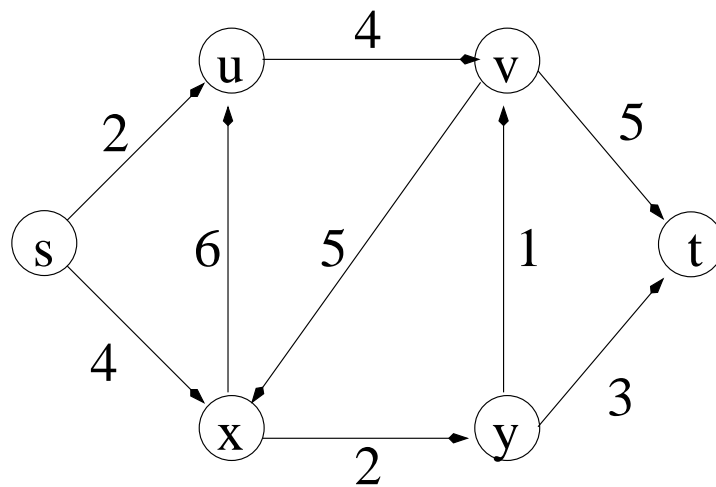


Network Flows

1. Flows deal with network models where edges have capacity constraints.
2. Shortest paths deal with edge costs.

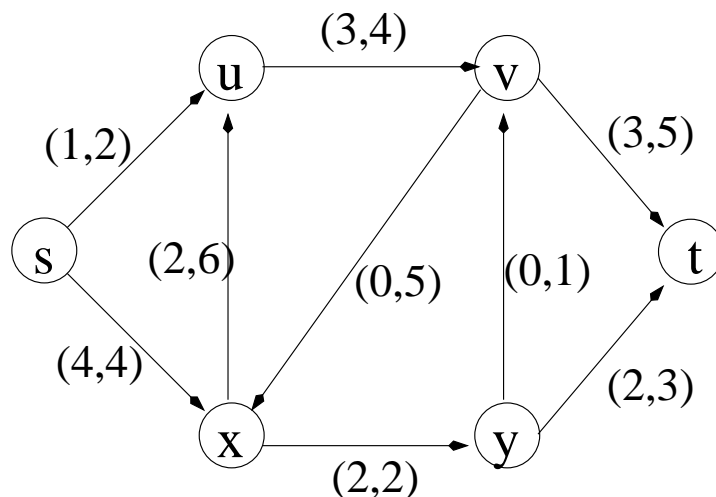


3. Network flow formulation:

- A network $G = (V, E)$.
- Capacity $c(u, v) \geq 0$ for edge (u, v) .
- Assume $c(u, v) = 0$ if $(u, v) \notin E$.
- Source s and sink t .

Network Flows

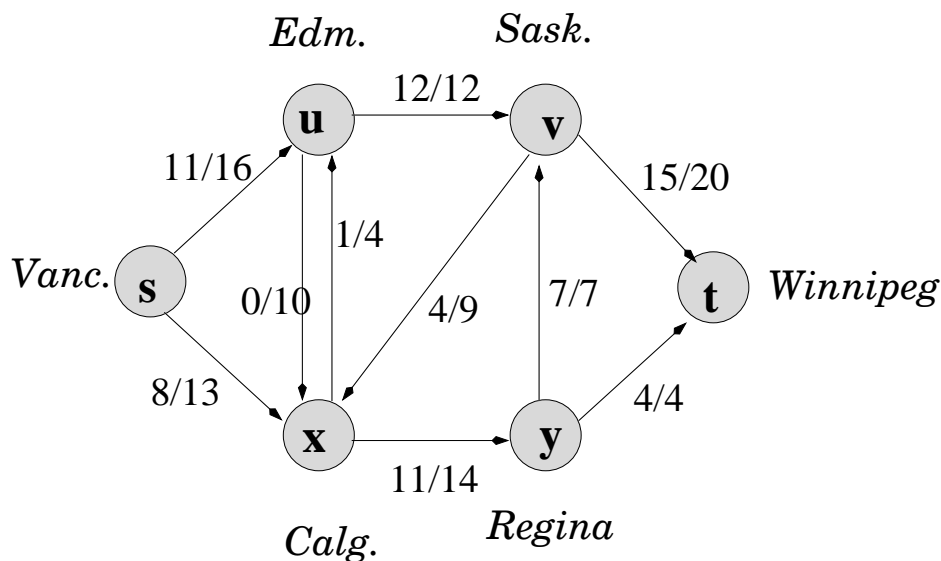
1. Flow $f : E \rightarrow R^+$ such that
 - $f(u, v) \leq c(u, v)$, for all $(u, v) \in E$.
 - $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$, for $u \neq s, t$.
2. These conditions are called **Capacity Constraint** and **Flow Conservation**.
3. Value of flow f is $|f| = \sum_{v \in V} f(s, v)$.



4. (Max Flow Problem:) Given G , s and t , determine max-valued flow from s to t .
5. Optional Lower Bound: For some edges, $f(u, v) \geq l(u, v)$.

Applications: Transportation

1. Company wants to ship hockey pucks from Vancouver (factory) to Winnipeg (warehouse).
2. Trucking companies lease space on established routes (lanes).
3. Lane capacity in number of standard crates.



4. What is the maximum number of units that can be shipped from V to W ?
5. In this example, $|f| = 19$.

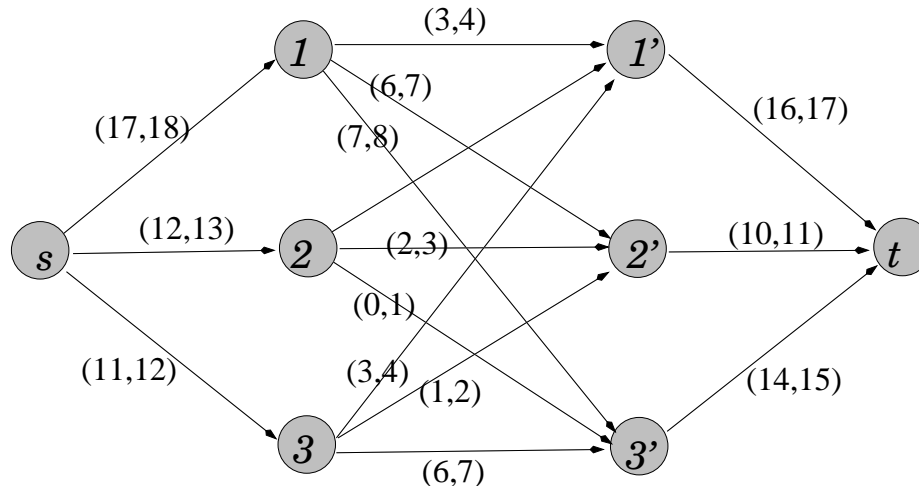
Applications: Matrix Rounding

1. Given a $p \times q$ matrix of reals, row sums α_i , and column sums β_j .
2. We can round any matrix entry a up or down (i.e., to $\lceil a \rceil$ or to $\lfloor a \rfloor$).
3. Round entries as well as row and column sums so that sums are consistent in the rounded matrix.

			Row Sum	
	3.1	6.8	7.3	17.2
	9.6	2.4	0.7	12.7
	3.6	1.2	6.5	11.3
Col Sum	16.3	10.4	14.5	

Applications: Matrix Rounding

1. Node i for row i , and node j' for column j . Two additional nodes s and t .
2. Edge (i, j') for each matrix entry D_{ij} . Edge (s, i) for row-sum i ; edge (j', t) for column sum j .
3. Lower and upper bounds for (i, j') correspond to rounding down and rounding up D_{ij} .



4. Consistent matrix rounding if and only if feasible flow in the network.

Applications: Scheduling

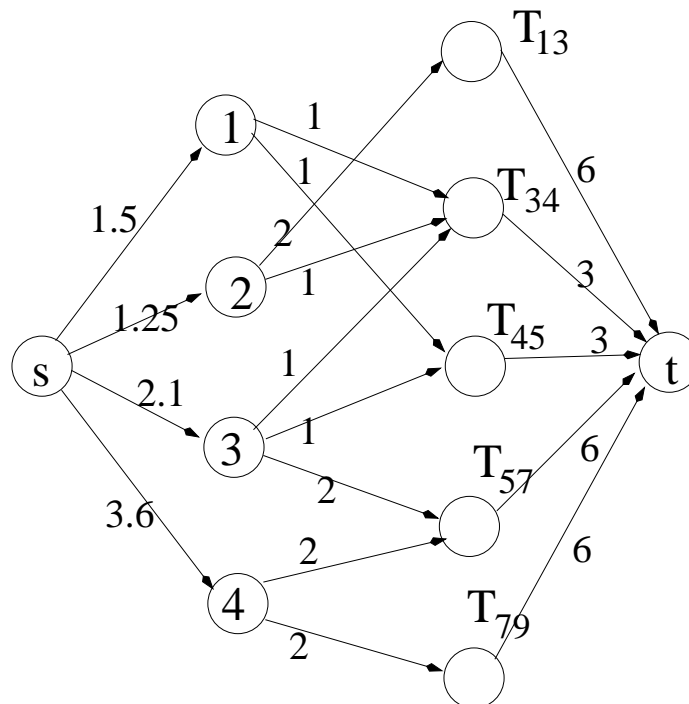
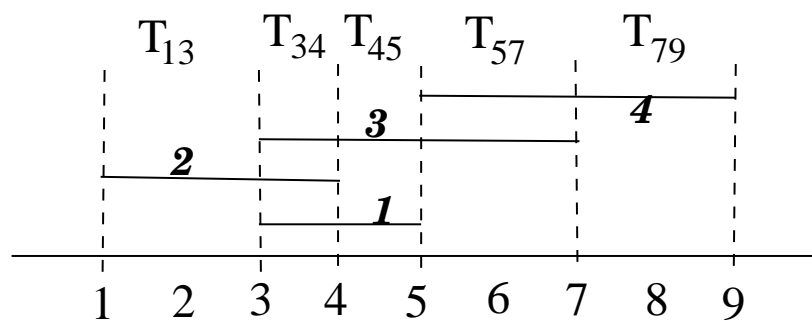
1. Set of jobs, J , to be scheduled on M identical processors.
2. Each job has 3 time parameters: p_i (processing time), r_i (ready for scheduling), d_i (due date).
3. Clearly, $d_i \geq r_i + p_i$ must hold.
4. Preemption allowed: jobs can be interrupted, and restarted later on a different machine.
5. Example.

Job (j)	1	2	3	4
Process Time p_i	1.5	1.25	2.1	3.6
Release Time r_i	3	1	3	5
Due Date d_i	5	4	7	9

6. $M = 3$ machines.

Applications: Scheduling

1. Break the time line into at most $2J - 1$ disjoint intervals. No job begins or ends inside an interval, so the set of jobs available within an interval is constant.



Applications: Scheduling

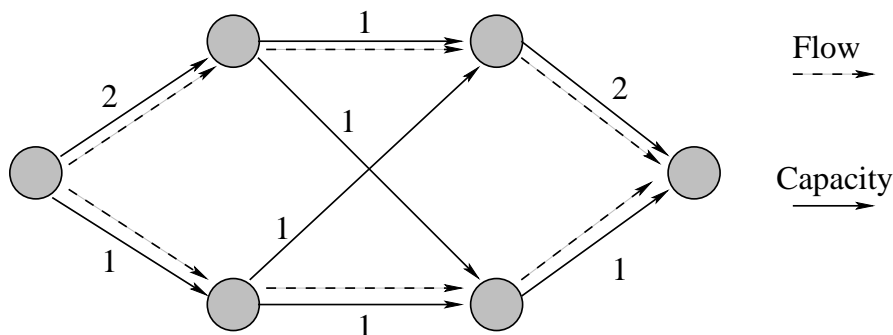
1. Source node s , sink node t , node for each job, and node for each interval.
2. Connect s to job j with capacity p_j .
3. Connect each interval node T_{jk} to sink t with capacity $(k - j)M$, indicating the number of machine units available in that interval.
4. Connect a job i to each interval T_{jk} such that $r_i \leq j$ and $d_i \geq k$. The capacity of this edge is $(k - j)$ indicating the number of machine units allottable to job i in this interval.
5. The scheduling problem has a solution if and only if there is max flow of value $\sum_j p_j$.

Ford-Fulkerson Method

1. Developed by Ford and Fulkerson [1956]
2. Widely used, and influential ideas:
Augmentation, residual network, and famous Min-Cut Max-Flow theorem.

Generic FF (G, s, t)

- Initialize $f = 0$;
 - while there is an augmenting path p
augment f along p
3. Basically, a greedy method, but...

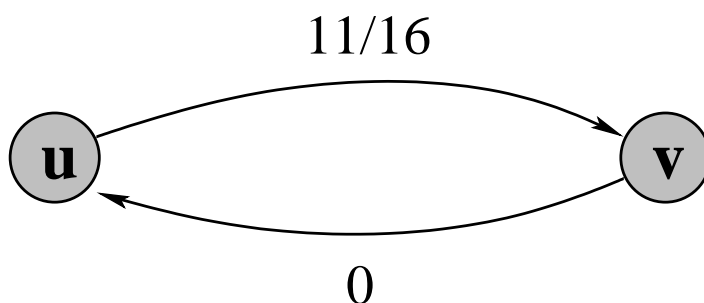


Residual Networks

1. Previous example shows augmenting on G may not find optimal flow.
2. Given current flow f , define residual capacity of an edge (u, v) as follows:

$$c_f(u, v) = c(u, v) - f(u, v).$$

3. Residual capacity is the additional flow one can send on an edge, possibly by canceling some flow in opposite edge.



4. $c_f(u, v) = 16 - 11 = 5.$

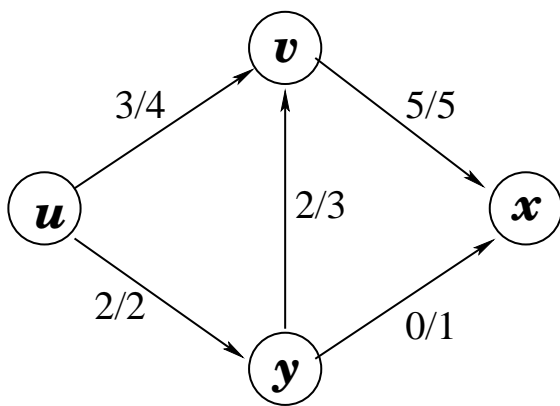
5. $c_f(v, u) = 0 - 11 = 11.$

Residual Networks

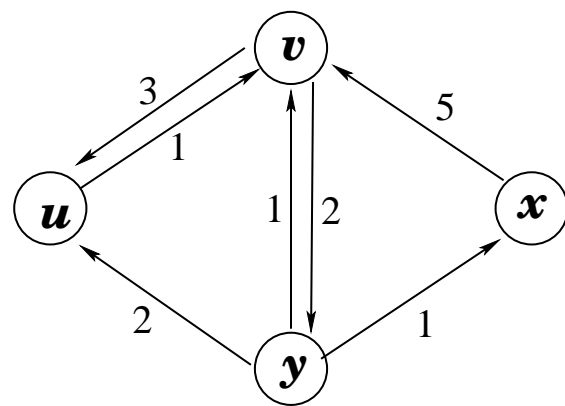
1. Residual network G_f is (V, E_f) , where

$$E_f = \{(u, v) \mid c_f(u, v) > 0\}.$$

2. Note that $|E_f| \leq 2|E|$.



Flow f



Residual Network

3. When is an edge $(u, v) \in E_f$?

- if $(u, v) \in E$ and $f(u, v) < c(u, v)$, or
- if $(v, u) \in E$ and $f(v, u) > 0$.

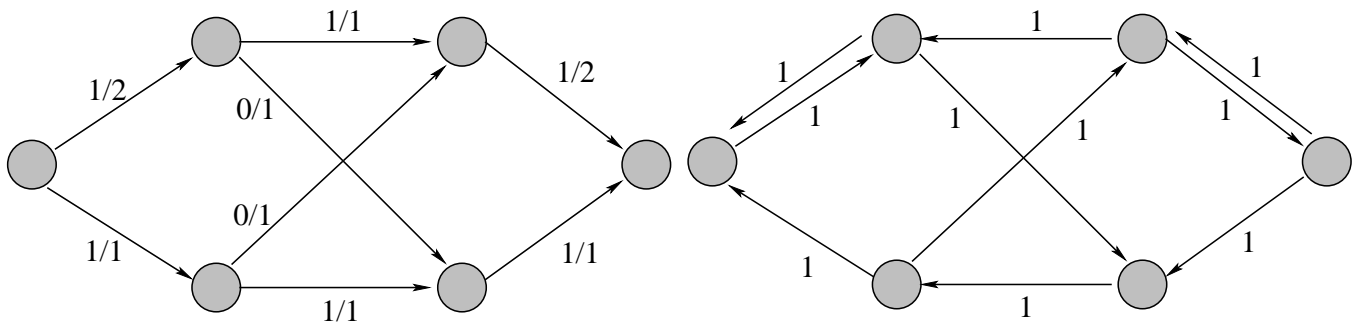
Augmentation

1. An augmenting path p is a simple path in G_f from s to t .

2. Residual capacity of p :

$$c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}.$$

3. By definition, $c_f(p) > 0$.



Flow f

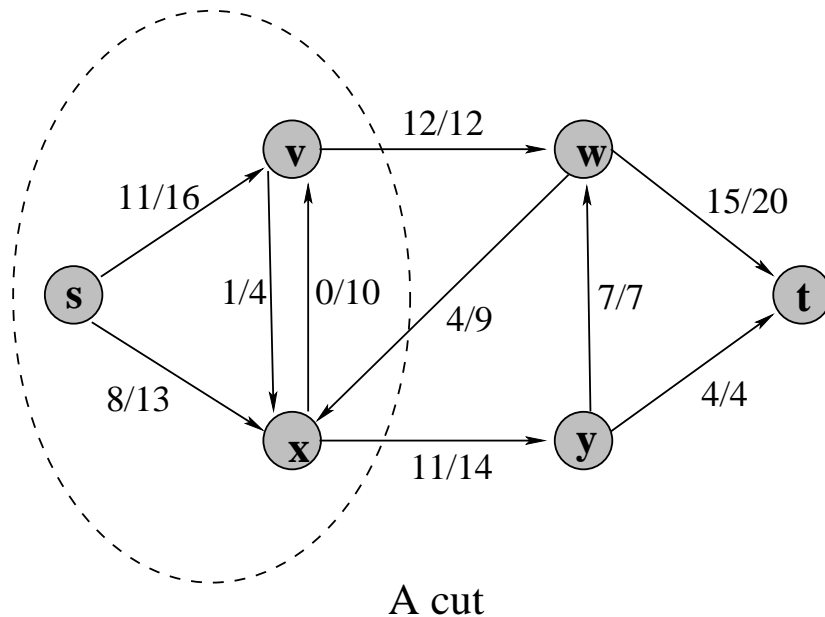
Residual Network

4. Can you spot an augmenting path?

5. How to prove no more flow possible?

Cuts

1. A cut (S, T) is a partition of V with $s \in S$ and $t \in T$.
2. $f(S, T)$ is the net flow across cut (S, T) .
3. $c(S, T)$ is the max capacity of cut (S, T) ; use only forward edges.



4. In this example, $f(S, T) = 12 - 4 + 11 = 19$.
5. $c(S, T) = 12 + 14 = 26$.

Flows and Cuts

- No matter how you separate s and t , the net flow across (S, T) is exactly $|f|$.

Cut Lemma: *If f is a flow and (S, T) a cut in G , then $f(S, T) = |f|$.*

Proof:

$$\begin{aligned} f(S, T) &= \sum_{v \in S, w \in T} f(v, w) \\ &= \sum_{v \in S, w \in V} f(v, w) - \sum_{v \in S, w \in S} f(v, w) \\ &= \sum_{v \in S, w \in V} f(v, w) \\ &= |f| \end{aligned}$$

- Use $\sum_{v \in S, w \in S} f(v, w) = 0$ since each (u, v) is counted twice canceling itself out.

Maxflow-Mincut Theorem

Theorem: *The following are equivalent:*

1. f is a maxflow.
2. No augmenting path in G_f .
3. $|f| = c(S, T)$ for some cut (S, T) .

Proof:

- (1) \Rightarrow (2). Aug. path increases f .
- (2) \Rightarrow (3). Define *reachable set*

$$S = \{v \mid \exists \text{ path from } s \text{ to } v \text{ in } G_f\}$$

$T = V - S$. Then, $t \in T$ —because no augmenting path. For any $u \in S$, $v \in T$, $f(u, v) = c(u, v)$; otherwise, $(u, v) \in E_f$, and v will be reachable from s . Thus, $|f| = f(S, T) = c(S, T)$.

- (3) \Rightarrow (1). By Cut Lemma, $|f| = f(S, T) \leq c(S, T)$. So, equality means flow is a maxflow.

Ford Fulkerson Algorithm

1. for each edge $(u, v) \in E$ do
2. $f(u, v) = f(v, u) = 0$;
3. while there is a path p in G_f do
4. $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
5. for each $(u, v) \in p$ do
6. $f(u, v) = f(u, v) + c_f(p)$
7. $f(v, u) = -f(u, v)$
8. end
9. end.

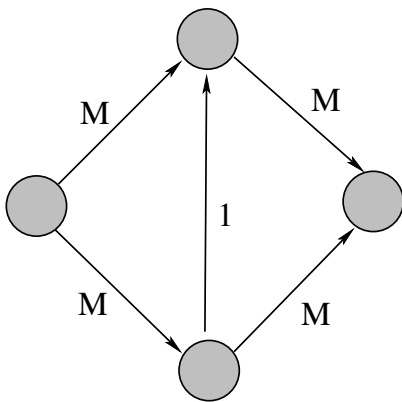
Example of FF

Analysis of Ford-Fulkerson

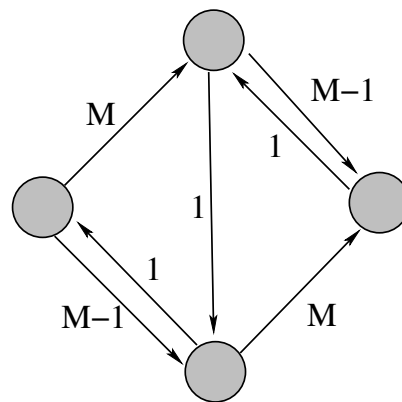
- Correctness follows from Mincut Maxflow theorem.
- BFS to find augmenting path takes $O(m)$ time.
- If all edge capacities are integers, with max capacity U , then number of augmentation is at most nU . Why
- Because $c(s, V - \{s\}) \leq nU$.
- FF increases flow by at least 1 unit in each augmentation, so running time is $O(nmU)$.
- However, if the capacities are irrational, the algorithm may not even terminate!

A Pathological Case for FF

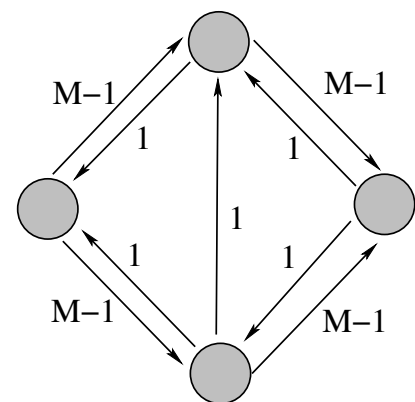
- Algorithm can select augmenting paths $s - a - b - t$ and $s - b - a - t$ alternatively M .
- M can be arbitrarily large, say, 10^9 .



Initial Network



After augmenting along path $s - a - b - t$

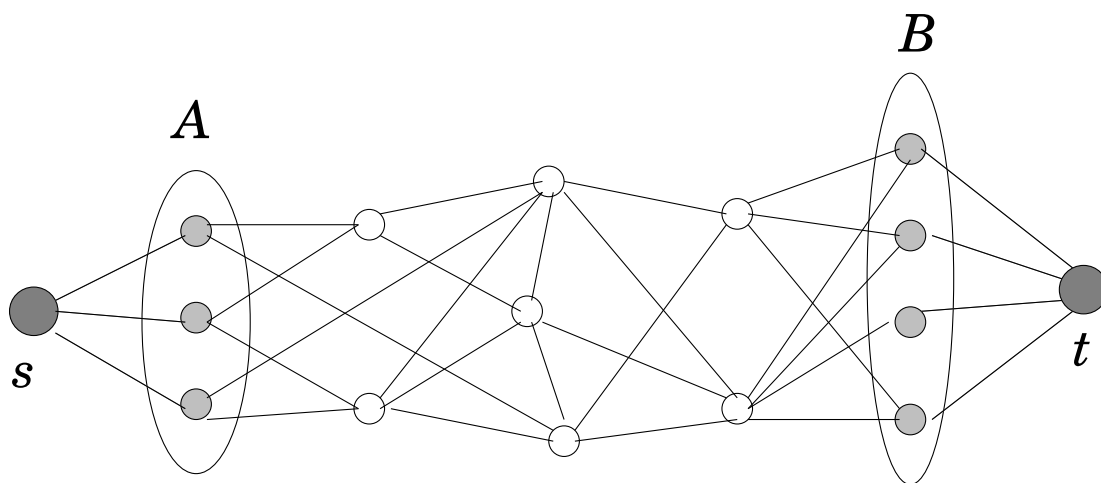


After augmenting along path $a - b - a - t$

Implications of Mincut Maxflow

Menger's Theorem (Edge). *Graph $G = (V, E)$, $A, B \subset V$. The min number of edges needed to separate A from B equals the max number of (edge) disjoint A - B paths.*

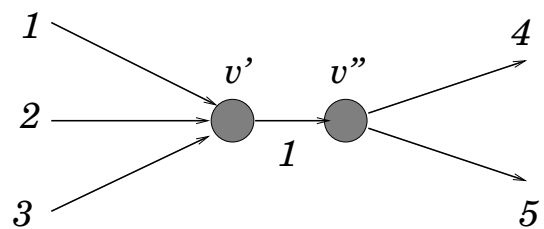
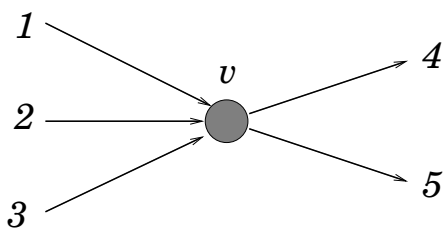
- Add nodes s, t ; join s to all of A ; t to all of B . These edges have capacity ∞ . All others have capacity 1.
- Max flow equals number of edge-disjoint paths from s to t .
- Min cut equals number edges that must be removed to disconnect A from B .



Implications of Mincut Maxflow

Menger's Theorem (Vertex) *Graph $G = (V, E)$, $A, B \subset V$. Then, the min number of vertices needed to separate A from B equals the max number of (vertex) disjoint A - B paths.*

- Transform G as follows.
- Split v into v', v'' . Direct all in-edges to v' , and all out-edges from v'' .
- Capacity of edge (v', v'') is 1. All others ∞ .



- Now, node disjoint paths in G correspond to edge disjoint paths in G' and vice versa.

Polynomial Algorithms

- FF method does not prescribe how to pick augmenting paths.
- Edmonds and Karp proposed 2 heuristics: *shortest path first*, and *max capacity augmentations*.
- Heuristics themselves are somewhat “obvious;” cleverness lies in their analysis.
- Shortest Path augmentation always augments flow along the shortest path from s to t .
- Let $\delta_f(s, v)$ denote the shortest path distance from s to v in the current residual graph G_f .
- Standard BFS algorithm will compute $\delta_f(s, t)$, and the path, in $O(m)$ time.

Shortest Path Augmentation (SPA)

Dist Lemma: Distances $\delta_f(s, v)$ increase monotonically throughout the algorithm.

Theorem: SPA algorithm performs $O(nm)$ augmentations.

Proof:

- Given an augmenting path p , call $(u, v) \in p$ critical if $c_f(p) = c_f(u, v)$.
- How many times can the same edge become critical?
- By SPA property, when (u, v) first becomes critical, $\delta_f(s, v) = \delta_f(s, u) + 1$.
- After augmentation, (u, v) is removed. It can reappear only after (v, u) is on some augmentation path.
- If f' is the existing flow when (v, u) becomes part of augmenting path:

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$$

Shortest Path Augmentation (SPA)

- Since $\delta_f(s, v) \leq \delta_{f'}(s, v)$, by **Dist Lemma**,

$$\begin{aligned}\delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2\end{aligned}$$

- Thus, $\delta_f(s, u)$ must increase by at least 2 before it can become critical again.
- Since $\delta_f(s, u)$ is at most $n - 2$, one edge can become critical $O(n)$ time.
- Total number of critical pairs is $O(nm)$, which is also the bound on number of augmentations—each augmentation has at least one critical edge.
- Since each augmentation can be done in $O(m)$ time, worst-case complexity of SPA is $O(nm^2)$.

Max Capacity Augmentation (MCA)

- For a path $p = (e_1, e_2, \dots, e_k)$, path capacity $c(p) = \min\{c(e_1), c(e_2), \dots, c(e_k)\}$.
- MCA always augments along the path of maximum capacity.
- How would you find a max capacity path in G_f ? Time complexity $O(m \log n)$.
- Starting with initial flow $f = 0$, how many augmentations will be done?

Flow Decomposition

F-D Lemma: *Starting with zero flow, one can construct the optimal flow f^* in at most m steps, where each step increases the flow along one path in the original graph G .*

Proof:

- Let G^* be graph induced by f^* .
(That is, $(v, w) \in E^*$ iff $f^*(v, w) > 0$.)
- Initialize $i = 0$. repeat:
 1. Find p_i from s to t in G^* .
 2. Set $\Delta_i = \min\{f^*(v, w) \mid (v, w) \in p_i\}$.
 3. Decrease $f^*(v, w)$ by Δ_i for all $(v, w) \in p_i$.
 4. Delete (v, w) from G^* if its flow is zero.
 5. $i = i + 1$.
- Each loop execution deletes at least one edge, so at most m iterations.
- Backward reconstruction of f^* : push Δ_i flow along each p_i .

Max Capacity Augmentation (MCA)

Theorem: Assuming integer capacities (max edge capacity C), MCA does $O(m \log C)$ augmentations.

Proof:

- Let f^* be maxflow, and f the current flow.
- $f' = f^* - f$ is a flow in G_f , and by F-D Lemma, we can reach f^* from f using $\leq m$ path augmentations.
- So, the max capacity path has residual capacity $\geq \frac{|f^*| - |f|}{m}$. (Why?)
- Look at $2m$ consecutive MCAs. One of them must augment by $\leq \frac{|f^*| - |f|}{2m}$. (Why?)
- In $2m$ augmentations, MCA capacity shrinks by $\frac{1}{2}$.
- Initial MCA capacity $\leq C$, and final is ≥ 1 .
- At most $\lceil \log C \rceil$ iterations of $O(m)$ augmentations.

Max Capacity Augmentation (MCA)

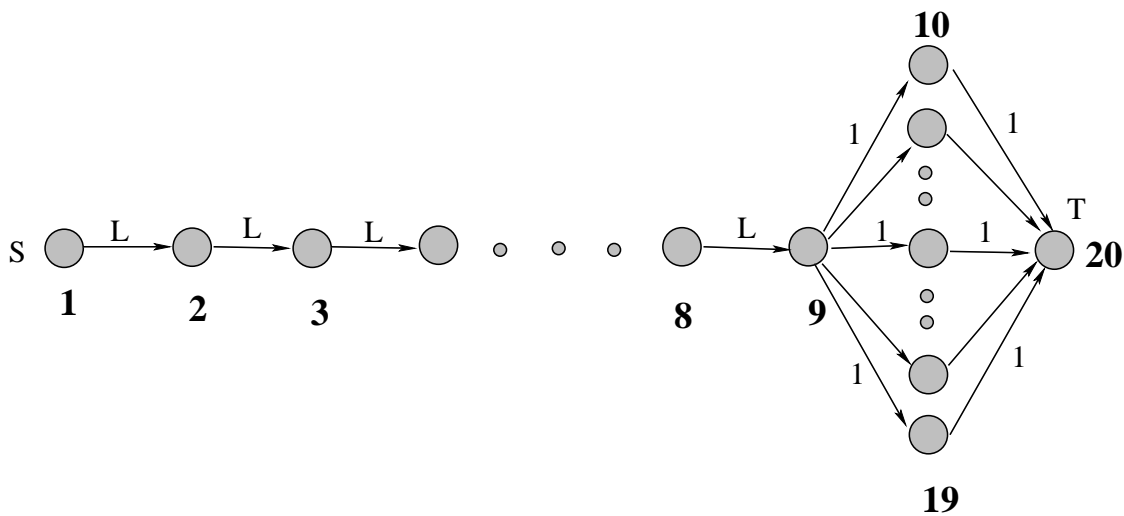
- Each MCA path can be found in $O(m \log n)$ time.
- MCA runs in worst-case time

$$O(m^2 \log n \log C).$$

- While sufficient for small values of C , this algorithm is not “strongly polynomial.”

Preflow-Push Paradigm

- Preflow is a fundamentally distinct style of algorithms.
- Leads to better running times, both in theory and practice.
- All previous algorithms are based on augmentation.
- A drawback of aug-based schemes: each augmentation takes $O(n)$ time. Multiple augmentations may partially share paths, but it's not exploited by algorithms.
- Example:



Preflow

- Augmentation algorithm maintain *flow* properties (conservation) and work towards optimality.
- Preflow algorithms allow temporary imbalance (excess) at nodes, and work towards *feasibility*.
- Formally, a preflow is a function $f : E \rightarrow R$ that satisfies capacity constraint and

$$\sum_j f_{ji} - \sum_j f_{ij} \geq 0, \quad \text{for all } i \notin \{s, t\}.$$

- For a given preflow, the excess at a node i is defined as

$$e(i) = \sum_j f_{ji} - \sum_j f_{ij}.$$

- Because no edge leaves t , $e(t) \geq 0$. Node s is the only one with $e(s) \leq 0$.

Distance Labels and Preflow

- Call a node active if it has +ve excess.
Source and sink never active.
- Preflow algorithms repeatedly select an active node and push its flow to neighbors. Which neighbors?
- Since we want to ultimately send flow to t , push flow to neighbors closer to t .
- Distance label $d(v)$ is the distance from v to t in the current residual graph.
- By the properties of shortest path:
 1. $d(t) = 0$.
 2. $d(i) \leq d(j) + 1$, for all $(i, j) \in G_f$.
- Call an edge (i, j) *admissible* if $d(i) = d(j) + 1$.
- Note that a shortest path from v to t must use *only admissible* edges.

Generic Preflow-Push

- **Initialize**

1. Set $f = 0$;
2. Compute distance labels $d(i)$, for all i .
3. $f_{sj} = c_{sj}$, for all $(s, j) \in E$.
4. $d(s) = n$.

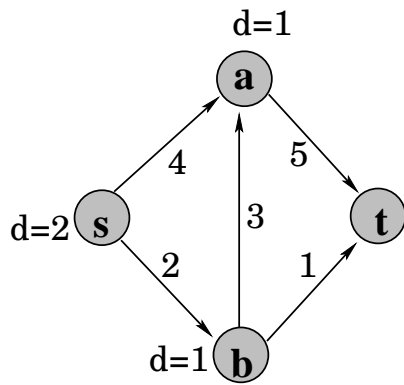
- **Push-Relabel (i)**

1. if there is an admissible edge (i, j) then
2. push $\min\{e(i), c_f(i, j)\}$ units on (i, j) .
3. else $d(i) = 1 + \min_{(i,j) \in E} \{d(j)\}$, $c_f(i, j) > 0$.

- **Generic-Preflow-Push**

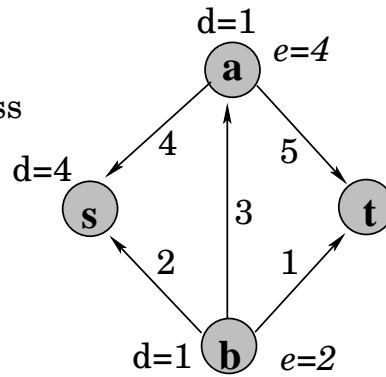
1. Initialize.
2. while exists an active node do
3. Select an active node i .
4. Push-Relabel (i)

Example

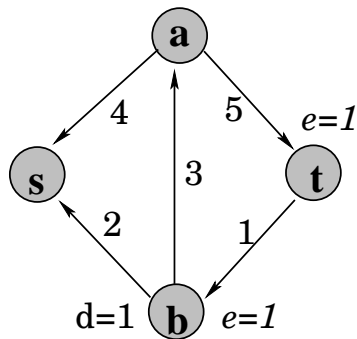


(i)

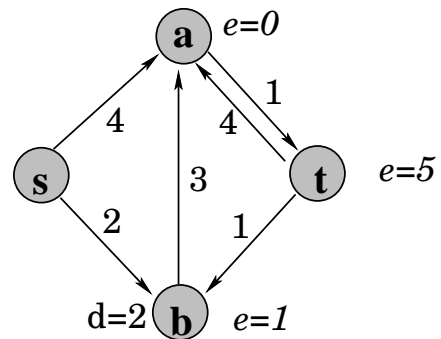
Preprocess



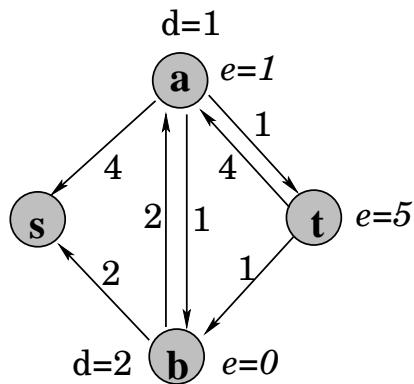
(ii)



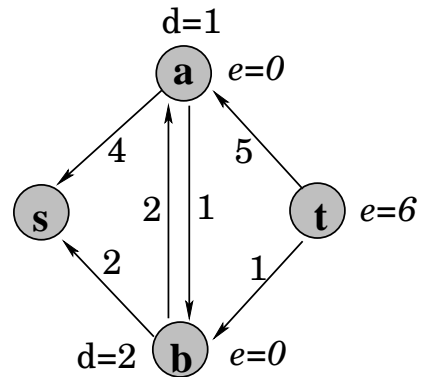
(iii)



(iv)



(v)



(vi)

Explanation

Defn. *Saturating vs. non-Saturating* pushes.

- (i) is the input network.
- (ii) shows effect of Initialization.
- Next, pick node b as active. Only (b, t) is admissible, so push 1 unit of flow. (A saturating push.) Result shown in (iii).
- Select node b again. No admissible edge incident, so Relabel distance as $d(b) = 1 + \min\{d(s), d(a)\} = 2$.
- Select node a . Edge (a, t) admissible, so push 4 units. (A non-saturating push.) Result shown in (iv).
- Select node b , and do a non-saturating push of 1 to a . Result shown in (v).
- Select node a , and do a saturating push of 1 on edge (a, t) . Result shown in (vi).

Analysis of Preflow

- Correctness.

1. Upon termination, the algorithm has maxflow. Why?
2. Since only s or t can have excess, the preflow is actually a flow.
3. Since $d(s) = n$, there is no path from s to t in G_f , so by augmentation theorem, the excess at t must equal maxflow. (Hint: Show that if f is a preflow, then there is no path from s to t in G_f .)

- Time Complexity.

1. Key proof components are to show (i) distance labels are always valid, and (ii) they do not increase too often.
2. Note that we do not recompute $d()$ s; we just locally update them, so validity needs proof.

Complexity of Preflow

Path Lemma. *Source node s is reachable from any node i with +ve excess in the current residual graph.*

Proof.

- By F-D theorem, a preflow can be decomposed into 3 parts: (a) flow along paths from s to t , (b) flow along paths from s to some excess nodes, and (c) flow along cycles.
- The excess at node i is unaffected by flows along cycles or flows along $s-t$ paths (passing through i).
- So, there must be a flow along some path from s to i .
- The residual network now contains the reversal of this path, and so s is reachable from i .

Complexity of Preflow

Label Lemma 1. $\forall i$, we have $d(i) < 2n$.

- Consider the last time i is relabeled. At that point, i has excess, and so has a path to s .
- Since $d(s) = n$, the path has at most $n - 2$ edges, and $d(k) \leq d(j) + 1$, for all edges (k, j) in the path, we get that $d(i) \leq d(s) + n - 2 < 2n$.

Relabeling Lemma. *Total number of Relabel operations is at most $2n^2$.*

Saturating Pushes. *Total number of Saturating Pushes is at most nm .*

Proof similar to E-K heuristic.

Complexity of Preflow

Non-Saturating Pushes. *Total number of non-Saturating Pushes is at most $O(n^2m)$.*

- Let I be the set of active nodes at any time.
- Use $\Phi = \sum_{i \in I} d(i)$ as potential.
- Since $|I| < n$, and $d(i) < 2n$, initially $\Phi < 2n^2$. Upon termination, $\Phi = 0$. During a Push-Relabel (i), one of the following occurs:
 - **Case 1.** No admissible edge found. In this case, $d(i)$ increases by $k \geq 1$. Φ also increases by k . Since $d(i)$ increases by at most $2n$, total increase to Φ caused by this operation is $2n^2$.

Complexity of Preflow–Contd.

- **Case 2.** Admissible edge (i, j) exists, and a push occurs. If the push is saturating, it may create a new excess node j , increasing Φ by $d(j)$. There are at most nm such pushes, each increasing Φ by at most $2n$, so total contribution is $O(n^2m)$ over all saturating pushes.

A non-saturating push does not change I , but it decreases Φ by $d(i)$, since i no longer has excess. But it also increases Φ by $d(j) = d(i) - 1$ if the push causes j to become a new excess node. In any case, the decrease in Φ is at least 1 per non-saturating pushes.

- In summary, Φ is initially at most $2n^2$. It is incremented by at most $2n^2 + 2n^2m$. Each non-saturating pushes decreases Φ by at least 1, so there are at most $O(n^2m)$ non-saturating pushes.

Theorem: *Generic Preflow-Push algorithm runs in worst-case time $O(n^2m)$.*

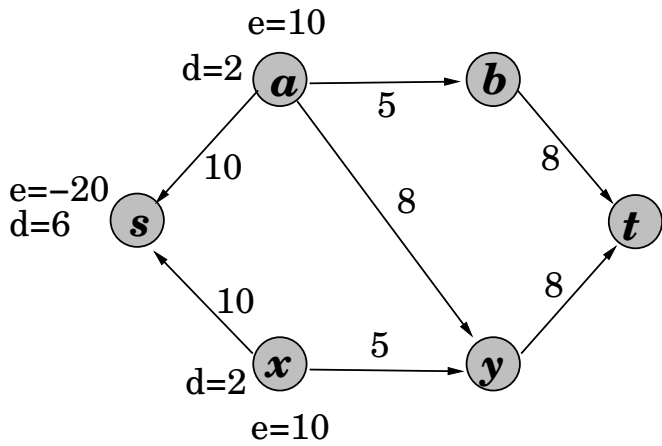
Improvements to Generic Preflow-Push

- Generic Preflow-Push's performance is comparable to shortest augmenting path algorithm.
- The preflow-push is flexible and can be further optimized.
- Different rules for choosing the active nodes lead to modified algorithms, with improved performance.
- The book describes 3 such heuristics:
 1. [FIRST-IN FIRST-OUT.] $O(n^3)$.
 2. [HIGHEST LABEL.] $O(n^2 m^{1/2})$.
 3. [EXCESS SCALING.] $O(nm + n^2 \log U)$.
- We will discuss FIFO only.

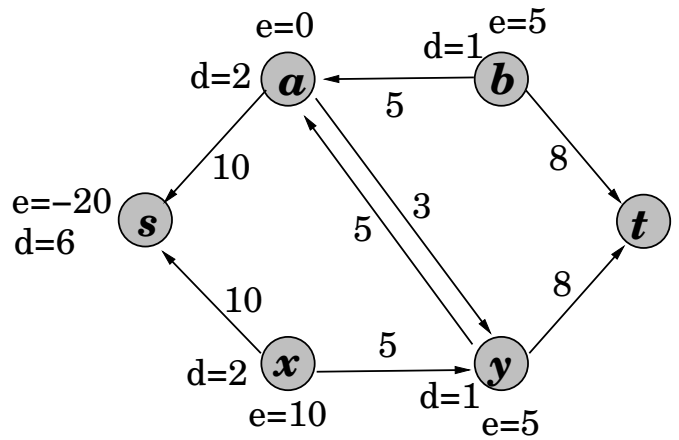
FIFO Preflow-Push

- Suppose Generic Preflow-Push picks node i and the push is *saturating*.
- Node i may be still active, but Generic need not pick it in next iteration.
- In FIFO, we keep pushing from i until its excess is zero or a relabel occurs.
- Call consecutive pushes at i , without relabeling, a *node examination*.
- FIFO initially orders nodes in a queue LIST, and always examines nodes in FIFO order.
- The algorithm examines the node at the head of LIST; newly active nodes put at the rear.
- When a relabel occurs, that node is placed at the rear of LIST.

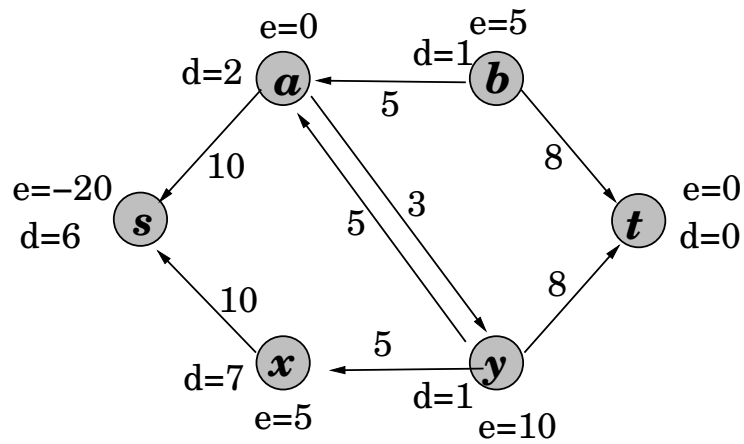
Example



(i)



(ii)



(iii)

Explanation

- The preprocessing step creates an excess of 10 units at a and x .
- Let $LIST = \{a, x\}$. Algorithm examines node a . Performs a saturating push of 5 units on (a, b) , and a non-sat push of 5 units on (a, y) . Figure (ii).
- Nodes b and y become active, and added to $LIST$ to give $\{x, b, y\}$.
- Next, examine x , do a saturating push of 5 units on (x, y) , followed by a relabel of node x , which makes x 's distance label 7. It then puts x at the end of the $LIST$ to get $\{b, y, x\}$. Figure (iii).
- Continue.

Analysis of FIFO Preflow

- Partition node examinations into different phases:
 1. Nodes that become active in Preproc.
 2. Nodes that are in queue after algorithm has examined nodes in phase 1.
 3. Nodes that are in queue after algorithm has examined nodes in Phase 2.
 4. So on...
- In the example, Phase 1 examines nodes $\{a, x\}$. Phase 2 examines nodes $\{b, y, x\}$.
- Total number of phases at most $2n^2 + n$.
(to be proved)
- Each phase examines a node at most once, and each node examination does at most one non-saturating push.
- Because non-saturating pushes are the bottleneck in the complexity, this gives a total bound of $O(n^3)$.

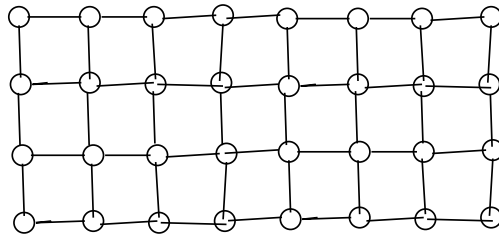
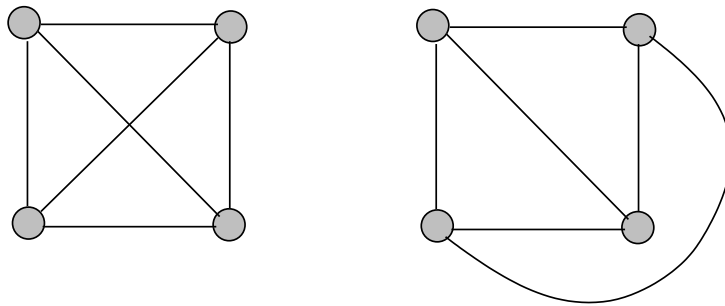
Analysis of FIFO Preflow

- Argument uses potential function $\Phi = \max\{d(i) \mid i \text{ is active}\}$.
- We consider total change to Φ during a phase.
 1. At least one relabel during the phase. Then, Φ increases by at most the max distance value. Over all phases, total increase in Φ at most $2n^2$.
 2. No relabel during the phase. Then, the excess of every node that was active at the start of phase moves to nodes with smaller dist label. So, Φ decreases by at least 1.
- Combining the two cases above, at most $2n^2 + n$ phases, since the second case can occur at most n time, the max initial value of Φ .

Theorem. *FIFO preflow-push runs in worst-case time $O(n^3)$.*

Special Topics

- Network flow in *planar* graphs.
- A graph is planar if it can be drawn in the plane without any two edges crossing.

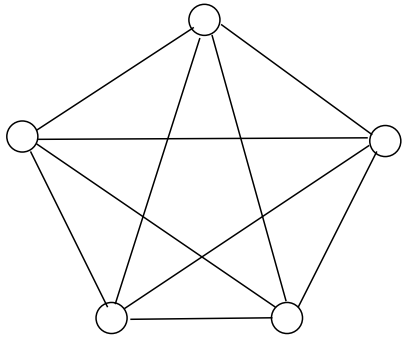


Planar Graphs

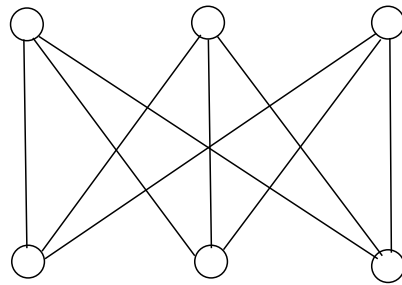
- The top-left graph is planar, as it can re-drawn without edge crossings.
- In some applications, networks are naturally planar, and their special structure leads to better flow algorithms.

Planar Graphs

- Determining whether a graph is planar is not easy.
- Kuratowski's Theorem: A graph G is planar if and only if it does not contain a K_5 or $K_{3,3}$ as a topological minor.



K_5

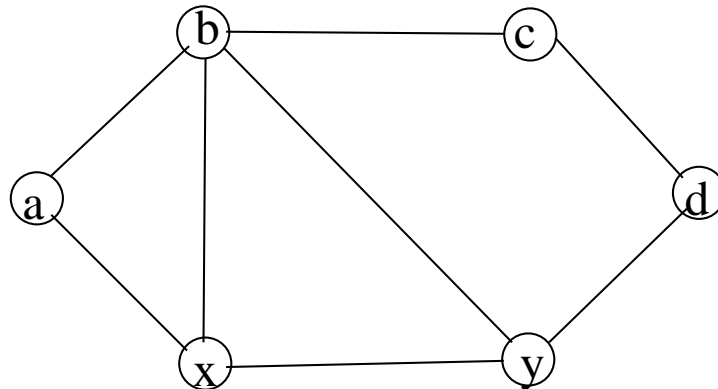


$K_{3,3}$

- This characterization does not give very efficient algorithm.
- Hopcroft-Tarjan developed a linear time algorithm to decide if G is planar.
- Fary's Theorem: Every planar graph has a **straight line** embedding.

Properties of Planar Graphs

- When drawn in the plane, a planar graph divides the plane into **faces**.
- Two faces are adjacent if they share an edge.
- In this example, faces abx and bxy adjacent.

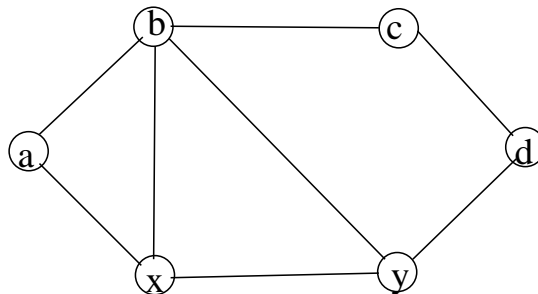


- The outer face is called **unbounded** face.
- A planar graph $G = (V, E)$ has interesting relationship between number of vertices, edges, and faces.

Euler's Formula

Theorem: If G has n vertices, m edges, and f faces, then $n - m + f = 2$.

- **Proof by induction.** Case $f = 1$ satisfies $m = n - 1$: a graph with one face is a tree.
- Suppose it holds for graphs with $\leq k - 1$ faces, and consider G with k faces.
- Pick an edge (i, j) common to two faces, and remove it. The two faces get merged, reducing both m and f by 1.
- The reduced graph has n vertices, $k - 1$ faces, and $m - 1$ edges. By induction, $k - 1 = (m - 1) - n + 2$.
- Adding (i, j) gives $k = m - n + 2$.



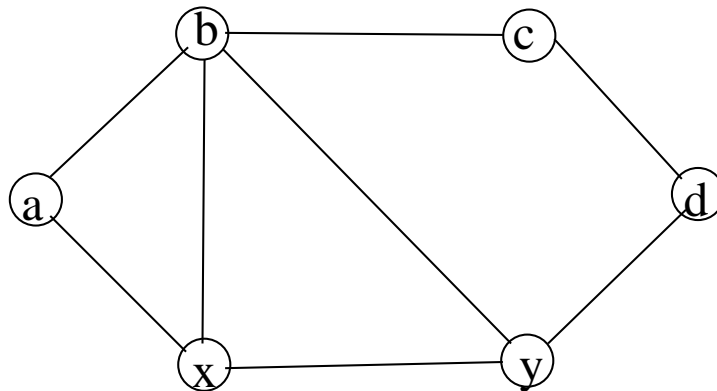
Euler's Formula

Theorem: In a planar graph, $m \leq 3n - 6$.

- The boundary of each face has ≥ 3 edges.
- Traversing all faces, we count $\geq 3f$ edges.
- Each edge counted at most twice, since an edge on two faces.
- Thus, $3f \leq 2m$. Plug it in Euler's Formula:

$$\begin{aligned} m &= n + f - 2 \\ &\leq n + 2m/3 - 2 \\ &\leq 3n - 6 \end{aligned}$$

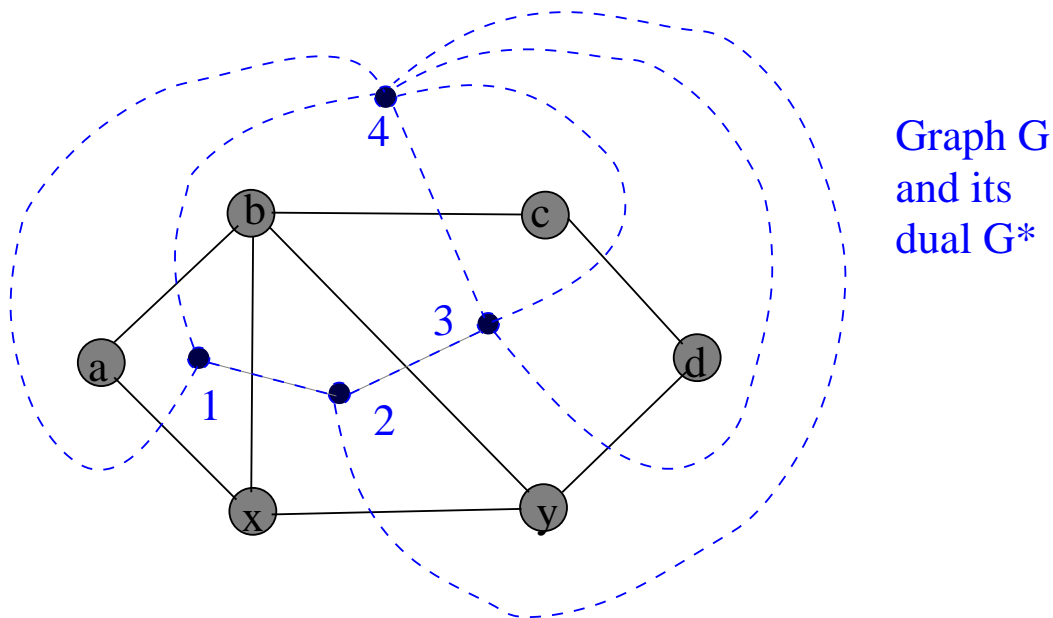
- Similar argument shows $f \leq 2n - 4$.



Flows in Planar Graphs

- Let $G = (V, E)$ be an undirected, planar graph.
- By Euler's Formula, $m < 3n$, so that by itself guarantees an improved time complexity: $O(nm \log n)$ becomes $O(n^2 \log n)$.
- Our new algorithm will achieve $O(n \log n)$ for min cut and maxflow in planar graphs.
- Idea is to reduce the flow problem to a **shortest path** problem.

Dual of a Planar Graph

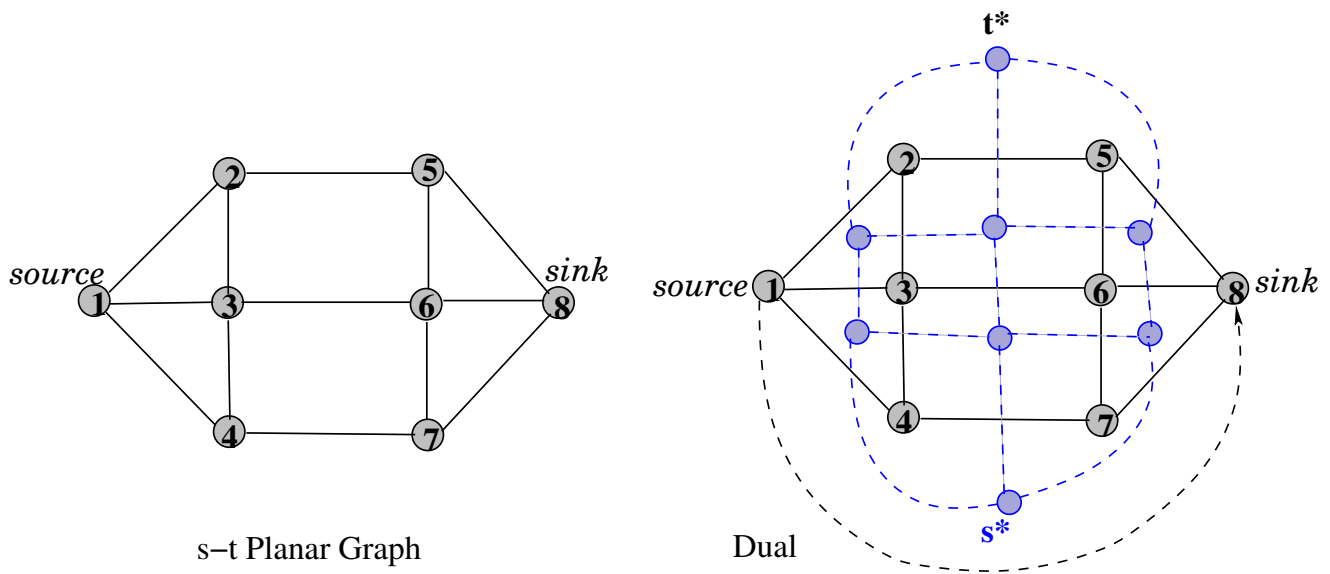


Graph G
and its
dual G^*

- Every planar graph G has an associated “twin,” called the dual graph G^* .
- Put a vertex f^* in each face f of G .
- If edge (i, j) of G is common to faces f_1 and f_2 , put an edge between vertices f_1^* and f_2^* .
- If an edge (i, j) is only on one face, it results in a loop.
- An easy observation: the dual G^* has $n^* = f$ vertices and $f^* = n$ faces, and the formula $n^* - m^* + f^* = 2$ still holds.

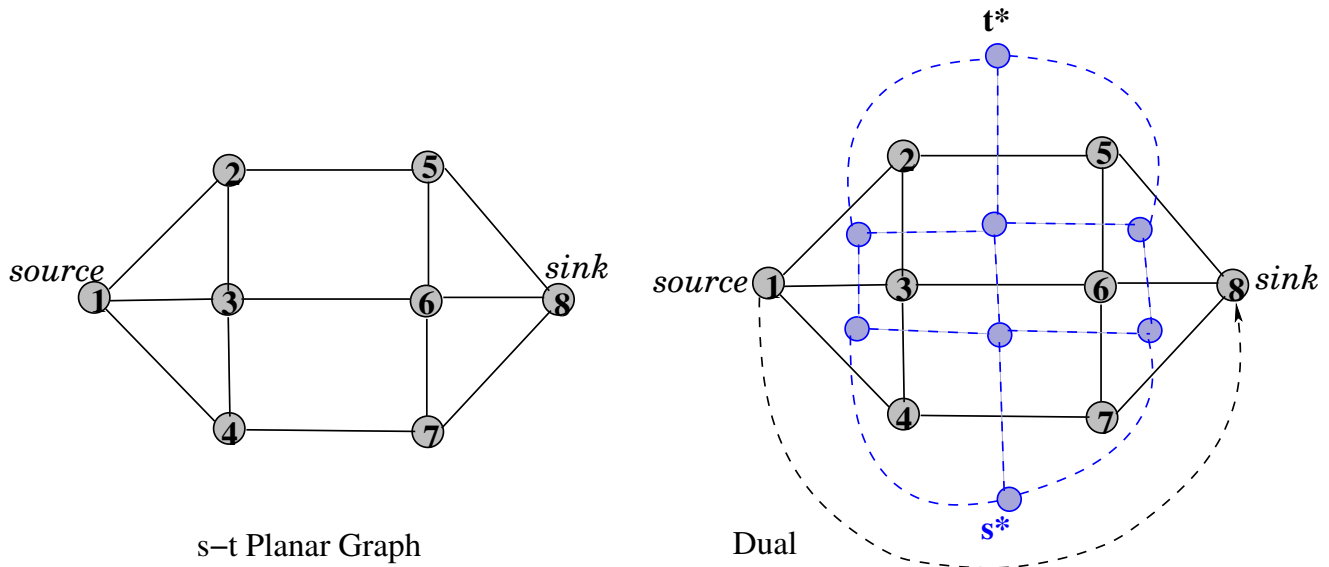
$s-t$ Planar Graphs

- The dual of G^* is G itself.
- A cycle in G^* corresponds to a cut in G .
Example: $(4, 3, 2, 4)$.
- We deal with $s-t$ planar graphs. G is $s-t$ planar if s and t both lie on the outer (unbounded) face.



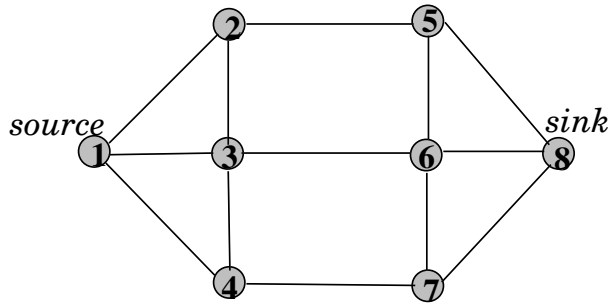
- This graph is $s-t$ planar for $s = 1, t = 8$, but not for $s = 1, t = 6$.

$s-t$ Planar Graphs

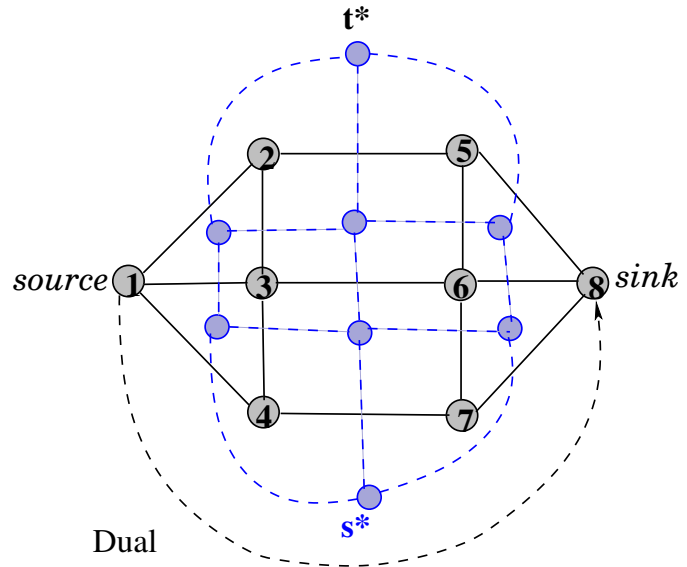


- Draw edge (s, t) ; does not violate planarity.
- Construct dual G^* . Make the node corresponding to unbounded face t^* , and the one for the new face s^* . Don't put the dual edge (s^*, t^*) .
- Make the **cost** of edge (i, j) equal to the **capacity** of the corresponding edge in G .
- An $s-t$ cut in G corresponds to a s^*-t^* path in G^* . The capacity of former equals the cost of latter.

$s-t$ Planar Graphs



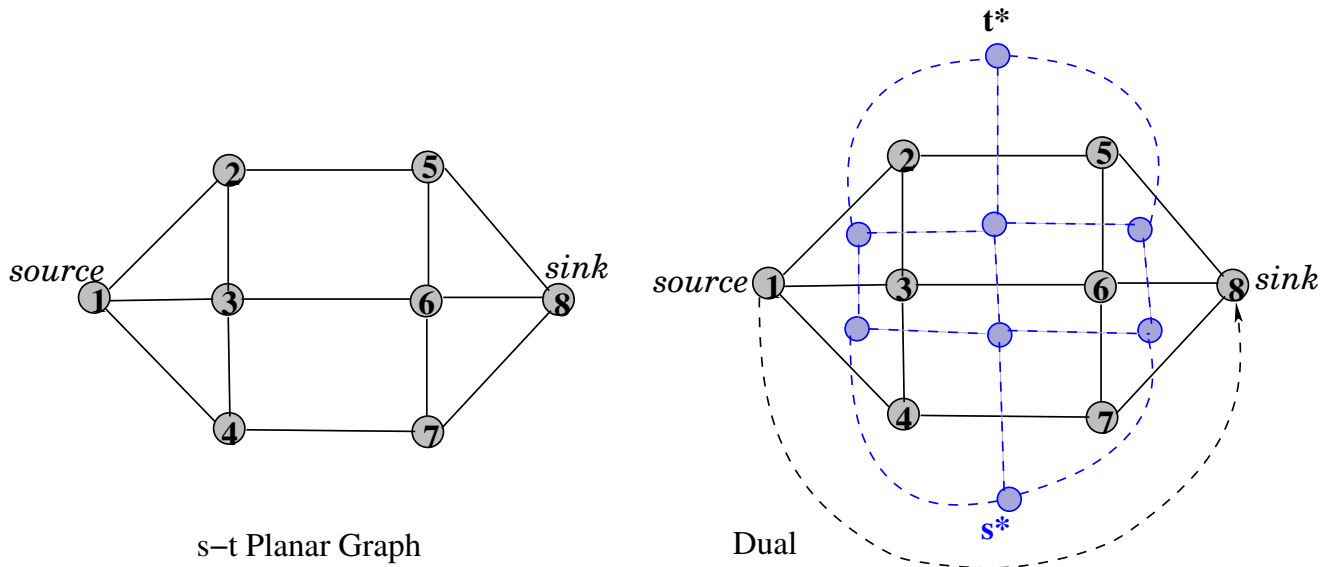
s-t Planar Graph



Dual

- Thus, a min cut in G can be computed by computing a shortest path in G^* .
- Motivation for adding extra node s^* is to convert a cycle problem into a path problem.
- The cut however does not by itself give the max flow.
- We now show the surprising fact: **shortest path distances in G^* can be used to obtain the maxflow.**

$s-t$ Planar Graphs



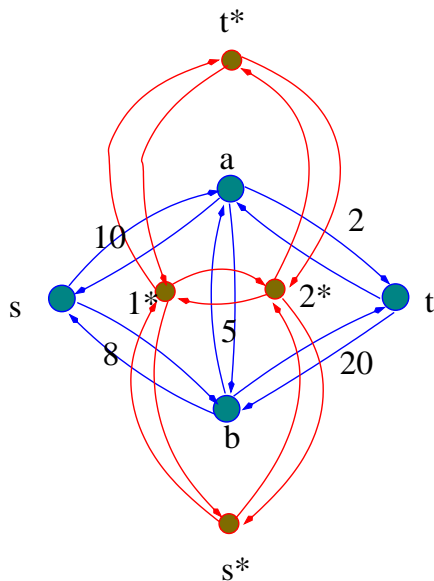
- If $d^*(j)$ is the SP distance from s^* to j^* in G^* , then $d(j^*) \leq d(i^*) + cost_{i^*j^*}$, for all $(i^*, j^*) \in E^*$.
- Let (i, j) be the edge in G corresponding to (i^*, j^*) . Define $f_{ij} = d(j^*) - d(i^*)$.
- G is undirected, so $-ve$ flow along (i, j) is positive flow along (j, i) .
- By definition, $f_{ij} \leq cost_{i^*j^*} = c(i, j)$. Thus, these flow values satisfy the capacity constraint. Next, show flow conservation.

$s-t$ Planar Graphs

- A node $k \neq s, t$, defines a cut $Q = (\{k\}, V - \{k\})$. Look at corres. cycle W in G^* .
- Clearly, $\sum_{(i^*, j^*) \in W} (d(j^*) - d(i^*)) = 0$, because of term cancellation.
- Thus, $\sum_{(i, j) \in Q} f_{ij} = 0$.
- The flow is maximum because let P^* be the shortest path from s^* to t^* . By definition of P^* , $d(j^*) - d(i^*) = cost_{i^*j^*}$, for any $(i^*, j^*) \in P^*$. Since $cost_{i^*j^*} = c(i, j)$, we get $f_{ij} = c(i, j)$, for any $(i, j) \in Q$.
- Thus, the flow saturates all edges of a $s-t$ cut, and so it is maxflow.
- A maxflow in a $s-t$ planar graph can be found in $O(n \log n)$ time.

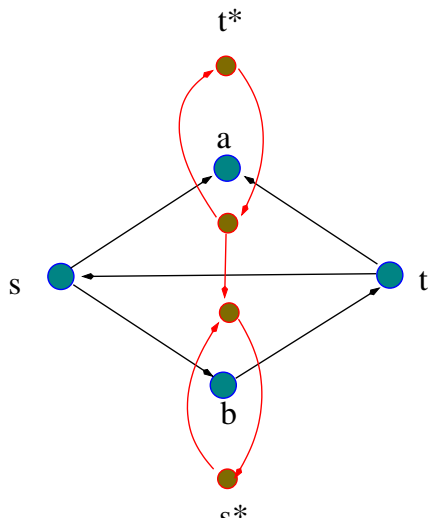
Subtleties

- In figuring out the direction of flows, use the convention that dual edge is directed so that the **primal source is on left**.



In dual graph, direct edge so that the source of primal edge is on left.

- If we start with a **directed** planar graph, and apply duality, the edge direction convention does not work. In this example, there is not even a path from s^* to t^* even though there is a flow.

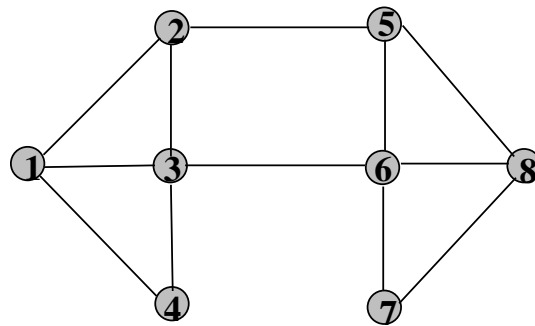


If we assume directed planar graph, and apply edge direction convention, it does not work.

There is no path from s^* to t^*

Min Cuts

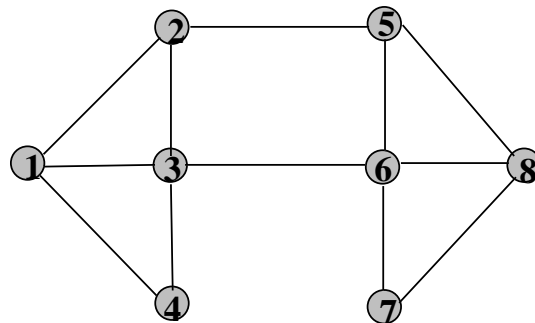
- $G = (V, E)$ an **undirected**, **multi-graph** with n vertices and m edges.
- Multigraphs can have multiple (parallel) edges between a pair (u, v) .
- A **cut** is a partition $(C, V - C)$ of the vertices of G into 2 non-empty sets.
- We refer to C as the cut, and its **size** is the number of edges crossing the cut $(C, V - C)$.
- We want to find a **minimum size** cut of G , called a **min cut**.



- In this example, mincut of 2.

Min Cuts

- Applications: network connectivity, reliability, TSP heuristics, databases partitioning.
- Traditionally, mincuts computed through maxflow algorithms.
- Maxflow algorithm finds a $s-t$ min cut.
- By trying different $s-t$ pairs, we can compute the global mincut.
- $\binom{n}{2}$ $s-t$ maxflow computations needed.
- $n - 1$ maxflow computations [Gomory-Hu]



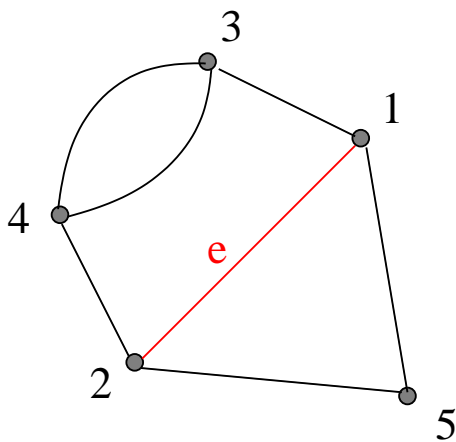
- Best maxflow algorithm takes about $O(nm \log n)$ time, so mincut takes $O(n^2m)$ time.

Min Cut vs. Max Flow

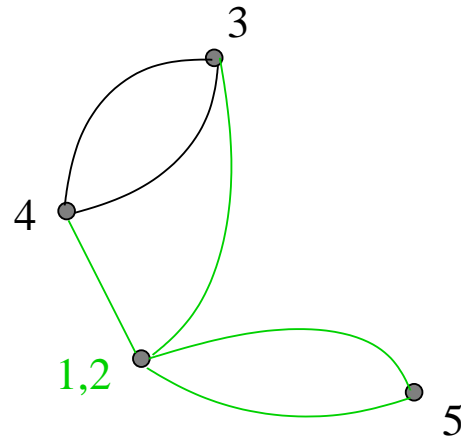
- **Open Question:** Can $s-t$ min cut problem be solved faster than the $s-t$ maxflow problem?
- A maxflow computation immediately yields mincut, but the converse does not seem to be true.
- We will discuss a remarkable result: There is a simple, randomized algorithm for computing global mincut in roughly $O(n^2)$ time.
- Thus, global mincut can be computed much faster than a **single** maxflow computation!
- Algorithm is by David Karger 1993. (Stanford PhD, now at MIT.)
- $O(n^2)$ algorithm complicated, so I describe the simpler, less efficient version. But ideas are interesting.

Edge Contraction

- Key step is the **contract** operation.
- Contracting edge (x, y) means **merging** x, y . Delete x, y ; put a new node z , and all edges incident to x, y (but not both) are made incident to z .



Contract e



Resulting Graph G'

- The contracted graph is denoted $G/(x, y)$.
- Contraction creates parallel edges, and **meta vertices**.
- Effect of contracting a set of edges F is independent of the **order** in which edges of F are contracted.

Contraction Algorithm

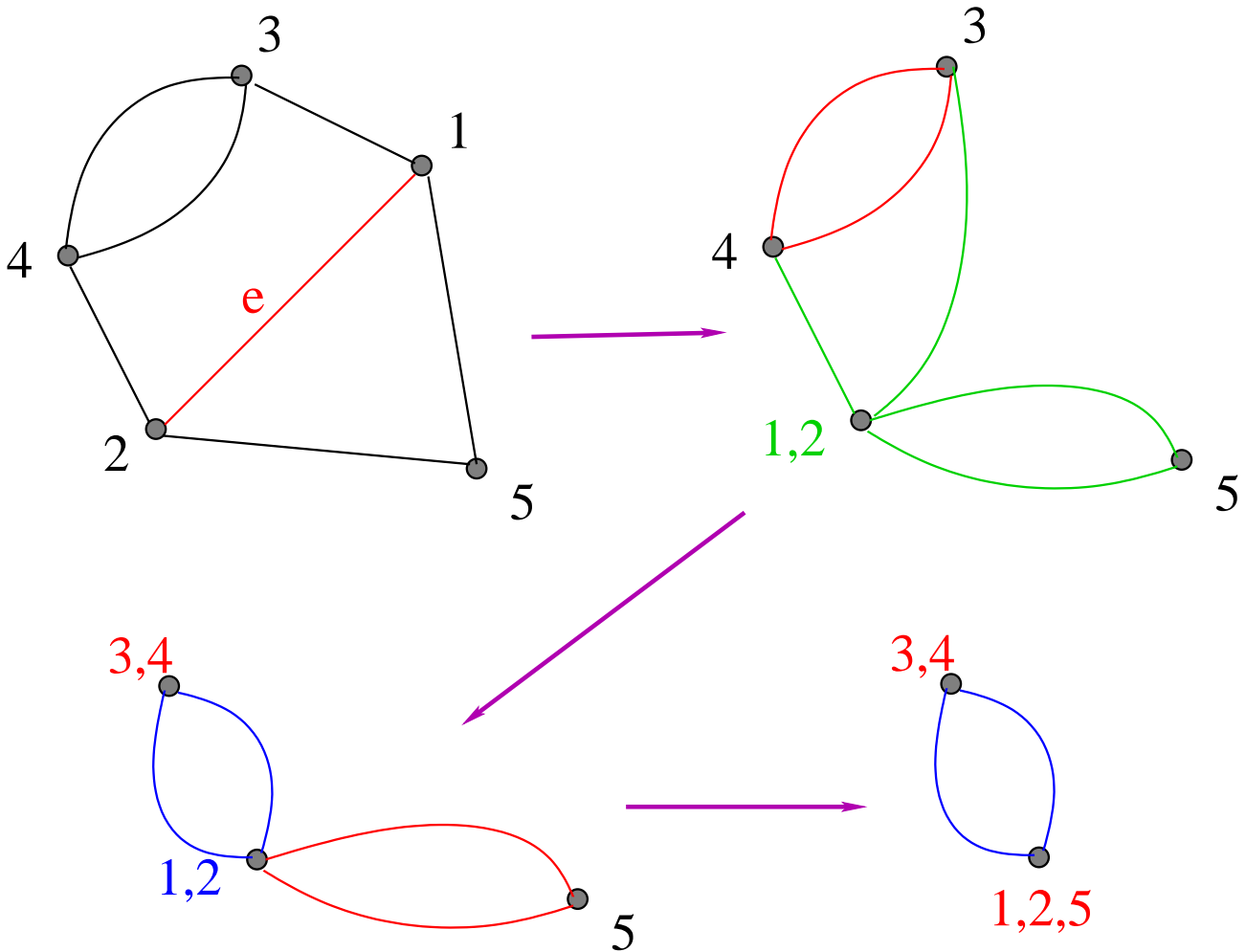
Input: A multigraph $G = (V, E)$.

Output: A cut C .

1. $H \leftarrow G$.
2. **while** H has more than 2 vertices **do**
 - **Uniformly at random** pick (x, y) in H .
 - $F \leftarrow F \cup \{x, y\}$.
 - $H \leftarrow H/(x, y)$.
3. **Output** $(C, V - C)$ as the set of vertices corresponding to two meta vertices in H .

Illustration

- **Contraction sequence:**
 $\{1, 2\}, \{3, 4\}, \{(1, 2), 5\}$.



- **Final mincut output:** $(\{3, 4\}, \{1, 2, 5\})$.

Analysis of Contraction

- With proper data structures, algorithm can be implemented in $O(n^2)$ time.
- Uniform random selection uses “multiplicity” of edges. **How?**
- Interesting part is to show that **Contraction** produces a mincut with **non-negligible probability**.
- **Easy claim: A cut C is produced as output by Contraction if and only if none of the edges crossing this cut is contracted by the algorithm.**
- If G has n vertices, and min cut value is k , minimum vertex degree is $\geq k$. Total number of edges in G is $m \geq nk/2$.
- **Min cut value in $G/(x, y)$ is at least as large as in G . **How?****

Analysis of Contraction

- Focus on a particular min cut K , and compute the probability that K is output by the algorithm.
- Vertex count of G drops by 1 in each iteration of Contraction.
- Let $n_i = n - i + 1$ be the vertex count at i th iteration.
- Suppose none of the edges in K are contracted in first $i - 1$ iterations.
- Since K is also a cut in H , the graph has mincut value k . Thus, H has $\geq n_i k / 2$ edges.
- Probability that an edge of K is contracted in this iteration is at most $k / (n_i k / 2) = 2 / n_i$.
- We now compute the prob. that no edge of K is ever contracted.

Analysis of Contraction

$$\begin{aligned}\Pr [\mathbf{K} \text{ output}] &\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n_i}\right) \\ &= \prod_{i=1}^{n-2} \left(1 - \frac{2}{n - i + 1}\right) \\ &= \prod_{j=n}^3 \left(\frac{j-2}{j}\right) \\ &= \frac{1}{\binom{n}{2}} \\ &= \Omega(n^{-2})\end{aligned}$$

- Any specific min cut K is output by algorithm with probability $\Omega(n^{-2})$.
- Use standard amplification technique: run the algorithm $O(n^2 \log n)$ times. K will be produced almost surely.

Extensions and Improvements

- An improved version, called **FastCut**, runs in $O(n^2 \log n)$ time, and produces a min cut with probability $\Omega(1/\log n)$.
- There are $O(n^2)$ different min cuts. (Earlier bound produces a specific min cut with probability $\Omega(n^{-2})$.)