Depth-First Search

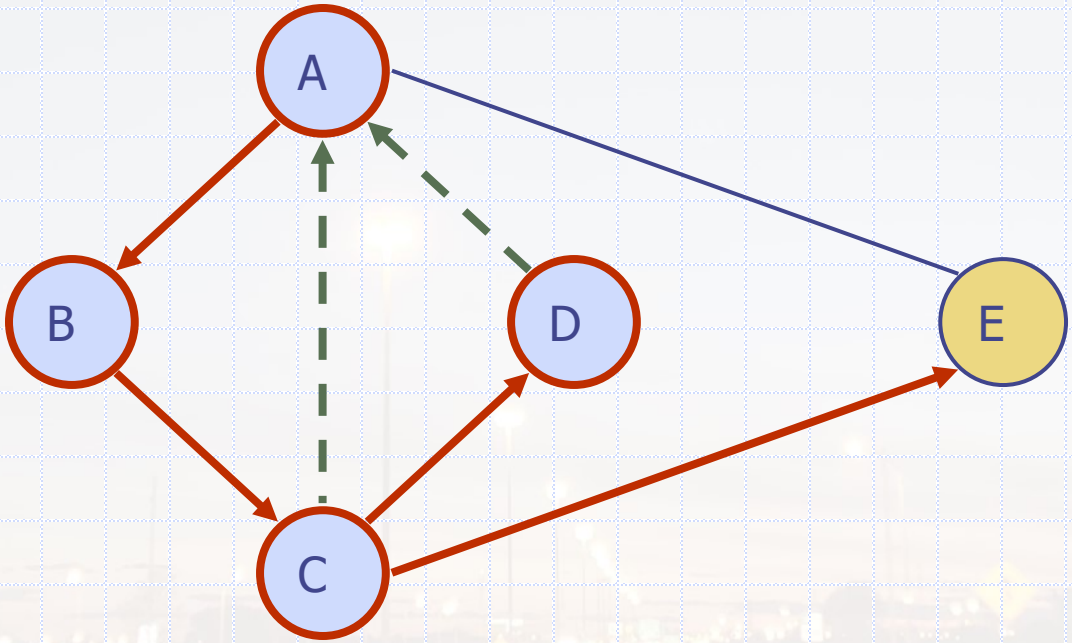
Dr. Aiman Hanna

Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

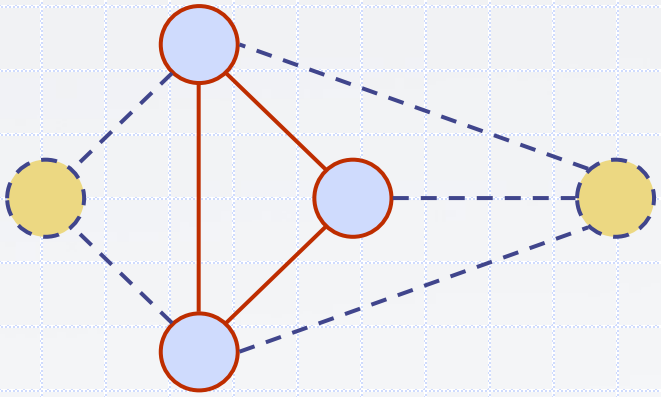
Coverage

- Graph Traversal
  - Depth-First Search (DFS)

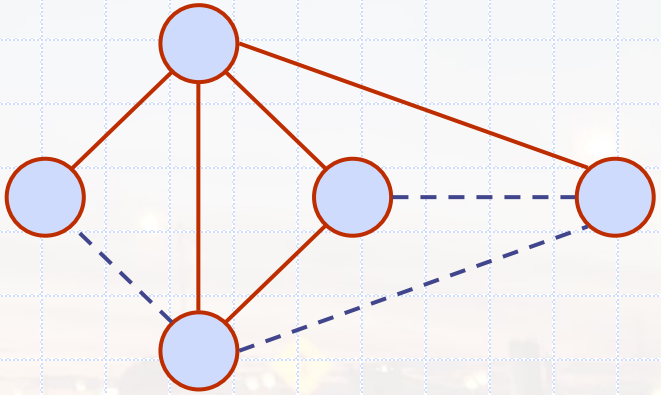


Subgraphs

- 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



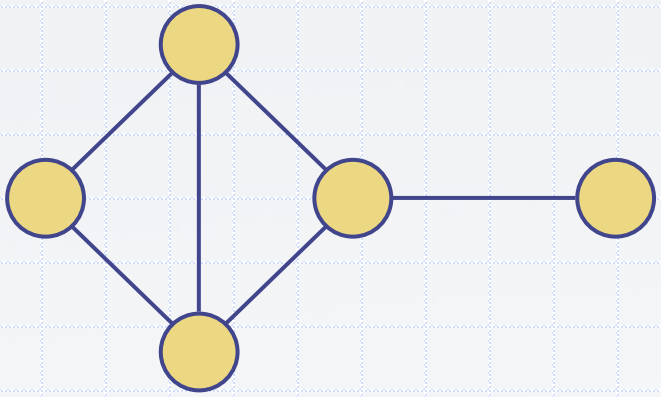
Subgraph (includes the vertices with the solid lines)



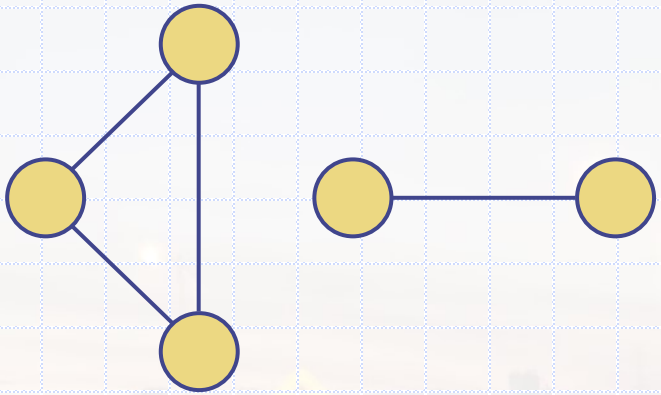
Spanning subgraph (includes all the vertices of the original graph)

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

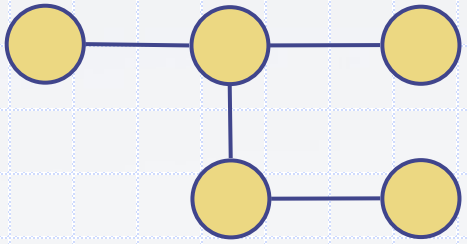


Non-connected graph with two connected components

Trees and Forests

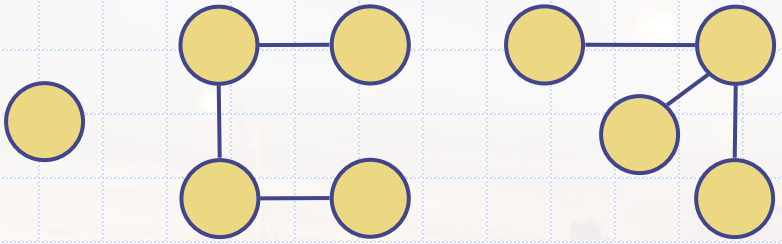
- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of a (free) tree is different from the one of a rooted tree; that is, it has no root



Tree

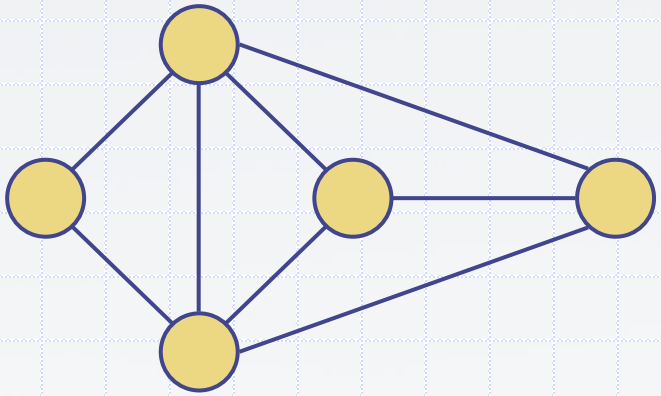
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



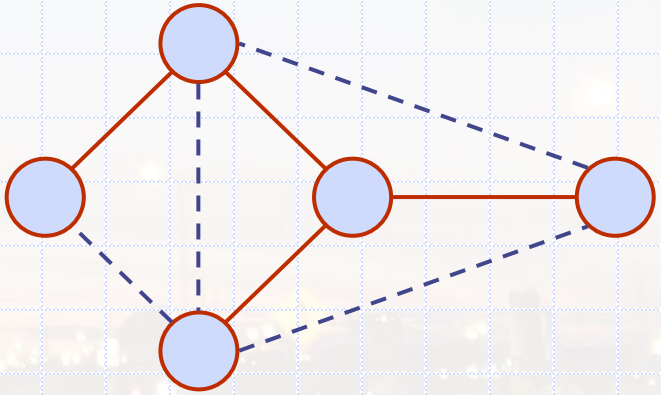
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

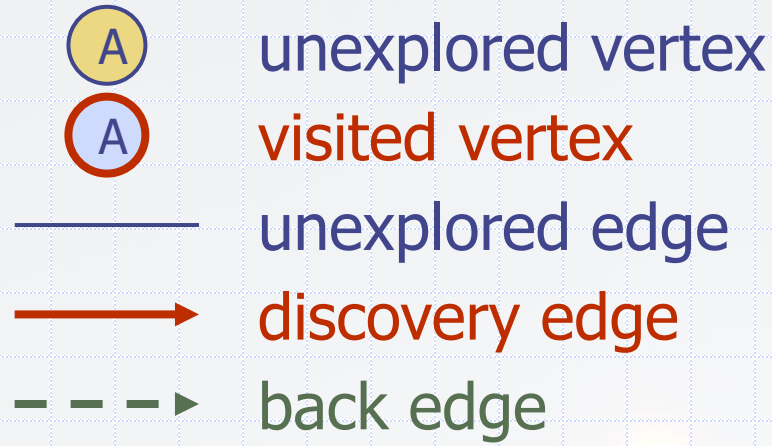
Graph Traversals

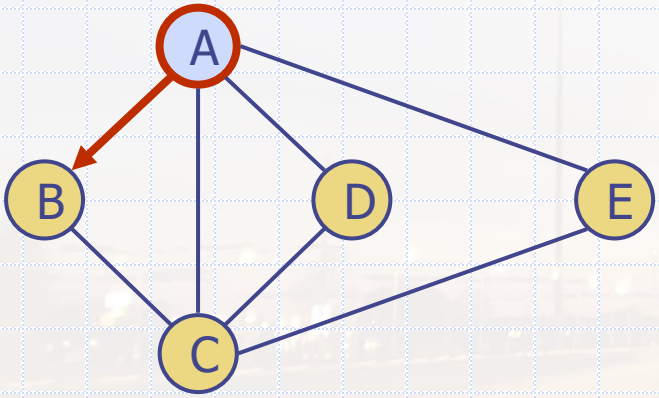
- Given a graph, it is important to be able to traverse the graph (visits all its vertices).
- Depth-first search (DFS) is a general technique for traversing a graph.
- Other graph traversal techniques exist.

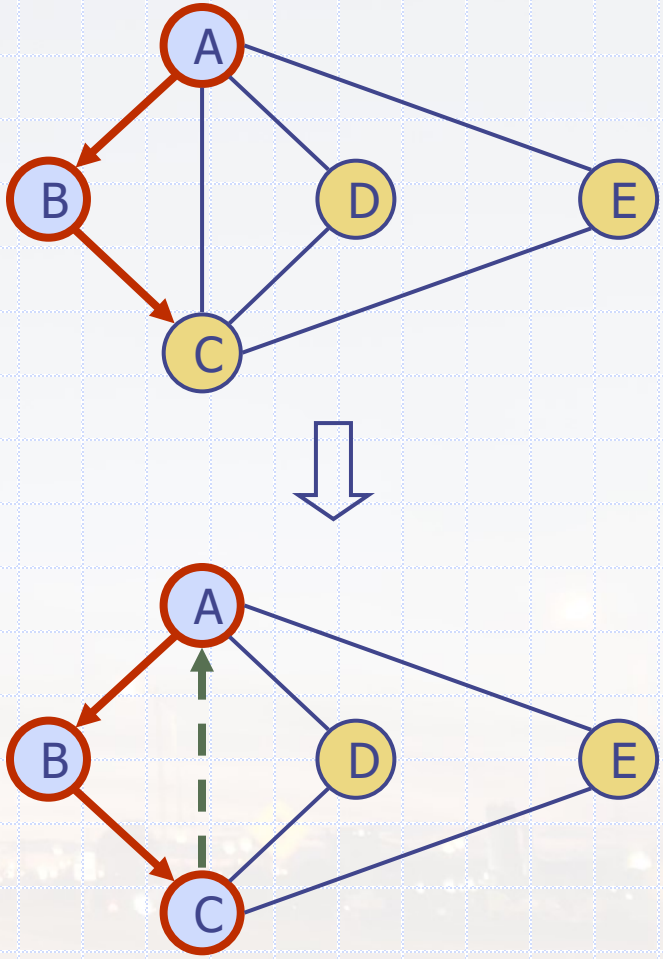
DFS

- The idea of DFS is similar to wondering inside a maze (*equipment needed are a rolled robe, and a can of paint spray*):
  - In order to be able to visit all the nodes (intersections) and come back out, fix one end of the rolled thread/robe to the entrance, and unroll the robe as you move
  - Every time you visit a node for the first time, mark it with the paint
  - If you come to a unpainted node, paint it, then go further to its (or one of its) next node(s), unroll the robe along the way as always
  - If you come to a previously visited node, immediately go back to where you came from (also roll back the robe along that way back)
  - If the node you are back to has other possible paths, explore each of them in a similar way
  - If all the paths have already been explored then go back the previous node to that node (the one you originally came from)
  - Repeat the operations along the way/ Notice that each node can either be visited or unvisited, and each node can either have all its further paths explored or some of its paths are not yet explored.
  - Rolling back all the way to the node you start from (the entrance), concludes successful termination of the traversal.

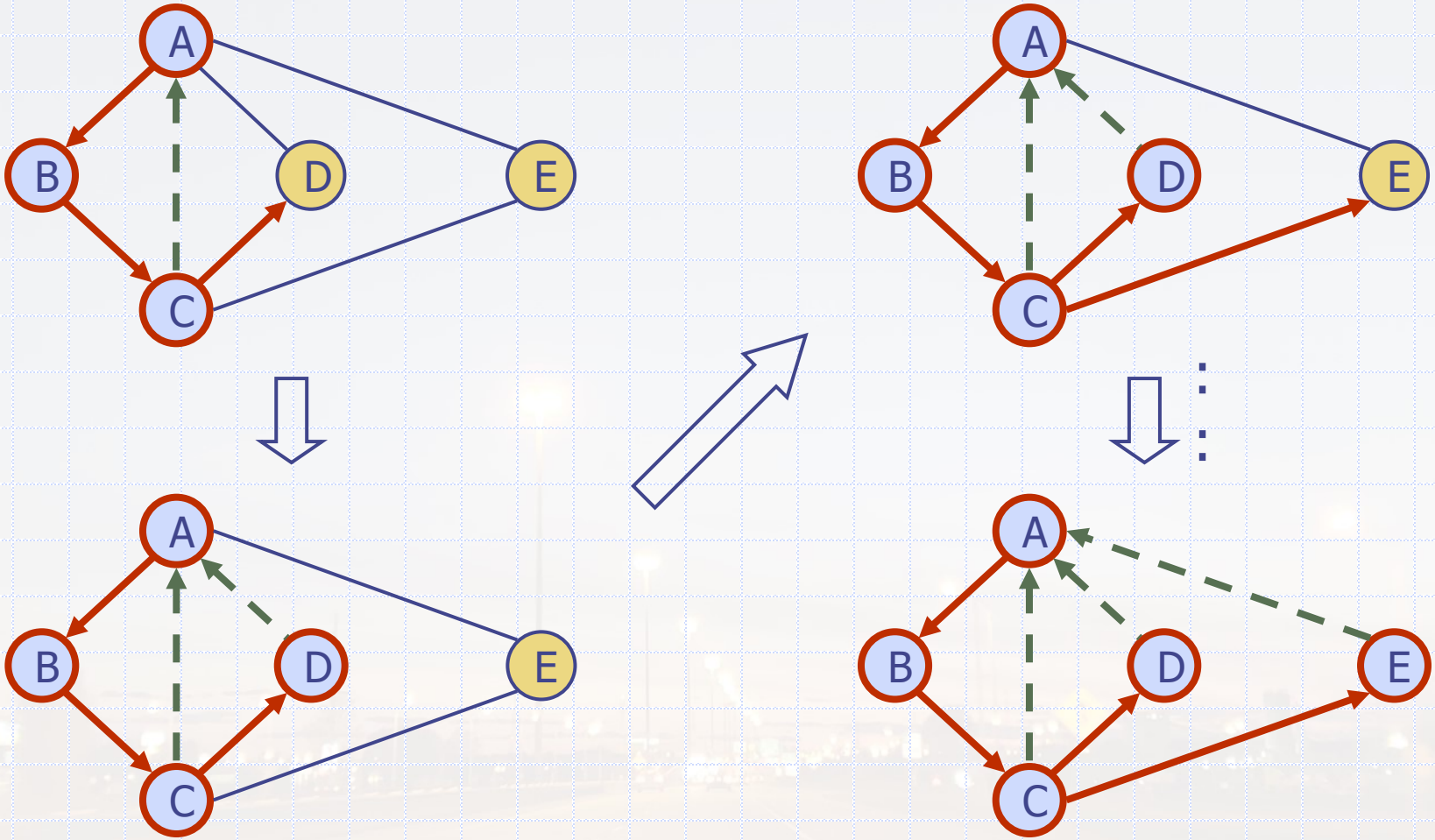
DFS Example1



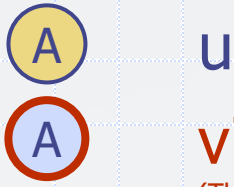




DFS Example1 (cont.)



DFS Example2



unexplored vertex

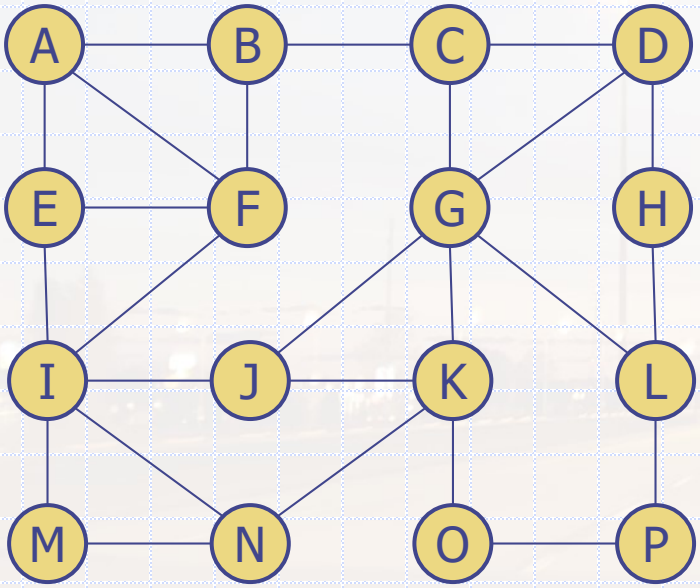
visited vertex

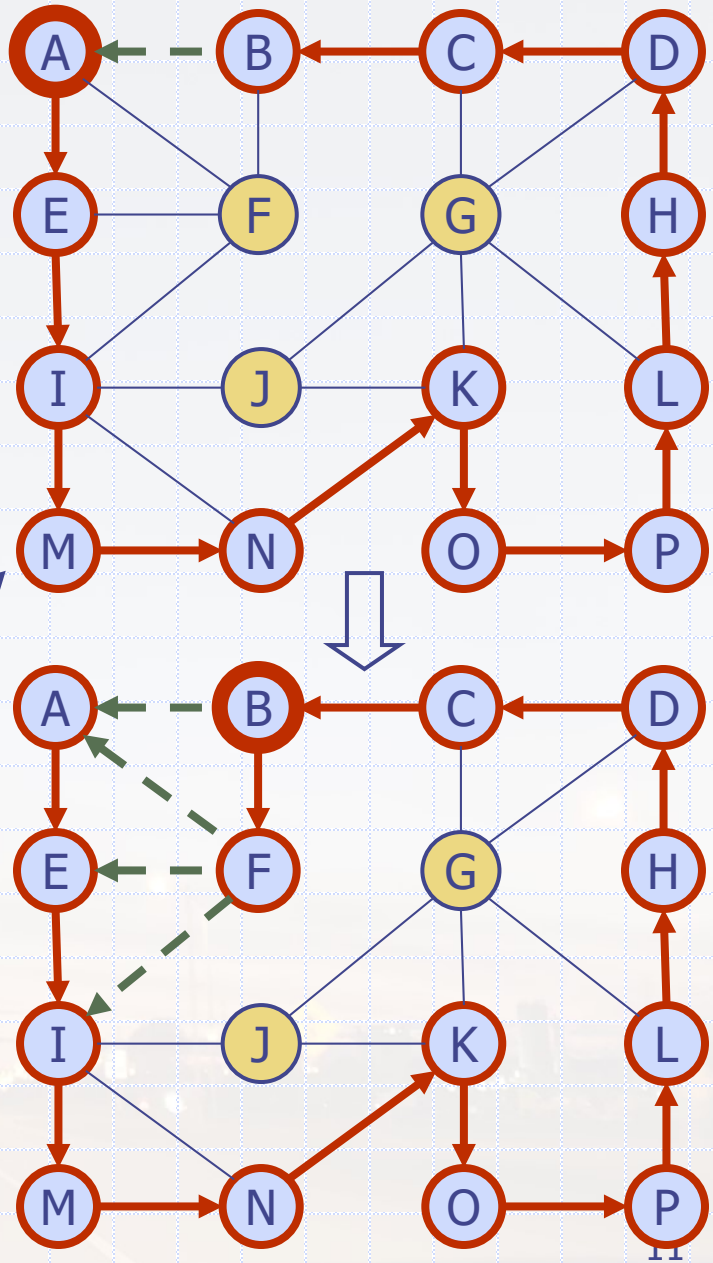
(Thicker one indicates where we have rolled back and were able to start exploring again)

unexplored edge

discovery edge

back edge





DFS Example2 (continues)

A unexplored vertex

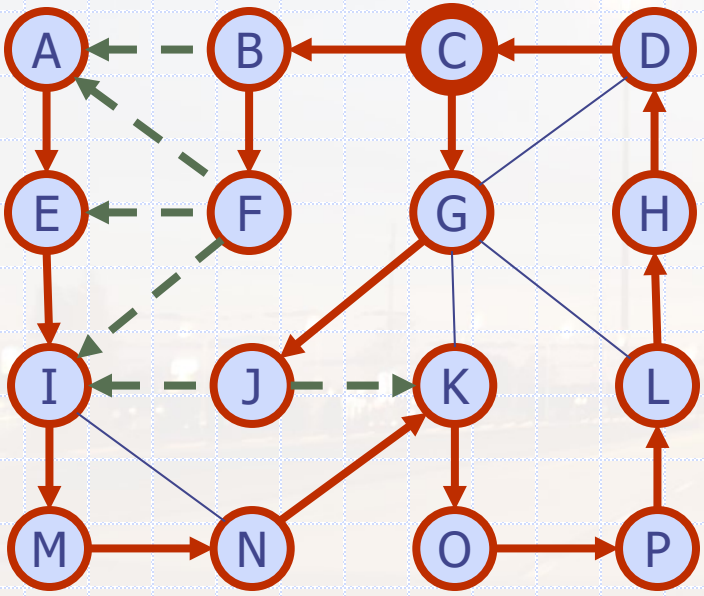
A visited vertex

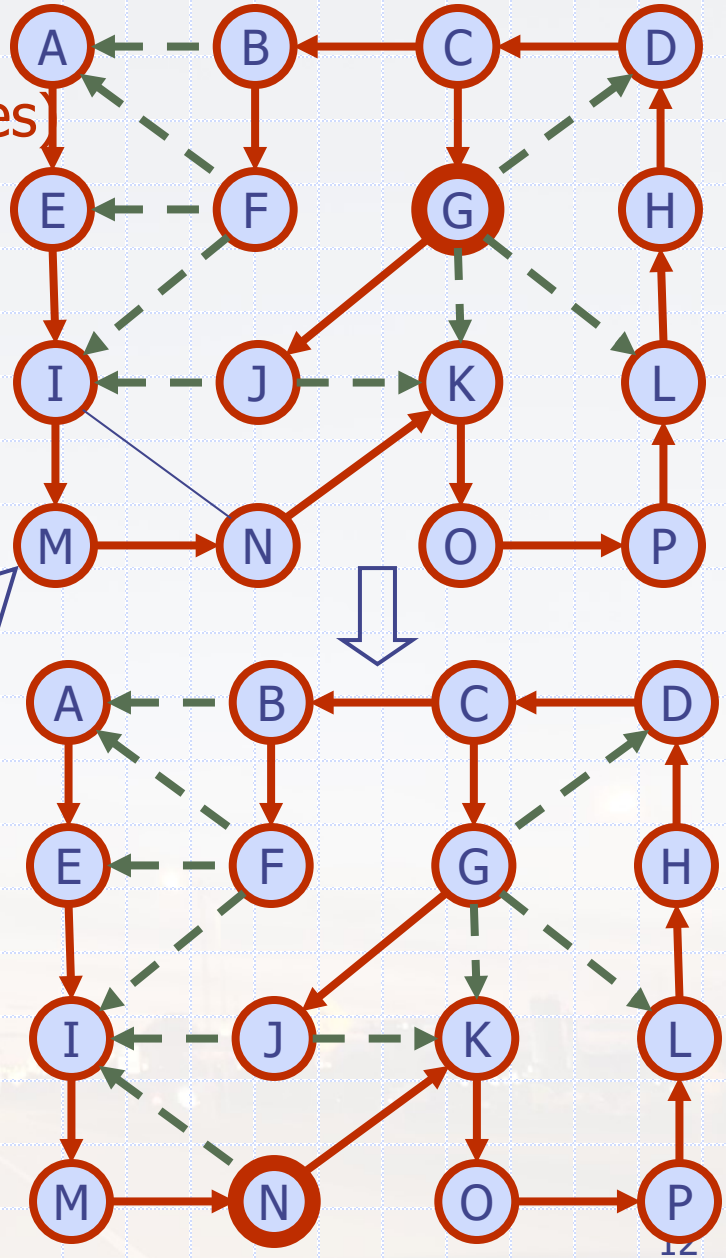
(Thicker one indicates where we have rolled back and were able to start exploring again)

unexplored edge

discovery edge

back edge





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()
        setLabel(u, UNEXPLORED)
    for all e ∈ G.edges()
        setLabel(e, UNEXPLORED)
    for all v ∈ G.vertices()
        if getLabel(v) = 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
    setLabel(v, VISITED)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w)
            else
                setLabel(e, BACK)
```

DFS Analysis

- Using a DFS traversal over a graph G, it is possible to solve the following problems:
  (note: this is actually done using algorithms and subroutines that are slightly different than the above simplified shown algorithm)
  - Visit all the vertices and edges of G
  - Determine whether G is connected
    ➔ start DFS and count reachable vertices then compare with total number of vertices in the graph
  - Compute spanning tree of G is G is connected
  - Compute the spanning forest of G (all spanning trees) if G is a non-connected graph
  - Compute the connected components of G
    ➔ Label each vertex with a connected component # then return the total number found
  - Find a path between any two given vertices, if exists.
  - Find a cycle of G, or report that G has no cycles
- **However, what is the complexity of DFS?**

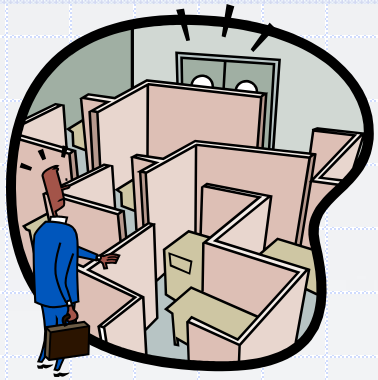
DFS Analysis

- DFS algorithm is called exactly once for every vertex
- Each edge is examined exactly twice (once from each of its two ends)
- Consequently, DFS on a graph with n vertices and m edges is $O(n + 2m) \rightarrow O(n + m)$
- It should be noted however that such analysis is assuming the existence of mechanisms, that can run in constant time, to:
  - Determine whether or not a vertex has been previously visited
  - Mark a vertex as "unexplored" or "visited" as appropriate
  - Determine whether or not an edge has been explored
  - Mark edges as "unexplored" , "discovery" or "back" as appropriate
    (Notice that the labeling of a vertex or an edge is conducted exactly twice (Each vertex is labeled twice: once as UNEXPLORED once as VISITED. Each edge is labeled twice: once as UNEXPLORED and once as DISCOVERY or BACK)
  - Mechanisms to find *incidentEdges()* and *opposite()*

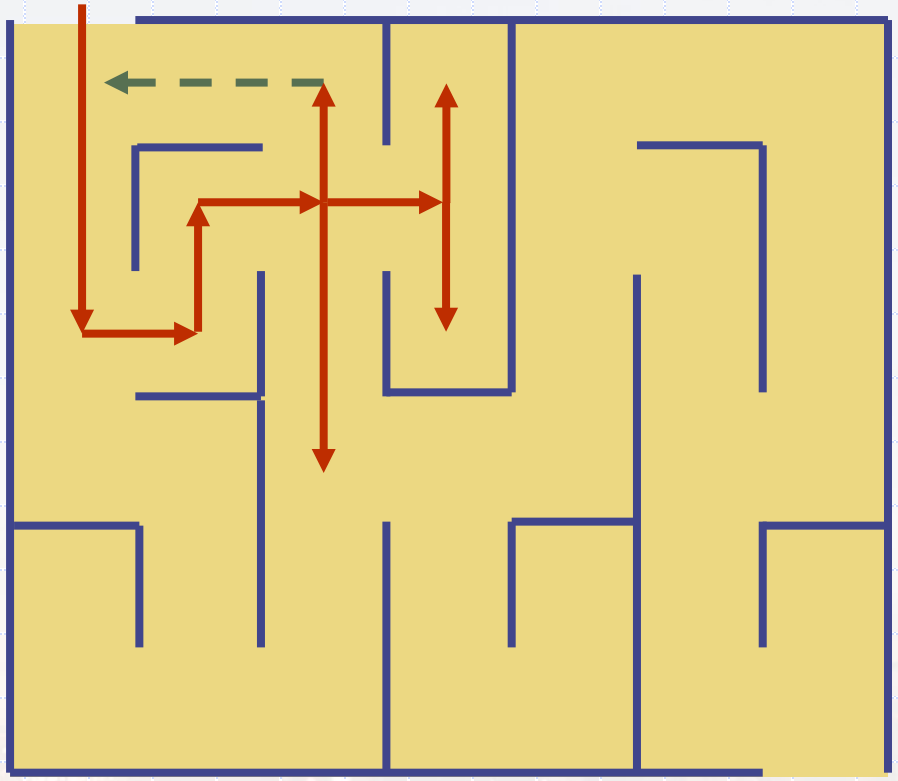
DFS Analysis

- These additional mechanisms will require additional space and may affect the running time of DFS
- Fortunately, such mechanisms can be designed
- For instance, if the graph is represented by a adjacency list data structure then *incidentEdges()* of a vertex v can be achieved in $O(deg(v))$ and *opposite()* can be achieved in $O(1)$
- Marking and testing vertices and edges can also be done in constant time by adding further attributes to these objects (this is possibly not best) or using concepts such as the *decorator* pattern to add further (temporary/scratch) attributes to existing objects
- Consequently, DFS can indeed achieve a running time of $O(n + m)$ which is efficient

DFS and Maze Traversal



- The DFS algorithm 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

Proof:

- Assume there is at least one vertex v that is unvisited. Assume that w is the first unvisited vertex in the path from s to v (v maybe = w)
- Since w is the first unvisited vertex in the path, then it has a neighbor u in the path that was visited
- However, if u is visited then the edge (u, w) must have been considered; hence w must have visited, which contradicts the original assumption that w is unvisited

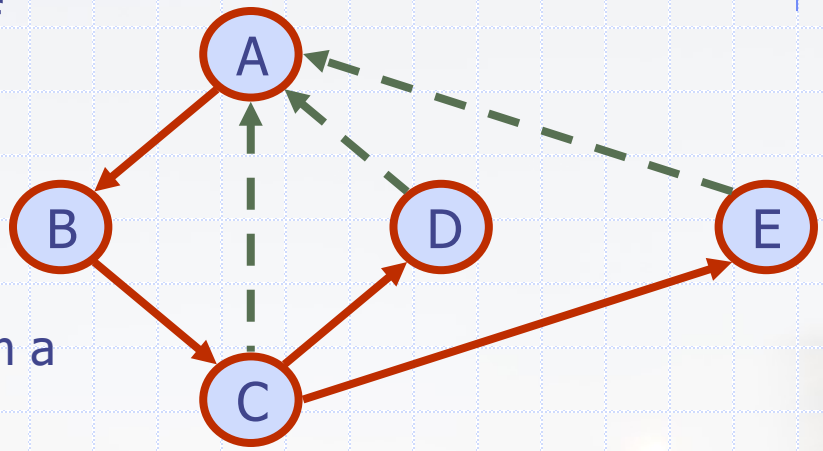
Properties of DFS

Property 2

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

Proof:

- Since DFS marks edges as discovery when it reaches an unvisited vertex, it will never form a cycle with these discovery edges
- Hence discovery edges form a tree
- Further, this tree is a spanning tree since (from Property 1), DFS visits each vertex in the connected component of s.



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