Complex Networks
fundamental network algorithms

2017.12.18(Mon)

Goal

metrics	algorithms
models	processes

contents of this chapter

- algorithms for calculating centrality indices, finding components, calculating shortest paths and maximum flow

software implementing common network algorithms

Name	Availability	Platform	Description
Pajek	Free	W	Interactive social network analysis and visualization
Net Workbench	Free	WML	Interactive network analysis and visualization
Netminer	Commercial	W	Interactive social network analysis and visualization
InFlow	Commercial	W	Interactive social network analysis and visualization
UCINET	Commercial	W	Interactive social network analysis
yEd	Free	WML	Interactive visualization
Visone	Free	WL	Interactive visualization
Graphviz	Free	L	Visualization
NetworkX	Free	WML	Interactive network analysis and Python library
JUNG	Free	WML	JAVA library for network analysis and visualization
igraph	Free	WML	C/R/Python libraries for network analysis
GTL	Free	WML	C++ library for network analysis
LEDA/AGD	Commercial	WL	C++ library for network analysis

Table 9.1: **A selection of software implementing common network algorithms.** Platforms are Microsoft Windows (W), Apple Macintosh (M), and Linux (L). Most Linux programs also run under Unix and Unix-like systems such as BSD, and many Windows programs can run on Macs and Linux systems using emulation software.

Reasons for studying algorithms (even though ready-made software packages are available)

- much time can be wasted when people misunderstand the answers the software can give them

- you will sooner or later find that you need to do something that cannot be done with standard software and you'll have to write some programs of your own

- by relying on ready-made packages, researchers have become limited in what types of analysis they can perform

running time and computational complexity

- if the calculation you have started will take a thousand years to finish, it is basically useless
- concept of complexity is useful because it helps us to avoid wasting our energies on programs that will not finish running in any reasonable amount of time

computational complexity

- a measure of the running time of a computer algorithm
- example: to find the largest number in a list of n numbers
  - simply go through the list to the end, keeping a record of the largest number we have seen
  - worst case: every number is bigger than all the ones before it
    - its running time is $n\tau$
    - time complexity of this algorithm is order n ($O(n)$)

algorithms for degrees and degree distributions

vertex	incoming edges	outgoing edges
1	4	3,4
2		
3	1	4
4	3,1,5	5,1
5	4	4

- degree of a vertex
  - if network is stored in adjacency list: O(1)
  - if network is stored in adjacency matrix: O(n)
- degree distribution p_k : O(n)

  # of vertices

- cumulative degree distribution P_k
  - form a histogram of the degrees and calculate : O(n)

$$P_k = \sum_{k'=k}^{\infty} p_{k'} = -p_{k-1} + \sum_{k'=k-1}^{\infty} p_{k'} = P_{k-1} - p_{k-1} \qquad P_0 = \sum_{k'=0}^{\infty} p_{k'} = 1$$

  - sorting degrees in descending order and rank from 1 to n : O(nlogn)
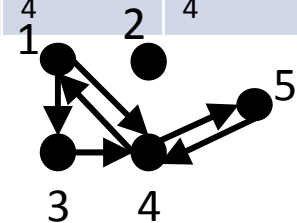
cf. assortative mix, homophily

- high-degree vertices tend to connect high-degree ones

- correlation coefficient
$$r = \frac{\sum_{ij}(A_{ij} - k_i k_j / 2m)k_i k_j}{\sum_{ij}(k_i \delta_{ij} - k_i k_j / 2m)k_i k_j}$$

- faster computation of r

$$r = \frac{S_1 S_e - S_2^2}{S_1 S_3 - S_2^2}$$

$$S_e = \sum_{ij} A_{ij} k_i k_j = 2 \sum_{edges(i,j)} k_i k_j$$

Given the degrees of all vertices, it takes time O(m)

$$S_1 = \sum_i k_i, \ S_2 = \sum_i k_i^2, \ S_3 = \sum_i k_i^3$$

time O(n)

time O(m+n)

- social networks: positive r

- other networks: negative r

correlation coefficient

- computing S_e takes time $O(m)$

  # of edges

- computing S_1, S_2, and S_3 take time $O(n)$

- total time is $O(m+n)$

  mean degree $c=2m/n$ is constant

  – sparse networks ($m \propto n$) : $O(m+n) \equiv O(n)$

 – dense networks ($m \propto n^2$): $O(m) \equiv O(n^2)$

cf. transitivity
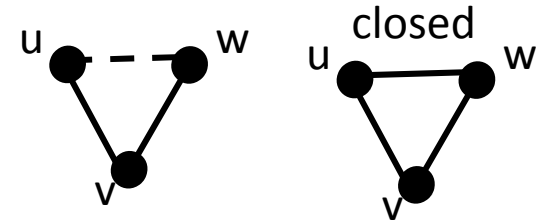
- a•b and b•c -> a•c
- u & v are friends and v & w are friends
- clustering coefficient: $C = \dfrac{(\text{\# of closed paths of length two})}{(\text{\# of paths of length two})}$
  - C=1:clique
  - C=0:tree, square lattice
- $C = \dfrac{(\text{\# of triangles}) \times 6}{(\text{\# of paths of length two})} = \dfrac{(\text{\# of triangles}) \times 3}{(\text{\# of connected triples})}$
- social networks tend to have high values

cf. local clustering coefficient

- $C_i = \dfrac{\text{(\# of pairs of neighbors of i that are connected)}}{\text{(\# of paths of neighbors of i)}}$

- vertices with higher degree have lower local clustering coefficient on average

- structural holes
  - bad for info spread or traffic
  - good for the central vertex
    - it can control the flow of information

- similar to betweenness centrality

structural holes
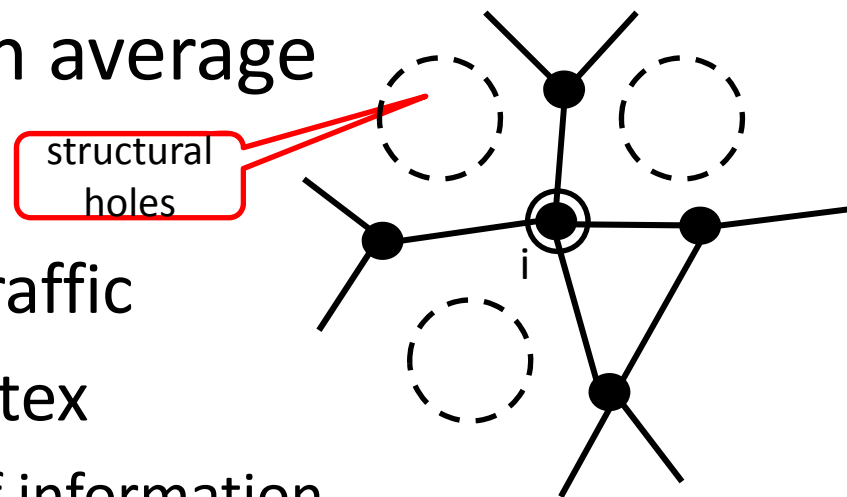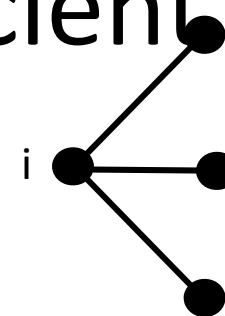
clustering coefficients

<naive method>

- ## local clustering coefficient

$$C_i = \frac{(\text{\# of pairs of neighbors of i that are connected})}{(\text{\# of paths of neighbors of i})}$$

  – go through every pair of neighbors of i, and count how many are connected

- ## overall clustering coefficient

$$C = \frac{(\text{\# of triangles}) \times 3}{(\text{\# of connected triples})}$$

  – we consider for every vertex each pair of neighbors and find whether they are connected

complexity of naive method

- vertex i with degree k_i has $\dfrac{1}{2}k_i(k_i-1)$ pairs of neighbors

- total number of checks:

$$\sum_i \frac{1}{2}k_i(k_i-1) = \frac{1}{2}n(\langle k^2\rangle - \langle k\rangle) \qquad \langle k\rangle = \frac{1}{n}\sum_i k_i = \frac{2m}{n} \qquad \langle k^2\rangle = \frac{1}{n}\sum_i k_i^2$$

- depends on degree distribution
  - if a network follows a power law $p_k \approx k^{-\alpha}$, the second moment $\langle k^2\rangle$ formally diverges if $\alpha < 3$
  - it will take an infinite amount of time!

more careful calculation (1)

vertex	neighbors
1	3,4
2	
3	4,1
4	5,1,3
5	4

- a network is stored in adjacency list

- for a single vertex i
  - # of pairs of neighbors (j and l) : $\frac{1}{2}k_i(k_i-1)$
  - for each pair, check whether an edge exist between them : time proportional to k_j or k_l
  - if chosen at random : time proportional to $k_j + k_l$
  - Γ_i : set of neighbors of vertex i
  - total time to check for edges between all pairs:

$$\sum_{j,l\in\Gamma_i:j<l}(k_j+k_l)=\frac{1}{2}\sum_{j,l\in\Gamma_i:j\neq l}(k_j+k_l)=\sum_{j,l\in\Gamma_i:j\neq l}k_j=(k_i-1)\sum_{j\in\Gamma_i}k_j$$
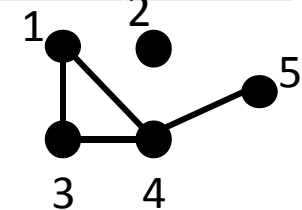
more careful calculation (2)

- sum of the quantity over all vertices i:

$$\sum_i (k_i - 1) \sum_{j \in \Gamma_i} k_j = \sum_{ij} A_{ij}(k_i - 1)k_j = \sum_{ij} A_{ij}k_i k_j - \sum_j k_j^2 \qquad \sum_i A_{ij} = k_j$$

- compare with the expression of assortativity

$$r = \frac{\sum_{ij}(A_{ij} - k_i k_j / 2m)k_i k_j}{\sum_{ij}(k_i \delta_{ij} - k_i k_j / 2m)k_i k_j}$$ same

- time to calculate clustering coefficient depends on whether the degrees of vertices are correlated or not

- if no correlation, r=0 $\sum_{ij} A_{ij}k_i k_j = \frac{1}{2m}\sum_{ij} k_i^2 k_j^2 = \frac{1}{2m}\left[\sum_i k_i^2\right]^2$

- running time for clustering coefficient:

$$\frac{1}{2m}\left[\sum_i k_i^2\right]^2 - \sum_j k_j^2 = n\langle k^2\rangle\left[\frac{\langle k^2\rangle}{\langle k\rangle} - 1\right]$$

$$2m = \sum_i k_i = n\langle k\rangle$$

depends on <k²>: it can become large for highly skewed degree distributions

cf. assortative mixing by degree

- assortative: high-degree vertices connect to other high-degree vertices
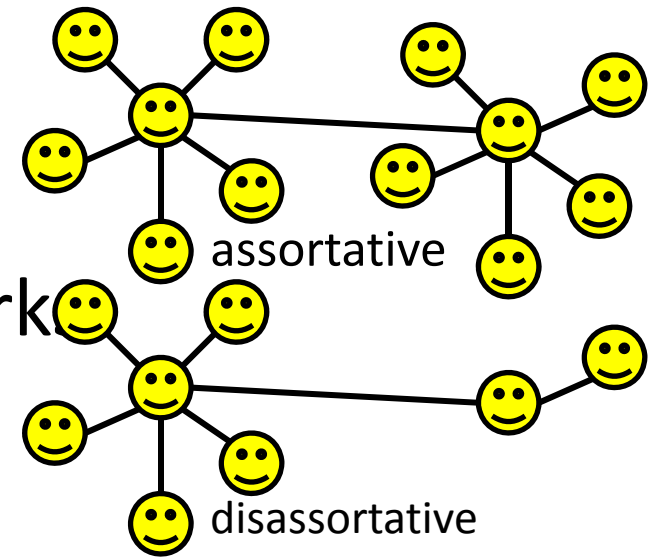
- core/periphery structure :

common feature of social network

assortative

disassortative

- covariance

$$\mathrm{cov}(k_i, k_j) = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) k_i k_j$$

- correlation coefficient (assortativity coefficient)

$$r = \frac{\sum_{ij} (A_{ij} - k_i k_j / 2m) k_i k_j}{\sum_{ij} (k_i \delta_{ij} - k_i k_j / 2m) k_i k_j}$$

more careful calculation (3)

- if a network is simple with no multiedges, →maximum degree is k=n, and the degree distribution is cut off → $<k^2>$ scales at worst as $n^{3-\alpha}$

- running time of clustering coefficient would go as $n \times n^{3-\alpha} \times n^{3-\alpha} = n^{7-2\alpha}$

- $\alpha=3$: $O(n)$

- $\alpha=2$: $O(n^3)$

shortest paths and breadth-first search

- the study of each of these algorithms has three parts
    - description of the algorithm
    - analysis of its running time
    - proof that the algorithm performs the calculation it claims to

breadth-first search (1)

- finds the shortest distance from a given starting vertex s to every other vertex in the same component as s
- s have distance 0 from itself
- find all the neighbors of s (distance 1)
- find all the neighbors of *those* vertices (distance 2)
- …

0 1 2 3
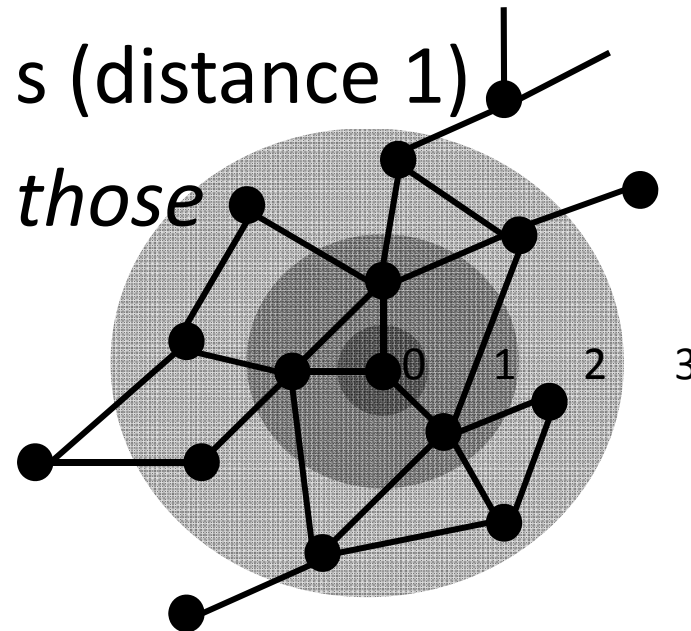
breadth-first search (2)

- every vertex whose shortest distance from s is d has a network neighbor whose shortest distance from s is d-1

- breadth-first search finds the component to which vertex s belongs
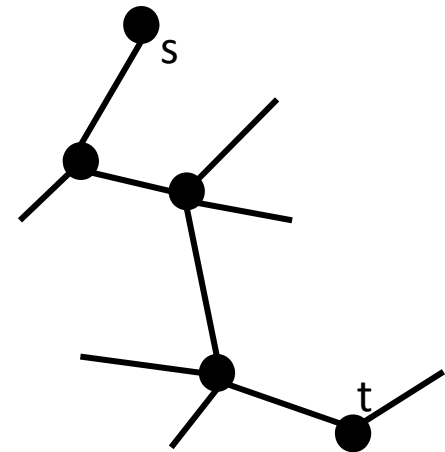
naive implementation

vertex	distance
1	-1
2	-1
s	0
4	-1
5	-1

O(n) time for setting up

- array of n elements to store the distance of each vertex from the source vertex s
- distance variable d

1. find all vertices that are distance d from s
2. find all the neighbors of those vertices and check to see if distance from s is unknown
3. if # of neighbors with unknown distance is zero, the algorithm is over. otherwise, set the distance of each of those neighbors to d+1
4. increase the value of d by 1
5. repeat from step 1

find vertices of distance d:O(n)
distance from 1 to r: O(rn)

vertices of distance d:
n × O(m/n)=O(m)

neighbors of a vertex

total running time:
O(n + rn + m)

r: diameter
n (in the worst case)->O(m+n^2)
log n (in most cases)->O(m+nlogn)

better implementation (1)

- time consuming part is step 1 (find vertices that are distance d from the starting vertex s)
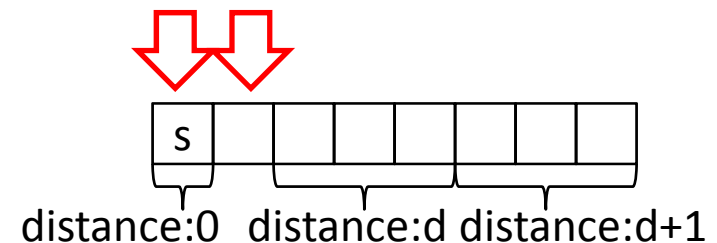- use distance array and queue

vertex	distance
1	-1
2	-1
s	0
4	-1
5	-1

read pointer
(next item
to read)

write pointer
(next empty
space to fill)

| 5 | 2 | 6 | 4 | 1 | 9 | 3 | | | |
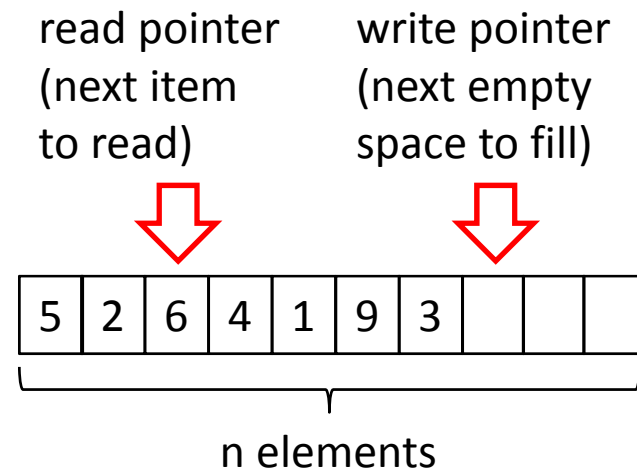
n elements

better implementation (2)

vertex	distance
1	-1
2	-1
s	0
4	-1
5	-1

1. place s in the queue, set 0 in distance array
2. if read and write pointers are pointing the same element, the algorithm is finished. otherwise, read the vertex label pointed by read pointer and increase that pointer by one
3. find the distance d for that vertex
4. go through each neighboring vertex in turn
   1. if it has a known distance, leave it alone
   2. if it has unknown distance, assign it distance d+1, store its label in the queue array
5. repeat from step 2

distance:0   distance:d   distance:d+1

better implementation (3)

- running time
  - O(n) to set up the distance array
  - for each element in the queue, we run through its neighbors (O(m/n) on average), calculate their distance and add them to the queue or do nothing (O(1))
    - we require overall at most at time n × O(m/n) = O(m)
  - the whole algorithm takes time O(m+n)
    - better than O(m+nlogn)
    - for sparse network with m ≈ n, O(n)

finding shortest paths (1)

- the previous breadth-first search does not tell us the particular path(s) by which that shortest distance is achieved
- a small modification of the bread-first search algorithm
  - if a vertex j (neighbor of i) has unknown distance
    - assign j a distance and store it in a queue
    - add a directed edge from j to i
- its running time is still O(m+n)
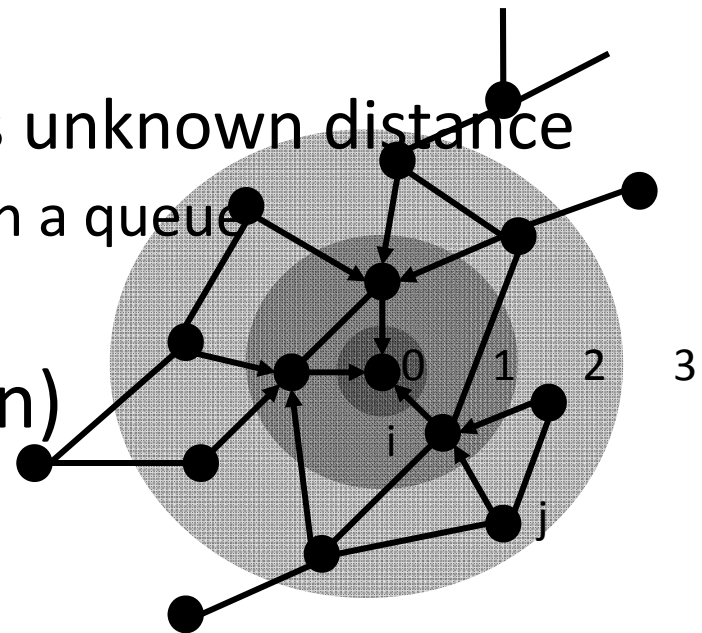
finding shortest paths (2)

- for multiple shortest paths
  - if a vertex j (neighbor of i) has unknown distance
    - assign j a distance and store it in a queue
    - add a directed edge from j to i
  - if a vertex j has distance d+1
    - add an extra directed edge from j to i
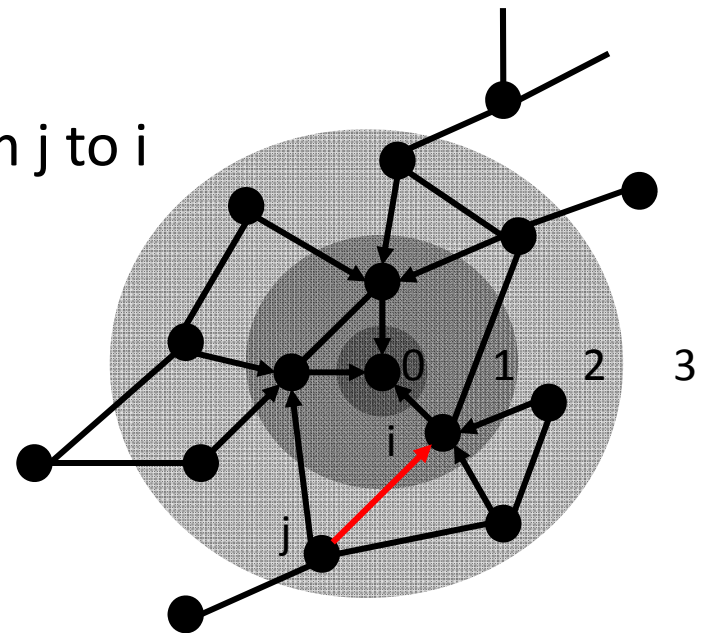
betweenness centrality

- betweenness of v : # of geodesic paths between pairs of vertices that pass through v
- simplest way: (1) use breadth-first search to find the shortest path between s and t, and see if the vertex v lies along the path (2) repeat this process for every distinct pair s, t.
- this is correct, but slow
  - $O(m+n)$ to find a shortest path between two vertices
  - $n(n-1)/2$ distinct pairs of vertices
  - calculating betweenness for v will take $O(n^2(m+n))$ (or $O(n^3)$ when $m \approx n$ (sparse network))

calculation of betweenness centrality (1)

- O(m+n) for finding paths between s and all other vertices (a shortest path tree)
- trace the paths from each vertex back to s and count the number of paths that go through v : O(nlogn)

  n paths of length logn

- O(m+nlogn) for each s
- repeat this calculation for all s -> O(n(m+nlogn)) (or O(n^2logn))
  - much better than O(n^3)
  - betweenness of all vertices will be calculated at the same time

we can still do better

- many of the shortest paths share many of the same edges
- score of each vertex (betweenness count for paths that end at vertex s)
  - leaves:1
  - others:1+∑(score of its immediate children)
- the queue array created by BFS have a

list of vertices in order of decreasing

distance from s -> O(m+n) for each s

- O(n(m+n)) for all vertices

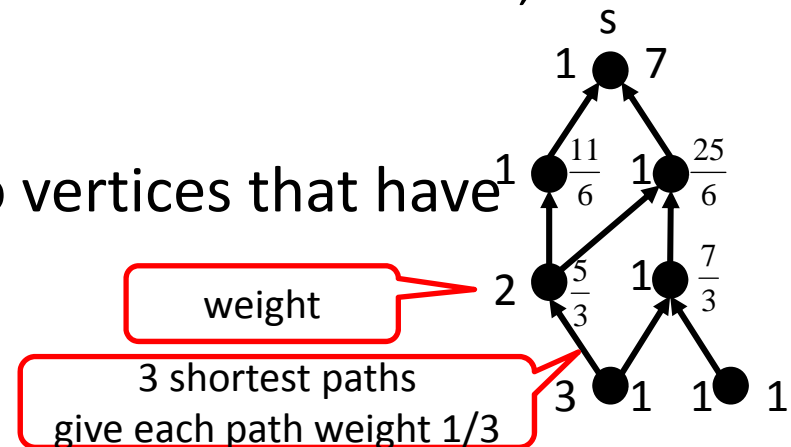worst case: checking every neighbor of every vertex -> O(m+n)

BFS: O(m+n)

s
7

2   4

1   1   2

1

leaves

distance:0   distance:d   distance:d+1
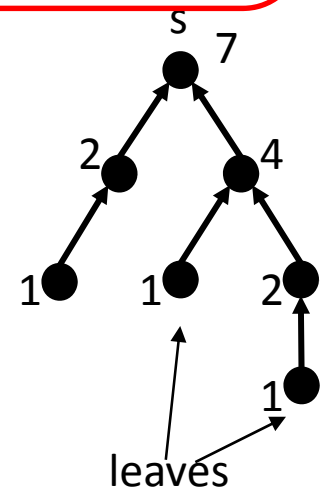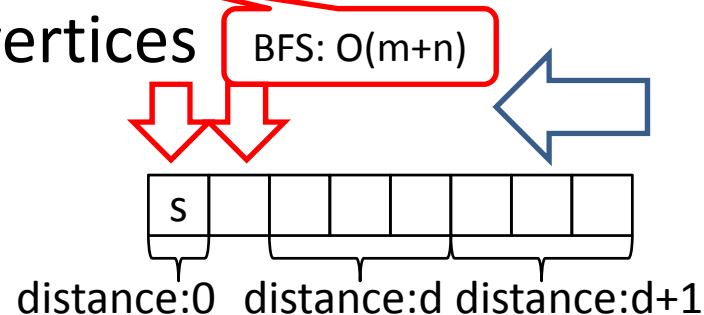
more than one shortest path (1)

counting the number of shortest paths from each vertex

1. assign vertex s distance zero, set d=0, and assign s a weight $w_s=1$

2. for each vertex i whose distance is d, follow each attached edge to the vertex j at its other end, and then do one of the following three things:

    - if j has not yet been assigned a distance, assign it distance d+1 and weight $w_j=w_i$
    - if j has already been assigned distance d+1, then w_j <- w_j+w_i
    - if j has already been assigned a distance less than d+1, do nothing

3. increase d by 1

4. repeat from 2 until there are no vertices that have distance d

S

1 ● 7

1 ● $\frac{11}{6}$ 1 ● $\frac{25}{6}$

weight

2 ● $\frac{5}{3}$ 1 ● $\frac{7}{3}$

3 shortest paths
give each path weight 1/3

3 ● 1 1 ● 1

more than one shortest path (2)

- the fraction of the paths to s that pass through j and that also pass through i is w_i/w_j

1. find every leaf vertex t and assign it a score of $x_i = 1$

2. from the bottom of the tree, assign to each vertex i a score $x_i = 1 + \sum_j x_j w_i / w_j$ (j is a neighbor immediately below vertex i)
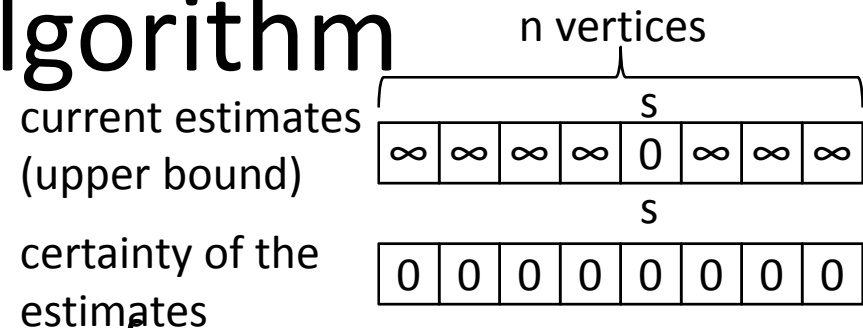
3. repeat 2 until vertex s is reached
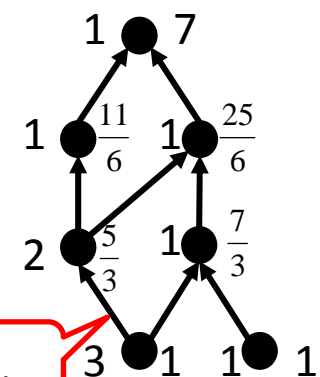
3 shortest paths
give each path weight 1/3

shortest paths in networks with varying edge lengths

- the shortest path may traverse many edges
- Dijkstra's algorithm
  - finds the shortest distance from a given source vertex s to every other vertex
  - it takes the lengths of edges into account
  - it keeps a record of the shortest distance so far and update that record whenever a shorter one is found
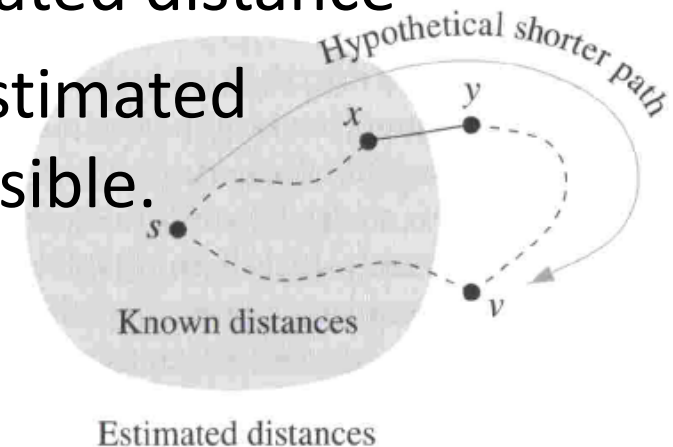
Dijkstra's algorithm

n vertices

current estimates (upper bound)

s

∞	∞	∞	∞	0	∞	∞	∞

s

certainty of the estimates

0	0	0	0	0	0	0	0
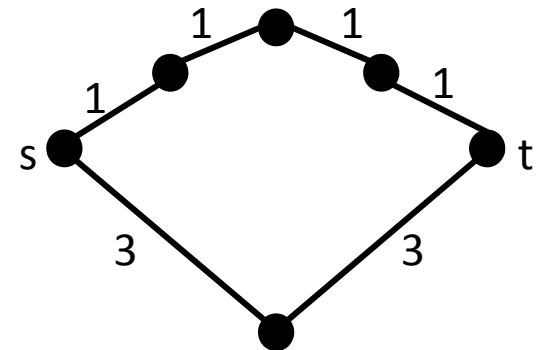
- two arrays of n elements
  - current estimates of the distance from s
  - certainty of the estimates

1. find vertex v that has the smallest estimated distance (smallest and not yet certain)

2. mark this distance as being certain

3. calculate the distances from s via v to each of the neighbors of v. if the resulting distance is smaller, replace the estimate

4. repeat from 1 until the distances to all vertices are certain

the smallest estimated distance is certain

- if v is the vertex with the smallest estimated distance from s, then that estimated distance must be the true shortest distance to v.
- if there were a shorter path s,...,x,y,...,v,
  - all points along the path must have shorter distances from s than v's estimated distance
 - it means that y has a smaller estimated distance than v, which is impossible.

Hypothetical shorter path

x y

s

v

Known distances

Estimated distances

running time of Dijkstra's algorithm

- simplest implementation
  - find the smallest estimated distance : O(n)
  - calculate new estimates of its neighbors:O(m/n) ⎤ O(m/n+n)
  - repeat above for n rounds : $O(m+n^2)$
- better implementation
  - find the smallest estimate from binary heap:O(logn)
  - replacing an estimated distance a new one : O(logn) (x O(m/n)times in the worst case)
  - repeat n rounds : O((m+n)logn) (or O(nlogn) for a sparse network)

maximum flows and minimum cuts

- augmenting path algorithm (Ford-Fulkerson) : $O((m+n)m/n)$

- basic idea:
  - find a path from s to t using BFS
  - find another using only the edges not used in the path
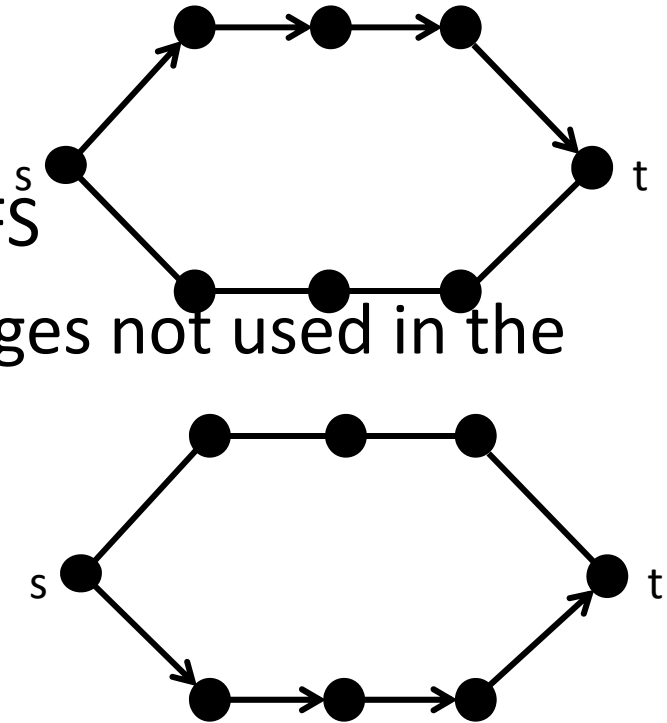
augmenting path algorithm

- the procedure will not always find the maximum flow

  but there are two edge independent paths

  – if we remove the edges of the shortest path, there is no more paths
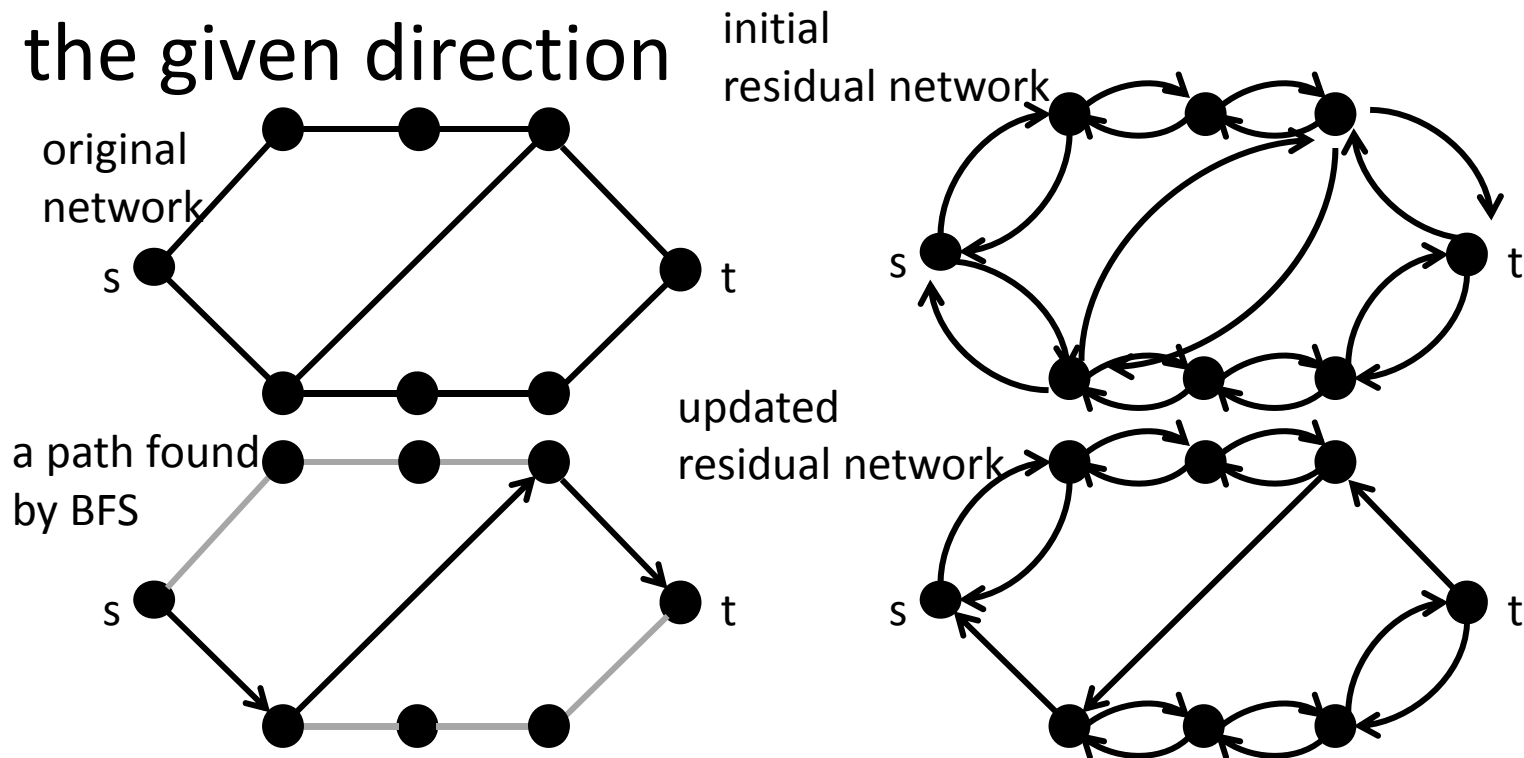
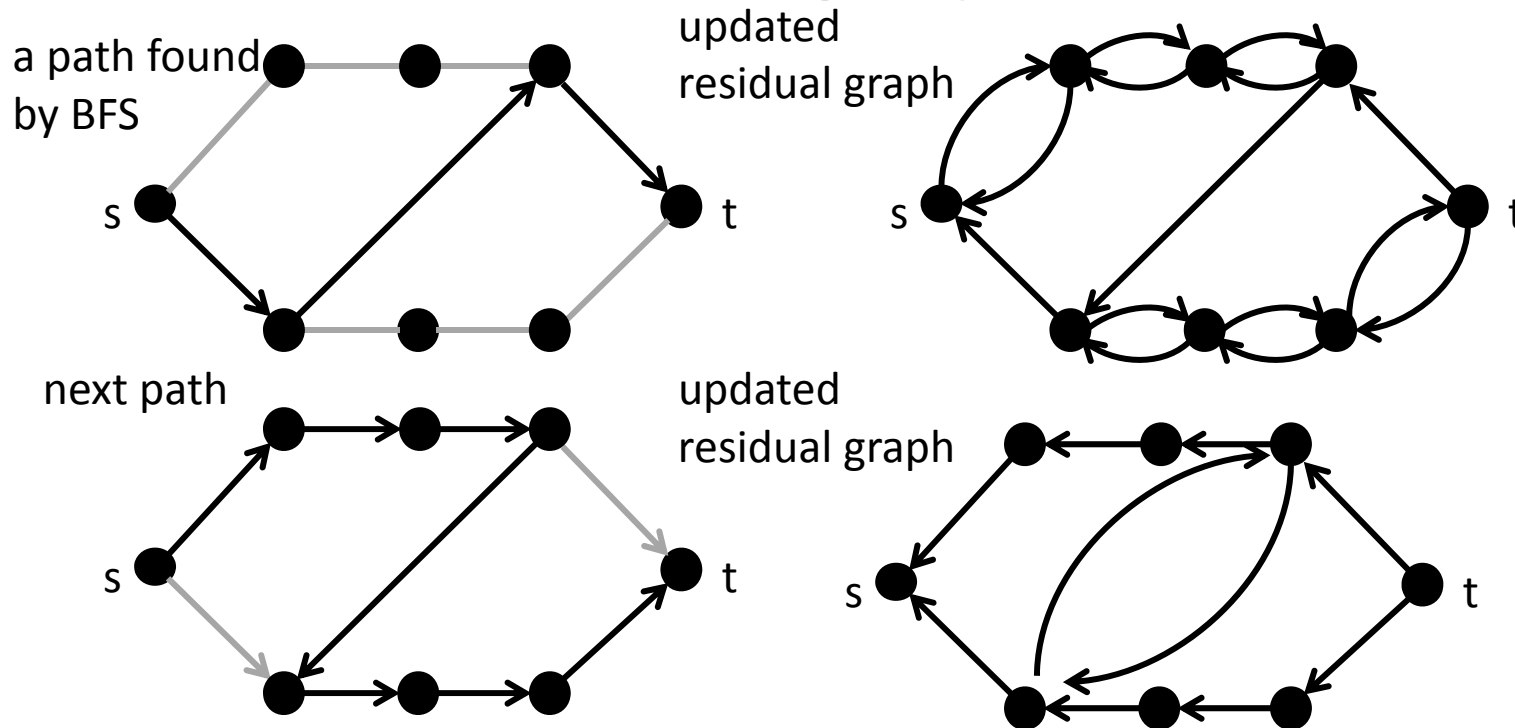- if we allow flows in both directions along an edge, we can find another

- the maximum possible flow is 2
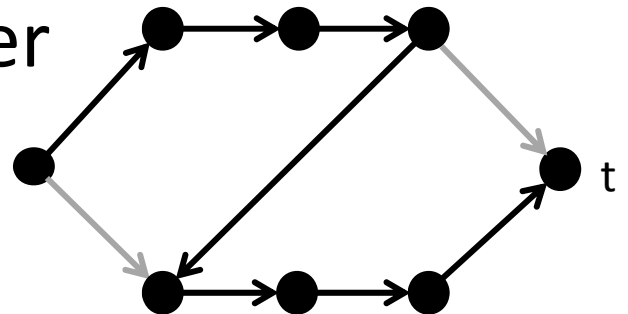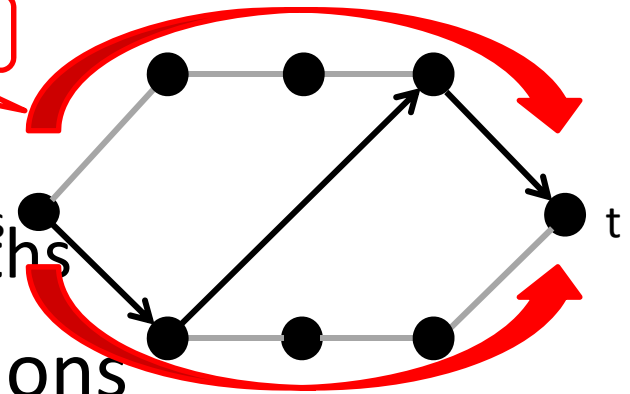
residual graph (1)

- a directed network in which the edges connect the pairs of vertices on the original network between which we still have capacity available to carry one or more units of flow in the given direction
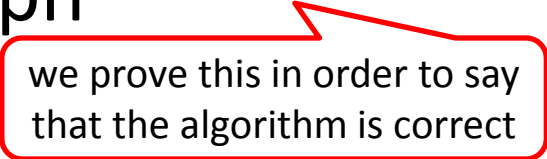
original network

initial residual network

s   t

a path found by BFS

updated residual network

s   t

residual graph (2)

a path found by BFS

updated residual graph

next path

updated residual graph

- the largest number of edges we update : O(m)
- BFS : O(m+n)
- the number of independent paths ≤ min(k_s,k_t)
- time complexity of this algorithm: O(min(k_s,k_t)(m+n))
- average running time: O((m+n)m/n) $\because \langle \min(k_s, k_t) \rangle \leq \langle k \rangle = 2m/n$

O(n) on a sparse network with m≈n

why augmenting path algorithm correctly finds maximum flows

- if at some point in our algorithm the flow from s to t is less than the maximum possible flow, then there must exist at least one augmenting path on the current residual graph

  we prove this in order to say that the algorithm is correct

- f: flows on the network

- f_{max}: maximum possible flow

- $\Delta f = f_{max} - f$

correctness of the augmenting path algorithm

maximum flow (f_{max})

submaximal flow (f)

difference flow (Δf)
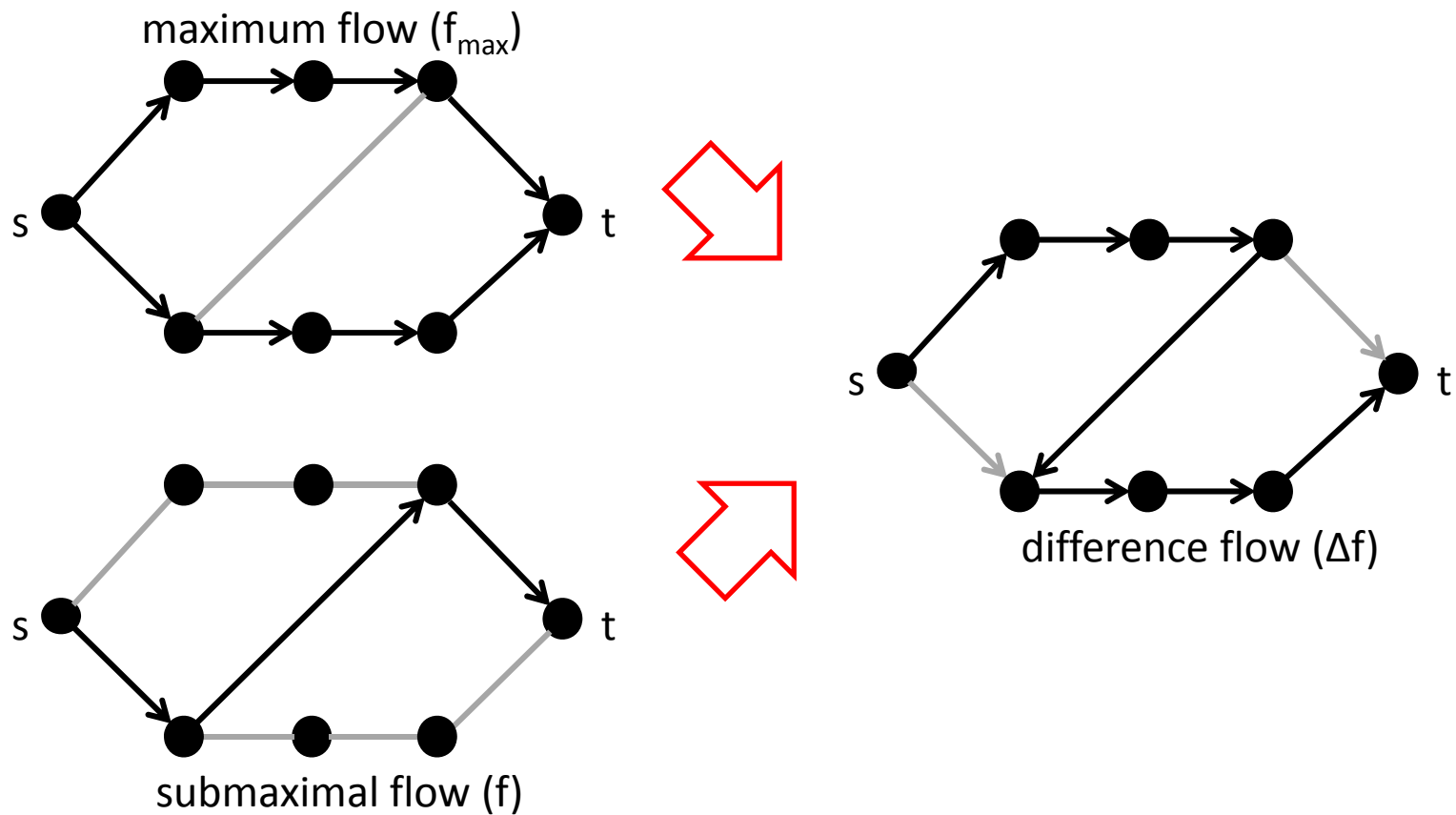
finding independent paths

- augmenting paths ≠ independent paths
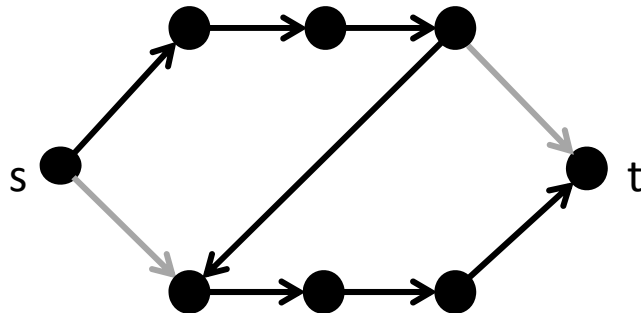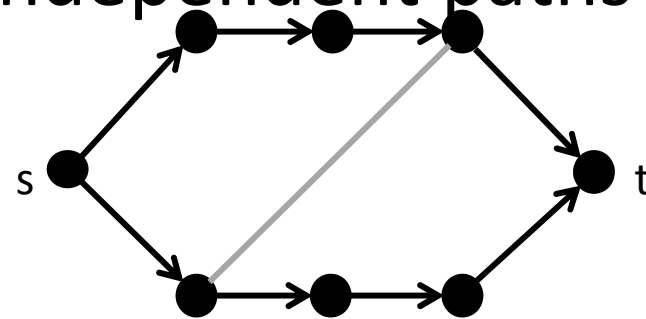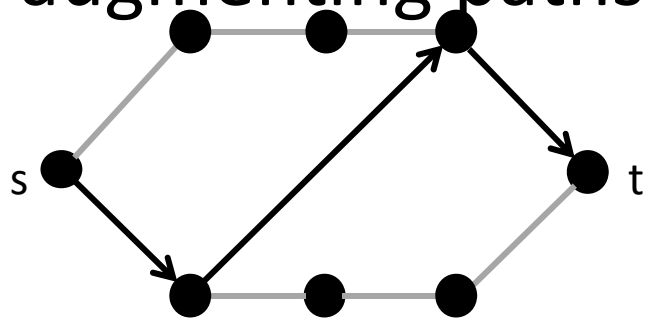
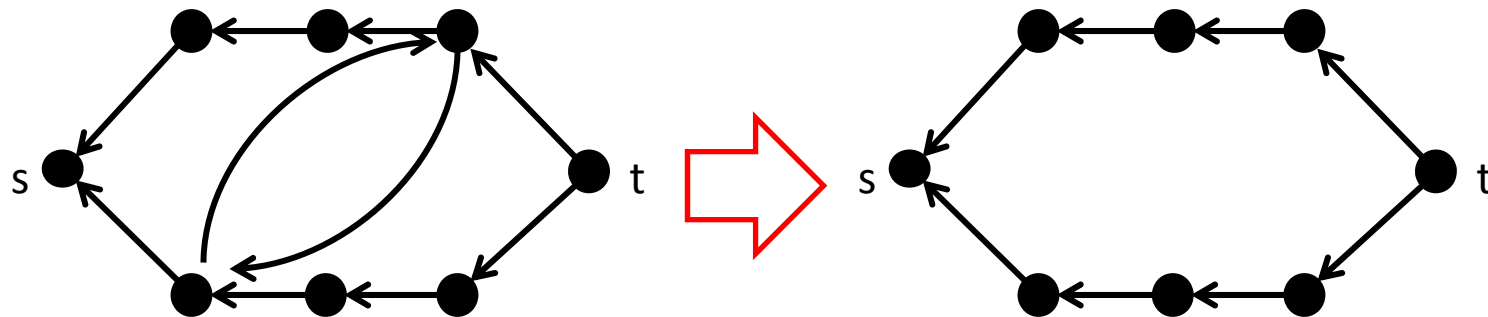every vertex (except s and t) must have zero net flow

the flow must leave every vertex except t

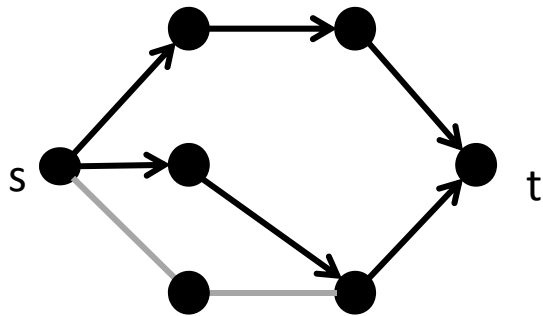this path (p) is different from other flows (f)

reconstructing the independent paths from the residual graph

- deleting every pair of edges that join the same two vertices in opposite directions -> graph consisting of the independent paths only
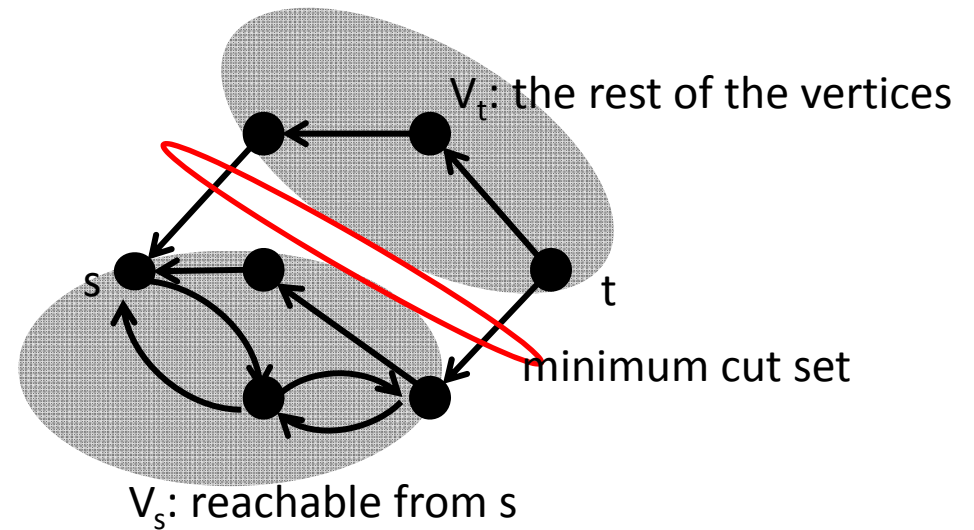
finding a minimum cut set

- ## maximum flows

residual graph

V_t: the rest of the vertices

minimum cut set

V_s: reachable from s

finding vertex-independent paths

- almost the same as augmenting path algorithm
  - but no two paths may pass through the same vertex

- mapping from the vertex-independent path problem to the edge-independent path problem
  - replace each vertex (except for s and t) with a pair of vertices with a directed edge between them

vertex transformation for the vertex-independent path algorithm

- each vertex is replaced by a pair of vertices joined by a single directed edge