Analysis of Algorithms

V. Adamchik CSCI 570 Fall 2016

Lecture 9 University of Southern California

The MAX Flow Problem

Based on Sections 7.1 & 7.2 & 7.3

Algorithm Design by Kleinberg & Tardos

The Network Flow Problem

Our fourth major algorithm design technique (greedy, divide-and-conquer, and dynamic programming are the others).

Plan:

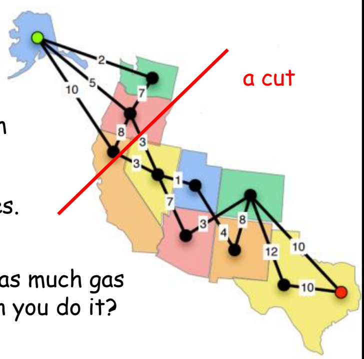
The Ford-Fulkerson algorithm

Application to Bipartite Matching

The Flow Problem

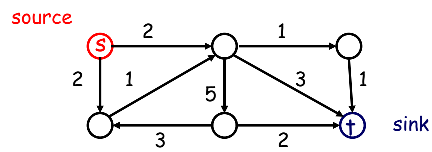
Suppose you want to ship natural gas from Alaska to Texas.

Pipes have capacities.



The goal is to send as much gas as possible. How can you do it?

The Flow Problem

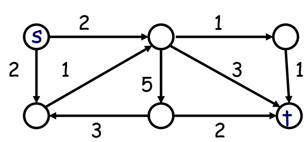


Given a connected, directed, weighted graph G=(V,E) with two distinguishes vertices, s and t. (no edges to s and no edges out of t)

Edge capacities c(e) are integers and c(e) ≥ 0.

If e∉E, then c(e) = 0

The Flow Problem



We will call f(e) a flow through e such that

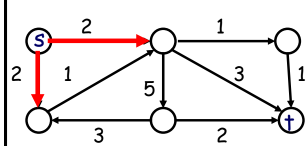
1) 0 ≤ f(e) ≤ c(e) (capacity law)

2) for each v∈V-{s, t}:

flow-in = flow-out

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

The Flow Problem



The value of a flow f is defined by

The flow that s is able to send.

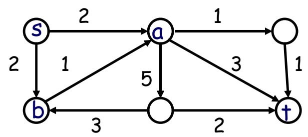
$$|f| = \sum_{e \text{ out of } s} f(e)$$

Max-flow problem: given a flow network G, find a flow f from s to t of the maximum possible value.

The max-flow here is 3.

How can you see that the above flow is really max?

The MAX Flow Problem

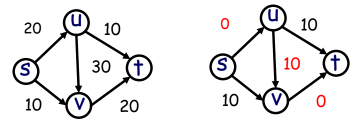


The max-flow here is 3. How can you see that the flow is really max?

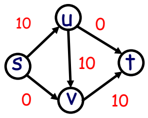
The flow saturates s-a and b-a. If we remove them, the graph becomes disconnected.

Find Maxflow: Greedy Algorithm

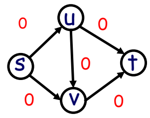
Push 20 via s-u-v-t



Not optimal…



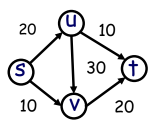
Push 10 via s-u-t and push 10 via s-v-t

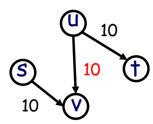


We can push 10 via s-u-v-t

Canceling Flow

Push 20 via s-u-v-t

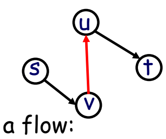




Not optimal…

How could we fix it and get 30?

We should allow 10 units flow from v to u by canceling an existing flow.



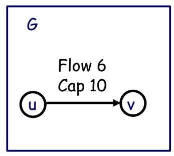
Thus there are two ways to increase a flow:

a) find unused capacity

b) find cancelable flow

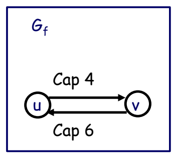
Residual Graph G_f

G_f contains the same vertices as G.



Forward edges:

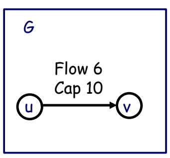
$\forall e \in E$, $f(e) < c(e)$ include e in G_f with capacity $c_f(e)=c(e)-f(e)$



Backward edges:

$\forall e \in E$, $f(e) > 0$ include $e^R = (v,u)$ in G_f with capacity $c_f(e^R)=f(e)$

Residual Graph $G_f(V, E_f)$

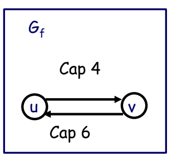


$e=(u,v)$

$c_f(e)=c(e)-f(e)=4$

$e^R=(v,u)$

$c_f(e^R)=f(e)=6$



Network: $G = (V, E)$ and flow f.

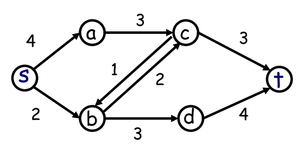
Residual capacity: $c_f(e) = c(e) - f(e)$.

Residual graph: $G_f(V, E_f)$, where

$E_f = \{e \mid f(e) < c(e)\} \cup \{e^R \mid f(e) > 0\}$

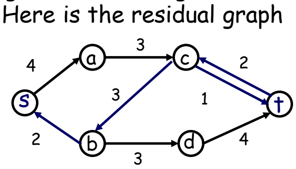
Example: residual graph

G



Push 2 along s-b-c-t and augment the flow along that path. Here is the residual graph

G_f



Augmenting Path = Path in G_f

Let P be an s–t path in the residual graph G_f.

Let bottleneck(P, f) be the smallest capacity in G_f on any edge of P.

If bottleneck(P, f) > 0 then we can increase the flow by sending bottleneck(P, f) along the path P.

```
augment(f, P):
b = bottleneck(P,f)
For each e = (u,v) ∈ P:
   if e = (u,v) is a forward edge:
        increase f(e) by b //add some flow
   else:
        decrease f(e^R) in G by b //erase some flow
```

The Ford-Fulkerson Algorithm

Algorithm. Given (G, s, t, c)

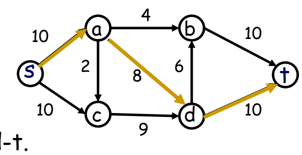
1) Start with |f|=0, so f(e)=0
2) Find an augmenting path in G_f
3) Augment flow along this path
4) Repeat until there is no an s-t path in G_f

How do we find that path?

by running DFS.

Example

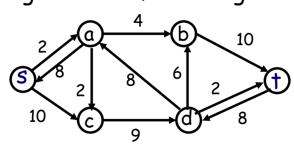
We start with |f| = 0, so $G = G_f$.



Path s-a-d-t.

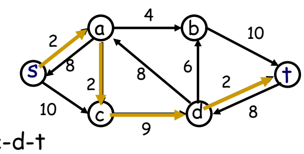
Push 8 and augment the flow along that path

G_f



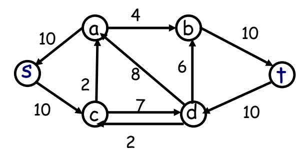
Flow=8

Example



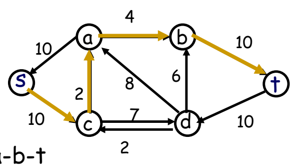
Path s-a-c-d-t

Push 2 and augment the flow along that path



Flow=8+2=10

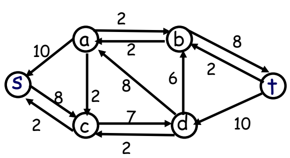
Example



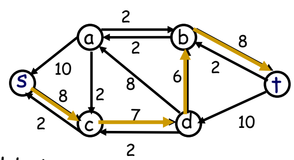
Path s-c-a-b-t

Push 2

Flow=10+2=12



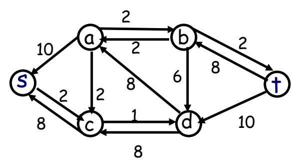
Example



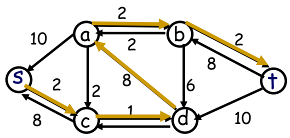
Path s-c-d-b-t

Push 6

Flow=12+6=18



Example

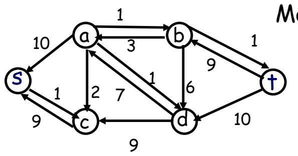


Path s-c-d-a-b-t

Push 1

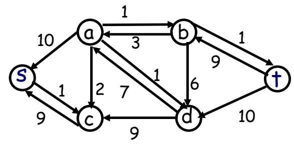
Flow=18+1=19

Maxflow is 19.



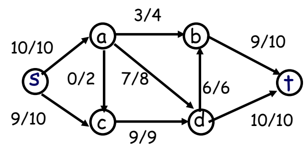
Notations

residual graph G_f



In graph G, we will label edges with flow/cap notation

graph G



The Ford-Fulkerson Algorithm

Algorithm. Given (G, s, t, c)

1) Start with |f|=0, so f(e)=0
2) Find an augmenting path in G_f
3) Augment flow along this path
4) Repeat until there is no an s-t path in G_f

We need to show that after augmenting we still have a flow.

Lemma 1

Let f be a flow in G.
Then, f' = augment(f, P) in G_f is a flow.

Proof.

if e is forward edge,

$f'(e) = f(e) + \text{bottleneck}(P,f) \leq f(e)+c(e)-f(e) \leq c(e)$

if e is backward edge,

$f'(e) = f(e) - \text{bottleneck}(P,f) \geq f(e)-f(e) = 0$

The Ford-Fulkerson Algorithm

Algorithm. Given (G, s, t, c)

1) Start with |f|=0, so f(e)=0
2) Find an augmenting path in G_f
3) Augment flow along this path
4) Repeat until there is no an s-t path in G_f

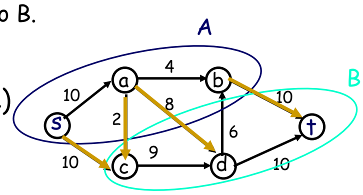
How do we know the flow is maximum?

Cuts and Cut Capacity

Def. A cut is a partition (A,B) of the vertices, s.t. $s \in A$ and $t \in B$.

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

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

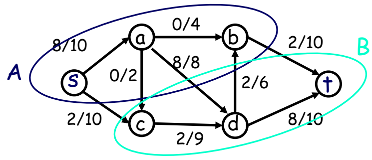


cap(A,B) = 10+2+8+10=30

Note, we do not count edges from B to A.

Cuts and Flows

Consider a graph with some flow and cut



The flow out of A is 2 + 0 + 8 + 2 = 12

The flow in to A is 2

The flow across (A,B) is 12 - 2 = 10

Recall the definition of |f| $\quad |f| = \sum_{e \text{ out of } s} f(e)$

What is |f| in this graph?

Lemma 2

For any flow and any cut

$$|f| = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

Proof.

Since there are no incoming edges to s.

$$|f| = \sum_{e \text{ out of } s} f(e) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ into } s} f(e)$$

Since flow conservation law.

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

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

Lemma 3

For any flow f and any (A,B) cut |f| ≤ cap(A,B)

Proof.

By previous lemma 2

$$|f| = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

Since flow is positive.

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

Max-flow Theorem

The following conditions are equivalent for any f:

1) ∃ cut(A,B) s.t. cap(A,B) = |f|

2) f is a max-flow

3) There is no augmenting path wrt f

1) -> 2)

1) ∃ cut(A,B) s.t. cap(A,B) = |f|

2) f is a max-flow

Proof.

By Lemma 3: |f| ≤ cap(A,B), a flow cannot exceed the capacity of A-B cut

If cap(A,B) = |f|, then it follows that flow f must be the max-flow.

2) -> 3)

2) f is a max-flow

3) There is no augmenting path wrt f

Proof.

By contrapositive.

Suppose there is an augmenting path.
We can improve the flow along this path.
So, it is not max.

3) -> 1)

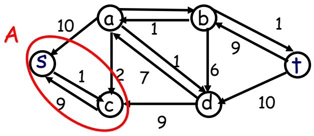
3) There is no augmenting path wrt f

1) ∃ cut(A,B) s.t. cap(A,B) = |f|

Proof. By Lemma 2 (for any flow f and cut)

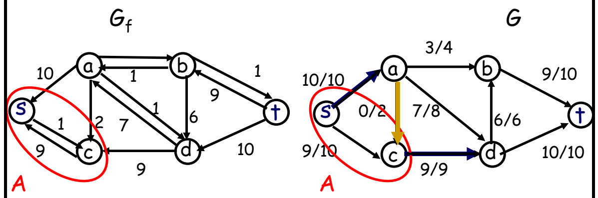
$$|f| = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

Let A be a set of vertices reachable from s in G_f.



Set B contains all other vertices.

3) -> 1)



In graph G:

e=(u, v) s.t. u∈A, v∈B, must be f(e) = c(e), saturated edges

e=(v, u) s.t. u∈A, v∈B, must have f(e) = 0

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

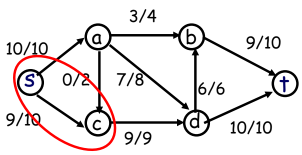
Max-flow Min-cut Theorem

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

Proof.

By Lemma 3: |f| ≤ cap(A,B)

By Max-flow theorem: ∃ cut(A,B) s.t. cap(A,B) = |f|



Min-cut

Finding the Min-cut

Find the maximum flow

Construct the residual graph G_f

Do a BFS to find the nodes reachable from s.
Let the set of these nodes be called A

Let B be all other nodes.

The Ford-Fulkerson Algorithm

Algorithm. Given (G, s, t, c)

1) Start with |f|=0, so f(e)=0
2) Find an augmenting path in G_f
3) Augment flow along this path
4) Repeat until there is no an s-t path in G_f

What is the runtime complexity?

Runtime Complexity

The total number of iterations is _ |f|

Each augmentation increases the value of flow by at least _ 1

The complexity of each iteration is _ by running DFS O(E+V)

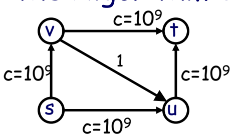
The total runtime is _ O(|f|·(E+V))

Is it polynomial?

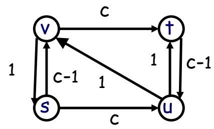
It is pseudo-polynomial because it depends on the size of the integers |f| in the input.

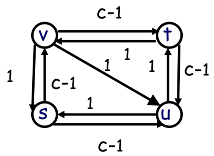
Understanding the Algorithm Complexity

$O(|f| \cdot (E+V))$



Push flow through s-v-u-t

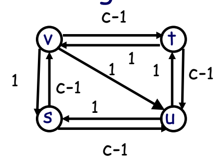




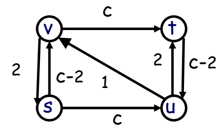
Push flow through s-u-v-t

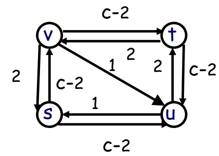
Understanding the Algorithm Complexity

$O(|f| \cdot (E+V))$



Push flow through s-v-u-t





Push flow through s-u-v-t

and so on…

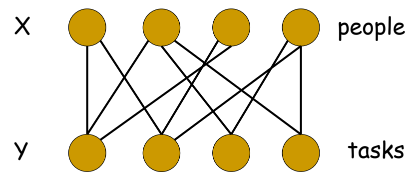
It takes $2 \cdot |c|$ iterations.

Applications

Though the flow problem is itself interesting, but more interesting is how you can use it to solve many problems that don't involve flows directly.

- Data mining.
- Project selection.
- Airline scheduling.
- Bipartite matching.
- Baseball elimination.
- Image segmentation.
- Network connectivity.

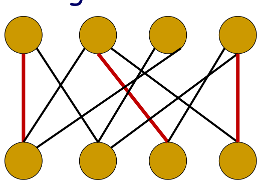
Bipartite Graph



A graph is bipartite if the vertices can be partitioned into two disjoint (also called independent) sets X and Y such that all edges go only between X and Y (no edges go from X to X or from Y to Y). Often writes G = (X, Y, E).

Bipartite Matching

Definition. A subset of edges is a matching if no two edges have a common vertex (mutually disjoint).



Definition. A maximum matching is a matching with the largest possible number of edges

Goal. Find a maximum matching in G.

Reduction

Given an instance of bipartite matching.

Create an instance of network flow.

The solution to the network flow problem can easily be used to find the solution to the bipartite matching.

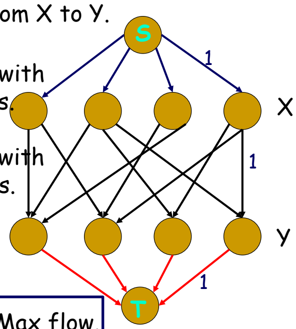
Reducing Bipartite Matching to Net Flow

$\forall e \in E$, direct edges from X to Y.

Create a new vertex S with outgoing directed edges.

Create a new vertex T with incoming directed edges.

Let each edge has capacity equal to 1.



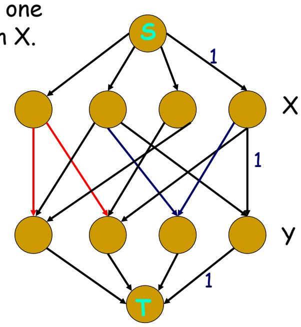
Claim: Max matching = Max flow.

Max matching = Max flow

We can choose at most one edge leaving any node in X.

We can choose at most one edge entering any node in Y.

If we chose more than one, we couldn't have balanced flow.

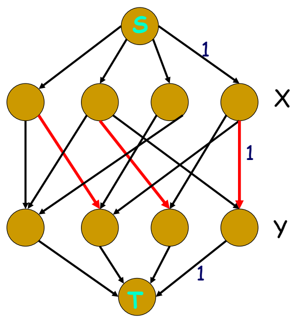


Max matching = Max flow

If there is a matching of k edges, there is a flow f of value k.

Proof.
f has 1 unit of flow across each edge.

$\leq$ 1 unit leaves & enters each node (except s,t)

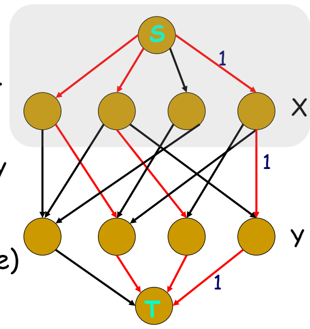


Max matching = Max flow

If there is a flow f of value k, there is a matching with k edges.

Proof.
Recall Lemma 2. For any flow and cut

$$|f| = \sum_{e \text{ out of } X} f(e) - \sum_{e \text{ into } X} f(e)$$

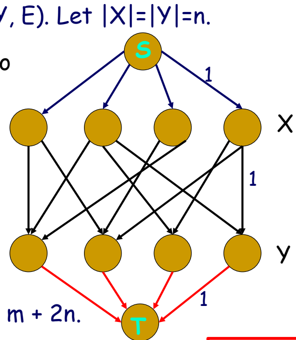


Runtime Complexity

Given bipartite $G = (X, Y, E)$. Let $|X|=|Y|=n$.

How long does it take to solve the network flow problem on the new graph G'?

The running time of Ford-Fulkerson is $O(|f| \cdot (E' + V'))$

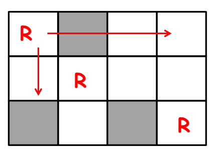


$|f| = n$, size of X. $E' = m + 2n$.

So, runtime is $O((m + 2n)\, n) = O(mn + n^2) = \boxed{O(m\, n)}$

Rook Attack

This problem asks us to place a maximum number of rooks on a chessboard with some squares cut out. The rook moves horizontally or vertically, through any number of squares.



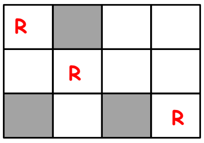
Make sure the rooks do not attack each other!!

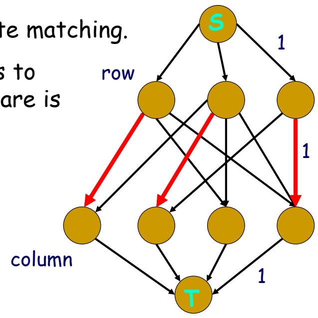
Rook Attack

Notice that at most one rook can be placed on each row or column.

This suggests a bipartite matching.

For each row add edges to every column if the square is not cut out.





Rook Attack

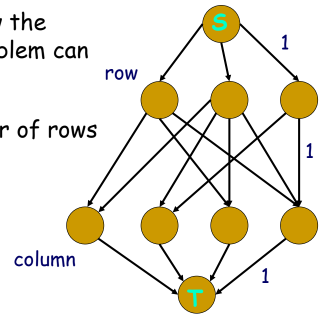
Runtime Complexity - ?

Using the network flow the bipartite matching problem can be solved in O(E V).

Our input is the number of rows and cols.

E = rows * cols.

V = rows or cols.

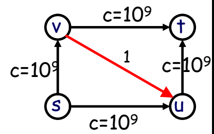


How to improve the efficiency of the Ford-Fulkerson Algorithm?

How to improve the efficiency?

O(|f|·(E+V))

In FF algorithm we run DFS. What about if we run BFS? BFS will return the <u>shortest</u> path in terms of edges.



This variation is called the Edmonds-Karp algorithm

It can be shown that this requires only O(V·E) iterations.

The total runtime: $O(V \cdot E^2)$.

Edmonds-Karp algorithm

Lemma. The algorithm makes at most E V iterations

Sketch of Proof. Let's layout G_f according to BFS.

A path from s to t contains exactly one vertex from each level.

Consider the distance d(v) from s to v.

On each iteration we will saturate at least one forward edge.

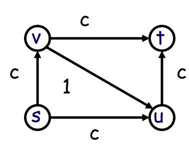
Within E iterations either t becomes disconnected or we use a non-forward edge, thus d(v) increases by 1.

The distance between s and t can increase at most V.

Total: E V iterations

Capacity-Scaling Algorithm

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



Intuition: choose augmenting path with highest bottleneck capacity.

This variation is called the capacity scaling algorithm

The algorithm constructs a series of max-flow problems.

Capacity-Scaling Algorithm

Set Δ to be 2^Δ < max(c(e) out of s).

Consider the residual graph $G_f(\Delta)$ consisting only of edges with c(e) > Δ.

```
while (Δ≥1)
   construct Gf(Δ)
   while (∃ augmenting path)
      augment flow
      update Gf(Δ)
   Δ = Δ/2
return f
```

the residual graph $G_f(\Delta)$ is only used to guide the selection of residual path.

Note, for Δ=1, this algorithm is identical to the FF.

Thus, when it terminates, the flow is max.

Runtime Complexity

Δ is the largest such that $2^\Delta \le$ max(c) = C.

```
while (Δ≥1)
   construct Gf(Δ)
   while (∃ augmenting path)
      augment flow
      update Gf(Δ)
   Δ = Δ/2
return flow
```

? E^2

log C

What is the number of augmentations in the algorithm?

What is the number of augmentations in the algorithm?

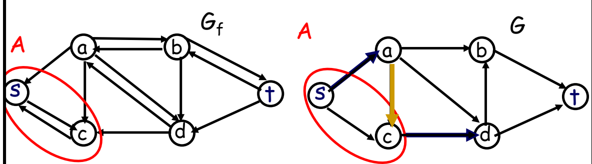
Lemma A. Let f be the flow at the end of Δ phase. Then, $\exists$ A-B cut s.t. cap(A, B) $\le$ |f| + E Δ.

Lemma B. The number of augmentations in a scaling phase is at most 2 E.

Theorem. The Scaling Max-flow algorithm finds a maximum flow in O(E^2 log C) time.

Let f be the flow at the end of Δ phase. Then, $\exists$ A-B cut s.t. cap(A, B) $\le$ |f| + E Δ.

Proof. Similar to the max flow theorem, see slides 31 and 32. The pix is from those slides



e=(u, v) s.t. u$\in$A, v$\in$B, must be f(e) > c(e) - Δ.

e=(v, u) s.t. u$\in$A, v$\in$B, must have f(e) < Δ.

Let f be the flow at the end of Δ phase. Then, $\exists$ A-B cut s.t. cap(A, B) $\le$ |f| + E Δ.

Proof. (continue)

$$|f| = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

$$\ge \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ into } A} \Delta$$

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

$$\ge cap(A, B) - E\,\Delta$$

The number of augmentations in a scaling phase is at most 2 E.

Proof. Consider two phases: Δ and Δ' = 2 Δ.

At the end of Δ' (by the prev. lemma)

maxflow $\le$ cap(A, B) $\le$ |f| + E Δ' = |f| + 2 E Δ

since cap(A, B) is an upper bound on the max flow, lemma 3, slide 26.

But each augmentation increases the flow by at least Δ. Thus, the total number of augmentation is 2 E.

Runtime history

year	discoverer(s)	bound
1951	Dantzig [11]	$O(n^2 mU)$
1956	Ford & Fulkerson [17]	$O(nmU)$
1970	Dinitz [13] Edmonds & Karp [15]	$O(nm^2)$ shortest path
1970	Dinitz [13]	$O(n^2 m)$
1972	Edmonds & Karp [15] Dinitz [14]	$O(m^2 \log U)$ capacity scaling
1973	Dinitz [14] Gabow [19]	$O(nm \log U)$
1974	Karzanov [36]	$O(n^3)$ preflow-push
1977	Cherkassky [9]	$O(n^2 m^{1/2})$
1980	Galil & Naamad [20]	$O(nm \log^2 n)$
1983	Sleator & Tarjan [46]	$O(nm \log n)$ splay tree
1986	Goldberg & Tarjan [26]	$O(nm \log(n^2/m))$ preflow-push
1987	Ahuja & Orlin [2]	$O(nm + n^2 \log U)$
1987	Ahuja et al. [3]	$O(nm \log(n\sqrt{\log U}/m))$
1989	Cheriyan & Hagerup [7]	$E(nm + n^2 \log^2 n)$
1990	Cheriyan et al. [8]	$O(n^3/\log n)$
1990	Alon [4]	$O(nm + n^{8/3} \log n)$
1992	King et al. [37]	$O(nm + n^{2+\epsilon})$
1993	Phillips & Westbrook [44]	$O(nm(\log_{m/n} n + \log^{2+\epsilon} n))$
1994	King et al. [38]	$O(nm \log_{m/(n \log n)} n)$
1997	Goldberg & Rao [24]	$O(\min(n^{2/3}, m^{1/2}) m \log(n^2/m) \log U)$
2013	Orlin	O(m n)