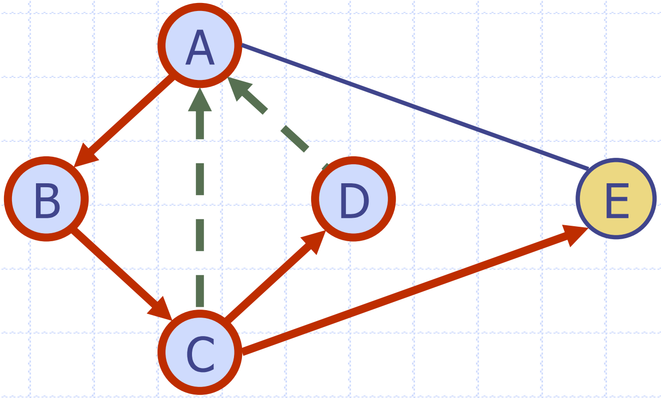
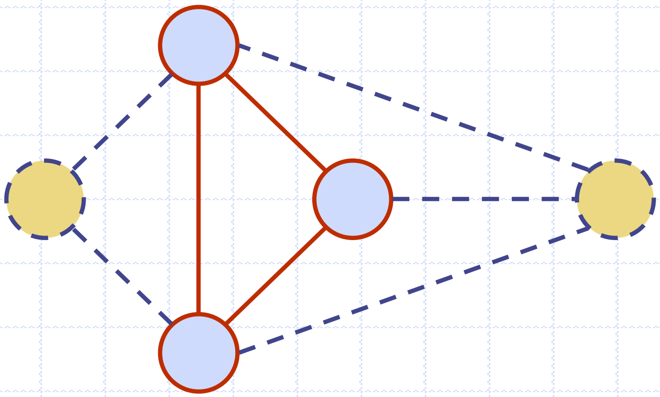
Depth-First Search

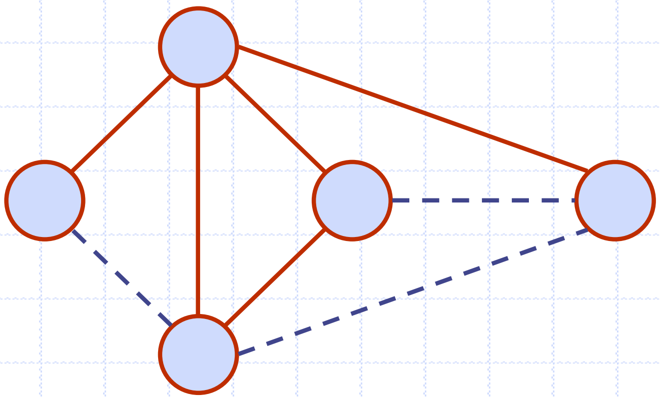


Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



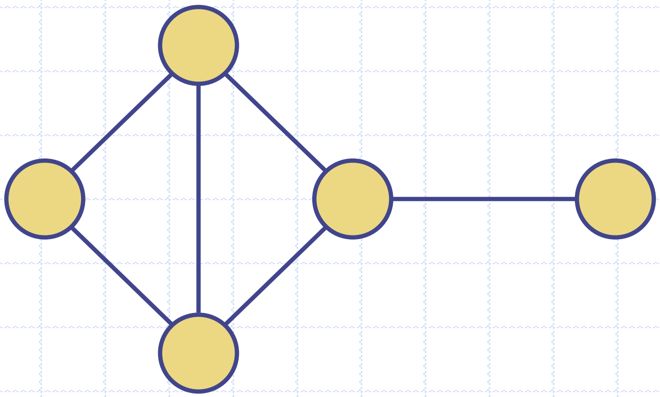
Subgraph



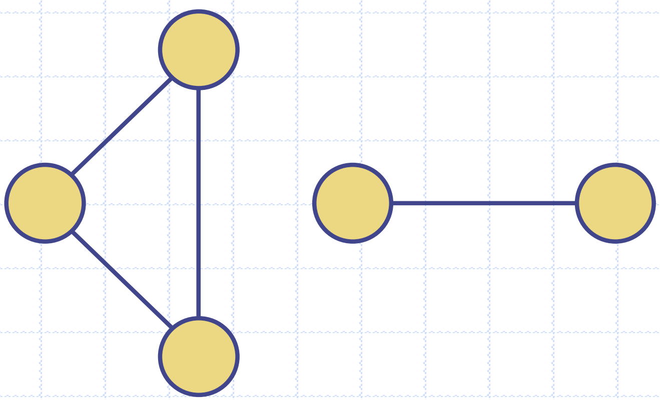
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



Connected graph

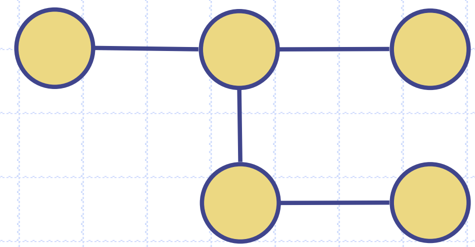


Non connected graph with two connected components

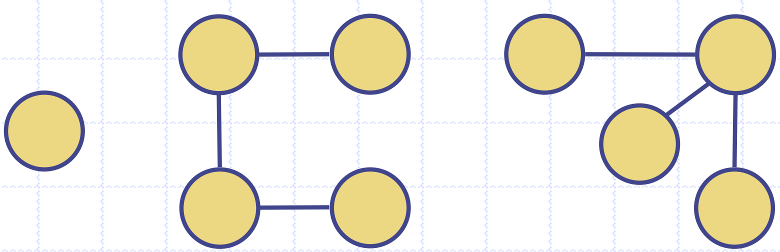
Trees and Forests

Any node can be the root to have a rooted tree

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles
- A forest is an undirected graph without cycles
  - The connected components of a forest are trees



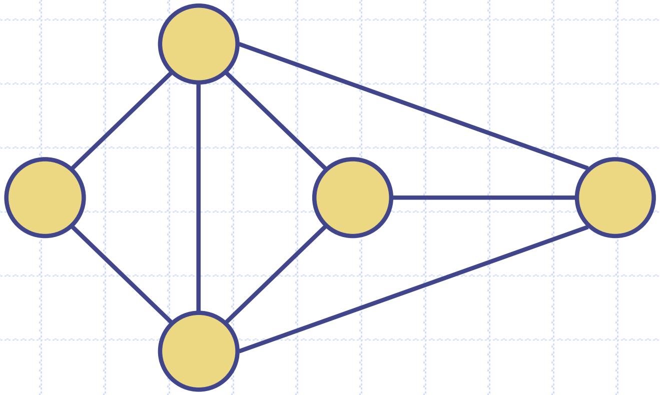
Tree



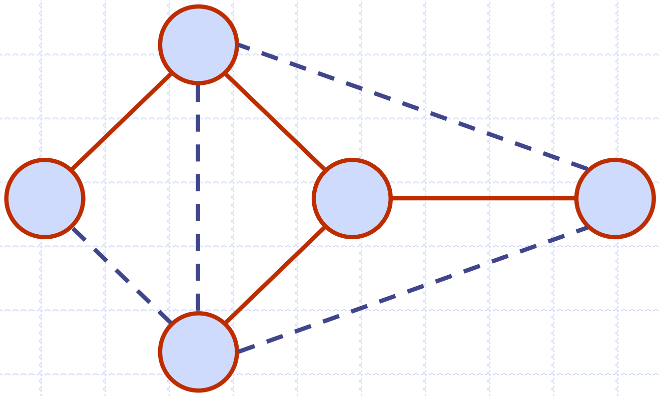
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Traversal of Graphs

- Traversal: given $G = (V,E)$ and vertex v, find or visit all $w \in V$, such that w connects v
  - Depth First Search (DFS)
  - Breadth First Search (BFS)
- Applications
  - Connected component
  - Spanning trees
  - ...

Depth-First Search (DFS)

- A general technique for traversing a graph
- DFS traversal of graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- Complexity: $O(n+m)$ for a graph with n vertices and m edges
- DFS for other graph problems (with less memory requirement)
  - Find a path between two given vertices
  - Find a cycle
- DFS is to graphs what preorder is to binary/general rooted trees

DFS Algorithm

- Begin the search by visiting the start vertex v
  - If v has an unvisited neighbor, traverse it recursively

```
void dfs(int v){
  node_pointer w;
  visited[v]=TRUE;
  printf("%5d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Simple pseudo code from the reference book

DFS Algorithm

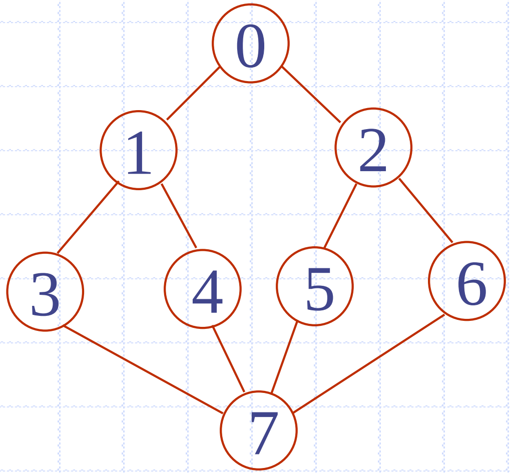
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm *DFS*(*G*)
  Input graph *G*
  Output labeling of the edges of *G*
    as discovery edges and
    back edges
  for all *u* ∈ *G.vertices*()
    u.setLabel(*UNEXPLORED*)
  for all *e* ∈ *G.edges*()
    e.setLabel(*UNEXPLORED*)
  for all *v* ∈ *G.vertices*()
    if *v.getLabel*() = *UNEXPLORED*
      DFS(*G*, *v*)

Algorithm *DFS*(*G*, *v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the edges of *G*
    in the connected component of *v*
    as discovery edges and back edges
  v.setLabel(*VISITED*)
  for all *e* ∈ *G.incidentEdges*(*v*)
    if *e.getLabel*() = *UNEXPLORED*
      w ← *e.opposite*(*v*)
      if *w.getLabel*() = *UNEXPLORED*
        e.setLabel(*DISCOVERY*)
        DFS(*G*, *w*)
      else
        e.setLabel(*BACK*)

Example via Recursive Calls

- Start vertex: 0
- Traverse order: 0, 1, 3, 7, 4, 5, 2, 6



- Equivalent adjacency list

Usually sorted

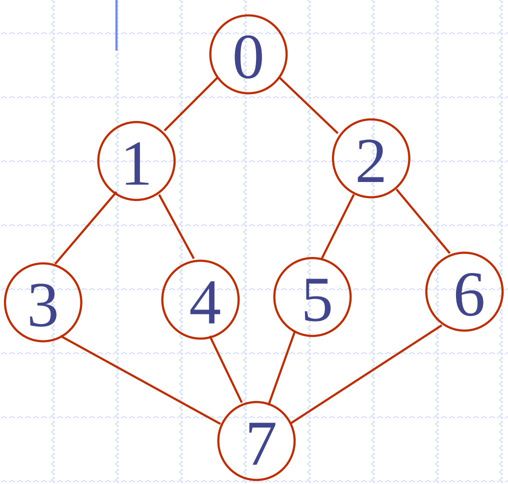
```
vertex 0 -> 1 -> 2
vertex 1 -> 0 -> 3 -> 4
vertex 2 -> 0 -> 5 -> 6
vertex 3 -> 1 -> 7
vertex 4 -> 1 -> 7
vertex 5 -> 2 -> 7
vertex 6 -> 2 -> 7
vertex 7 -> 3 -> 4 -> 5 -> 6
```

Animation

Example via Stack

Quiz!

- Start vertex: 0
- Traverse order: 0, 1, 3, 7, 4, 5, 2, 6



vertex 0 -> 1 -> 2
vertex 1 -> 0 -> 3 -> 4
vertex 2 -> 0 -> 5 -> 6
vertex 3 -> 1 -> 7
vertex 4 -> 1 -> 7
vertex 5 -> 2 -> 7
vertex 6 -> 2 -> 7
vertex 7 -> 3 -> 4 -> 5 -> 6

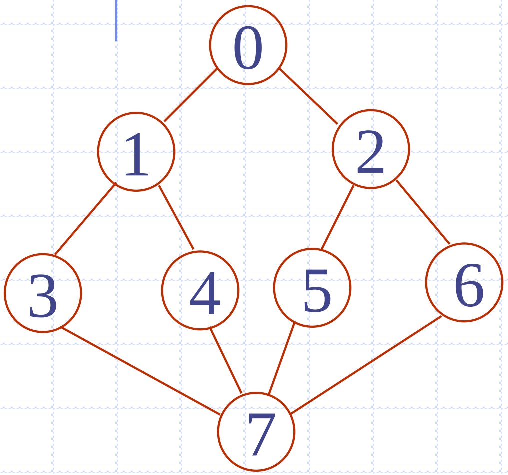
- Stack contents at each step:

Output	Stack
	0
0	2 1
1	2 4 3 0
3	2 4 7 1
7	2 4 6 5 4 3
4	2 4 6 5 7 1
5	2 4 6 7 2
2	2 4 6 6 5 0
6	2 4 6 7 2

Another Example via Stack

Quiz!

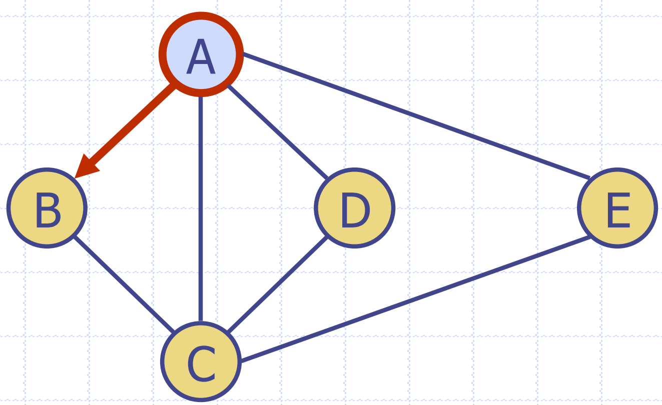
- Start vertex: 4
- Traverse order: ?
- Stack contents at each step:

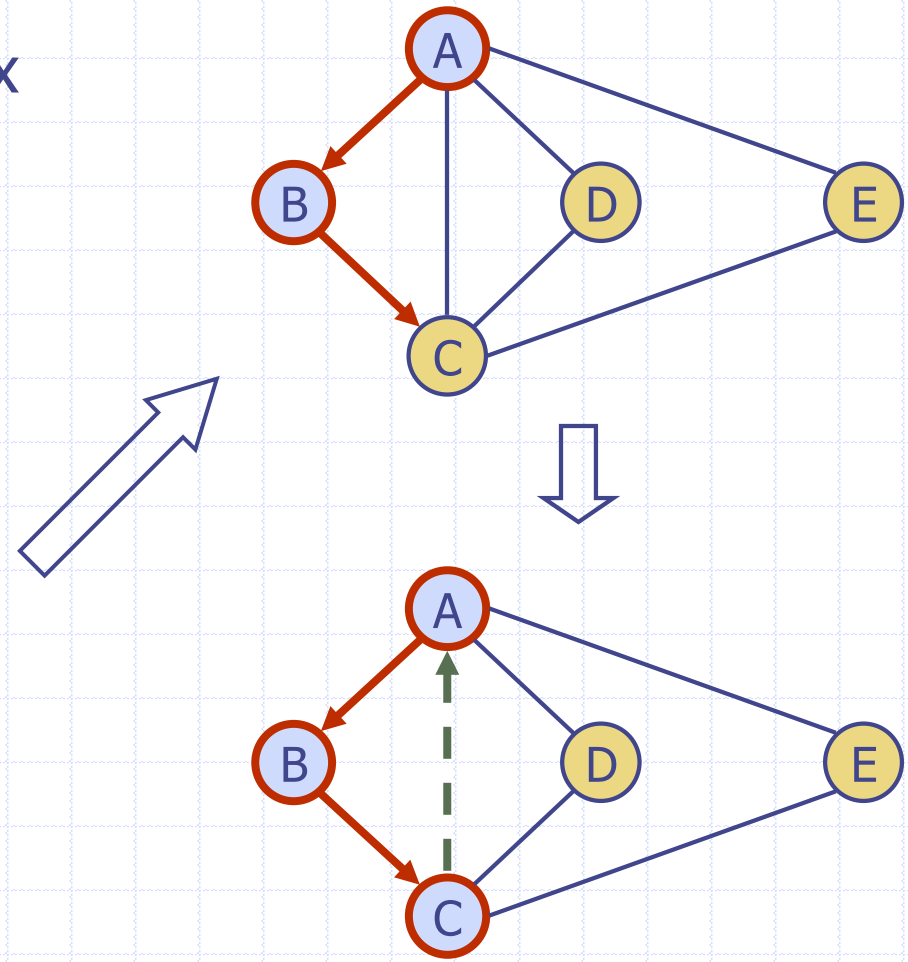


vertex 0 -> 1 -> 2
vertex 1 -> 0 -> 3 -> 4
vertex 2 -> 0 -> 5 -> 6
vertex 3 -> 1 -> 7
vertex 4 -> 1 -> 7
vertex 5 -> 2 -> 7
vertex 6 -> 2 -> 7
vertex 7 -> 3 -> 4 -> 5 -> 6

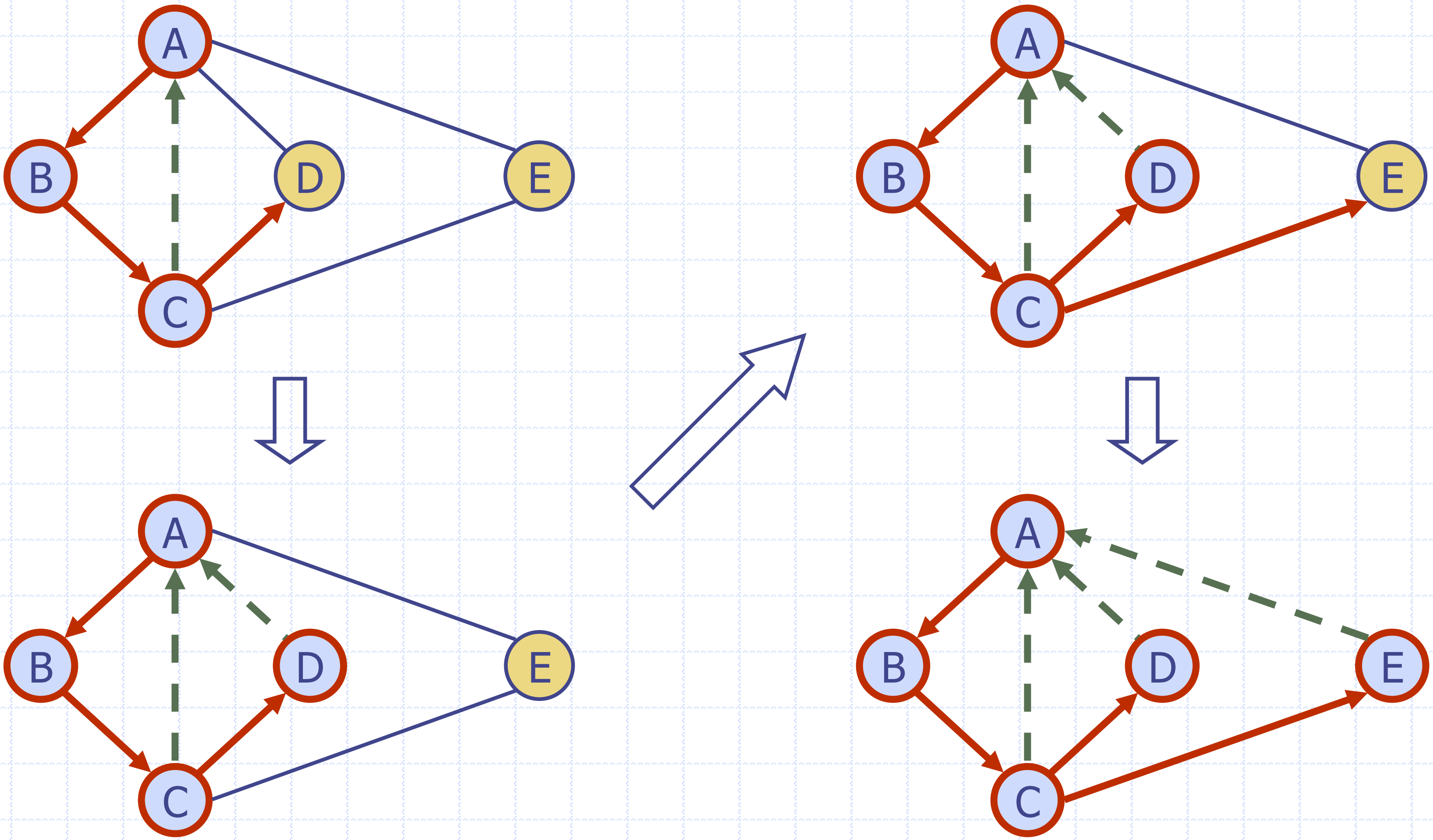
Example in Textbook

- A unexplored vertex
- A visited vertex
- unexplored edge
- discovery edge
- back edge

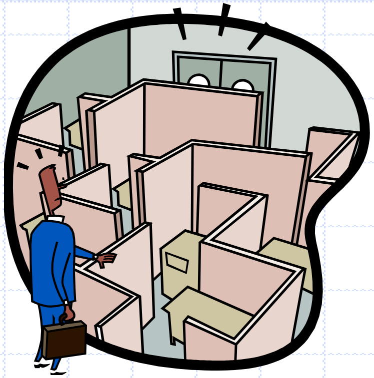




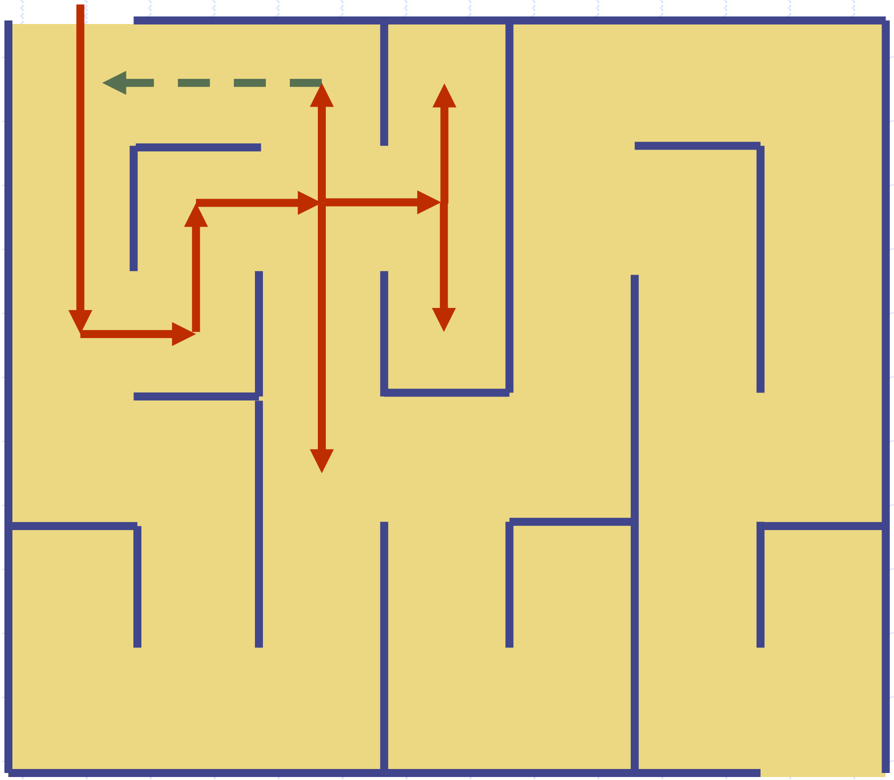
Example (cont.)



DFS and Maze Traversal



- DFS is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



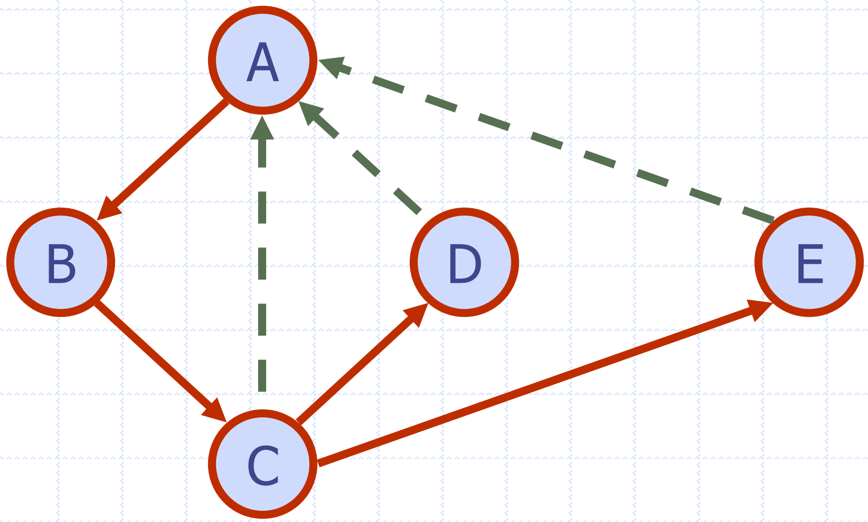
Properties of DFS

Property 1

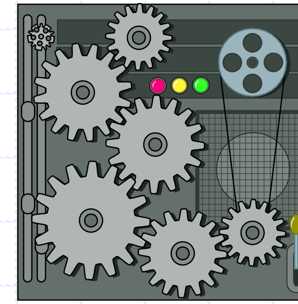
DFS*(*G*, *v*)** visits all the vertices and edges in the connected component of ***v

Property 2

The discovery edges labeled by ***DFS*(*G*, *v*)** form a spanning tree of the connected component of ***v***



Analysis of DFS



- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
  v.setLabel(VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ v.incidentEdges()
    if e.getLabel() = UNEXPLORED
      w ← e.opposite(v)
      if w.getLabel() = UNEXPLORED
        e.setLabel(DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        e.setLabel(BACK)
  S.pop(v)
```

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS(G, v, z)
    v.setLabel(VISITED)
    S.push(v)
    for all  e ∈ v.incidentEdges()
        if  e.getLabel() = UNEXPLORED
            w ← e.opposite(v)
            S.push(e)
            if w.getLabel() = UNEXPLORED
                e.setLabel(DISCOVERY)
                pathDFS(G, w, z)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

Applications of DFS

- DFS applications
  - Cycle detection in a graph
  - Topological sorting
  - Finding strongly connected components
  - Path finding
  - Detecting bipartite graph
  - Solving puzzles/games
- Reference