Module 3 - Graph Algorithms

This week

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

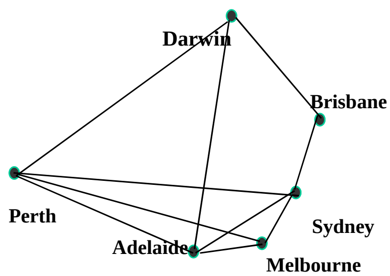
Please see Reference Books

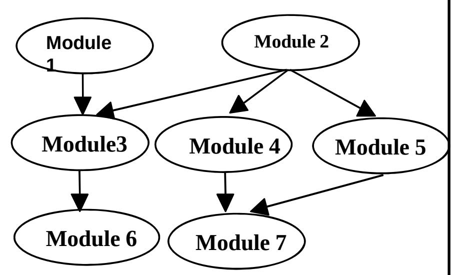
Course Syllabus

- Review of Asymptotic Analysis and Growth of Functions, Recurrences
- Sorting Algorithms
- **Graphs and Graph Algorithms.**
- Greedy Algorithms:
  - Minimum spanning tree,Union-Find algorithms, Kruskal's Algorithm,
  - Clustering,
  - Huffman Codes, and
  - Multiphase greedy algorithms.
- Dynamic Programming:
  - Shortest paths, negative cycles, matrix chain multiplications, sequence alignment, RNA secondary structure, application examples.
- Network Flow:
  - Maximum flow problem, Ford-Fulkerson algorithm, augmenting paths, Bipartite matching problem, disjoint paths and application problems.
- NP and Computational tractability:
  - Polynomial time reductions; The Satisfiability problem; NP-Complete problems; and Extending limits of tractability.
- Approximation Algorithms, Local Search and Randomized Algorithms

Graph Preliminaries

Examples of modeling by Graphs

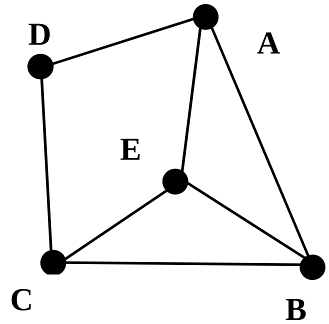


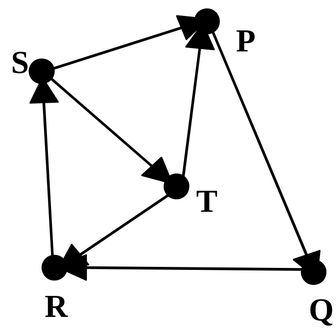


Graph Terminologies

- **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

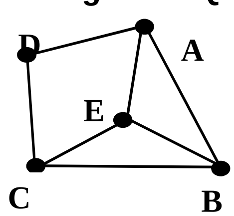
Graph Terminologies

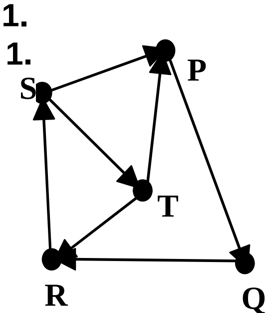
The degree *d*(*v*) of a vertex *v* is the number of edges incident to *v*.

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

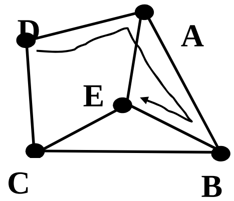
In directed graphs, indegree is the number of incoming edges at the vertex and outdegree is the number of outgoing edges from the vertex.

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





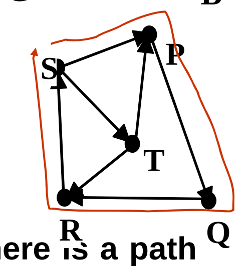
Paths and Cycles



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

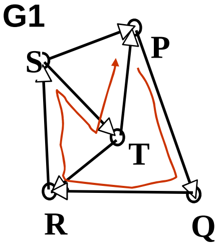
A simple circuit is a cycle if except for the first (and last) vertex, no other vertex appears more than once.

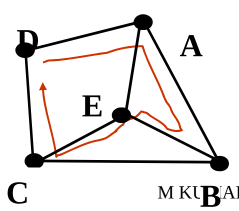
ABEA, DABECD, SPQRS, and STRS are cycles.

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.

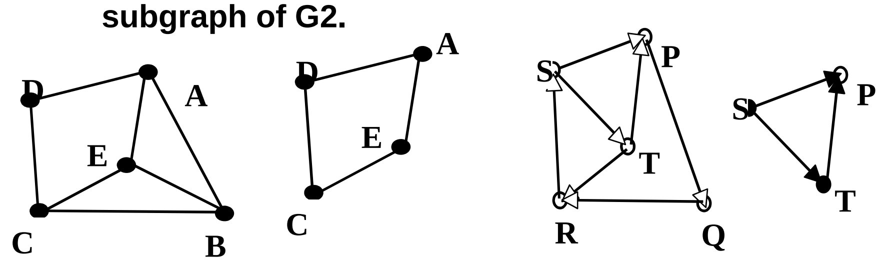




A subgraph of a graph G = (V,E) is a graph H(U,F) such that U ⊆ V and F⊆E.

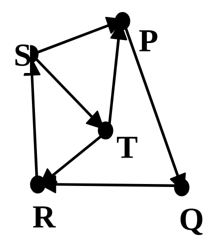
H1 {[U1:A,E,C,D], F1:[(A,E),(E,C),(C,D),(D,A)]} is a subgraph of G1

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

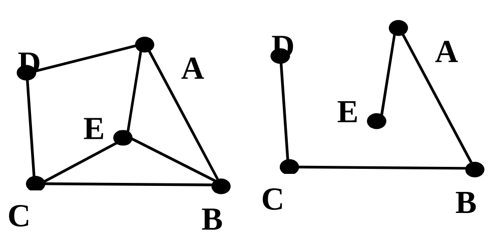


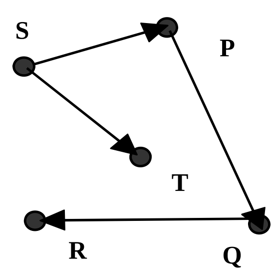
Spanning tree of G1

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

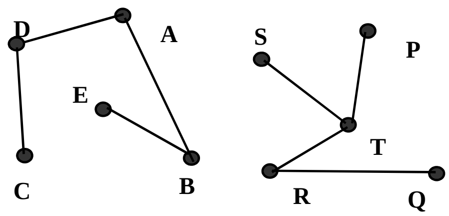
Connectivity

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.



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

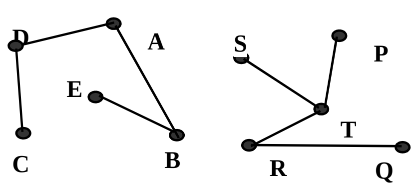
G(A,B,C,D,E) is a tree

Connectivity

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.



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

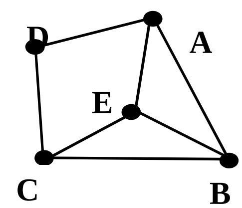
Graph Representations

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



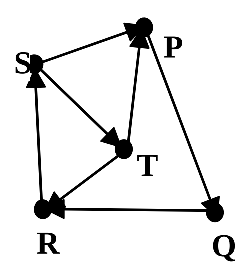
Graph Representations

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



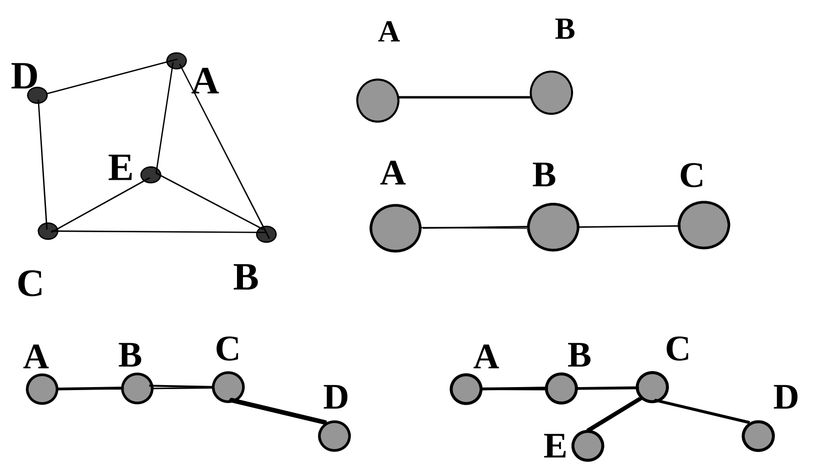
Depth-first search

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;

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

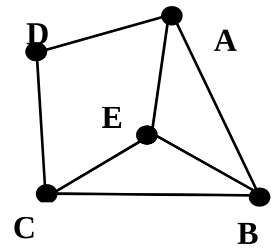
```
v ←old;  v∈ V
push (S,v);
while  S is nonempty do
    while  there exists a vertex w in L[x] and marked new  do
        T ← T ∪ (x,w) ;
        w ← old;
        push w onto S
    pop S
```


DFS

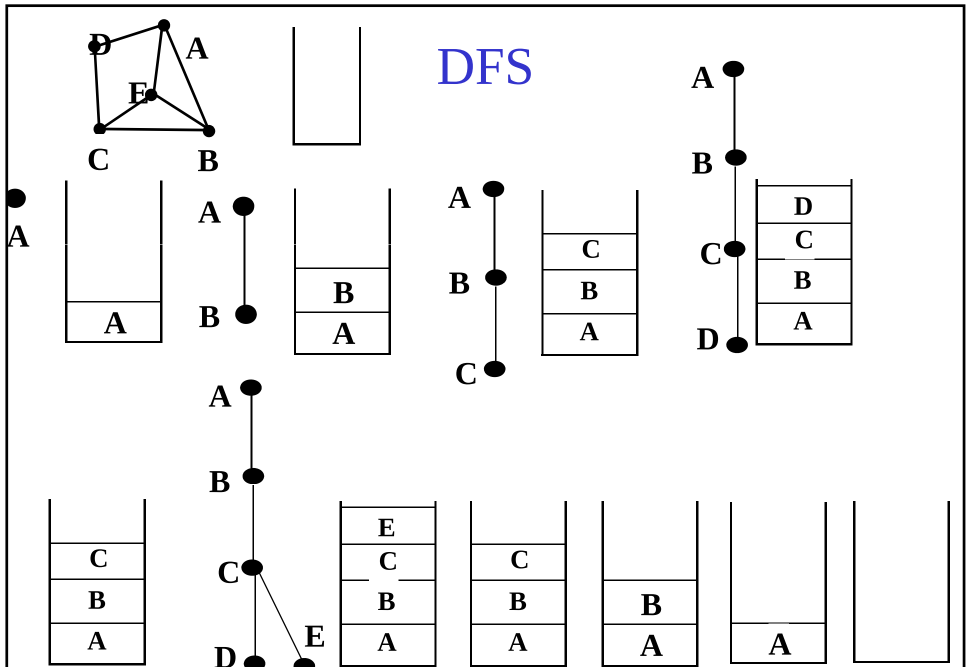


DFS

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}



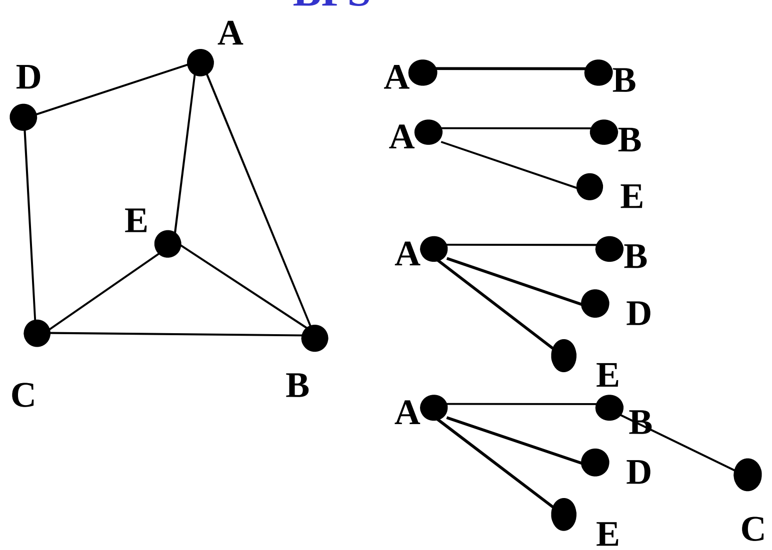
DFS



Breadth-first search

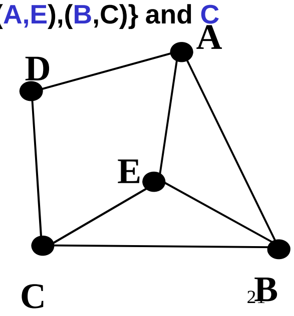
```
Procedure BFS_Tree G(V,E)
Input: G = (V,E); Q is a queue - initially empty;
        x ←Q : remove  the front item of  queue and
                                        denote it by x;
        initially mark all vertices  'new';
        L[x] refers to the adjacency list of x.
        T ← {0}
Output: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

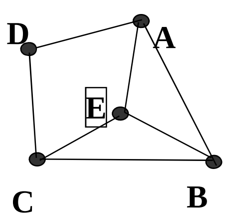

BFS

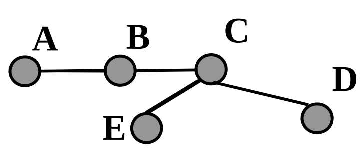


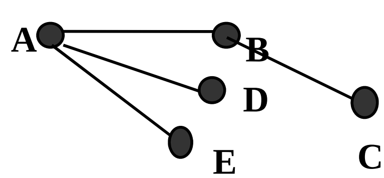
BFS

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};









DFS and BFS

- **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;**

- **1. $v \leftarrow$ old; $v \in V$**
- **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. $T \leftarrow T \cup (x,w)$;**
- **6. $w \leftarrow$ old;**
- **7. push w onto S**
- **8. pop S**

- **Procedure BFS_Tree G(V,E)**
- **Input: G = (V,E); Q is a queue - initially empty;**
- **$x \leftarrow Q$: 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.**
- **$T \leftarrow \{0\}$**
- **Output: The BFS tree T;**
- **1. $v \leftarrow$ old; $v \in V$**
- **2. insert (Q,v);**
- **3. while Q is nonempty do**
- **4. $x \leftarrow Q$**
- **5. for each vertex w in L[x] and marked 'new' do**
- **6. $T \leftarrow T \cup \{x,w\}$;**
- **7. $w \leftarrow$ old;**
- **8. insert (Q,w);**

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. $V' \leftarrow V$;
2. $c \leftarrow 0$;
3. while $V' \neq 0$ do
4. choose $u \in V'$;
5. $T \leftarrow$ all nodes reachable from u (by DFS_Tree)
6. $V' \leftarrow V' - T$;
7. $c \leftarrow c+1$;
8. $G_c \leftarrow T$;
9. $T \leftarrow 0$;

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.

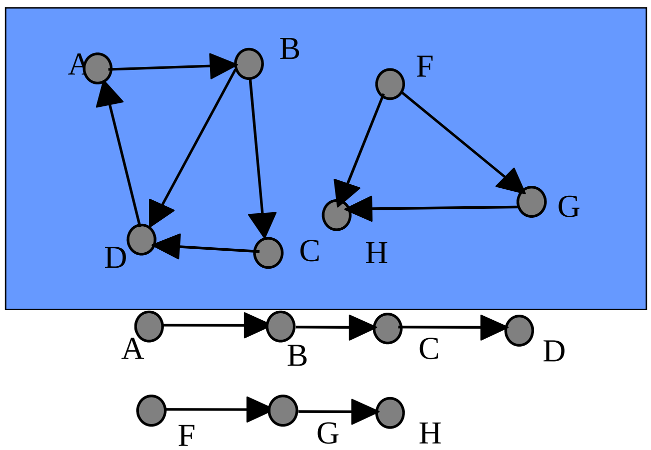
DFS Forest

```
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
```

- 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;

```
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 Forest



8/30/2009 M KUMAR CSE5311 27

- 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.

8/30/2009 M KUMAR CSE5311 28

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.

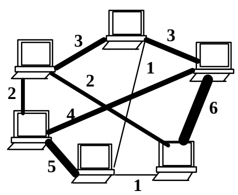
Suppose we want to broadcast a message to all the computers from an arbitrary computer.

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

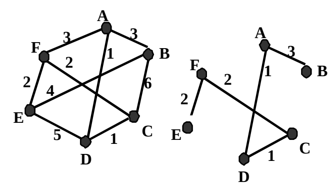
Minimum-Cost Spanning Trees

- 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

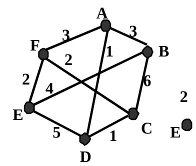
Kruskal's Algorithm

- 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

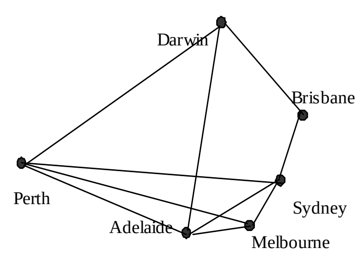
Complexity

- 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)] ≤ 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.

- 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:
- $v0 \rightarrow v1 \rightarrow v2 \; . \; . \; . \rightarrow vk\text{-}1 \rightarrow vk$
-

- 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

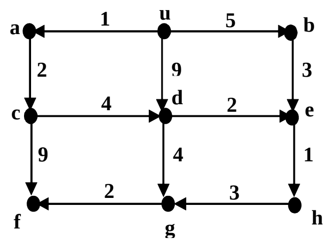
Given G (V,E), find the shortest path from a given vertex u ∈ V to every vertex v ∈ V (u ≠v).

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

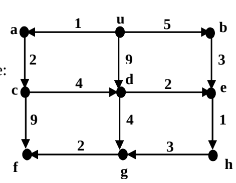
Initially, d[v] = 0 when u = v.
d[v] = ∞ if (u,v) is not an edge
d[v] = weight of edge (u,v) if (u,v) exists.

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

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



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 u 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]

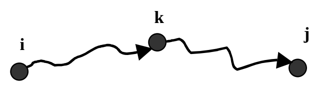
Analysis

- 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)

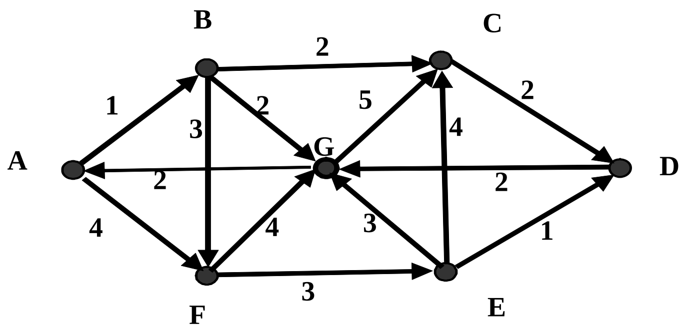


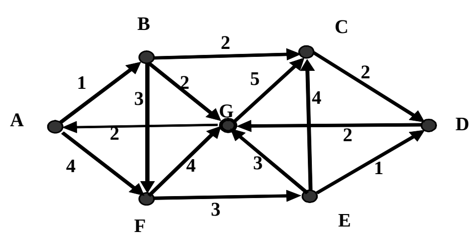
Recursive solution

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

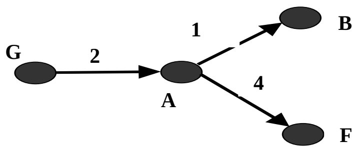
Floyd' s Algorithm for Shortest Paths

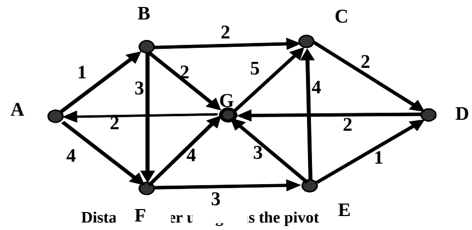
- Procedure **FLOYDs_G=[V,E]**
- **Input:** $n \times n$ matrix W representing the edge weights of an n-vertex directed graph. That is W =w(i,j) where, (Negative weights are allowed)
- **Output:** shortest path matrix, dist(i,j) is the shortest path between vertices i and j.
-
- **for** $v \leftarrow 1$ to n **do**
- **for** $w \leftarrow 1$ to n **do**
- dist[v,w] $\leftarrow$arc[v,w];
- **for** $u \leftarrow 1$ to n **do**
- **for** $v \leftarrow 1$ to n **do**
- **for** $w \leftarrow 1$ to n **do**
- **if** dist[v,u] + dist[u,w] < dist[v,w] **then**
- dist[v,w] $\leftarrow$ dist[v,u] + dist[u,w]
- Complexity : $\Theta(n^3)$





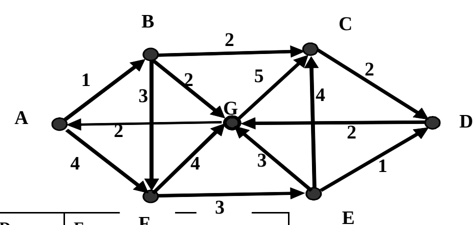
	A	B	C	D	E	F	G
A	**0**	**1**	∞	∞	∞	**4**	∞
B	∞	**0**	**2**	∞	∞	**3**	**2**
C	∞	∞	**0**	**2**	∞	∞	∞
D	∞	∞	∞	**0**	∞	∞	**2**
E	∞	∞	**4**	**1**	**0**	∞	**3**
F	∞	∞	∞	∞	**3**	**0**	**4**
G	**2**	∞	**5**	∞	∞	∞	**0**





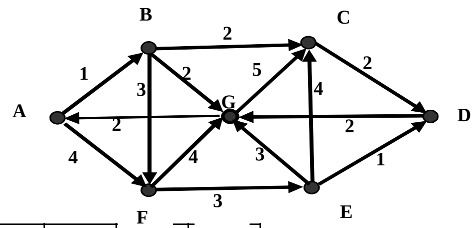
Dista er u s the pivot

	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	[illegible]	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

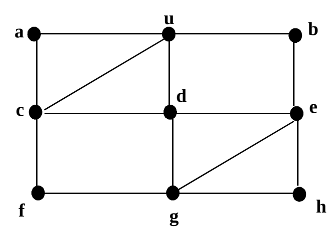
Hamiltonian Cycle

- Visit each vertex (except first and last exactl once)
- Can we use a variation of DFS to find the Hamiltonian Cycle?
- Finding the Hamiltonian cycle in a given graph is an NP-Complete problem.
- We will study NP-Complete problems and the heuristic solutions for the Hamiltonian Cycle problem later in the course

8/30/2009 M KUMAR CSE5311 55

Euler Path

- Traverse through every edge exactly once
- In the traversal, you arrive at a vertex using one edge and leave the same vertex using another edge.
  - **Therefore every vertex should be of even degree**
  - **This is a condition for the existence of the Euler path in a given graph**

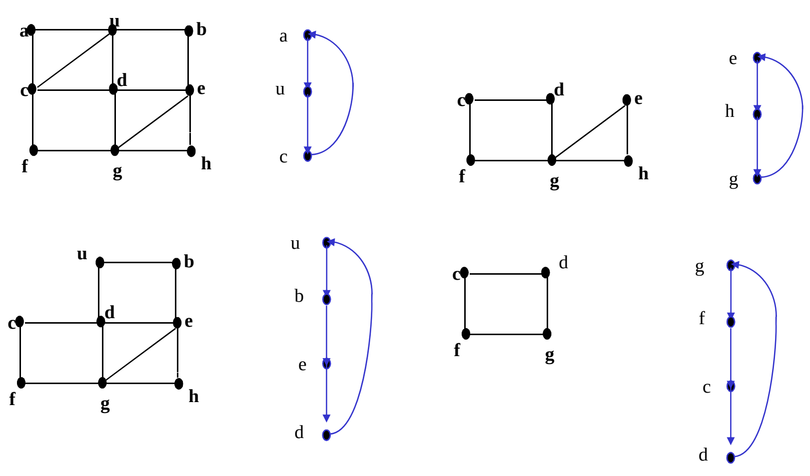


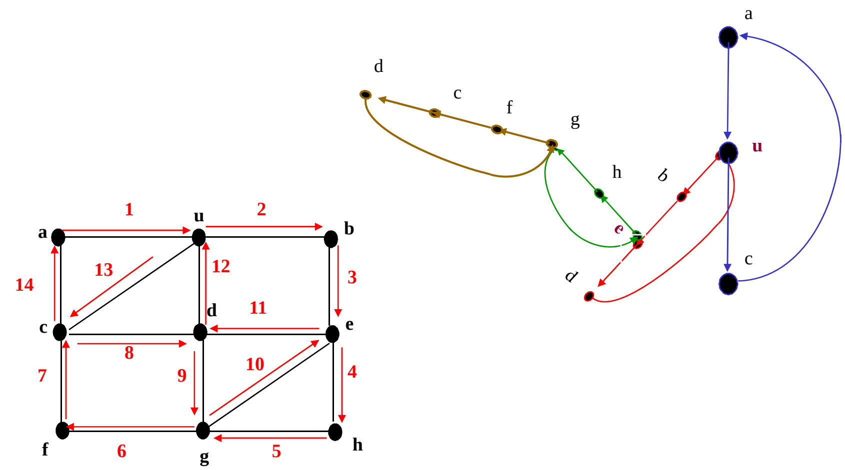
8/30/2009 M KUMAR CSE5311 56

Euler cycle

- Note: a disconnected graph *G* does not have an Euler path.
- Every node should have even degree.

1. Start a DFS-like search on *G* at any arbitrary node ‘*x*’
2. Continue DFS until a cycle is found
   a. That is the path returns to ‘*x*’ – we have a cycle -- *x*-*x*1-*x*2 …*x*
   b. Remove the edges of the cycle from *G*, to obtain *G* = *G*- *cycle.*
   c. All nodes with connectivity ‘=2’ will be removed as well
   d. If *G* is null, then the cycle above is the Euler cycle.
   e. Else, nodes with connectivity equal to 4 are greater will remain.
   f. Pick the first node ‘y’ of the above cycle, that is retained in the new *G*.
3. Set *x* equal to *y* and repeat the above step
4. Repeat above steps until G is null
5. The Euler cycle is *x-x1-x2-x3-* …*x*. If any xi was further explored, replace such *xi* with its corresponding cycle.





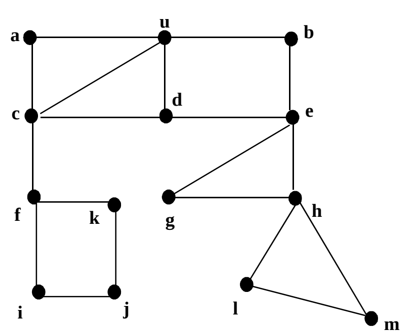
Biconnected Components

- Bionnected Components
- Bridges
- Articulation Points

aubedc, *ehg*, *hlm*, *fijk* – biconnected components

cf is a bridge

c, *f*, *e*, and *h* are articulation points

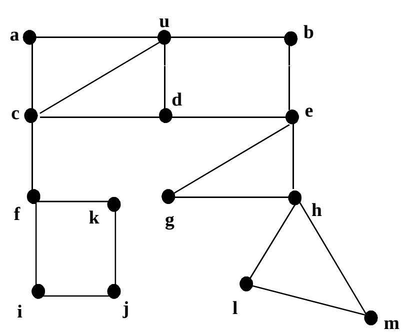


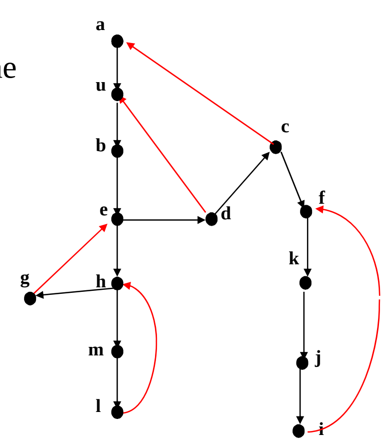
Find the biconnected components in the graph?

Definitions

- Vertex '*a*' is an articulation point in *G*, if the removal of '*a*' splits *G* into two or more parts.
  - Consider any two vertices, *u* and v of *G* – if every path between *u* and *v* in *G* contains '*a*' then, *a* is an *articulation point*.
  - G is *biconnected* if there is exists at least one path between *u* and *v* not containing *a*.
  - A graph with no articulation points is biconnected.
- Biconnected components of a graph are separated by articulation points.
- A *bridge* is a link connecting two articulation points.

The DFS of the given Graph

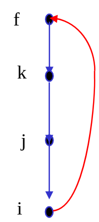




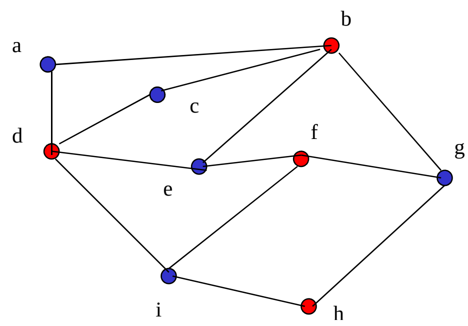
Each edge e ∈ E is either an edge in the depth-first spanning forest or it connects an ancestor to a descendent in some tree of the depth-first spanning forest. The latter are called back edges.

Algorithms

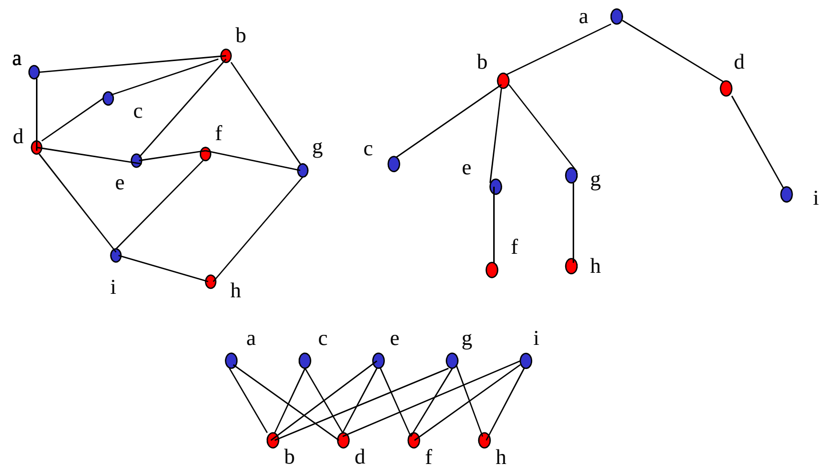
- A vertex $a \in V$ of G is an articulation point if any one of two conditions below are satisfied
  - *a* is the root of the depth-first spanning forest of *G* and it has multiple child vertices.
  - If *a* is not the root –
  - Let *C (x)* represent a child of *x* and
  - *A (x)* represent an ancestor of x.
  - *D (x)* represent a descendent of x.
  - then for some child *C(a)*, there exists no back edge between a *D(C(a))* and a proper *A(a)*. Proper – does not include itself.
  -



Bipartite Matching



Bipartite Matching



Bipartite matching

- Find the BFS tree of the given graph *G*
- Mark the root of the tree ‘blue’
- Mark all nodes at the next level, the children of the root node ‘red’.
- Mark the next level of nodes, blue and repeat the two-coloring for alternating levels
- If the adjacency list of any node has other color nodes at the same level, then the graph is not bipartite
- The adjacency list of each node must have nodes of a different color