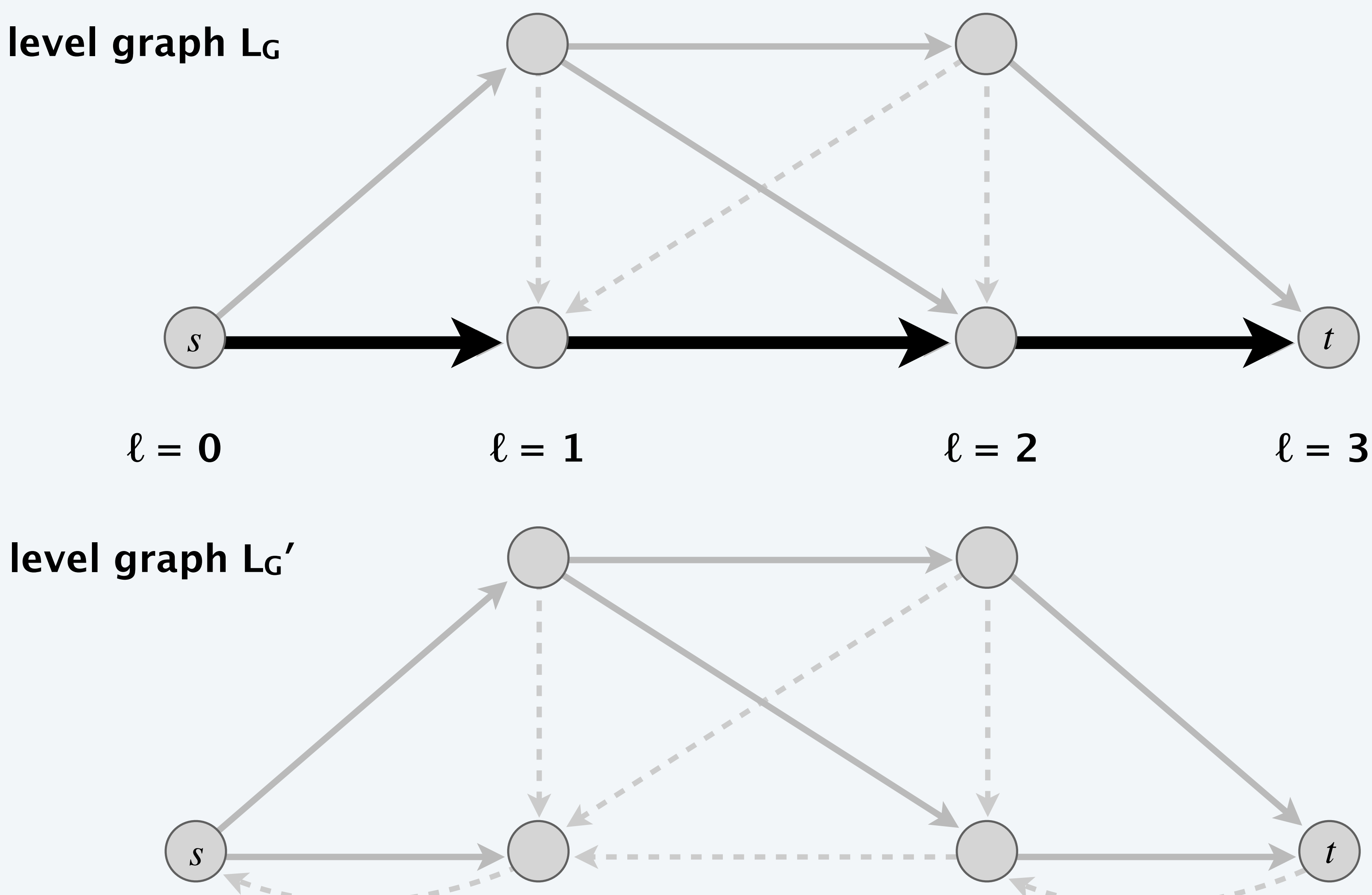
- Let f and f' be flow before and after a shortest-path augmentation.
- Let L_G and $L_{G'}$ be level graphs of G_f and $G_{f'}$.
- Only back edges added to $G_{f'}$
  (any $s \rightarrow t$ path that uses a back edge is longer than previous length) ▪



Shortest augmenting path: analysis

Lemma 2. After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

- At least one (bottleneck) edge is deleted from L_G per augmentation.
- No new edge added to L_G until shortest path length strictly increases. ▪



Shortest augmenting path: review of analysis

Lemma 1. Throughout the algorithm, the length of a shortest augmenting path never decreases.

Lemma 2. After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

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

Shortest augmenting path: improving the running time

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

- Try to decrease time per augmentation instead.
- Simple idea $\Rightarrow$ $O(mn^2)$ [Dinitz 1970] ← ahead
- Dynamic trees $\Rightarrow$ $O(m\,n \log n)$ [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:

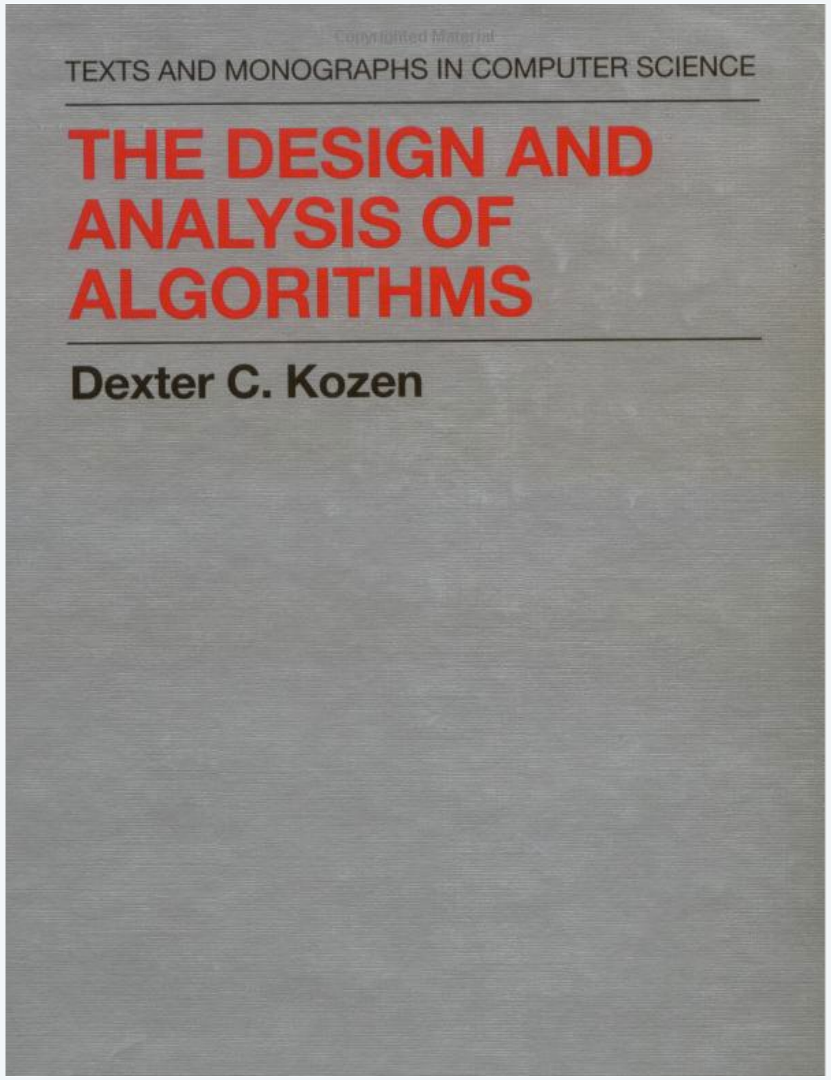
(1) Computing nearest common ancestors.

(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.



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*
- *Dinitz′ algorithm*
- *simple unit-capacity networks*

Dinitz' algorithm

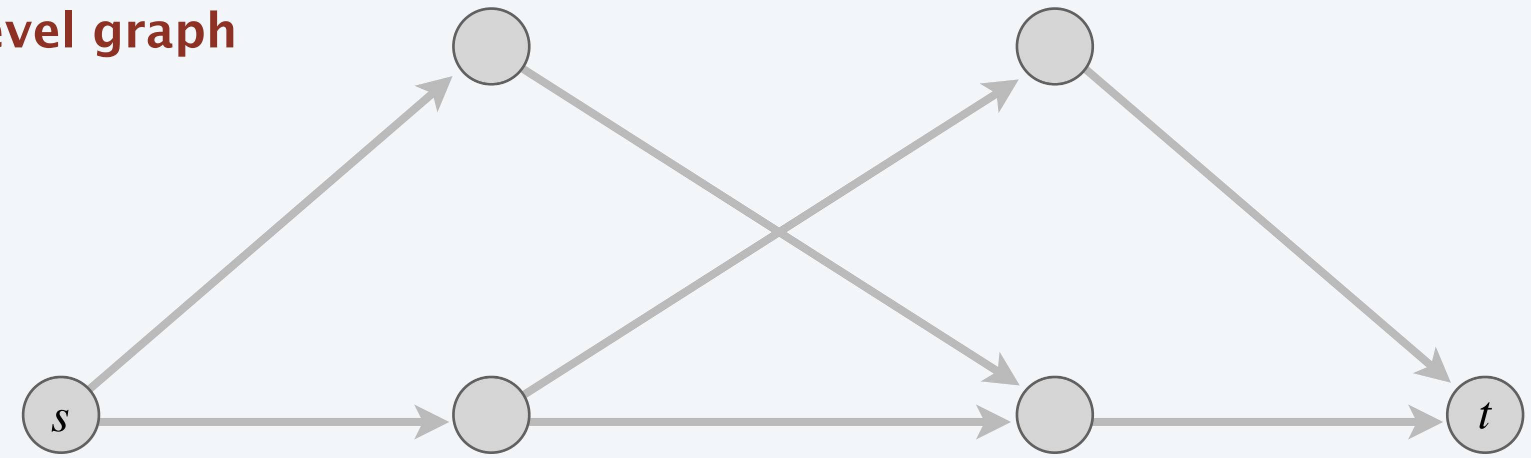
Two types of augmentations.

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

Phase of normal augmentations. ← within a phase, length of shortest augmenting path does not change

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

construct level graph



level graph L_G

Dinitz' algorithm

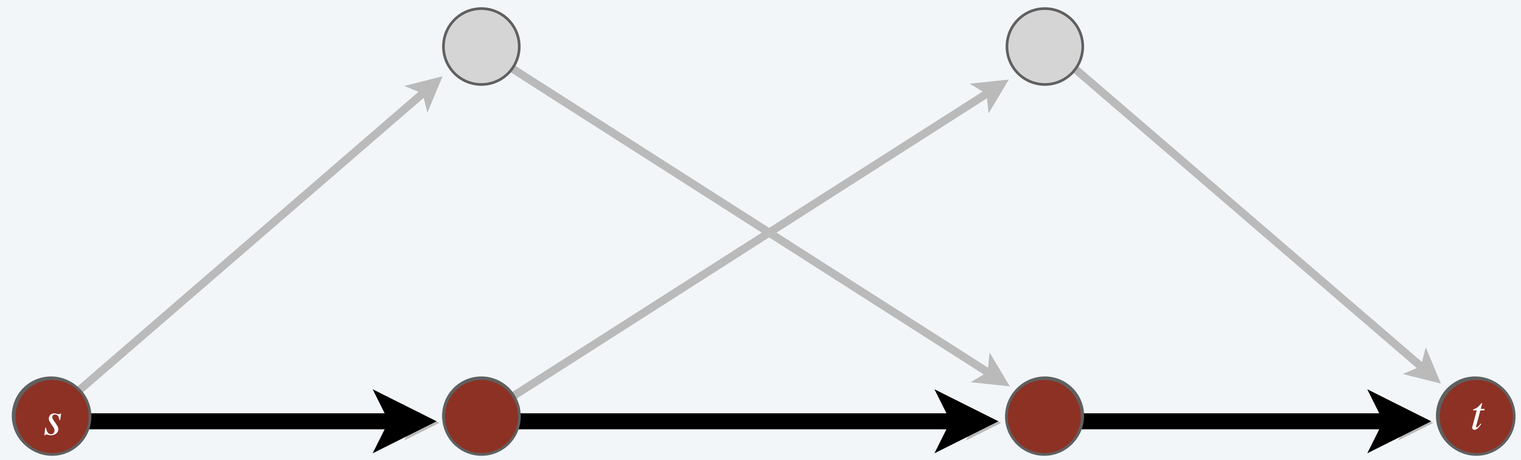
Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

advance



level graph L_G

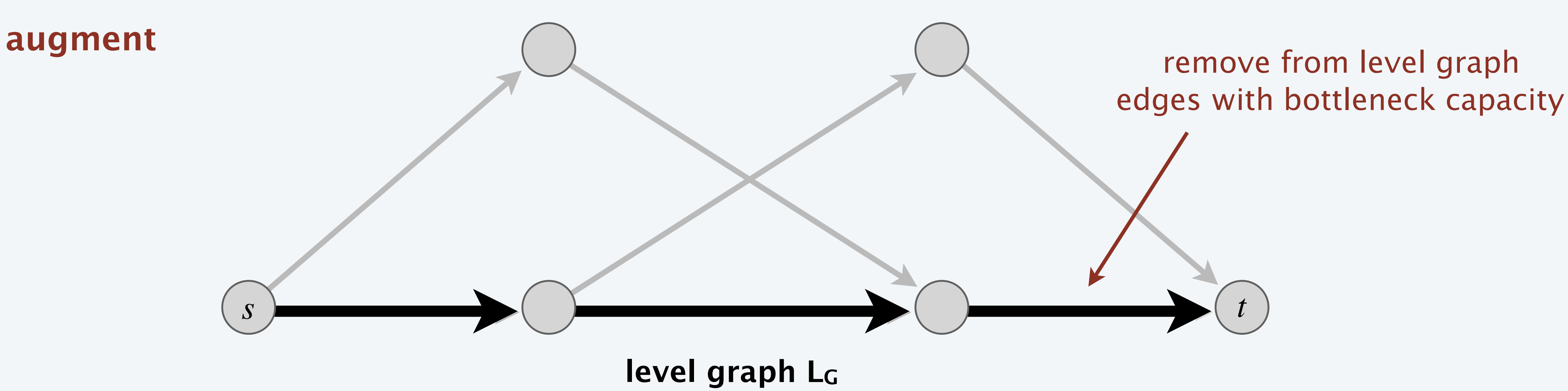
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



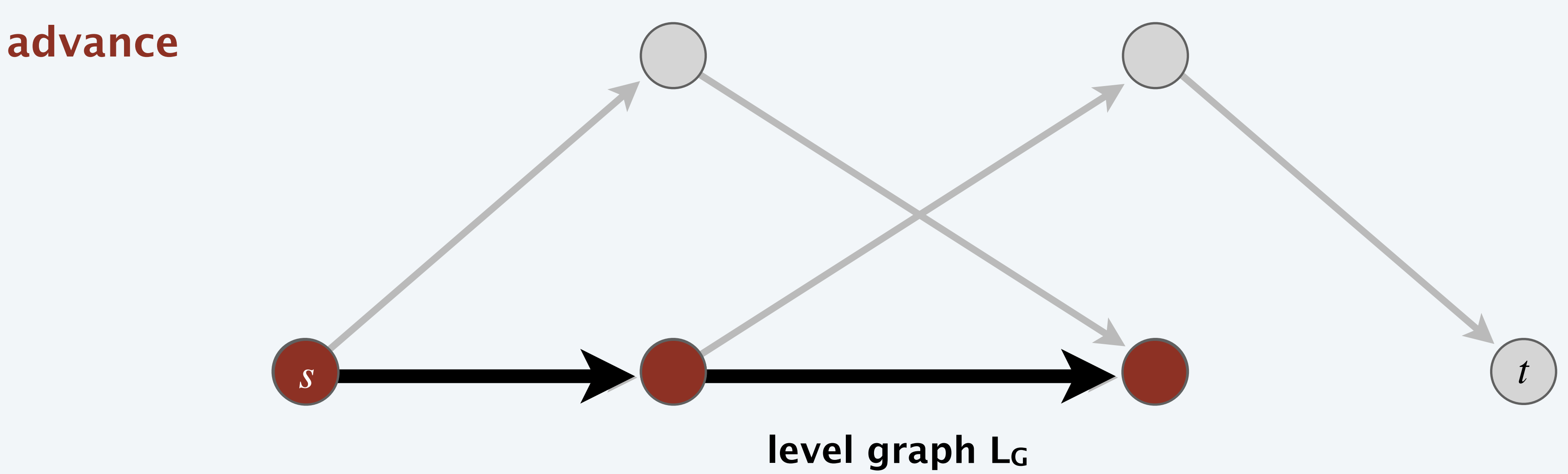
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



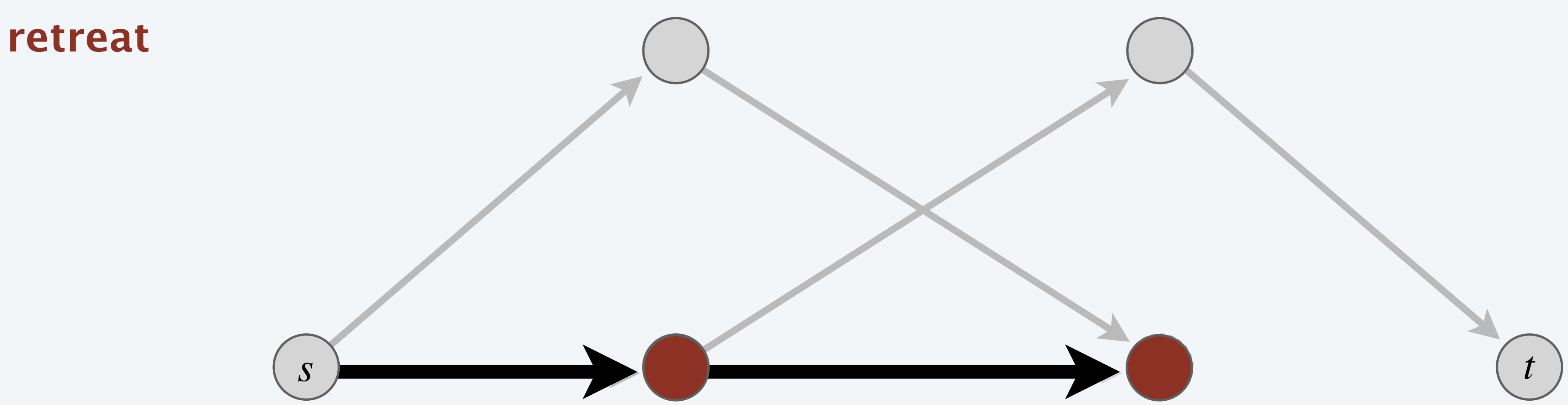
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



level graph L_G

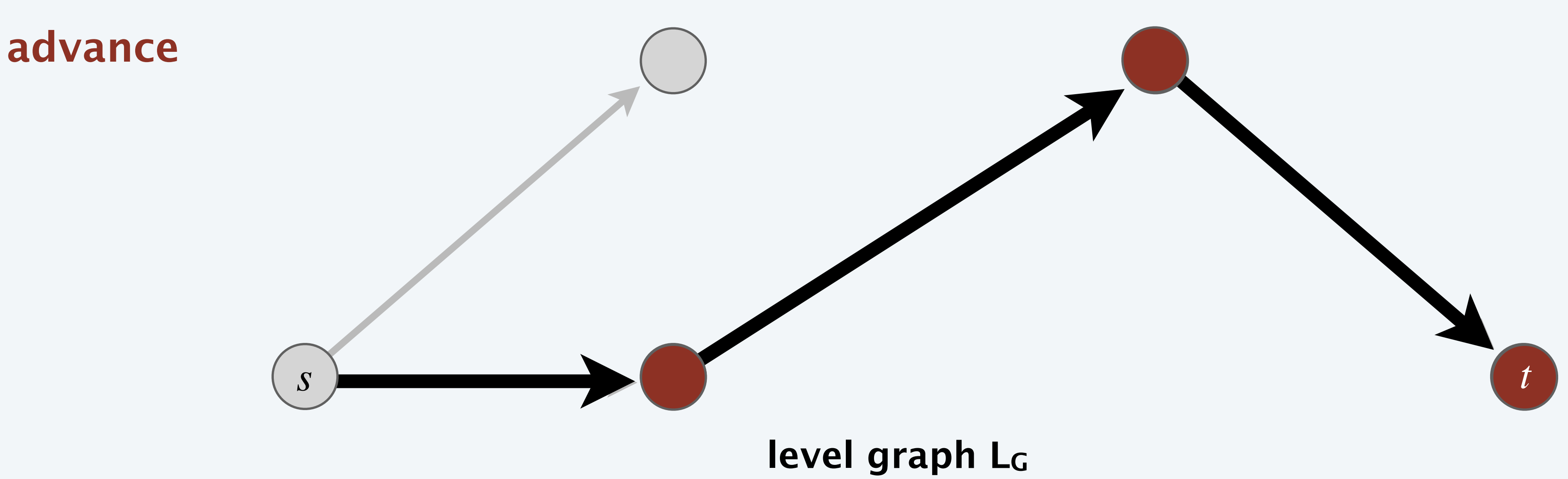
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



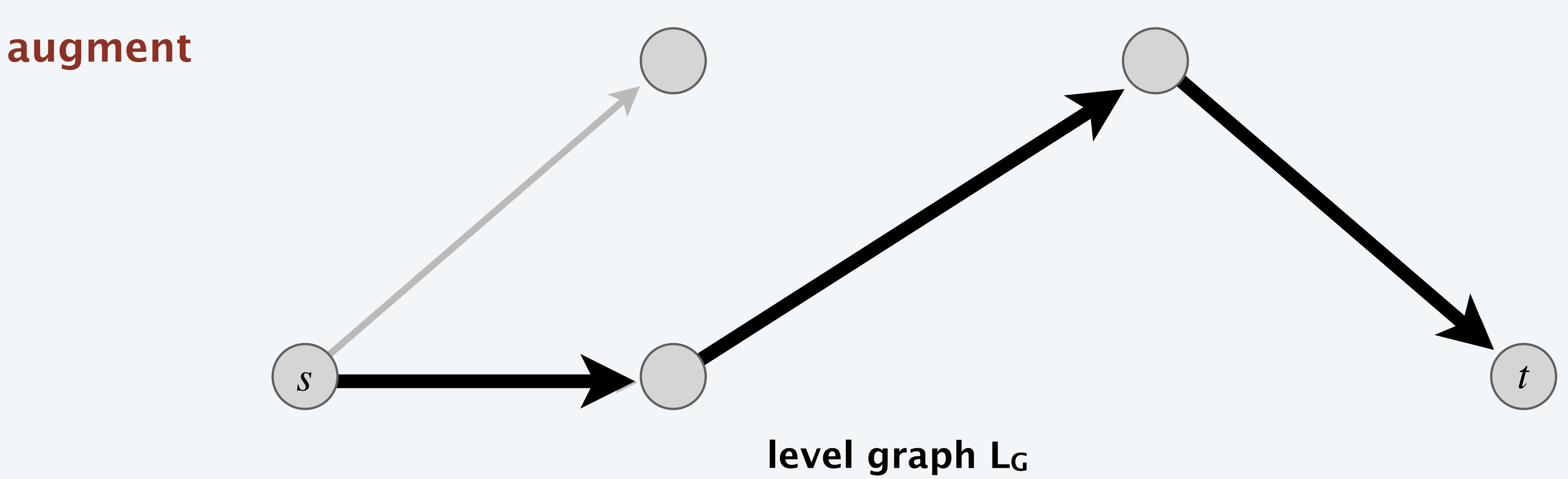
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



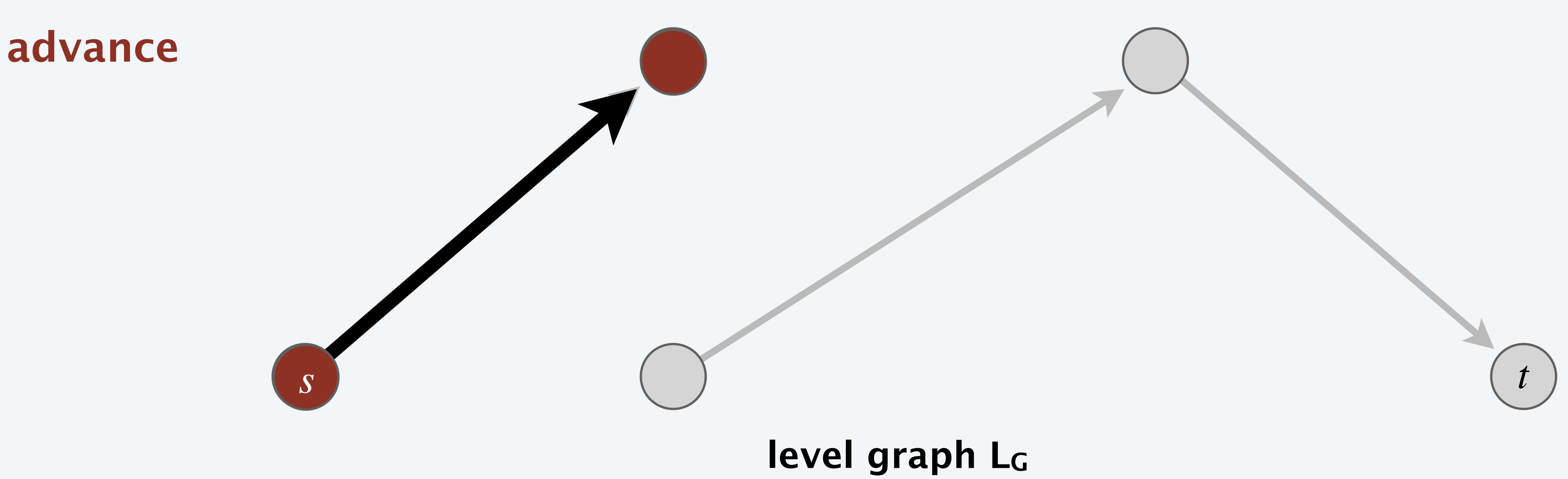
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



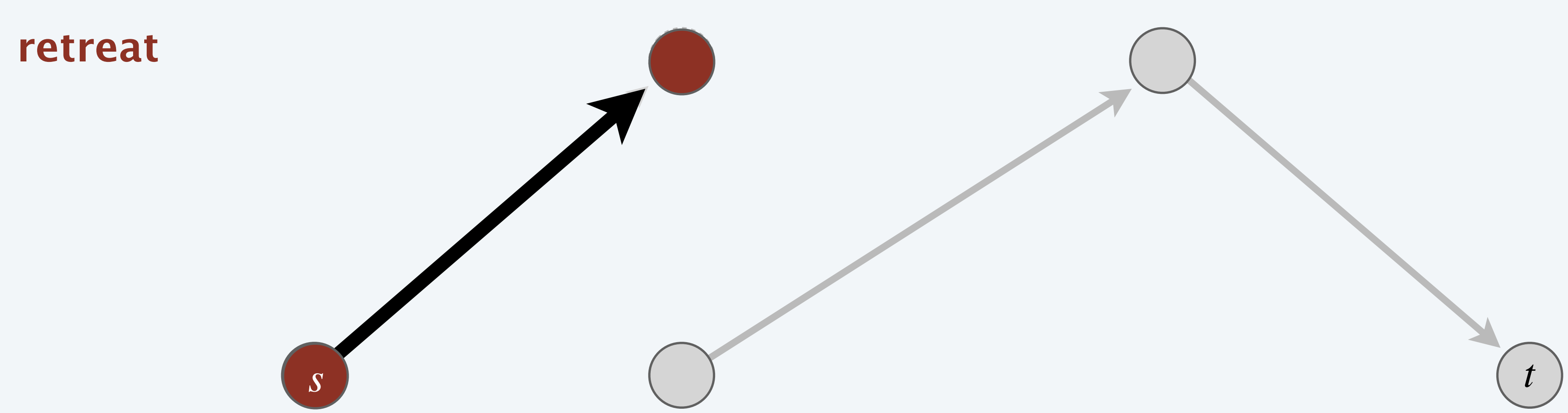
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



level graph L_G

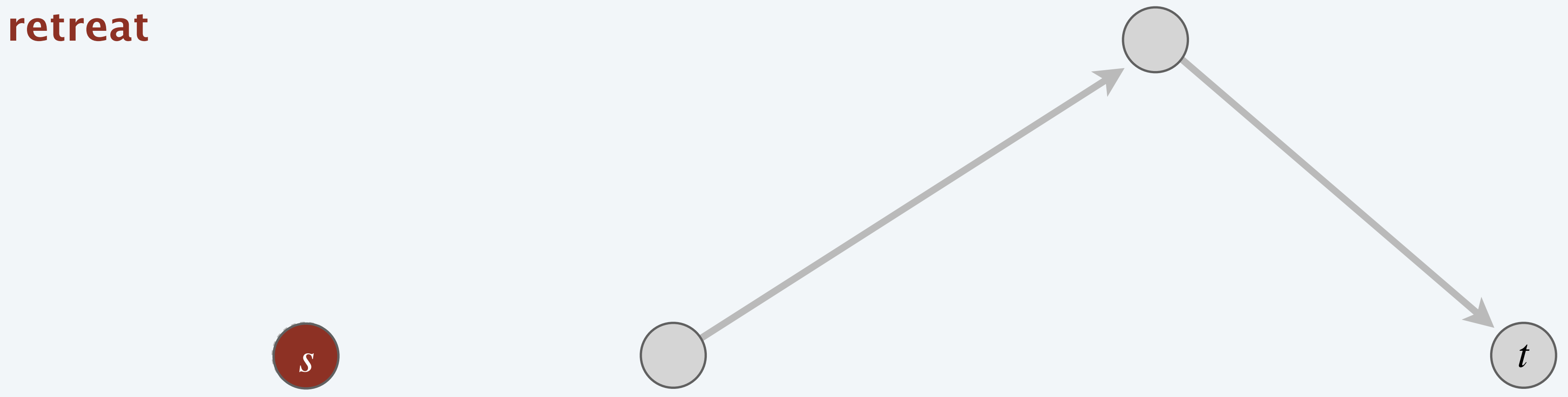
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



level graph L_G

Dinitz' algorithm

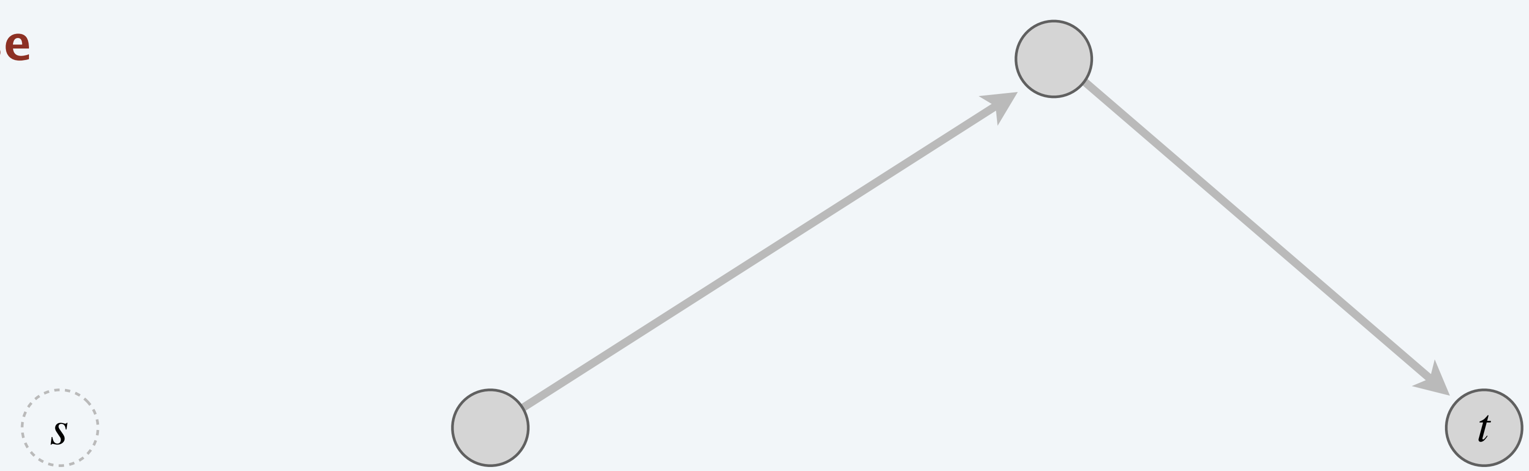
Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

end of phase



level graph L_G

Dinitz' algorithm (as refined by Even and Itai)

INITIALIZE(G, f)

$L_G \leftarrow$ 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).

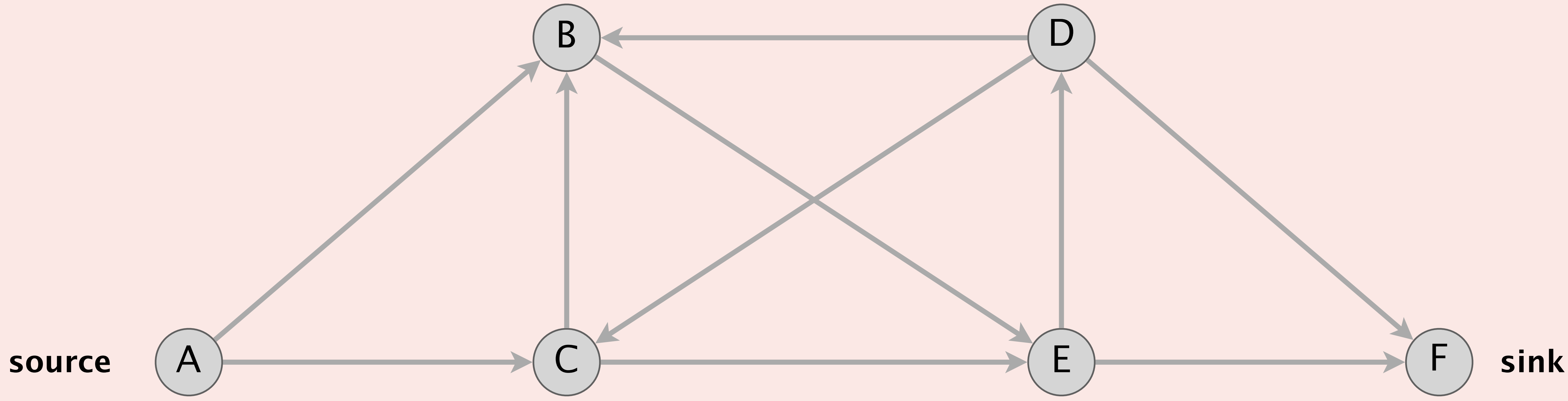
ELSE

GOTO RETREAT(v).

Network flow: quiz 6

How to compute the level graph L_G efficiently?

- **A.** Depth-first search.
- **B.** Breadth-first search.
- **C.** Both A and B.
- **D.** Neither A nor B.



Dinitz' algorithm: analysis

Lemma. A phase can be implemented to run in $O(mn)$ 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 mn advances per phase. ← $O(mn)$ per phase
  (because at most n advances before retreat or augmentation) ▪

Theorem. [Dinitz 1970] Dinitz' algorithm runs in $O(mn^2)$ time.

Pf.

- By Lemma, $O(mn)$ time per phase.
- At most $n-1$ phases (as in shortest-augmenting-path analysis). ▪

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

Maximum-flow algorithms: theory highlights

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

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

Caveat. 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: $O(m^{3/2})$ in practice.

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.

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

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

Maximum-flow algorithms: Matlab

MathWorks®

Documentation

CONTENTS

maxflow

R2018a

Maximum flow in graph

collapse all in page

Syntax

```
mf = maxflow(G,s,t)
mf = maxflow(G,s,t,algorithm)
[mf,GF] = maxflow(___)
[mf,GF,cs,ct] = maxflow(___)
```

Description

`mf = maxflow(G,s,t)` returns the maximum flow between nodes `s` and `t`. If graph `G` is unweighted (that is, `G.Edges` does not contain the variable `Weight`), then `maxflow` treats all graph edges as having a weight equal to 1. example

`mf = maxflow(G,s,t,algorithm)` specifies the maximum flow algorithm to use. This syntax is only available if `G` is a directed graph. example

Maximum-flow algorithms: Google

Google Optimization Tools

SIGN IN

Products › Optimization › Reference

Contents

Classes

C++ Reference: max_flow

This documentation is automatically generated.

An implementation of a push-relabel algorithm for the max flow problem.

In the following, we consider a graph G = (V,E,s,t) where V denotes the set of nodes (vertices) in the graph, E denotes the set of arcs (edges). s and t denote distinguished nodes in G called source and target. n = |V| denotes the number of nodes in the graph, and m = |E| denotes the number of arcs in the graph.

Each arc (v,w) is associated a capacity c(v,w).

7. NETWORK FLOW I

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

Network flow: quiz 7

Which max-flow algorithm to use for bipartite matching?

A. Ford–Fulkerson: $O(m\,n\,C)$.

B. Capacity scaling: $O(m^2 \log C)$.

C. Shortest augmenting path: $O(m^2\,n)$.

D. Dinitz' algorithm: $O(m\,n^2)$.

Simple unit-capacity networks

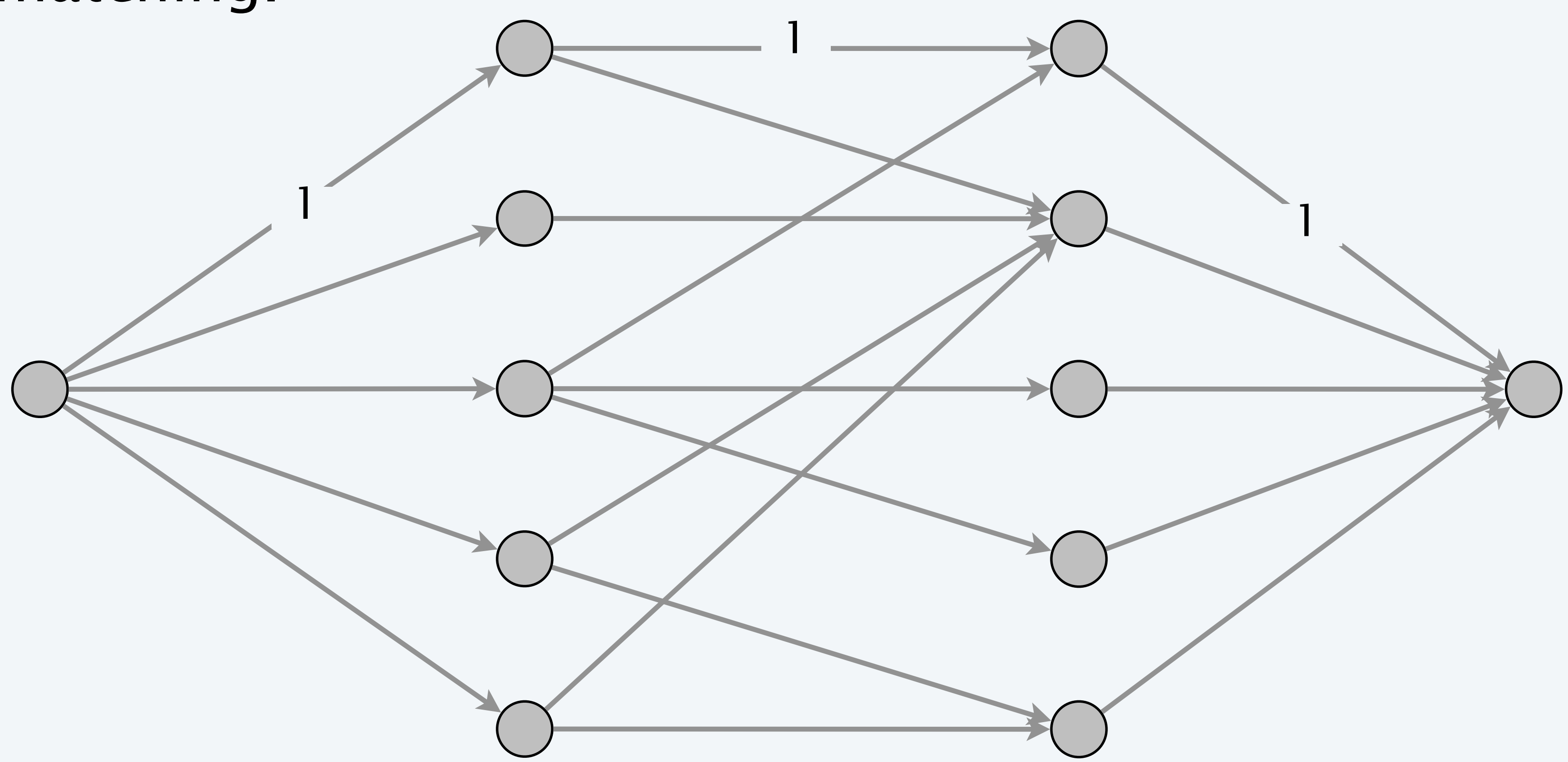
Def. A flow network is a simple unit-capacity network if:

- Every edge has capacity 1.
- Every node (other than s or t) has exactly one entering edge, or exactly one leaving edge, or both.

node capacity = 1

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

Ex. Bipartite matching.



Simple unit-capacity 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 simple unit-capacity networks, Dinitz' algorithm computes a maximum flow in $O(m\, n^{1/2})$ time.

Pf.

- Lemma 1. Each phase of normal augmentations takes $O(m)$ time.
- Lemma 2. After $n^{1/2}$ phases, $val(f) \geq val(f^*) - n^{1/2}$.
- Lemma 3. After $\leq n^{1/2}$ additional augmentations, flow is optimal. ▪

Lemma 3. After $\leq n^{1/2}$ additional augmentations, flow is optimal.

Pf. Each augmentation increases flow value by at least 1. ▪

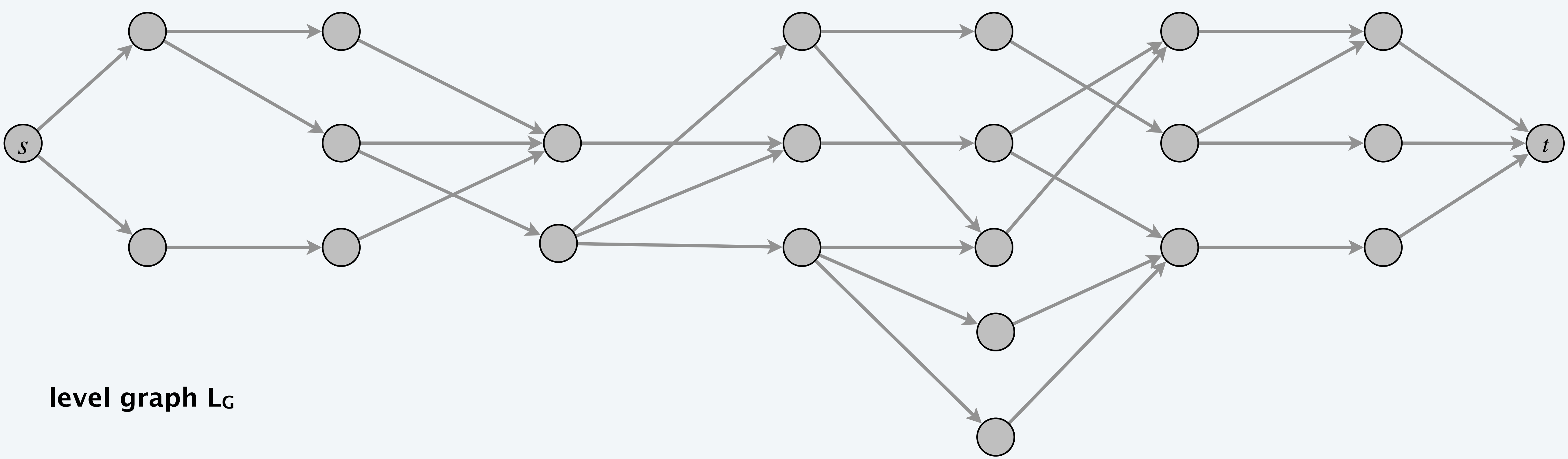
Lemma 1 and Lemma 2. Ahead.

Simple unit-capacity networks

Phase of normal augmentations. ← within a phase, length of shortest augmenting path does not change

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

construct level graph



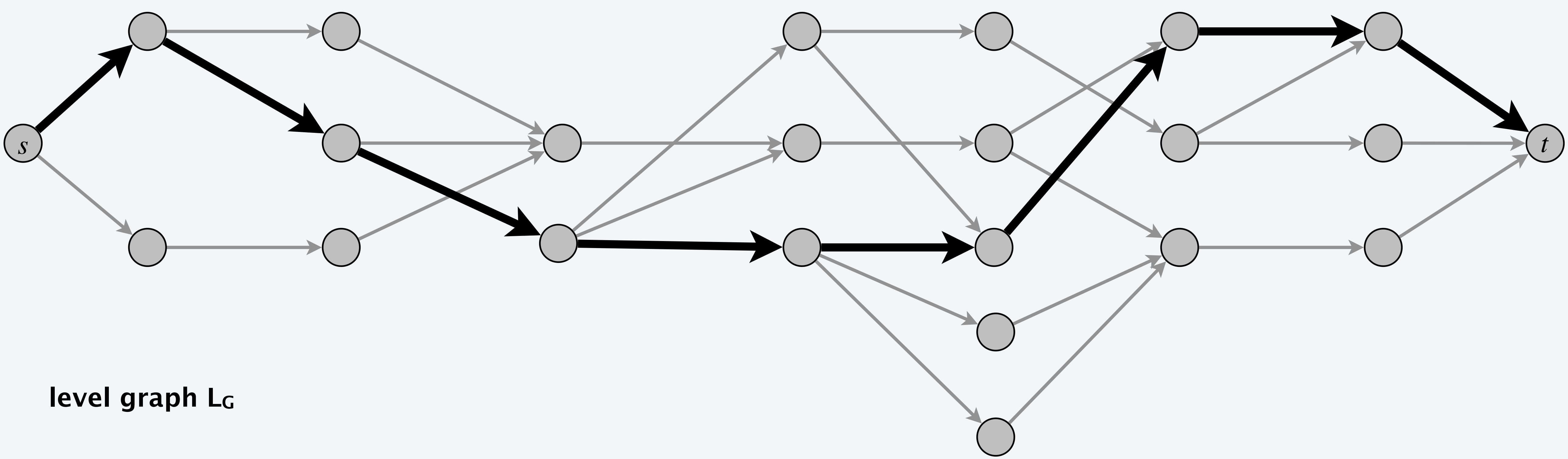
level graph L_G

Simple unit-capacity networks

Phase of normal augmentations.

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

advance

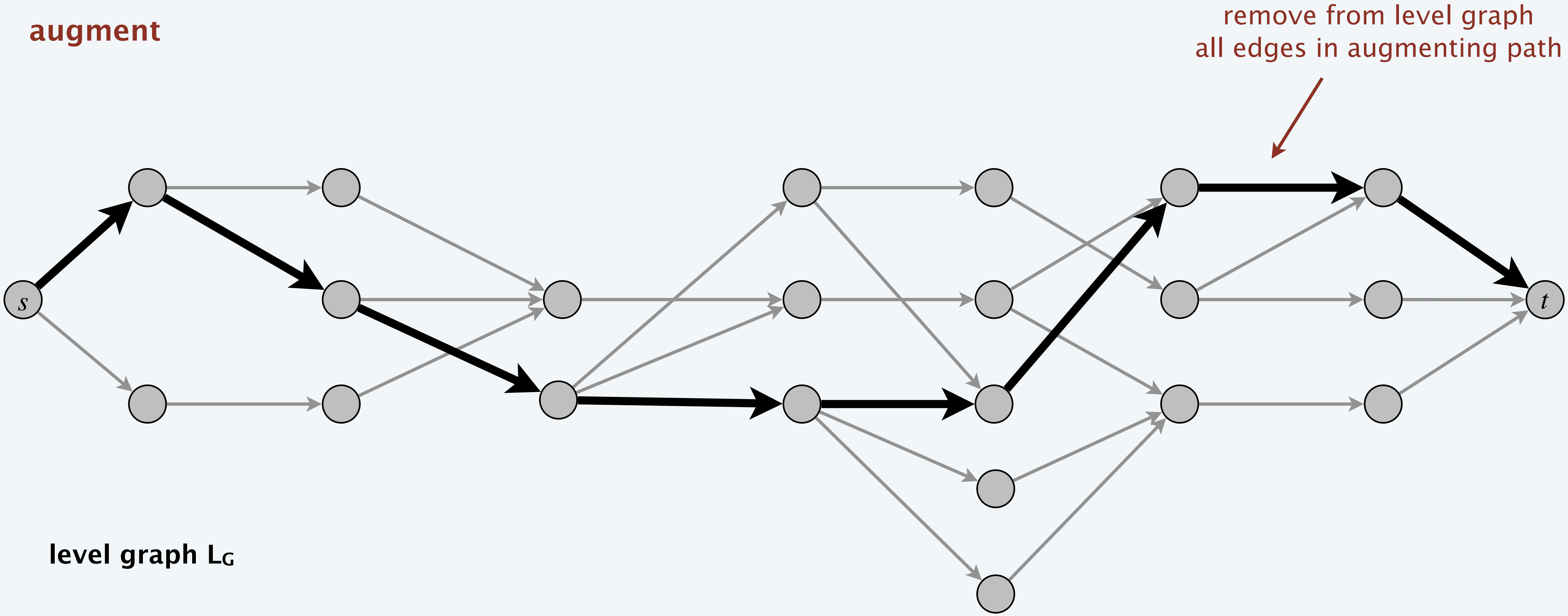


level graph L_G

Simple unit-capacity networks

Phase of normal augmentations.

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

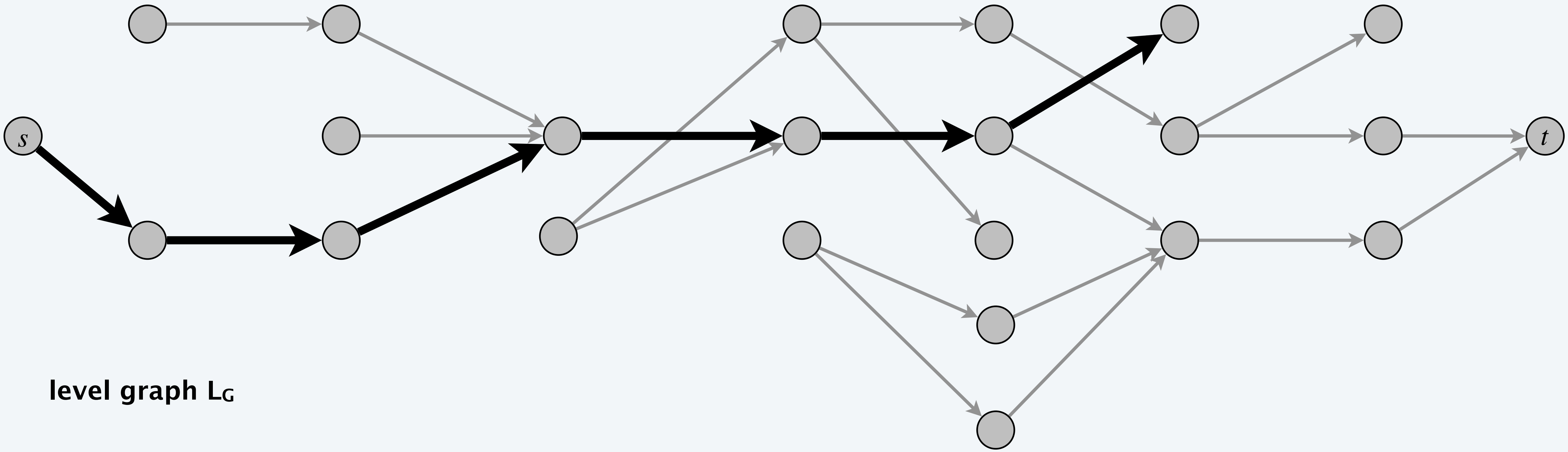


Simple unit-capacity networks

Phase of normal augmentations.

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

advance



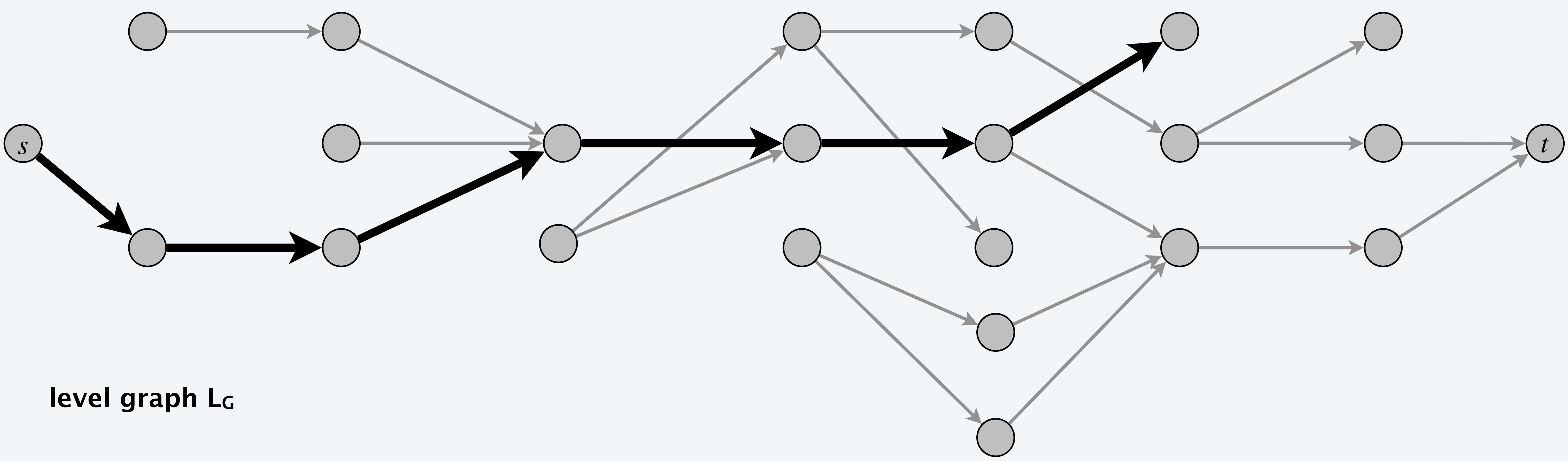
level graph L_G

Simple unit-capacity networks

Phase of normal augmentations.

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

retreat

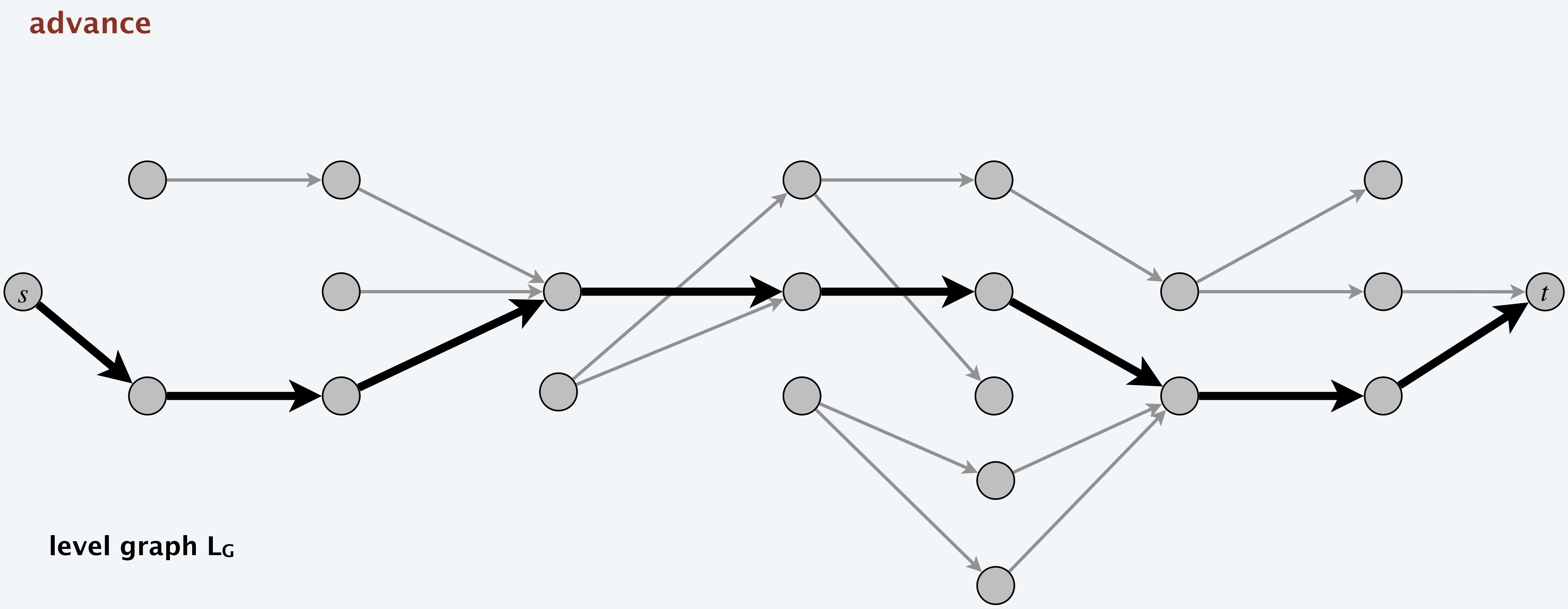


level graph L_G

Simple unit-capacity networks

Phase of normal augmentations.

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

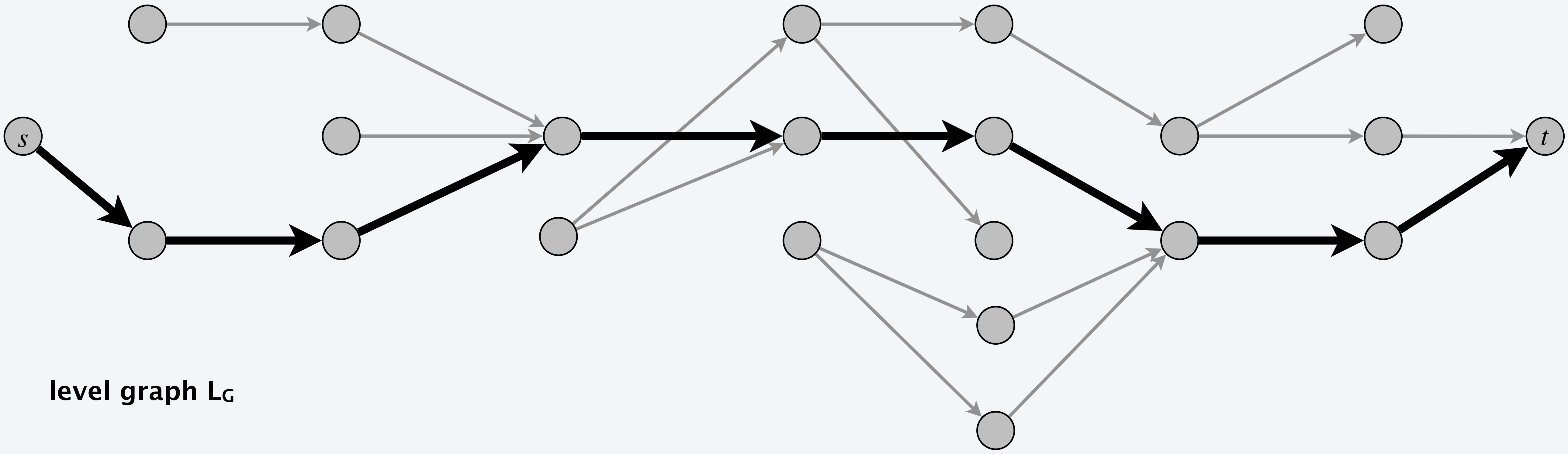


Simple unit-capacity networks

Phase of normal augmentations.

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

augment



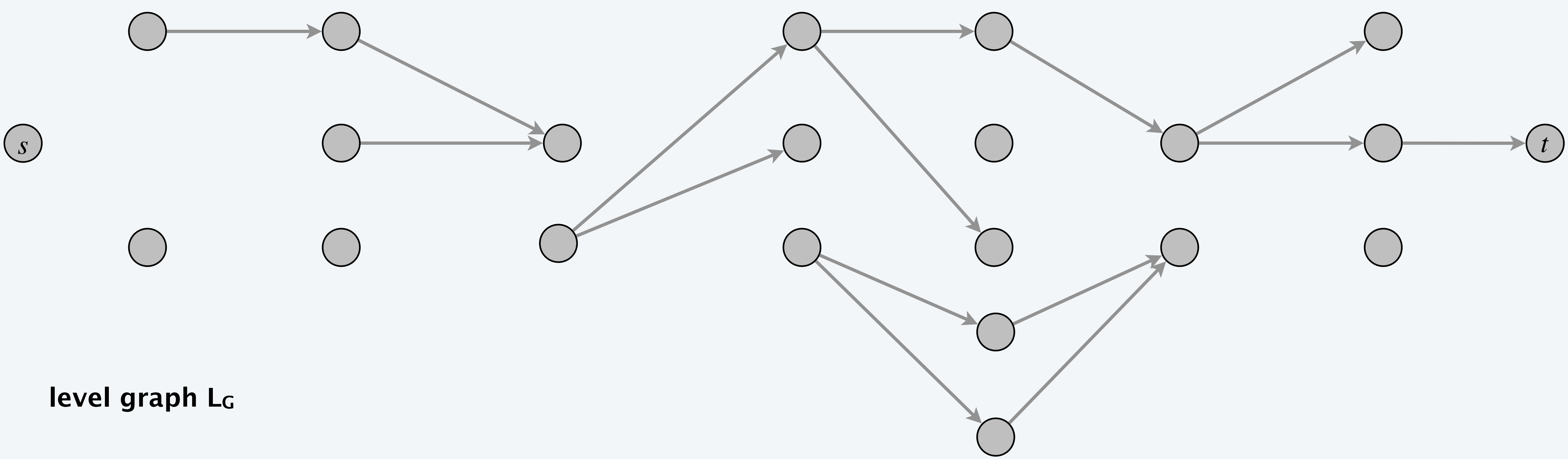
level graph L_G

Simple unit-capacity networks

Phase of normal augmentations.

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

end of phase (length of shortest augmenting path has increased)



level graph L_G

Simple unit-capacity networks: analysis

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- 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 (each edge involved in at most one advance, retreat, and augmentation).
- $O(1)$ per node (each node deleted at most once). ▪

Network flow: quiz 8

Consider running advance-retreat algorithm in a unit-capacity network (but not necessarily a simple one). What is running time?

both indegree and outdegree of a node can be larger than 1

A. $O(m)$.

B. $O(m^{3/2})$.

C. $O(m\,n)$.

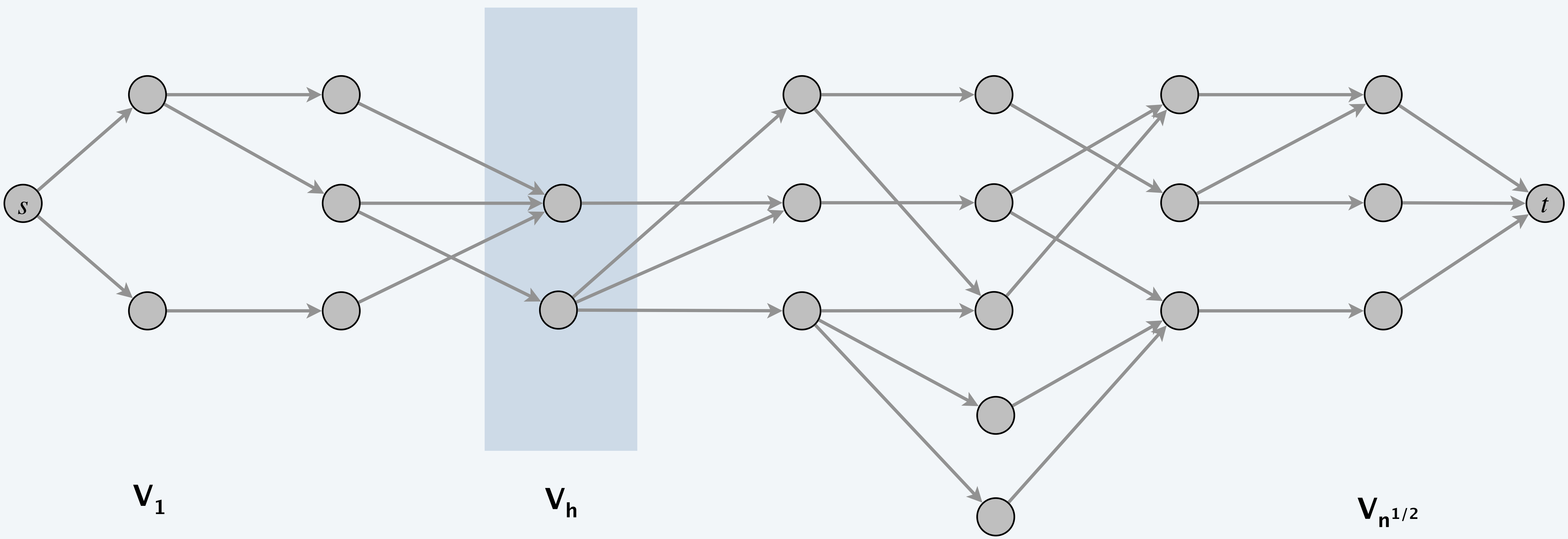
D. May not terminate.

Simple unit-capacity networks: analysis

Lemma 2. After $n^{1/2}$ phases, $val(f) \geq val(f^*) - n^{1/2}$.

- After $n^{1/2}$ phases, length of shortest augmenting path is $> n^{1/2}$.
- Thus, level graph has $\geq n^{1/2}$ levels (not including levels for s or t).
- Let $1 \leq h \leq n^{1/2}$ be a level with min number of nodes $\Rightarrow |V_h| \leq n^{1/2}$.

level graph L_G for flow f

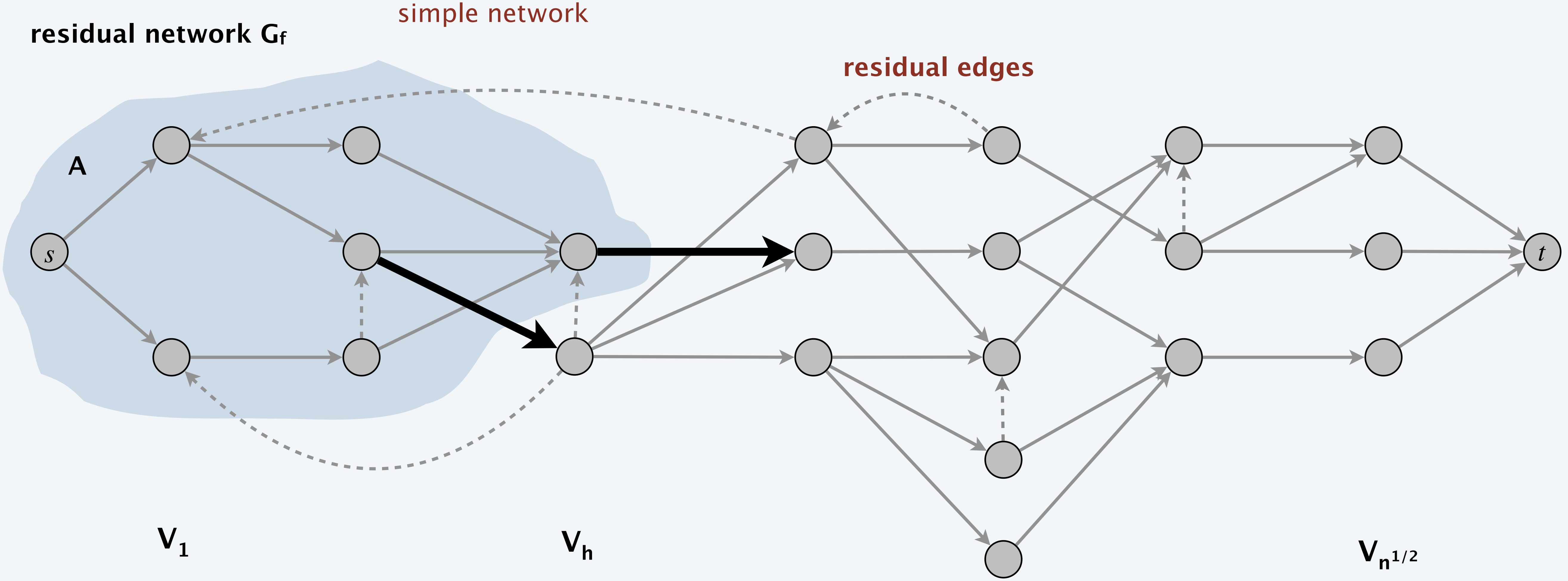


Simple unit-capacity networks: analysis

Lemma 2. After $n^{1/2}$ phases, $val(f) \geq val(f^*) - n^{1/2}$.

- After $n^{1/2}$ phases, length of shortest augmenting path is $> n^{1/2}$.
- Thus, level graph has $\geq n^{1/2}$ levels (not including levels for s or t).
- Let $1 \leq h \leq n^{1/2}$ be a level with min number of nodes $\Rightarrow |V_h| \leq n^{1/2}$.
- Let $A = \{v : \ell(v) < h\} \cup \{v : \ell(v) = h$ and v has ≤ 1 outgoing residual edge$\}$.
- $cap_f(A, B) \leq |V_h| \leq n^{1/2} \Rightarrow val(f) \geq val(f^*) - n^{1/2}$. ▪

unit-capacity simple network



Simple unit-capacity networks: review

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

Pf.

- Lemma 1. Each phase takes $O(m)$ time.
- Lemma 2. After $n^{1/2}$ phases, $val(f) \geq val(f^*) - n^{1/2}$.
- Lemma 3. After $\leq n^{1/2}$ additional augmentations, flow is optimal. ▪

Corollary. Dinitz' algorithm computes max-cardinality bipartite matching in $O(m\, n^{1/2})$ time.