UNIT 2:
Decrease and Conquer

Depth First Search

Graph

a graph G = (V, E) is defined by a pair of two sets:

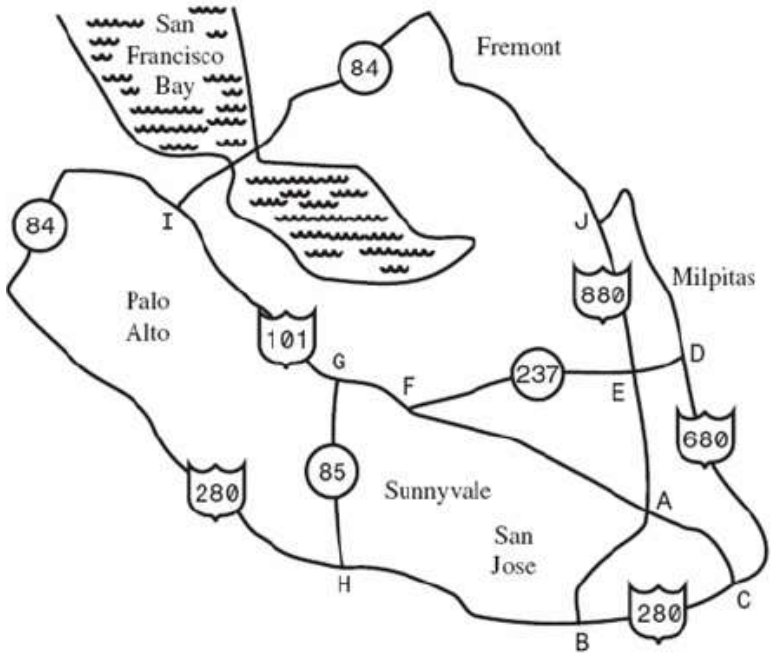
a finite set V of items called vertices and

a set E of pairs of these items called edges

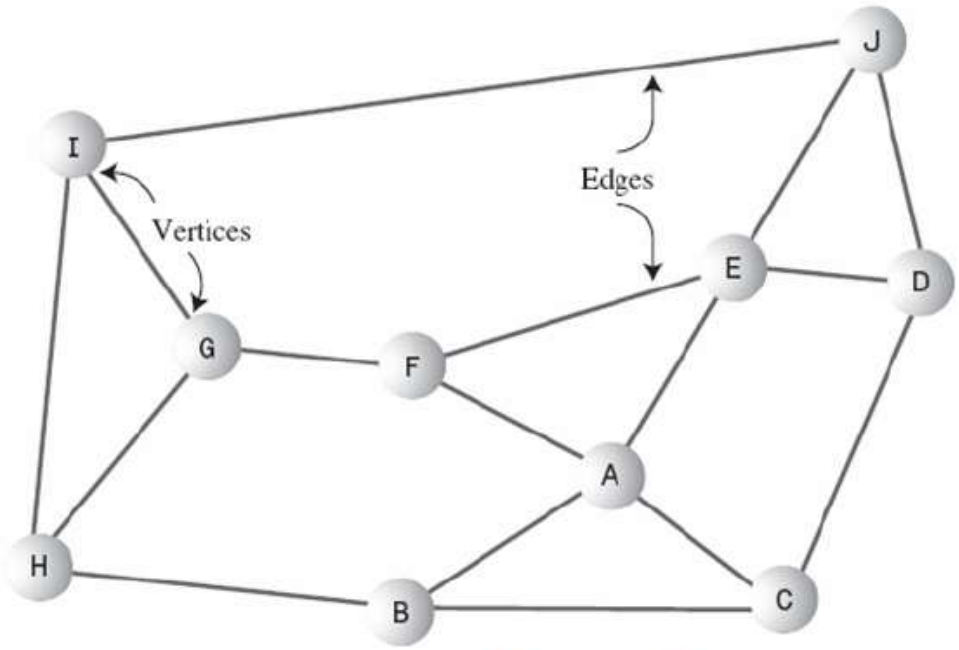
Graph

- Edges can be directed (digraphs) or undirected
- Edges can be labelled (weighted graphs)
- Graph may have loop (edges connecting vertices to themselves), cycles.
- Graph can be
  - dense or sparse
  - Complete
  - Connected or disconnected

Graph in real-world problem

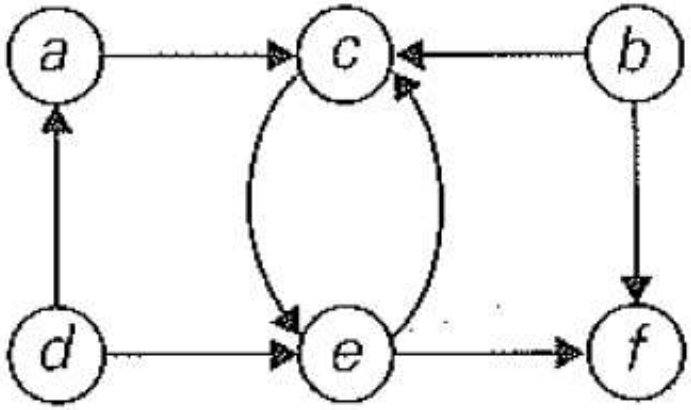


Roadmap



Corresponding graph

Graph representation



Digraph

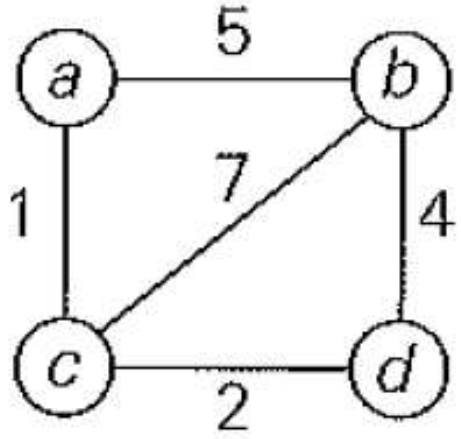
	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

Adjacency matrix

a	→	c	→	d		
b	→	c	→	f		
c	→	a	→	b	→	e
d	→	a	→	e		
e	→	c	→	d	→	f
f	→	b	→	e		

Adjacency lists

Graph representation



Weighted graph

	a	b	c	d
a	∞	5	1	∞
b	5	∞	7	4
c	1	7	∞	2
d	∞	4	2	∞

Weight matrix

a	→ b, 5 → c, 1
b	→ a, 5 → c, 7 → d, 4
c	→ a, 1 → b, 7 → d, 2
d	→ b, 4 → c, 2

Adjacency lists

Let's check our understanding…

Let A be the adjacency matrix of an undirected graph. What **property of the matrix** indicates that

1. the graph is complete.
2. the graph has a loop, i.e., an edge connecting a vertex to itself.
3. the graph has an isolated vertex, i.e., a vertex with no edges incident to it.

- (A **complete graph** has all of its vertices connected to every other vertex which may or may not include self loops to that vertex.)

  The matrix is filled with ones in all the places (which may not include the primary diagonal).

- (A graph with a **loop** has a connection from a vertex to itself.)

  Any one of the element in the primary diagonal is non zero, i.e. '1'.

- A graph has an isolated vertex when the vertex has no connection to any of the other vertices in the graph.

  One row and column is filled with zeros.

Depth First Search (DFS)

- Graph traversal algorithm
- investigated by French mathematician Charles Pierre Trémauxas a strategy for solving mazes
- uses decrease and conquer (decrease-by-one) strategy

Working of DFS

- DFS starts at the root node (selecting some arbitrary node as the root node) by marking it as having been visited.
- On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in.
- This process continues until a dead end-a vertex with no adjacent unvisited vertices-is encountered. At a dead end, the algorithm backtracks.
- The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.

Note: tie can be resolved arbitrarily, may also depends on data structure representing the graph

DFS uses Stack to trace the operation!

- push a vertex onto the stack when the vertex is reached for the first time (i.e., the visit of the vertex starts)
- pop a vertex off the stack when it becomes a dead end (i.e., the visit of the vertex ends)

Thus DFS yields two orderings of vertices:

1. Push order
2. Pop order

These orders are qualitatively different, and various applications can take advantage of either of them.

DFS forest

- DFS traversal's starting vertex serves as the root of the first tree in DFS forest.

- **Tree edge**: Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex (using tree edge) from which it is being reached.

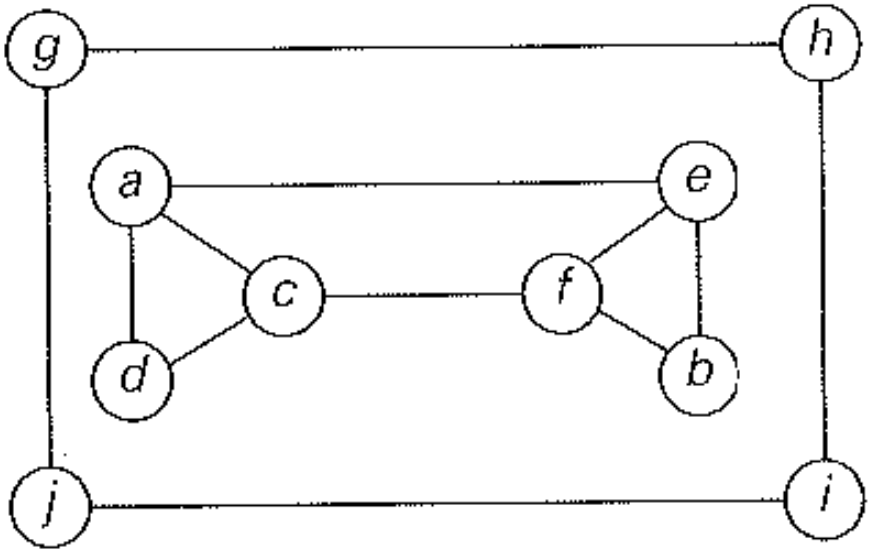
- **Back edge**: Edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). It connects a vertex to its ancestor, other than the parent.

Depth First Search

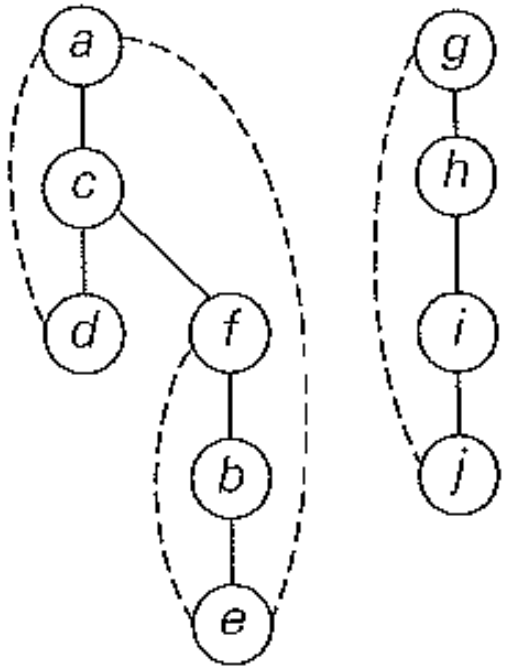
ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count*
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        dfs(w)



(a) Graph.



(c) DFS forest

	$e_{6,2}$	
	$b_{5,3}$	$j_{10,7}$
$d_{3,1}$	$f_{4,4}$	$i_{9,8}$
$c_{2,5}$		$h_{8,9}$
$a_{1,6}$		$g_{7,10}$

(b) Traversal's stack

DFS: Visualization

https://visualgo.net/en/dfsbfs

https://www.cs.usfca.edu/~galles/visualization/DFS.html

DFS algorithm analysis

Running time is proportional to the size of the data structure used for representing the graph.

Efficiency for adjacent matrix	$\Theta(\|V^2\|)$
Efficiency for adjacent lists	$\Theta(\|V\| + \|E\|)$

DFS: Applications

- checking connectivity, finding connected components
- checking acyclicity (if no back edges)
- finding articulation points and biconnected components
- searching the state-space of problems for solutions (in AI)

Algorithms that use depth-first search as a building block:

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding 3-(edge or vertex)-connected components.
- Finding the bridges of a graph.
- Generating words in order to plot the limit set of a group.
- Finding strongly connected components.
- Planarity testing.
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- Maze generation may use a randomized depth-first search.
- Finding biconnectivity in graphs.

Next session…

Decrease and Conquer…

- Breadth First Search