Data Structures & Algorithms 2

Topic 7 – Graphs (part 2)

Lecturer: Dr. Hadi Tabatabaee
Materials: Dr. Phil Maguire & Dr. Hadi Tabatabaee
Maynooth University
Online at http://moodle.maynoothuniversity.ie

Overview

Aims

- Introducing graph search approaches

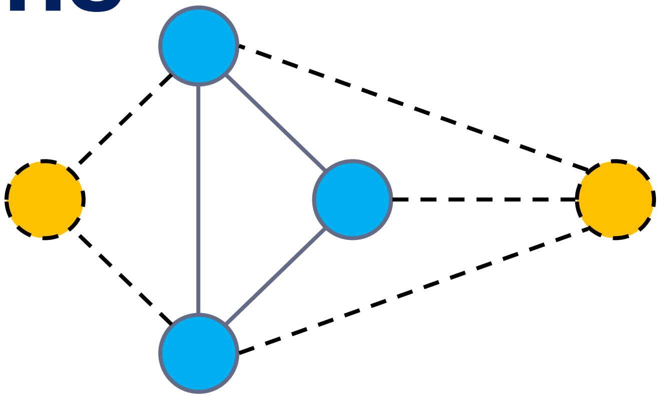
Learning outcomes: You should be able to...

- Learn different graph search methods
- Learn to implement a depth-first search method

Subgraphs

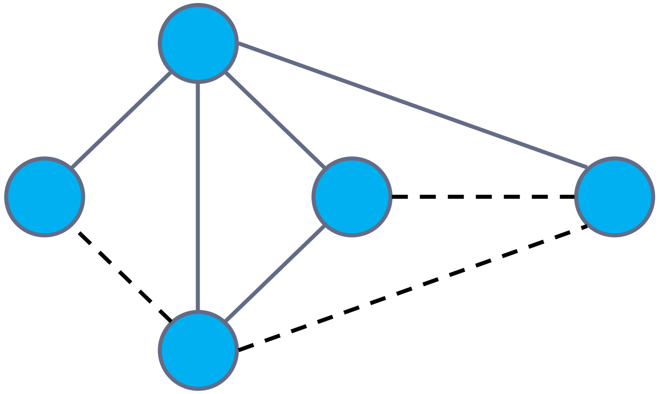
A subgraph S of a graph G is a graph such that:

- 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

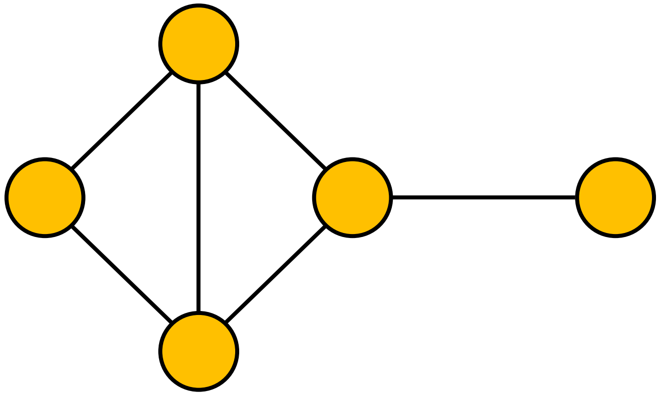
A spanning subgraph of G is a subgraph that contains all the vertices of G.



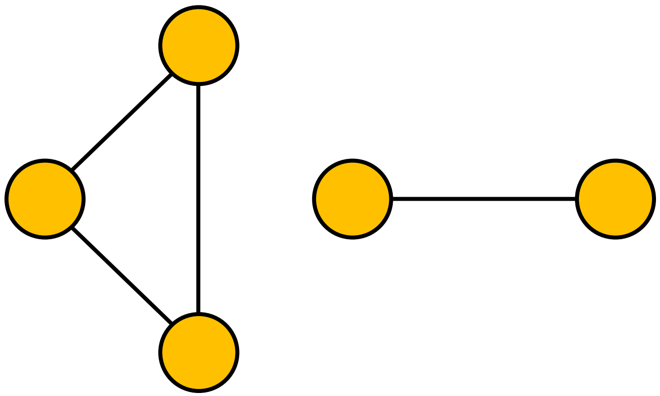
Spanning subgraph

Connectivity

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



Connected graph

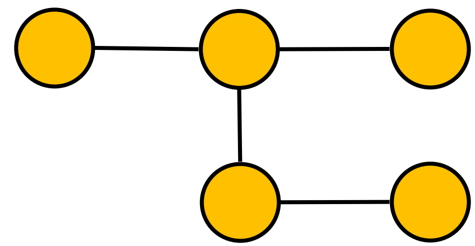


Non connected graph with two connected components

Trees and Forests

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

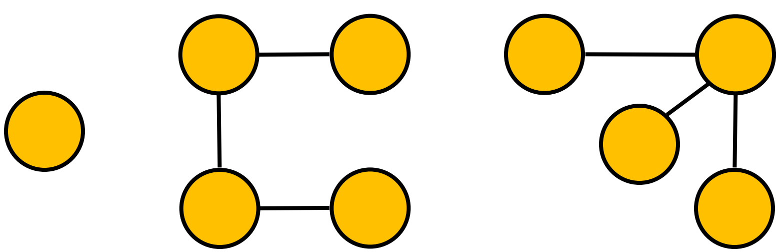
  This definition of a tree is different from the one of a rooted tree



Tree

- A forest is an undirected graph without cycles.

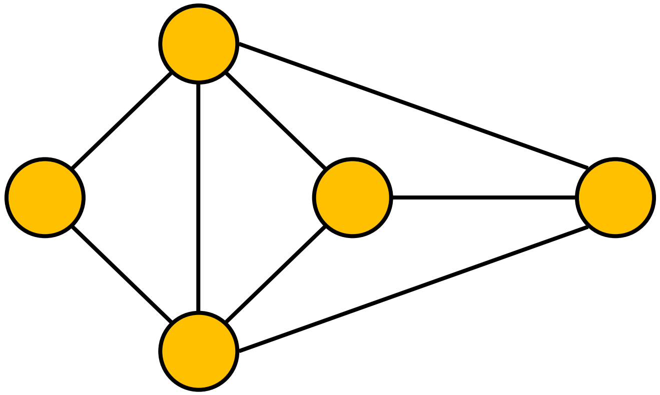
- The connected components of a forest are trees.



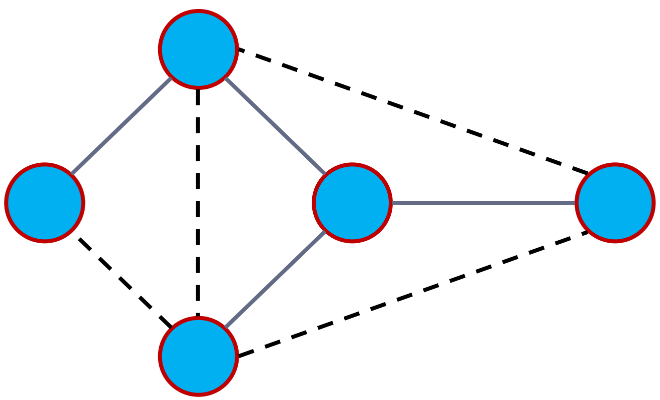
Forest

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 in the design of communication networks.
- A spanning forest of a graph is a spanning subgraph that is a forest.



Graph



Spanning tree

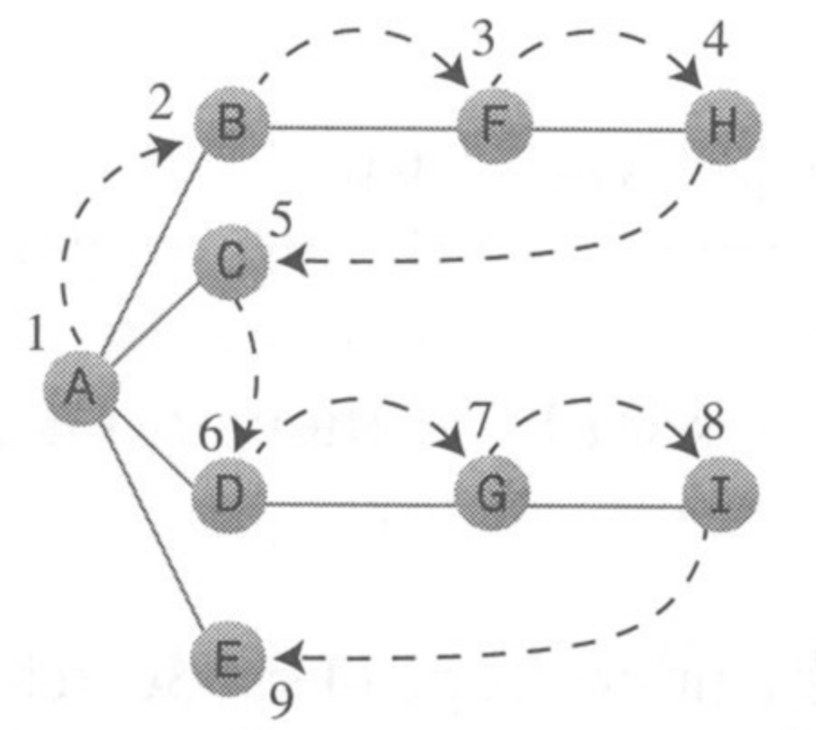
Searching

- One of the most useful operations you can perform on your graph is searching.
- For example, find all the towns that can be visited by rail, leaving Dublin.
- There are two very well-known approaches for searching a space
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
- Depth-first search explores one possibility as far as it can and then backtracks when it meets a dead end.
- Breadth-first search explores all possibilities of the same depth at the same time and spreads itself equally.

Depth-First search

- The depth-first search uses a stack to remember where it should go when it reaches a dead end.
- Pick a starting point, then push this vertex onto the stack and mark it so that you won't revisit it.
- Next, visit any adjacent vertex and start doing the same thing.
- If there are no unvisited adjacent vertices, then just pop a vertex off the stack and try again (backtrack).

Depth-First search



Analogy

- Depth-first search is like the "ball and wool" approach of finding the exit to a maze.
- Any time you come up against a dead-end, you backtrack and mark the path you've been on, so you won't try it again
- Eventually, you will have explored every possible path without retracing your steps.
- Depth-first search is used when you are trying to solve a problem (like finding the exit to a maze!)

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 from a Vertex

Algorithm DFS(G, u):

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

Mark vertex u as visited.

for each of u's outgoing edges, $e = (u, v)$ **do**

  if vertex v has not been visited **then**

    Record edge e as the discovery edge for vertex v.

    Recursively call DFS(G, v).

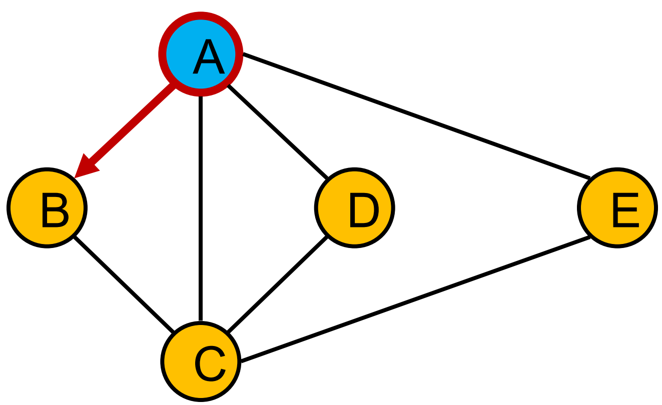
Java Implementation

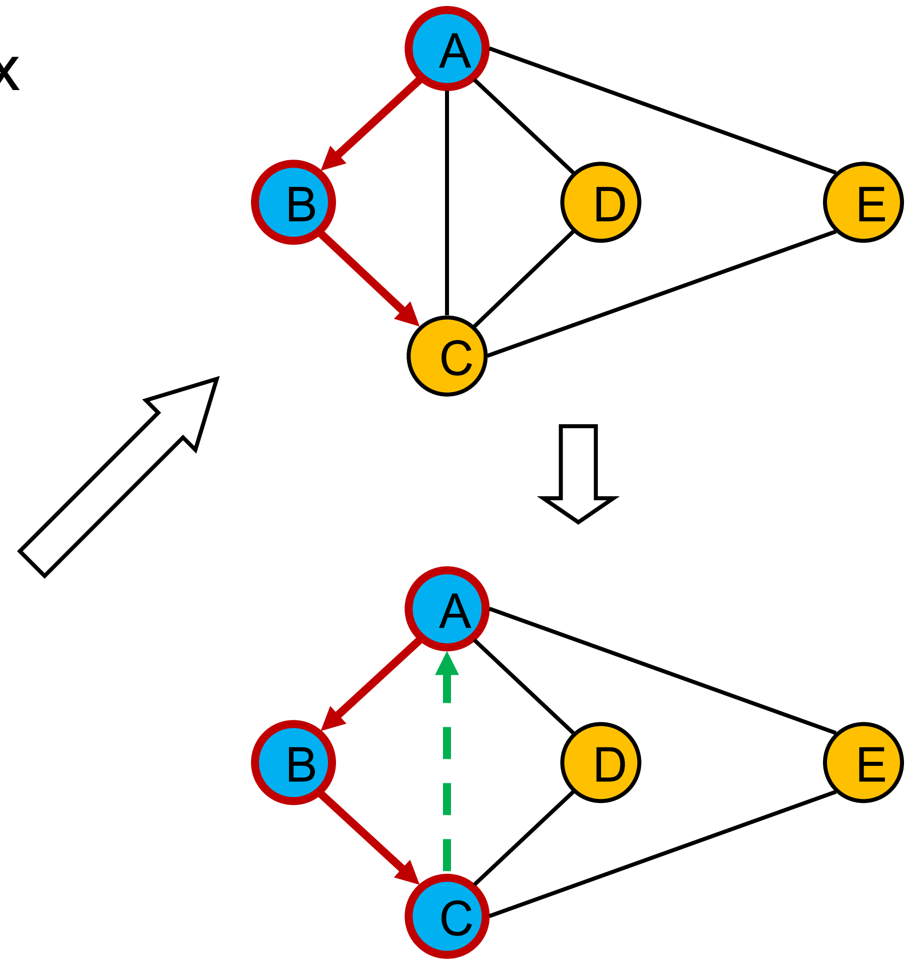
First, we maintain a set, named known, containing vertices that have already been visited. Second, we keep a map, named forest, that associates, with a vertex *v*, the edge *e* of the graph that is used to discover *v* (if any).

```
/** Performs depth-first search of Graph g starting at Vertex u. */
public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
                Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  known.add(u);                              // u has been discovered
  for (Edge<E> e : g.outgoingEdges(u)) {     // for every outgoing edge from u
    Vertex<V> v = g.opposite(u, e);
    if (!known.contains(v)) {
      forest.put(v, e);                      // e is the tree edge that discovered v
      DFS(g, v, known, forest);              // recursively explore from v
    }
  }
}
```

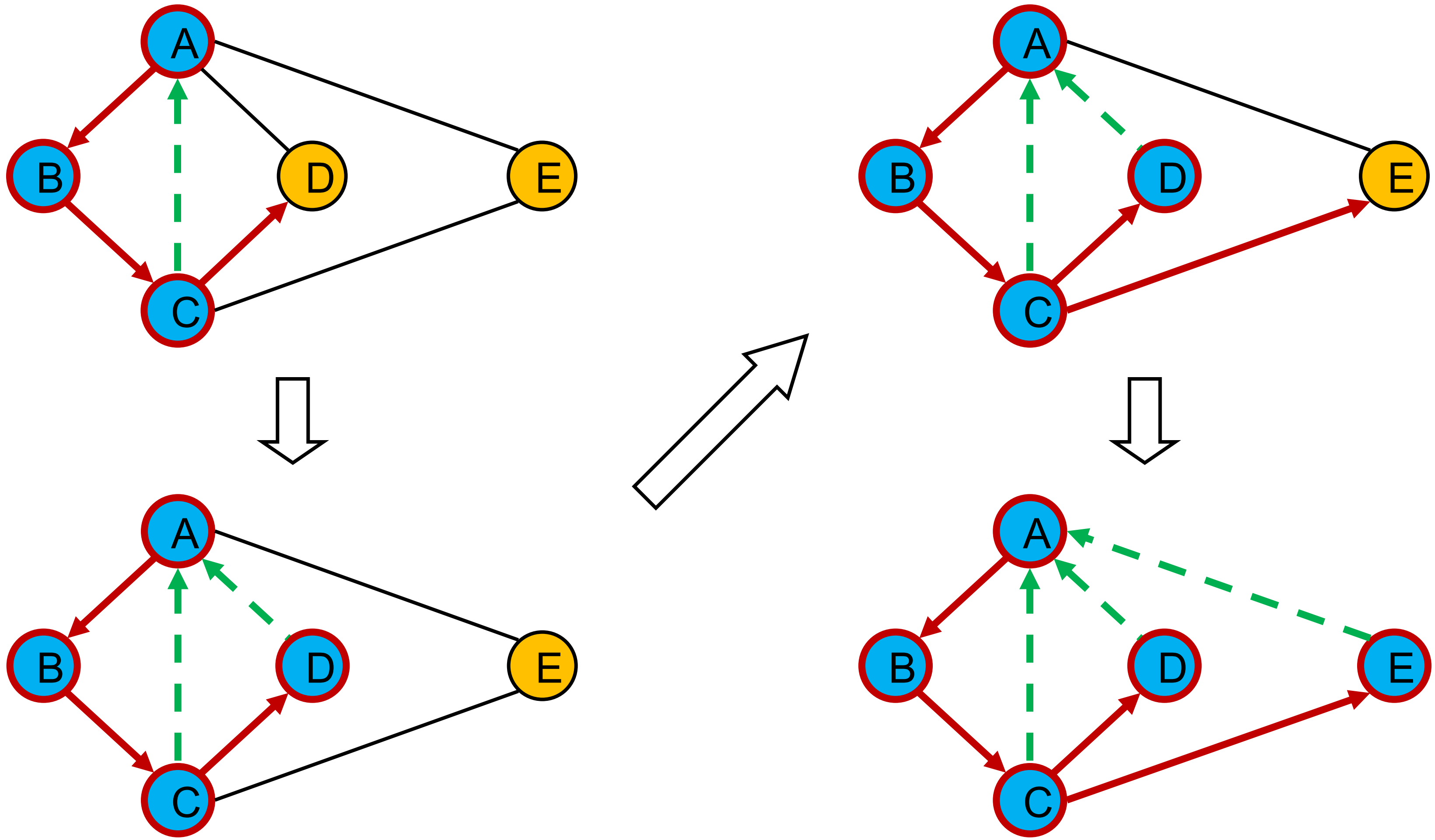
Example

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

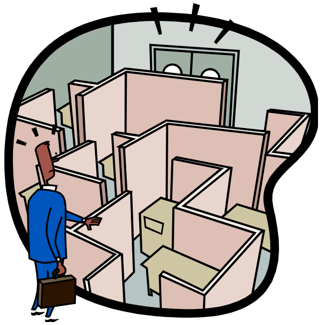




Example (cont.)

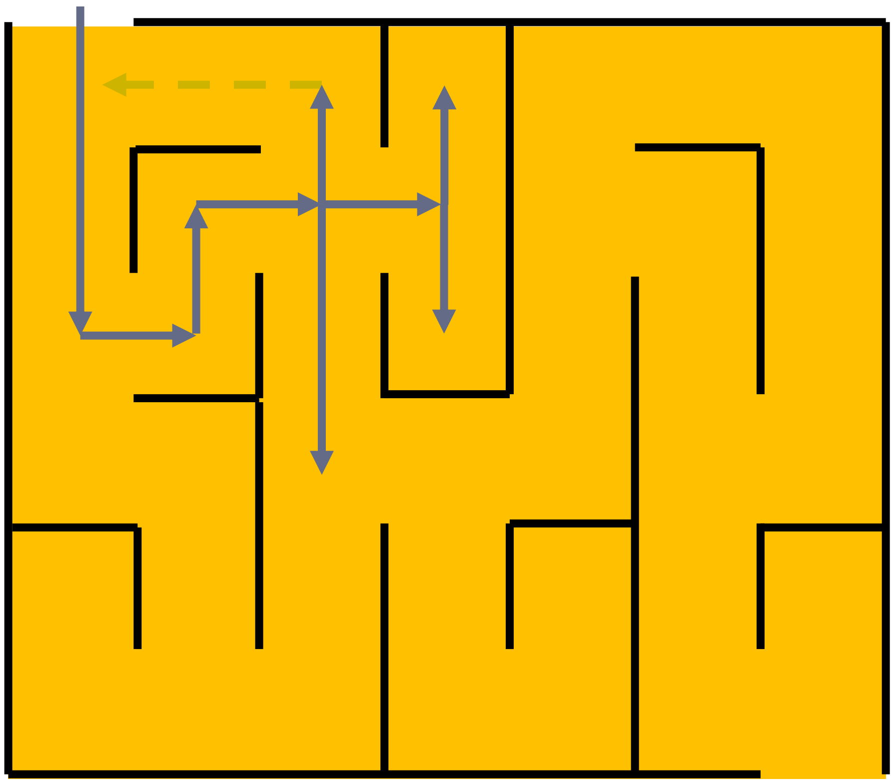


DFS and Maze Traversal



The DFS algorithm is like 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) using a rope (recursion stack).



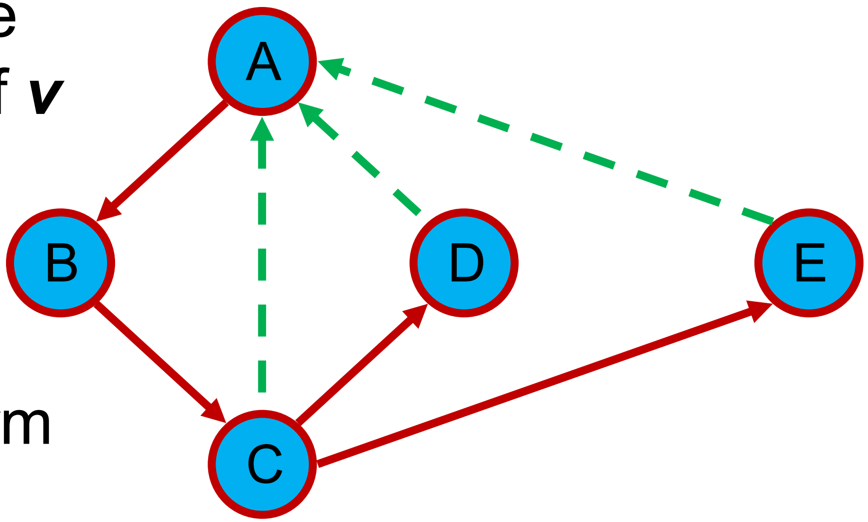
Properties of DFS

Property 1

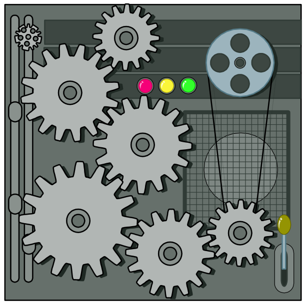
DFS(***G*, *v***) visits all the vertices and edges in the connected component of ***v***

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 $\sum_v \deg(v) = 2m$

Implementing DFS

- The key to DFS is to be able to find the vertices that are unvisited and adjacent to a specified vertex
- Consult the adjacency matrix for this
- Go to the row for the specified vertex and you can pick out all the columns containing a one
- Check these adjacent vertices to see if they have been visited or not
- Use a stack structure to track the path you've followed so far and flag the vertices that have already been visited so you can avoid them
- Pop off the stack to backtrack

Path Finding

- Using the template method pattern, we can specialize the DFS algorithm to find a path between two given vertices u and z.
- We call ***pathDFS***(***G, v,z***) with ***v*** 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)
```

Path Finding in Java

To reconstruct the path, we begin at the *end* of the path, examining the forest of discovery edges to determine what edge was used to reach vertex *v*. We then determine the opposite vertex of that edge and repeat the process to determine what edge was used to discover it. By continuing this process until reaching *u*, we can construct the entire path.

```
/** Returns an ordered list of edges comprising the directed path from u to v. */
public static <V,E> PositionalList<Edge<E>>
constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
               Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Edge<E>> path = new LinkedPositionalList<>();
  if (forest.get(v) != null) {              // v was discovered during the search
    Vertex<V> walk = v;                     // we construct the path from back to front
    while (walk != u) {
      Edge<E> edge = forest.get(walk);
      path.addFirst(edge);                  // add edge to *front* of path
      walk = g.opposite(walk, edge);        // repeat with opposite endpoint
    }
  }
  return path;
}
```

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)
        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)
```

DFS for an Entire Graph

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 $u \in$ *G.vertices*()
    setLabel(*u, UNEXPLORED*)
  for all $e \in$ *G.edges*()
    setLabel(*e, UNEXPLORED*)
  for all $v \in$ *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 $e \in$ *G.incidentEdges*(*v*)
    if *getLabel*(*e*) = *UNEXPLORED*
      $w \leftarrow$ *opposite*(*v,e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e, DISCOVERY*)
        DFS(*G, w*)
      else
        setLabel(*e, BACK*)

All Connected Components

- When a graph is not connected, the next goal we may have is to identify all of the ***connected components*** of an undirected graph.
- If an initial call to DFS fails to reach all vertices of a graph, we can restart a new call to DFS at one of those unvisited vertices.
- It returns a map that represents a ***DFS forest*** for the entire graph. We say this is a forest rather than a tree, because the graph may not be connected.

```
/** Performs DFS for the entire graph and returns the DFS forest as a map. */
public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
  Set<Vertex<V>> known = new HashSet<>();
  Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
  for (Vertex<V> u : g.vertices())
    if (!known.contains(u))
      DFS(g, u, known, forest);                    // (re)start the DFS process at u
  return forest;
}
```

Loop over all vertices, doing a DFS from each unvisited one.

Questions