Weeks 3, 4, and 5
Graph Algorithms
and Maximum Flow Networks

This week

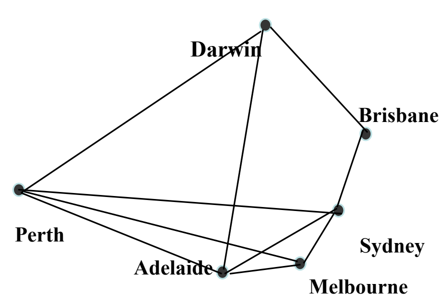
- Graph terminology
- Stacks and Queues
- Breadth-first-search
- Depth-first-search
- Connected Components
- Analysis of BFS and DFS Algorithms

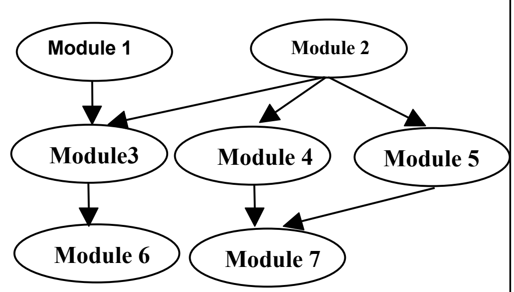
Further Reading

Chapter 22 .. 26 from Textbook

Graph Preliminaries

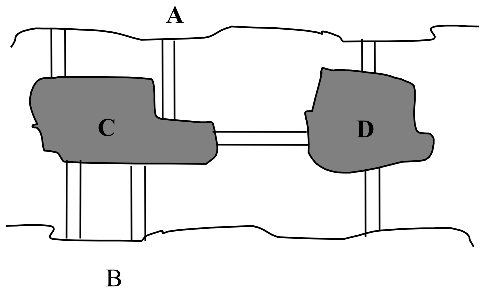
Examples of modeling by Graphs

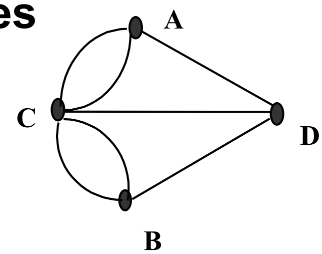




Konigsberg bridges

Konigsberg bridges



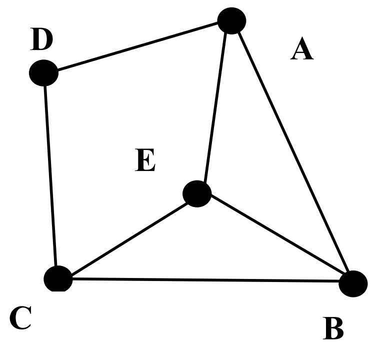


The town of Konigsberg (now Kaliningrad) lay on the banks and on two islands of the Pregel river. The city was connected by 7 bridges.
The puzzle (as encountered by Leonhard Euler in 1736) :
Whether it was possible to start walking from anywhere in town and return to the starting point by crossing all bridges exactly once.

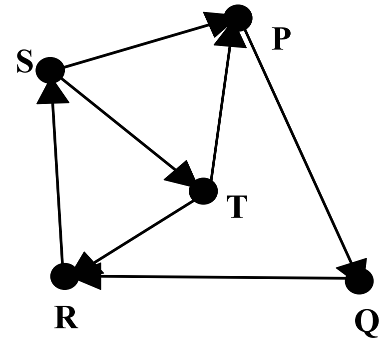
Graph Terminologies

- **A Graph consists of a set 'V' of vertices (or nodes) and a set 'E' of edges (or links).**
- **A graph can be directed or undirected.**
- **Edges in a directed graph are ordered pairs.**
  - **The order between the two vertices is important.**
    - **Example: (S,P) is an ordered pair because the edge starts at S and terminates at P.**
    - **The edge is unidirectional**
    - **Edges of an undirected graph form unordered pairs.**
- **A multigraph is a graph with possibly several edges between the same pair of vertices.**
- **Graphs that are not multigraphs are called simple graphs.**

Graph Terminologies (Contd)



G1 :Undirected Graph



G2: Directed Graph

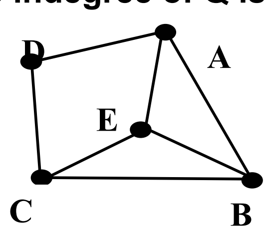
Graph Terminologies

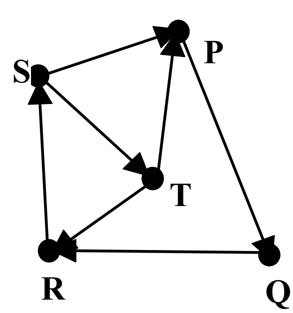
The degree *d*(*v*) of a vertex *v* is the number of edges incident to *v*.

d (A) = three, d (D) = two

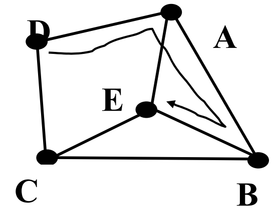
In directed graphs, indegree is the number of incoming edges at the vertex and outdegree is the number of outgoing edges from the vertex.

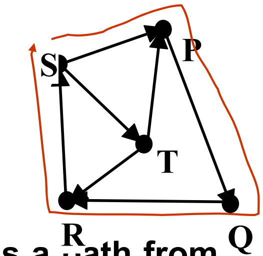
The indegree of P is 2, its outdegree is 1.
The indegree of Q is 1, its outdegree is 1.





Paths and Cycles





A path from vertex v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1,v_2), (v_2,v_3), \ldots, (v_{k-1},v_k)$.

Path from D to E : (D,A,B,E)

Edges in the path : (D,A), (A,B), (B,E)

A path is simple if each vertex in it appears only once.

DABE is a simple path.

ABCDAE is not a simple path.

Vertex u is said to be reachable from v if there is a path from v to u.

A circuit is a path whose first and last vertices are the same.

DAEBCEAD, ABEA, DABECD, SPQRS, STRS are circuits

Paths and Cycles

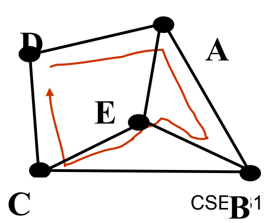
A simple circuit is a cycle if except for the first (and last) vertex, no other vertex appears more than once.

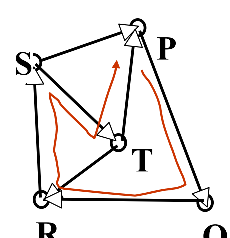
ABEA, DABECD, SPQRS, and STRS are cycles.

A Hamiltonian cycle of a graph G is a cycle that contains all the vertices of G

DABECD is a Hamiltonian cycle of G1

PQRSTP is a Hamiltonian of G2.

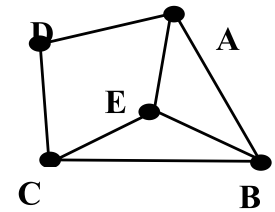


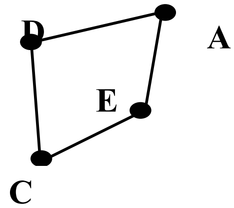


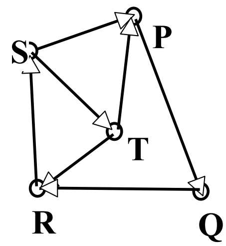
A **subgraph** of a graph G = (V,E) is a graph H(U,F) such that U $\subseteq$ V and F$\subseteq$E.

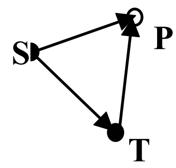
H1 {[U1:A,E,C,D], F1:[(A,E),(E,C),(C,D),(D,A)]} is a subgraph of G1

H2 {[U2:S,P,T],F2:[(S,P),(S,T),(T,P)]} is a subgraph of G2.



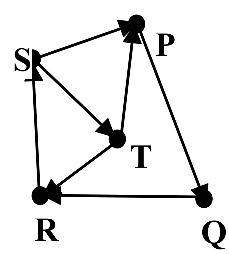




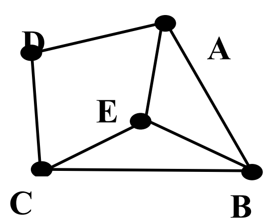


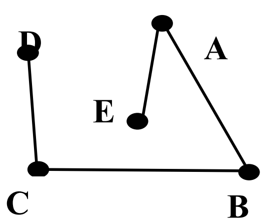
Spanning tree of G1

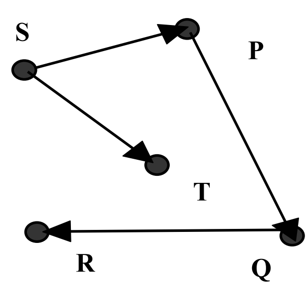
Spanning Tree



A **spanning tree** of a graph G is a subgraph of G that is a tree and contains all the vertices of G.







Spanning tree of G2

Connectivity

A graph is said to be connected if there is a path from any vertex to any other vertex in the graph.

G1 and G2 are both connected graphs

A forest is a graph that does not contain a cycle.

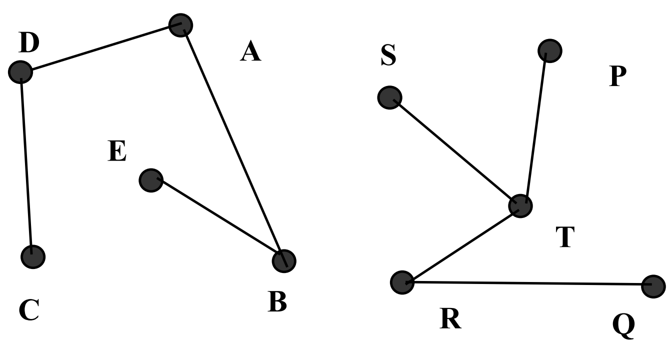
A tree is a connected forest.

A spanning forest of an undirected graph G is a subgraph of G that is a forest and contains all the vertices of G.

If a graph G(V,E) is not connected, then it can be partitioned in a unique way into a set of connected subgraphs called connected components.

A connected component of G is a connected subgraph of G such that no other connected subgraph of G contains it.

Forest



G(A,B,C,D,E,P,Q,R,S,T) is a forest

G(A,B,C,D,E) is a tree

(A,B,C,D,E) and (P,Q,R,S,T) are connected components

Graph Representations

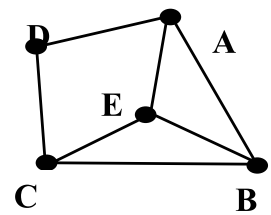
G1: undirected graph

Adjacency Matrix

	A	B	C	D	E
A	0	1	0	1	1
B	1	0	1	0	1
C	0	1	0	1	1
D	1	0	1	0	0
E	1	1	1	0	0

Adjacency list

A	B	D	E
B	A	C	E
C	B	D	E
D	A	C	\
E	A	B	C



Graph Representations

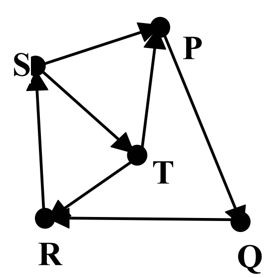
G2: Directed Graph

Adjacency matrix

	P	Q	R	S	T
P	0	1	0	0	0
Q	0	0	1	0	0
R	0	0	0	1	0
S	1	0	0	0	1
T	1	0	1	0	0

Adjacency list

P	Q	/
Q	R	/
R	S	/
S	P	T
T	P	R



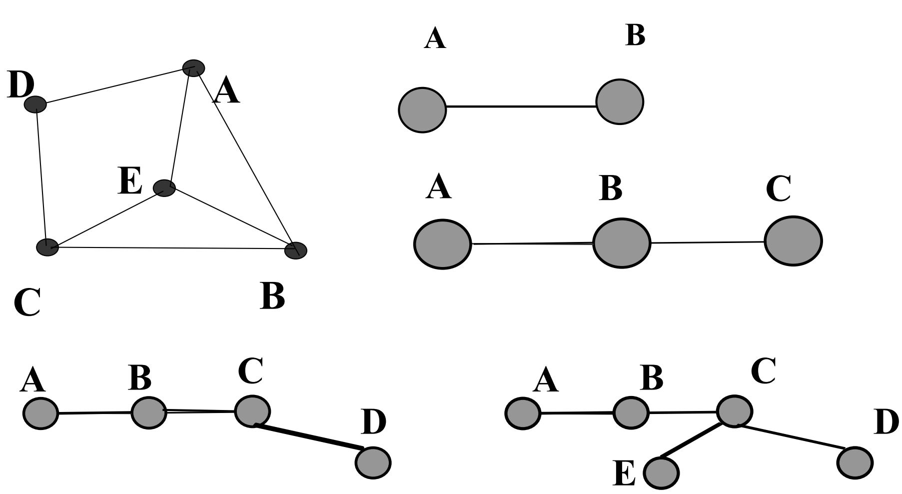
Depth-first search

Procedure DFS_Tree G(V,E)
Input: G = (V,E); S is a stack - initially empty;
'x' refers to the top of stack;
initially mark all vertices 'new';
L[x] refers to the adjacency list of x.
$T \leftarrow \{0\}$;
Output : The DFS tree T;

$$O(|V| + |E|)$$

```
1.      v ←old;  v∈ V
2.      push (S,v);
3.      while  S is nonempty do
4.          while  there exists a vertex w in L[x] and marked new  do
5.              T ← T ∪ (x,w) ;
6.              w ← old;
7.              push w onto S
8.          pop S
```

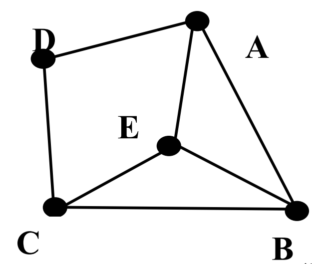

DFS



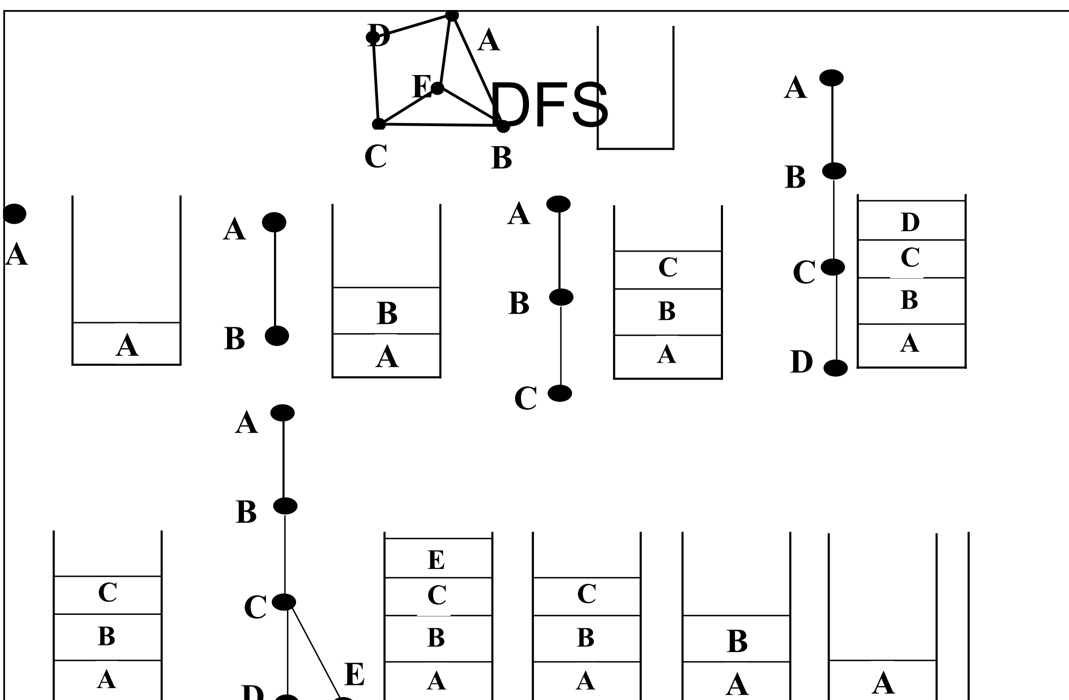
DFS

Initially, T = {0}; S {0}, A,B,C,D,E (all new)
Starts at A : A, S : {A}, L[A] = {B,D,E}

- **Pick B from L[A]; T = {(A,B)} and B (it's marked old}**
- **S = {A,B}, L[B] = {A,C,E}**
- **Pick C from L[B]; T = {(A,B), (B,C)} and C**
- **S = {A,B,C}; L[C] = {B,D,E}**
- **Pick D from L[C] ; T = {(A,B), (B,C), (C,D)} and D**
- **S = { A,B,C, D} ; L[D] ={A,C}; no new vertices;**
- **S = { A,B,C}; L[C] = { B,D,E}**
- **Pick E from L[C]; T ={ (A,B), (B,C), (C,D),(C,E)} and E**
- **S = { A,B,C,E} ; L[E] = {A,B,C}**
- **S = { A,B,C}; L[C] = { B,D,E}**
- **S ={ A,B} ; L[B]= { A,C,E }**
- **S ={A} ; L[A] = { B,C,E}**
- **S = {0}**



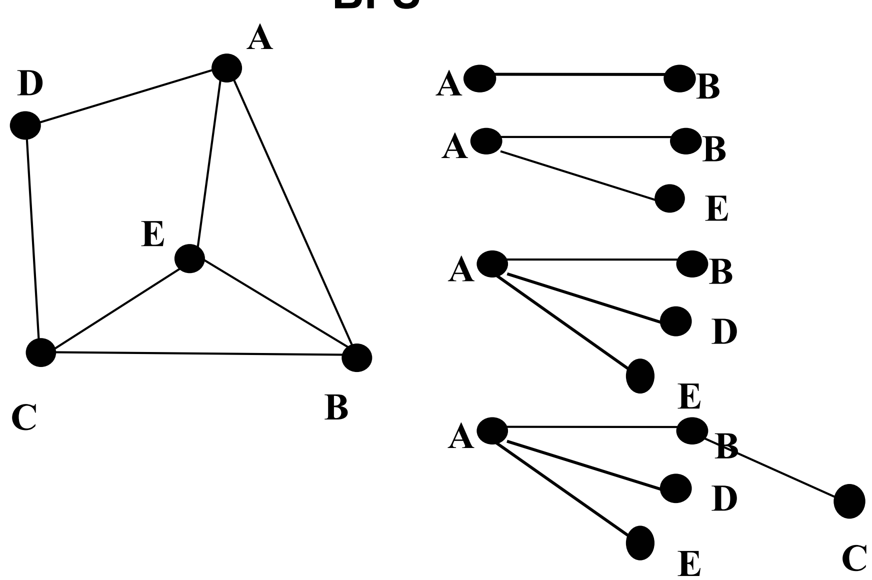
DFS



Breadth-first search

Procedure BFS_Tree G(V,E)
Input: G = (V,E); Q is a queue - initially empty;
x ← Q : remove the front item of queue and denote it by x;
initially mark all vertices 'new';
L[x] refers to the adjacency list of x.
T ← {0}
Output: The BFS tree T;
1. v ←old; v∈ V
2. insert (Q,v);
3. **while** Q is nonempty **do**
4. x ← Q
5. **for** each vertex w in L[x] and marked 'new'
6. T ← T ∪ {x,w} ;
7. w ← old;
8. insert (Q,w);

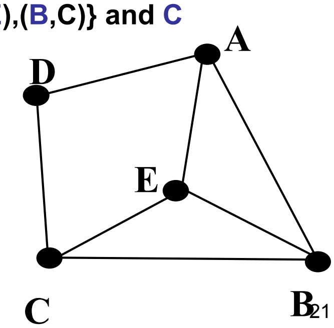
BFS

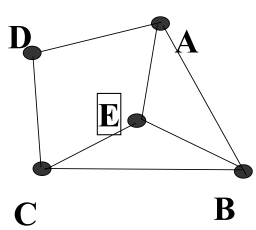


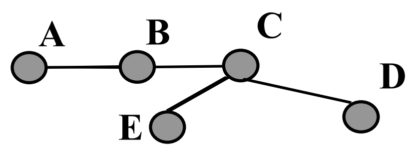
BFS

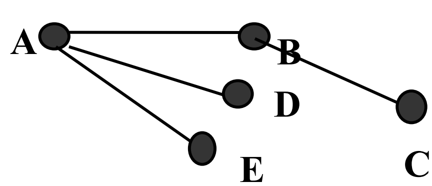
Initially, T = {0}; Q {0}, A,B,C,D,E (all new)
Starts at A : A, Q : {A}, L[A] = {B,D,E}

Pick B from L[A]; T = {(A,B)} and B (it's marked old}
Q = {B}, L[A] = {B,D,E}
Pick D from L[A]; T = {(A,B), (A,D)} and D
Q = {B,D}; L[A] = {B,D,E}
Pick E from L[A] ; T = {(A,B), (A,D), (A,E)} and E
Q = { B,D,E} ; L[A] ={B,D,E}; no new vertices;
Dequeue, Q = {D,E} L[B] = { A,C,E};
Pick C from L[B]; T ={ (A,B), (A,D), (A,E),(B,C)} and C
Q = {E, C} ; L[D] = {A,C}
Q = {C} ; L[E] = {A,B,C}
Q = { 0) ; L[C] = (B,C,E)
Q = {0};









Connected Components of a Graph

The connected component of a graph G = (V,E) is a maximal set of vertices $U \subseteq V$ such that for every pair of vertices u and v in U, we have both u and v reachable from each other. In the following we give an algorithm for finding the connected components of an undirected graph.

```
Procedure Connected_Components G(V,E)
Input : G (V,E)
Output : Number of Connected Components and G1, G2 etc, the
connected components
1.      V' ← V;
2.      c ← 0;
3.      while  V' ≠ 0 do
4.              choose  u ∈ V' ;
5.              T ← all nodes reachable from u (by DFS_Tree)
6.              V' ←V' - T;
7.              c ← c+1;
8.              G_c ← T;
9.              T ←  0;
```

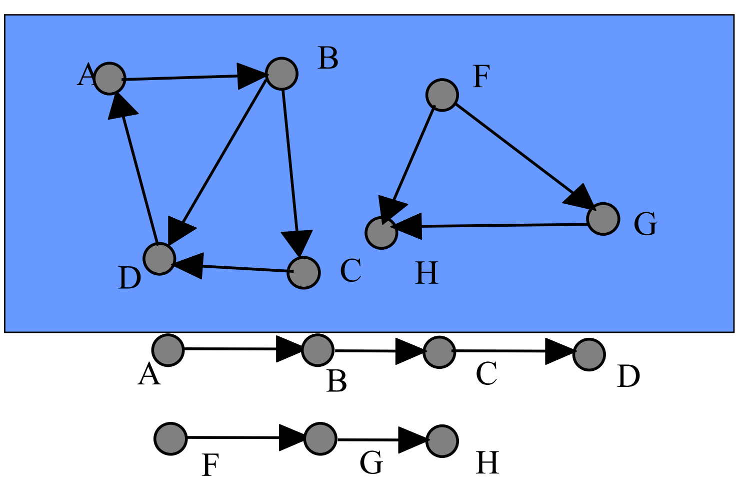

Suppose the DFS tree starts at A, we traverse from $A \rightarrow B \rightarrow C \rightarrow D$ and do not explore the vertices F, G, and H at all! The DFS_tree algorithm does not work with graphs having two or more connected parts.

We have to modify the DFS_Tree algorithm to find a DFS forest of the given graph.

DFS Forest

```
Procedure DFSForest _G(V,E)
Input: G = (V,E); S is a stack - initially empty;
        'x' refers to the top of stack; initially mark all vertices 'new';
        L[x] refers to the adjacency list of x.
        F ← {0}; The DFS Forest
Output: The DFS tree F;
1.      For each vertex v ∈ V do
2.          if v is new
3.              v ←old;
4.              push (S,v);
5.              while S is nonempty do
6.                  while there exists a vertex w in L[x] and marked
                                                          new do
7.                      F ← F ∪ (x,w) ;
8.                      w ← old;
9.                      push w onto S
10.                 pop S
```


DFS Forest



- Do you know the difference between a simple graph and a multiple graph?
- What is an adjacency matrix ?
- What is a Hamiltonian path? What is an Euler path?
- Given a graph, can you find the Hamiltonian and Eulerian paths?
- Given a graph, can you perform DFS and BFS traversals?
- What is the difference between a cycle and a path?
- What are the complexities of basic operations on stacks and queues? Give proof.

Minimum-Cost Spanning Trees

Consider a network of computers connected through bidirectional links. Each link is associated with a positive cost: the cost of sending a message on each link.

This network can be represented by an undirected graph with positive costs on each edge.

In bidirectional networks we can assume that the cost of sending a message on link does not depend on the direction.

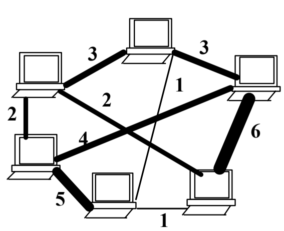
Suppose we want to broadcast a message to all the computers from an arbitrary computer.

The cost of the broadcast is the sum of the costs of links used to forward the message.

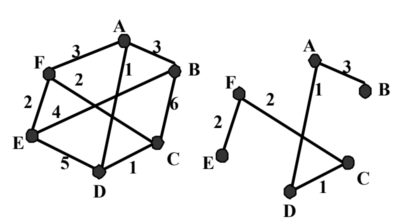
Minimum-Cost Spanning Trees

- Find a fixed connected subgraph, containing all the vertices such that the sum of the costs of the edges in the subgraph is minimum. This subgraph is a tree as it does not contain any cycles.
- Such a tree is called the spanning tree since it spans the entire graph G.
- A given graph may have more than one spanning tree
- The minimum-cost spanning tree (MCST) is one whose edge weights add up to the least among all the spanning trees

MCST



A Local Area Network



The equivalent Graph and the MCST

MCST

- **The Problem**: Given an undirected connected weighted graph G =(V,E), find a spanning tree T of G of minimum cost.

- **Greedy Algorithm for finding the Minimum Spanning Tree of a Graph G =(V,E)**

The algorithm is also called **Kruskal's** algorithm**.**

- At each step of the algorithm , one of several possible choices must be made,
- The greedy strategy: make the choice that is the best at the moment

Kruskal's Algorithm

- Procedure **MCST_G(V,E)**
- (Kruskal's Algorithm)
- **Input**: An undirected graph G(V,E) with a cost function c on the edges
- **Output**: T the minimum cost spanning tree for G
- T $\leftarrow$ 0;
- VS $\leftarrow$0;
- **for** each vertex v $\in$ V **do**
-         VS = VS $\cup$ {v};
- sort the edges of E in nondecreasing order of weight
- **while** |VS| > 1 **do**
-         choose (v,w) an edge E of lowest cost;
-         delete (v,w) from E;
-         **if** v and w are in different sets W1 and W2 in VS **do**
-                 W1 = W1 $\cup$ W2;
-                 VS = VS - W2;
-                 T $\leftarrow$ T$\cup$ (v,w);
- return T

MCST

- The algorithm maintains a collection VS of disjoint sets of vertices
- Each set W in VS represents a connected set of vertices forming a spanning tree
- Initially, each vertex is in a set by itself in VS
- Edges are chosen from E in order of increasing cost, we consider each edge (v, w) in turn; v, w $\in$ V.
- If v and w are already in the same set (say W) of VS, we discard the edge
- If v and w are in distinct sets W1 and W2 (meaning v and/or w not in T) we merge W1 with W2 and add (v, w) to T.

MCST

Consider the example graph shown earlier,

The edges in nondecreasing order

[(A,D),1],[(C,D),1],[(C,F),2],[(E,F),2],[(A,F),3],[(A,B),3],

[(B,E),4],[(D,E),5],[(B,C),6]

EdgeActionSets in VSSpanning Tree, T =[{A},{B},{C},{D},{E},{F}]{0}(A,D)merge

[{A,D}, {B},{C}, {E}, {F}] {(A,D)} (C,D) merge

[{A,C,D}, {B}, {E}, {F}] {(A,D), (C,D)} (C,F) merge

[{A,C,D,F},{B},{E}]{(A,D),(C,D), (C,F)} (E,F) merge

[{A,C,D,E,F},{B}]{(A,D),(C,D), (C,F),(E,F)}(A,F) reject

[{A,C,D,E,F},{B}]{(A,D),(C,D), (C,F), (E,F)}(A,B) merge

[{A,B,C,D,E,F}]{(A,D),(A,B),(C,D), (C,F),(E,F)}(B,E) reject

(D,E) reject

(B,C) reject

Complexity

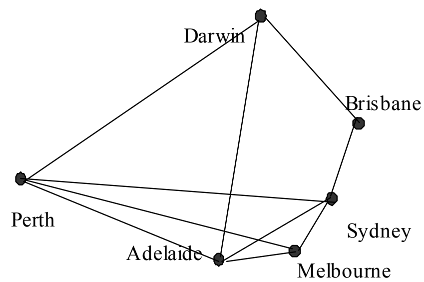
- Steps 1 thru 4 take time O (V)
- Step 5 sorts the edges in nondecreasing order in O (E log E) time
- Steps 6 through 13 take O (E) time
- The total time for the algorithm is therefore given by O (E log E)
- The edges can be maintained in a heap data structure with the property,
- E[PARENT(i)] $\leq$ E[i]
- remember, this property is the opposite of the one used in the heapsort algorithm earlier during Week 2. This property can be used to sort data elements in nonincreasing order.
- Construct a heap of the edge weights, the edge with lowest cost is at the root
- During each step of edge removal, delete the root (minimum element) from the heap and rearrange the heap.
- The use of heap data structure reduces the time taken because at every step we are only picking up the minimum or root element rather than sorting the edge weights.

Week 4

- Single Source Shortest Paths
- All Pairs Shortest Path Problem

Single-Source Shortest Paths

A motorist wishes to find the shortest possible route from from Perth to Brisbane. Given the map of Australia on which the distance between each pair of cities is marked, how can we determine the shortest route?



Single Source Shortest Path

- In a shortest-paths problem, we are given a weighted, directed graph G = (V,E), with weights assigned to each edge in the graph. The weight of the path p = (v0, v1, v2, …, vk) is the sum of the weights of its constituent edges:
- v0 → v1 → v2 . . . → vk-1→ vk
-
- The shortest-path from u to v is given by
- d(u,v) = min {weight (p) : if there are one or more paths from u to v
- = ∞ otherwise

The single-source shortest paths problem

Given G (V,E), find the shortest path from a given vertex $u \in V$ to every vertex $v \in V$ ($u \neq v$).

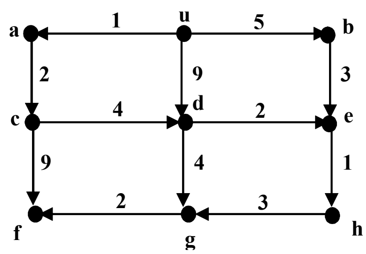
For each vertex $v \in V$ in the weighted directed graph, d[v] represents the distance from u to v.

Initially, d[v] = 0 when u = v.
d[v] = ∞ if (u,v) is not an edge
d[v] = weight of edge (u,v) if (u,v) exists.

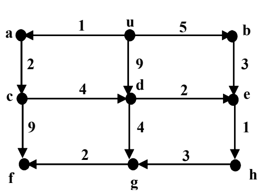
Dijkstra's Algorithm : At every step of the algorithm, we compute,
$d[y] = \min \{d[y], d[x] + w(x,y)\}$, where $x,y \in V$.

Dijkstra's algorithm is based on the greedy principle because at every step we pick the path of least weight.

- Dijkstra's Algorithm : At every step of the algorithm, we compute,
  $d[y] = \min \{d[y], d[x] + w(x,y)\}$, where $x,y \in V$.
- Dijkstra's algorithm is based on the greedy principle because at every step we pick the path of least path.



Example:



Step #	Vertex to be marked	Distance to vertex									Unmarked vertices
		u	a	b	c	d	e	f	g	h	
0	u	0	1	5	∞	9	∞	∞	∞	∞	a,b,c,d,e,f,g,h
1	a	0	1	5	3	9	∞	∞	∞	∞	b,c,d,e,f,g,h
2	c	0	1	5	3	7	∞	12	∞	∞	b,d,e,f,g,h
3	b	0	1	5	3	7	8	12	∞	∞	d,e,f,g,h
4	d	0	1	5	3	7	8	12	11	∞	e,f,g,h
5	e	0	1	5	3	7	8	12	11	9	f,g,h
6	h	0	1	5	3	7	8	12	11	9	g,h
7	g	0	1	5	3	7	8	12	11	9	h
8	f	0	1	5	3	7	8	12	11	9	--

Dijkstra's Single-source shortest path

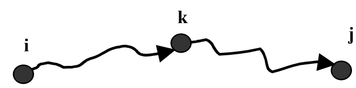
- Procedure **Dijkstra's Single-source shortest path_G(V,E,u)**
- Input: $G = (V,E)$, the weighted directed graph and v the source vertex
- Output: for each vertex, v, $d[v]$ is the length of the shortest path from u to v.
- mark vertex u;
- $d[u] \leftarrow 0$;
- **for** each unmarked vertex $v \in V$ **do**
- **if** edge (u,v) exists d $[v] \leftarrow$ weight (u,v);
- **else** $d[v] \leftarrow \infty$;
- **while** there exists an unmarked vertex **do**
- let v be an unmarked vertex such that $d[v]$ is minimal;
- mark vertex v;
- **for** all edges (v,x) such that x is unmarked **do**
- if d[x] > d[v] + weight[v,x] **then**
- $d[x] \leftarrow d[v] + weight[v,x]$

- Complexity of Dijkstra's algorithm:
- Steps 1 and 2 take Θ (1) time
- Steps 3 to 5 take $O(|V|)$ time
- The vertices are arranged in a heap in order of their paths from u
- Updating the length of a path takes *O(log V)* time.
- There are $|V|$ iterations, and at most $|E|$ updates
- Therefore the algorithm takes $O((|E|+|V|) \log |V|)$ time.

All-Pairs Shortest Path Problem

Consider a shortest path p from vertex *i* to vertex *j*
If *i* =*j* then there is no path from *i* to *j*.
If $i \neq j$, then we decompose the path *p* into two parts,
dist(*i,k*) and dist(*k,j*)

dist (*i,j*) = dist(*i,k*) + dist(*k,j*)



Recursive solution

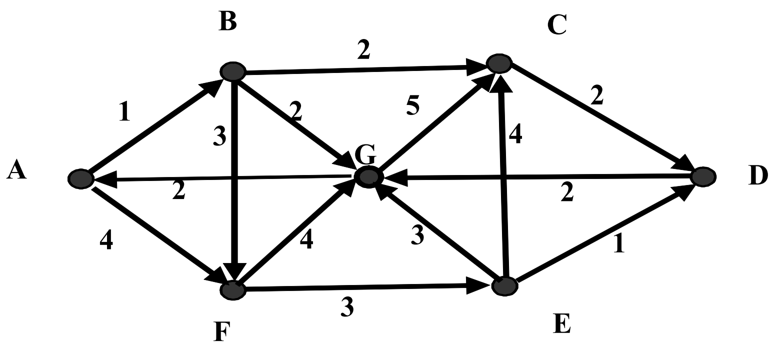
$$dist(i,j) = \begin{cases} w(i,j) & if\ k = 0 \\ \min\{dist(i,j), [dist(i,k) + dist(k,j)]\} & if\ k \geq 1 \end{cases}$$

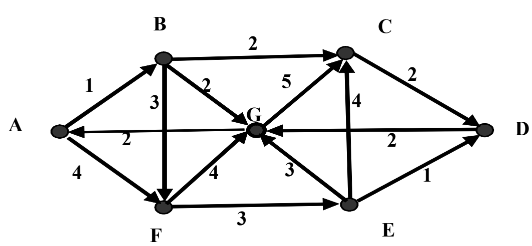
Floyd' s Algorithm for Shortest Paths

- Procedure **FLOYDs_G=[V,E]**
- **Input:** $n \times n$ matrix *W* representing the edge weights of an n-vertex directed graph. That is W =w(*i,j*) where, (Negative weights are allowed)
- **Output:** shortest path matrix, dist(*i,j*) is the shortest path between vertices *i* and *j*.

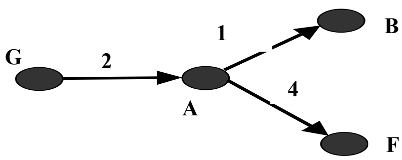
```
for v ← 1 to n do
    for w ← 1 to n do
        dist[v,w] ←arc[v,w];
for u ← 1 to n do
    for v ← 1 to n do
        for w ← 1 to n do
            if dist[v,u] + dist[u,w] < dist[v,w] then
                dist[v,w] ← dist[v,u] + dist[u,w]
```

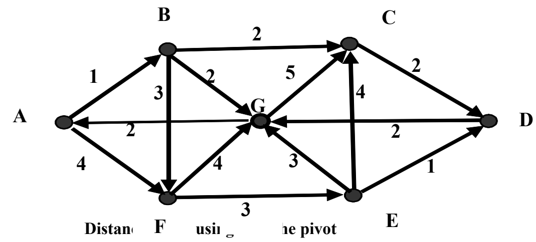
- Complexity : $\Theta(n^3)$



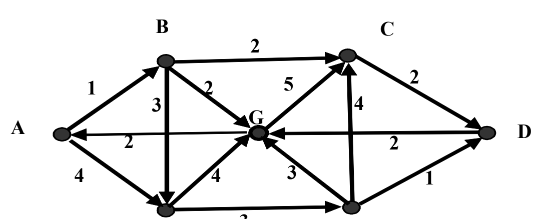


	A	B	C	D	E	F	G
A	**0**	**1**	∞	∞	∞	**4**	∞
B	∞	**0**	**2**	∞	∞	**3**	**2**
C	∞	∞	**0**	**2**	∞	∞	∞
D	∞	∞	∞	**0**	∞	∞	**2**
E	∞	∞	**4**	**1**	**0**	∞	**3**
F	∞	∞	∞	∞	**3**	**0**	**4**
G	**2**	∞	**5**	∞	∞	∞	**0**



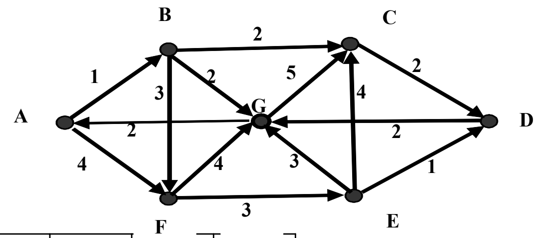


	A	B	C	D	E	F	G
A	0	1	∞	∞	∞	4	∞
B	∞	0	2	∞	∞	3	2
C	∞	∞	0	2	∞	∞	∞
D	∞	∞	∞	0	∞	∞	2
E	∞	∞	4	1	0	∞	3
F	∞	∞	∞	∞	3	0	4
G	2	3	5	∞	∞	6	0



	A	B	C	D	E	F	G
A	0	1	3	∞	∞	4	3
B	∞	0	2	∞	∞	3	2
C	∞	∞	0	2	∞	∞	∞
D	∞	∞	∞	0	∞	∞	2
E	∞	∞	4	1	0	∞	3
F	∞	∞	∞	∞	3	0	4
G	2	3	5	∞	∞	6	0

Distances after using B as the pivot



	A	B	C	D	E	.	G
A	0	1	3	5	7	4	3
B	4	0	2	4	6	3	2
C	6	7	0	2	13	10	4
D	4	5	7	0	11	8	2
E	5	6	4	1	0	9	3
F	6	7	7	4	3	0	4
G	2	3	5	7	9	6	0

Distances after using G as the pivot

Transitive Closure

- Given a directed graph G=(V,E), the transitive closure C =(V,F) of G is a directed graph such that there is an edge (v,w) in C if and only if there is a directed path from v to w in G.
- Security Problem: the vertices correspond to the users and the edges correspond to permissions. The transitive closure identifies for each user all other users with permission (either directly or indirectly) to use his or her account. There are many more applications of transitive closure.
- The recursive definition for transitive closure is

$$t(i,j) = \begin{cases} 0 \;\; if\; i \neq j \;\; and \;\; (i,j) \notin E \\ \;\; 1 \;\; f\; ij \;\; and \;\; (i,j) \in E \end{cases}$$

Warshall's Algorithm for Transitive Closure

- Procedure **WARSHALL's(G=[V,E])**
- **Input:** n×n matrix A representing the edge weights of an n-vertex directed graph. That is a =a(i,j) where,
- **Output:** transitive closure matrix, t(i,j) =1 if there is a path from i to j, 0 otherwise
- **for** v ← 1 to n **do**
-     **for** w ← 1 to n **do**
-             t[v,w] ← a(v,w)
- **for** u ← 1 to n **do**
-     **for** v ← 1 to n **do**
-         **for** w ← 1 to n **do**
-             **if NOT** t[v,w] **then**
-                 t[v,w] ← t[v,u] **AND** t[u,w]
- **return** T

- Hamiltonian Cycle
- Eulerian Path
- Biconnected Components
- Bipartite Graph Matching

Euler Circuit

- An Euler circuit of an undirected graph *G*(*V*,*E*) is a path that starts and ends at the same node and contains each edge of G exactly once.
- Show that a connected, undirected graph has an Euler circuit if and only if each node is of even degree.
- Let *G* (*V*,*E*) be an undirected graph with m edges in which every node is of even degree. Give an $O(|V|)$ algorithm to construct an Euler circuit for G.

Maximum Flow Networks

Topics

Flow Networks
Residual networks
Ford-Fulkerson's algorithm
- **Ford-Fulkerson's Algorithm**

<u>Further Reading</u>

Chapter 25 from Text book

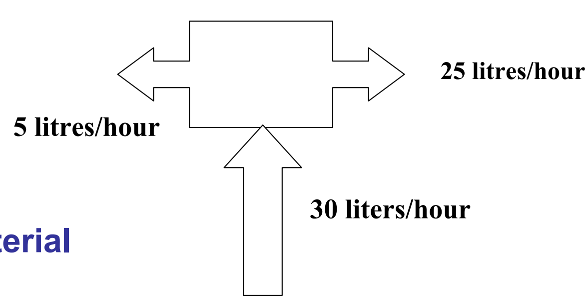
Flow Networks

A directed graph can be interpreted as a flow network to analyze material flows through networks.

Material courses through a system from a source (where it is produced) to a sink (where it is consumed).
Examples :
- **Water through pipelines**
- **Newspapers through distribution system**
- **Electricity through cables**
- **Cars on a production line**
- **on roads**

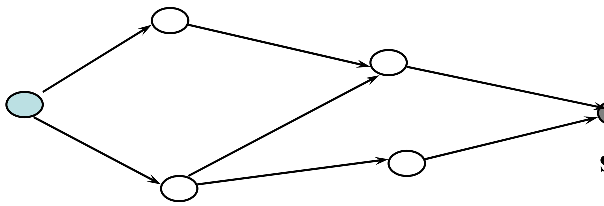
The source produces the material at a steady rate .
The sink consumes the material at a steady rate

Flow: the rate at which the material moves from one point to another
- **100 litres of water per hour in a pipe**
- **30 Amperes of electric current in a circuit**



The rate at which a material enters a vertex = the rate at which the material leaves the vertex

The flow network G =(V,E) is a directed graph in which each edge (u,v) $\in$ E has a nonnegative capacity c(u,v) $\geq$ 0.
If (u,v) $\notin$ E then c(u,v) = 0.
A flow network has a source vertex s, and a sink vertex t.
For every vertex v $\in$ V there is a path from s to v and v to t in a connected graph.



A flow in G is a real-valued function f : V $\times$ V $\rightarrow$ R that satisfies the following three properties:

1. Capacity constraint : For all u,v $\in$ V, we require f(u,v) $\leq$ c(u,v).
The net flow from one vertex to another must not exceed the given capacity.

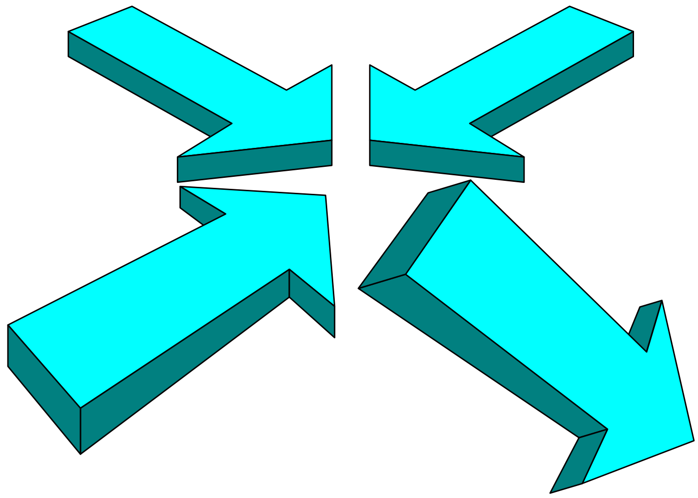
2. Skew symmetry : For all u,v $\in$ V, we require f(u,v) = -f(v,u).

The net flow from a vertex u to a vertex v is the negative of the net flow in the reverse direction.
The net flow from a vertex to itself is zero for all u $\in$ V, that is f(u,u) = 0.

3. Flow conservation : For all u$\in$ V - {s,t}, we require

$$\sum_{v \in V} f(u,v) = 0$$

The total net flow out of a vertex other than the source or sink is zero.



The quantity f(u,v) can be negative or positive, it is called the net flow from vertex u to v.

The value of a flow is defined as

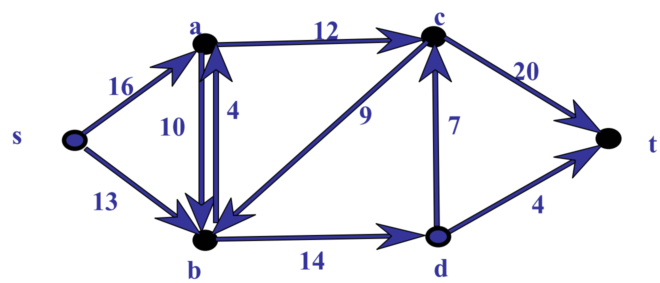
$$|f| = \sum_{v \in V} f(s,v)$$

In the maximum-flow problem, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

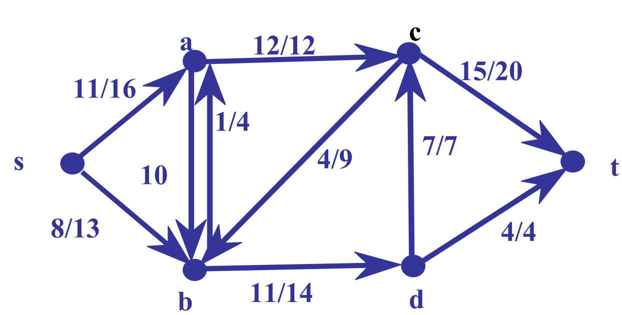
There is no net flow between u and v if there is no edge between them.
If (u,v) $\notin$ E and (v,u) $\notin$ E, then c(u,v) = c(v,u) = 0.
Hence, the capacity constraint, f(u,v) $\leq$ 0 and f(v,u) $\leq$ 0.
By skew symmetry, f(u,v) = -f(v,u),
therefore, f(u,v) + f(v,u) = 0.

Nonzero net flow from vertex u to vertex v implies that (u,v)$\in$E or (v,u)$\in$E (or both).

Consider the network G=(V,E) shown in the figure below. The network is for a transport system that transports crates of an item from source vertex s to sink vertex t through a number of intermediate points. Each edge (u,v) ∈ E in the network is labeled with its capacity c(u,v).



Let us consider a flow in G, |f|=19
If f(u,v) >0, edge (u,v) is labeled f(u,v)/c(u,v)
If f(u,v) ≤ 0, the edge is labeled by its capacity only.



The positive net flow entering a vertex v is defined by

$$\sum_{\substack{u \in V \\ f(u,v)>0}} f(u,v)$$

Initially, c(a,b) = 8, and c(b,a) = 3 as shown in Fig. a.
f(a,b) = 5 and f(b,a) = 2, the net flow is shown as 3/8 in direction a to b

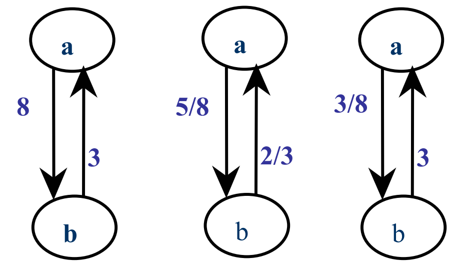


Fig.a Fig.b Fig.c

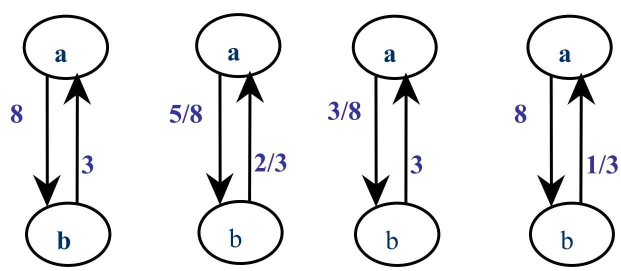


Fig.a Fig.b Fig.c Fig.d

f(a,b) = 5 and f(b,a) = 2, the net flow is shown as 5/8 in direction a to b and 2/3 in direction b to a as shown in Fig. b.
Then the equivalent flow is 3/8 in the direction a to b as shown in Fig. c.
If we increase the flow from b to a from 2 to 6 then the netflow is 1/3 in the direction b to a as shown in Fig. d.

The Ford Fulkerson method

The method is iterative,
Starts with f(u,v) for (u,v) $\in$ V, initial flow of value 0.
The method is based on the augmenting path which is defined as a path from s to t along which we can push more flow and then augment flow along this path.

Procedure Ford_Fulkerson_method(G,s,t)

1. **f $\leftarrow$ 0;**
2. **while there exists an augmenting path p**
3. **do augment flow along path p**
4. **return f**

Residual Networks

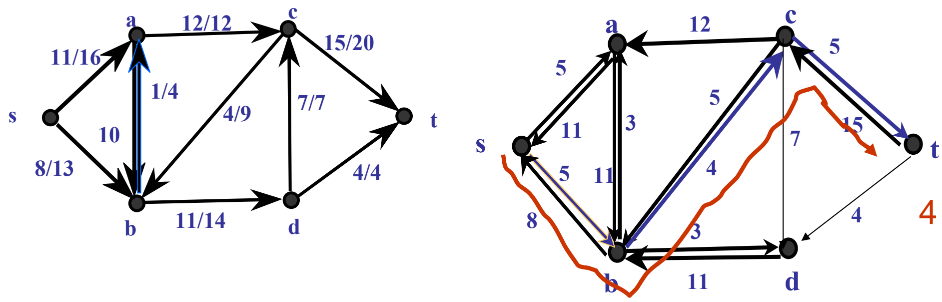
Consider a flow network G(V,E) with source s and sink t and let f be a flow in G.
Consider a pair of vertices u,v $\in$ V.
Residual capacity between u and v is given by

$$r(u,v) = c(u,v) - f(u,v)$$

- **the additional net flow we can push from u to v before exceeding the capacity.**
  For example, if c(u,v) = 25 and f(u,v) = 19, then r(u,v) = 6.

If f(u,v) < 0 then r(u,v) > c(u,v)

Given a flow network G=(V,E) and a flow f, the residual network of G induced by f is $G_f=(V,E_f)$,
where $E_f = \{(u,v) \in V \times V : r(u,v) > 0\}$



Each edge in the residual network can admit positive net flow only. The residual network may include several edges that are not in the original network, $(u,v) \in E_f$ and $(u,v) \notin E$ is possible (E_f is not a subset of E). However, (u,v) appears in G_f only if $(v,u) \in E$ and there is a positive flow from v to u. Because the net flow f(u,v) is negative,

$$r(u,v) = c(u,v)-f(u,v) > 0 \text{ and } (u,v) \in E_f$$

An edge (u,v) can appear in a residual network only if at least one of (u,v) and (v,u) appears in the original network.

$$|E_f| \leq 2|E|$$

Augmenting Paths

It is a simple path from s to t in G_f. Each edge (u,v) on an augmenting path admits some additional positive net flow from u to v without violating the capacity constraint on the edge. The residual capacity of a path p is given by,

$$r(p) = \min \{ r(u,v) : (u,v) \text{ is in } p \}$$

Let's define a flow function f_p,

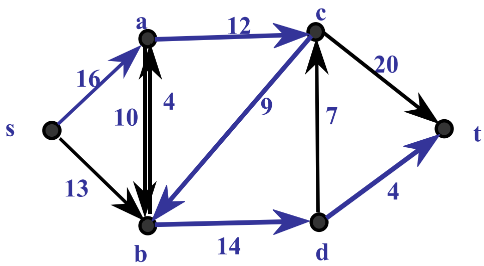
$$f_p = \begin{cases} r(p) & \text{if } (u,v) \text{ is on } p, \\ -r(p) & \text{if } (v,u) \text{ is on } p \\ 0 & \text{otherwise} \end{cases}$$

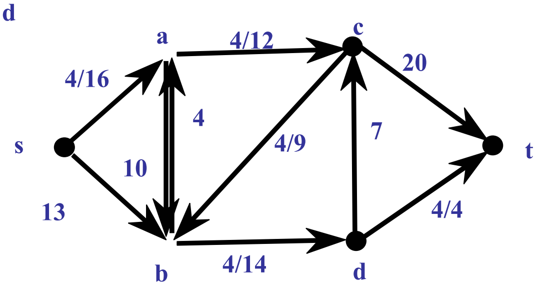
f_p is a flow in G_f with value $|f_p| = r(p) > 0$.
If we add f_p to f, we get another flow in G whose value is closer to the maximum.

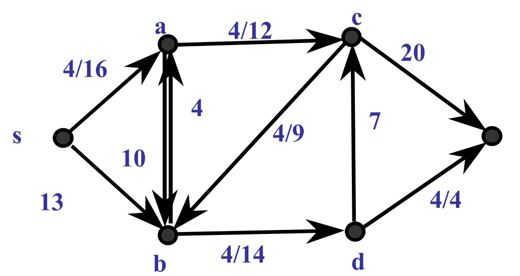
Algorithm

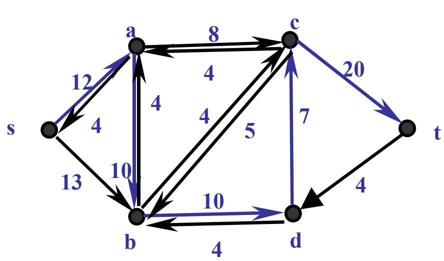
Procedure Ford-Fulkerson(G,s,t)
Input : Flow Network G(V,E)
Output : Maximum flow for the given network

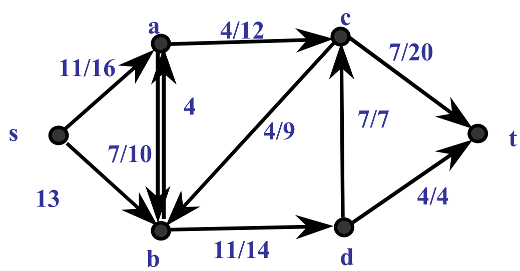
```
for each edge (u,v) ∈ E
    do    f[u,v] ← 0;
          f[v,u] ← 0;
while there exists a path p from s to t in the
                        residual network G_f
    do    r(p) ← min{r(u,v) : (u,v) is in p};
          for each edge (u,v) in p
                do    f[v,u] ← - f[u,v];
                      f[u,v] ← f[u,v] + r(p);
return
```

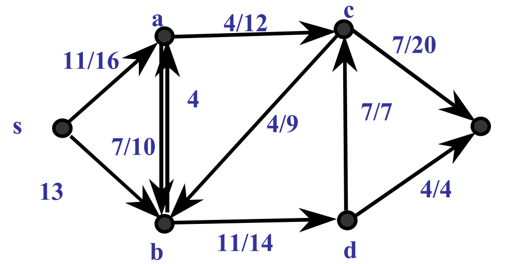

a
c
s
t
b
d
16
12
20
10
4
9
7
13
14
4


a
c
s
t
b
d
4/16
4/12
20
4
4/9
7
10
13
4/14
4/4

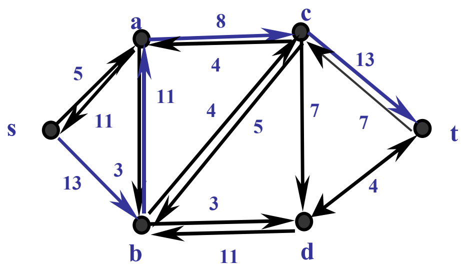

a
c
s
t
b
d
4/16
4/12
20
4
4/9
7
10
13
4/14
4/4


a
c
s
t
b
d
8
12
4
20
4
4
4
5
7
13
10
10
4
4

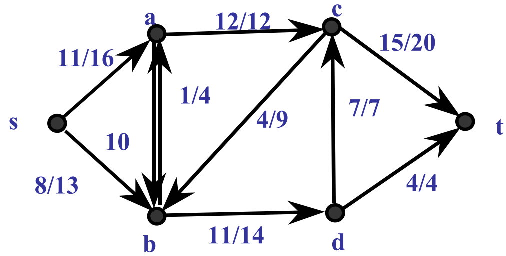

a
c
s
t
b
d
11/16
4/12
7/20
4
4/9
7/7
7/10
13
11/14
4/4



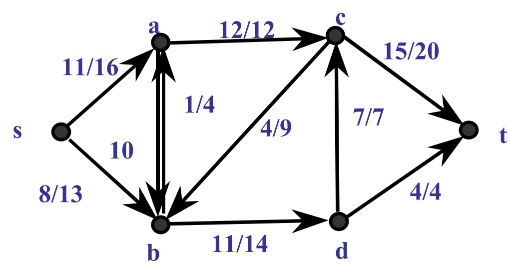
a
c
11/16
4/12
7/20
4
4/9
7/7
s
7/10
t
13
4/4
b
11/14
d



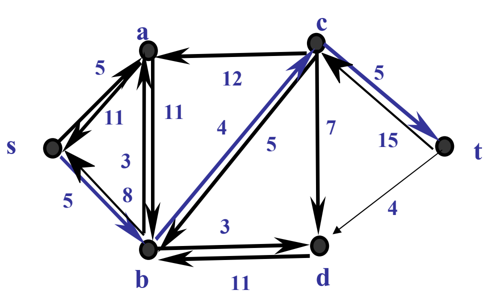
a
8
c
5
4
13
11
s
11
4
5
7
7
t
13
3
3
4
b
11
d



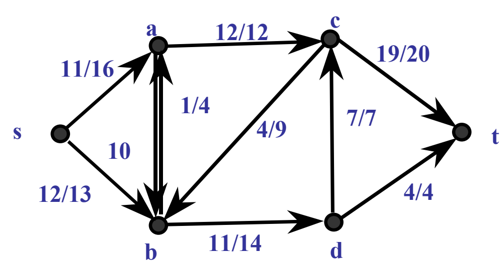
a
12/12
c
11/16
15/20
1/4
s
4/9
7/7
t
10
8/13
4/4
b
11/14
d



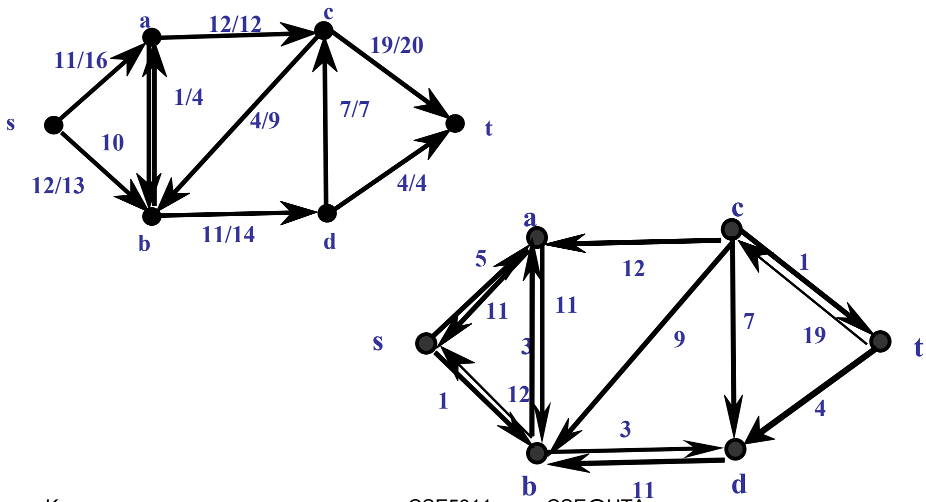
a
12/12
c
11/16
15/20
1/4
s
4/9
7/7
t
10
8/13
4/4
b
11/14
d



a
c
5
12
5
11
11
s
4
5
7
15
t
3
8
5
3
4
b
11
d



a
12/12
c
11/16
19/20
1/4
s
4/9
7/7
t
10
12/13
4/4
b
11/14
d


s
a
b
c
d
t
11/16
12/13
10
1/4
12/12
4/9
7/7
19/20
4/4
11/14
5
11
3
12
1
11
12
9
7
1
19
4
3
11