CS38
Introduction to Algorithms

Lecture 11

May 6, 2014

Outline

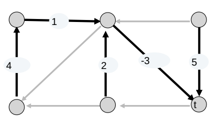
- Dynamic programming design paradigm
  - detecting negative cycles in a graph
  - all-pairs-shortest paths
- Network flow

* some slides from Kevin Wayne

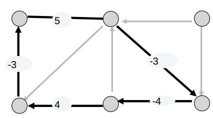
Shortest paths

Shortest path problem. Given a digraph with edge weights c_{vw} and no negative cycles, find cheapest $v \rightsquigarrow t$ path for each node v.

Negative cycle problem. Given a digraph with edge weights c_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Bellman-Ford

```
BELLMAN-FORD (V, E, c, t)

FOREACH node v ∈ V
    d(v) ← ∞.
    successor(v) ← null.
d(t) ← 0.
FOR i = 1 TO n – 1
    FOREACH node w ∈ V
        IF (d(w) was updated in previous iteration)
            FOREACH edge (v, w) ∈ E
                IF ( d(v) > d(w) + cvw)
                    d(v) ← d(w) + cvw.
                    successor(v) ← w.
    IF no d(w) value changed in iteration i, STOP.
```

1 pass

early stopping rule

Bellman-Ford

Lemma: Throughout algorithm, $d(v)$ is the cost of some $v \rightsquigarrow t$ path; after the i^{th} pass, $d(v)$ is no larger than the cost of the shortest $v \rightsquigarrow t$ path using $\leq i$ edges.

Proof (induction on i)

- Assume true after i^{th} pass.
- Let P be any $v \rightsquigarrow t$ path with $i + 1$ edges.
- Let (v, w) be first edge on path and let P' be subpath from w to t.
- By inductive hypothesis, $d(w) \leq c(P')$ since P' is a $w \rightsquigarrow t$ path with i edges.
- After considering v in pass i+1:

$$\begin{aligned} d(v) &\leq c_{vw} + d(w) \\ &\leq c_{vw} + c(P') \\ &= c(P) \end{aligned}$$

Theorem: Given digraph with no negative cycles, algorithm computes cost of shortest $v \rightsquigarrow t$ paths in $O(mn)$ time and $O(n)$ space.

Bellman-Ford

Lemma: If successor graph contains directed cycle W, then W is a negative cycle.

Proof:

- if $successor(v) = w$, we must have $d(v) \geq d(w) + c_{vw}$.
  (LHS and RHS are equal when $successor(v)$ is set; $d(w)$ can only decrease; $d(v)$ decreases only when $successor(v)$ is reset)
- Let $v_1 \to v_2 \to \ldots \to v_k$ be the nodes along the cycle W.
- Assume that (v_k, v_1) is the last edge added to the successor graph.
- Just prior to that:

$$\begin{aligned} d(v_1) &\geq d(v_2) + c(v_1, v_2) \\ d(v_2) &\geq d(v_3) + c(v_2, v_3) \\ &\vdots \\ d(v_{k-1}) &\geq d(v_k) + c(v_{k-1}, v_k) \\ d(v_k) &> d(v_1) + c(v_k, v_1) \end{aligned}$$

holds with strict inequality since we are updating $d(v_k)$

- add inequalities: $c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k) + c(v_k, v_1) < 0$

Bellman-Ford

Theorem: Given a digraph with no negative cycles, algorithm finds the shortest $s \rightsquigarrow t$ paths in $O(mn)$ time and $O(n)$ space.

Proof:

- The successor graph cannot have a cycle (previous lemma).
- Thus, following the successor pointers from s yields a directed path to t.
- Let $s = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = t$ be the nodes along this path P.
- Upon termination, if $successor(v) = w$, we must have $d(v) = d(w) + c_{vw}$. (LHS and RHS are equal when $successor(v)$ is set; $d(\cdot)$ did not change)
- Thus:

$$\begin{aligned} d(v_1) &= d(v_2) + c(v_1, v_2) \\ d(v_2) &= d(v_3) + c(v_2, v_3) \\ &\vdots \\ d(v_{k-1}) &= d(v_k) + c(v_{k-1}, v_k) \end{aligned}$$

since algorithm terminated

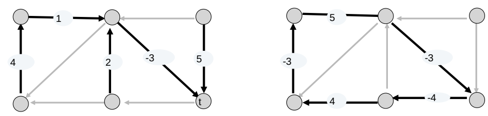
Adding equations yields $d(s) = d(t) + c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k)$

min cost of any $s \rightsquigarrow t$ path; 0; cost of path P

negative cycles

Shortest path problem. Given a digraph with edge weights c_{vw} and no negative cycles, find cheapest $v \rightsquigarrow t$ path for each node v.

Negative cycle problem. Given a digraph with edge weights c_{vw}, find a negative cycle (if one exists).

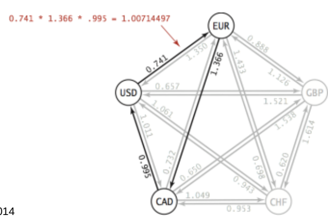


shortest-paths tree

negative cycle

negative cycles

- a motivating application: given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

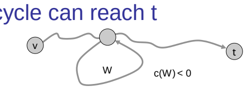


negative cycles

Lemma: OPT(n, v) = OPT(n – 1, v) for all *v* iff no negative cycle can reach *t*

Proof: (⇒)

- OPT(n, ·) = OPT(n-1, ·) implies OPT(i, ·) = OPT(n-1, ·) for all i > n
- but if negative cycle can reach t



- then OPT(i, v) → - ∞ as i → ∞

negative cycles

Lemma: OPT(n, v) = OPT(n – 1, v) for all *v* iff no negative cycle can reach *t*

Proof: (⇐)

- already argued no negative cycle implies shortest paths are all simple
- simple paths have at most n-1 edges

Bellman-Ford can *detect* negative cycles that reach t

negative cycles

- Can detect negative cycles that reach t; can we *find* from the successor graph?
  - yes, by the following lemma

Lemma: If OPT(n, v) < OPT(n – 1, v), the associated shortest path from v to t contains a cycle and every such cycle is negative

- can then find a negative cycle by tracing successor pointers seeking first repeat

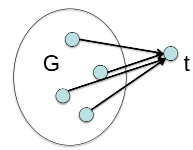
negative cycles

Lemma: If OPT(n, v) < OPT(n – 1, v), the associated shortest path from v to t contains a cycle and every such cycle is negative

Proof:
- trace the path from v to t following successor pointers, find a repeat; this implies a cycle W
- removing W results = path with $\leq$ n-1 edges
- OPT(n-1, v) > OPT(n, v) so W must be negative

negative cycles

- can detect and find negative cycles that reach t
  - how to solve the general problem?



- add weight 0 edges to new t
- negative cycle iff negative cycle that reaches t

Bellman-Ford

We have proved:

Theorem: Bellman-Ford operates in O(nm) time and O(n) space, and compute shortest s-t path in digraph G with no negative cycles.

If G has a negative cycle, Bellman-Ford detects and can find within same time bound.

all-pairs shortest paths

- Given directed graph with weighted edges (possibly negative) but no negative cycles
- Goal: compute shortest-path costs for all pairs of vertices
  - vertex set V = {1,2,…,n}
  - subproblems: OPT(i,j,k) = cost of shortest path from i to j with all intermediate nodes from {1,2,…, k}

all-pairs shortest paths

- OPT(i,j,k) = cost of shortest path from i to j with all intermediate nodes from {1,2,…, k}
- consider optimal path p
  - case 1: k is not on path p
    - OPT(i,j,k) = OPT(i,j,k-1)
  - case 2: k is on path p
    - break into path p_1 from i to k and path p_2 from k to j
    - path p simple, so p_1 doesn't use k as intermediate node
    - path p simple, so p_2 doesn't use k as intermediate node
    - OPT(i,j,k) = OPT(i,k,k-1) + OPT(k,j,k-1)

all-pairs shortest paths

Floyd-Warshall (directed graph with weights $c_{i,j}$)
```
1. OPT(i,j,0) = c_{i,j} for all i,j
2. for k = 1 to n
3.   for i = 1 to n
4.     for j = 1 to n
5.       OPT(i,j,k) = min{OPT(i,j,k-1), OPT(i,k,k-1)+OPT(k,j,k-1)}
6. return(OPT(·, ·, n))
```

- running time?
  - $O(n^3)$

Dynamic programming summary

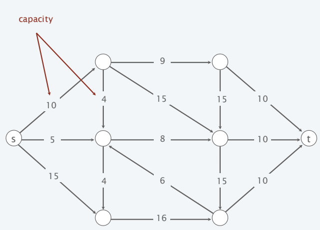
- identify subproblems:
  - present in recursive formulation, or
  - reason about what residual problem needs to be solved after a simple choice
- find order to fill in table
- running time (size of table)·(time for 1 cell)
- optimize space by keeping partial table
- store extra info to reconstruct solution

Max-Flow and Min-Cut

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 $e \in E$.

no parallel edges
no edge enters s
no edge leaves t

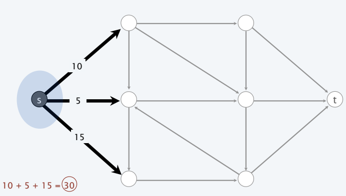


Minimum cut problem

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.

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



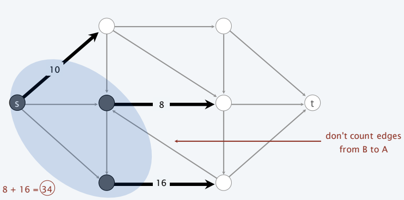
capacity = 10 + 5 + 15 = 30

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.

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



capacity = 10 + 8 + 16 = 34

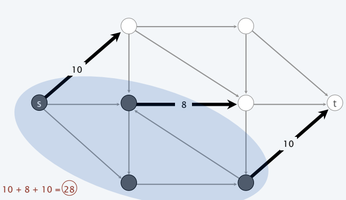
Minimum cut problem

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.

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

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

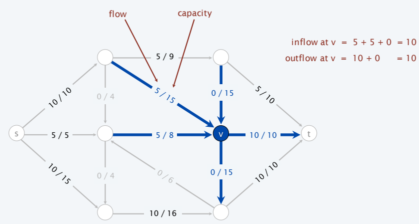


capacity = 10 + 8 + 10 = 28

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]



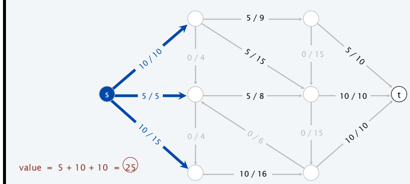
25

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: $val(f) = \sum_{e \text{ out of } s} f(e)$.



26

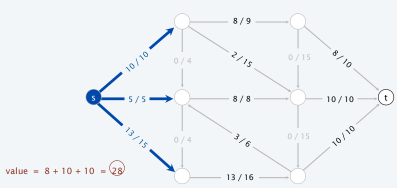
Maximum flow problem

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: $val(f) = \sum_{e \text{ out of } s} f(e)$.

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



27

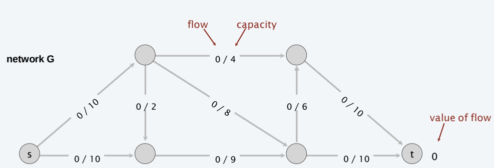
Ford-Fulkerson method

May 6, 2014 CS38 Lecture 11 28

Towards a max-flow algorithm

Greedy algorithm.

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

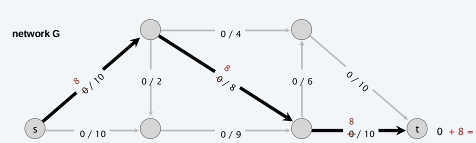


29

Towards a max-flow algorithm

Greedy algorithm.

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

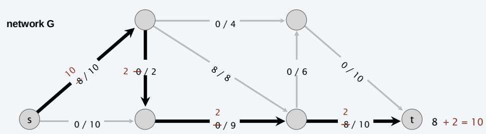


30

Towards a max-flow algorithm

Greedy algorithm.

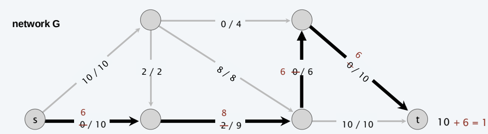
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.



Towards a max-flow algorithm

Greedy algorithm.

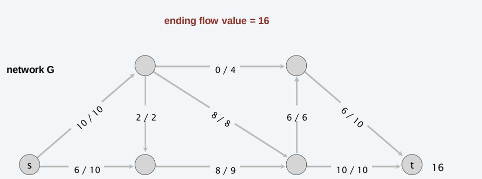
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.



Towards a max-flow algorithm

Greedy algorithm.

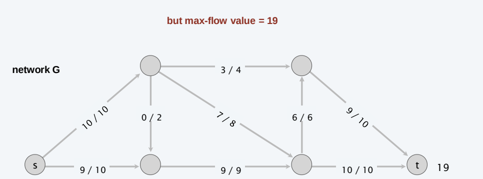
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.



Towards a max-flow algorithm

Greedy algorithm.

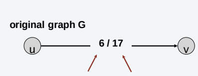
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.



Residual graph

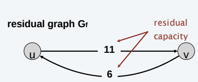
Original edge: $e = (u, v) \in E$.

- 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.
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$.
- Key property: f' is a flow in G_f iff $f + f'$ is a flow in G.

where flow on a reverse edge negates flow on a forward edge

Augmenting path

Def. An augmenting path is a simple $s \rightsquigarrow t$ 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 $val(f') = val(f) + bottleneck(G_f, P)$.

```
AUGMENT (f, c, P)

b ← bottleneck capacity of path P.
FOREACH edge e ∈ P
   IF (e ∈ E ) f(e) ← f(e) + b.
   ELSE        f(e^R) ← f(e^R) – b.
RETURN f.
```

Ford-Fulkerson algorithm

Ford-Fulkerson augmenting path algorithm.

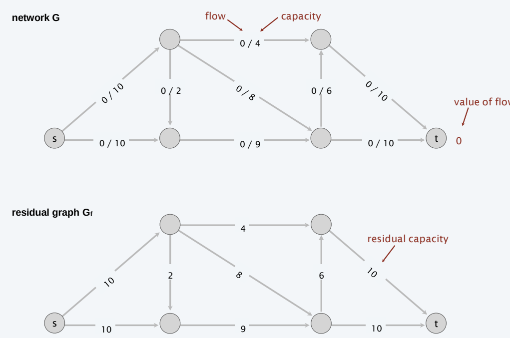
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an augmenting path P in the residual graph G_f.
- Augment flow along path P.
- Repeat until you get stuck.

FORD-FULKERSON (G, s, t, c)

FOREACH edge $e \in E : f(e) \leftarrow 0$.
  $G_f \leftarrow$ residual graph.
  WHILE (there exists an augmenting path P in G_f)
    $f \leftarrow$ AUGMENT (f, c, P).
    Update G_f.
  RETURN f.
}

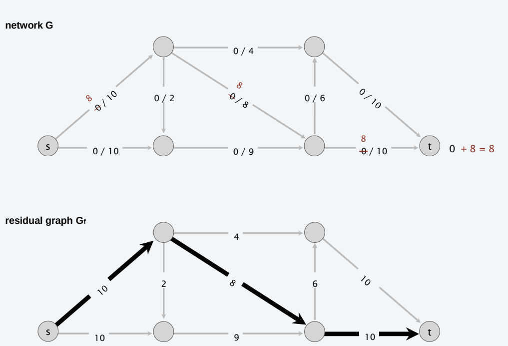
37

Ford-Fulkerson algorithm demo



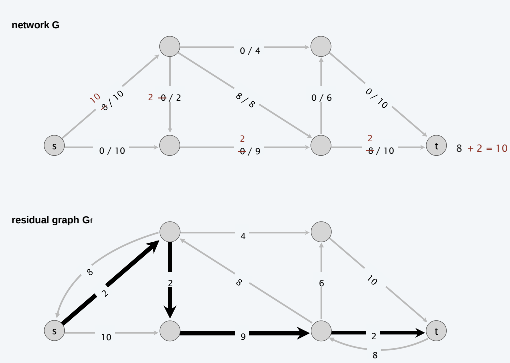
38

Ford-Fulkerson algorithm demo



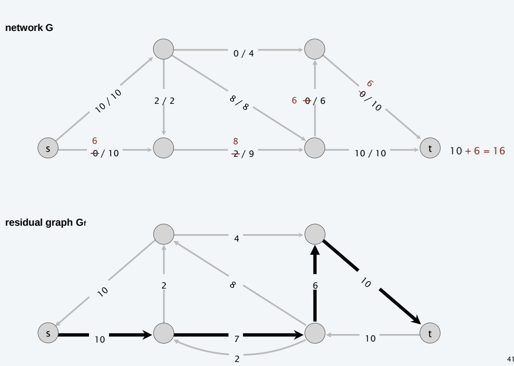
39

Ford-Fulkerson algorithm demo



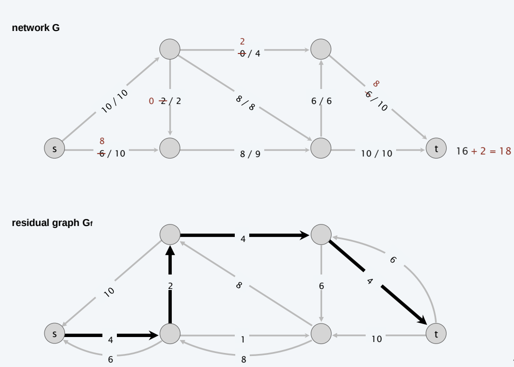
40

Ford-Fulkerson algorithm demo



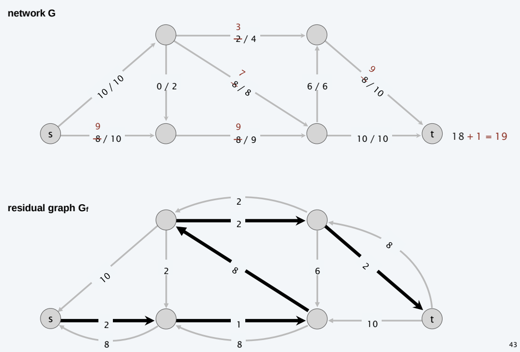
41

Ford-Fulkerson algorithm demo

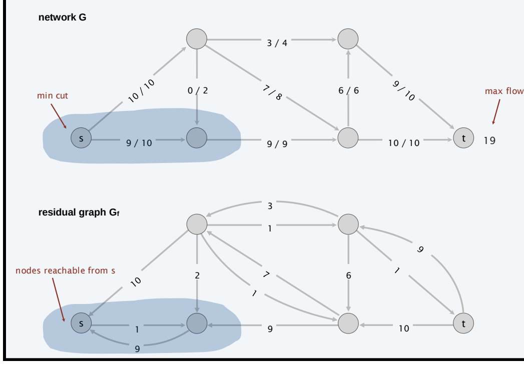


42

Ford-Fulkerson algorithm demo



Ford-Fulkerson algorithm demo

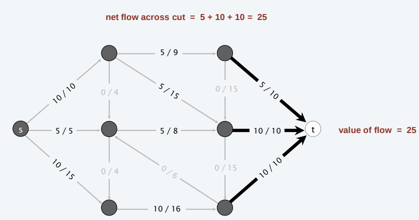


Min-flow max-cut Theorem

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.

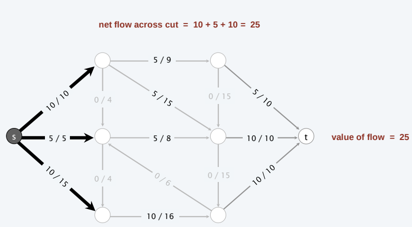
$$\sum_{e \text{ out of } A} f(e) \; - \sum_{e \text{ in to } A} f(e) \; = \; v(f)$$



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.

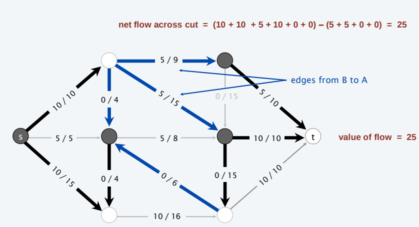
$$\sum_{e \text{ out of } A} f(e) \; - \sum_{e \text{ in to } A} f(e) \; = \; v(f)$$



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.

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



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 ⟶

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

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

49

Relationship between flows and cuts

Weak duality. Let f be any flow and (A, B) be any cut. Then, $v(f) \le cap(A, B)$.

Pf.

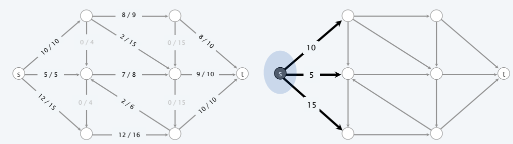
$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

flow-value lemma

$$\le \sum_{e \text{ out of } A} f(e)$$

$$\le \sum_{e \text{ out of } A} c(e)$$

$$= \text{cap}(A, B) \;\; ▪$$



value of flow = 27 ≤ capacity of cut = 30

50

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 $cap(A, B) = val(f)$.
ii. f is a max-flow.
iii. There is no augmenting path with respect to f.

[i ⇒ ii]
- Suppose that (A, B) is a cut such that $cap(A, B) = val(f)$.
- Then, for any flow f', $val(f') \le cap(A, B) = val(f)$.
- Thus, f is a max-flow. ▪

weak duality ↑ by assumption ↑

52

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 $cap(A, B) = val(f)$.
ii. f is a max-flow.
iii. There is no augmenting path with respect to f.

[ii ⇒ iii] We prove contrapositive: ~iii ⇒ ~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. ▪

53

Max-flow min-cut theorem

[iii ⇒ i]
- Let f be a flow with no augmenting paths.
- Let A be set of nodes reachable from s in residual graph G_f.
- By definition of cut A, $s \in A$.
- By definition of flow f, $t \notin A$.

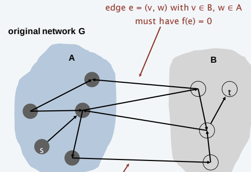
$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

flow-value lemma

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

$$= cap(A, B) \;\; ▪$$

edge e = (v, w) with v ∈ B, w ∈ A must have f(e) = 0



edge e = (v, w) with v ∈ A, w ∈ B must have f(e) = c(e)

54

Capacity-scaling algorithm

May 6, 2014 CS38 Lecture 11 56

Running time

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 $val(f^*) \leq nC$ 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. ▪

57

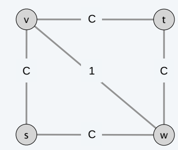
Bad case for Ford-Fulkerson

Q. Is generic Ford-Fulkerson algorithm poly-time in input size?
(m, n, and log C)

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

- $s \to v \to w \to t$
- $s \to w \to v \to t$
- $s \to v \to w \to t$
- $s \to w \to v \to t$
- …
- $s \to v \to w \to t$
- $s \to w \to v \to t$

← each augmenting path sends only 1 unit of flow (# augmenting paths = 2C)



58

Choosing good augmenting paths

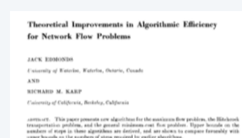
Use care when selecting augmenting paths.
- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.
- If capacities are irrational, algorithm not guaranteed to terminate!

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

59

Choosing good augmenting paths

Choose augmenting paths with:
- Max bottleneck capacity.
- Sufficiently large bottleneck capacity.
- Fewest number of edges.


Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems

JACK EDMONDS

AND

RICHARD M. KARP

Edmonds-Karp 1972 (USA)


ALGORITHM FOR SOLUTION OF A PROBLEM OF MAXIMUM FLOW IN A NETWORK WITH POWER ESTIMATION

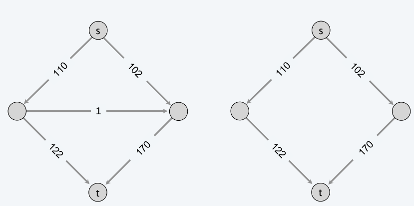
E. A. DINIC

Dinic 1970 (Soviet Union)

60

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 Δ.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting only of arcs with capacity $\geq \Delta$.



G_f $G_f(\Delta)$, $\Delta = 100$

61

Capacity-scaling algorithm

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

FOREACH edge e ∈ E : f(e) ← 0.
Δ ← largest power of 2 ≤ C.

WHILE (Δ ≥ 1)
   G_f(Δ) ← Δ-residual graph.
   WHILE (there exists an augmenting path P in G_f(Δ))
      f ← AUGMENT (f, c, P).
      Update G_f(Δ).
   Δ ← Δ / 2.

RETURN f.
```

62

Capacity-scaling algorithm: proof of correctness

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 $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths. ▪

63

Capacity-scaling algorithm: analysis of running time

Lemma 1. The outer while loop repeats $1 + \lceil \log_2 C \rceil$ times.
Pf. Initially $C/2 < \Delta \le C$; Δ decreases by a factor of 2 in each iteration. ▪

Lemma 2. Let f be the flow at the end of a Δ-scaling phase. Then, the value of the max-flow $\le$ $val(f) + m\,\Delta$. ⟵ proof on next slide

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.
- Lemma 2 $\Rightarrow$ $val(f^*) \le val(f) + 2\,m\,\Delta$.
- Each augmentation in a Δ-phase increases $val(f)$ by at least Δ. ▪

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. ▪

64

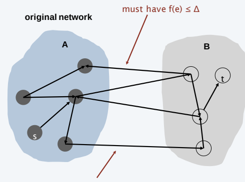
Capacity-scaling algorithm: analysis of running time

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

- We show there exists a cut (A, B) such that $cap(A, B) \le val(f) + m\,\Delta$.
- 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, $t \notin A$.

$$\begin{aligned} val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\ge \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ in to } A} \Delta \\ &= \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\ &\ge cap(A, B) - m\Delta \;\; ▪ \end{aligned}$$

edge e = (v, w) with v ∈ B, w ∈ A must have f(e) ≤ Δ



edge e = (v, w) with v ∈ A, w ∈ B must have f(e) ≥ c(e) − Δ

65