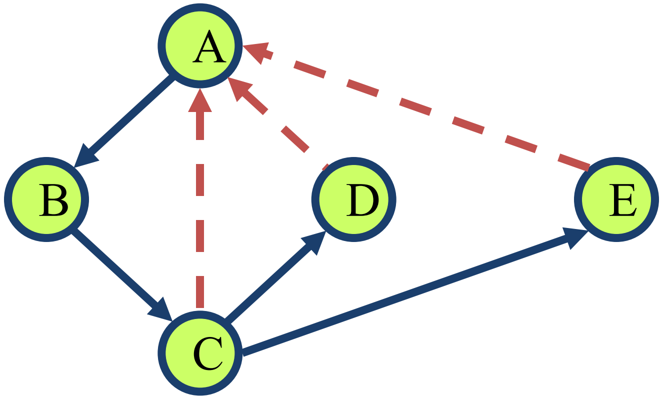
# Depth-First Search



# Outline and Reading

Definitions (6.1)

- Subgraph
- Connectivity
- Spanning trees and forests

Depth-first search (6.3.1)

- Algorithm
- Example
- Properties
- Analysis

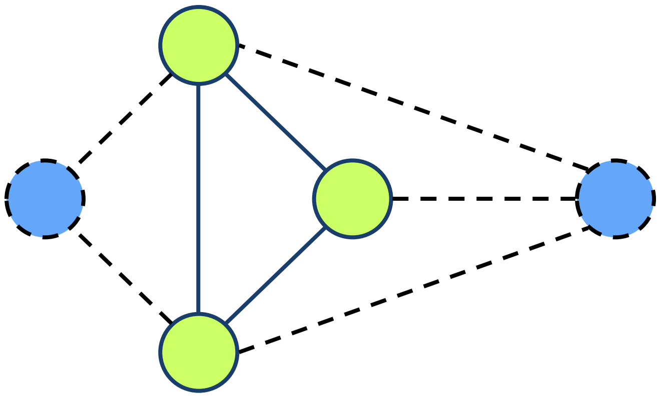
Applications of DFS (6.5)

- Path finding
- Cycle finding

# Subgraphs

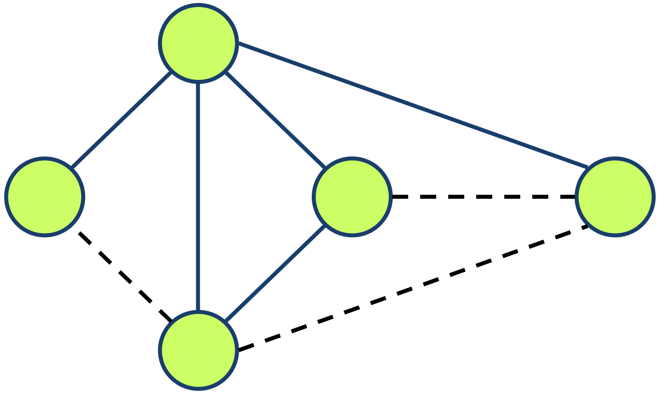
A subgraph S of a graph G is a graph such that

- the vertices of S are a subset of the vertices of G
- the edges of S are a subset of the edges of G



Subgraph

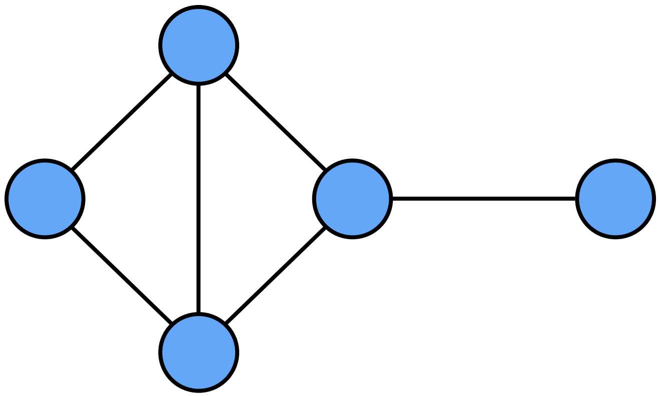
A spanning subgraph of G is a subgraph that contains all the vertices of G



Spanning subgraph

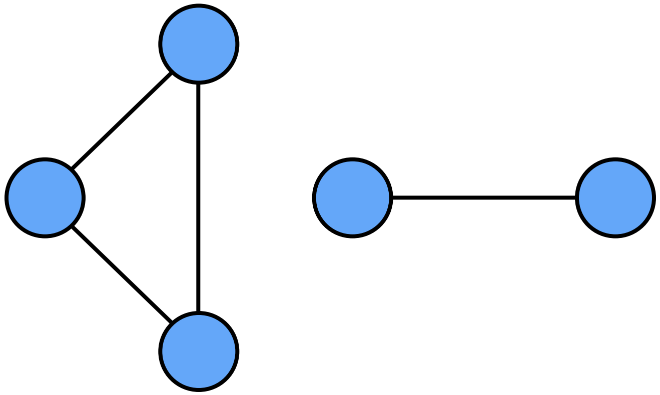
# Connectivity

A graph is connected if there is a path between every pair of vertices



Connected graph

A connected component of a graph G is a maximal connected subgraph of G

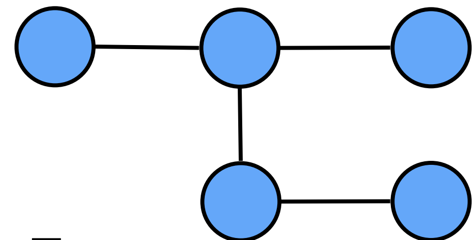


Non connected graph with two connected components

# Trees and Forests

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

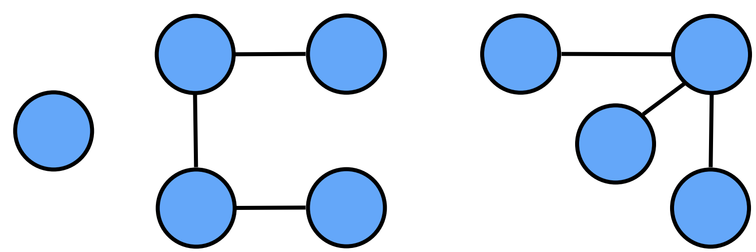
  This definition of tree is different from the one of a rooted tree



Tree

- A forest is an undirected graph without cycles

- The connected components of a forest are trees

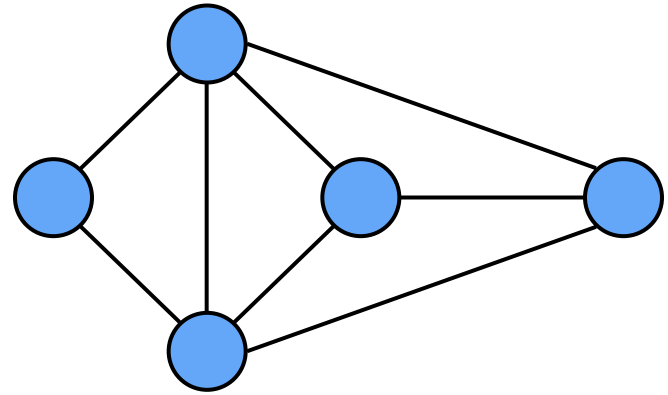


Forest

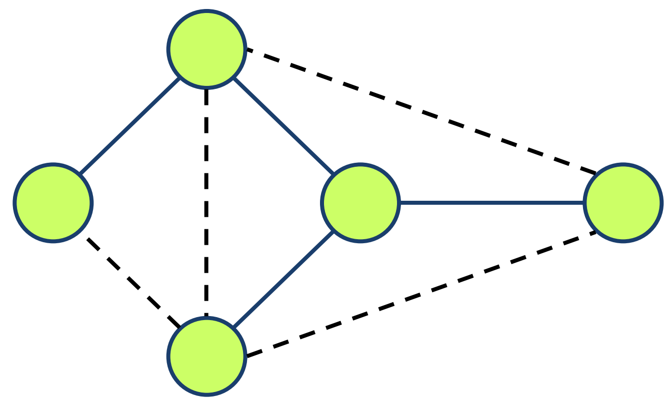
# Spanning Trees and Forests

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

# Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph. A DFS traversal of a 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
- DFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - find and report a path between two given vertices
  - find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

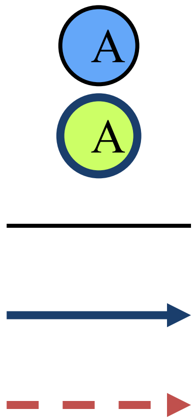
# DFS Algorithm

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 ← G.opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

# Example



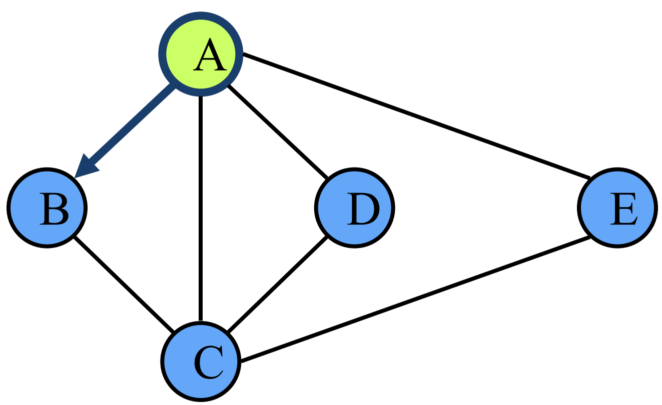

unexplored vertex

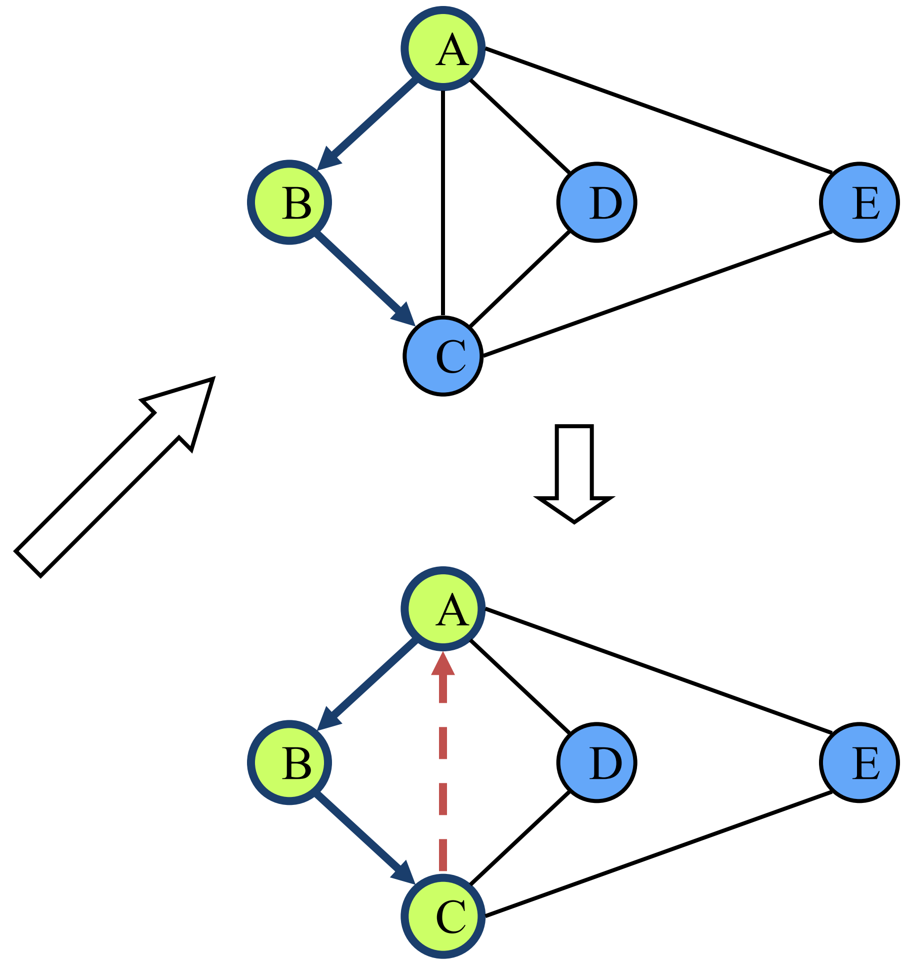
visited vertex

unexplored edge

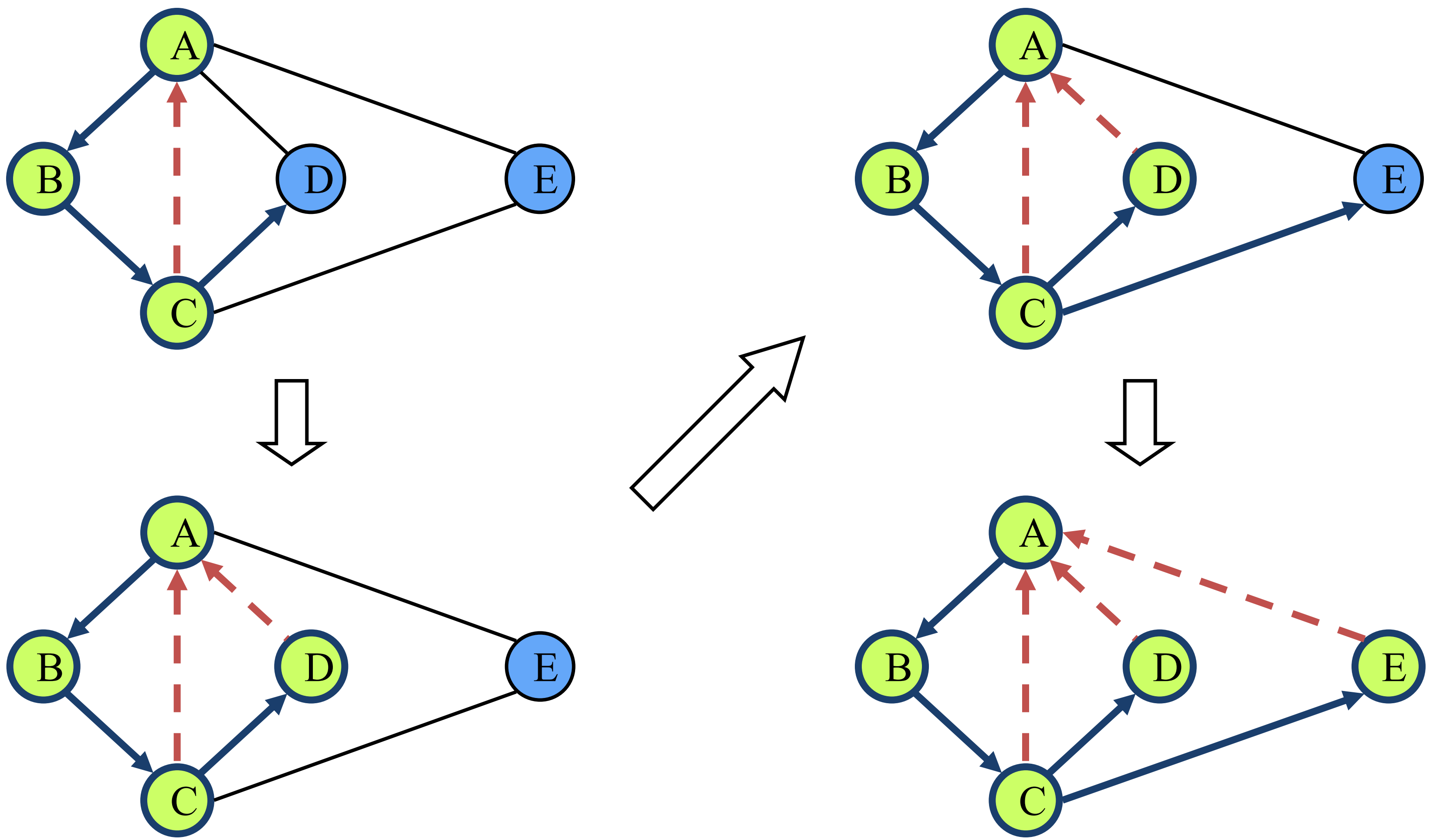
discovery edge

back edge

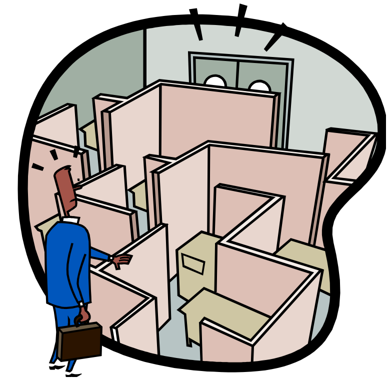


# Example (cont.)

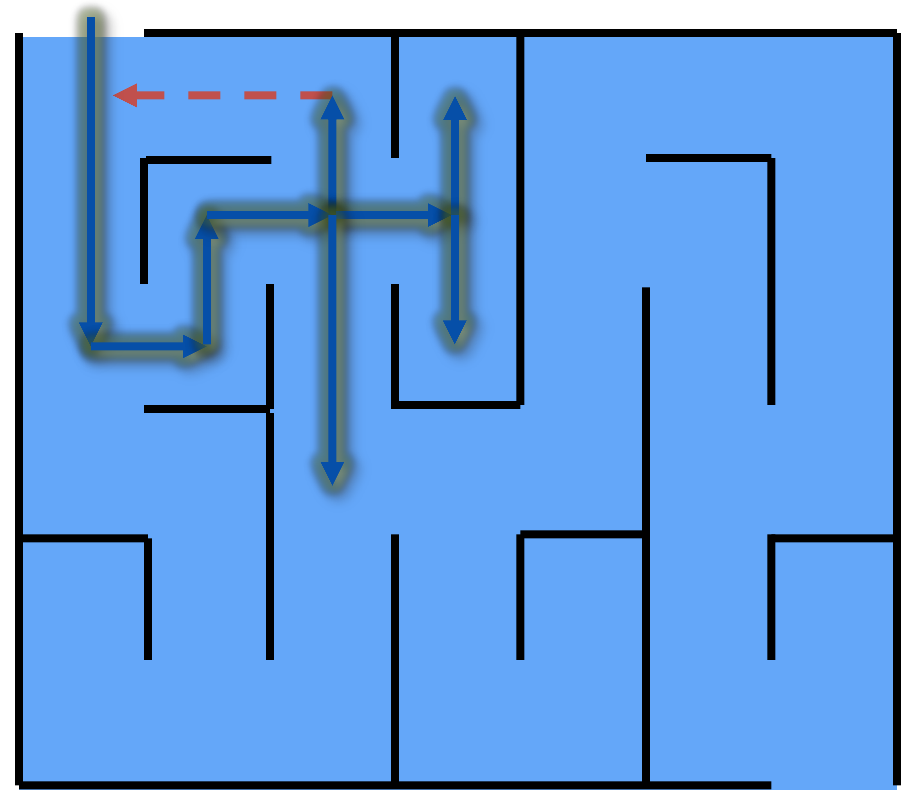


# DFS and Maze Traversal



The DFS algorithm is similar to a classic strategy for exploring a maze

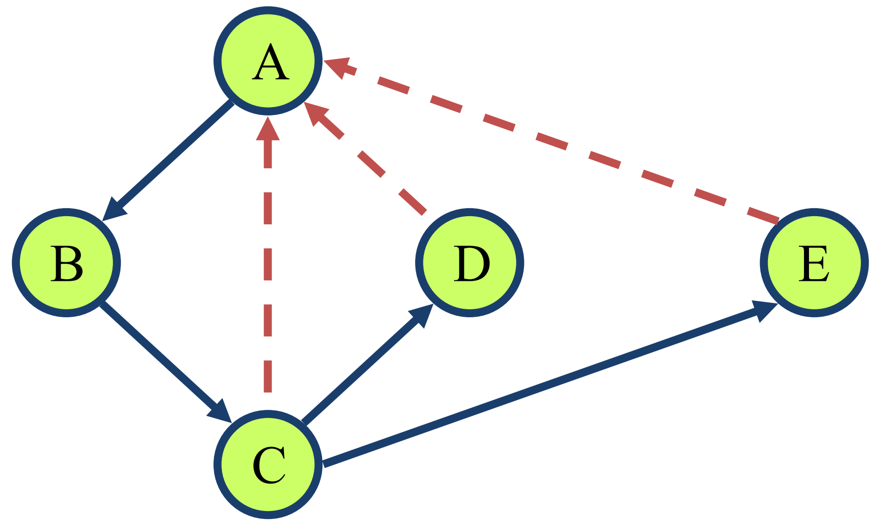
- We mark each intersection, corner and dead end (vertex) visited
- We mark each corridor (edge ) traversed
- We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



# Properties of DFS

Property 1

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




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)
  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 gets popped }
      else
        setLabel(e, BACK)
  S.pop()              { v gets popped }
```

# 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)
  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)
        S.pop()
      else
        C ← new empty stack
        repeat
          o ← S.pop()
          C.push(o)
        until o = w
        return C.elements()
  S.pop()
```