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 $c(e) \geq 0$ 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 $B$.

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

capacity $= 10 + 5 + 15 = 30$
**Minimum-cut problem**

**Def.** An *sr-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 \(B\).

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

- **Network flow: quiz 1**
  **Which is the capacity of the given *sr*-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 *sr-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]

\[
\text{inflow at } v = 5 + 5 + 0 = 10
\]

\[
\text{outflow at } v = 10 + 0 = 10
\]
Maximum-flow problem

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

- For each $e \in E$:
  \[ 0 \leq f(e) \leq c(e) \]  
  [capacity]

- For each $v \in V - \{s, t\}$:
  \[ \sum_{u \text{ in } v} f(e) - \sum_{v \text{ out } v} f(e) \]  
  [flow conservation]

Def. The *value* of a flow $f$ is:

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

---

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 $s$-$t$ 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.

---

**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.

---

**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.

---

**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.

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.

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 $G$. 

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

**Def.** An augmenting path is a simple $s \to t$ 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 $\text{val}(f') = \text{val}(f) + \text{bottleneck}(G_f, P)$.

\[
\text{AUGMENT}(f, c, P)
\]

\[
\delta \leftarrow \text{bottleneck capacity of augmenting path } P.
\]

\[
\text{FOR EACH edge } e \in P:
\]

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

\[
\text{ELSE } \ f(\text{reverse}(e)) \leftarrow f(\text{reverse}(e)) - \delta.
\]

\[
\text{RETURN } f.
\]

Ford–Fulkerson algorithm

**Ford–Fulkerson augmenting path algorithm.**

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

\[
\text{FORD–FULKERSON}(G)
\]

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

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

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

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

\[
\text{Update } G_f.
\]

\[
\text{RETURN } f.
\]

Network flow: quiz 2

**Which is the augmenting path of highest bottleneck capacity?**

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

7. NETWORK FLOW I

- max-flow and min-cut problems
- Ford–Fulkerson algorithm
- max-flow min-cut theorem
- capacity-scaling algorithm
- shortest augmenting paths
- Diniz' 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**

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

**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 \((A, B)\).

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

Pf.

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

by flow conservation, all terms except for \( v = s \) are 0

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

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

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 \((A', B')\): \( \text{cap}(A', B') \geq \text{val}(f) = \text{cap}(A, B) \).

Max-flow min-cut theorem

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

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."

A Note on the Maximum Flow Through a Network

P. ELLER, A. FISHER, AND C. E. RUSSELL

Summary. This note discusses the problem of maximizing the number of passenger train trips between two cities, given a number of constraints on the number of trains that can travel on any given route. It presents an algorithm for solving this problem and demonstrates its application to a real-world scenario. The algorithm is based on the Ford-Fulkerson method, which iteratively finds augmenting paths to increase the flow through the network until no more improvements can be made. The solution determines the maximum number of train trips that can be accommodated under the given constraints, ensuring an efficient use of resources.
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 $f$.

[$i \Rightarrow ii$]

* This is the weak duality corollary.

Max-flow min-cut theorem

[$ii \Rightarrow iii$] 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 $A = \text{set of nodes reachable from } s$ in residual network $G_f$.

* By definition of $A$: $s \in A$.

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

$$
\text{val}(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = \sum_{e \text{ out of } A} c(e) - 0 = \text{cap}(A, B)
$$

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 $G_f$.

* Argument from previous slide implies that 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).

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.

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 $\geq \Delta$.

Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems

JACK EDMONDS

University of Washington, Seattle, Seattle, Canada

RICHARD M. KARP

University of California, Berkeley, California

ABSTRACT. This paper presents new algorithms for the maximum flow problem, the bottleneck minimum-cost flow problem, and the $k$-reinforcement maximum flow problem. Upper bounds on the numbers of steps in these algorithms are derived, and it shows how to compare algorithms with respect to both the number of steps required by each algorithm.

Edmonds–Karp 1972 (USA)

Dinitz 1970 (Soviet Union)

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

Choose augmenting paths with:
- Max bottleneck capacity ("fattest"). ← how to find?
- Sufficiently large bottleneck capacity. ← next
- Fewest edges. ← ahead
Capacity-scaling algorithm: proof of correctness

**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 $c_r(e)$ 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 $\Delta = 1 \implies G_f(\Delta) = G_f$.
- 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.

**Pf.** 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 $\leq val(f) + m \Delta$.

**Pf.** Next slide.

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

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

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

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

7. Network Flow

» 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.

**Shortest-Augmenting-Path(G)**

FOREACH e ∈ E: f(e) ← 0.

G_f ← residual network of G with respect to flow f.

WHILE (there exists an s↝t path in G_f)

P ← BREADTH-FIRST-SEARCH(G_f).

f ← 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.

**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 \(O(m^2 \cdot n)\) time.

Pf.

\begin{itemize}
  \item \(O(m)\) time to find a shortest augmenting path via BFS.
  \item There are \(\leq mn\) augmentations.
    \begin{itemize}
      \item at most \(m\) augmenting paths of length \(k\) \(\rightarrow\) Lemma 1 + Lemma 2
      \item at most \(n−1\) different lengths ★
    \end{itemize}
\end{itemize}

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:

- \(ℓ(v)\) = number of edges in shortest \(s↝v\) path.
- \(L_0 = (V, E_0)\) is the subgraph of \(G\) that contains only those edges \((v, w) \in E\) with \(ℓ(w) = ℓ(v) + 1\).

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.

Visual representation of the level graph and network flow.
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 $s\to v$ 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 $s\to v$ path in $G$ iff $P$ is an $s\to v$ path in $L_G$.

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 $\Rightarrow O(m^2)$ [Dinitz 1970]
- Dynamic trees $\Rightarrow O(m n \log n)$ [Sleator–Tarjan 1983]

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. $\Rightarrow$ within a phase, length of shortest augmenting path does not change
- Construct level graph $L_0$.
  - Start at $s$, advance along an edge in $L_0$ until reach $t$ or get stuck.
  - If reach $t$, augment flow; update $L_0$; and restart from $s$.
  - If get stuck, delete node from $L_0$ 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 $L_0$.
  - Start at $s$, advance along an edge in $L_0$ until reach $t$ or get stuck.
  - If reach $t$, augment flow; update $L_0$; and restart from $s$.
  - If get stuck, delete node from $L_0$ 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.

---

**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)

\[
\text{INITIALIZE}(G, f)
\]
\[
L_G \leftarrow \text{level-graph of } G_f.
\]
\[
P \leftarrow \emptyset.
\]
\[
\text{GO TO ADVANCE}(s).
\]

\[
\text{ADVANCE}(v)
\]
\[
\text{IF } (v = t)
\]
\[
\text{AUGMENT}(P).
\]
\[
\text{Remove saturated edges from } L_G.
\]
\[
P \leftarrow \emptyset.
\]
\[
\text{GO TO ADVANCE}(s).
\]
\[
\text{RETREAT}(v)
\]
\[
\text{IF } (v = s)
\]
\[
\text{STOP}.
\]
\[
\text{ELSE}
\]
\[
\text{Delete } v \text{ (and all incident edges) from } L_G.
\]
\[
\text{Remove last edge } (u, v) \text{ from } P.
\]
\[
\text{GO TO ADVANCE}(u).
\]

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. \( \mathop{\longrightarrow} O(mn) \) using BFS
• At most \( m \) augmentations per phase. \( \mathop{\longrightarrow} O(mn) \) per phase (because an augmentation deletes at least one edge from \( L_G \))
• At most \( n \) retreats per phase. \( \mathop{\longrightarrow} O(m + n) \) per phase (because a retreat deletes one node from \( L_G \))
• At most \( mn \) advances per phase. \( \mathop{\longrightarrow} O(mn) \) 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>fastest 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 $C$.

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>$O(mn^2 C)$</td>
<td>Dantzig</td>
</tr>
<tr>
<td>1955</td>
<td>augmenting paths</td>
<td>$O(mnC)$</td>
<td>Ford-Fulkerson</td>
</tr>
<tr>
<td>1970</td>
<td>shortest augmenting paths</td>
<td>$O(mn^2)$</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>$O(mn \log n)$</td>
<td>Sleator-Tarjan</td>
</tr>
<tr>
<td>1985</td>
<td>improved capacity scaling</td>
<td>$O(mn \log C)$</td>
<td>Gabow</td>
</tr>
<tr>
<td>1988</td>
<td>push-relabel</td>
<td>$O(mn \log (n^2/m))$</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>$O(nm)$</td>
<td>Orlin</td>
</tr>
<tr>
<td>2014</td>
<td>interior–point methods</td>
<td>$\tilde{O}(m^{1/2} \log C)$</td>
<td>Lee–Sidford</td>
</tr>
<tr>
<td>2016</td>
<td>electrical flows</td>
<td>$\tilde{O}(m^{10/3} C^{1/4})$</td>
<td>Madry</td>
</tr>
</tbody>
</table>

20xx max-flow algorithms with $m$ edges, $n$ nodes, and integer capacities between 1 and $C$.

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 or 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 push-relabel concept of Karzanov is introduced. A push-flow is a flow, except that the total amount flowing into a vertex is allowed to exceed the total amount flowing out. The method maintains a push-flow in the original network and pushes local flow among vertices adjacent to zero residual in the sink that 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 $O(m^2)$ 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 allows efficient distributed and parallel implementations. A parallel implementation running in $O(n \log n)$ time using $n$ processors and $O(n)$ space is obtained. This time bound matches that of the McGeoch-Vazirani algorithm, which also uses $n$ processors but requires $O(n^2)$ space.

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

Rita V. Christley1 and Andrew V. Goldberg2

1 CREST Institute for Economic and Management Research, Kolkata 700052, India
2 Computer Science Department, Stanford University, Stanford, CA 94305, USA

Abstract. We describe efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous codes, yet more basic in some respects. The speedup comes from the realization that some basic algorithms, which are not intuitive, are in fact optimal in the worst case. We also include two novel techniques to which the current state of this problem remains open. These techniques allow the push-relabel algorithm to be efficiently implemented in the presence of all known restrictions while keeping a largely quadratic time bound.

Theory and Methodology

Computational investigations of maximum flow algorithms

Ravindra K. Ahuja 1, 2, Mauli Kodialam 3, Ajay K. Mishra 3, James B. Orlin 1, 4

1 Department of Operations Management Engineering, Indian Institute of Technology, Kharagpur, 721302, India
2 CREST Institute for Economic and Management Research, Kolkata 700052, India
3 School of Information Technology, New York University, New York, NY 10012, USA
4 School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Received 4 February 2001; accepted 27 June 1999.
Maximum-flow algorithms: practice

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

Maximum-flow algorithms: Matlab

Documentation

CONTENTS

maxflow

Syntax

mf = maxflow(G,s,t)

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.

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: $O(m n C)$.
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.

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.
- 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.

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.

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.

**advance**

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**

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.

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

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).
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. $O(m n)$. 

D. May not terminate.

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

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.