Clustering

K-means

Haotian Wang and Yiyan Li

School of Software, Shanghai Jiao Tong University

Introduction to clustering

Clustering

Definition

A cluster is a collection of objects which are “similar” between them and are “dissimilar” to the objects belonging to other clusters.

Definition

Clustering is the algorithm that recognizes clusters from a given data set.

Clustering

Definition

A cluster is a collection of objects which are “similar” between them and are “dissimilar” to the objects belonging to other clusters.

Definition

Clustering is the algorithm that recognizes clusters from a given data set.

Part of common application domains in which the clustering problem arises are as follows:

- Multimedia Data Analysis
- Responding to public health crises
- Intermediate Step for other fundamental data mining problems
- Intelligent Transportation

K-means Algorithm

The k-means clustering problem is one of the oldest and most important questions in all of computational geometry.

Given an integer k and a set of n data points in $\mathbb{R}^d$, the goal of this problem is to choose k centers so as to minimize the total squared distance between each point and its closest center.

The most common K-means algorithm was first proposed by Stuart Lloyd of Bell Labs in 1957.

The objective function to minimize is the within-cluster sum of squares (WCSS) cost:

$$\mathrm{Cost}(C_{1:k}, c_{1:k}) = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - c_i\|^2$$

where c_i is the centroid of cluster

Definition

Cluster centroid is the middle of a cluster.
A centroid is a vector that contains one number for each variable, where each number is the mean of a variable for the observations in that cluster.
The centroid can be thought of as the multi-dimensional average of the cluster.

Lemma

Let C be a cluster of points with its mean to be μ, and let c to be and arbitrary point.Then $\sum_{x \in C} \|x - c\|^2 = \sum_{x \in C} \|x - \mu\|^2 + |C| \cdot \|c - \mu\|^2$

So we denote that:

$$
\begin{aligned}
\mathrm{Cost}(C_{1:k}, c_{1:k}) &= \sum_{i=1}^{k} \sum_{x \in C_i} \|x - c_i\|^2 \\
&= \sum_{i=1}^{k} \Big(\sum_{x \in C_i} \|x - \mu_i\|^2 + |C_i| \cdot \|c_i - \mu_i\|^2 \Big) \\
&= \mathrm{Cost}(C_{1:k}, \mathrm{mean}(C_{1:k})) + \sum_{i=1}^{k} |C_i| \cdot \|c_i - \mu_i\|^2
\end{aligned}
$$

Toward a K-means Algorithm

The k-means algorithm iteratively calculates the sum of distance within a cluster and updates the partition.

1. Arbitrarily choose and initial k centroids $\mathcal{C} = \{c_1, c_2 \ldots c_k\}$
2. For each $i \in \{1, 2 \ldots k\}$, set the cluster C_i to be the set of points that are closer to c_i than they are to c_j for all $j \neq i$
3. For each $i \in \{1, 2 \ldots k\}$, set c_i to be the center of all points in C_i where $c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$
4. Repeat Step 2 and Step 3 until $\mathcal{C}$ no longer changes.

Max-Flow and Min-Cut Problems

A Flow network

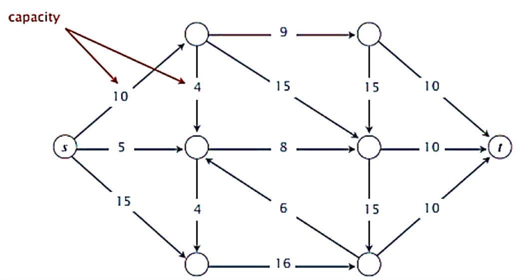
A flow network is a tuple $G = (V, E, s, t, c)$.

- Diagraph (V, E) with source $s \in V$ and sink $t \in V$.
- Capacity $c(e) > 0$ for each $e \in E$.

A flow network is a tuple $G = (V, E, s, t, c)$.

- Diagraph (V, E) with source $s \in V$ and sink $t \in V$.
- Capacity $c(e) > 0$ for each $e \in E$.

Intuition. Material flowing through a transportation network, which originates at source and is sent to sink.



Minimum-cut problem

Definition

An st-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Definition

An st-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Definition

Its capacity is the sum of the capacities of the edges from A to B.

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

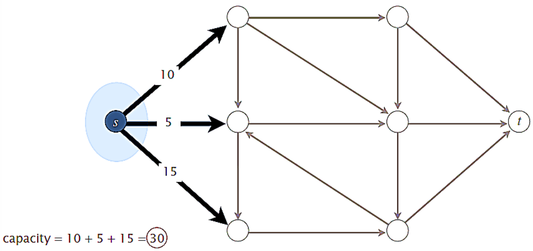
Definition

An st-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Definition

Its capacity is the sum of the capacities of the edges from A to B.

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



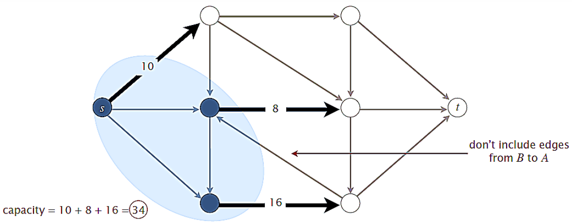
Definition

An st-cut (cut) is a partition (A, B) of the nodes with $s \in A$ and $t \in B$.

Definition

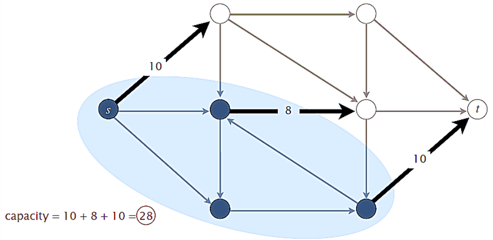
Its capacity is the sum of the capacities of the edges from A to B.

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



Minimum-cut problem

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



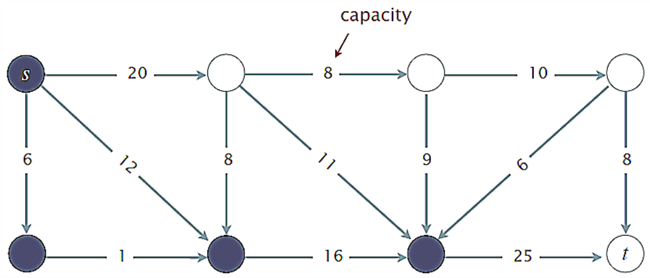
Which is the capacity of the given st-cut?

A. 11 $(20 + 25 - 8 - 11 - 9 - 6)$

B. 34 $(8 + 11 + 9 + 6)$

C. 45 $(20 + 25)$

D. 79 $(20 + 25 + 8 + 11 + 9 + 6)$

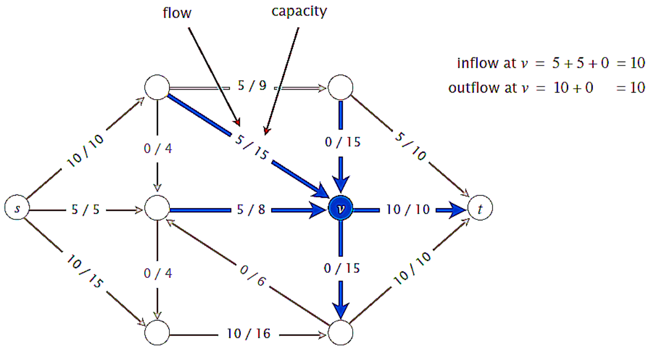


Maximum-flow problem

Definition

An st-flow(flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$



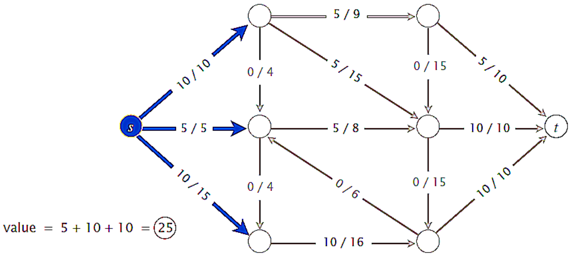
Definition

An st-flow(flow) f is a function that satisfies:

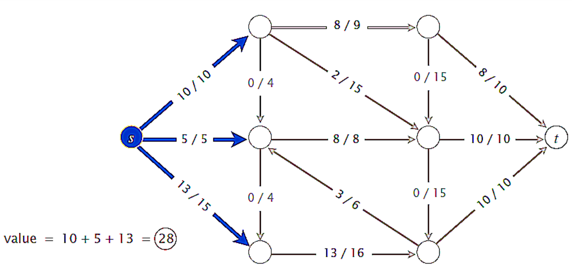
- For each $e \in E$: $0 \leq f(e) \leq c(e)$
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$

Definition

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



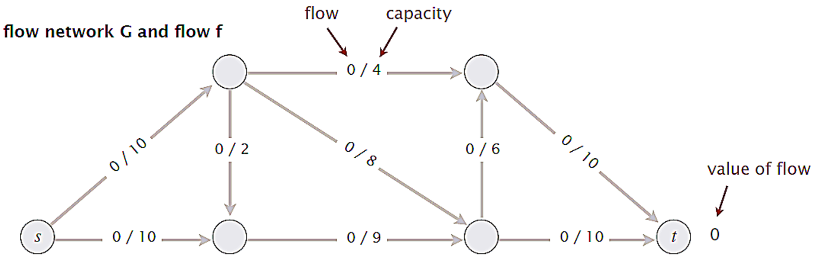
Max-flow problem. Find a flow of maximum value.



Ford-Fulkerson Algorithm

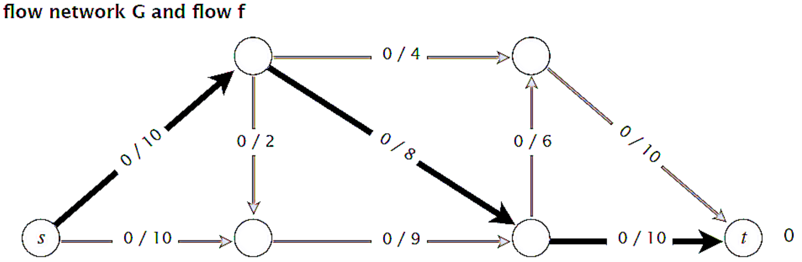
Greedy algorithm.

- Start with $f(e) = 0$ for each 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.



Greedy algorithm.

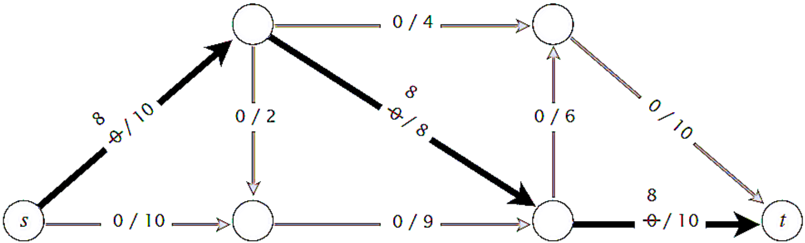
- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P.
- Repeat until you get stuck.



Toward a max-flow algorithm

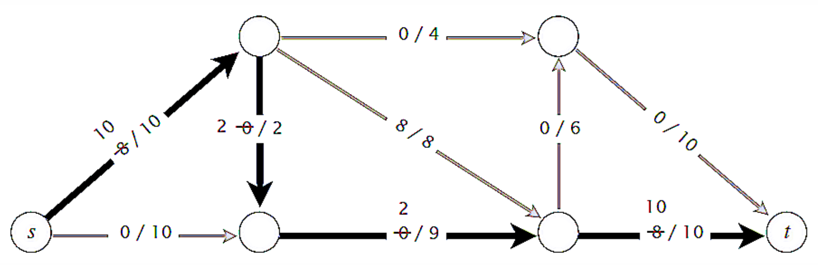
Greedy algorithm.

- Start with $f(e) = 0$ for each 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.



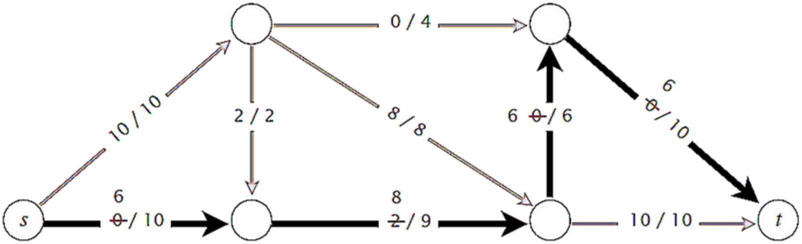
Greedy algorithm.

- Start with $f(e) = 0$ for each 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.



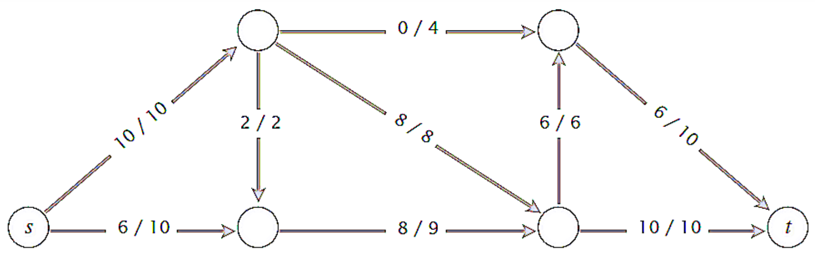
Greedy algorithm.

- Start with $f(e) = 0$ for each 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.



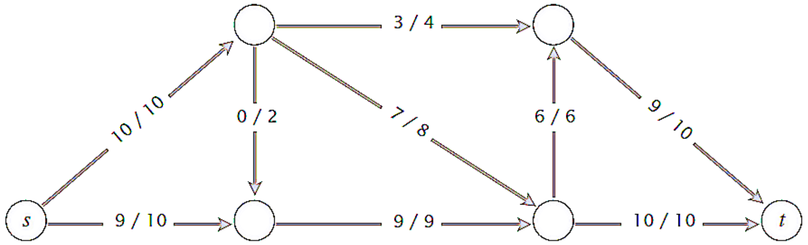
Greedy algorithm.

- Start with $f(e) = 0$ for each 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.



Greedy algorithm.

- Start with $f(e) = 0$ for each 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.



Why the greedy algorithm fails

Q. Why does the greedy algorithm fail?

Why the greedy algorithm fails

Q. Why does the greedy algorithm fail?

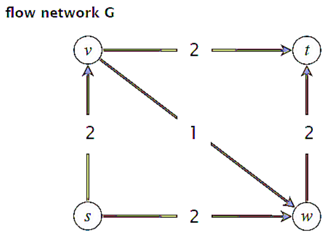
A. Once greedy algorithm increases flow on an edge, it never decreases it.

Q. Why does the greedy algorithm fail?

A. Once greedy algorithm increases flow on an edge, it never decreases it.

Ex. Consider flow network G.

- The unique max flow has $f^*(v, w) = 0$.
- Greedy algorithm could choose $s \to v \to w \to t$ as first augmenting path.



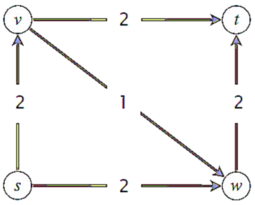
Q. Why does the greedy algorithm fail?

A. Once greedy algorithm increases flow on an edge, it never decreases it.

Ex. Consider flow network G.

- The unique max flow has $f^*(v, w) = 0$.
- Greedy algorithm could choose $s \to v \to w \to t$ as first augmenting path.

flow network G



Bottom line. Need some mechanism to undo a bad decision.

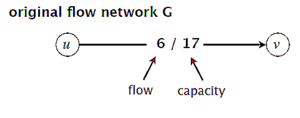
Residual network

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

- Flow $f(e)$.
- Capacity $c(e)$

Reverse edge $e^{reverse} = (v, u)$

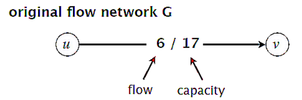
- Undo flow sent.



Residual network

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

- Flow $f(e)$.
- Capacity $c(e)$

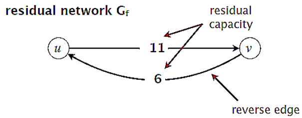


Reverse edge $e^{\text{reverse}} = (v, u)$

- Undo flow sent.

Residual capacity

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^{\text{reverse}} \in E \end{cases}$$



Residual network $G_f = (V, E_f, s, t, c_f)$

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

Definition

An augmenting path is a simple $s \rightsquigarrow t$ path in the residual network G_f.

Augmenting path

Definition

An augmenting path is a simple $s \rightsquigarrow t$ path in the residual network G_f.

Definition

The bottleneck capacity of an augmenting path P is the minimum residual capacity of any edge in P.

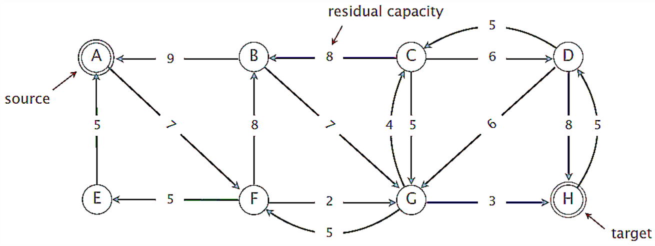
Key Property. Let f be a flow and let P be an augmenting path in G_f. After calling $f' \leftarrow \mathrm{Augment}(f, c, P)$, the resulting f' is a flow and $\mathrm{val}(f') = \mathrm{val}(f) + \mathrm{bottleneck}(G_f, P)$.

Key Property. Let f be a flow and let P be an augmenting path in G_f. After calling $f' \leftarrow \text{Augment}(f, c, P)$, the resulting f' is a flow and $\text{val}(f') = \text{val}(f) + \text{bottleneck}(G_f, P)$.

```
Augment(f,c,P)
δ ← bottleneck capacity of augmenting path P;
for each edge e ∈ P do
  | if (e ∈ E) then f(e) ← f(e) + δ;
  | else
  |   | f(e^reverse) ← f(e^reverse) − δ
  | end
end
Return f;
```

Which is the augmenting path of highest bottleneck capacity?

1. A → F → G → H
2. A → B → C → D → H
3. A → F → B → G → H
4. A → F → B → G → C → D → H



Ford–Fulkerson algorithm

Ford–Fulkerson augmenting path algorithm.

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

Ford–Fulkerson algorithm

Ford–Fulkerson augmenting path algorithm.

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

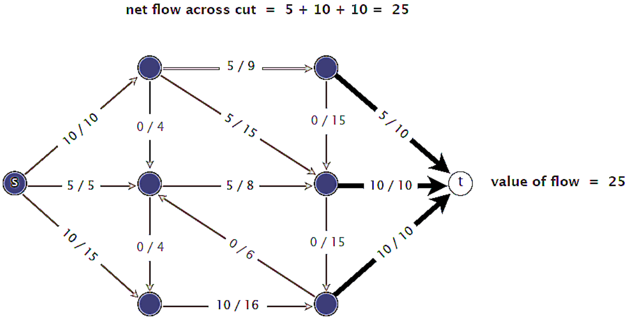
```
Ford–Fulkerson(G)

for each edge e ∈ E do
|  f(e) ← 0
end
G_f ← residual network of G with respect to flow f;
while there exists an s ⇝ t path P in G_f do
|  f ← Augment(f,c,P);
|  Update(G_f);
end
Return f;
```

Max-Flow Min-Cut Theorem

Lemma

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

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

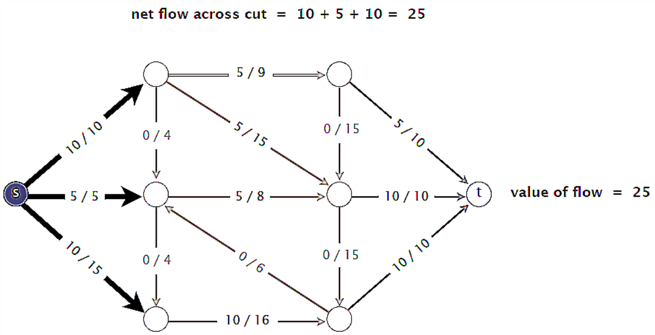


Relationship between flows and cuts

Lemma

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

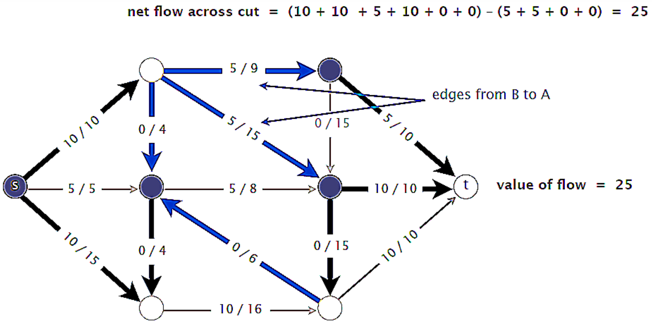
$$\mathrm{val}(f) = \sum_{\text{out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$



Lemma

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

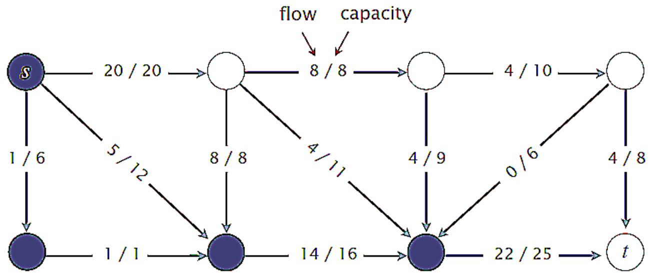
$$\mathrm{val}(f) = \sum_{\text{out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$



Network flow: quiz 3

Which is the net flow across the given cut?

1. 11 $(20 + 25 - 8 - 11 - 9 - 6)$
2. 26 $(20 + 22 - 8 - 4 - 4)$
3. 42 $(20 + 22)$
4. 45 $(20 + 25)$



Lemma

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

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

Lemma

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

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

Proof.

$$\begin{aligned}
\text{val}(f) &= \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e) \\
&= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right) \\
&= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e).
\end{aligned}$$

Theorem

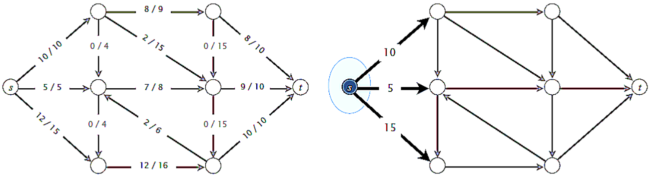
Weak Duality Let f be any flow and (A, B) be any cut. Then, $\mathrm{val}(f) \leq \mathsf{cap}(A, B)$.

Theorem

Weak Duality Let f be any flow and (A, B) be any cut. Then, $\mathsf{val}(f) \leq \mathsf{cap}(A, B)$.

Proof.

$$
\begin{aligned}
\mathsf{val}(f) &= \sum_{\text{e out of A}} f(e) - \sum_{\text{e in to A}} f(e) \\
&\leq \sum_{\text{e out of A}} f(e) \\
&\leq \sum_{\text{e out of A}} c(e) \\
&= \mathsf{cap}(A, B)
\end{aligned}
$$



Corollary

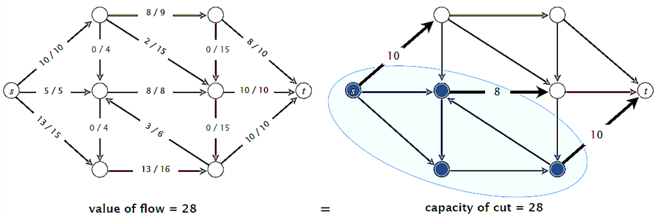
Let f be a flow and let (A, B) be any cut. If $\mathrm{val}(f) = \mathsf{cap}(A, B)$, then f is a max flow and (A, B) is a min cut.

Corollary

Let f be a flow and let (A, B) be any cut. If $\mathrm{val}(f) = \mathsf{cap}(A, B)$, then f is a max flow and (A, B) is a min cut.

Proof.

- For any flow f' : $\mathrm{val}\,(f') \leq \mathsf{cap}(A, B) = \mathrm{val}(f)$.
- For any cut (A', B') : $\mathsf{cap}\,(A', B') \geq \mathrm{val}(f) = \mathsf{cap}(A, B)$



Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

MAXIMAL FLOW THROUGH A NETWORK

L. R. FORD, JR. AND D. R. FULKERSON

Introduction. The problem discussed in this paper was formulated by T. Harris as follows:

"Consider a rail network connecting two cities by way of a number of intermediate cities, where each link of the network has a number assigned to it representing its capacity. Assuming a steady state condition, find a maximal flow from one given city to the other."

ON THE MAX FLOW MIN CUT THEOREM OF NETWORKS

G. B. Dantzig

D. R. Fulkerson

P-826

April 15, 1955

A Note on the Maximum Flow Through a Network*

P. ELIAS†, A. FEINSTEIN‡, AND C. E. SHANNON§

Summary—This note discusses the problem of maximizing the rate of flow from one terminal to another, through a network which consists of a number of branches, each of which has a limited capacity. The main result is a theorem: The maximum possible flow from left to right through a network is equal to the minimum value among all simple cut-sets. This theorem is applied to solve a more general problem, in which a number of input nodes and a number of output nodes are used.

from one terminal to the other in the original network passes through at least one branch in the cut-set. In the network above, some examples of cut-sets are (d, e, f), and (b, c, e, g, h), (d, g, h, i). By a *simple cut-set* we will mean a cut-set such that if any branch is omitted it is no longer a cut-set. Thus (d, e, f) and (b, c, e, g, h) are simple

Max-flow min-cut theorem

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Max-flow min-cut theorem

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Proof. The following three conditions are equivalent for any flow f:

Max-flow min-cut theorem

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Proof. The following three conditions are equivalent for any flow f:

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

ii. f is a max flow.

iii. There is no augmenting path with respect to f.

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Proof. The following three conditions are equivalent for any flow f:

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

ii. f is a max flow.

iii. There is no augmenting path with respect to f.

[i$\Rightarrow$ ii]

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Proof. The following three conditions are equivalent for any flow f:

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

ii. f is a max flow.

iii. There is no augmenting path with respect to f.

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

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Proof. The following three conditions are equivalent for any flow f:

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

ii. f is a max flow.

iii. There is no augmenting path with respect to f.

[ii$\Rightarrow$ iii]

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Proof. The following three conditions are equivalent for any flow f:

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

ii. f is a max flow.

iii. There is no augmenting path with respect to f.

[ii$\Rightarrow$ iii] We prove contrapositive: $\neg$ iii $\Rightarrow$ $\neg$ii.

Max-Flow Min-Cut Theorem

Value of a max flow = capacity of a min cut.

Augmenting path theorem

A flow f is a max flow iff no augmenting paths.

Proof. The following three conditions are equivalent for any flow f:

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

ii. f is a max flow.

iii. There is no augmenting path with respect to f.

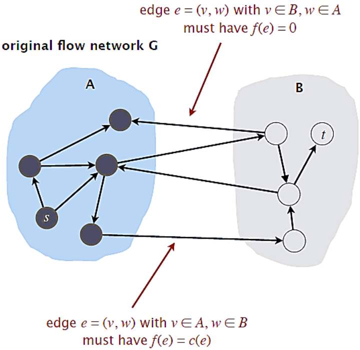
[ii$\Rightarrow$ iii] We prove contrapositive: $\neg$ iii $\Rightarrow$ $\neg$ii.

- Suppose that there is an augmenting path with respect to f.
- Can improve flow f by sending flow along this path.
- Thus, f is not a max flow.

[iii $\Rightarrow$ i]

- Let f be a flow with no augmenting paths.
- Let A be set of nodes reachable from s in residual network $\mathrm{G_f}$.
- By definition of $\mathrm{A} : \mathrm{s} \in \mathrm{A}$.
- By definition of flow $\mathrm{f} : \mathrm{t} \notin \mathrm{A}$.

$$\begin{aligned}\mathrm{val(f)} &= \sum_{\text{e out of A}} \mathrm{f(e)} - \sum_{\text{e in to A}} \mathrm{f(e)} \\ &= \sum_{\text{e out of A}} \mathrm{c(e)} - 0 \\ &= \mathsf{cap}(\mathrm{A}, \mathrm{B})\end{aligned}$$



Capacity-Scaling Algorithm

Assumption. Every edge capacity $c(e)$ is an integer between 1 and C.

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Assumption. Every edge capacity $c(e)$ is an integer between 1 and C.

Integrality invariant. Throughout Ford-Fulkerson, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Assumption. Every edge capacity $c(e)$ is an integer between 1 and C.

Integrality invariant. Throughout Ford-Fulkerson, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

Proof. By induction on the number of augmenting paths.

Assumption. Every edge capacity $c(e)$ is an integer between 1 and C.

Integrality invariant. Throughout Ford-Fulkerson, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

Proof. By induction on the number of augmenting paths.

Theorem

Ford–Fulkerson terminates after at most $\text{val}(f^*) \leq nC$ augmenting paths, where f^* is a max flow.

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Assumption. Every edge capacity $c(e)$ is an integer between 1 and C.

Integrality invariant. Throughout Ford-Fulkerson, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

Proof. By induction on the number of augmenting paths.

Theorem

Ford–Fulkerson terminates after at most $\mathrm{val}(f^*) \leq nC$ augmenting paths, where f^* is a max flow.

Proof. Each augmentation increases the value of the flow by at least 1.

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Corollary

The running time of Ford–Fulkerson is $O(mnC)$.

Corollary

The running time of Ford–Fulkerson is $O(mnC)$.

Proof. Can use either BFS or DFS to find an augmenting path in $O(m)$ time.

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Corollary

The running time of Ford–Fulkerson is $O(mnC)$.

Proof. Can use either BFS or DFS to find an augmenting path in $O(m)$ time.

Integrality Theorem

There exists an integral max flow f^*

Analysis of Ford–Fulkerson algorithm (when capacities are integral)

Corollary

The running time of Ford–Fulkerson is $O(mnC)$.

Proof. Can use either BFS or DFS to find an augmenting path in $O(m)$ time.

Integrality Theorem

There exists an integral max flow f^*

Proof. Since Ford–Fulkerson terminates, theorem follows from integrality invariant.

Q. Is generic Ford–Fulkerson algorithm poly-time in input size?

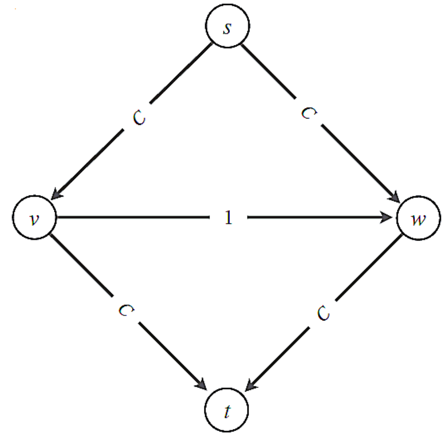
Q. Is generic Ford–Fulkerson algorithm poly-time in input size?

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

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 $\rightarrow$ t
- s $\rightarrow$ w $\rightarrow$ v $\rightarrow$ t
- s $\rightarrow$ v $\rightarrow$ w $\rightarrow$ t
- s $\rightarrow$ w $\rightarrow$ v $\rightarrow$ t
- ...
- s $\rightarrow$ v $\rightarrow$ w $\rightarrow$ t
- s $\rightarrow$ w $\rightarrow$ v $\rightarrow$ t



Network flow: quiz 4

The Ford–Fulkerson algorithm is guaranteed to terminate if the edge capacities are . . .

A. Rational numbers.

B. Real numbers.

C. Both A and B.

D. Neither A nor B.

Choosing good augmenting paths

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

Choosing good augmenting paths

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

Pathology. When edge capacities can be irrational, no guarantee that Ford–Fulkerson terminates (or converges to a maximum flow)!

Choosing good augmenting paths

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

Pathology. When edge capacities can be irrational, no guarantee that Ford–Fulkerson terminates (or converges to a maximum flow)!

Goal. Choose augmenting paths so that:

- Can find augmenting paths efficiently.
- Few iterations.

Choose augmenting paths with:

Choose augmenting paths with:

- Max bottleneck capacity("fattest").

Choosing good augmenting paths

Choose augmenting paths with:

- Max bottleneck capacity("fattest").
- Sufficiently large bottleneck capacity.

Choose augmenting paths with:

- Max bottleneck capacity("fattest").
- Sufficiently large bottleneck capacity.
- Fewest edges.

Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems

JACK EDMONDS

University of Waterloo, Waterloo, Ontario, Canada

AND

RICHARD M. KARP

University of California, Berkeley, California

ABSTRACT. This paper presents new algorithms for the maximum flow problem, the Hitchcock transportation problem, and the general minimum-cost flow problem. Upper bounds on the numbers of steps in these algorithms are derived, and are shown to compare favorably with upper bounds on the numbers of steps required by earlier algorithms.

Edmonds-Karp 1972 (USA)

Dokl. Akad. Nauk SSSR
Tom 194 (1970), No. 4

Soviet Math. Dokl.
Vol. 11 (1970), No. 5

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

UDC 518.5

E. A. DINIC

Different variants of the formulation of the problem of maximal stationary flow in a network and its many applications are given in [1]. There also is given an algorithm solving the problem in the case where the initial data are integers (or, what is equivalent, commensurable). In the general case this algorithm requires preliminary rounding off of the initial data, i.e. only an approximate solution of the problem is possible. In this connection the rapidity of convergence of the algorithm is inversely proportional to the relative precision.

Dinitz 1970 (Soviet Union)

Capacity-scaling algorithm

Overview. Choosing augmented paths with large bottleneck capacity.

Capacity-scaling algorithm

Overview. Choosing augmented paths with large bottleneck capacity.

- Maintain scaling parameter Δ.

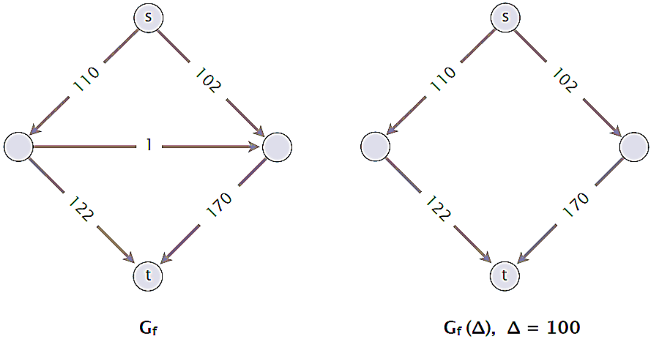
Capacity-scaling algorithm

Overview. Choosing augmented paths with large bottleneck capacity.

- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the part of the residual network containing only those edges with capacity $\geq \Delta$.

Overview. Choosing augmented paths with large bottleneck capacity.

- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the part of the residual network containing only those edges with capacity $\geq \Delta$.
- Any augmenting path in $G_f(\Delta)$ has bottleneck capacity $\geq \Delta$.



```
Capacity-Scaling(G)
for each edge e ∈ E do
|   f(e) ← 0
end
Δ ← largest power of 2 ≤ C;
while Δ ≥ 1 do
|   G_f(Δ) ← Δ-residual network of G with respect to flow f;
|   while there exists an s ⇝ t path P in G_f(Δ) do
|   |   f ← Augment(f, c, P);
|   |   Update(G_Δ(f));
|   end
|   Δ = Δ/2;
end
Return f;
```

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

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

Invariant. The scaling parameter Δ is a power of 2.

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

Invariant. The scaling parameter Δ is a power of 2.

Proof. Initially a power of 2; each phase divides Δ by exactly 2.

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

Invariant. The scaling parameter Δ is a power of 2.

Proof. Initially a power of 2; each phase divides Δ by exactly 2.

Integrality invariant. Throughout the algorithm, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

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

Invariant. The scaling parameter Δ is a power of 2.

Proof. Initially a power of 2; each phase divides Δ by exactly 2.

Integrality invariant. Throughout the algorithm, every edge flow $f(e)$ and residual capacity $c_f(e)$ is an integer.

Proof. Same as for generic Ford–Fulkerson.

Capacity-scaling algorithm: proof of correctness

Theorem

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

Theorem

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

Proof.

Capacity-scaling algorithm: proof of correctness

Theorem

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

Proof.

- By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths.
- Result follows augmenting path theorem.

Capacity-scaling algorithm: analysis of running time

Lemma 1

There are $1 + \lfloor \log_2 C \rfloor$ scaling phases.

Lemma 2

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

Lemma 3

Let f be the flow at the end of a Δ-scaling phase.
Then, the max-flow value $\leq \mathsf{val}(f) + m\Delta$.

Capacity-scaling algorithm: analysis of running time

Theorem

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

Capacity-scaling algorithm: analysis of running time

Theorem

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

Proof.

Capacity-scaling algorithm: analysis of running time

Theorem

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

Proof.

- Lemma 1+ Lemma 2 $\Rightarrow$ $O(m \log C)$ augmentations.
- Finding an augmenting path takes $O(m)$ time.

Lemma 1

There are $1 + \lfloor \log_2 C \rfloor$ scaling phases.

Capacity-scaling algorithm: analysis of running time

Lemma 1

There are $1 + \lfloor \log_2 C \rfloor$ scaling phases.

Proof. Initially $C/2 < \Delta \leq C$; Δ decreases by a factor of 2 in each iteration.

Capacity-scaling algorithm: analysis of running time

Lemma 2

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

Lemma 2

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

Proof.

- Let f be the flow at the beginning of a Δ-scaling phase.
- Lemma 3 $\Rightarrow$ max-flow value $\leq val(f) + m(2\Delta)$.
- Each augmentation in a Δ-phase increases $val(f)$ by at least Δ.

Capacity-scaling algorithm: analysis of running time

Lemma 3

Let f be the flow at the end of a $\mathit{\Delta}$-scaling phase.

Then, the max-flow value $\leq \mathsf{val}(f) + m\mathit{\Delta}$.

Lemma 3

Let f be the flow at the end of a Δ-scaling phase.
Then, the max-flow value $\leq \mathsf{val}(f) + m\Delta$.

Proof.

Lemma 3

Let f be the flow at the end of a Δ-scaling phase.
Then, the max-flow value $\leq \mathsf{val}(f) + m\Delta$.

Proof.

- We show there exists a cut (A, B) such that $\mathrm{cap}(A, B) \leq \mathrm{val}(f) + m\Delta$.
- Choose A to be the set of nodes reachable from s in $G_f(\Delta)$.
- By definition of $A : s \in A$.
- By definition of flow $f : t \notin A$.

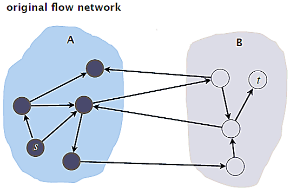
Lemma 3

Let f be the flow at the end of a Δ-scaling phase.
Then, the max-flow value $\leq \mathsf{val}(f) + m\Delta$.

Proof.

- We show there exists a cut (A, B) such that $\mathsf{cap}(A, B) \leq \mathsf{val}(f) + m\Delta$.
- Choose A to be the set of nodes reachable from s in $G_f(\Delta)$.
- By definition of $A : s \in A$.
- By definition of flow $f : t \notin A$.

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



Shortest Augmenting Paths

Shortest augmenting path

Q. How to choose next augmenting path in Ford–Fulkerson?

Shortest augmenting path

Q. How to choose next augmenting path in Ford–Fulkerson?

A. Pick one that uses the fewest edges.

Q. How to choose next augmenting path in Ford–Fulkerson?

A. Pick one that uses the fewest edges.

```
Shortest-Augmenting-Path(G)

for each edge e ∈ E do
 |  f(e) ← 0
end
G_f ← residual network of G with respect to flow f;
while there exists an s ⇝ t path in G_f do
 |  P ← BFS((G_f));
 |  f ← Augment(f, c, P);
 |  Update(G_f);
end
Return f;
```

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem

The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem

The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

Proof.

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem

The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

Proof.

- $O(m)$ time to find a shortest augmenting path via BFS.

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem

The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

Proof.

- $O(m)$ time to find a shortest augmenting path via BFS.
- There are $\leq mn$ augmentations

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem

The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

Proof.

- $O(m)$ time to find a shortest augmenting path via BFS.
- There are $\leq mn$ augmentations
  - at most m augmenting paths of length k

Shortest augmenting path: overview of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem

The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

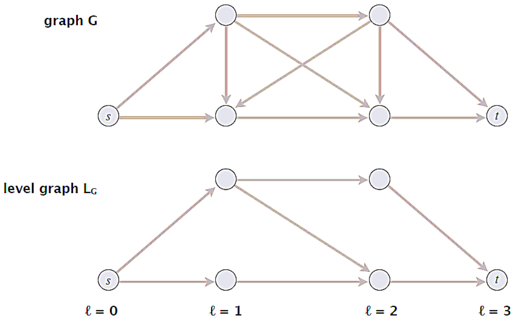
Proof.

- $O(m)$ time to find a shortest augmenting path via BFS.
- There are $\leq mn$ augmentations
  - at most m augmenting paths of length k $\longleftarrow$ Lemma 1 + Lemma 2
  - at most $n - 1$ different lengths

Definition

Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell(v)$ = number of edges in shortest $s \rightsquigarrow v$ path.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v, w) \in E$ with $\ell(w) = \ell(v) + 1$.



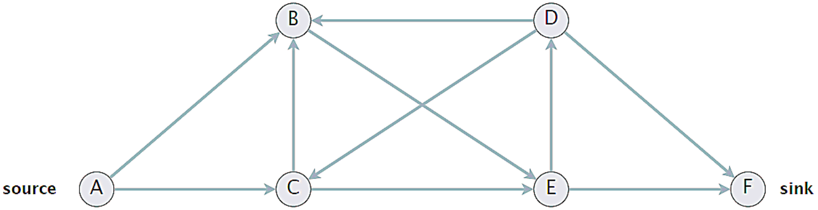
Which edges are in the level graph of the following digraph?

A. D → F

B. E → F

C. Both A and B.

D. Neither A nor B.

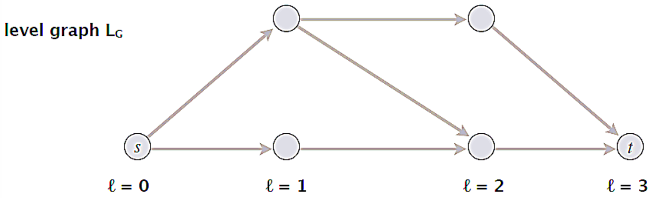


Definition

Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell(v)$ = number of edges in shortest $s \rightsquigarrow v$ path.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v, w) \in E$ with $\ell(w) = \ell(v) + 1$.

Key property. P is a shortest $s \rightsquigarrow v$ path in G iff P is an $s \rightsquigarrow v$ path in L_G.



Shortest augmenting path: analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Shortest augmenting path: analysis

Lemma 1

The length of a shortest augmenting path never decreases.

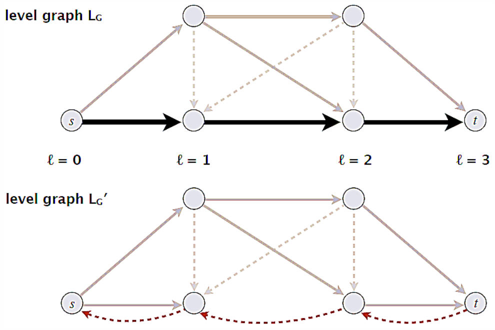
Proof.

Lemma 1

The length of a shortest augmenting path never decreases.

Proof.

- Let f and f' be flow before and after a shortest-path augmentation.
- Let L_G and $L_{G'}$ be level graphs of G_f and $G_{f'}$.
- Only back edges added to $G_{f'}$
  (any $s \rightsquigarrow t$ path that uses a back edge is longer than previous length)



Shortest augmenting path: analysis

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Shortest augmenting path: analysis

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Proof.

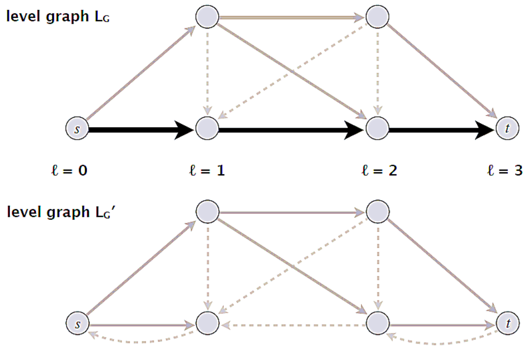
Shortest augmenting path: analysis

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Proof.

- At least one (bottleneck) edge is deleted from L_G per augmentation.
- No new edge added to L_G until shortest path length strictly increases.



Shortest augmenting path: review of analysis

Lemma 1

The length of a shortest augmenting path never decreases.

Lemma 2

After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

Theorem

The shortest-augmenting-path algorithm takes $O(m^2 n)$ time.

Note. $\Theta(mn)$ augmentations necessary for some flow networks.

Note. $\Theta(mn)$ augmentations necessary for some flow networks.

- Try to decrease time per augmentation instead.
- Simple idea $\Rightarrow$ $O(mn^2)$ [Dinitz 1970]

Shortest augmenting path: improving the running time

Note. $\Theta(mn)$ augmentations necessary for some flow networks.

- Try to decrease time per augmentation instead.
- Simple idea $\Rightarrow$ $O(mn^2)$ [Dinitz 1970]
- Dynamic trees $\Rightarrow$ $O(mn \log n)$ [Sleator–Tarjan 1983]

A Data Structure for Dynamic Trees

DANIEL D. SLEATOR AND ROBERT ENDRE TARJAN

Bell Laboratories, Murray Hill, New Jersey 07974

Received May 8, 1982; revised October 18, 1982

A data structure is proposed to maintain a collection of vertex-disjoint trees under a sequence of two kinds of operations: a *link* operation that combines two trees into one by adding an edge, and a *cut* operation that divides one tree into two by deleting an edge. Each operation requires $O(\log n)$ time. Using this data structure, new fast algorithms are obtained for the following problems:

(1) Computing nearest common ancestors.

(2) Solving various network flow problems including finding maximum flows, blocking flows, and acyclic flows.

(3) Computing certain kinds of constrained minimum spanning trees.

(4) Implementing the network simplex algorithm for minimum-cost flows.

The most significant application is (2); an $O(mn \log n)$-time algorithm is obtained to find a maximum flow in a network of n vertices and m edges, beating by a factor of $\log n$ the fastest algorithm previously known for sparse graphs.

Dinitz' Algorithm

Two types of augmentations.

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

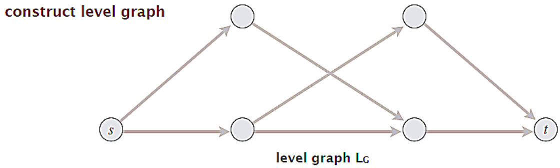
Dinitz' algorithm

Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

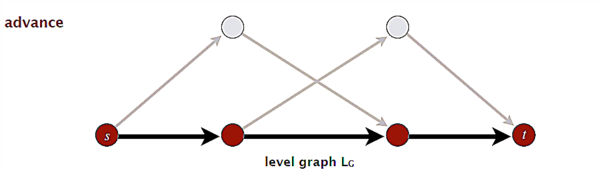


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

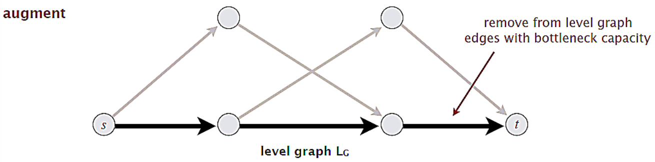


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

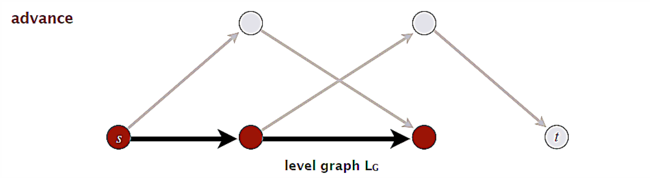


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

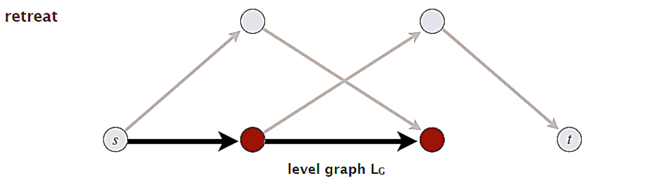


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

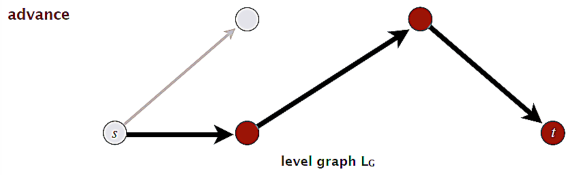


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

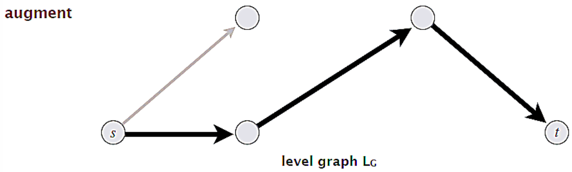


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

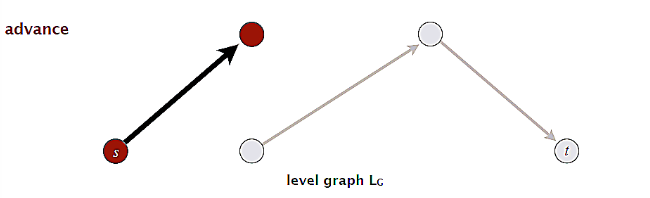


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

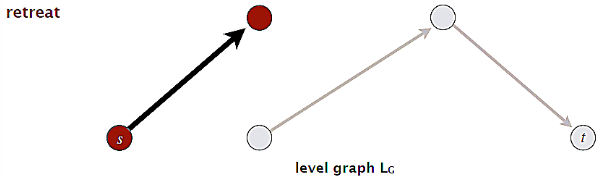


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

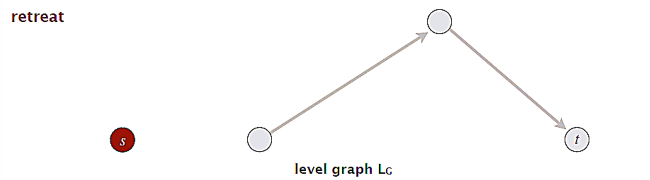


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.

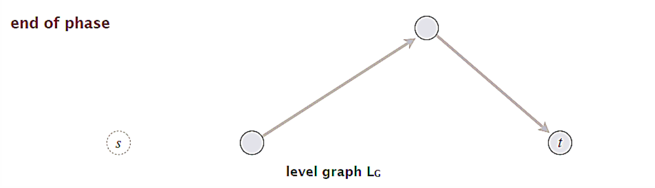


Two types of augmentations.

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

Phase of normal augmentations.

- Construct level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment flow; update L_G; and restart from s.
- If get stuck, delete node from L_G and retreat to previous node.



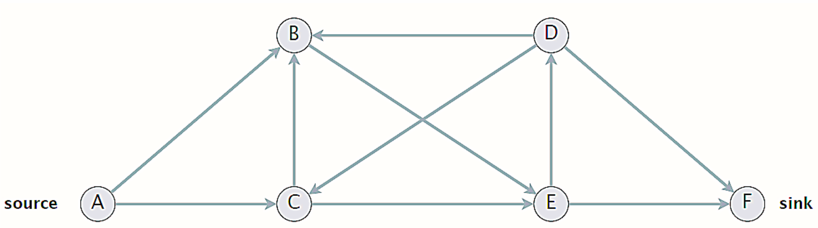
Dinitz' algorithm (as refined by Even and Itai)

```
Initialize(G, f)

L_G ← level-graph of G_f
P ← ∅
goto Advance(s);


Retreat(v)

if v = s then Stop;
else
    Delete v from L_G;
    Remove last edge (u, v)
     from P;
end
goto Advance(u);
```

```
Advance(v)

if v = t then
    Augment(P);
    Remove saturated edges
     from L_G;
    P ← ∅;
    goto Advance(s);
end
if there exists edge (v, w) ∈ L_G
 then
    Add edge (v, w) to P;
    goto Advance(w);
end
else
    goto Retreat(v);
end
```

How to compute the level graph L_G efficiently?

1. Depth-first search.
2. Breadth-first search.
3. Both A and B.
4. Neither A nor B.



Dinitz' algorithm: analysis

Lemma

A phase can be implemented to run in $O(mn)$ time.

Dinitz' algorithm: analysis

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase.

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase.

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase. $\longleftarrow$ $O(mn)$ per phase

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase. $\longleftarrow$ $O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase. $\longleftarrow$ $O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase.

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase. $\longleftarrow O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase. $\longleftarrow O(m+n)$ per phase

Dinitz' algorithm: analysis

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase. $\longleftarrow$ $O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase. $\longleftarrow$ $O(m + n)$ per phase
  (because a retreat deletes one node from L_G)

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase. $\longleftarrow O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase. $\longleftarrow O(m+n)$ per phase
  (because a retreat deletes one node from L_G)
- At most mn advances per phase.

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

- Initialization happens once per phase. using BFS
- At most m augmentations per phase. $\longleftarrow$ $O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase. $\longleftarrow$ $O(m+n)$ per phase
  (because a retreat deletes one node from L_G)
- At most mn advances per phase. $\longleftarrow$ $O(mn)$ per phase

Lemma

A phase can be implemented to run in $O(mn)$ time.

Proof.

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

Theorem (Dinitz 1970)

Dinitz' algorithm runs in $O(mn^2)$ time.

Dinitz' algorithm: analysis

Theorem (Dinitz 1970)

Dinitz' algorithm runs in $O(mn^2)$ time.

Proof.

Theorem (Dinitz 1970)

Dinitz' algorithm runs in $O(mn^2)$ time.

Proof.

- By Lemma, $O(mn)$ time per phase.

Theorem (Dinitz 1970)

Dinitz' algorithm runs in $O(mn^2)$ time.

Proof.

- By Lemma, $O(mn)$ time per phase.
- At most $n-1$ phases

Dinitz' algorithm: analysis

Theorem (Dinitz 1970)

Dinitz' algorithm runs in $O(mn^2)$ time.

Proof.

- By Lemma, $O(mn)$ time per phase.
- At most $n - 1$ phases (as in shortest-augmenting-path analysis).

Augmenting-path algorithms: summary

year	method	# augmentations	running time
1955	augmenting path	nC	$O(mnC)$
1972	fattest path	$m \log(mC)$	$O\left(m^2 \log n \log(mC)\right)$
1972	capacity scaling	$m \log C$	$O\left(m^2 \log C\right)$
1985	improved capacity scaling	$m \log C$	$O(mn \log C)$
1970	shortest augmenting path	mn	$O\left(m^2 n\right)$
1970	level graph	mn	$O\left(mn^2\right)$
1983	dynamic trees	mn	$O(mn \log n)$

augmenting-path algorithms with m edges, n nodes, and integer capacities between 1 and C

Maximum-flow algorithms: theory highlights

year	method	worst case	discovered by
1951	simplex	$O\left(mn^2C\right)$	Dantzig
1955	augmenting paths	$O(mnC)$	Ford–Fulkerson
1970	shortest augmenting paths	$O\left(mn^2\right)$	Edmonds–Karp, Dinitz
1974	blocking flows	$O\left(n^3\right)$	Karzanov
1983	dynamic trees	$O(mn \log n)$	Sleator–Tarjan
1985	improved capacity scaling	$O(mn \log C)$	Gabow
1988	push–relabel	$O\left(mn \log\left(n^2/m\right)\right)$	Goldberg–Tarjan
1998	binary blocking flows	$O\left(m^{3/2} \log\left(n^2/m\right) \log C\right)$	Goldberg–Rao
2013	compact networks	$O(mn)$	Orlin
2014	interior-point methods	$\tilde{O}\left(mm^{1/2} \log C\right)$	Lee–Sidford
2016	electrical flows	$\tilde{O}\left(m^{10/7}C^{1/7}\right)$	Madry
20xx		???	

augmenting-path algorithms with m edges, n nodes, and integer capacities between 1 and C