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
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

Section 7.1
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 \( c(e) \geq 0 \) for each \( e \in E \).

**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 \(B\).

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

![Diagram of a graph with labeled capacities]

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 \(B\).

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

![Graph diagram with edge capacities and the statement that doesn't include edges from \(B\) to \(A\).]
**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 \(B\).

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

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

[Diagram showing a graph with nodes and edges labeled with capacities, including a highlighted cut with a total capacity calculated as 28.]
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 $e \in E$: $0 \leq f(e) \leq c(e)$ [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]
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: $\text{val}(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$

```
value = 5 + 10 + 10 = 25
```
Maximum-flow problem

Def. An \textit{st-flow} (flow) \( f \) is a function that satisfies:

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

Def. The \textbf{value} of a flow \( f \) is:

\[ \text{val}(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e) \]

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

\[
\text{value} = 10 + 5 + 13 = 28
\]
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

*Section 7.1*
Toward a max-flow algorithm

Greedy algorithm.

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

**Greedy algorithm.**
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \leadsto t$ 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 $f$**

![Flow network diagram with capacities and flows marked on edges.](image)
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.
Toward a max-flow algorithm

Greedy algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ 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 $s \rightarrow t$ 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 $f^*$ has $f^*(v, w) = 0$.
- Greedy algorithm could choose $s \rightarrow v \rightarrow w \rightarrow t$ as first path.

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

Original edge. \( e = (u, v) \in E \).
- Flow \( f(e) \).
- Capacity \( c(e) \).

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. \( G_f = (V, E_f, s, t, c_f) \).
- \( E_f = \{e : f(e) < c(e)\} \cup \{e_{\text{reverse}} : f(e) > 0\} \).
- 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 $s \rightarrow t$ 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 $f' \leftarrow \text{AUGMENT}(f, c, P)$, the resulting $f'$ is a flow and $\text{val}(f') = \text{val}(f) + \text{bottleneck}(G_f, P)$.

\begin{verbatim}
\text{AUGMENT}(f, c, P)

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

\text{FOREACH} edge $e \in P$:

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

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

\text{RETURN} $f$.
\end{verbatim}
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 augmenting path algorithm.

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

\begin{verbatim}
FORD–FULKERSON(G)

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

G_f \leftarrow \text{residual network of } G \text{ with respect to flow } f.

WHILE (there exists an } s \rightarrow t \text{ path } P \text{ in } G_f

f \leftarrow \text{AUGMENT}(f, c, P).

Update } G_f.

RETURN } f.
\end{verbatim}
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 $(A, B)$.

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

**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 $(A, B)$.

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

**net flow across cut** = \((10 + 10 + 5 + 10 + 0 + 0) - (5 + 5 + 0 + 0)\) = 25

```
edges from q to p
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.**

\[
val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)
\]

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

\[
\Rightarrow \sum_{v \in A} \left( \sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right)
\]

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

\[\blacksquare\]
Relationship between flows and cuts

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

**Pf.**

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

\[
\leq \sum_{e \text{ out of } A} f(e)
\]

\[
\leq \sum_{e \text{ out of } A} c(e)
\]

\[
= \text{cap}(A, B)
\]

---

**Diagram:**

- **Flow value lemma:**
- **Flow value = 27**
- **Capacity of cut = 30**
Certificate of optimality

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

Pf.

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

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.
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.

\[ \text{Pf. The following three conditions are equivalent for any flow } f: \]

i. There exists a cut \( (A, B) \) such that \( \text{cap}(A, B) = \text{val}(f) \).

ii. \( f \) is a max flow.

iii. There is no augmenting path with respect to \( f \). \[ \text{if Ford–Fulkerson terminates, then } f \text{ is max flow} \]

\[ [i \Rightarrow ii] \]

• This is the weak duality corollary. \[ \blacksquare \]
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 $\text{cap}(A, B) = \text{val}(f)$.

ii. $f$ is a max flow.

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

[ii $\Rightarrow$ iii ] We prove contrapositive: $\neg$ iii $\Rightarrow$ $\neg$ 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 $\Rightarrow$ i ]

- Let $f$ be a flow with no augmenting paths.
- Let $A = \text{set of nodes reachable from } s \text{ in residual network } G_f$.
- By definition of $A$: $s \in A$.
- By definition of flow $f$: $t \notin A$.

\[
val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)
\]
\[
= \sum_{e \text{ out of } A} c(e) - 0
\]
\[
= \text{cap}(A, B)
\]
Computing a minimum cut from a maximum flow

**Theorem.** Given any max flow $f$, can compute a min cut $(A, B)$ in $O(m)$ time.

**Pf.** Let $A =$ set of nodes reachable from $s$ in residual network $G_f$. □

---

Diagram:

1. The diagram illustrates a network with nodes $s$, $t$, and other nodes, connected by directed edges with capacities.
2. The minimum cut $(A, B)$ is highlighted, along with the capacity values of the edges.
3. An annotation indicates that the argument from previous slide implies that the capacity of $(A, B) =$ value of flow $f$. 

---
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. □

**Theorem.** Ford–Fulkerson terminates after at most $\text{val}(f^*) \leq nC$ 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. □

**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?

**A.** No. If max capacity is \( C \), then algorithm can take \( \geq C \) iterations.

- \( s \rightarrow v \rightarrow w \rightarrow t \)
- \( s \rightarrow w \rightarrow v \rightarrow t \)
- \( s \rightarrow v \rightarrow w \rightarrow t \)
- \( s \rightarrow w \rightarrow v \rightarrow t \)
- \( \ldots \)
- \( s \rightarrow v \rightarrow w \rightarrow t \)
- \( s \rightarrow w \rightarrow v \rightarrow t \)

Each augmenting path sends only 1 unit of flow (\# augmenting paths = \( 2C \))
Network flow: quiz 4

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"). \textcolor{red}{how to find?}
- Sufficiently large bottleneck capacity. \textcolor{red}{next}
- Fewest edges. \textcolor{red}{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)

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.

- Maintain scaling parameter $\Delta$.
- 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 $\geq \Delta$.

\[ G_f \]

\[ G_f(\Delta), \ \Delta = 100 \]
Capacity-scaling algorithm

**CAPACITY-SCALING**(*G*)

**FOREACH** edge *e* ∈ *E*: *f*(*e*) ← 0.

*Δ* ← largest power of 2 ≤ *C*.

**WHILE** (*Δ* ≥ 1)

*G*<sub>f</sub>(*Δ*) ← *Δ*-residual network of *G* with respect to flow *f*.

**WHILE** (there exists an *s*→*t* path *P* in *G*<sub>f</sub>(*Δ*))

 *f* ← **AUGMENT**(*f*, *c*, *P*).

Update *G*<sub>f</sub>(*Δ*).

*Δ* ← *Δ*/ 2.

**RETURN** *f*.
Assumption. All edge capacities are integers between 1 and $C$.

Invariant. The scaling parameter $\Delta$ is a power of 2.

Pf. Initially a power of 2; each phase divides $\Delta$ 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 $1 + \lceil \log_2 C \rceil$ scaling phases.

*Proof.* Initially $C/2 < \Delta \leq C$; $\Delta$ decreases by a factor of 2 in each iteration. □

**Lemma 2.** Let $f$ be the flow at the end of a $\Delta$-scaling phase. Then, the max-flow value $\leq \text{val}(f) + m \Delta$.

*Proof.* Next slide.

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

*Proof.*
- Let $f$ be the flow at the beginning of a $\Delta$-scaling phase.
- Lemma 2 $\Rightarrow$ max-flow value $\leq \text{val}(f) + m (2 \Delta)$.
- Each augmentation in a $\Delta$-phase increases $\text{val}(f)$ by at least $\Delta$. □

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

*Proof.*
- Lemma 1 + Lemma 3 $\Rightarrow$ $O(m \log C)$ augmentations.
- Finding an augmenting path takes $O(m)$ time. □
Lemma 2. Let $f$ be the flow at the end of a $\Delta$-scaling phase. Then, the max-flow value $\leq val(f) + m \Delta$.

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

\[
val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
\geq \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ in to } A} \Delta \\
\geq \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\
\geq cap(A, B) - m\Delta \quad \blacksquare
\]
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.

\[
\text{SHORTEST-Augmenting-Path}(G)
\]

\[
\text{FOREACH} \ e \in E: f(e) \leftarrow 0.
\]

\[
G_f \leftarrow \text{residual network of } G \text{ with respect to flow } f.
\]

\[
\text{WHILE} \ (\text{there exists an } s \rightarrow t \text{ path in } G_f)
\]

\[
P \leftarrow \text{Breadth-First-Search}(G_f).
\]

\[
f \leftarrow \text{Augment}(f, c, P).
\]

Update \( G_f \).

\[
\text{RETURN} \ f.
\]
Shortest augmenting path: overview of analysis

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

**Pf.** Ahead.

**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 $\leq mn$ augmentations.
  - at most $m$ augmenting paths of length $k$ \(\rightarrow \text{Lemma 1 + Lemma 2}\)
  - 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 $\ell(w) = \ell(v) + 1$. 

![Graph $G$](image1)

$G$ is defined by $G = (V, E)$ and contains edges $(v, w)$ with $\ell(w) = \ell(v) + 1$.

![Level graph $L_G$](image2)

$L_G$ is the subgraph of $G$ defined by $\ell(w) = \ell(v) + 1$.

Levels $\ell$:
- $\ell = 0$
- $\ell = 1$
- $\ell = 2$
- $\ell = 3$
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 $s \leadsto v$ path.
- $L_G = (V, E_G)$ is the subgraph of $G$ that contains only those edges $(v, w) \in E$ with $\ell(w) = \ell(v) + 1$.

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

level graph $L_G$
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 \( s \sim t \) path that uses a back edge is longer than previous length)
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. $\blacksquare$
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. \( \Theta(mn) \) augmentations necessary for some flow networks.

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

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.
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.
- 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.

![Diagram of level graph $L_G$ with nodes $s$, $t$ and an edge between them.](advance)
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.

![Diagram of Dinitz’ algorithm](image)
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.

![Diagram](image)
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.

![Diagram of Dinitz' algorithm](image)
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.

end of phase
Dinitz’ algorithm (as refined by Even and Itai)

**INITIALIZE**($G, f$)

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

**ADVANCE**($v$)

**IF** ($v = t$)

**AUGMENT**($P$).
Remove saturated edges from $L_G$.
$P \leftarrow \emptyset$.
**GOTO ADVANCE**($s$).

**IF** (there exists edge ($v, w$) $\in L_G$)

Add edge ($v, w$) to $P$.
**GOTO ADVANCE**($w$).

**ELSE**

**GOTO RETREAT**($v$).

**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**($u$).
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. $\leftarrow O(m)$ using BFS
- At most $m$ augmentations per phase. $\leftarrow O(mn)$ per phase (because an augmentation deletes at least one edge from $L_G$)
- At most $n$ retreats per phase. $\leftarrow O(m + n)$ per phase (because a retreat deletes one node from $L_G$)
- At most $mn$ advances per phase. $\leftarrow O(mn)$ 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

<table>
<thead>
<tr>
<th>Year</th>
<th>Method</th>
<th># Augmentations</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1955</td>
<td>augmenting path</td>
<td>$nC$</td>
<td>$O(mnC)$</td>
</tr>
<tr>
<td>1972</td>
<td>fattest path</td>
<td>$m \log(mC)$</td>
<td>$O(m^2 \log n \log (mC))$</td>
</tr>
<tr>
<td>1972</td>
<td>capacity scaling</td>
<td>$m \log C$</td>
<td>$O(m^2 \log C)$</td>
</tr>
<tr>
<td>1985</td>
<td>improved capacity scaling</td>
<td>$m \log C$</td>
<td>$O(mn \log C)$</td>
</tr>
<tr>
<td>1970</td>
<td>shortest augmenting path</td>
<td>$mn$</td>
<td>$O(m^2 n)$</td>
</tr>
<tr>
<td>1970</td>
<td>level graph</td>
<td>$mn$</td>
<td>$O(mn^2)$</td>
</tr>
<tr>
<td>1983</td>
<td>dynamic trees</td>
<td>$mn$</td>
<td>$O(mn \log n)$</td>
</tr>
</tbody>
</table>

Augmenting-path algorithms with $m$ edges, $n$ nodes, and integer capacities between 1 and $C$. 

- **Fat paths**: fattest path, capacity scaling, improved capacity scaling
- **Shortest paths**: shortest augmenting path, level graph, dynamic trees
# Maximum-flow algorithms: theory highlights

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

Max-flow algorithms with $m$ edges, $n$ nodes, and integer capacities between 1 and $C$. 

---

[Image: Table of maximum-flow algorithms with their years, methods, worst-case complexities, and discoverers. The table includes algorithms like simplex, augmenting paths, shortest augmenting paths, blocking flows, dynamic trees, improved capacity scaling, push–relabel, binary blocking flows, compact networks, interior–point methods, electrical flows, and an unspecified algorithm for dates 20xx. The complexities range from $O(m n^2 C)$ to $\tilde{O}(m^{10/7} C^{1/7})$.]
Maximum-flow algorithms: practice

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 \(O(n^2)\) 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 \(O(n^3)\) space.
Maximum-flow algorithms: practice

Caveat. Worst-case running time is generally not useful for predicting or comparing max-flow algorithm performance 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
csher@csmi.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.
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
tanmay08054@iiitd.ac.in
Dhruv Batra
dbatra@ttic.edu
IIIT-Delhi
Delhi, India
TTI-Chicago
Chicago, USA

Abstract

Algorithms for finding the maximum amount of flow possible in a network (or maxflow) 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

Documentation

CONTENTS

maxflow
Maximum flow in graph

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.

`mf = maxflow(G,s,t,algorithm)` specifies the maximum flow algorithm to use. This syntax is only available if `G` is a directed graph.
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*
Which max-flow algorithm to use for bipartite matching?

A. Ford–Fulkerson: $O(mnC)$.

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

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

D. Dinitz’ algorithm: $O(mn^2)$. 
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.

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 $O(m \cdot n^{1/2})$ 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.

- 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.

within a phase, length of shortest augmenting path does not change
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

remove from level graph all edges in augmenting path

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.
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.

level graph $L_G$

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

A. $O(m)$.
B. $O(m^{3/2})$.
C. $O(mn)$.
D. May not terminate.
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 $\Rightarrow |V_h| \leq n^{1/2}$.
Simple unit-capacity networks: analysis

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

- 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 $\leq 1$ outgoing residual edge$\}$.  
- $\text{cap}_f(A, B) \leq |V_h| \leq n^{1/2} \Rightarrow \text{val}(f) \geq \text{val}(f^*) - n^{1/2}$. ■
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, $\text{val}(f) \geq \text{val}(f^*) - n^{1/2}$.
- 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.