### Flow network

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

#### Def.
- A **st-cut (cut)** is a partition $(A, B)$ of the vertices with $s \in A$ and $t \in B$.
- Its **capacity** is the sum of the capacities of the edges from $A$ to $B$.

#### Minimum cut problem

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$
Minimum cut problem
Def. A st-cut (cut) is a partition \((A, B)\) of the vertices with \(s \in A\) and \(t \in B\).
Def. Its capacity is the sum of the capacities of the edges from \(A\) to \(B\).
Min-cut problem. Find a cut of minimum capacity.

Minimum cut problem
Def. A st-cut (cut) is a partition \((A, B)\) of the vertices with \(s \in A\) and \(t \in B\).
Def. Its capacity is the sum of the capacities of the edges from \(A\) to \(B\).

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

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

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

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

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

Towards a max-flow algorithm

Greedy algorithm.
- Start with \(f(e) = 0\) for all 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.

network G

7. Network Flow I

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

Towards a max-flow algorithm

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

network G
Towards a max-flow algorithm

Greedy algorithm.

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

network \( G \)

\[
\begin{align*}
s & \quad 0/10 \\
0/10 & \quad 2/2 \\
8/8 & \quad 0/6 \\
0/10 & \\
2/10 & \\
8 + 2 = 10 & \\
t & \\
0/10 & \\
2 & \\
2 & \\
8/10 & \\
\end{align*}
\]

ending flow value = 16

Towards a max-flow algorithm

Greedy algorithm.

- Start with \( f(e) = 0 \) for all 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.

network \( G \)

\[
\begin{align*}
10/10 & \quad 2/2 \\
6/10 & \quad 8/8 \\
2/10 & \quad 2/2 \\
8/10 & \\
8/10 & \\
10 + 6 = 16 & \\
t & \\
8/9 & \\
6/9 & \\
10/10 & \\
\end{align*}
\]

but max-flow value = 19
Residual graph

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

Residual edge.
- "Undo" flow sent.
- \( e = (u, v) \) and \( e^R = (v, u) \).
- Residual capacity:
  \[
  c_f(e) = \begin{cases} 
  c(e) - f(e) & \text{if } e \in E \\
  f(e) & \text{if } e^R \in E
  \end{cases}
  \]

Residual graph: \( G_f = (V, E_f) \).
- Residual edges with positive residual capacity.
- \( E_f = \{ e : f(e) < c(e) \} \cup \{ e^R : f(e^R) > 0 \} \).
- 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 \( P \) in the residual graph \( G_f \).

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

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

Ford-Fulkerson algorithm

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

Ford-Fulkerson algorithm demo

Augmenting path

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

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

\[
\text{FOREACH} \; e \in P
\]

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

\[
\text{ELSE} \; f(e^R) \leftarrow f(e^R) - b.
\]

\[
\text{RETURN} \; f.
\]
Ford-Fulkerson algorithm demo

network G

residual graph $G_f$

Ford-Fulkerson algorithm demo

network G

residual graph $G_f$

Ford-Fulkerson algorithm demo

network G

residual graph $G_f$

Ford-Fulkerson algorithm demo

network G

residual graph $G_f$
7. **Network Flow I**

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

---

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

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

**Relationship between flows and cuts**

Net flow across cut $= 5 + 10 + 10 = 25$

**Value of flow** $= 25$
Relationship between flows and cuts

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

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

Pf. 

\[
v(f) = \sum_{e \text{ out of } s} f(e)
\]

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

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

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

\[
\]

Relationship between flows and cuts

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

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

Pf. 

\[
v(f) = \sum_{e \text{ out of } s} f(e)
\]

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

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

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

\[
\]

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

Pf. 

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

flow-value lemma

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

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

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

\[
\]

value of flow = 27

\[
\leq
\]

capacity of cut = 30
**Max-flow min-cut theorem**

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

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

**Pf.** The following three conditions are equivalent for any flow \( f \):

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

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

iii. There is no augmenting path with respect to \( f \).

\[ \text{[ i \Rightarrow ii ]} \]

- Suppose that \((A, B)\) is a cut such that \( \text{cap}(A, B) = \text{val}(f) \).
- Then, for any flow \( f' \), \( \text{val}(f') \leq \text{cap}(A, B) = \text{val}(f) \).
- Thus, \( f \) is a max-flow.  ▪

\[ \text{by assumption} \]

weak duality

\[ \text{[ ii \Rightarrow iii ]} \]

We prove contrapositive: \( \neg \text{iii} \Rightarrow \neg \text{ii} \).

- 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**

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

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

**Pf.** The following three conditions are equivalent for any flow \( f \):

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

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

iii. There is no augmenting path with respect to \( f \).

\[ \text{[ ii \Rightarrow iii ]} \]

We prove contrapositive: \( \neg \text{iii} \Rightarrow \neg \text{ii} \).

- 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.  ▪

7. Network Flow I

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

**SECTION 7.3**
Assumption. Capacities are integers between 1 and \(C\).

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

Theorem. The algorithm terminates in at most \(\text{val}(f^*) \leq nC\) iterations.

Pf. Each augmentation increases the value by at least 1.

Corollary. The running time of Ford-Fulkerson is \(O(mnC)\).

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

Integrality theorem. Then exists a max-flow \(f^*\) for which every flow value \(f^*(e)\) is an integer.

Pf. Since algorithm terminates, theorem follows from invariant.

Choosing good augmenting paths

Use care when selecting augmenting paths.
- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.
- If capacities are irrational, algorithm not guaranteed to terminate.

Goal. Choose augmenting paths so that:
- Can find augmenting paths efficiently.
- Few iterations.

Bad case for Ford-Fulkerson

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.
   - \(s \rightarrow v \rightarrow w \rightarrow f\)
   - \(s \rightarrow v \rightarrow w \rightarrow f\)
   - \(s \rightarrow v \rightarrow w \rightarrow f\)
   - \(s \rightarrow v \rightarrow w \rightarrow f\)
   - \(s \rightarrow v \rightarrow w \rightarrow f\)
   - \(s \rightarrow v \rightarrow w \rightarrow f\)

Each augmenting path sends only \(1\) unit of flow (# augmenting paths = \(2C\))
Capacity-scaling algorithm

**Intuition.** Choose augmenting path with highest bottleneck capacity:
it increases flow by max possible amount in given iteration.
- Don’t worry about finding exact highest bottleneck path.
- Maintain scaling parameter $\Delta$.
- Let $G_\Delta$ be the subgraph of the residual graph consisting only of arcs with capacity $\geq \Delta$.

```
G
G(\Delta), \Delta = 100
```

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

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

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

**Pf.**
- By integrality invariant, when $\Delta = 1 \Rightarrow G_\Delta = G_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths.

```
CAPACITY-SCALING(G, s, t, c)
```

- FOREACH edge $e \in E : f(e) \leftarrow 0$.
- $\Delta \leftarrow$ largest power of 2 $\leq C$.
- WHILE ($\Delta \geq 1$)
  - $G(\Delta) \leftarrow \Delta$-residual graph.
  - WHILE (there exists an augmenting path $P$ in $G_\Delta$)
    - $f \leftarrow$ AUGMENT($f$, $c$, $P$).
    - Update $G_\Delta$.
    - $\Delta \leftarrow \Delta / 2$.
  - RETURN $f$.

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

**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 value of the max-flow $\leq val(f) + m \Delta$.

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

**Pf.**
- Let $f$ be the flow at the end of the previous scaling phase.
  - $\text{LEMA 2 } \Rightarrow val(f^*) \leq val(f) + 2m \Delta$.
  - Each augmentation in a $\Delta$-phase increases $val(f)$ by at least $\Delta$.

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

**Pf.** Follows from **Lemma 1** and **Lemma 3**.
7. **Network Flow I**

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

---

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

**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 cut \( A, s \in A \).
- By definition of flow \( f, t \notin A \).

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

---

**Shortest augmenting path**

**Q.** Which augmenting path?

**A.** The one with the fewest number of edges.

---

**Shortest augmenting path: overview of analysis**

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

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

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

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

\( \Rightarrow 1 \leq k < n \)

\( \Rightarrow O(m n) \) augmentations.
Shortest augmenting path: analysis

**Def.** Given a digraph $G = (V, E)$ with source $s$, its level graph is defined by:
- $\ell(v) =$ number of edges in shortest path from $s$ to $v$.
- $L_G = (V, E_G)$ is the subgraph of $G$ that contains only those edges $(v, w) \in E$ with $\ell(w) = \ell(v) + 1$.

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

**L1.** Throughout the algorithm, length of the shortest path never decreases.
- Let $f$ and $f'$ be flow before and after a shortest path augmentation.
- Let $L$ and $L'$ be level graphs of $G_f$ and $G_{f'}$.
- Only back edges added to $G_{f'}$ (any path with a back edge is longer than previous length).

**L2.** After at most $m$ shortest path augmentations, the length of the shortest augmenting path strictly increases.
- The bottleneck edge(s) is deleted from $L$ after each augmentation.
- No new edge added to $L$ until length of shortest path strictly increases.
Shortest augmenting path: review of analysis

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

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

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

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

**Note.** $\Theta(m n)$ augmentations necessary on some networks.
- Try to decrease time per augmentation instead.
- Simple idea $\Rightarrow O(m n^2)$ [Dinic 1970]
- Dynamic trees $\Rightarrow O(m n \log n)$ [Sleator-Tarjan 1983]

A data structure is proposed to maintain a collection of vertex-disjoint trees under a sequence of two kinds of operations: (1) update the structure to make a tree root at an arbitrary vertex, and (2) cut an edge, and it has operation that divides one tree into two by deleting an edge. Each operation requires $O(k)$ time. Using this data structure, new fast algorithms are obtained for the following problems:
2. Finding shortest network flow problems including finding maximum flows, blocking flows, and $\epsilon$-approx. flows.
3. Computing some kinds of constrained minimum spanning trees.
4. Understanding the network simplex algorithm for maximum cost flows.

The most significant application is (1), an O(log 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 the fastest algorithm previously known for sparse graphs.

**Blocking-flow algorithm**

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

**Phase of normal augmentations.**
- Explicitly maintain level graph $L_0$.
- Start at $s$, advance along an edge in $L_0$ until reach $t$ or get stuck.
- If reach $t$, augment and and update $L_0$.
- If get stuck, delete node from $L_0$ and go to previous node.
Two types of augmentations.
- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.
- Explicitly maintain level graph $L_G$.
- Start at $s$, advance along an edge in $L_G$ until reach $t$ or get stuck.
- If reach $t$, augment and and update $L_G$.
- If get stuck, delete node from $L_G$ and go to previous node.
Two types of augmentations.
 • Normal: length of shortest path does not change.
 • Special: length of shortest path strictly increases.

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

Blocking-flow algorithm

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

Phase of normal augmentations.
 • Explicitly maintain level graph \( L_G \).
 • Start at \( s \), advance along an edge in \( L_G \) until reach \( t \) or get stuck.
 • If reach \( t \), augment and update \( L_G \).
 • If get stuck, delete node from \( L_G \) and go to previous node.
Two types of augmentations.
- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

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

---

**Blocking-flow algorithm**

**INITIALIZE** \( (G, s, t, f, c) \)

\[ L_G \leftarrow \text{level-graph of } G \_\_\_\_\_\_\_\_. \]

\[ P \leftarrow \emptyset. \]

**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** \( (u) \).

---

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

---

**Blocking-flow algorithm**

**Lemma.** A phase can be implemented in \( O(mn) \) time.

**Pf.**
- Initialization happens once per phase. \( \quad \text{O}(m) \text{ using BFS} \)
- At most \( m \) augmentations per phase. \( \quad \text{O}(mn) \text{ per phase} \)
- (because an augmentation deletes at least one edge from \( L_G \))
- At most \( n \) retreats per phase. \( \quad \text{O}(m + n) \text{ per phase} \)
- (because a retreat deletes one node from \( L_G \))
- At most \( mn \) advances per phase. \( \quad \text{O}(mn) \text{ per phase} \)
- (because at most \( n \) advances before retreat or augmentation)

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

**Pf.**
- By lemma, \( O(mn) \) time per phase.
- At most \( n \) phases (as in shortest augment path analysis).
Choosing good augmenting paths: summary

Assumption. Integer capacities between 1 and C.

<table>
<thead>
<tr>
<th>method</th>
<th># augmentations</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>augmenting path</td>
<td>n C</td>
<td>O(m n C)</td>
</tr>
<tr>
<td>fattest augmenting path</td>
<td>m log (mC)</td>
<td>O(m log n log (mC))</td>
</tr>
<tr>
<td>capacity scaling</td>
<td>m log C</td>
<td>O(m^2 log C)</td>
</tr>
<tr>
<td>improved capacity scaling</td>
<td>m log C</td>
<td>O(m n log C)</td>
</tr>
<tr>
<td>shortest augmenting path</td>
<td>m n</td>
<td>O(m^2 n)</td>
</tr>
<tr>
<td>improved shortest augmenting path</td>
<td>m n</td>
<td>O(m n^2)</td>
</tr>
<tr>
<td>dynamic trees</td>
<td>m n</td>
<td>O(m n log n)</td>
</tr>
</tbody>
</table>

Maximum flow algorithms: theory

<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^3 C)</td>
<td>Dantzig</td>
</tr>
<tr>
<td>1955</td>
<td>augmenting path</td>
<td>O(m^2 C)</td>
<td>Ford-Fulkerson</td>
</tr>
<tr>
<td>1970</td>
<td>shortest augmenting path</td>
<td>O(m^3)</td>
<td>Dinic, Edmonds-Karp</td>
</tr>
<tr>
<td>1970</td>
<td>fattest augmenting path</td>
<td>O(m^3 log m log (m C))</td>
<td>Dinic, Edmonds-Karp</td>
</tr>
<tr>
<td>1977</td>
<td>blocking flow</td>
<td>O(m^5/2)</td>
<td>Cherkasy</td>
</tr>
<tr>
<td>1978</td>
<td>blocking flow</td>
<td>O(m^3/2)</td>
<td>Galil</td>
</tr>
<tr>
<td>1983</td>
<td>dynamic trees</td>
<td>O(m^2 log m)</td>
<td>Sleator-Tarjan</td>
</tr>
<tr>
<td>1985</td>
<td>capacity scaling</td>
<td>O(m^2 log C)</td>
<td>Gabow</td>
</tr>
<tr>
<td>1997</td>
<td>length function</td>
<td>O(m^2/2 log m log C)</td>
<td>Goldberg-Rao</td>
</tr>
<tr>
<td>2012</td>
<td>compact network</td>
<td>O(m^2 / log m)</td>
<td>Orlin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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 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(mn) 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(mn log(mn)) 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 log n) time using n processors and O(n) space is obtained. This time bound matches that of the Skloosh-Yamdan algorithm, which also uses n processors but requires O(\sqrt{n}) space.

On Implementing Push-Relabel Method for the Maximum Flow Problem

Boris V. Cherkasyov1 and Andrew V. Goldberg2

1 Centro de Investigación y de Estudios Avanzados, Mexico City, Mexico 2 University of Waterloo, Waterloo, Ontario, Canada

Abstract. We describe efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous codes, and much faster than some previous codes. The quality of the codes, as measured by their speed and ease of implementation, is discussed. The implementation of all known methods seems to have a roughly quadratic growth rate.


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

---

**Bipartite matching**

**Q.** Which max-flow algorithm to use for bipartite matching?
- **Generic augmenting path:** $O(m |f*|) = O(m n)$.
- **Capacity scaling:** $O(m^2 \log U) = O(m^2)$.
- **Shortest augmenting path:** $O(m n^2)$.

**Q.** Suggests "more clever" algorithms are not as good as we first thought? **A.** No, just need more clever analysis!

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

---

**Unit-capacity simple networks**

**Def.** A network is a unit-capacity simple network if:
- Every edge capacity is 1.
- Every node (other than $s$ or $t$) has either (i) at most one entering edge or (ii) at most one leaving edge.

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

**Ex.** Bipartite matching.
Unit-capacity simple networks

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

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

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

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

Unit-capacity simple networks

---

Unit-capacity simple networks

---

Phase of normal augmentations.

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

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

**Lemma 1.** A phase of normal augmentations takes \(O(m)\) time.

**Proof.**
- \(O(m)\) 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. ▫

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

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

**Proof.**
- Let \(A = \{v : \ell(v) < h\} \cup \{v : \ell(v) = h\ \text{and} \ v \ \text{has} \leq 1 \ \text{outgoing residual edge}\}\).
- \(\text{cap}_f(A, B) \leq |V_h| \leq n^{1/2} \Rightarrow |f| \geq |f^*| - n^{1/2}\). ▫