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 $e \in E$.

**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 $B$.

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

![Diagram of a flow network with capacities indicated on the edges.]

**Example:**

- Flow network example with capacities on the edges.
- Minimum-cut calculation: $10 + 5 + 15 = 30$.
Minimum-cut problem

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

**Def.** Its \textit{capacity} is the sum of the capacities of the edges from \(A\) to \(B\).

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

Network flow: quiz 1

\textbf{Which is the capacity of the given \textit{st}-cut?}

\begin{itemize}
  \item \textbf{A.} \(11 \ (20 + 25 - 8 - 11 - 9 - 6)\)
  \item \textbf{B.} \(34 \ (8 + 11 + 9 + 6)\)
  \item \textbf{C.} \(45 \ (20 + 25)\)
  \item \textbf{D.} \(79 \ (20 + 25 + 8 + 11 + 9 + 6)\)
\end{itemize}

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]}

Maximum-flow problem
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]

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

---

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

---

**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.
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 has \( f'(v, w) = 0 \).
• Greedy algorithm could choose \( s \rightarrow v \rightarrow w \rightarrow t \) as first augmenting 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.
\[
\begin{align*}
c_f(e) &= \begin{cases} 
  c(e) - f(e) & \text{if } e \in E \\
  f(e) & \text{if } e_{\text{reverse}} \in E 
\end{cases}
\end{align*}
\]

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 $\text{val}(f') = \text{val}(f) + \text{bottleneck}(G_f, P)$.

\[
\text{AUGMENT}(f, c, P)\\
\delta \leftarrow \text{bottleneck capacity of augmenting path } P.\\
\text{FOR EACH edge } e \in P:\\\n\quad \text{IF } (e \in E) \text{ } f(e) \leftarrow f(e) + \delta.\\n\quad \text{ELSE } f(\text{reverse}) \leftarrow f(\text{reverse}) - \delta.\\n\text{RETURN } f.
\]

Ford–Fulkerson algorithm

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

\[
\text{FORD–FULKERSON}(G)\\
\text{FOR EACH edge } e \in E: \quad f(e) \leftarrow 0.\\nG_f \leftarrow \text{residual network of } G \text{ with respect to flow } f.\\n\text{WHILE } (\text{there exists an } s \rightarrow t \text{ path } P \text{ in } G_f)\\n\quad f \leftarrow \text{AUGMENT}(f, c, P).\\n\quad \text{Update } G_f.\\n\text{RETURN } f.
\]

### 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. $A \rightarrow F \rightarrow B \rightarrow G \rightarrow C \rightarrow D \rightarrow H$
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 = $5 + 10 + 10 = 25$

edges from $q$ to $p$

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.

\[
\text{val}(f) = \sum_{v \in A} \left( \sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right) \\
\text{by flow conservation, all terms except for } v = s \text{ are 0} \\
\text{val}(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e).
\]

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 maximum flow from one given city to the other.”

A Note on the Maximum Flow Through a Network

P. Flanagan, A. Friesen, and C. F. Underwood

Summary. This note describes the problem of maximizing the number of passengers between a number of stations, each of which has a limited capacity. The problem is solved by an algorithm that is based on the Ford-Fulkerson method. The algorithm is verified by means of a computer program, and it is shown that the maximum number of passengers is achieved when the network is a spanning tree. The results are then applied to a real-world problem, in which the maximum number of passengers is achieved when the network is a spanning tree of the original network.
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 \( f \).  

\[ [i \Rightarrow ii] \]
  - This is the weak duality corollary.

\[ [ii \Rightarrow iii] \]
  - 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 \( f \): \( t \notin A \).

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

\[ [iii \Rightarrow i] \]
  - 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.

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? \( m, n, \text{ and } \log C \)

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

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

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

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 $\geq \Delta$.

Algorithm:

1. Set $\Delta = 100$.
2. Solve the maximum flow problem in $G_f(\Delta)$.
3. If no augmenting paths exist, terminate.
4. Otherwise, update $G_f(\Delta)$ and $\Delta$, and repeat.

Results:
- The algorithm terminates in polynomial time.
- The maximum flow $f$ is integral.
- The scaling parameter $\Delta$ is a power of 2.

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.

**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 is at least \(\val(f) + m \Delta\).

**Pf.** Next slide.

**Lemma 3.** There are \(2m\) augmentations per scaling phase.

**Pf.**

- Let \(f\) be the flow at the beginning of a \(\Delta\)-scaling phase.
- Lemma 2 \(\Rightarrow\) max-flow value is at least \(\val(f) + m (2 \Delta)\).
- 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 \(\Rightarrow\) \(O(m \log C)\) augmentations.
- Finding an augmenting path takes \(O(m)\) time. ●

---

**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.

**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 \(s \rightarrow t\) 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.

**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 $k$ \[ \text{Lemma 1 + Lemma 2} \]
  - at most $n-1$ different values of $k$ \[ \text{augmenting paths are simple paths} \]

---

Network flow: quiz 5

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

A. $D\rightarrow F$
B. $E\rightarrow F$
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 $s\rightarrow v$ 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 $s\rightarrow v$ path in $G$ iff $P$ is an $s\rightarrow v$ path in $L_G$. 

---

Network flow: quiz 5

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

A. $D\rightarrow F$
B. $E\rightarrow F$
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 $s\rightarrow v$ 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 $s\rightarrow v$ path in $G$ iff $P$ is an $s\rightarrow v$ 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 path that uses a back edge is longer than previous length) •

\[
\begin{array}{c}
\text{level graph } L_G \\
\begin{array}{c}
\begin{array}{c}
\bullet \quad \bullet \quad \bullet \\
\text{s} \quad \text{e} \quad \text{t}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{level graph } L_G' \\
\begin{array}{c}
\begin{array}{c}
\bullet \quad \bullet \quad \bullet \\
\text{s} \quad \text{e} \quad \text{t}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\ell = 0 \quad \ell = 1 \quad \ell = 2 \quad \ell = 3
\end{array}
\]

**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. •

\[
\begin{array}{c}
\text{level graph } L_G \\
\begin{array}{c}
\begin{array}{c}
\bullet \quad \bullet \quad \bullet \\
\text{s} \quad \text{e} \quad \text{t}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{level graph } L_G' \\
\begin{array}{c}
\begin{array}{c}
\bullet \quad \bullet \quad \bullet \\
\text{s} \quad \text{e} \quad \text{t}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\ell = 0 \quad \ell = 1 \quad \ell = 2 \quad \ell = 3
\end{array}
\]

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.

**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] •
  - Dynamic trees \( \Rightarrow O(m n \log n) \) [Sleator–Tarjan 1983]

A Data Structure for Dynamic Trees

David D. Sleator and Robert Endre Tarjan
Bell Laboratories, Murray Hill, NJ. 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 version of a level of operations: (1) insert a new tree, (2) delete a tree, (3) combine trees, (4) make a tree root, (5) delete a root, and (6) find the common ancestor of any two vertices in a tree. Each operation requires \( O(\log n) \) time. Using this data structure, one fast algorithm is obtained for the following problems:

1. Computing certain common ancestors.
2. Solving certain network flow problems including finding maximum flows, blocking trees, and cyclic flows.
4. Maintaining the network simplex algorithm for minimum-cost flows.

A most significant application of (1), an O(n\log n) time algorithm is obtained to find a maximum flow in a network of \( m \) vertices and \( m \) edges, beating by a factor of \( \log n \) the fastest algorithms 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.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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.
• Explicitly maintain level graph $L_G$.
• Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
• If reach $t$, augment 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.
• Explicitly maintain level graph $L_G$.
• Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
• If reach $t$, augment 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.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment flow; update $L_G$; and restart from $s$.
- If get stuck, delete node from $L_G$ and retreat to previous node.

Phase of normal augmentations.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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

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

Phase of normal augmentations.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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)

**Dinitz’ algorithm**

```plaintext
INITIALIZE(G, f)

\[ L_G \leftarrow \text{level-graph of } G_f. \]
\[ P \leftarrow \emptyset. \]
\[ \text{GOTO ADVANCE}(s). \]

**ADVANCE(v)**

IF \( v = t \)

AUGMENT(P).

Remove saturated edges from \( L_G \).
\[ P \leftarrow \emptyset. \]
\[ \text{GOTO ADVANCE}(s). \]

ELSE

If (there exists edge \( (v, w) \in L_G \))
Add edge \( (v, w) \) to \( P \).
\[ \text{GOTO ADVANCE}(w). \]

ELSE

\[ \text{GOTO RETREAT}(v). \]
```

**RETREAT(v)**

IF \( v = s \)

STOP.
ELSE

Delete \( v \) (and all incident edges) from \( L_G \).
Remove last edge \( (u, v) \) from \( P \).
\[ \text{GOTO ADVANCE}(u). \]
```

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
- At most \( n \) retreats per phase. \( O(m + n) \) per phase
- At most \( mn \) advances per phase. \( O(mn) \) per phase

**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).

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.

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(nC) )</td>
<td>( O(m^2 \log n \log(2C)) )</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>
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, Dinizt</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>$O(m \log C)$</td>
<td>Gabow</td>
</tr>
<tr>
<td>1988</td>
<td>push-relabel</td>
<td>$O(m n \log (m^2/m))$</td>
<td>Goldberg-Tarjan</td>
</tr>
<tr>
<td>1998</td>
<td>binary blocking flows</td>
<td>$O(m^{1/2} \log (n^2/m) \log C)$</td>
<td>Goldberg-Rao</td>
</tr>
<tr>
<td>2013</td>
<td>compact networks</td>
<td>$O(mn)$</td>
<td>Orlin</td>
</tr>
<tr>
<td>2014</td>
<td>interior-point methods</td>
<td>$O(m^{1/2} \log C)$</td>
<td>Lee-Sidford</td>
</tr>
<tr>
<td>2016</td>
<td>electrical flows</td>
<td>$O(m^{3/2} C^{1/2})$</td>
<td>Madry</td>
</tr>
<tr>
<td>20xx</td>
<td></td>
<td>???</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Bret V. Olander and Andrew V. Goldberg

Abstract. We provide an efficient implementation of push-relabel method for the maximum flow problem. The running time is linear to the number of edges in the graph. The method is more efficient than previous implementations. The times are given with benchmarks for a wide variety of problems. The performance of the methods of all known algorithms is better than quadratic.

Maximum-flow algorithms: practice


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 push-relabel 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 central mechanism is a push in the original network and push local flow excess toward the sink along what are estimated to be augmenting 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 a n-vertex graph. By incorporating the dynamic tree data structure of Sleator and Tarjan, we obtain a version of the algorithm running in $O(n^{1.5} \log n)$ 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 yields efficient distributed and parallel implementations. A parallel implementation running in $O(n^{1.5} \log n)$ time using $n^{1/8}$ processors and $O(n)$ space is obtained. This time bound matches that of the Tsukiyamakatcha algorithms, which also uses $O(n)$ processors but requires $O(n)$ space.

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

Yuri Boykov and Vladimir Kolmogorov

Abstract. We compare the performance of several classical max-flow algorithms, including the recent max-flow/min-cut algorithm of V. Kolmogorov and D. Cremers. The algorithms are based on Ford-Fulkerson and push-relabel methods, implemented in C++ and C. We compare the running times of several standard algorithms, as well as a new algorithm that we have recently developed. The algorithms are tested on both large and small graphs.

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

Tommy Varma

Abstract. We compare 30 different implementations and find that the most popular and well-optimized versions of Maxflow (G+) are no longer the fastest algorithm available, especially for dense graphs.
Maximum-flow algorithms: Matlab

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: $O(m \text{val}(f^*))$.
B. Capacity scaling: $O(m^2 \log C)$.
C. Shortest augmenting path: $O(m^2 n)$.
D. Dinitz’ algorithm: $O(m n^2)$.

Maximum-flow algorithms: Google
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 either (i) exactly one entering edge or (ii) exactly one leaving edge (or both).

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

**Ex.** Bipartite matching.

\[
\begin{array}{c}
\text{Simple unit-capacity networks} \\
\text{Phase of normal augmentations.} \\
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment flow; update $L_G$; and restart from $s$.
- If get stuck, delete node from $L_G$ and go to previous node.
\end{array}
\]

**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 $O(m \sqrt{n})$ 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.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment 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.**
- Explicitly maintain level graph \( L_G \).
- Start at \( s \), advance along an edge in \( L_G \) until reach \( t \) or get stuck.
- If reach \( t \), augment 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 since each edge traversed and deleted at most once.
- \( O(1) \) per node since each node deleted at most once.

---

Simple unit-capacity networks: analysis

**Phase of normal augmentations.**
- Explicitly maintain level graph \( L_G \).
- Start at \( s \), advance along an edge in \( L_G \) until reach \( t \) or get stuck.
- If reach \( t \), augment flow; update \( L_G \); and restart from \( s \).
- If get stuck, delete node from \( L_G \) and go to previous node.

**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 \( > n^{1/2} \).
- Level graph has \( > n^{1/2} \) levels.
- Let \( 1 \leq h \leq n^{1/2} \) be layer with min number of nodes \( \Rightarrow |V_h| \leq n^{1/2} \).
**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 $> n^{1/2}$.
- Level graph has $> n^{1/2}$ levels.
- Let $1 \leq h \leq n^{1/2}$ be layer 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\}.
- $\text{cap}(A, B) \leq |V_h| \leq n^{1/2} \Rightarrow \text{val}(f) \geq \text{val}(f^*) - n^{1/2}$. 

![Diagram of residual network and residual edges]