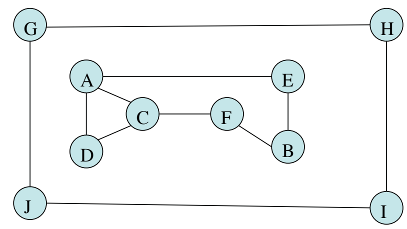
Breadth-First Search using a FIFO Queue

```
BFS-Init (G):
 for all nodes v in V
    if v is white
        BFS(G, v)
```

```
BFS (G, s):
 s.d = 0
 s.c = gray
 for each node v != s
    v.d = ∞
    v.c = white
 Q.enqueue (s)
 while Q ≠ Ø
    u = Q.dequeue()
    for each v adjacent to u
        if v.c == white
            v.c = gray
            v.d = u.d + 1
            v.π = u
            Q.enqueue(v)
    u.c = black
```

An initialization algorithm like that shown above would ensure that all nodes are visited, even in disjoint sets of nodes.

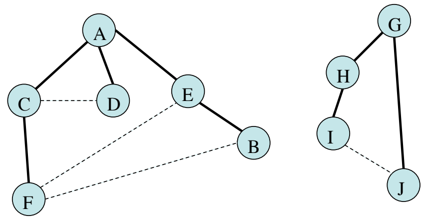
Example BFS Traversal



Order of visiting: a_1 c_2 d_3 e_4 f_5 b_6 g_7 h_8 i_9 j_{10}

Distance of vertex : 0 1 1 1 2 3 ∞ ∞ ∞ ∞

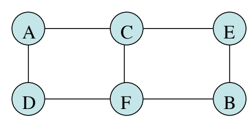
Breadth-first Search Forest



Tree edges are solid lines and dashed lines are cross edges.

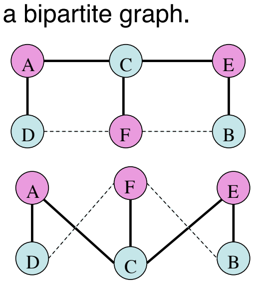
Bipartite Graphs

A graph is bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y, i.e., if its vertices can be colored in 2 colors so that every edge has its end points colored in different colors.



Bipartite Graphs

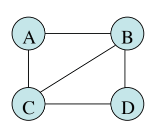
Explain how BFS could be used to detect a bipartite graph.



Mark the source, A, with color1, mark the nodes at level 1 with color2, and so on. Every node on an even numbered level will be color1 and every odd color2

Bipartite Graphs

Is this graph bipartite? No. The edge (B,C) would have to connect nodes of the same color



Depth-First Traversal

Depth-First Traversal is another algorithm for traversing a graph.

Called depth-first because it searches "deeper" in the graph whenever possible.

Edges are explored out of the most recently discovered vertex v that still has unexplored edges. When all of v's edges have been explored, the search "backtracks" to explore the edges incident on the vertex from which v was discovered.

We will use an algorithm with a stack, S, to manage the set of nodes.

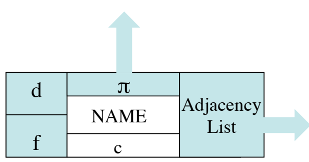
Depth-First Search

DFS algorithm maintains the following information for each vertex **u**:

- **u.c** (white, gray, or black) : indicates status
  - white = not discovered yet
  - gray = discovered, but not finished
  - black = finished
- **u.d** : discovery time of node u
- **u.f** : finishing time of node u
- **u.π** : predecessor of u in Depth-First tree

DFS node

Each node has fields for predecessor (π), discovery time (d), finish time (f) and color (c). Each node also has an associated adjacency list with pointers to neighboring nodes.



Depth-First Search

```
DFS-Init (G, s):
time = 0
for all nodes v
    v.d = v.f = ∞
    v.c = white
    v.π = NONE
S = ∅
DFS (G, s)
```

Initialize global timer to 0.

Set discovery time and finish times of all nodes to infinity and color them white.

Initialize stack **S** to **∅.**

Call DFS (G, s)

Depth-First Search Using a Stack

```
DFS (G,s)
S.push(s)
while S is not empty
   u = S.peek()
   if u.c == WHITE
       u.c = GRAY
       u.d = time
       time = time + 1
       for all white neighbors v of u
           v.π = u
           S.push(v)
   else if u.c == GRAY
       S.pop()
       u.c = BLACK
       u.f = time
       time = time + 1
   else // u is BLACK
       S.pop()
end while
```

Complexity is based on number of edges |E|

Complexity (Adjacency List)
- check all edges adjacent to each node from both directions - O(E) time
- total = O(V + E) = O(V^2) (w.c.)

Depth-First Search (recursive version)

Initially, time (counter) = 0

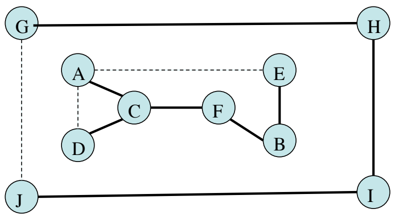
After execution, for every vertex u, u.d < u.f

DFS (G)
```
for each w ∈ G
    if w.c == white
        DFS-Visit (G,w)
```

Note: If G = (V, E) is not connected, then DFS will still visit the entire graph with the additional code above.

DFS-Visit (G,u)
```
u.c = gray
u.d = time
time = time + 1
for each v adjacent to u
    if v.c == white
        v.π = u
        DFS-Visit(G,v)
    end if
end for
u.c = black
u.f = time
time = time + 1
```

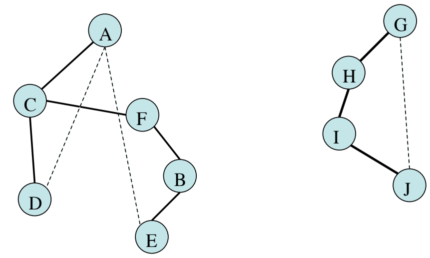
Example DFS Traversal



The first subscript indicates the time at which each node is discovered and pushed onto stack; the second indicates the time at which the node was finished.

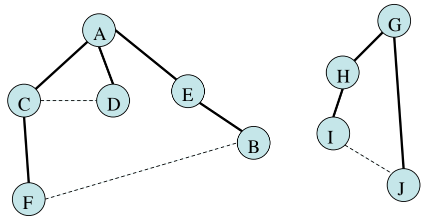
$e_{7,8}$
$b_{6,9}$
$f_{5,10}$
$d_{3,4}$
$c_{2,11}$
$a_{1,12}$
$j_{16,17}$
$i_{15,18}$
$h_{14,19}$
$g_{13,20}$

Depth-first Search Forest



Tree edges are solid lines and dashed lines are back edges.

Breadth-first Search Forest



Tree edges are solid lines and dashed lines are cross edges.

Facts about DFS and BFS on undirected graph

	DFS	BFS
Data Structure	Stack	Queue
No. of vertex orderings	2 orderings	1 ordering
Edge Types	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Depth-First Search

Theorem 22.7 (parenthesis theorem) For any two vertices u and v, exactly one of the following three conditions hold:

1. **Either the intervals [u.d, u.f] and [v.d, v.f] are entirely disjoint, or**
2. **the interval [u.d, u.f] is contained entirely within the interval [v.d, v.f], (and u is a descendant of v), or**
3. **the interval [v.d, v.f] is contained entirely within the interval [u.d, u.f], (and v is a descendant of u).**

Proof: <u>Case 1</u> – u.d < v.d. If v.d < u.f, then v was discovered when u was still gray, implying v is a descendant of u. Since v was discovered before u is finished, all of v's outgoing edges will have been explored before u is finished. Therefore, the interval [v.d,v.f] is contained within the interval [u.d,u.f] (condition 3 holds, v is descendant of u).

If u.d < v.d and u.f < v.d, then u was discovered and finished before v was discovered (condition 1 holds, intervals are disjoint).

Depth-First Search

Theorem 22.7 (parenthesis theorem) For any two vertices u and v, exactly one of the following three conditions hold:

1. **Either the intervals [u.d, u.f] and [v.d, v.f] are entirely disjoint, or**
2. **the interval [u.d, u.f] is contained entirely within the interval [v.d, v.f], (and u is a descendant of v), or**
3. **the interval [v.d, v.f] is contained entirely within the interval [u.d, u.f], (and v is a descendant of u).**

Proof: <u>Case 2</u> – v.d < u.d. Similar argument to case 1 on last slide (condition 2 if u.f < v.f or condition 1 if v.f < u.f). ■

Corollary 22.8 (nesting of descendant's intervals): Vertex v is a descendant of vertex u in the DFS forest for a graph G iff u.d < v.d < v.f < u.f.

Depth-First Search

Theorem 22.9 (White-Path theorem):
In a depth-first forest of a graph G, vertex v is a descendant of vertex u iff at the time u.d that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Proof:
Forward direction of iff: Assume v is a descendant of u in a depth-first tree. Let w be any vertex on the path between u and v in the depth-first tree, so that w is also a descendant of u. By Corollary 22.8, u.d < w.d, and so w is white at time u.d and v is reachable on a path of white vertices.

Theorem 22.9 (White-Path theorem):
In a depth-first forest of a graph G, vertex v is a descendant of vertex u iff at the time u.d that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Proof:
Backward Direction of iff: (by contradiction) Assume vertex v is reachable from u along a path consisting of white vertices at time u.d, but v does not become a descendant of u in the DFS. Assume, wlog, that every other vertex between u and v becomes a descendant of u. Let w be the immediate predecessor of v on a path from u. Then by corollary 22.8, w.f ≤ u.f (or w.f=u.f if w=u).

Since v is reachable from u via a path of white vertices by assumption, v must be discovered after u, but before w is finished. So u.d < v.d < w.f ≤ u.f. So by the parenthesis theorem (22.7), [v.d, v.f] must be contained within [u.d,u.f] and v must be a descendant of u, a contradiction. ∎

DFS Tree

DFS builds a depth-first tree whose edges can be traced from any node to s using the π values at each node.

The DFS algorithm defines a depth-first forest G_π.

Topological Sort - Application of DFS

input: directed acyclic graph (DAG)
output: ordering of nodes s.t. if (u,v) ∈ E, then u comes before v in ordering

Topological-Sort (G)
1. call DFS(G,s) to compute finishing times v.f for each v
2. as each vertex is finished, insert it at *head* of a linked list
3. return the linked list of vertices

Complexity (Adjacency List Representation) - O(V + E)

Topologically sorted vertices are ordered in *reverse* order of their finishing times. An application of this type of sorting algorithm is to indicate precedence among ordered events represented in a DAG.

Topological Sort - Application of DFS

Lemma 22.11: A directed graph G is *acyclic* iff a DFS of G yields no back edges.

Proof: (by contrapositive: show if !q then !p)
➔ Suppose there is a back edge (u,v). Then vertex v is an ancestor of u in the depth-first forest. Thus, there is a path from v to u in G and the back edge (u,v) completes the cycle, so G is not a DAG.
← Suppose G contains a cycle c (therefore is not a DAG). Let v be the first vertex discovered in c, and let (u,v) be the edge incoming at v in c. At time v.d, the vertices of c form a white path from v to u. Then by Thm. 22.9, u becomes a descendant of v. Therefore, (u,v) is a back edge.

Topological Sort - Application of DFS

Theorem 22.12: Topological-Sort(G) produces a topological sort or *precedence graph* of a DAG.

Proof: Consider any edge in DAG G from u to v. We need to show that, for any pair of distinct vertices u and v s.t. there is an edge from u to v, v.f < u.f.

When edge (u,v) is explored, v cannot be gray, since then (u,v) would be a back edge, contradicting L. 22.11. Therefore, v must be either white or black when edge (u,v) is explored. If v is white, it becomes a descendant of u and finishes before u. If v is black, it has already finished. In either case, v.f < u.f.

Finding Strongly Connected Components of a Digraph

A digraph is strongly connected if, for any distinct pair of vertices u and v there exists a directed path from u to v and a directed path from v to u. In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths of the digraph; these subsets are called strongly connected components.

input: directed graph G
output: strongly connected components of G

1. do a DFS traversal of the digraph and number its vertices in the order that they become dead ends.
2. reverse the directions of all the edges of the digraph to get (G^T)
3. do a DFS traversal of G^T by starting the traversal at the highest numbered vertex and consider the vertices in order of decreasing *u.f*
4. output the vertices of each tree in the DFF from line 3 as G^{SCC}

Finding Strongly Connected Components of a Digraph

input: directed graph G
output: strongly connected components of G

1. do a DFS traversal of the digraph and number its vertices in the order that they become dead ends.
2. reverse the directions of all the edges of the digraph to get (G^T)
3. do a DFS traversal of G^T by starting the traversal at the highest numbered vertex and consider the vertices in order of decreasing *u.f*
4. output the vertices of each tree in the DFF from line 3 as G^{SCC}

The strongly connected components G^{SCC} are exactly the subsets of vertices in each DFS tree obtained during step 3, the last traversal.

Time complexity of this algorithm?