Network Flow

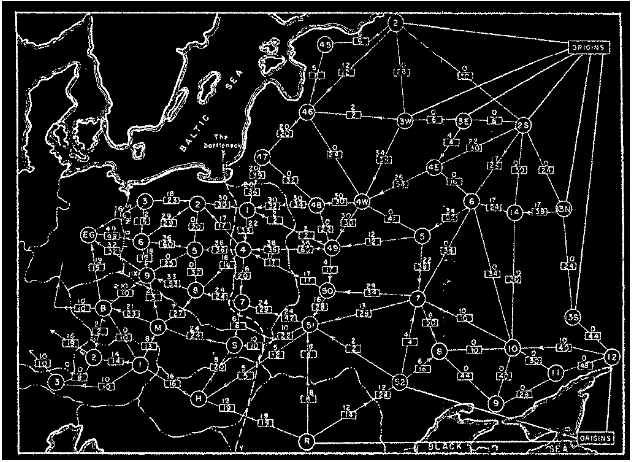
T. M. Murali

November 4, 9, 2009

Maximum Flow and Minimum Cut

- Two rich algorithmic problems.
- Fundamental problems in combinatorial optimization.
- Beautiful mathematical duality between flows and cuts.
- Numerous non-trivial applications:
  - Bipartite matching.
  - Data mining.
  - Project selection.
  - Airline scheduling.
  - Baseball elimination.
  - Image segmentation.
  - Network connectivity.
  - Open-pit mining.
  - Network reliability.
  - Distributed computing.
  - Egalitarian stable matching.
  - Security of statistical data.
  - Network intrusion detection.
  - Multi-camera scene reconstruction.
  - Gene function prediction.

History



(Soviet Rail Network, Tolstoi, 1930; Harris and Ross, 1955; Alexander Schrijver, *Math Programming*, 91: 3, 2002.)

Flow Networks

- Use directed graphs to model *transporation networks*:
  - edges carry traffic and have capacities.
  - nodes act as switches.
  - *source* nodes generate traffic, *sink* nodes absorb traffic.

Flow Networks

- Use directed graphs to model *transporation networks*:
  - edges carry traffic and have capacities.
  - nodes act as switches.
  - *source* nodes generate traffic, *sink* nodes absorb traffic.

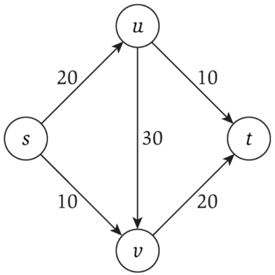


Figure 7.2 A flow network, with source s and sink t. The numbers next to the edges are the capacities.

- A *flow network* is a directed graph $G(V, E)$
  - Each edge $e \in E$ has a capacity $c(e) > 0$.
  - There is a single *source* node $s \in V$.
  - There is a single *sink* node $t \in V$.
  - Nodes other than s and t are *internal*.

Defining Flow

- In a flow network $G(V, E)$, an *s-t flow* is a function $f : E \rightarrow \mathbb{R}^+$ such that
  - (i) (*Capacity conditions*) For each $e \in E$, $0 \leq f(e) \leq c(e)$.
  - (ii) (*Conservation conditions*) For each internal node v,

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

- The *value* of a flow is $\nu(f) = \sum_{e \text{ out of } s} f(e)$.

Defining Flow

- In a flow network $G(V, E)$, an *s-t flow* is a function $f : E \to \mathbb{R}^+$ such that
  (i) (*Capacity conditions*) For each $e \in E$, $0 \leq f(e) \leq c(e)$.
  (ii) (*Conservation conditions*) For each internal node v,

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

- The *value* of a flow is $\nu(f) = \sum_{e \text{ out of } s} f(e)$.
- Useful notation:

$$f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e) \qquad f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$$

For $S \subseteq V$,

$$f^{\text{out}}(S) = \sum_{e \text{ out of } S} f(e) \qquad f^{\text{in}}(S) = \sum_{e \text{ into } S} f(e)$$

Maximum-Flow Problem

MAXIMUM FLOW

INSTANCE: A flow network G

SOLUTION: The flow with largest value in G

Maximum-Flow Problem

MAXIMUM FLOW

INSTANCE: A flow network G

SOLUTION: The flow with largest value in G

- Assumptions:
  1. No edges enter s, no edges leave t.
  2. There is at least one edge incident on each node.
  3. All edge capacities are integers.

Developing the Algorithm

- No known dynamic programming algorithm.
- Let us take a greedy approach.

Developing the Algorithm

- No known dynamic programming algorithm.
- Let us take a greedy approach.
  1. Start with zero flow along all edges (Figure 7.3(a)).

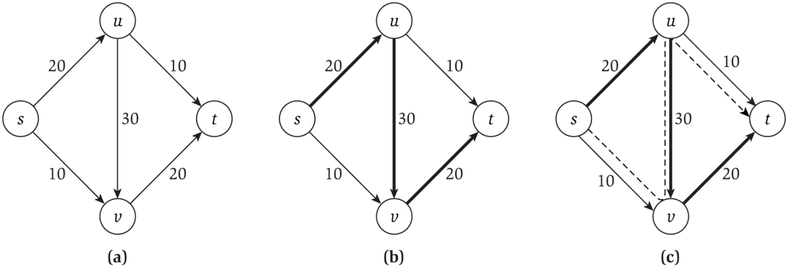


Figure 7.3 (a) The network of Figure 7.2. (b) Pushing 20 units of flow along the path s, u, v, t. (c) The new kind of augmenting path using the edge (u, v) backward.

Developing the Algorithm

- No known dynamic programming algorithm.
- Let us take a greedy approach.
  1. Start with zero flow along all edges (Figure 7.3(a)).
  2. Find an s-t path and push as much flow along it as possible (Figure 7.3(b)).

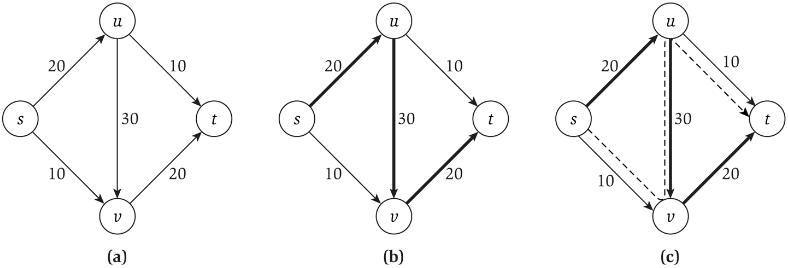


Figure 7.3 (a) The network of Figure 7.2. (b) Pushing 20 units of flow along the path s, u, v, t. (c) The new kind of augmenting path using the edge (u, v) backward.

Developing the Algorithm

- No known dynamic programming algorithm.
- Let us take a greedy approach.
  1. Start with zero flow along all edges (Figure 7.3(a)).
  2. Find an s-t path and push as much flow along it as possible (Figure 7.3(b)).
  3. Idea: Push flow along edges with leftover capacity and undo flow on edges already carrying flow.

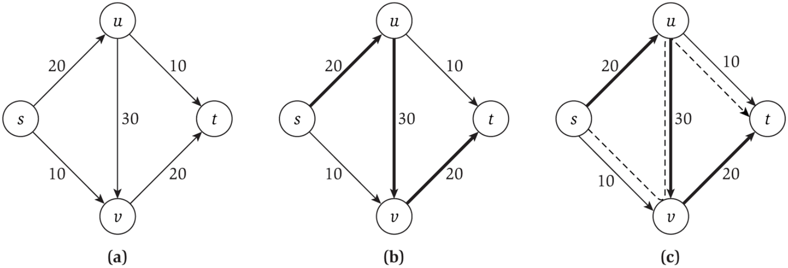


Figure 7.3 (a) The network of Figure 7.2. (b) Pushing 20 units of flow along the path s, u, v, t. (c) The new kind of augmenting path using the edge (u, v) backward.

Residual Graph

- Given a flow network $G(V, E)$ and a flow f on G, the *residual graph* G_f of G with respect to f is a directed graph such that
  (i) (*Nodes*) G_f has the same nodes as G.
  (ii) (*Forward edges*) For each edge $e = (u, v) \in E$ such that $f(e) < c(e)$, G_f contains the edge (u, v) with a *residual capacity* $c(e) - f(e)$.
  (iii) (*Backward edges*) For each edge $e \in E$ such that $f(e) > 0$, G_f contains the edge $e' = (v, u)$ with a residual capacity $f(e)$.

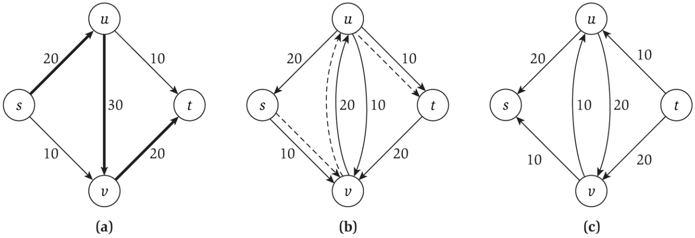


Figure 7.4 (a) The graph G with the path s, u, v, t used to push the first 20 units of flow. (b) The residual graph of the resulting flow f, with the residual capacity next to each edge. The dotted line is the new augmenting path. (c) The residual graph after pushing an additional 10 units of flow along the new augmenting path s, v, u, t.

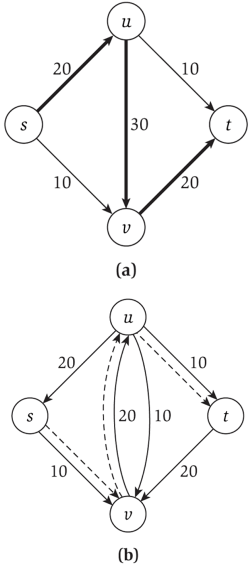
Augmenting Paths in a Residual Graph

- Let P be a simple s-t path in G_f.
- $bottleneck(P, f)$ is the minimum residual capacity of any edge in P.

Augmenting Paths in a Residual Graph

- Let P be a simple s-t path in G_f.
- *bottleneck*(P, f) is the minimum residual capacity of any edge in P.
- The following operation augment(f, P) yields a new flow f' in G:

```
augment(f, P)
  Let b = bottleneck(P, f)
  For each edge (u, v) ∈ P
    If e = (u, v) is a forward edge then
      increase f(e) in G by b
    Else ((u, v) is a backward edge, and let e = (v, u))
      decrease f(e) in G by b
    Endif
  Endfor
  Return(f)
```



Correctness of $\texttt{augment}(f, P)$

- A simple s-t path in the residual graph is an *augmenting path*.
- Let f' be the flow returned by $\texttt{augment}(f, P)$.
- Claim: f' is a flow. Verify capacity and conservation conditions.

Correctness of $\texttt{augment}(f, P)$

- A simple s-t path in the residual graph is an *augmenting path*.
- Let f' be the flow returned by $\texttt{augment}(f, P)$.
- Claim: f' is a flow. Verify capacity and conservation conditions.
  - Only need to check edges and internal nodes in P.

Correctness of $\texttt{augment}(f, P)$

- A simple s-t path in the residual graph is an *augmenting path*.
- Let f' be the flow returned by $\texttt{augment}(f, P)$.
- Claim: f' is a flow. Verify capacity and conservation conditions.
  - Only need to check edges and internal nodes in P.
  - Capacity condition on $e = (u, v) \in G_f$: Note that bottleneck$(P, f) \leq$ residual capacity of (u, v).

Correctness of $\texttt{augment}(f, P)$

- A simple s-t path in the residual graph is an *augmenting path*.
- Let f' be the flow returned by $\texttt{augment}(f, P)$.
- Claim: f' is a flow. Verify capacity and conservation conditions.
  - Only need to check edges and internal nodes in P.
  - Capacity condition on $e = (u, v) \in G_f$: Note that bottleneck$(P, f) \leq$ residual capacity of (u, v).
    - e is a forward edge:
      $0 \leq f(e) \leq f'(e) = f(e) + \text{bottleneck}(P, f) \leq f(e) + (c(e) - f(e)) = c(e)$.

Correctness of $\texttt{augment}(f, P)$

- A simple s-t path in the residual graph is an *augmenting path*.
- Let f' be the flow returned by $\texttt{augment}(f, P)$.
- Claim: f' is a flow. Verify capacity and conservation conditions.
  - Only need to check edges and internal nodes in P.
  - Capacity condition on $e = (u, v) \in G_f$: Note that bottleneck$(P, f) \leq$ residual capacity of (u, v).
    - e is a forward edge: $0 \leq f(e) \leq f'(e) = f(e) + \text{bottleneck}(P, f) \leq f(e) + (c(e) - f(e)) = c(e)$.
    - e is a backward edge: $c(e) \geq f(e) \geq f'(e) = f(e) - \text{bottleneck}(P, f) \geq f(e) - f(e) = 0$.

Correctness of $\texttt{augment}(f, P)$

- A simple s-t path in the residual graph is an *augmenting path*.
- Let f' be the flow returned by $\texttt{augment}(f, P)$.
- Claim: f' is a flow. Verify capacity and conservation conditions.
  - Only need to check edges and internal nodes in P.
  - Capacity condition on $e = (u, v) \in G_f$: Note that bottleneck$(P, f) \leq$ residual capacity of (u, v).
    - e is a forward edge:
      $0 \leq f(e) \leq f'(e) = f(e) + \text{bottleneck}(P, f) \leq f(e) + (c(e) - f(e)) = c(e)$.
    - e is a backward edge:
      $c(e) \geq f(e) \geq f'(e) = f(e) - \text{bottleneck}(P, f) \geq f(e) - f(e) = 0$.
  - Conservation condition on internal node $v \in P$.

Correctness of augment(f, P)

- A simple s-t path in the residual graph is an *augmenting path*.
- Let f' be the flow returned by augment(f, P).
- Claim: f' is a flow. Verify capacity and conservation conditions.
  - Only need to check edges and internal nodes in P.
  - Capacity condition on $e = (u, v) \in G_f$: Note that bottleneck$(P, f) \leq$ residual capacity of (u, v).
    - e is a forward edge:
      $0 \leq f(e) \leq f'(e) = f(e) + \text{bottleneck}(P, f) \leq f(e) + (c(e) - f(e)) = c(e)$.
    - e is a backward edge:
      $c(e) \geq f(e) \geq f'(e) = f(e) - \text{bottleneck}(P, f) \geq f(e) - f(e) = 0$.
  - Conservation condition on internal node $v \in P$. Four cases to work out.

Ford-Fulkerson Algorithm

```
Max-Flow
  Initially f(e) = 0 for all e in G
  While there is an s-t path in the residual graph G_f
    Let P be a simple s-t path in G_f
    f' = augment(f, P)
    Update f to be f'
    Update the residual graph G_f to be G_f'
  Endwhile
  Return f
```

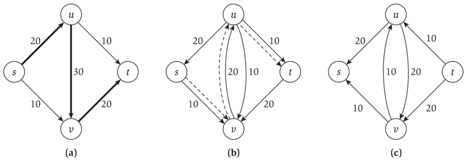


Figure 7.4 (a) The graph G with the path s, u, v, t used to push the first 20 units of flow. (b) The residual graph of the resulting flow f, with the residual capacity next to each

Analysis of the Ford-Fulkerson Algorithm

- Running time
  - Does the algorithm terminate?
  - If so, how many loops does the algorithm take?
- Correctness: if the algorithm terminates, why does it output a maximum flow?

Termination of the Ford-Fulkerson Algorithm

- Claim: at each stage, flow values and residual capacities are integers.

Termination of the Ford-Fulkerson Algorithm

- Claim: at each stage, flow values and residual capacities are integers. Prove by induction.

Termination of the Ford-Fulkerson Algorithm

- Claim: at each stage, flow values and residual capacities are integers. Prove by induction.
- Claim: Flow value strictly increases when we apply `augment`(f, P).

Termination of the Ford-Fulkerson Algorithm

- Claim: at each stage, flow values and residual capacities are integers. Prove by induction.
- Claim: Flow value strictly increases when we apply $\texttt{augment}(f, P)$. $v(f') = v(f) + \text{bottleneck}(P, f) > v(f)$.

Termination of the Ford-Fulkerson Algorithm

- Claim: at each stage, flow values and residual capacities are integers. Prove by induction.
- Claim: Flow value strictly increases when we apply `augment`(f, P). $v(f') = v(f) + \text{bottleneck}(P, f) > v(f)$.
- Claim: Maximum value of any flow is $C = \sum_{e \text{ out of } s} c(e)$.

Termination of the Ford-Fulkerson Algorithm

- Claim: at each stage, flow values and residual capacities are integers. Prove by induction.
- Claim: Flow value strictly increases when we apply $\texttt{augment}(f, P)$. $v(f') = v(f) + \text{bottleneck}(P, f) > v(f)$.
- Claim: Maximum value of any flow is $C = \sum_{e \text{ out of } s} c(e)$.
- Claim: Algorithm terminates in at most C iterations.

Termination of the Ford-Fulkerson Algorithm

- Claim: at each stage, flow values and residual capacities are integers. Prove by induction.
- Claim: Flow value strictly increases when we apply $\texttt{augment}(f, P)$. $v(f') = v(f) + \text{bottleneck}(P, f) > v(f)$.
- Claim: Maximum value of any flow is $C = \sum_{e \text{ out of } s} c(e)$.
- Claim: Algorithm terminates in at most C iterations.
- Claim: Algorithm runs in $O(mC)$ time.

Correctness of the Ford-Fulkerson Algorithm

- How large can the flow be?

Correctness of the Ford-Fulkerson Algorithm

- How large can the flow be?
- Can we characterise the magnitude of the flow in terms of the structure of the graph? For example, for every flow f, $\nu(f) \leq C = \sum_{e \text{out of } s} c(e)$.
- Is there a better bound?

Correctness of the Ford-Fulkerson Algorithm

- How large can the flow be?
- Can we characterise the magnitude of the flow in terms of the structure of the graph? For example, for every flow f, $\nu(f) \leq C = \sum_{e \text{out of } s} c(e)$.
- Is there a better bound?
- Idea: An *s-t cut* is a partition of V into sets A and B such that $s \in A$ and $t \in B$.
  - *Capacity* of the cut (A, B) is $c(A, B) = \sum_{e \text{ out of } A} c(e)$.
  - Intuition: For every flow f, $\nu(f) \leq c(A, B)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.
  - For every other node $v \in A$, $\quad 0 = f^{\text{out}}(v) - f^{\text{in}}(v)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.
  - For every other node $v \in A$, $0 = f^{\text{out}}(v) - f^{\text{in}}(v)$.
  - Summing up all these equations, $\nu(f) = \sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.
  - For every other node $v \in A$, $\quad 0 = f^{\text{out}}(v) - f^{\text{in}}(v)$.
  - Summing up all these equations, $\nu(f) = \sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right)$.
    - An edge e that has both ends in A or both ends out of A does not contribute.
    - An edge e that has its tail in A contributes $f(e)$.
    - An edge e that has its head in A contributes $-f(e)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.
  - For every other node $v \in A$, $\quad 0 = f^{\text{out}}(v) - f^{\text{in}}(v)$.
  - Summing up all these equations, $\nu(f) = \sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right)$.
    - An edge e that has both ends in A or both ends out of A does not contribute.
    - An edge e that has its tail in A contributes $f(e)$.
    - An edge e that has its head in A contributes $-f(e)$.
  - $\sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.
  - For every other node $v \in A$, $0 = f^{\text{out}}(v) - f^{\text{in}}(v)$.
  - Summing up all these equations, $\nu(f) = \sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right)$.
    - An edge e that has both ends in A or both ends out of A does not contribute.
    - An edge e that has its tail in A contributes $f(e)$.
    - An edge e that has its head in A contributes $-f(e)$.
  - $\sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
- Corollary: $\nu(f) = f^{\text{in}}(B) - f^{\text{out}}(B)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.
  - For every other node $v \in A$, $0 = f^{\text{out}}(v) - f^{\text{in}}(v)$.
  - Summing up all these equations, $\nu(f) = \sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right)$.
    - An edge e that has both ends in A or both ends out of A does not contribute.
    - An edge e that has its tail in A contributes $f(e)$.
    - An edge e that has its head in A contributes $-f(e)$.
  - $\sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
- Corollary: $\nu(f) = f^{\text{in}}(B) - f^{\text{out}}(B)$.
- $\nu(f) \leq c(A, B)$.

Fun Facts about Cuts

- Let f be any s-t flow and (A, B) any s-t cut.
- Claim: $\nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
  - $\nu(f) = f^{\text{out}}(s)$ and $f^{\text{in}}(s) = 0 \Rightarrow \nu(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.
  - For every other node $v \in A$, $\quad 0 = f^{\text{out}}(v) - f^{\text{in}}(v)$.
  - Summing up all these equations, $\nu(f) = \sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right)$.
    - An edge e that has both ends in A or both ends out of A does not contribute.
    - An edge e that has its tail in A contributes $f(e)$.
    - An edge e that has its head in A contributes $-f(e)$.
  - $\sum_{v \in A} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
- Corollary: $\nu(f) = f^{\text{in}}(B) - f^{\text{out}}(B)$.
- $\nu(f) \leq c(A, B)$.

$$\begin{aligned}
\nu(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) \\
&\leq f^{\text{out}}(A) = \sum_{e \text{ out of } A} f(e) \\
&\leq \sum_{e \text{ out of } A} c(e) = c(A, B).
\end{aligned}$$

Max-Flows and Min-Cuts

- Let f be any s-t flow and (A, B) any s-t cut. We proved $\nu(f) \leq c(A, B)$.

Max-Flows and Min-Cuts

- Let f be any s-t flow and (A, B) any s-t cut. We proved $\nu(f) \leq c(A, B)$.
- Very strong statement: The value of every flow is $\leq$ capacity of any cut.
- Corollary: The maximum flow is at most the smallest capacity of a cut.

Max-Flows and Min-Cuts

- Let f be any s-t flow and (A, B) any s-t cut. We proved $\nu(f) \leq c(A, B)$.
- Very strong statement: The value of every flow is $\leq$ capacity of any cut.
- Corollary: The maximum flow is at most the smallest capacity of a cut.
- Question: Is the reverse true? Is the smallest capacity of a cut at most the maximum flow?

Max-Flows and Min-Cuts

- Let f be any s-t flow and (A, B) any s-t cut. We proved $\nu(f) \leq c(A, B)$.
- Very strong statement: The value of every flow is $\leq$ capacity of any cut.
- Corollary: The maximum flow is at most the smallest capacity of a cut.
- Question: Is the reverse true? Is the smallest capacity of a cut at most the maximum flow?
- Answer: Yes, and the Ford-Fulkerson algorithm computes this cut!

Flows and Cuts

- Let $\bar{f}$ denote the flow computed by the Ford-Fulkerson algorithm.
- Enough to show $\exists$ s-t cut (A^*, B^*) such that $\nu(\bar{f}) = c(A^*, B^*)$.
- When the algorithm terminates, the residual graph has no s-t path.

Flows and Cuts

- Let $\bar{f}$ denote the flow computed by the Ford-Fulkerson algorithm.
- Enough to show $\exists$ s-t cut (A^*, B^*) such that $\nu(\bar{f}) = c(A^*, B^*)$.
- When the algorithm terminates, the residual graph has no s-t path.
- Claim: If f is an s-t flow such that G_f has no s-t path, then there is an s-t cut (A^*, B^*) such that $\nu(f) = c(A^*, B^*)$.
    - Claim applies to *any* flow f such that G_f has no s-t path, and not just to the flow $\bar{f}$ computed by the Ford-Fulkerson algorithm.

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- $A^* =$ set of nodes reachable from s in G_f, $B^* = V - A^*$.

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- A^* = set of nodes reachable from s in G_f, $B^* = V - A^*$.

- Claim: (A^*, B^*) is an s-t cut.

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- A^* = set of nodes reachable from s in G_f, $B^* = V - A^*$.

- Claim: (A^*, B^*) is an s-t cut.
- Claim: If $e = (u, v)$ such that $u \in A^*$, $v \in B^*$, then .

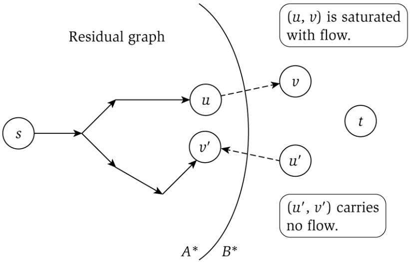


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- A^* = set of nodes reachable from s in G_f, $B^* = V - A^*$.

- Claim: (A^*, B^*) is an s-t cut.
- Claim: If $e = (u, v)$ such that $u \in A^*$, $v \in B^*$, then $f(e) = c(e)$.

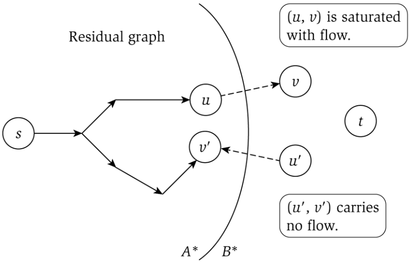


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- $A^* =$ set of nodes reachable from s in G_f, $B^* = V - A^*$.

- Claim: (A^*, B^*) is an s-t cut.
- Claim: If $e = (u, v)$ such that $u \in A^*$, $v \in B^*$, then $f(e) = c(e)$.
- Claim: If $e' = (u', v')$ such that $u' \in B^*$, $v' \in A^*$, then .

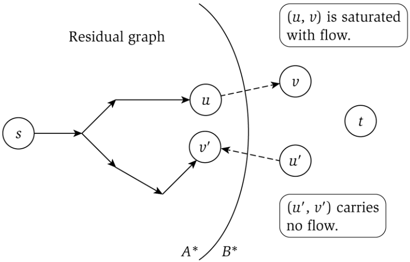


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- $A^* =$ set of nodes reachable from s in G_f, $B^* = V - A^*$.

- Claim: (A^*, B^*) is an s-t cut.
- Claim: If $e = (u, v)$ such that $u \in A^*$, $v \in B^*$, then $f(e) = c(e)$.
- Claim: If $e' = (u', v')$ such that $u' \in B^*$, $v' \in A^*$, then $f(e') = 0$.

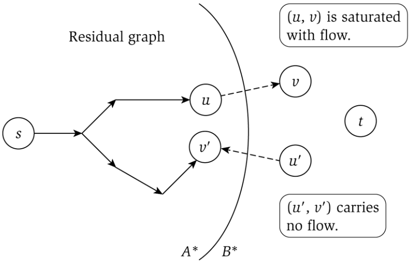


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- $A^* =$ set of nodes reachable from s in G_f, $B^* = V - A^*$.

- Claim: (A^*, B^*) is an s-t cut.
- Claim: If $e = (u, v)$ such that $u \in A^*$, $v \in B^*$, then $f(e) = c(e)$.
- Claim: If $e' = (u', v')$ such that $u' \in B^*$, $v' \in A^*$, then $f(e') = 0$.
- Claim: $\nu(f) = c(A^*, B^*)$.

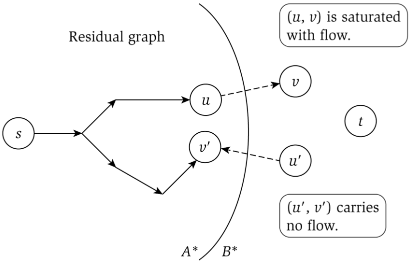


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Proof of Claim Relating Flows to Cuts

- Claim: f is an s-t flow and G_f has no s-t path $\Rightarrow \exists$ s-t cut (A^*, B^*), $\nu(f) = c(A^*, B^*)$.
- A^* = set of nodes reachable from s in G_f, $B^* = V - A^*$.

- Claim: (A^*, B^*) is an s-t cut.
- Claim: If $e = (u, v)$ such that $u \in A^*$, $v \in B^*$, then $f(e) = c(e)$.
- Claim: If $e' = (u', v')$ such that $u' \in B^*$, $v' \in A^*$, then $f(e') = 0$.
- Claim: $\nu(f) = c(A^*, B^*)$.

$$\begin{aligned}\nu(f) &= f^{\text{out}}(A) - f^{\text{in}}(A)\\ &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)\\ &= \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ into } A} 0 = c(A, B).\end{aligned}$$

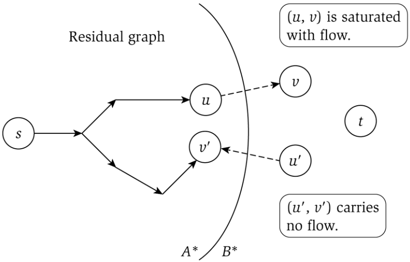


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Max-Flow Min-Cut Theorem

- The flow $\bar{f}$ computed by the Ford-Fulkerson algorithm is a maximum flow.
- Given a flow of maximum value, we can compute a minimum s-t cut in $O(m)$ time.
- In every flow network, there is a flow f and a cut (A, B) such that $\nu(f) = c(A, B)$.

Max-Flow Min-Cut Theorem

- The flow $\bar{f}$ computed by the Ford-Fulkerson algorithm is a maximum flow.
- Given a flow of maximum value, we can compute a minimum s-t cut in $O(m)$ time.
- In every flow network, there is a flow f and a cut (A, B) such that $\nu(f) = c(A, B)$.
- Max-Flow Min-Cut Theorem: in every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut.

Max-Flow Min-Cut Theorem

- The flow $\bar{f}$ computed by the Ford-Fulkerson algorithm is a maximum flow.
- Given a flow of maximum value, we can compute a minimum s-t cut in $O(m)$ time.
- In every flow network, there is a flow f and a cut (A, B) such that $\nu(f) = c(A, B)$.
- Max-Flow Min-Cut Theorem: in every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut.
- Corollary: If all capacities in a flow network are integers, then there is a maximum flow f where every flow value $f(e)$ is an integer.

Real-Valued Capacities

- If capacities are real-valued, Ford-Fulkerson algorithm may not terminate!
- But Max-Flow Min-Cut theorem is still true. Why?

Bad Augmenting Paths

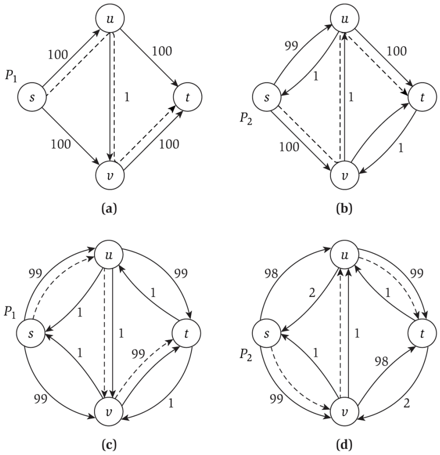


Figure 7.6 Parts (a) through (d) depict four iterations of the Ford-Fulkerson Algorithm using a bad choice of augmenting paths: The augmentations alternate between the path P_1 through the nodes s, u, v, t in order and the path P_2 through the nodes s, v, u, t in order.

Improving Ford-Fulkerson Algorithm

- Bad case for Ford-Fulkerson algorithm is when the bottleneck edge is the augmenting path has a low capacity.
- Idea: decrease number of iterations by picking s-t path with bottleneck edge of largest capacity.

Improving Ford-Fulkerson Algorithm

- Bad case for Ford-Fulkerson algorithm is when the bottleneck edge is the augmenting path has a low capacity.
- Idea: decrease number of iterations by picking s-t path with bottleneck edge of largest capacity. Computing this path can slow down each iteration considerably.

Improving Ford-Fulkerson Algorithm

- Bad case for Ford-Fulkerson algorithm is when the bottleneck edge is the augmenting path has a low capacity.
- Idea: decrease number of iterations by picking s-t path with bottleneck edge of largest capacity. Computing this path can slow down each iteration considerably.
- Modified idea: Maintain a *scaling parameter* Δ and choose only augmenting paths with bottleneck capacity at least Δ.

Improving Ford-Fulkerson Algorithm

- Bad case for Ford-Fulkerson algorithm is when the bottleneck edge is the augmenting path has a low capacity.
- Idea: decrease number of iterations by picking s-t path with bottleneck edge of largest capacity. Computing this path can slow down each iteration considerably.
- Modified idea: Maintain a *scaling parameter* Δ and choose only augmenting paths with bottleneck capacity at least Δ.
- $G_f(\Delta)$: residual network restricted to edges with residual capacities $\geq \Delta$.

Scaling Max-Flow Algorithm

```
Scaling Max-Flow
  Initially f(e) = 0 for all e in G
  Initially set Δ to be the largest power of 2 that is no larger
        than the maximum capacity out of s: Δ ≤ max_{e out of s} c_e
    While Δ ≥ 1
      While there is an s-t path in the graph G_f(Δ)
        Let P be a simple s-t path in G_f(Δ)
        f' = augment(f, P)
        Update f to be f' and update G_f(Δ)
      Endwhile
      Δ = Δ/2
    Endwhile
Return f
```

Correctness of the Scaling Max-Flow Algorithm

- Flow and residual capacities are integer valued throughout.
- When $\Delta = 1$, $G_f(\Delta)$ and G_f are identical.
- Therefore, when the scaling algorithm terminates, the flow is a maximum flow.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.
- Need to bound the number of iterations in each Δ-scaling phase.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.
- Need to bound the number of iterations in each Δ-scaling phase.
- Claim: During a Δ-scaling phase, each iteration increases the flow by $\geq \Delta$.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.
- Need to bound the number of iterations in each Δ-scaling phase.
- Claim: During a Δ-scaling phase, each iteration increases the flow by $\geq \Delta$.
- Claim: Let f be the flow at the end of a Δ-scaling phase. Then there is an s-t cut in (A, B) in G such that $\nu(f) \leq \nu(\bar{f}) \leq c(A, B) \leq \nu(f) + m\Delta$.
  - Proof idea: construct cut based on nodes reachable from s in $G_f(\Delta)$.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.
- Need to bound the number of iterations in each Δ-scaling phase.
- Claim: During a Δ-scaling phase, each iteration increases the flow by $\geq \Delta$.
- Claim: Let f be the flow at the end of a Δ-scaling phase. Then there is an s-t cut in (A, B) in G such that $\nu(f) \leq \nu(\bar{f}) \leq c(A, B) \leq \nu(f) + m\Delta$.
  - Proof idea: construct cut based on nodes reachable from s in $G_f(\Delta)$.
- Claim: the number of augmentations in a Δ-scaling phase is $\leq 2m$.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.
- Need to bound the number of iterations in each Δ-scaling phase.
- Claim: During a Δ-scaling phase, each iteration increases the flow by $\geq \Delta$.
- Claim: Let f be the flow at the end of a Δ-scaling phase. Then there is an s-t cut in (A, B) in G such that $\nu(f) \leq \nu(\bar{f}) \leq c(A, B) \leq \nu(f) + m\Delta$.
  - Proof idea: construct cut based on nodes reachable from s in $G_f(\Delta)$.
- Claim: the number of augmentations in a Δ-scaling phase is $\leq 2m$.
  - Base case: In the first Δ-scaling phase, each edge incident on s can be used in at most one augmenting path.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.
- Need to bound the number of iterations in each Δ-scaling phase.
- Claim: During a Δ-scaling phase, each iteration increases the flow by $\geq \Delta$.
- Claim: Let f be the flow at the end of a Δ-scaling phase. Then there is an s-t cut in (A, B) in G such that $\nu(f) \leq \nu(\bar{f}) \leq c(A, B) \leq \nu(f) + m\Delta$.
  - Proof idea: construct cut based on nodes reachable from s in $G_f(\Delta)$.
- Claim: the number of augmentations in a Δ-scaling phase is $\leq 2m$.
  - Base case: In the first Δ-scaling phase, each edge incident on s can be used in at most one augmenting path.
  - Induction: At the end of the previous Δ-scaling phase, let value of Δ be Γ and let f' be the flow: $\nu(f') \geq \nu(\bar{f}) - m\Gamma$.

Running time of the Scaling Max-Flow Algorithm

- *Δ-scaling phase*: one iteration of the algorithm's outer loop, with Δ fixed.
- Claim: the number of Δ-scaling phases is at most $1 + \lceil \log_2 C \rceil$.
- Need to bound the number of iterations in each Δ-scaling phase.
- Claim: During a Δ-scaling phase, each iteration increases the flow by $\geq \Delta$.
- Claim: Let f be the flow at the end of a Δ-scaling phase. Then there is an s-t cut in (A, B) in G such that $\nu(f) \leq \nu(\bar{f}) \leq c(A, B) \leq \nu(f) + m\Delta$.
  - Proof idea: construct cut based on nodes reachable from s in $G_f(\Delta)$.
- Claim: the number of augmentations in a Δ-scaling phase is $\leq 2m$.
  - Base case: In the first Δ-scaling phase, each edge incident on s can be used in at most one augmenting path.
  - Induction: At the end of the previous Δ-scaling phase, let value of Δ be Γ and let f' be the flow: $\nu(f') \geq \nu(\bar{f}) - m\Gamma$.
  - In the current Δ-scaling phase, the value of Δ is $\Gamma/2$. Let f be the flow at the end of this phase.
  - Since each iteration increases the flow by $\geq \Gamma/2$, if the current Δ-scaling phase continues for more than $2m$ iterations, then $\nu(f) > \nu(f') + 2m\Gamma/2 \geq \nu(\bar{f})$.
- Claim: the running time of the scaling max-flow algorithm is $O(m^2 \log C)$.

Other Maximum Flow Algorithms

- Running time of the Ford-Fulkerson algorithm is $O(mC)$, which is *pseudo-polynomial*: polynomial in the magnitudes of the numbers in the input.
- Scaling algorithm runs in time polynomial in the size of the input (the graph and the number of bits needed to represent the capacities).

Other Maximum Flow Algorithms

- Running time of the Ford-Fulkerson algorithm is $O(mC)$, which is *pseudo-polynomial*: polynomial in the magnitudes of the numbers in the input.
- Scaling algorithm runs in time polynomial in the size of the input (the graph and the number of bits needed to represent the capacities).
- Desire a *strongly polynomial* algorithm: running time is depends only on the *size* of the graph and is *independent* of the numerical values of the capacities (as long as numerical operations take $O(1)$ time).

Other Maximum Flow Algorithms

- Running time of the Ford-Fulkerson algorithm is $O(mC)$, which is *pseudo-polynomial*: polynomial in the magnitudes of the numbers in the input.
- Scaling algorithm runs in time polynomial in the size of the input (the graph and the number of bits needed to represent the capacities).
- Desire a *strongly polynomial* algorithm: running time is depends only on the *size* of the graph and is *independent* of the numerical values of the capacities (as long as numerical operations take $O(1)$ time).
- Edmonds-Karp, Dinitz: choose augmenting path to be the shortest path in G_f (use breadth-first search). Algorithm runs in $O(mn)$ iterations.
- Improved algorithms take time $O(mn \log n)$, $O(n^3)$, etc.
- Chapter 7.4: Preflow-push max-flow algorithm that is not based on augmenting paths. Runs in $O(n^2 m)$ or $O(n^3)$ time.