Announcements

- Reminder: Midterm 1 is next Thursday, i.e., in a week
  - Feel free to let us know if you need any accommodation, due to, e.g., time zone difference, etc.
  - HWs are the best practice questions, make sure you really understand how each problem is solved
  - You will apply the same techniques learned in class, but likely, you will not encounter the same algorithmic problem

CS6161: Design and Analysis of Algorithms (Fall 2020)

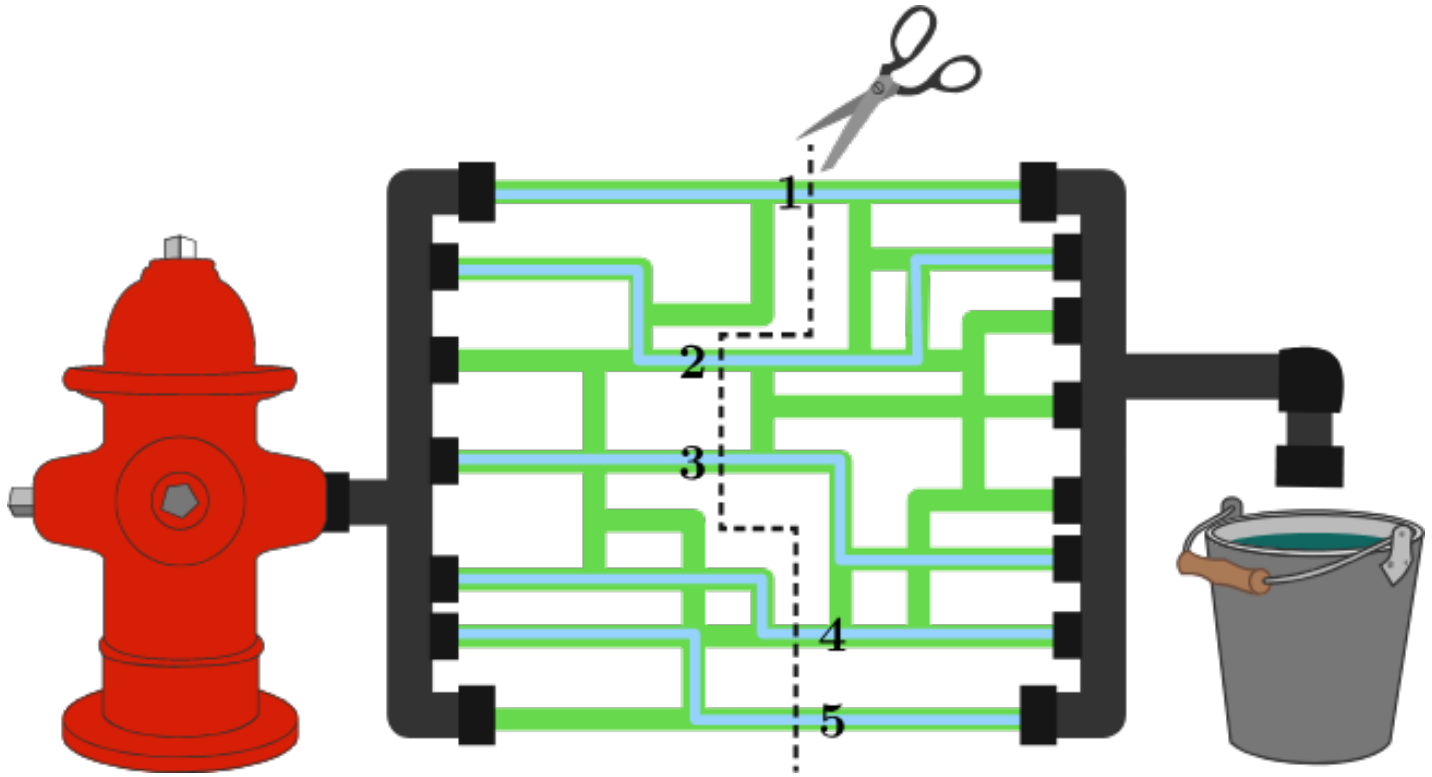
Max Flow and Min Cut (II)

Instructor: Haifeng Xu

Outline

- Ford-Fulkerson (FF) Algorithm and Correctness
- Efficient Implementation

Max Flow and Min Cut Problems

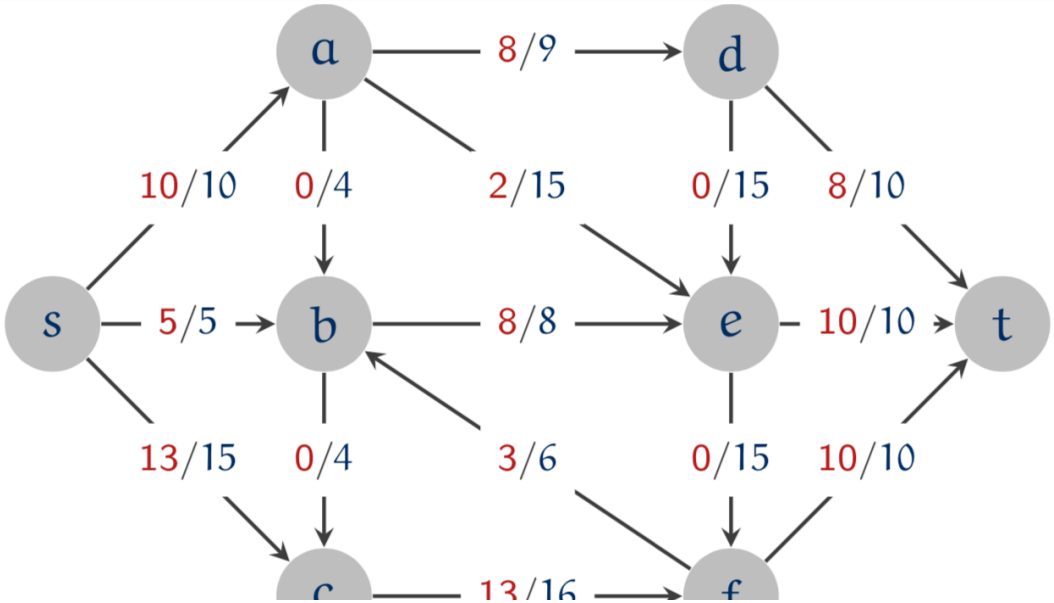


Flow in Flow Networks

- An s-t flow $f: E \to \mathbb{R}$ is a real-valued function that satisfies:
  - $\forall\, e \in E : 0 \leq f(e) \leq c(e)$ [capacity]
  - $\forall\, v \neq \{s, t\}$: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]
- Th value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e)$

Max-Flow Problem

Find a (feasible) flow of maximum value

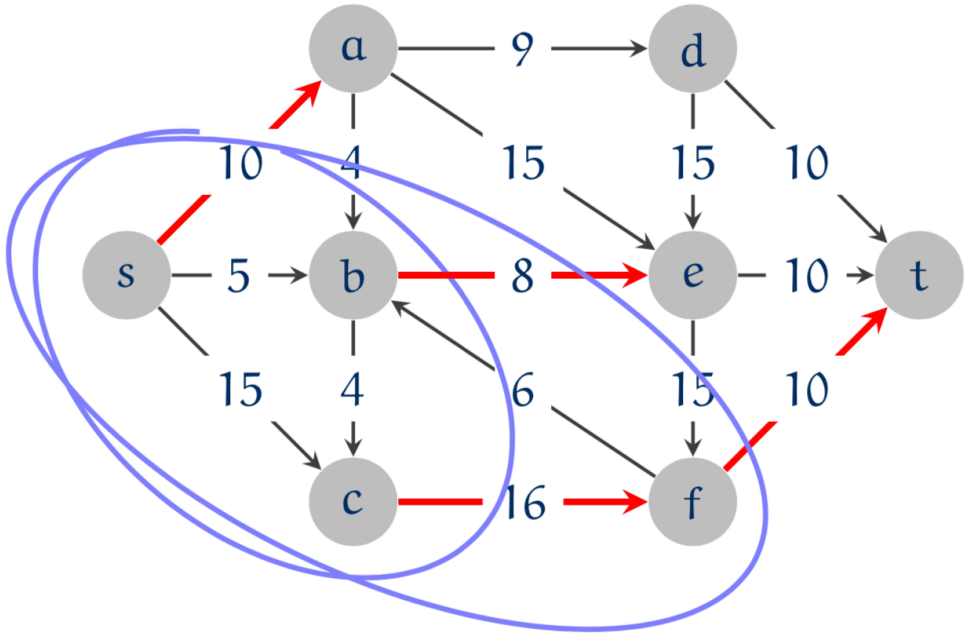


Cut in Flow Networks

- An s-t cut is a partition (A, B) of the vertices with $s \in A$ and $t \in B$
- The capacity of a cut = sum of capacities of edges from A to B.

Min-Cut Problem

Find a cut of minimum capacity



Ford-Fulkerson Alg for Max Flow

A natural greedy algorithm using augmenting path in residual graph

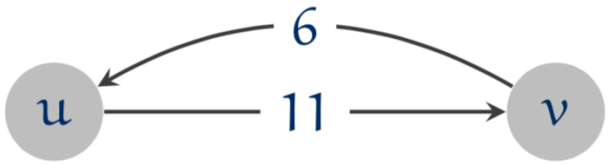
Original edge: $e = (u, v) \in E$

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

Residual edges: $e = (u, v) \in E$

- Can "undo" sent flow
- $e = (u, v)$ and $e^R = (v, u)$
- Residual capacity

$$\begin{cases} c_f(e) = c(e) - f(e) \\ c_f(e^R) = f(e) \end{cases}$$



Residual Graph: $G_f = (V, E_f)$

- $E_f = \{e: f(e) < c(e)\} \cup \{e^R: f(e) > 0\}$
  - Residual edges are all edges that are allowed to send flows
- Key property: if f' is a flow in G_f, then $f + f'$ is a flow in G

Ford-Fulkerson Alg for Max Flow

A natural greedy algorithm using augmenting path in residual graph

An augmenting path is a simple $s - t$ path in residual graph G_f.

- bottleneck capacity = minimum residual capacity among path edges

Residual Graph: $G_f = (V, E_f)$

- $E_f = \{e: f(e) < c(e)\} \cup \{e^R: f(e) > 0\}$
  - Residual edges are all edges that are allowed to send flows
- Key property: if f' is a flow in G_f, then $f + f'$ is a flow in G

Ford-Fulkerson Alg for Max Flow

A natural greedy algorithm using augmenting path in residual graph

- Start with $f(e) = 0$ for all edges $e \in E$
- Find an augmenting $s - t$ path P in the residual graph G_f
- Augment flow along path P
- Repeat, until stuck

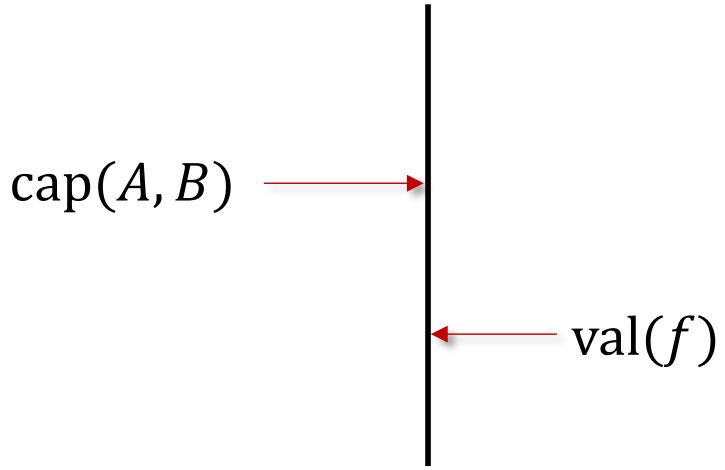
Not clear yet whether it will terminate!

Algorithm: Ford-Fulkerson(G, s, t, c):

```
foreach edge e ∈ E do
    f(e) = 0;
G_f = residual graph;
while there exists an augmenting path P in G_f do
    f = Augment(f, c, P);
    Update G_f;
return f
```

Weak Duality: Min-Cut is an Upper Bound

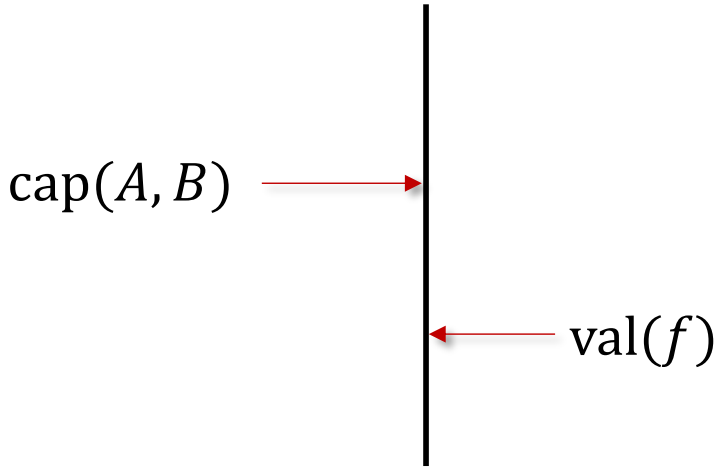
Weak Duality Lemma: Let f be any flow and (A, B) be any cut. Then $\text{val}(f) \leq \text{cap}(A, B)$.



Strong Duality: Min-Cut is the Exact Bound

Max-Flow Min-Cut Strong Duality Theorem:

$$\max_{\text{f}} \ \text{val}(f) = \min_{(A,B)} \ \text{cap}(A,B).$$



Strong Duality: Min-Cut is the Exact Bound

Max-Flow Min-Cut Strong Duality Theorem:

$$\max_{\text{f}} \ \text{val}(f) = \min_{(A,B)} \ \text{cap}(A, B).$$

Augmenting Path Theorem: A flow f is a max-flow if and only if there are not augmenting paths.

- Augmenting path theorem implies that, *if FF algorithm terminates*, it must output the max flow

Strong Duality: Min-Cut is the Exact Bound

Max-Flow Min-Cut Strong Duality Theorem:

$$\max_{\text{f}} \ \text{val}(f) = \min_{(A,B)} \ \text{cap}(A,B).$$

Augmenting Path Theorem: A flow f is a max-flow if and only if there are not augmenting paths.

Proof: show following three conditions are equivalent for any flow f

1. There exists a cut (A, B) such that $cap(A, B) = val(f)$
2. f is a max-flow
3. There is no augmenting path with respect to f

1 + Weak Duality ⇒ Strong Duality Theorem

2 + 3 ⇒ Augmenting Path theorem

Proofs

- [(1) ⇒ (2)]: trivial by weak duality.

- [(2) ⇒ (3)]:
  - Assume by contradiction 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

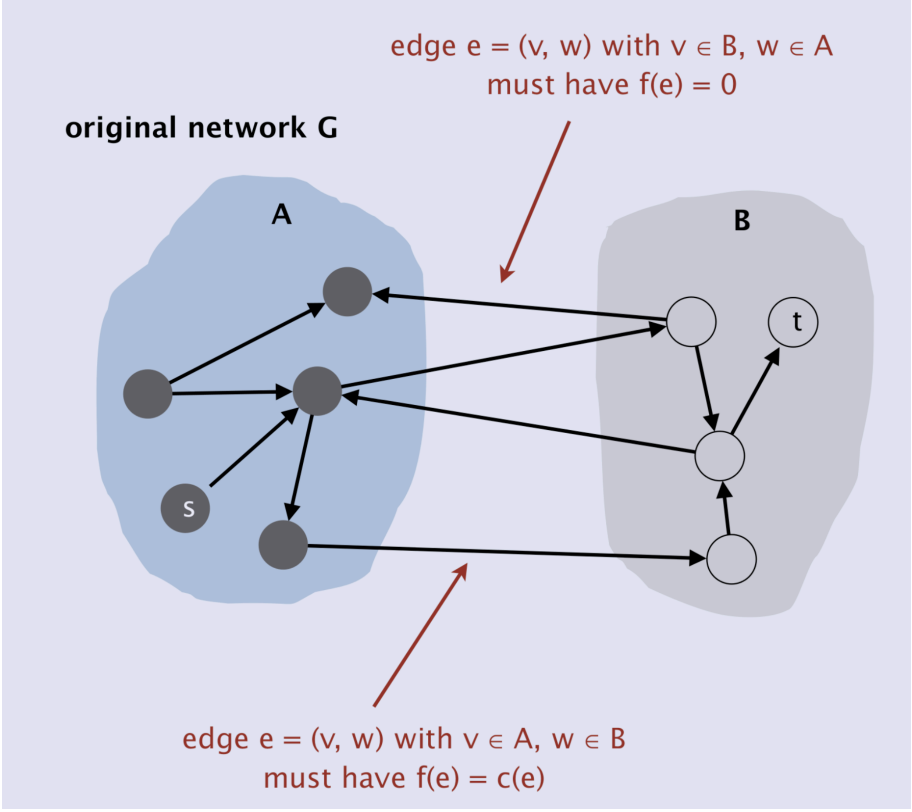
Proofs

- [(3) ⇒ (1)]: i.e., no augmenting path ⇒ ∃ (A, B) s.t. $\text{cap}(A, B) = \text{val}(f)$
  - Let f be a flow with no augmenting paths
  - Let A be the set of vertices reachable from s in residual graph G_f
  - By definition of $A, s \in A$
  - By definition of flow $f, t \notin A$

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

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

$$= \text{cap}(A, B)$$



Back to Ford-Fulkerson

Algorithm: Ford-Fulkerson(G, s, t, c):

foreach *edge* $e \in E$ **do**
    $f(e) = 0$;
$G_f =$ residual graph;
while *there exists an augmenting path* P *in* G_f **do**
    $f =$ Augment(f, c, P);
    Update G_f;
return f

Corollary: If FF terminates, it outputs a maximum flow.

Running Time Analysis of FF

- Assumption. Capacities are integers between 1 and C.
- Implication: integrality invariant. That is, throughout the algorithm, the flow values $f(e)$ and residual capacities $c_f(e)$ are integers

Theorem: FF terminates in at most $val(f^*) \leq nC$ iterations.

Proof: each augmentation increases the value by at least 1

Corollary: Running time of FF $O(mnC)$.

Why?
Updating residual graph and finding an augmenting path (via DFS or BFS) takes $O(m)$ time

Running Time Analysis of FF

This is in pseudo-polynomial time!

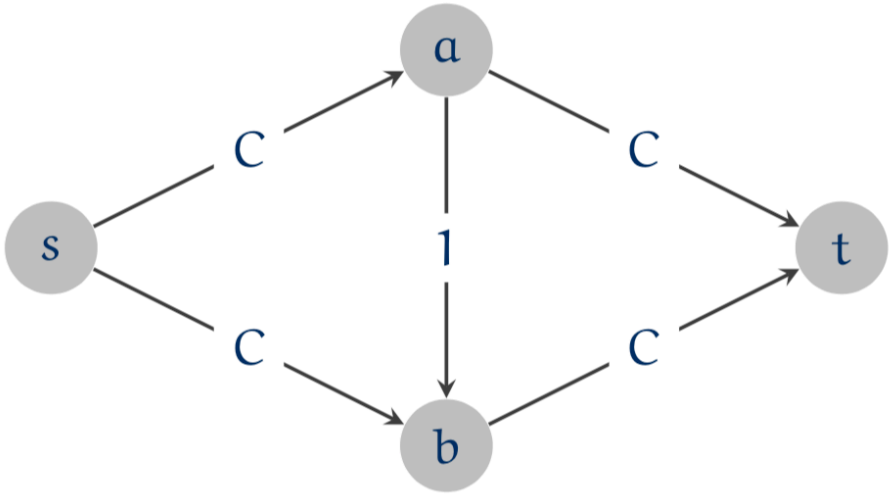
The input size is polynomial in $m, n, \log C$

Corollary: Running time of FF $O(mnC)$.

Outline

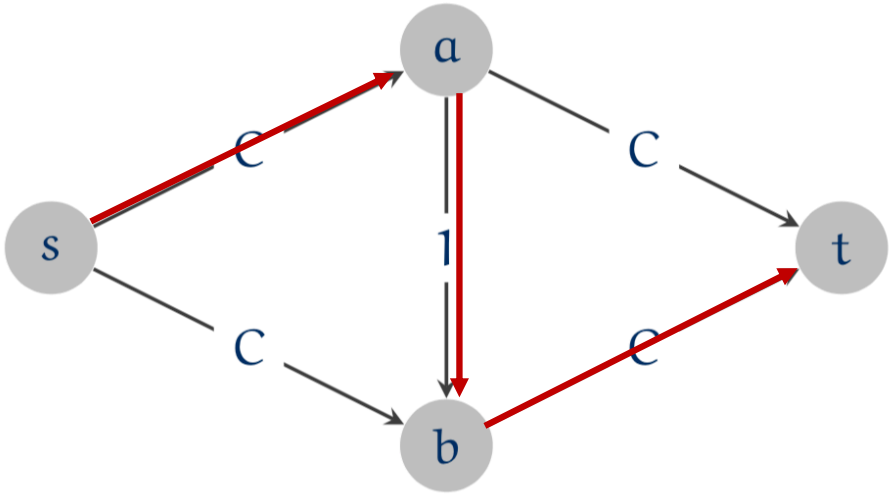
- Ford-Fulkerson (FF) Algorithm and Correctness
- Efficient Implementation

A bad Case for Ford-Fulkerson



Requires $2C$ iterations.

A bad Case for Ford-Fulkerson

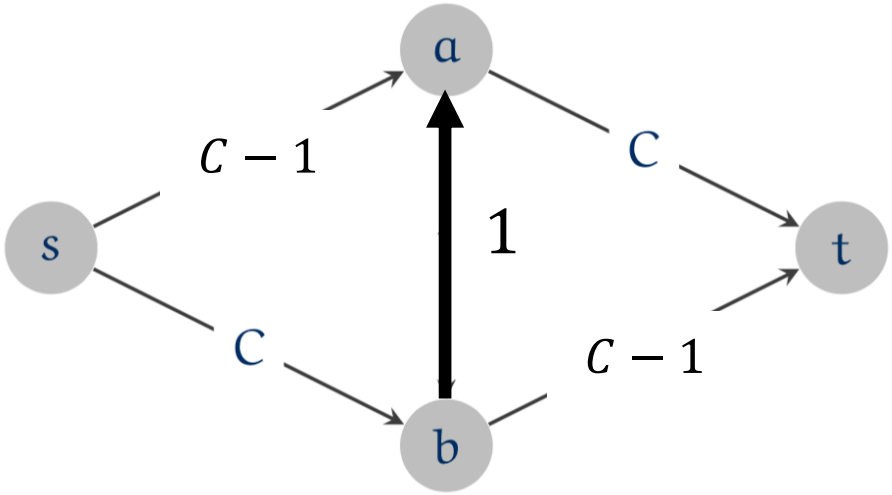


Requires $2C$ iterations.

Augmenting Path:

- $s \to a \to b \to t$

A bad Case for Ford-Fulkerson



Augmenting Path:

- $s \to a \to b \to t$
- $s \to b \to a \to t$

Requires $2C$ iterations.

Q: can we have a truly polynomial time algorithm?

Choosing Good Augmenting Paths

Goal: choose augmenting paths so that:

1) Can find augmenting paths efficiently.
2) Algorithm takes few iterations.

Options: choose augmenting paths with

- Max bottleneck capacity
- Or, sufficiently large bottleneck capacity
- Or, fewest number of edges

Option 1: Max Bottleneck Capacity

- Idea: choose augmenting path with (almost) largest bottleneck capacity: it increases flow by max possible amount at each iteration.

- Turns out to work, but detail of the algorithm is omitted here

Theorem: The above variant of FF finds a max flow in $O(m \log C)$ augmentations. It can be implemented to run in $O(m^2 \log C)$ time.

Option II: Shortest Augmenting Paths

- Idea: choose the shortest (in terms of number of edges) augmenting path
- This version is also called Edmonds-Karp (EK) Implementation

Theorem: The EK implementation of FF runs in $O(m^2 n)$ time.

High-level idea:

- Can prove $O(mn)$ augmentations suffice
  - $O(m)$ augmentations for paths of exactly k edges.

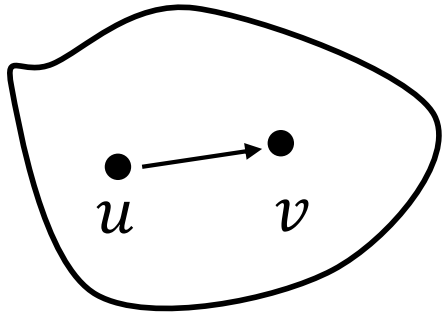
Proof of the Efficiency of FF with EK

- Define $d(v)$ be the distance, measured by #edges, from s to v

Lemma 1: Throughout the algorithm, $d(v)$ never decreases.

Proof by contradiction

- Suppose it ever decreases, let v be the node that is closest to s among all those that decreased in new residual graph
- Consider edge (u, v) in the shortest path from s to v in new residual graph



New residual graph after adding flow from augmenting path

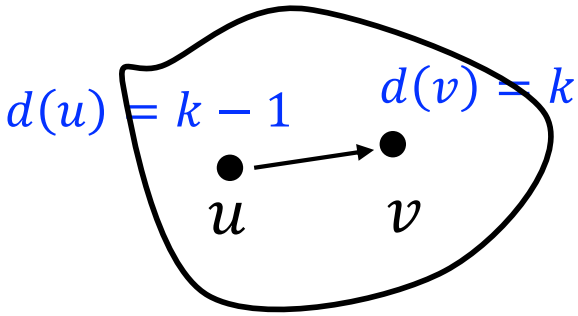
Proof of the Efficiency of FF with EK

- Define $d(v)$ be the distance, measured by #edges, from s to v

Lemma 1: Throughout the algorithm, $d(v)$ never decreases.

Proof by contradiction

- Suppose it ever decreases, let v be the node that is closest to s among all those that decreased in new residual graph
- Consider edge (u, v) in the shortest path from s to v in new residual graph



New residual graph after adding flow from augmenting path

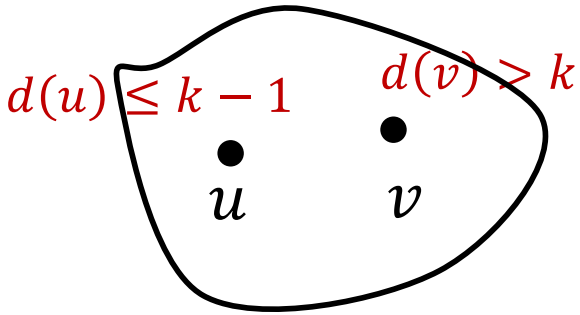
Proof of the Efficiency of FF with EK

- Define $d(v)$ be the distance, measured by #edges, from s to v

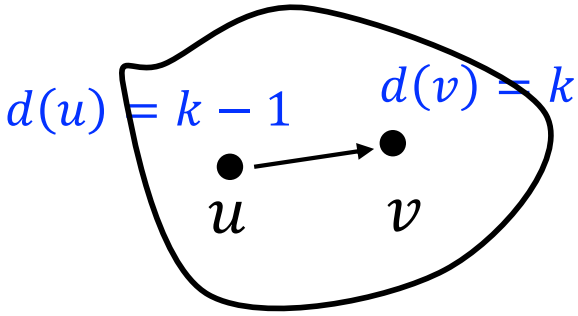
Lemma 1: Throughout the algorithm, $d(v)$ never decreases.

Proof by contradiction

- Suppose it ever decreases, let v be the node that is closest to s among all those that decreased in new residual graph
- Consider edge (u, v) in the shortest path from s to v in new residual graph



Original residual graph



New residual graph after adding flow from augmenting path

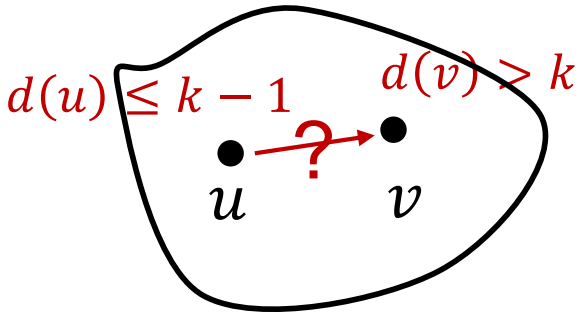
Proof of the Efficiency of FF with EK

- Define $d(v)$ be the distance, measured by #edges, from s to v

Lemma 1: Throughout the algorithm, $d(v)$ never decreases.

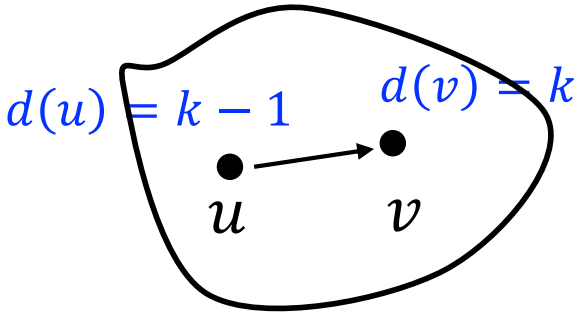
Proof by contradiction

- Suppose it ever decreases, let v be the node that is closest to s among all those that decreased
- Consider edge (u, v) in the shortest path from s to v in new residual graph



Original residual graph

Cannot be



New residual graph after adding flow from augmenting path

Proof of the Efficiency of FF with EK

- Define $d(v)$ be the distance, measured by #edges, from s to v

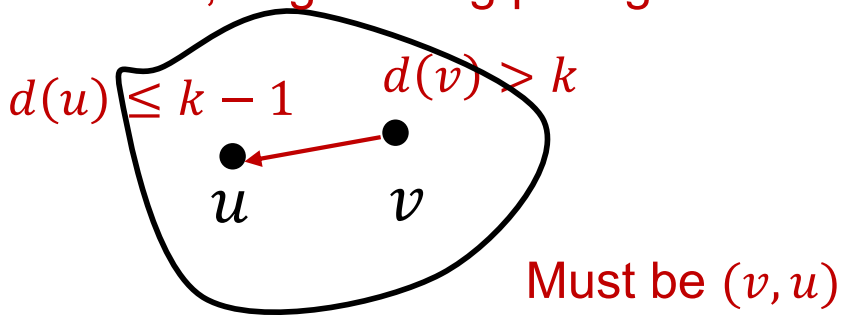
Lemma 1: Throughout the algorithm, $d(v)$ never decreases.

Proof by contradiction

- Suppose it ever decreases, let v be the node that is closest to s among all those that decreased
- Consider edge (u, v) in the shortest path from s to v in new residual graph

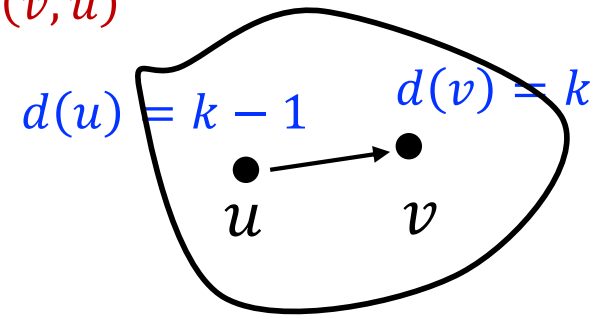
Moreover, augmenting path goes through (v, u)

$\Rightarrow d(u) > k$ contradiction!



Must be (v, u)

Original residual graph



New residual graph after adding flow from augmenting path

Proof of the Efficiency of FF with EK

- We call an edge critical if it is an edge with bottleneck capacity in the augmenting path
- So in each augmenting path, at least one edge has the chance to be crucial

Lemma 2: For any edge (u, v), between two consecutive occasions when (u, v) is critical, $d(u)$ increases by at least 2.

Proof of Lemma 2: HW exercise (use Lemma 1)

Proof of the Efficiency of FF with EK

- We call an edge critical if it is an edge with bottleneck capacity in the augmenting path
- So in each augmenting path, at least one edge has the chance to be crucial

Lemma 2: For any edge (u, v), between two consecutive occasions when (u, v) is critical, $d(u)$ increases by at least 2.

Proof of the efficiency of FF with EK:

- Each edge can be critical for at most $n/2$ times
- There are m edges, so we need to augment at most $\frac{nm}{2}$ times
- Running time $O(m^2 n)$

Choosing Good Augmenting Paths: Summary

Assumption. Capacities are integers between 1 and C.

method	augmentations	running time
augmenting path	nC	$O(mnC)$
fattest augmenting path	$m \log(mC)$	$O(m^2 \log n \log(mC))$
capacity scaling	$m \log C$	$O(m^2 \log C)$
improved capacity scaling	$m \log C$	$O(mn \log C)$
shortest augmenting path	mn	$O(m^2 n)$
improved shortest augmenting path	mn	$O(mn^2)$
dynamic trees	mn	$O(mn \log n)$

Max-Flow Algorithms: Summary

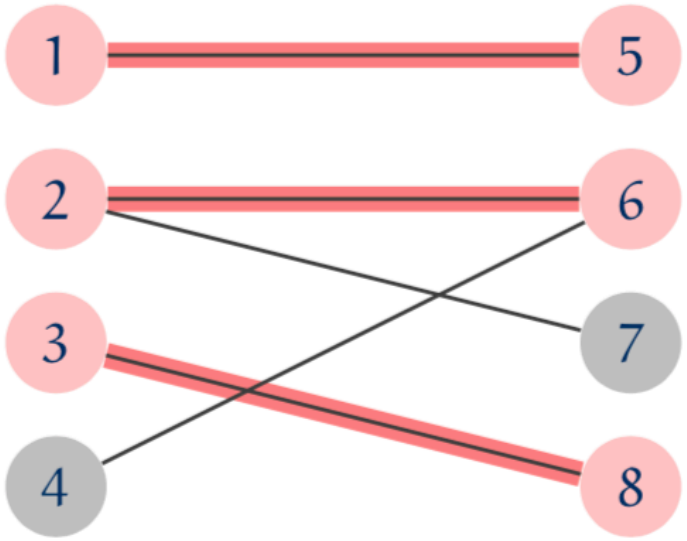
- There is a Wikipedia page on this!

Method	Complexity	Description
Linear programming		Constraints given by the definition of a legal flow. See the linear program here.
Ford–Fulkerson algorithm	$O(E \max\|f\|)$	As long as there is an open path through the residual graph, send the minimum of the residual capacities on the path. The algorithm is only guaranteed to terminate if all weights are rational. Otherwise it is possible that the algorithm will not converge to the maximum value. However, if the algorithm terminates, it is guaranteed to find the maximum value.
Edmonds–Karp algorithm	$O(VE^2)$	A specialization of Ford–Fulkerson, finding augmenting paths with breadth-first search.
Dinic's blocking flow algorithm	$O(V^2E)$	In each phase the algorithms builds a layered graph with breadth-first search on the residual graph. The maximum flow in a layered graph can be calculated in $O(VE)$ time, and the maximum number of the phases is V-1. In networks with unit capacities, Dinic's algorithm terminates in $O(\min\{V^{2/3}, E^{1/2}\}E)$ time.
MPM (Malhotra, Pramodh-Kumar		
Push-relabel algorithm with *FIFO* vertex selection rule	$O(V^3)$	Push-relabel algorithm variant which always selects the most recently active vertex, and performs push operations until the excess is positive or there are admissible residual edges from this vertex.
Push-relabel algorithm with dynamic trees	$O\left(VE \log \frac{V^2}{E}\right)$	The algorithm builds limited size trees on the residual graph regarding to height function. These trees provide multilevel push operations.
KRT (King, Rao, Tarjan)'s algorithm[11]	$O(EV \log_{\frac{E}{V \log V}} V)$	
Binary blocking flow algorithm[12]	$O\left(E \cdot \min(V^{\frac{2}{3}}, \sqrt{E}) \cdot \log \frac{V^2}{E} \log U\right)$	The value U corresponds to the maximum capacity of the network.
James B Orlin's + KRT (King, Rao, Tarjan)'s algorithm[9]	$O(VE)$	Orlin's algorithm solves max-flow in $O(VE)$ time for $E \leq O(V^{\frac{16}{15}-\epsilon})$ while KRT solves it in $O(VE)$ for $E > V^{1+\epsilon}$.

One of the biggest open problems in theory: is there an almost linear (in m) time algorithm???

Application: Bipartite Matching

- Bipartite graph is a graph, in which nodes can be separated into two sets L, R, such that all the edges are between L and R
- A bipartite matching is a subset of edges $M \subseteq E$ such that each vertex appears in at most one edge in M

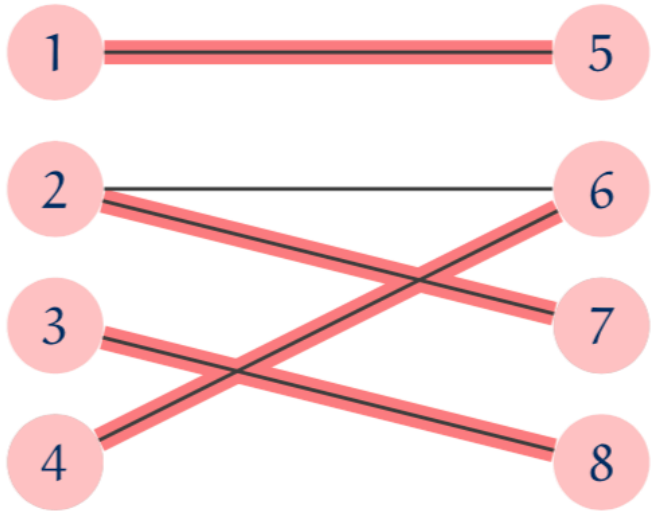


Application: Bipartite Matching

- Bipartite graph is a graph, in which nodes can be separated into two sets L, R, such that all the edges are between L and R
- A bipartite matching is a subset of edges $M \subseteq E$ such that each vertex appears in at most one edge in M

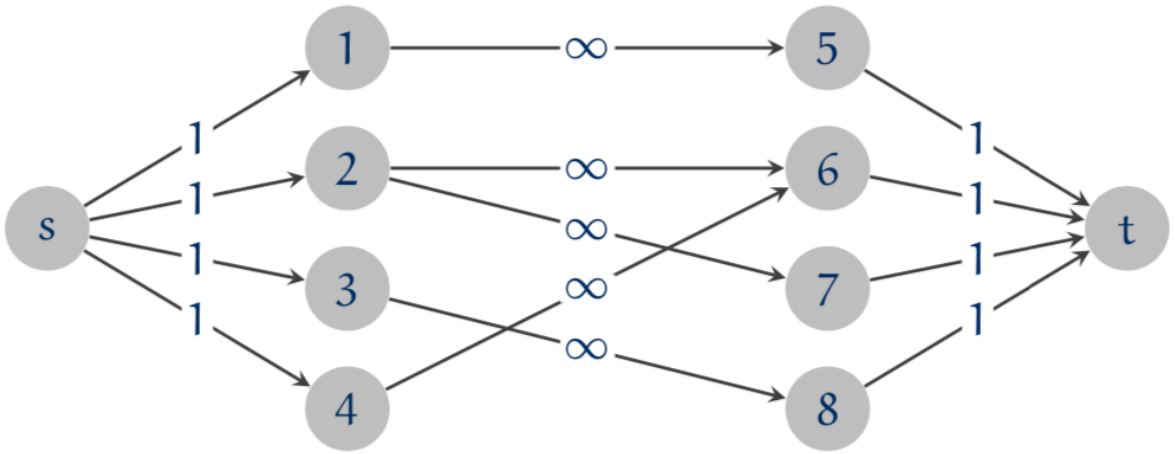
The Max Bipartite Matching Problem

Find the max cardinality bipartite matching



Solving Max Bipartite Matching via Max-Flow

- Create a directed graph $G' = \{L \cup R \cup \{s, t\}, \quad E'\}$.
- Direct all edges from L to R, and assign infinite capacity.
- Add source s, and unit capacity edges from s to each vertex in L
- Add sink t, and unit capacity edges from each vertex in R to t

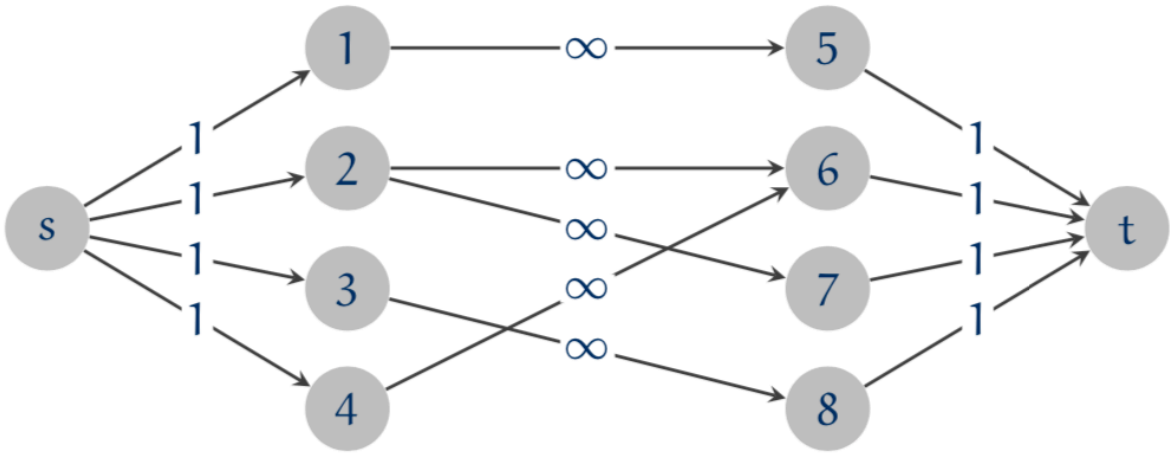


G'

Solving Max Bipartite Matching via Max-Flow

Theorem:

1) Max flow in G' can be computed in $O(nm)$ time
2) The edges used by any max flow of G', restricted to graph G, forms a max bipartite matching



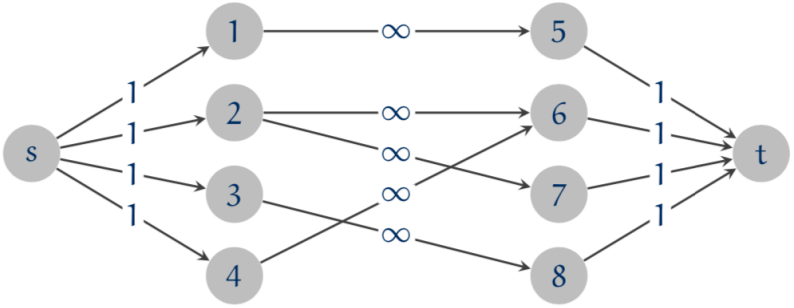
G'

Solving Max Bipartite Matching via Max-Flow

Theorem:

1) Max flow in G' can be computed in $O(nm)$ time
2) The edges used by any max flow of G', restricted to graph G, forms a max bipartite matching

- Proof
- (1): max flow is value at most n, each augmenting paths increases the value by at least 1
- (2):
  - First, any max bipartite matching can be converted to a flow, so $val(f^*) \geq$ size of max bipartite matching
  - Second, any max flow found by FF gives rise to a bipartite matching, since each node has at most one unit flow going through $\Rightarrow val(f^*) \leq$ matching size



Thank You

Haifeng Xu

University of Virginia

hx4ad@virginia.edu