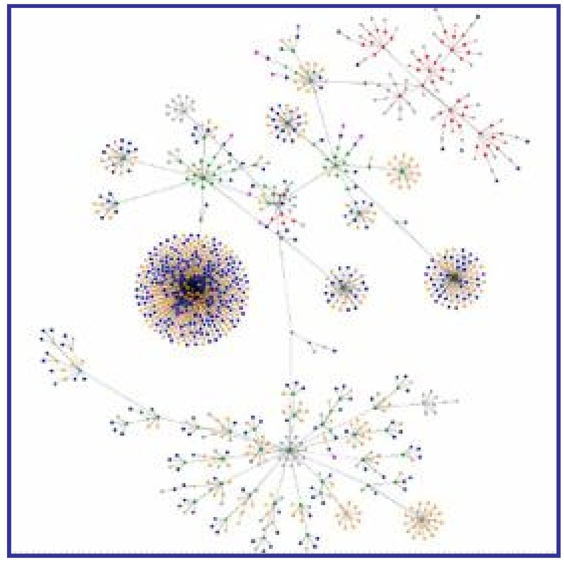
Graphs

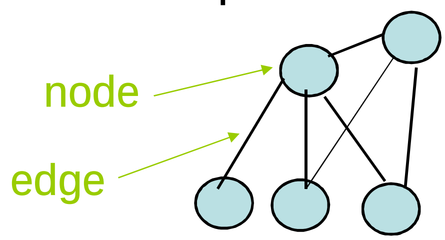


Today

- Graph Traversal
- Topological Sort

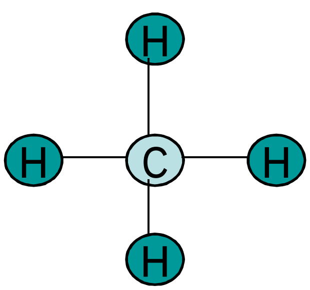
What is a graph?

- Graphs represent the relationships among data items
- A graph G consists of
  - a set V of nodes (vertices)
  - a set E of edges: each edge connects two nodes
- Each node represents an item
- Each edge represents the relationship between two items

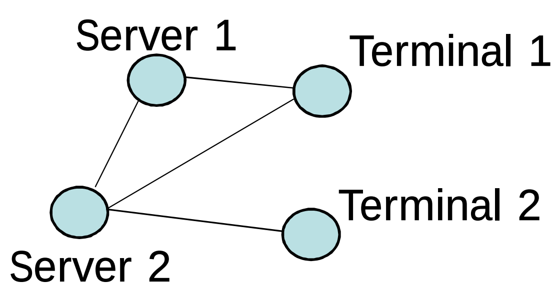


Examples of graphs

Molecular Structure



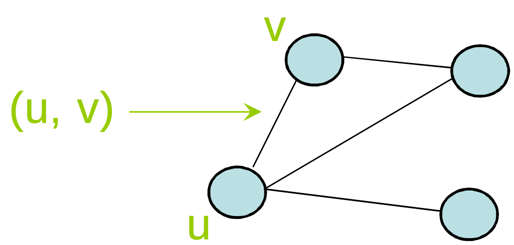
Computer Network



Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects

Formal Definition of graph

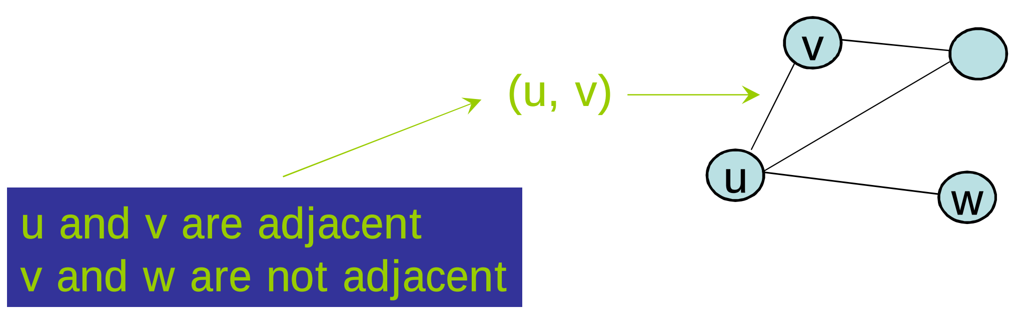
- The set of nodes is denoted as V
- For any nodes u and v, if u and v are connected by an edge, such edge is denoted as (u, v)



- The set of edges is denoted as E
- A graph G is defined as a pair (V, E)

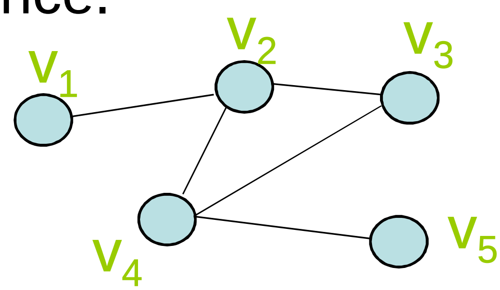
Adjacent

- Two nodes u and v are said to be adjacent if $(u, v) \in E$



Path and simple path

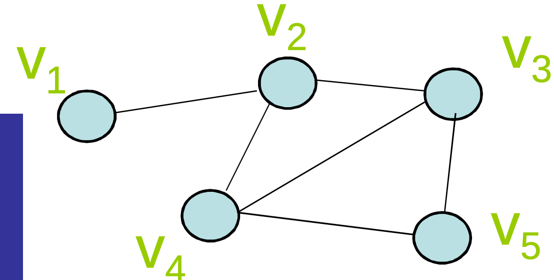
- A path from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A path is called a simple path if every node appears at most once.



- v_2, v_3, v_4, v_2, v_1 is a path
- v_2, v_3, v_4, v_5 is a path, also it is a simple path

Cycle and simple cycle

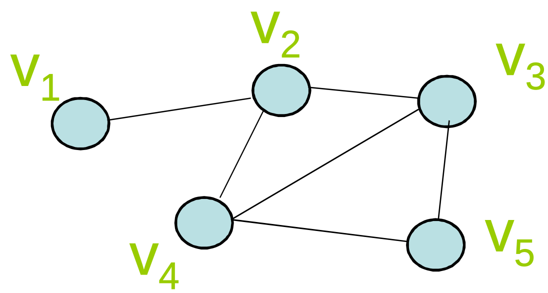
- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes



- $v_2, v_3, v_4, v_5, v_3, v_2$ is a cycle
- v_2, v_3, v_4, v_2 is a cycle, it is also a simple cycle

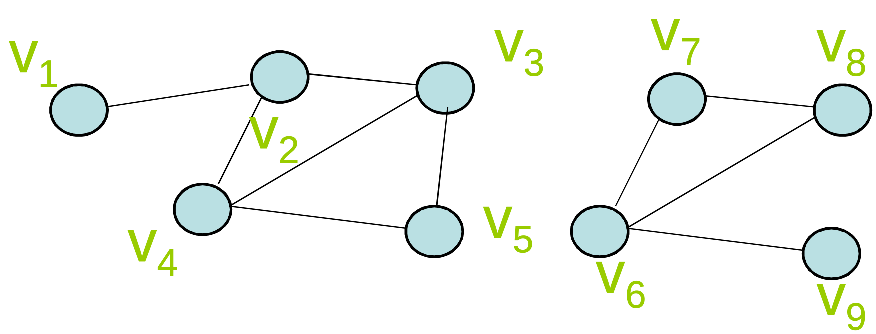
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; otherwise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

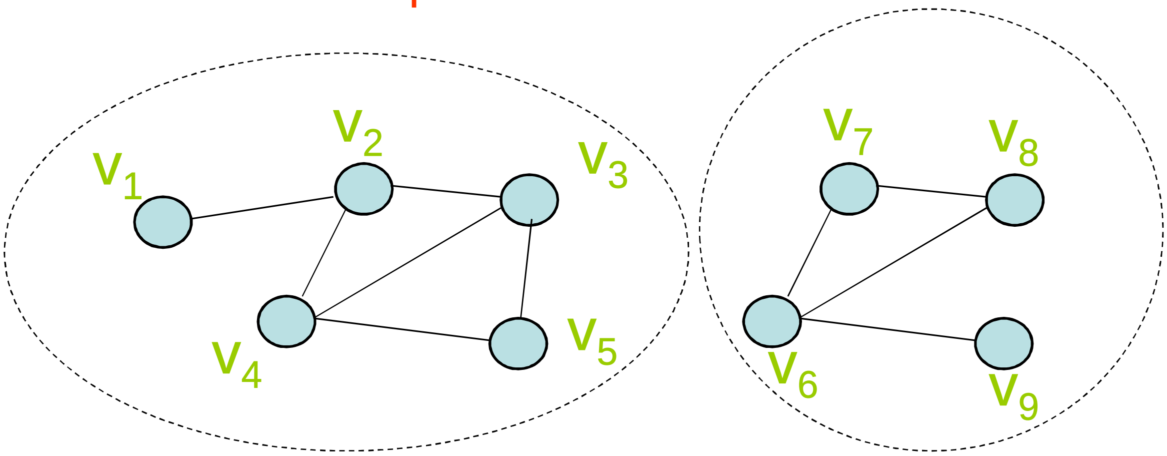
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

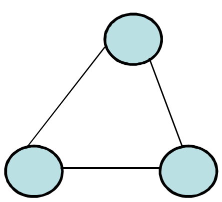
Connected component

- If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a connected component.

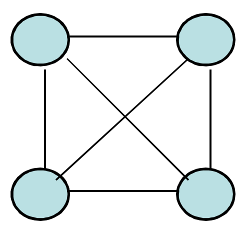


Complete graph

- A graph is complete if each pair of distinct nodes has an edge



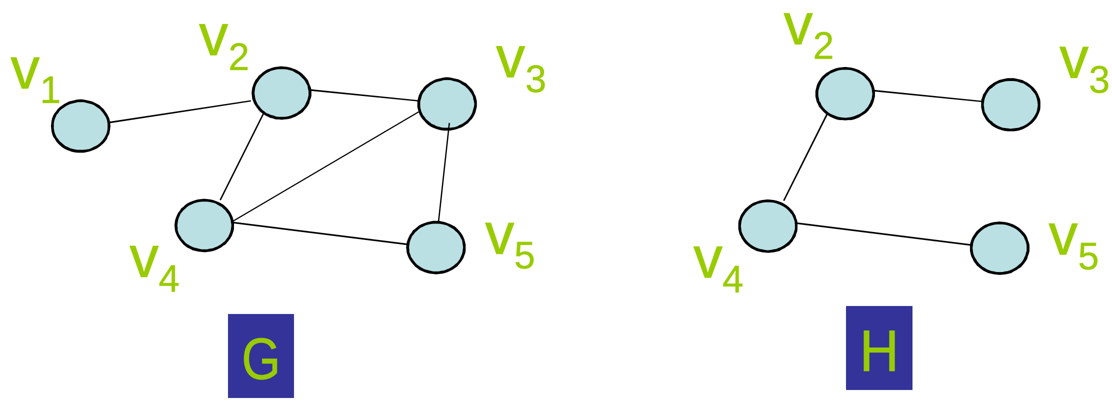
Complete graph with 3 nodes



Complete graph with 4 nodes

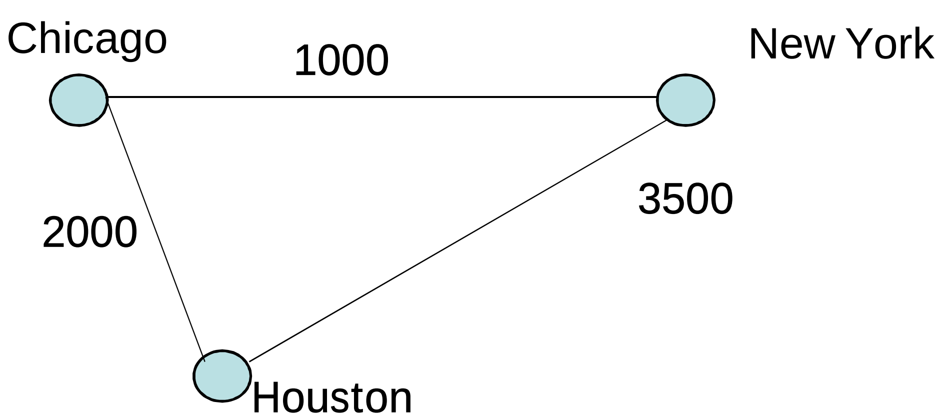
Subgraph

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



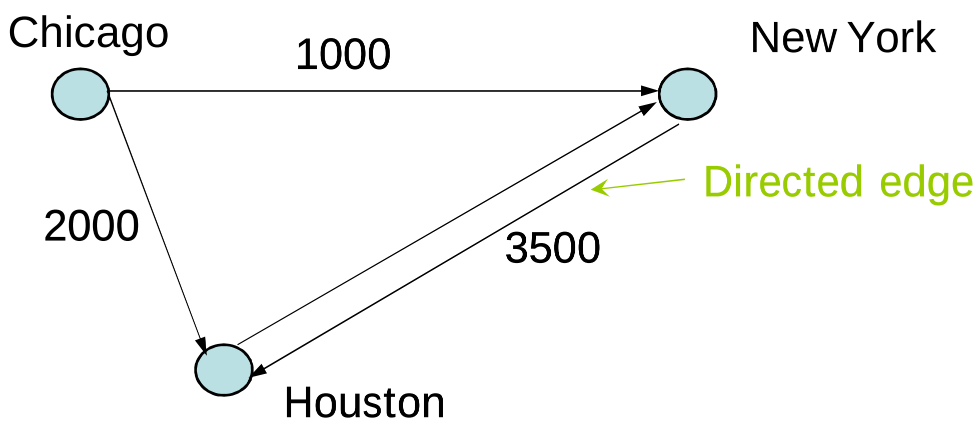
Weighted graph

- If each edge in G is assigned a weight, it is called a weighted graph

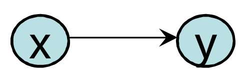


Directed graph (digraph)

- All previous graphs are undirected graph
- If each edge in E has a direction, it is called a directed edge
- A directed graph is a graph where every edges is a directed edge



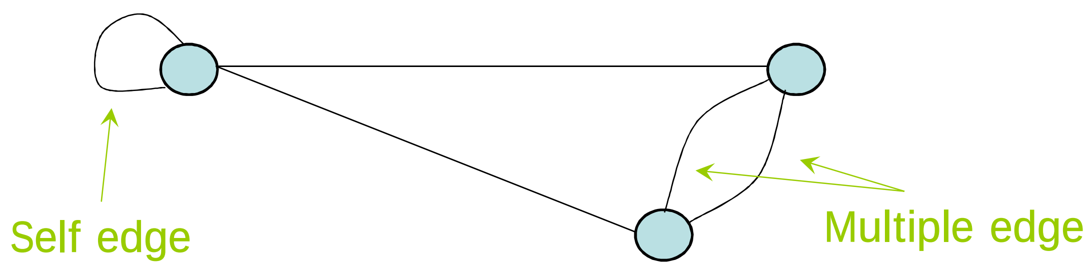
More on directed graph



- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y
- In a directed graph, directed path, directed cycle can be defined similarly

Multigraph

- A graph cannot have duplicate edges.
- Multigraph allows multiple edges and self edge (or loop).



Property of graph

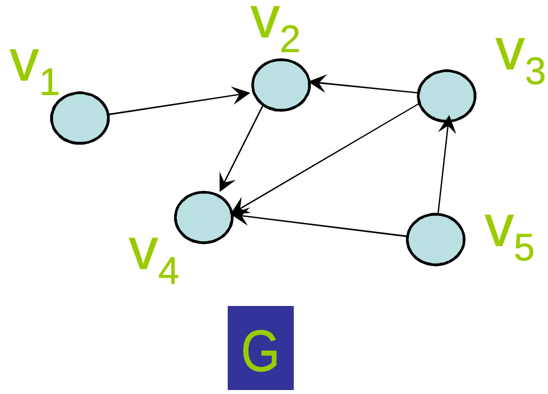
- A undirected graph that is connected and has no cycle is a tree.
- A tree with n nodes have exactly n-1 edges.
- A connected undirected graph with n nodes must have at least n-1 edges.

Implementing Graph

- Adjacency matrix
  - Represent a graph using a two-dimensional array
- Adjacency list
  - Represent a graph using n linked lists where n is the number of vertices

Adjacency matrix for directed graph

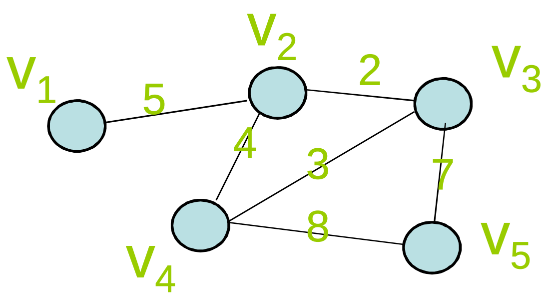
$$
\text{Matrix}[i][j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}
$$



		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

Adjacency matrix for weighted undirected graph

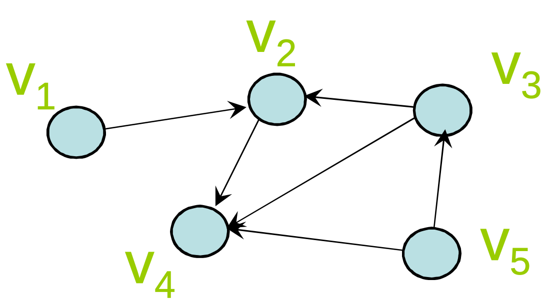
$$\text{Matrix}[i][j] = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \text{ or } (v_j, v_i) \in E \\ \infty & \text{otherwise} \end{cases}$$



G

		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	0	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

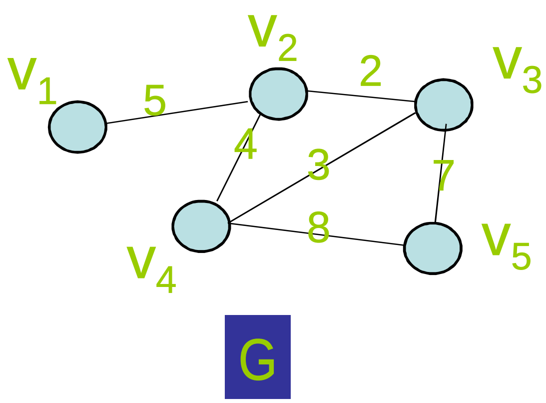
Adjacency list for directed graph



G

1	v_1	$\rightarrow v_2$
2	v_2	$\rightarrow v_4$
3	v_3	$\rightarrow v_2 \rightarrow v_4$
4	v_4	
5	v_5	$\rightarrow v_3 \rightarrow v_4$

Adjacency list for weighted undirected graph



1	v_1	$\rightarrow v_2(5)$
2	v_2	$\rightarrow v_1(5) \rightarrow v_3(2) \rightarrow v_4(4)$
3	v_3	$\rightarrow v_2(2) \rightarrow v_4(3) \rightarrow v_5(7)$
4	v_4	$\rightarrow v_2(4) \rightarrow v_3(3) \rightarrow v_5(8)$
5	v_5	$\rightarrow v_3(7) \rightarrow v_4(8)$

Pros and Cons

- Adjacency matrix
  - Allows us to determine whether there is an edge from node i to node j in O(1) time
- Adjacency list
  - Allows us to find all nodes adjacent to a given node j efficiently
  - If the graph is sparse, adjacency list requires less space

Problems related to Graph

- Graph Traversal
- Topological Sort
- Spanning Tree
- Minimum Spanning Tree
- Shortest Path

Graph Traversal Algorithm

- To traverse a tree, we use tree traversal algorithms like pre-order, in-order, and post-order to visit all the nodes in a tree
- Similarly, graph traversal algorithm tries to visit all the nodes it can reach.
- If a graph is disconnected, a graph traversal that begins at a node v will visit only a subset of nodes, that is, the connected component containing v.

Two basic traversal algorithms

- Two basic graph traversal algorithms:
  - Depth-first-search (DFS)
    - After visit node v, DFS strategy proceeds along a path from v as deeply into the graph as possible before backing up
  - Breadth-first-search (BFS)
    - After visit node v, BFS strategy visits every node adjacent to v before visiting any other nodes

Breadth-first search

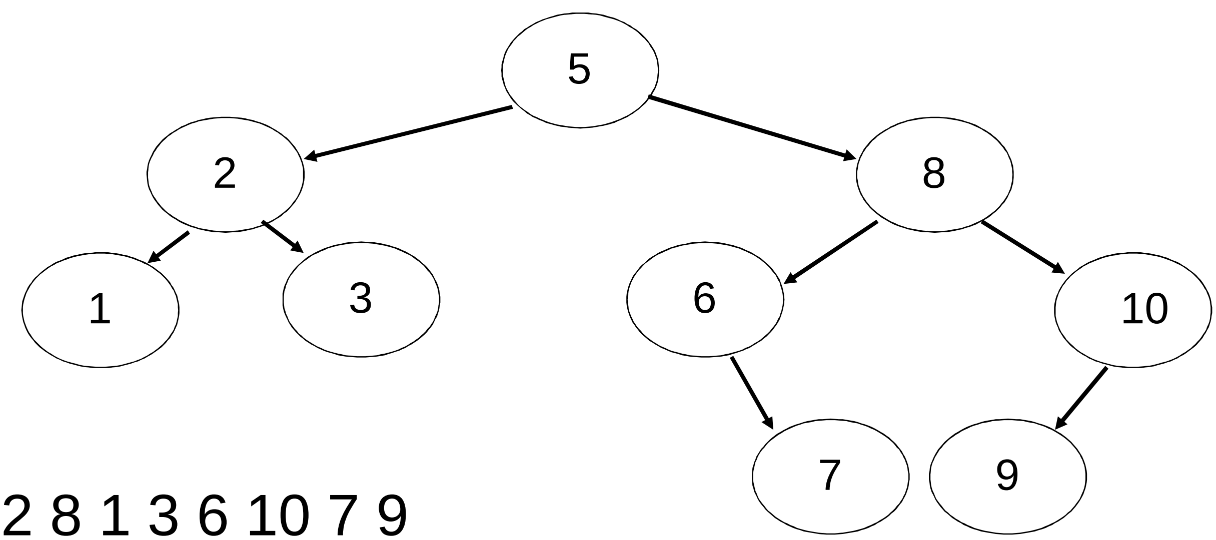
- One of the simplest algorithms
- Also one of the most important
  - It forms the basis for MANY graph algorithms

BFS: Level-by-level traversal

- Given a starting vertex s
- Visit all vertices at increasing distance from s
  - Visit all vertices at distance k from s
  - Then visit all vertices at distance k+1 from s
  - Then ….

BFS in a binary tree (reminder)

BFS: visit all siblings before their descendents



5 2 8 1 3 6 10 7 9

BFS(tree t)

```
NodePrt curr;
Queue q;
initialize(q);
Insert(q,t);
while (not Isempty(q))
  curr = delete(q)
  visit curr // e.g., print curr.datum
  insert(q, curr->left)
  insert(q, curr->right)
```

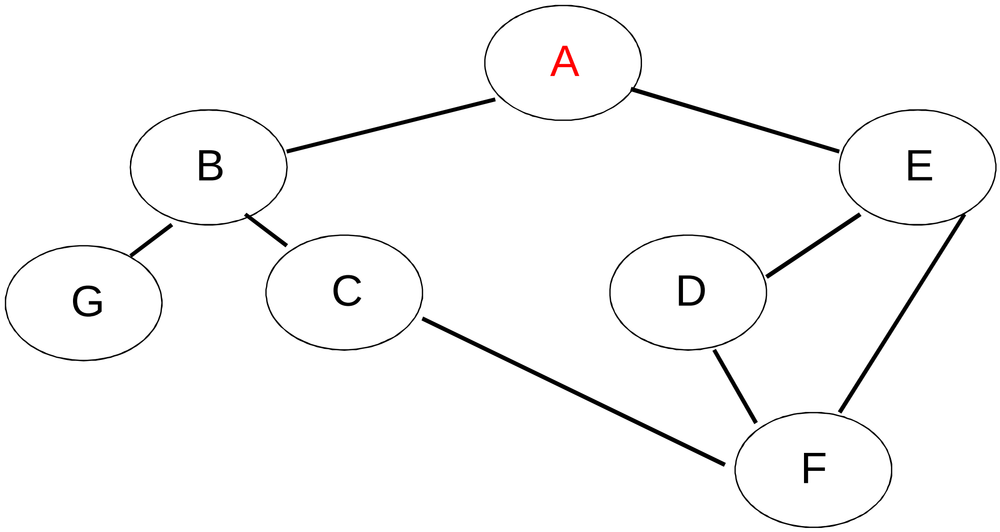
This version for binary trees only!

BFS for general graphs

- This version assumes vertices have two children
  - left, right
  - This is trivial to fix
- But still no good for general graphs
- It does not handle cycles

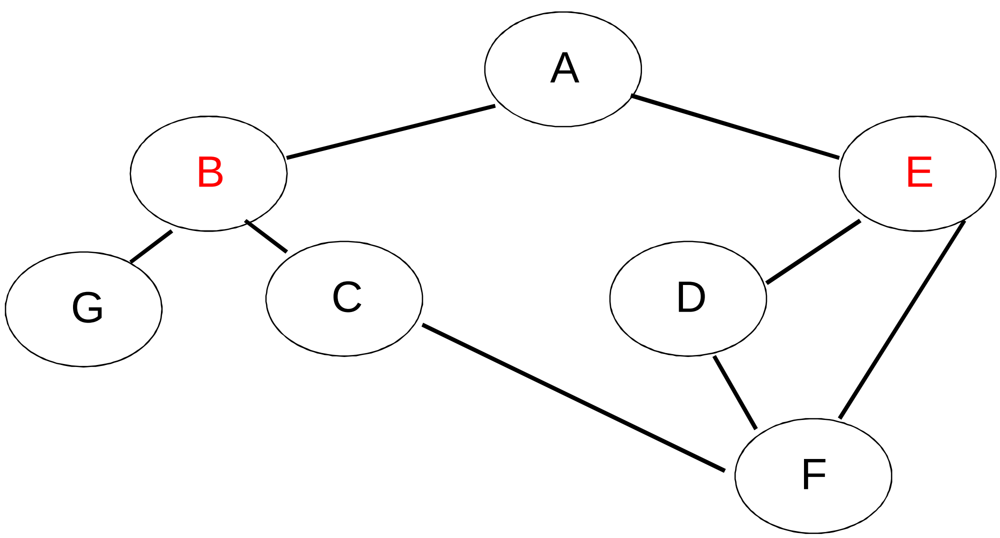
Example.

Queue: A



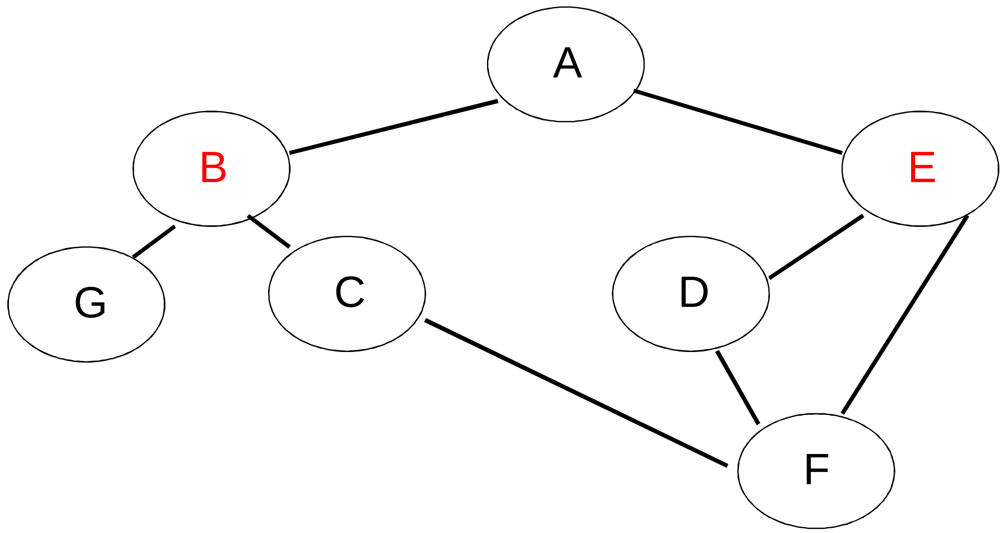
Start with A. Put in the queue (marked red)

Queue: A B E



B and E are next

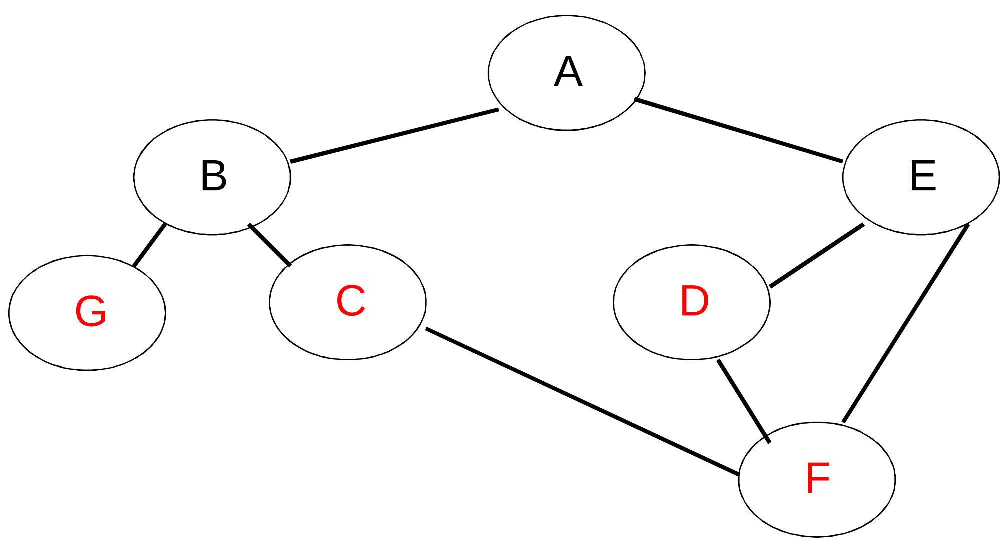
Queue: A B E C G D F



When we go to B, we put G and C in the queue

When we go to E, we put D and F in the queue

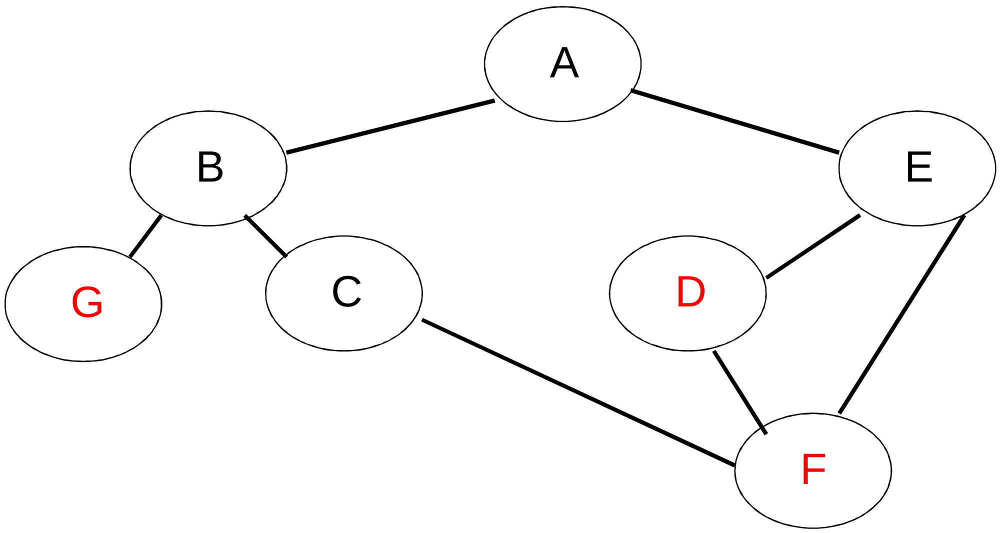
Queue: A B E C G D F



When we go to B, we put G and C in the queue

When we go to E, we put D and F in the queue

Queue: A B E C G D F F



Suppose we now want to expand C.

We put F in the queue again!

Generalizing BFS

- Cycles:
- We need to save auxiliary information
- Each node needs to be marked
  - Visited: No need to be put on queue
  - Not visited: Put on queue when found

What about assuming only two children vertices?

- Need to put all adjacent vertices in queue

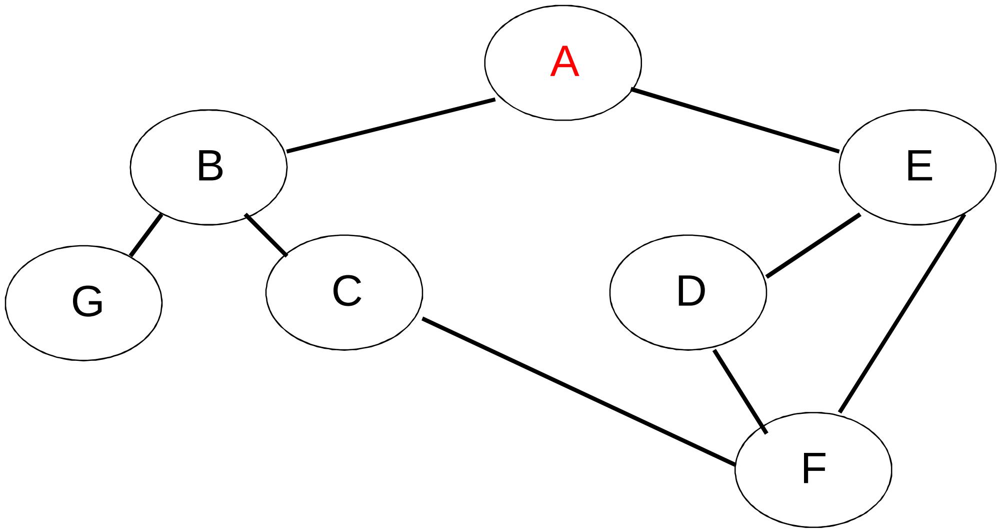
The general BFS algorithm

- Each vertex can be in one of three states:
  - Unmarked and not on queue
  - Marked and on queue
  - Marked and off queue
- The algorithm moves vertices between these states

Handling vertices

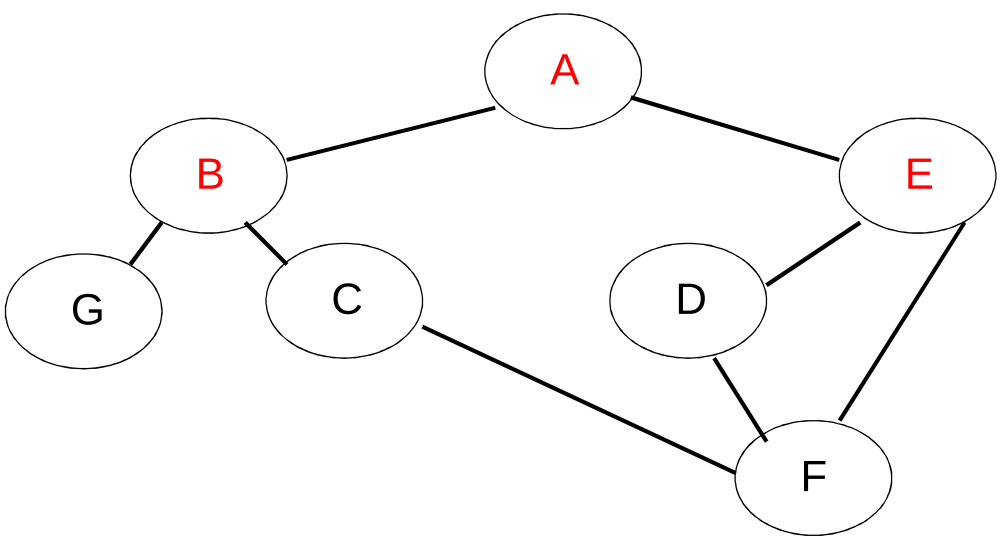
- Unmarked and not on queue:
  - Not reached yet
- Marked and on queue:
  - Known, but adjacent vertices not visited yet (possibly)
- Marked and off queue:
  - Known, all adjacent vertices on queue or done with

Queue: A



Start with A. Mark it.

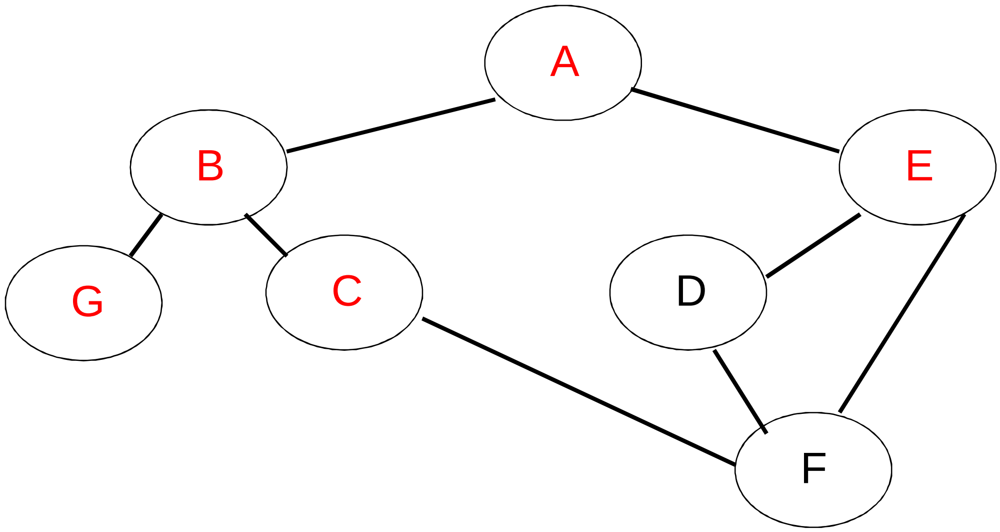
Queue: A B E



Expand A’s adjacent vertices.

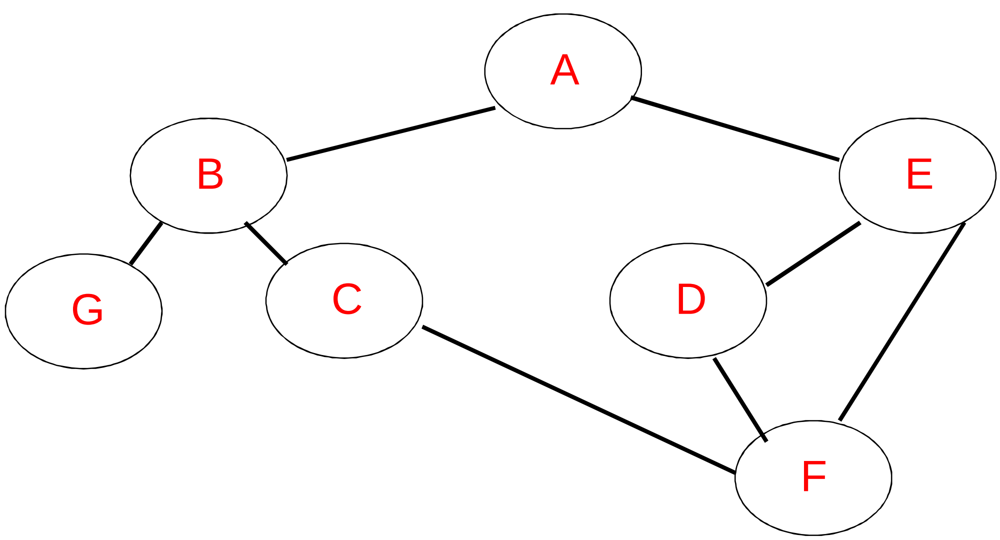
Mark them and put them in queue.

Queue: A B E C G



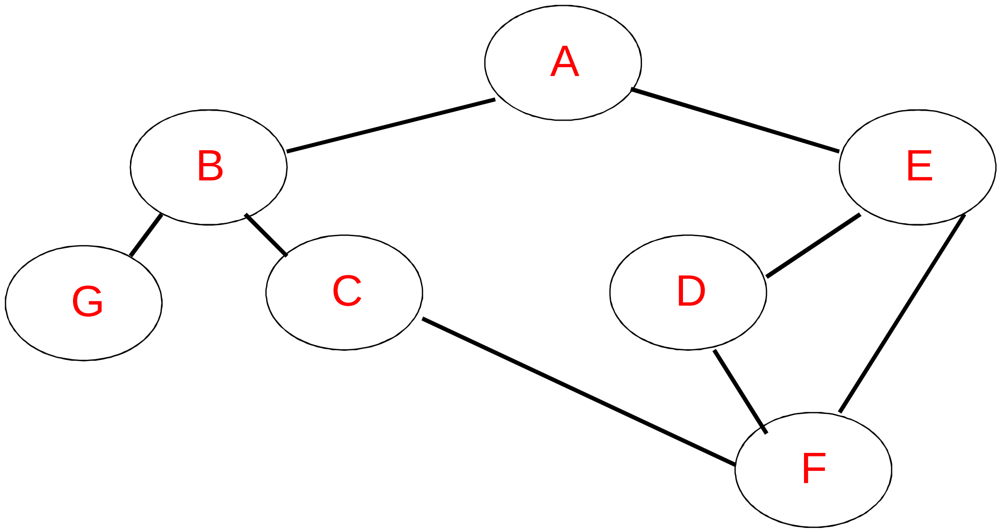
Now take B off queue, and queue its neighbors.

Queue: A B E C G D F



Do same with E.

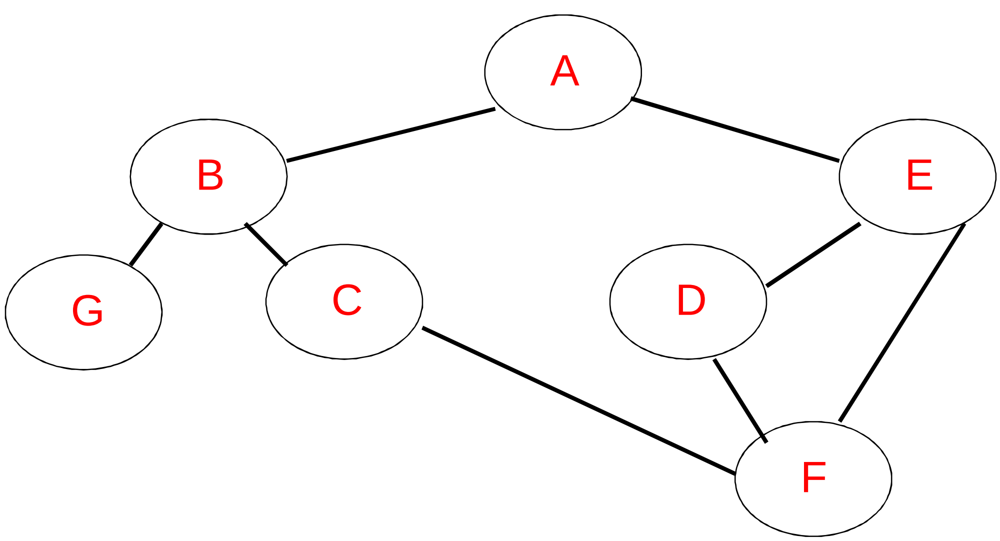
Queue: A B E C G D F



Visit C.

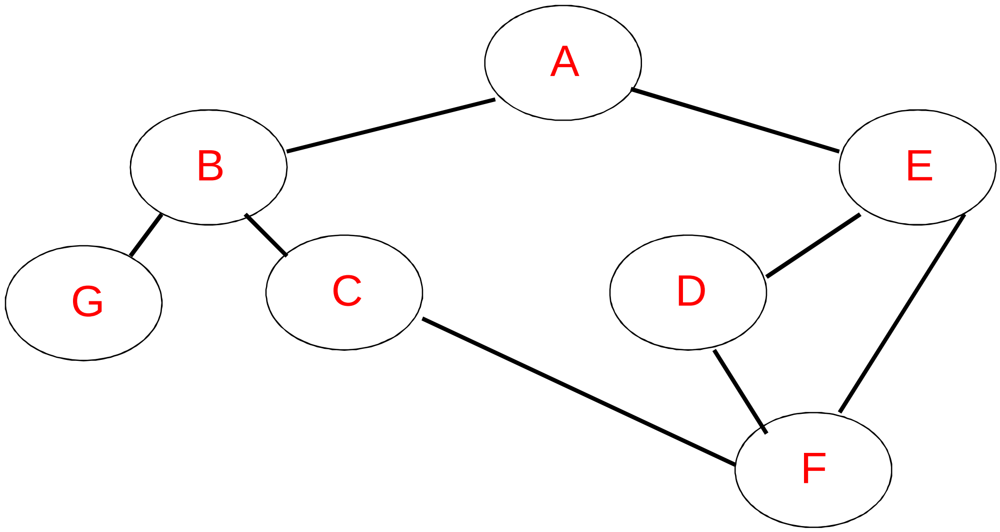
Its neighbor F is already marked, so not queued.

Queue: A B E C G D F



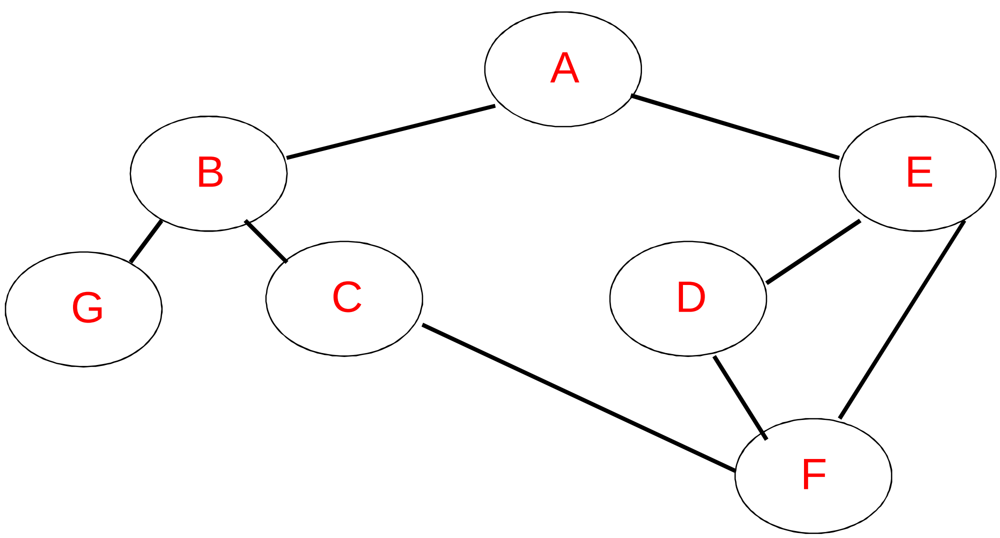
Visit G.

Queue: A B E C G D F



Visit D. F, E marked so not queued.

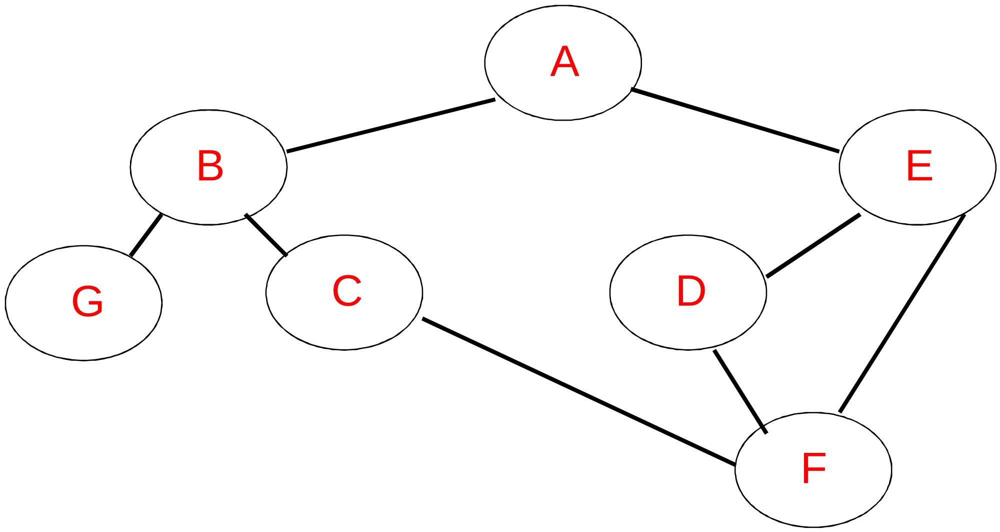
Queue: A B E C G D F



Visit F.

E, D, C marked, so not queued again.

Queue: A B E C G D F



Done. We have explored the graph in order:

A B E C G D F.

Breadth-first search (BFS)

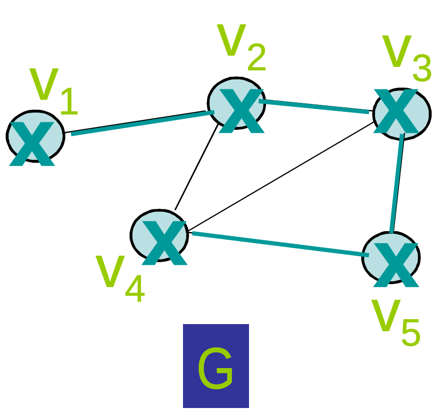
- BFS strategy looks similar to level-order. From a given node v, it first visits itself. Then, it visits every node adjacent to v before visiting any other nodes.
  - 1. Visit v
  - 2. Visit all v’s neigbours
  - 3. Visit all v’s neighbours’ neighbours
  - …
- Similar to level-order, BFS is based on a queue.

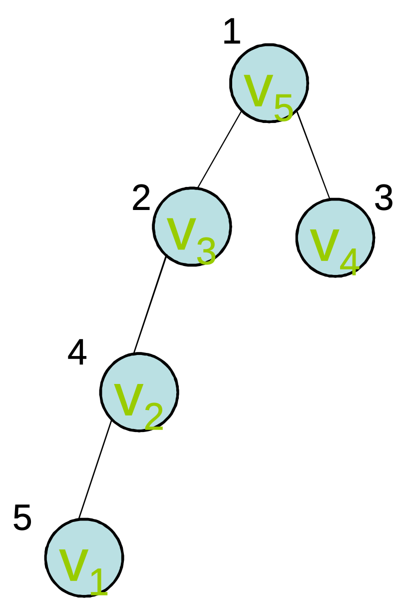
BFS(graph g, vertex s)

```
1. unmark all vertices in G;
2. Creat a queue q;
3. mark s;
4. insert(s,q)
5. while (!isempty(q))
6.   curr = delete(q);
7.   visit curr; // e.g., print its data
8.   for each edge <curr, V>
9.     if V is unmarked
10.        mark V;
11.        insert(V,q);
```

BFS example

- Start from v_5





Visit	Queue (front to back)
v_5	v_5
	empty
v_3	v_3
v_4	v_3, v_4
	v_4
v_2	v_4, v_2
	v_2
	empty
v_1	v_1
	empty

Interesting features of BFS

- Complexity: O(|V| + |E|)
  - All vertices put on queue exactly once
  - For each vertex on queue, we expand its edges
  - In other words, we traverse all edges once
- BFS finds shortest path from s to each vertex
  - Shortest in terms of number of edges
  - Why does this work?

Depth-first search

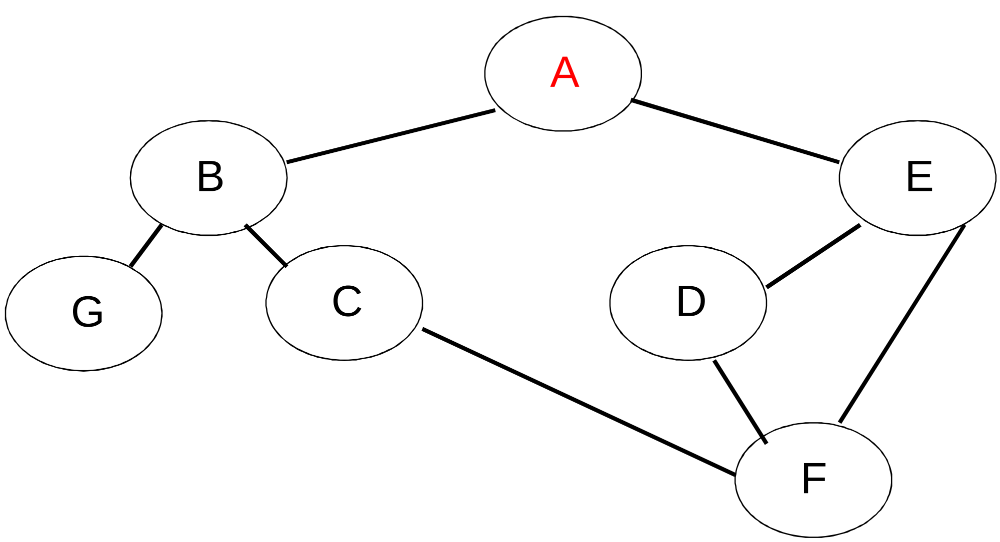
- Again, a simple and powerful algorithm
- Given a starting vertex s
- Pick an adjacent vertex, visit it.
  - Then visit one of its adjacent vertices
  - …..
  - Until impossible, then backtrack, visit another

DFS(graph g, vertex s)

Assume all vertices initially unmarked

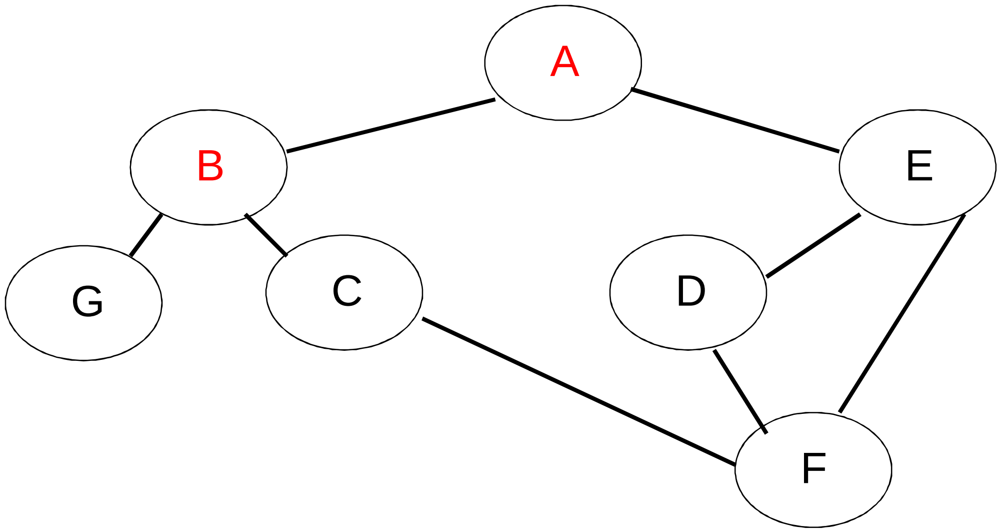
```
mark s;
visit s; // e.g., print its data
for each edge <s, V>
   if V is not marked
     DFS(G, V);
```

Current vertex: A



Start with A. Mark it.

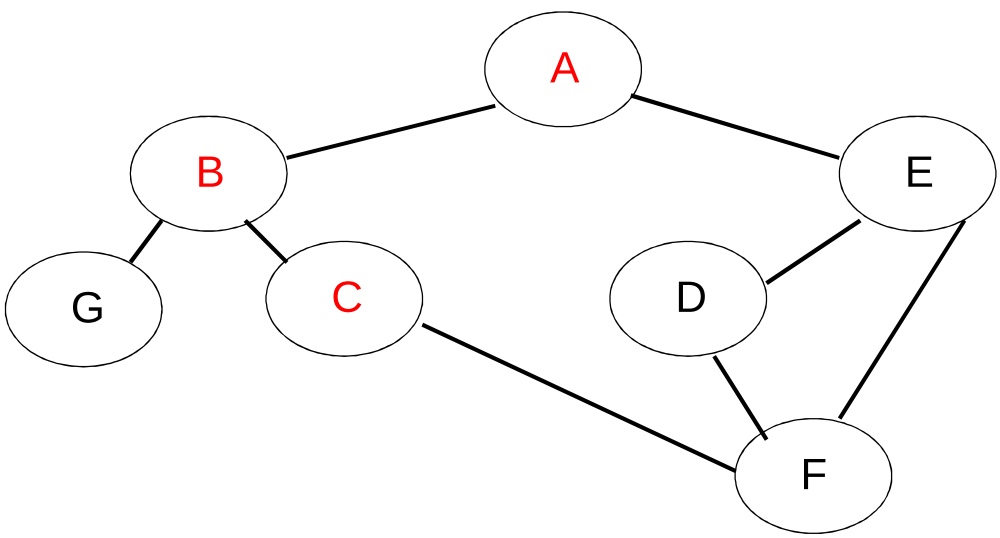
Current: B



Expand A’s adjacent vertices. Pick one (B).

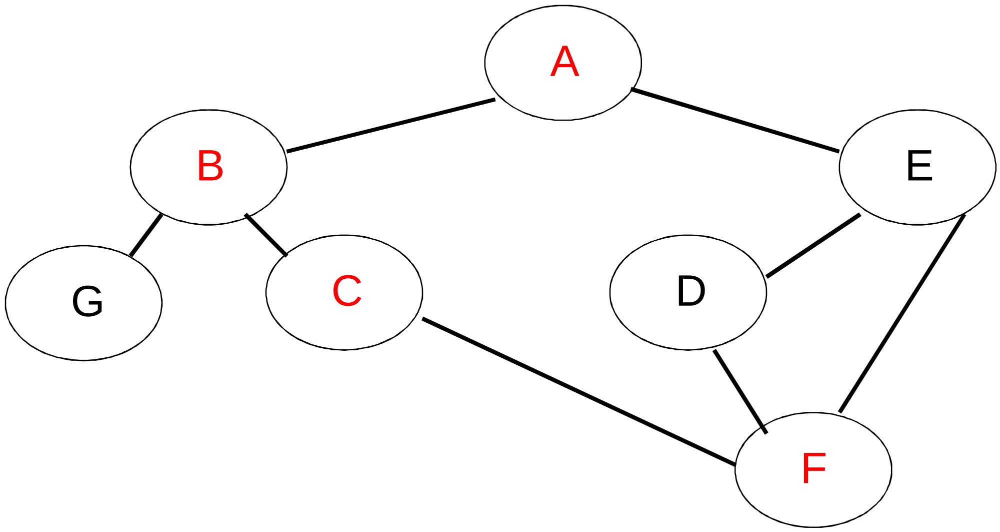
Mark it and re-visit.

Current: C



Now expand B, and visit its neighbor, C.

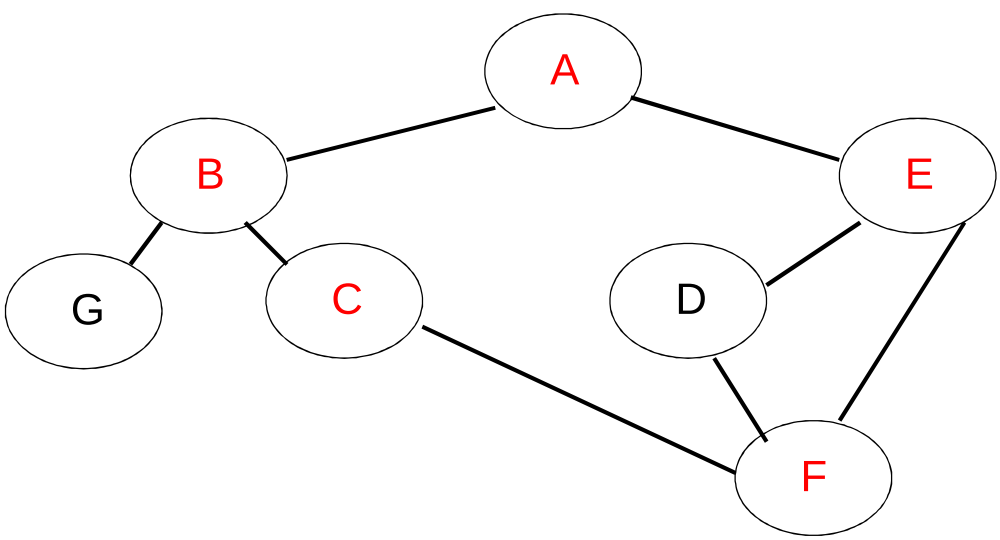
Current: F



Visit F.

Pick one of its neighbors, E.

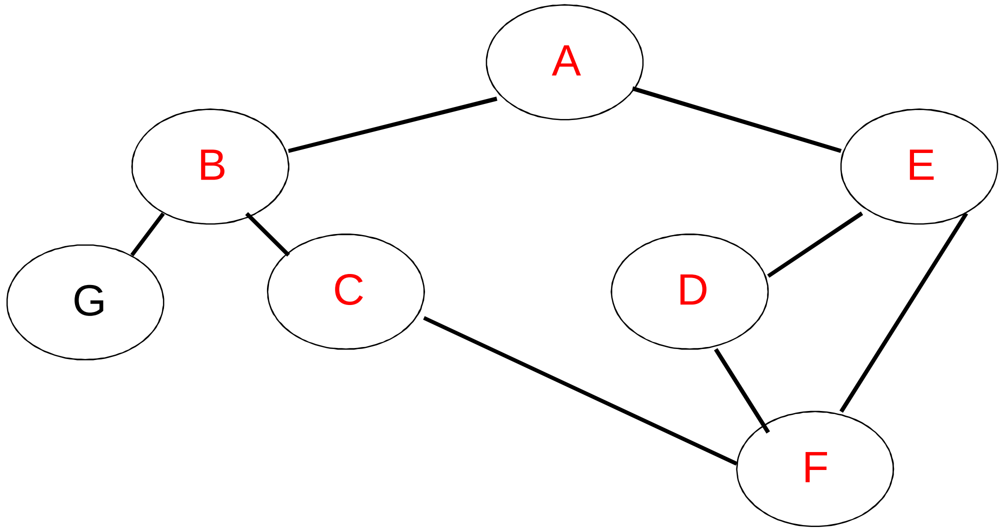
Current: E



E’s adjacent vertices are A, D and F.

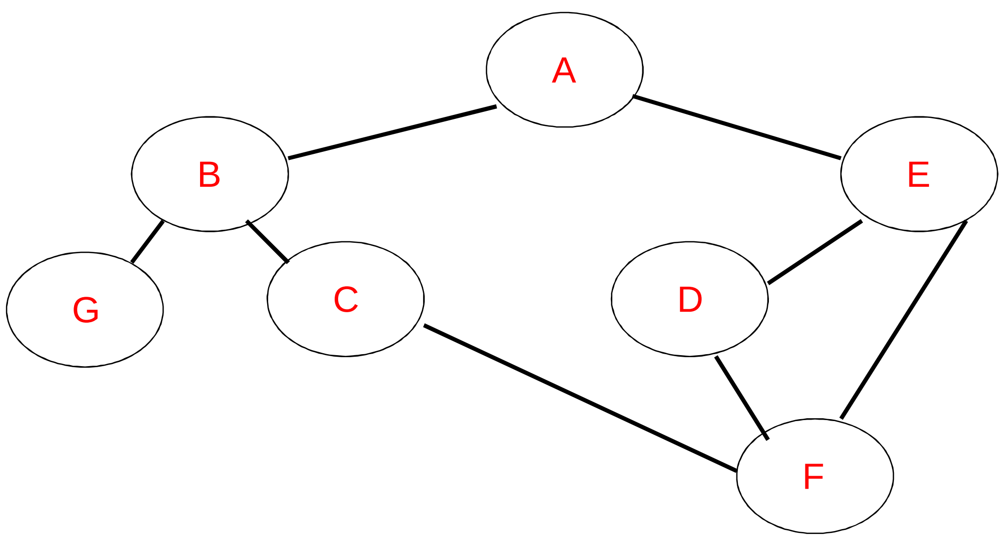
A and F are marked, so pick D.

Current: D



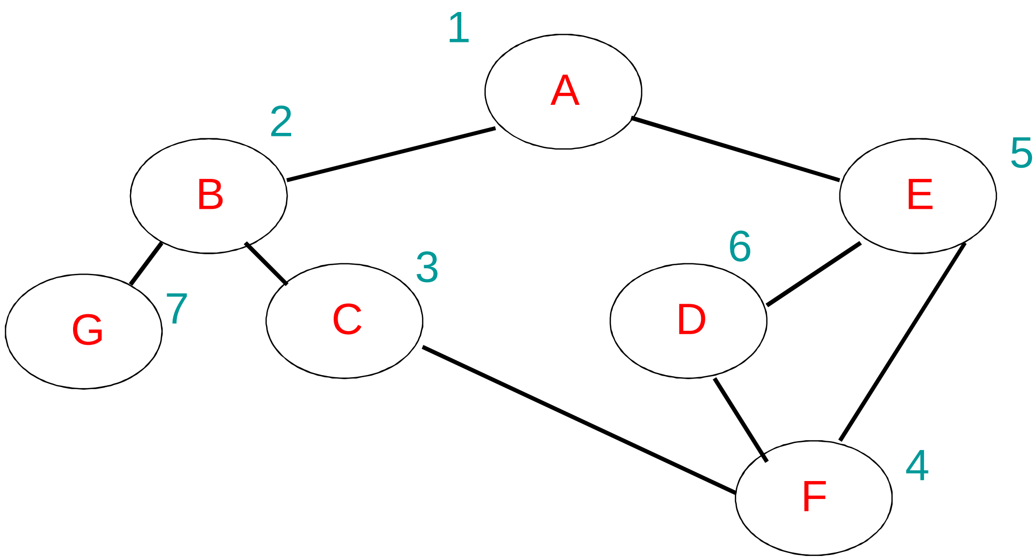
Visit D. No new vertices available. Backtrack to E. Backtrack to F. Backtrack to C. Backtrack to B

Current: G



Visit G. No new vertices from here. Backtrack to B. Backtrack to A. E already marked so no new.

Current:



Done. We have explored the graph in order:

A B C F E D G

Interesting features of DFS

- Complexity: O(|V| + |E|)
  - All vertices visited once, then marked
  - For each vertex on queue, we examine all edges
  - In other words, we traverse all edges once
- DFS does not necessarily find shortest path
  - Why?

Depth-first search (DFS)

- DFS strategy looks similar to pre-order. From a given node v, it first visits itself. Then, recursively visit its unvisited neighbours one by one.
- DFS can be defined recursively as follows.

Algorithm DFS(v)

printf v; // you can do other things!

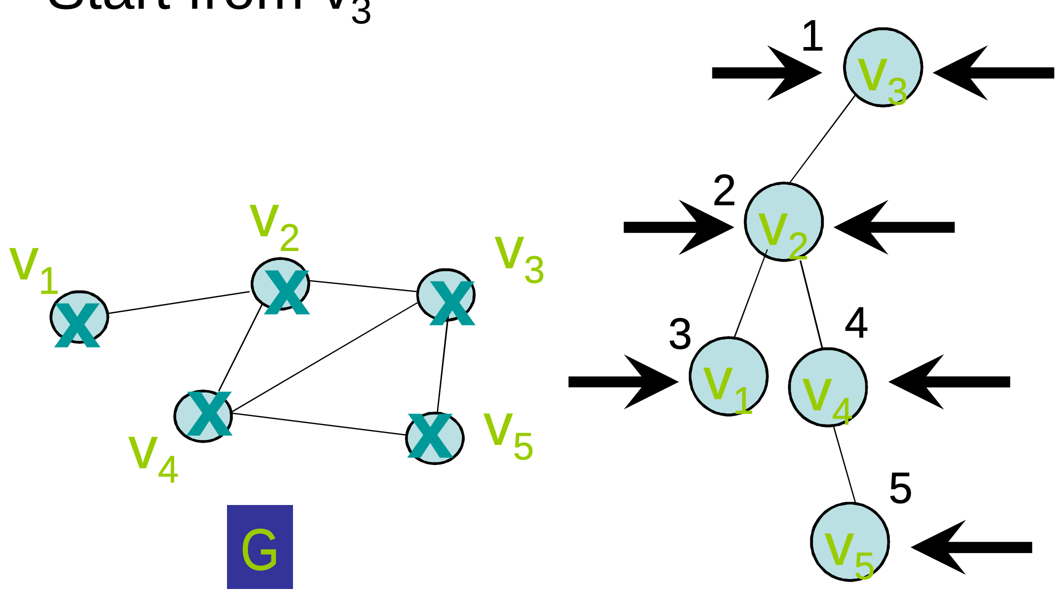
mark v as visited;

for (each unvisited node u adjacent to v)

    DFS(u);

DFS example

- Start from v_3

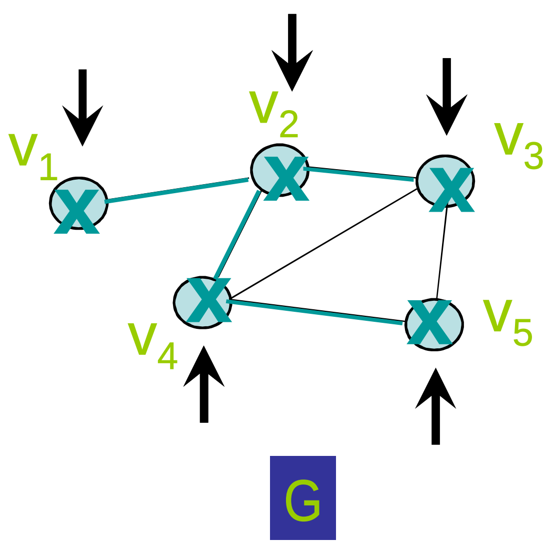


Non-recursive version of DFS algorithm

```
Algorithm dfs(v)
Initialize(s);
push(v,s);
mark v as visited;
while (!isEmpty(s)) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x)
        pop(s); // blacktrack
    else {
        select an unvisited node u adjacent to x;
        push(u,s);
        mark u as visited;
    }
}
```

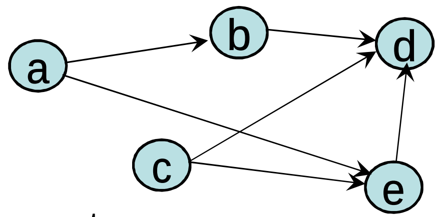
Non-recursive DFS example

visit	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4, v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty



Topological order

- Consider the prerequisite structure for courses:

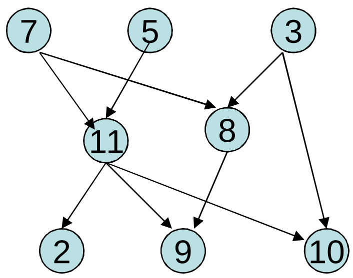


- Each node x represents a course x
- (x, y) represents that course x is a prerequisite to course y
- Note that this graph should be a directed graph without cycles (called a directed acyclic graph).
- A linear order to take all 5 courses while satisfying all prerequisites is called a topological order.
- E.g.
  - a, c, b, e, d
  - c, a, b, e, d

Topological Sort

- Topological sort: ordering of vertices in a directed *acyclic* graph such that if there is a path from vi to vj then vj appears after vi in the ordering
- Application: scheduling jobs.
  - Each job is a vertex in a graph, and there is an edge from *x* to *y* if job *x* must be completed before job *y* can be done.
  - topological sort gives the order in which to perform the jobs.
  - Instruction scheduling in
  - Example: Topological sort

Topological Sort



Topological sorts:
7, 5, 3, 11, 8, 2, 10, 9
5, 7, 3, 11, 8, 2, 10, 9
5, 7, 11, 2, 3, 8, 9, 10

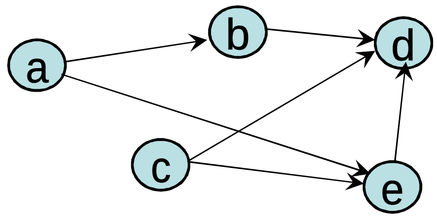
Topological sort

- Arranging all nodes in the graph in a topological order

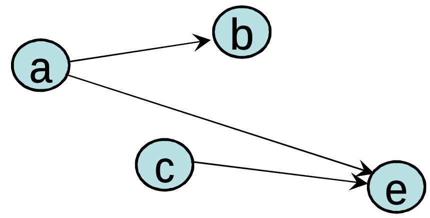
Algorithm topSort1

```
n = |V|;
for i = 1 to n {
    select a node v that has no successor (no outgoing edge);
    print this vertex;
    delete node v and its edges from the graph;
}
```

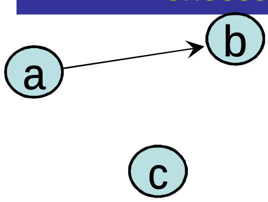
Example

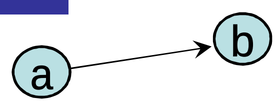


1. d has no successor! Choose d!



2. Both b and e have no successor! Choose e!





3. Both b and c have no successor! Choose c!

4. Only b has no successor! Choose b!

5. Choose a!
The topological order is **a,b,c,e,d**

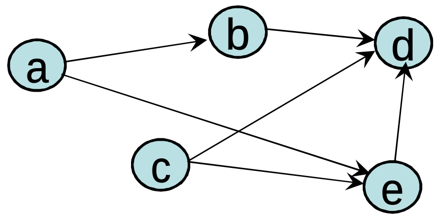
Topological sort

- Arranging all nodes in the graph in a topological order

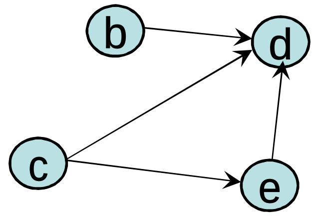
Algorithm topSort2

```
n = |V|;
for i = 1 to n {
   select a node v that has no ancestors (no incoming edges);
   print this vertex;
   delete node v and its edges from the graph;
}
```

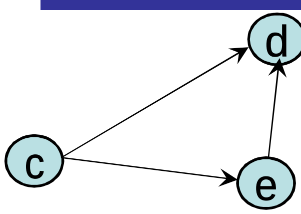
Example



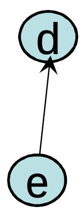
1. a has no ancestors! Choose a!



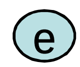
2. Both b and c have no ancestosr! Choose b!



3. Only c has no ancestors! Choose c!



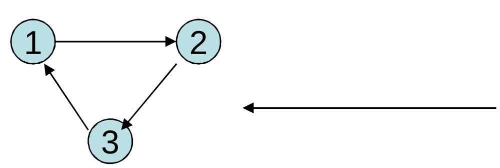
4. Only e has no ancestors! Choose !



5. Choose e!
The topological order is **a,b,c,e,d**

Topological Sorting

- What happens if graph has a cycle?
  - Topological ordering is not possible
  - For two vertices v & w, v precedes w and w precedes v



Every edge has an incoming vertex so topological sort can not be performed

- Topological sorts can have more than one ordering

Topological Sorting

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    output error message (graph has at least one cycle)
else
    output message (proposed topologically sorted order: L)
```

Topological sort algorithm 2

- This algorithm is based on DFS

Algorithm topSort2

```
createStack(s);
for (all nodes v in the graph) {
    if (v has no incomming edges) {
        push(v,s);
        mark v as visited;
    }
}
while (!isEmpty(s)) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x) { // i.e. x has no unvisited successor
        printf x;
        pop(s); // blacktrack
    } else {
        select an unvisited node u adjacent to x;
        push(u,s);
        mark u as visited;
    }
}
;
```