This lecture notes is based on Chapter 7 in "Algorithm Design" by Kleinberg and Tardos.

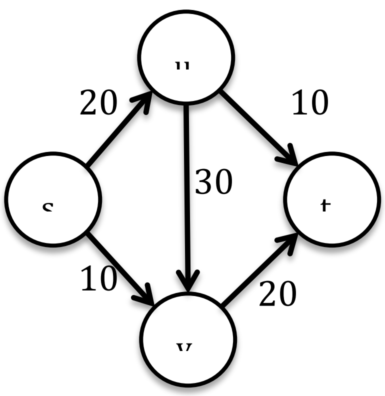
1. Flow networks

One often uses graphs to model transportation networks: networks whose edges carry some sort of traffic and whose nodes act as "switches" passing traffic between different edges, for example, a high system in which edges are highways, and nodes are interchanges (cities), or a computer network in which edges are links carrying packets and nodes are switches.

Definition 1. A *flow network* is a directed graph G=(V, E) with the following features:

1) associated with each edge e is a *capacity*, a non-negative number denoted as c_e;
2) there is a single source node s $\in$ V;
3) there is a single sink node t $\in$ V.

The other nodes than s and t are called *internal* nodes.



Definition 2. In a flow network G, a *flow* is a function f that maps each edge e to a non-negative real number, f: E→R^+; the value f(e) intuitively represents the amount of flow carried by edge e. A flow must satisfy the following properties:

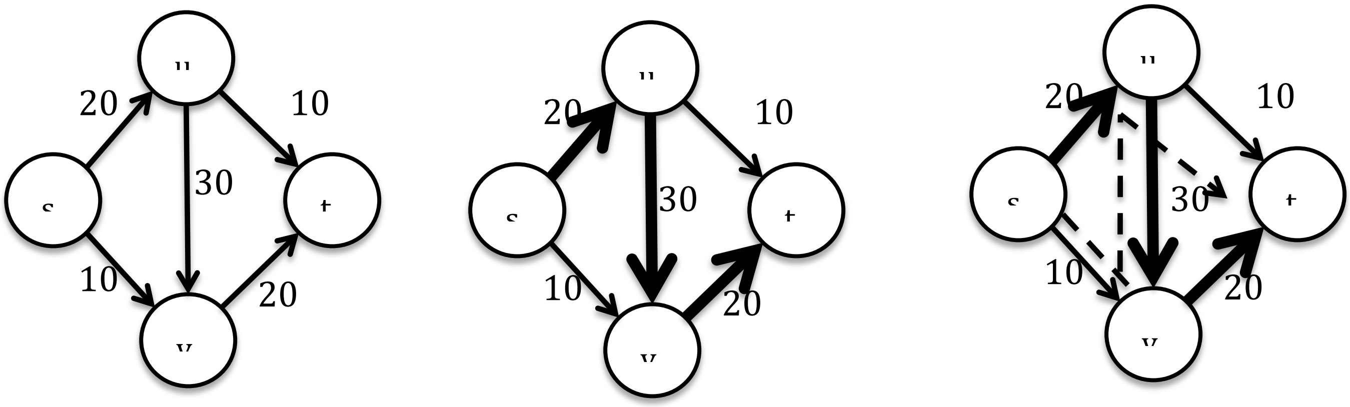
1) (Capacity condition) for each e $\in$ E, we have $0 \leq f(e) \leq c_e$;
2) (Conservation condition) for each node v other than s and t, we have $\sum_{e \in In(e)} f(e) = \sum_{e \in Out(e)} f(e)$, where In(e) and Out(v) represent the set of incoming and outgoing edges incident to v, respectively.

The value of a flow f, denoted by v(f), is defined as the amount flow generated at s, which is equal to the flow entering t: $v(f) = \sum_{e \in Out(s)} f(e) = \sum_{e \in In(s)} f(e)$; this is because

$$\sum_{e \in Out(s)} f(e) = \sum_{v \in V} \sum_{e \in Out(v)} f(e).$$

Maximum-flow problem. Given a flow network, find a flow of maximum possible value.

Design an algorithm: we would like to push as much as flow from s to t.



Definition 3. Given a flow network G and a flow f, a *residual network (or graph)* G_f with respect to f is defined as,

1) The set of vertices is the same as G;
2) For each edge e=(u,v) in G on which f(e)<c_e, there are c_e – f(e) "leftover" capacity; these edges are called the *forward* edges;
3) For each edge e=(u,v) in G on which f(e)>0, there are f(e) units of flow that we can "undo" (push backward) if we want to, thus we add an edge e'=(v,u) in G_f with a capacity of f(e); these edges are called the *backward* edges;

Let P be a simple s-t path (i.e. augmenting path) in G_f. We define bottleneck(P, f) as the minimum residue capacity of any edge on P, w. r. t. flow f. Below we define an algorithm to find a new flow f' w.r.t. a given flow f and a path P.

1) let b= bottleneck(P, f);
2) for each edge (u, v) ∈ P, if e is a forward edge, increase f(e) in G by b; otherwise decrease f(e') by b, where e'=(v,u);

Lemma 1. f' is a flow in G.
Proof: 1) each edge e in f' has capacity ≤ c_e because if e is a forward edge, f'(e)=f(e)+b ≤ f(e) + c_e – f(e) = c_e ; otherwise, f'(e)=f(e)-b≥0. 2). We only need to check the conservation condition on the internal nodes in P. Let v be such a node, the changes in the amount of flow entering v is the same as the changes of the amount of flow exiting v, depending on the types of incoming edge and outgoing edge (forward or backward).

Also because v(f')=v(f)+bottleneck(P, f), and bottleneck(P, f) > 0, v(f')>v(f). That means we always improve the flow by using the following algorithm.

Ford-Fulkerson algorithm for Maximum flow problem
Initially set f(e)=0 for all e in G
while there is a s-t path in G in the residue graph G_f

let P be a simple path from s to t in the residue graph G_f
$f' \leftarrow$ augment(f, P)
$f \leftarrow f'$
$G_f \leftarrow G_{f'}$
Return f;

Let all capacities in G are integers, and the flow out from s be $C= f^{out}(s)=\sum_{e \text{ out of } s} c_e$, which is the maximum value of any flow. Then Ford-Fulkerson algorithm terminates in at most C iterations. So the run time of Ford-Fulkerson algorithm is O(mC).

Definition 4: In a flow network G=(V, E), a *s-t cut* (A, B) is a partition of V into A and B so that $s\in A$ and $t\in B$. The capacity of a cut (A, B), $c(A,B)=f^{out}(A)=\sum_{e \text{ out of } A} c_e$. The cut with the minimum capacity is called the *minimum cut* of G.

Lemma 2. Let f be any s-t flow, and (A,B) be any s-t cut. Then $v(f)=f^{out}(A)-f^{in}(A)=f^{in}(B)-f^{out}(B)$.
Proof: For any internal node v, $f^{out}(v)-f^{in}(v)=0$; for s, $v(f)=f^{out}(s)$; $f^{in}(s)=0$.

Corollary 1. Let f be any s-t flow, and (A,B) be any s-t cut. Then $v(f)\leq c(A,B)$.
Proof: $v(f)=f^{out}(A)-f^{in}(A)\leq f^{out}(A)\leq\sum_{e \text{ out of } A} c_e=c(A,B)$.

Corollary 2. If f is a maximum flow in G, there exists a s-t cut (A, B) in G for which v(f)=c(A,B), and (A,B) is a minimum cut.
Proof: Let A be the set of nodes reachable by s in G_f, and B=V-A. All edges from a node in A to a node in B have capacity of c_e, and all edges from a node in B to a node in A has a capacity of 0. $v(f)= f^{out}(A)-f^{in}(A)= \sum_{e \text{ out of } A} c_e=c(A,B)$, i.e., c(A,B) reaches the upper bound v(f).

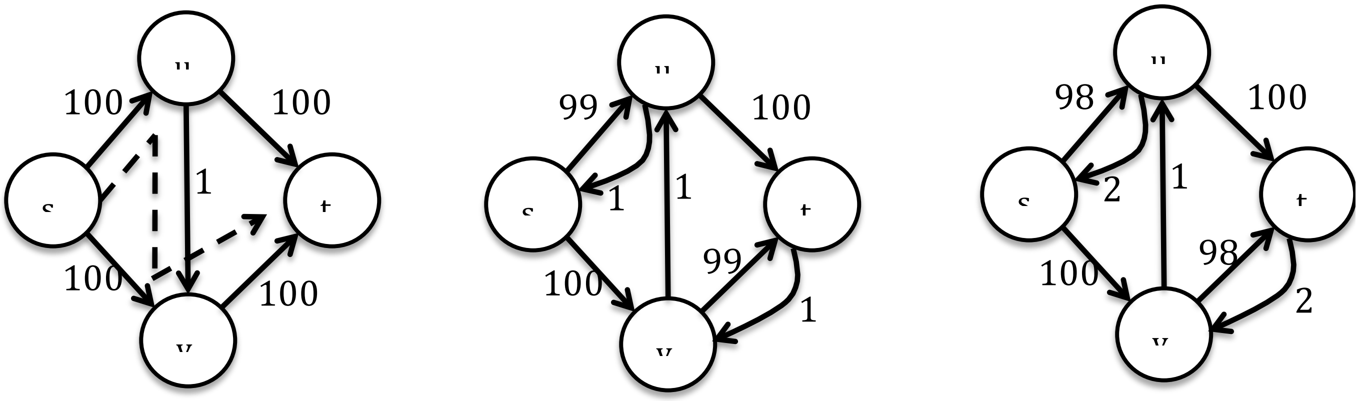
Ford-Fulkerson algorithm can also be used to find a minimum cut in a flow network.

Maximum flow minimum cut theorem. In every flow network, there is a flow f and a cut (A,B) so that v(f)=c(A,B).

Note this theorem is not dependent on Ford-Fulkerson algorithm, and thus does not require the capacities to be integers.

2. Choosing good augmenting paths.

F-F algorithm run time: O(mC). This could be very bad when C is a large integer.



Recall that augmentation increases the value of maximum flow by the bottleneck capacity of the augmenting path; so if we choose augmenting paths with large bottleneck capacity, we will make better progress. However, to find the path with the largest bottleneck capacity is hard. We will select a path with relatively large capacity, i.e., we maintain a scaling parameter Δ, and look for paths that have bottleneck capacity of at least Δ.

Let $G_f(\Delta)$ be the subgraph of residual graph G_f with capacity $\geq\Delta$: any augmenting path in this subgraph has bottleneck capacity $\geq\Delta$. We will work with Δ=powers of 2, and decrease it by half if we cannot find an augmenting path in $G_f(\Delta)$.

Scaling Max-flow Algorithm

```
Initially set f(e)=0 for all e in G
Set Δ to be the largest power of 2 that is smaller than C.
while Δ≥1
        while there is a s-t path in G in the residue graph Gf(Δ)
                let P be a simple path from s to t in the residue graph Gf(Δ)
                f'←augment(f, P)
                f←f'
                Gf(Δ) ← Gf'(Δ)
        Δ←Δ/2
Return f;
```

Observations: 1) the number of iterations of the outer loop is at most $1+\lfloor \log_2 C \rfloor$. 2) Within each inner loop, the flow value increases by at least Δ.

Lemma 3. Let f be the flow at the end of each inner loop. There is an s-t cut (A,B) in G for which $c(A,B)\leq v(f)+m\Delta$, and the maximum flow in the network has a value at most $v(f)+m\Delta$.

Corollary 3. The number of augmentations in the scaling phase is at most 2m.
Proof: in last scaling phase, $\Delta'=2\Delta$. Let f_p be the flow at the end of the last phase, the maximum value flow f* is $v(f^*)\leq v(f_p)+m\Delta'= v(f_p)+2m\Delta$. In the current scaling phase, each augmentation will increase the value of flow by Δ, so there are at most 2m augmentations.

Theorem. The run time of the scaling max-flow algorithm is at most $O(m^2 \log_2 C)$. When C >> m, this is much faster than the max-flow algorithm O(mC).

3. Preflow-push maximum-flow algorithms

Definition 4. A *preflow* in a flow network is a function f that maps each edge e to a non-negative real number, $f: E \rightarrow R^+$, satisfying 1) the capacity conditions: for each e for each $e \in E$, we have $0 \le f(e) \le c_e$; and 2) for each node v other than the source s, we have $\sum_{e \in In(v)} f(e) \ge \sum_{e \in Out(v)} f(e)$. We call the difference $e_f(v) = \sum_{e \in In(v)} f(e) - \sum_{e \in Out(v)} f(e)$ the *excess* of the preflow at v.

We can define the value of a preflow following definition 2, and a residual graph w.r.t. a preflow f, following definition 3.

Definition 5. A *labeling* (or *height*) is a function $h: V \rightarrow Z^+$, from the nodes to non-negative integers. A labeling h and a preflow f are *compatible* if 1) h(t)=0, h(s)=n; and 2) (steepness condition) for all edges (v, w) in the residual graph, we have $h(v) \le h(w)+1$.

Lemma 4. If a preflow f is compatible with a labeling h, there is no s-t path in residual graph G_f.
Proof: We prove it by contradiction. Let P be a simple s-t path in G_f, denoted by s, v_1, …, v_k=t. By definition, we have h(s)=n, $h(v_1) \ge h(s)-1=n-1$; …, $h(v_i) \ge n-i$. So $h(t) \ge n-k=0$. So k=n, i.e., the path visit one node more than one time, which is not a simple path. Contradiction.

Corollary 4. If s-t flow is compatible with a labeling h, f is a flow of maximum value.

Preflow-Push algorithm

Initiation: h(s)=n, for all other v, h(v)=0; $f(e)=c_e$ for all edges leaving s e=(s,v) (these edges will not be in the residual graph), and f(e)=0 for all other edges.
Note: initial f and h are compatible.

Pushing and Relabeling: we want to turn preflow into a feasible flow, while keeping it compatible.

push(f, h, v, w) //pushing the excess of v, $e_f(v)$ along any edge (v,w) in the residual graph, where h(w)<h(v), i.e. h(v)=h(w)+1.
if e=(v,w) is a forward edge
    increase f(e) to $\min(c_e, f(e)+e_f(v))$
else if (v,w) is a backward edge
    e $\leftarrow$ (w,v)
    decrease f(e) to $\max(0, f(e)-e_f(v))$

relabel(f, h, v) //if cannot push the excess of v along any edge (v,w) in the residual graph, i.e., $e_f(v)>0$ & for all edges (v,w), $h(w)\geq h(v)$, we need to increase the label (height) of v. Increase h(v) by 1

Preflow-Push
$h(s)\leftarrow n$, for all other v, $h(v)\leftarrow 0$;
$f(e)\leftarrow c_e$ for all edges leaving s, e=(s,v), and f(e)=0 for all other edges.
while there is a node $v\neq t$ with $e_f(v)>0$
    if there is w, s.t. $(v,w)\in E_f$, and $h(w)<h(v)$
        push(f, h, v, w)
    else
        relabel(f, h, v)

Observations: Throughout the algorithm, 1) the labels are non-negative integers; 2) f is a preflow with integer value; 3) the f and h are compatible; 4) the algorithm terminate when no node other than s or t has excess, so the final preflow f is flow.

Theorem. Preflow-Push algorithm terminates with a maximum flow.

Run time analysis.

Lemma 5. Let f be a preflow. If node v has excess, there is a path in G_f from v to the source s.

Proof: Let A be the set of nodes w such that there is a path from w to s in G_f and B=V-A. We need to show all nodes in B with excess of 0.

1) $s\in A$; and no edge e leaving A with f(e)>0, otherwise there is a backward edge (y,x) in G_f and then y should be in A;
2) Now consider the sum of excesses in B, recall that each node in B has nonnegative excess since s is not in B, so $0\leq\sum_{v\in B}e_f(v)=\sum_{v\in B}\left(f^{in}(v)-f^{out}(v)\right)=-f^{out}(B)$, indicating the sum of excesses of nodes in B=0.

Lemma 6. Throughout the algorithm, all nodes have $h(v)\leq 2n-1$.

Proof: h(t)=0 and h(s)=n do not change throughout the algorithm. For any other node v, its height can be increased by 1 only when it has excess (with preflow f), and then according to Lemma 5, there exist a path P from v to s in G_f. Let |P| be the number of edges in the path. Along each edge (v,w) in the path, the height of nodes can decrease at most 1; hence, $h(v)-h(s)\leq|P|\leq n-1$.

Corollary 5. Throughout the algorithm, each node is relabeled at most 2n-1 times, and the total number of relabeling operations is less than $2n^2$.

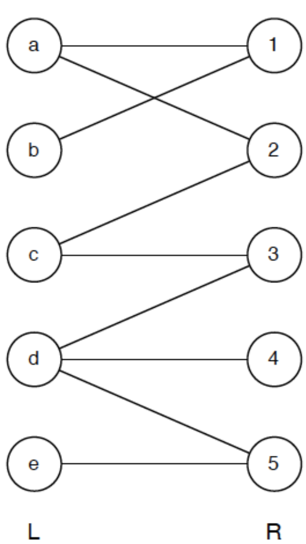
Definition 6. A push(f, h, v, w) operation is *saturating* if either e=(v,w) is a forward edge in G_f and f(e) increases c_e, or e is a backward edge and to 0, i.e., after the push, edge e is no longer in the residual graph G_f. All other push operations are called *nonsaturating*.

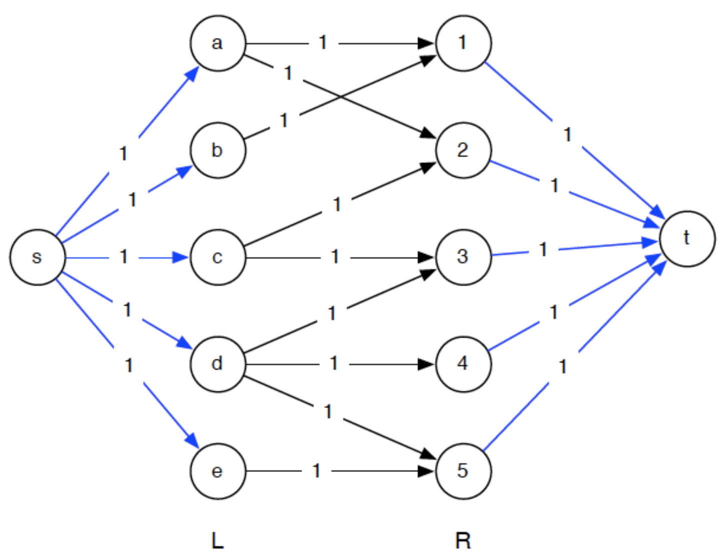
Lemma 7. Throughout the algorithm, the number of saturating push operations is at most 2nm.
Proof: When we have a saturating push on (v, w), we have h(v)=h(w)+1, and after the push, the edge (v, w) is no longer in G_f. To perform another push on (v, w), we must make the edge appear again in G_f, i.e., to push from w to v, which requires to increase w's height by 2 (so that w's height above v's). Notice that w's height can increase at most n-1 times (Corollary 5); hence a saturating push from v to w can occur at most n times. Since every edge e∈E can give rise to two edges (one forward and one backward) in the residual graph, overall, the saturating push can be performed at most 2mn.

Lemma 8. Throughout the algorithm, the number of non-saturating push operations is at most $4n^2m$.
Proof: For a preflow f and a compatible labeling h, we define $\Phi(f,h) = \sum_{v:e_f(v)>0} h(v)$ to be the sum of heights of all nodes with positive excess. In the initial preflow and labeling, Φ(f,h)=0; it remains nonnegative afterwards. When the algorithm terminates, Φ(f,h)=0 again. A nonsaturating push decreases Φ(f,h) by at least 1, since after the push the node v will have no excess (get out of the sum) and h(v)=h(w)+1≥1. Each relabeling will increase Φ(f,h) exactly by 1; so the total increment of Φ(f,h) by relabeling is at most $2n^2$. A saturating push may increase Φ(f,h) by the height of w, which is at most 2n-1. There are at most 2nm saturation pushes, so the total increment of Φ(f,h) by saturating pushes is at most 2mn(2n-1). So Φ(f,h) can increase at most $4mn^2$ indicating there can be at most $4mn^2$ nonsaturating pushes.

Theorem. The run time of preflo-push algorithm is $O(mn^2)$.

4. bipartite matching problem and disjoint path problem.



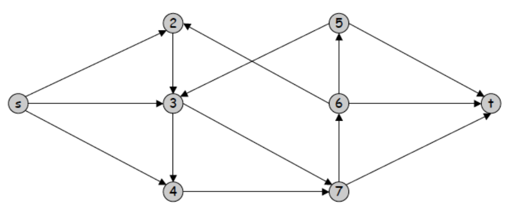


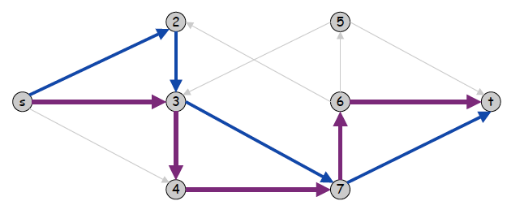
Theorem. The size of the maximum matching in G is equal to the value of the maximum flow in G'; and the edges in such a matching in G are the edges that carry flow from s to t in G’.

F-F algorithm can solve maximum matching problem in O(mn).

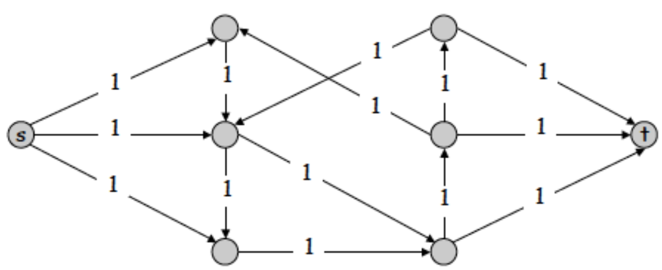
Definition 7. A set of paths in a graph G is *edge-disjoint* if their edges are disjoint, i.e., no two paths share an edge, though they may go through the same vertex.

Given a directed graph G with two distinct nodes s and t, the *directed edge-disjoint path problem* is to find the maximum number edge-disjoint s-t paths in G. Similarly, the *undirected edge-disjoint path problem* is to find the maximum number of edge-disjoint s-t paths in an undirected graph G.





Each directed graph G with two distinct nodes s and t can be converted into a flow network G’ by assigning the capacity 1 to each edge in G, and setting s as the source and t as the sink.



Theorem. There are k edge-disjoint paths from s to t in a directed graph G from s to t iff the value of the maximum flow in G’ is at least k.
Proof: => is straightforward, set f(e)=1 if e is included in one of the k edge-disjoint paths, and set f(e)=0 for remaining edges; This form a flow of value k.
<= by induction on the value of flow k. if k=0, trivial. Otherwise there must be an edge (s,u) going out of s. We trace a path of edges that must also carry flow– there is always some edges carrying flow along the path—until either 1) we reach t; or 2) we reach a vertex v that has been visited in the path. If 1), we find a path P from s to t. Let f’ be the flow obtained by decreasing f(e) for e in P. f’ has a value at least k-1, according to induction, we have k-1 edge-disjoint paths in G using edges$\notin$ P, and thus in total we have k edge-disjoint paths in G. If 2), eliminate the cycle from v to v and continue traverse the graph along the flow until reaching t. This will also give a edge-disjoint path.

Corollary 5. F-F algorithm can be used to find a maximum set of edge disjoint s-t paths in a directed graph G in O(mn) time.

We can convert an undirected graph G into a directed graph G’ by replacing each edge (u,v) in G with two directed edge (u,v) and (v,u).

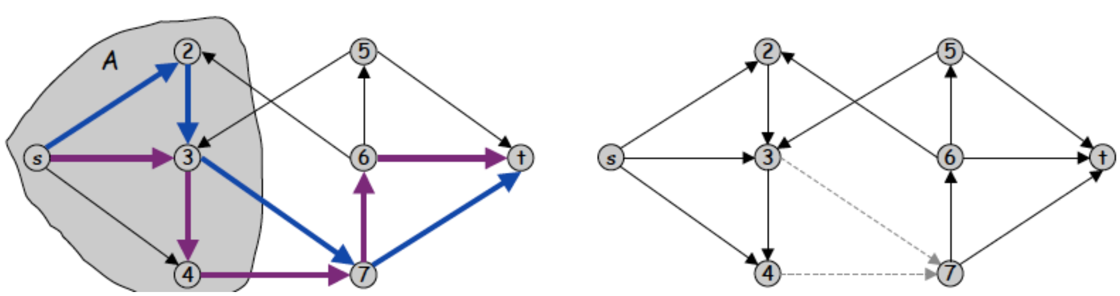
Lemma 9. In any flow network, there is maximum flow f where for all opposite directed edges e=(u,v) and e’=(v,u), either f(e)=0 or f(e’)=0.
Proof: Assume f is a maximum flow. We can convert f to another flow f’ that satisfying the condition: for every pair of e and e’, let δ=min(f(e), f(e’)), decrease both f(e) and f(e’) by δ. It is easy to show that f’ is a flow with the same value as f.

Corollary 6. There are k edge-disjoint paths in an undirected graph G from s to t iff the maximum value of an s-t flow in the directed version G’ of G is at least k.

Corollary 7. F-F algorithm can be used to find a maximum set of edge disjoint s-t paths in an undirected graph G in O(mn) time.

Network connectivity problem. Given a directed graph G=(V,E) and two distinct nodes s and t in the graph, find a minimum number of edges whose removal will disconnect s and t.

Theorem (Menger, 1927) The maximum number of edge-disjoint path is equal to the minimum number of edges whose removal will disconnect s and t.
Proof: <=, Assume F is the minimum set of edges whose removal disconnect s and t. Every path from s to t will use one edge e in F. So there are at most |F| edge-disjoint paths.
=> Assume there are k edge-disjoint paths from s to t. Then the maximum flow in the graph G’ is k, and thus the minimum cut (A, B) of the graph has capacity of k. This indicates there are k edges from A to B, which forms the set of edges whose removal will disconnect s and t.

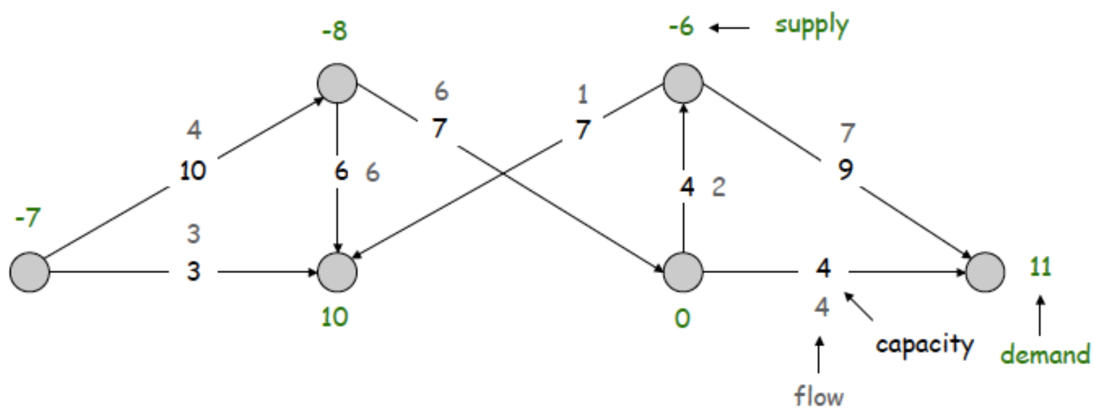


5. Circulation with supplies and demands

Definition 8. A directed graph G=(V,E) with *supplies and demands* is defined as a directed graph, with 1) edge capacities c(e), for each e∈E, and 2) node supply and demands d(v), v∈V: if d(v)=0 we call the node transshipment; if d(v) > 0; we call it demand; if d(v) < 0; we call it supply.

Definition 9. A *circulation* is a function that satisfies: 1) for each e∈E: 0 ≤ f(e) ≤ c(e) (capacity condition) and 2) for each v∈V: $\sum_{v \in V} f^{in}(v) - \sum_{v \in V} f^{out}(v) = d(v)$ (conservation condition)

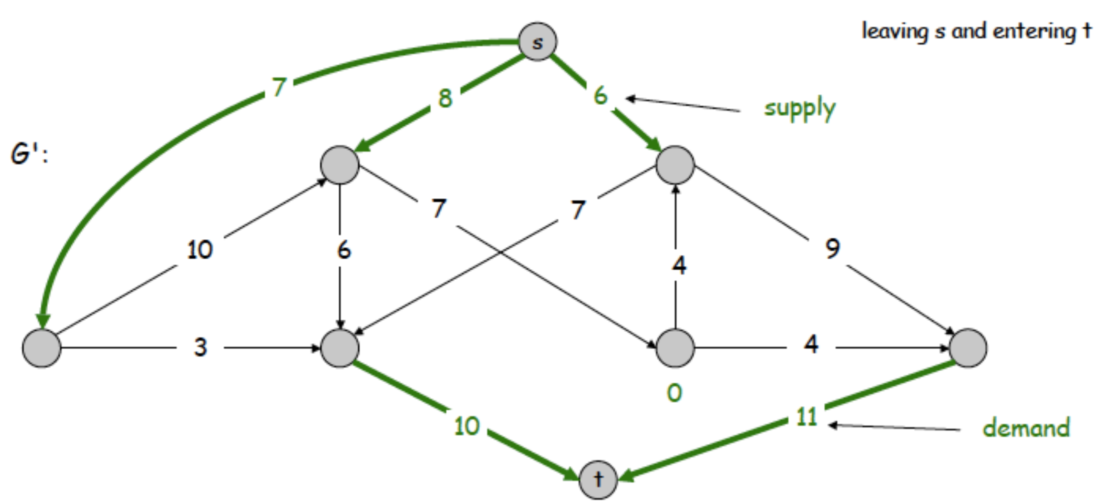
Circulation problem: given (V, E, c, d), is there a feasible circulation?



Lemma 10. (Necessary condition) if there is a feasible circulation with demands {d(v)}, then $\sum_v d(v)=0$.

Convert circulation problem into a maximum flow problem. Given G=(V,E), c and d, we build a flow network G' by

1) adding new source s and sink t;
2) for each v with d(v) < 0, adding edge (s, v) with capacity -d(v);
3) for each v with d(v) > 0, adding edge (v, t) with capacity d(v).



Observation: the resulting graph is a flow network.

Lemma 11 Let D be the total demands in the graph: $D = \sum_{v:d(v)>0} d(v) = -\sum_{v:d(v)<0} d(v)$. The

graph G has a feasible circulation with demands {d(v)} iff the maximum s-t flow in G' has value D (saturated).
Definition 10. A *circulation with lower bound* is a function that satisfies: 1) for each e∈E: $l(e) \le f(e) \le c(e)$ (capacity condition) and 2) for each v∈V:
$\sum_{v\in V} f^{in}(v) - \sum_{v\in V} f^{out}(v) = d(v)$ (conservation condition).

Circulation with lower bound problem: given (V, E, c, l, d), is there a feasible circulation?

Convert circulation with lower bound problem into a circulation problem (i.e., modeling lower bound by demands).

6. Applications

Survey design
Design survey asking n_1 consumers about n_2 products.
Can only survey consumer i about product j if they own it.
Ask consumer i between c_i and $c_{i'}$ questions, each for a different product.
Ask between p_j and $p_{j'}$ consumers about product j.
Goal: design a survey that meets these specifics, if possible.
Special case: when $c_i = c_{i'} = p_i = p_{i'} = 1$, the problem can be formulated as a bipartite perfect matching.

Formulated as a circulation problem with lower bounds: include an edge (i, j) if consumer j owns product i, a circulation←→feasible survey design.

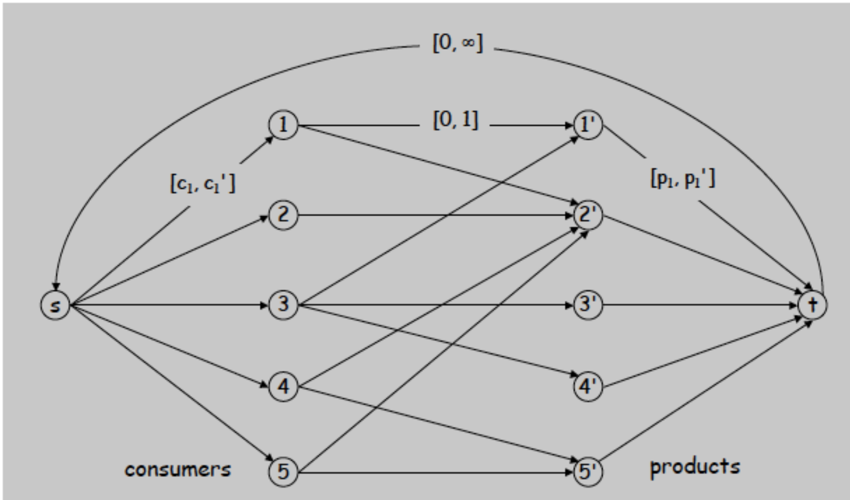


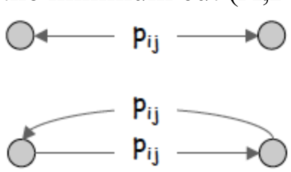
Image segmentation
To divide image into coherent regions, e.g., three people standing in front of complex background scene, to identify each person as a coherent object.

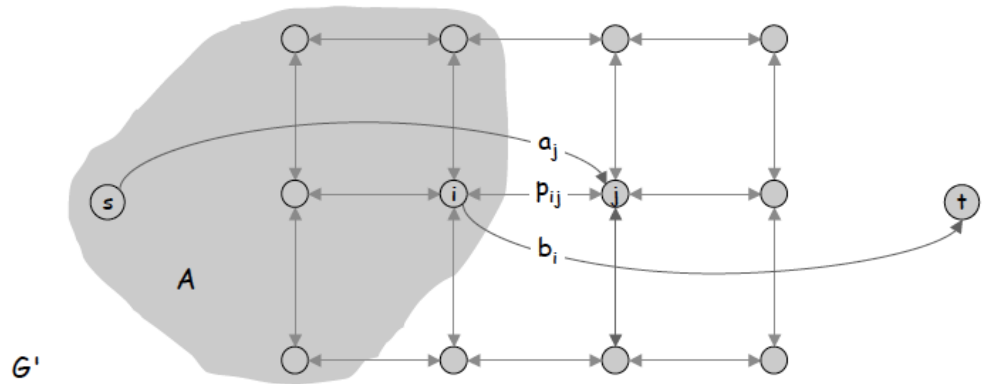
Problem formulation: foreground / background segmentation.
1) label each pixel in picture as belonging to foreground or background.
2) V = set of pixels, E = pairs of neighboring pixels.
3) $a_i \geq 0$ is likelihood pixel i in foreground; $b_i \geq 0$ is likelihood pixel i in background.
4) $p_{ij} \geq 0$ is separation penalty for labeling one of two neighboring pixels i and j as foreground, and the other as background.
Goals:
1) Accuracy: if $a_i > b_i$ in isolation, prefer to label i in foreground.
2) Smoothness: if many neighbors of i are labeled foreground, we should be inclined to label i as foreground.

Scoring: to find partition (A=foreground, B=background) that maximizes:
$\sum_{i \in A} a_i + \sum_{i \in B} b_i - \sum_{\substack{(i,j) \in E, \\ |A \cap \{i,j\}|=1}} p_{ij}$, is equivalent to minimize $\sum_{i \in B} a_i + \sum_{i \in A} b_i + \sum_{\substack{(i,j) \in E, \\ |A \cap \{i,j\}|=1}} p_{ij}$.

Formulating it as a minimum cut problem:
1) G' = (V', E').
2) add source to correspond to foreground;
3) add sink to correspond to background;
4) use two anti-parallel edges instead of undirected edge.

Then the minimum cut (A,B) have the minimum score as indicated above.

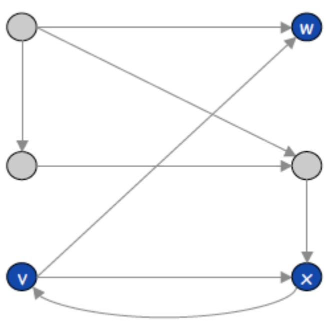




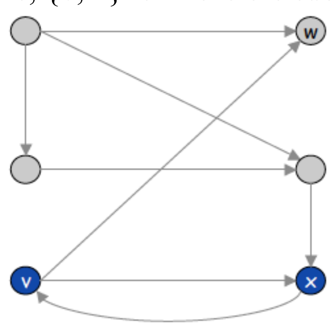
Project selection
We are given a set of projects with prerequisites
1) Set P of possible projects. Project v has associated revenue p(v): some projects generate money: create interactive e-commerce interface, redesign web page; others cost money: upgrade computers, get site license
2) Set of prerequisites E. If (v, w) ∈ E, can't do project v and unless also do project w.
3) A subset of projects A ⊆ P is feasible if the prerequisite of every project in A also belongs to A.

Goal: choose a feasible subset of projects to maximize revenue.

Build a prerequisite graph: include an edge from v to w if can't do v without also doing w, then {v, w, x} is feasible subset of projects; {v, x} is infeasible subset of projects.

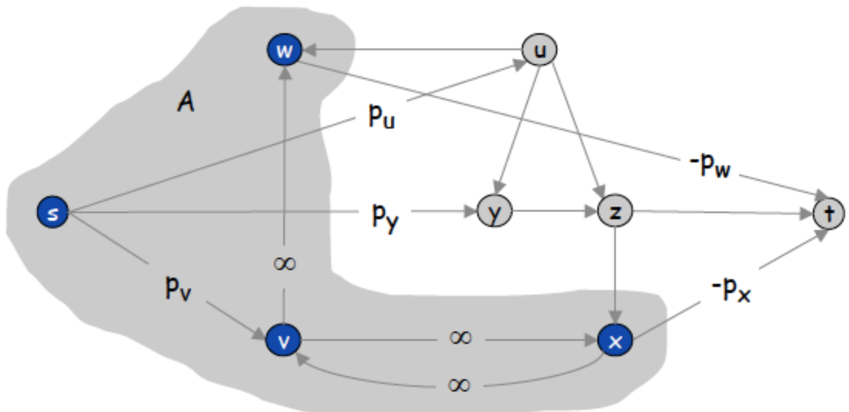


feasible



infeasible

Minimum cut formulation: constructing the flow network of G'
1) Assign capacity ∞ to all prerequisite edge.
2) Add two nodes s, t.
2) Add edge (s, v) with capacity p(v) if p(v) > 0.
3) Add edge (v, t) with capacity –p(v) if p(v) < 0.



Theorem: The cut (A, B) in G' is minimum iff A-{s} is the optimal set of projects.
Proof: 1) infinite capacities on prerequisite edges ensure A-{s} is feasible; and 2) the revenue is maximized because the capacity of the cut is minimized:

$$C(A,B) = \sum_{v \in B : p(v) > 0} p(v) + \sum_{v \in A : p(v) < 0} -p(v) = \sum_{v : p(v) > 0} p(v) - \sum_{v \in A} p(v)$$