CSE 250: Graphs and Depth-First Search

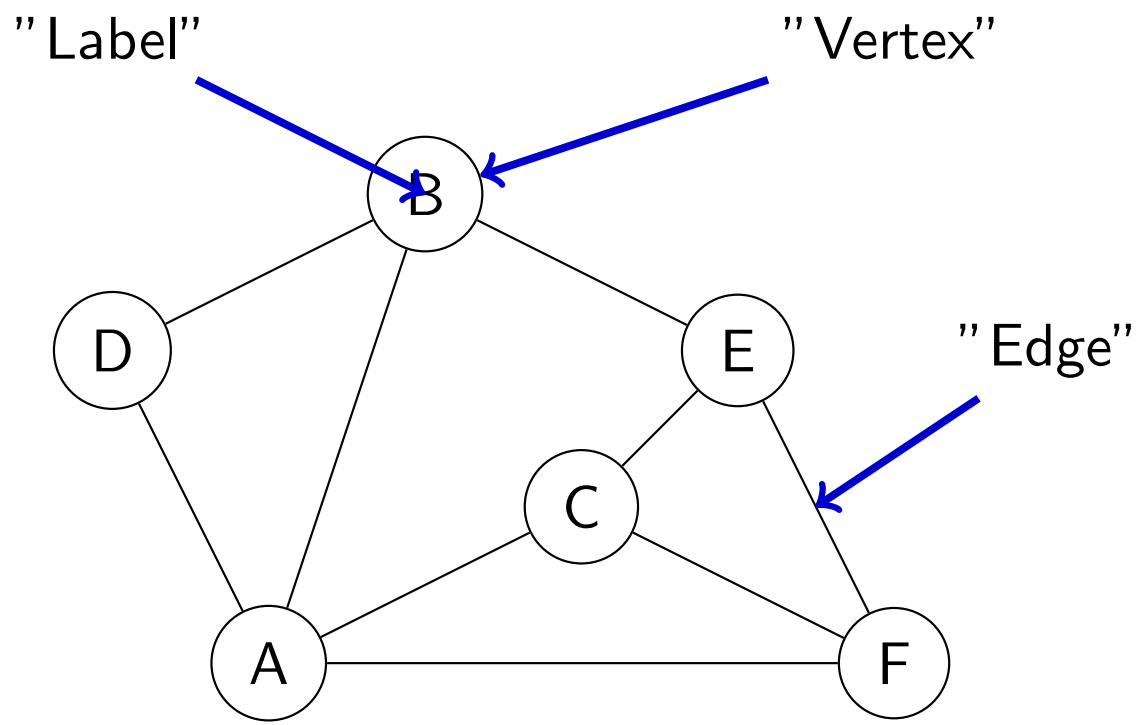
Lecture 19

Oct 16, 2023

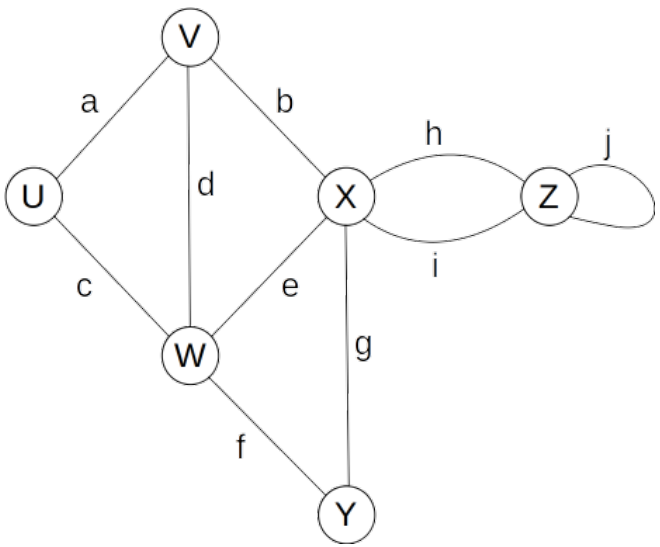
Reminders

- PA2 released soon: Implement **Map Routing**
  1. Create an adjacency list (discussed today)
  2. Find a path from A to B with the fewest intersections
  3. Find a path from A to B with the shortest distance
- PA2 test cases due Sun, Oct 22 at 11:59 PM
- PA2 implementation due Sun, Nov 5 at 11:59 PM

Graphs



Graph Terminology



- **Endpoints of an edge**
  U, V are the endpoints of a.
- **Edges incident on a vertex**
  a, b, d are incident on V.
- **Adjacent Vertices**
  U, V are adjacent.
- **Degree of a vertex (# of incident edges)**
  X has degree 5.
- **Parallel Edges** (same endpoints)
  h, i are parallel.
- **Self-loop** (same vertex is start and end)
  j is a self-loop.
- **Simple Graph**
  A graph with no parallel edges or self-loops.

Notation

- N: The number of vertices
- M: The number of edges
- $\deg(v)$: The degree of a vertex

The Directed Graph ADT

```
1  interface Graph<V, E> {
2    public Iterator<Vertex> vertices()
3    public Iterator<Edge> edges()
4    public Vertex addVertex(V label): Vertex
5    public Edge addEdge(Vertex orig, Vertex dest, E label): Edge
6    public void removeVertex(Vertex vertex)
7    public void removeEdge(Edge edge)
8  }
```

The Directed Graph ADT

```
1  interface Vertex<V, E> {
2    public V getLabel();
3    public Iterator<Edge> getOutEdges();
4    public Iterator<Edge> getInEdges();
5    public Iterator<Edge> getIncidentEdges();
6    public boolean hasEdgeTo(Vertex v);
7  }
```

The Directed Graph ADT

```
1  interface Edge<V, E> {
2    public E getLabel();
3    public Vertex getOrigin();
4    public Vertex getDestination();
5  }
```

Graph ADT Guarantees

What guarantees does the ADT provide?

- The origin and destination of each edge are in the graph.

Edge List

```
1  class EdgeListV2<V, E> implements Graph<V, E>
2  {
3    List<Vertex> vertices = new BetterLinkedList<Vertex>();
4    List<Edge>   edges    = new BetterLinkedList<Edge>();
5
6    /*...*/
7  }
```

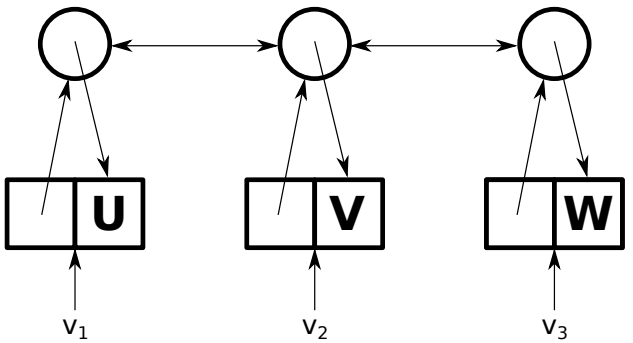
Edge List

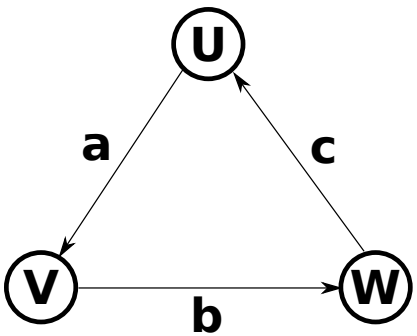
```
1   class BetterLinkedList<T> implements List<T>
2   {
3     public Node<T> add(T element);
4       /* O(1) with tail pointer */
5
6     public void remove(Node<T> node)
7       /* O(1) */
8
9     public ListIterator<T> iterator
10      /* O(1) + O(1) per call to next */
11
12    /*...*/
13  }
```

Edge List Summary

LinkedList[Vertex]

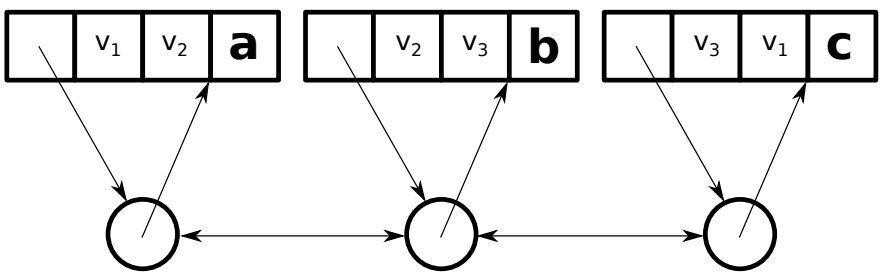
Vertex





Edge

LinkedList[Edge]



Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

Space Used: $O(N + M)$
(constant space per vertex, edge)

incidentEdges

Problem: Finding incident edges is $O(M)$

Improving on the Edge List

How can we avoid searching every edge in the edge list to find the incident edges?

Idea: Store each edges in/out edge list.

Adjacency List

```
1  public class Vertex<V, E>
2  {
3    Node<Vertex> node = null;
4    List<Edge> inEdges = new BetterLinkedList<Edge>();
5    List<Edge> outEdges = new BetterLinkedList<Edge>();
6    /*...*/
7  }
```

addEdge

```
1   public Edge addEdge(Vertex origin, Vertex dest, E label)
2   {
3     assert(origin.node != null);
4     assert(dest.node != null);
5     Edge edge = new Edge(label);
6     Node<Edge> node = edges.append(edge);
7     edge.node = node;
8     origin.outEdges.add(edge);
9     dest.inEdges.add(edge);
10    return edge;
11  }
```

What's the complexity? $O(1)$

removeEdge

```
1   public void removeEdge(Edge edge)
2   {
3     edges.remove(edge.node);
4     for(iter : edge.orig.outEdges)
5     {
6       if(iter.next().equals(edge)){ iter.remove(); }
7     }
8     for(iter : edge.dest.inEdges)
9     {
10      if(iter.next().equals(edge)){ iter.remove(); }
11    }
12    edge.node = null;
13  }
```

What's the complexity? $O(M)$

removeEdge

Idea: Store the outEdge/inEdge node with the Edge.

```
1  public class Edge<V, E>
2  {
3    Node<Edge> node = null;
4    Node<Edge> inNode = null;
5    Node<Edge> outNode = null;
6    /*...*/
7  }
```

addEdge

```
1   public Edge addEdge(Vertex origin, Vertex dest, E label)
2   {
3     assert(origin.node != null);
4     assert(dest.node != null);
5     Edge edge = new Edge(label);
6     Node<Edge> node = edges.append(edge);
7     edge.node = node;
8     edge.outNode = origin.outEdges.add(edge);
9     edge.inNode = dest.inEdges.add(edge);
10    return edge;
11  }
```

What's the complexity? $O(1)$

removeEdge

```
1  public void removeEdge(Edge edge)
2  {
3    edges.remove(edge.node);
4    edge.orig.outEdges.remove(edge.outNode);
5    edge.dest.inEdges.remove(edge.inNode);
6    edge.node = null;
7    edge.inNode = null;
8    edge.outNode = null;
9  }
```

What's the complexity? $O(1)$

removeVertex

```
1   public void removeVertex(Vertex vertex)
2   {
3     for(edge : vertex.inEdges)
4     {
5       removeEdge(edge.node)
6     }
7     for(edge : vertex.outEdges)
8     {
9       removeEdge(edge.node)
10    }
11    vertices.remove(vertex.node);
12    vertex.node = null;
13  }
```

What's the complexity? $O(\texttt{deg}(\texttt{vertex}))$

Adjacency List Summary

Starting with an edge list:

- Store a linked list of in/out edges with each vertex
- Store the linked list node for the in/out lists with each edge

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\texttt{deg}(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`: $O(\texttt{deg}(v))$

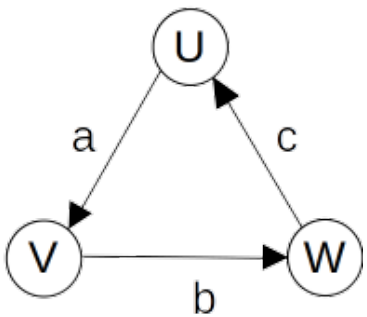
Space Used: $O(N + M)$
(constant space per vertex, edge)

hasEdgeTo

Can we cut `hasEdgeTo` down to $O(1)$?

The Adjacency Matrix Data Structure

Origin \ Destination	U	V	W
U	-	a	-
V	-	-	b
W	c	-	-



Edge List Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used: $O(N^2)$

Connectivity Problems

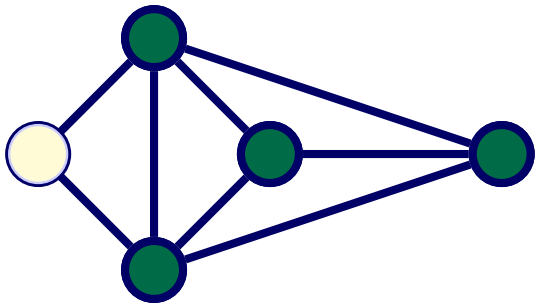
Back to things to do with graphs...

- Is vertex U **adjacent** to vertex V?
- Is vertex U **connected** to vertex V via some path?
- Which vertices are connected to vertex V?
- What is the shortest path between vertices U and V?

A few more definitions...

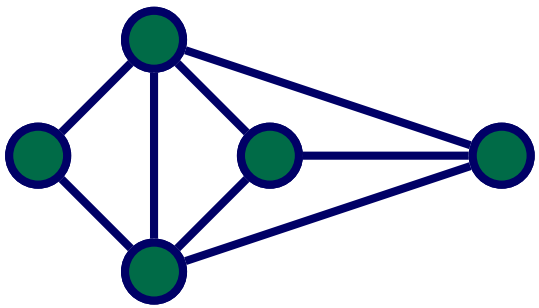
A **subgraph** S of a graph G is a graph where ...

- S's vertices are a subset of G's vertices.
- S's edges are a subset of G's edges.



A **spanning subgraph** S of a graph G is a graph where ...

- S is a subgraph of G
- S contains all of G's vertices.



A few more definitions

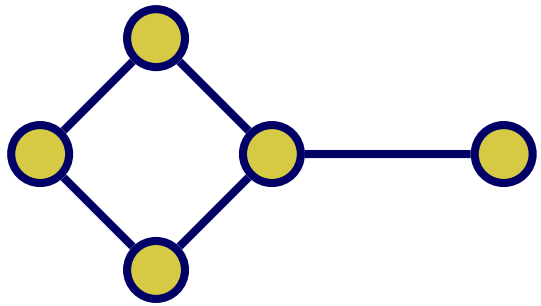
A graph is **connected** if...

- ...there is a path between every pair of vertices.

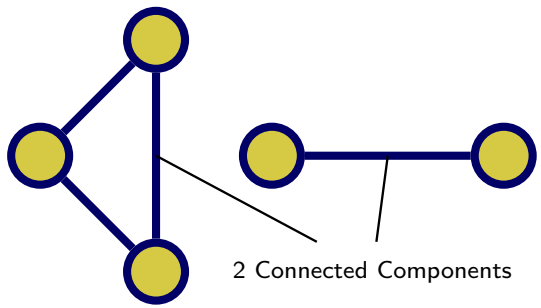
A **connected component** of G is a *maximal*, *connected* subgraph of G

- "*maximal*" means that adding any other vertices from G would break the connected property.
- Any subset of G's edges that makes the subgraph connected is fine.

Connected Graph



Disconnected Graph



A few more definitions...

A **free tree** is an *un*directed graph T such that:

- There is *exactly* one simple path between any two nodes
  - T is connected.
  - T has no cycles.

A **rooted tree** is a directed graph T such that:

- One vertex of T is the **root**.
- There is exactly one simple path from the root to every other vertex in T.

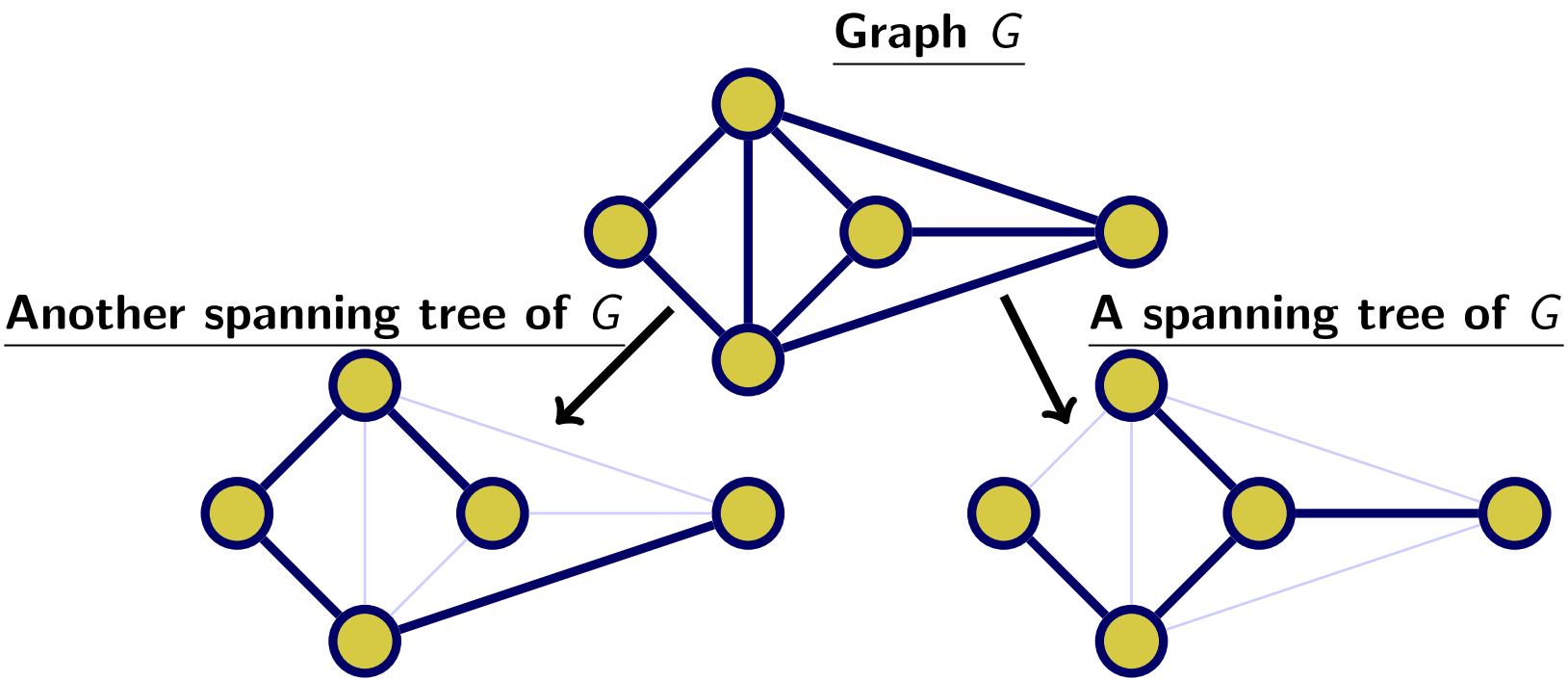
A **free/rooted forest** is a graph F such that:

- Every connected component is a tree.

A few more definitions...

A **spanning tree** of a connected graph G is:

- A spanning subgraph of G
- A tree

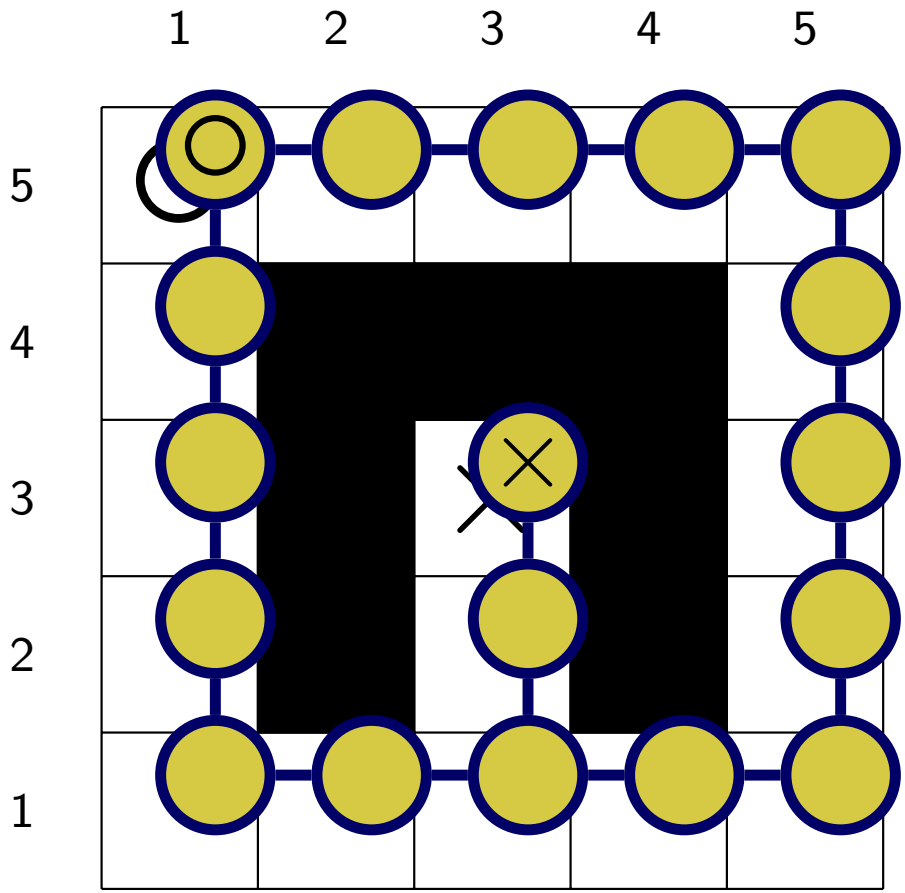


Note: A graph has multiple spanning trees (unless it is already a tree).

Back to connectivity...

So how about connectivity?

... or mazes?



Recall Mazes

- Start at a vertex
- If this is the target vertex, done!
- Push the next adjacent vertex (not filled, or already on the stack) onto the stack
- Explore from that vertex
- When that vertex is explored, pop it and move to the next adjacent vertex
- When everything is explored, return.

This is called Depth-First Search Contrast with Breadth-First Search

Depth First Search (DFS)

Primary Goals

- Visit every vertex in graph $G = (V, E)$.
- Construct a spanning tree for every connected component.
  - **Side Effect**: Compute connected components.
  - **Side Effect**: Compute a path between all connected vertices.
  - **Side Effect**: Determine if the graph is connected.
  - **Side Effect**: Identify any cycles (if they exist).
- Complete in time $O(N + M)$.

Depth First Search (DFS)

`DFS(G)`

Input

- Graph $G = (V, E)$

Output

- Label every edge as a:
  - Spanning Edge: Part of the spanning tree
  - Back Edge: Part of a cycle

Depth First Search (DFS)

`DFSOne(G, v)`

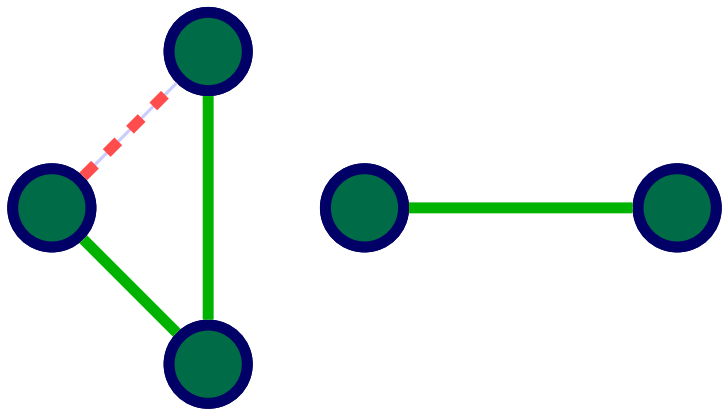
Input

- Graph $G = (V, E)$
- Start vertex $v \in V$

Output

- Label every edge in v's connected component as a:
  - Spanning Edge: Part of the spanning tree
  - Back Edge: Part of a cycle

DFS Example



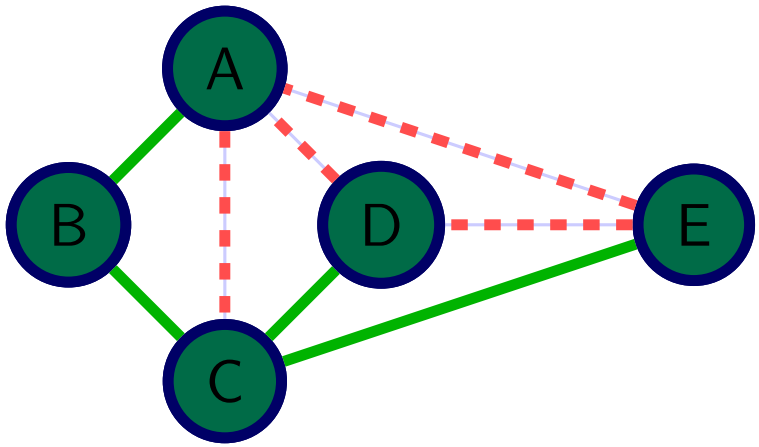
Depth First Search (DFS)

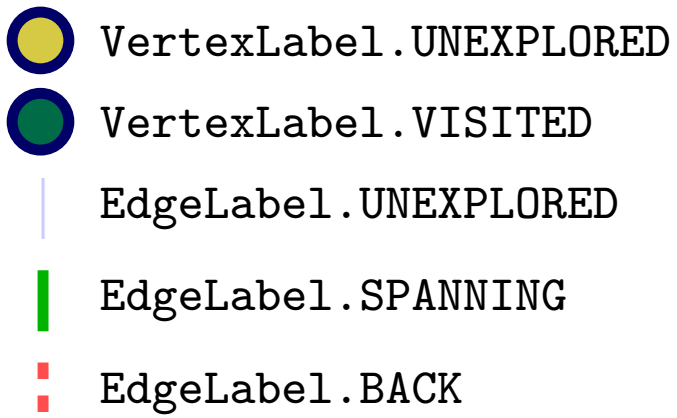
```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     for(Vertex v : graph.vertices)
7                    { v.setLabel(VertexLabel.UNEXPLORED); }
8     for(Edge e   : graph.edges)
9                    { e.setLabel(EdgeLabel.UNEXPLORED); }
10    for(Vertex v : graph.vertices)
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        DFSOne(graph, v)
15      }
16    }
17  }
```

DFSOne

```
1  public void DFSOne(Graph<...> graph, Vertex start)
2  {
3    start.setLabel(VertexLabel.VISITED)
4
5    for( edge : start.getIncident() ) {
6      if(edge.getLabel == EdgeLabel.UNEXPLORED){
7        Vertex otherVertex = edge.getOppositeVertex(start);
8
9        if(otherVertex.getLabel == VertexLabel.UNEXPLORED){
10         edge.setLabel(EdgeLabel.SPANNING);
11         DFSOne(graph, otherVertex);
12       } else {
13         edge.setLabel(EdgeLabel.BACK);
14       }
15     }
16   }
17 }
```

Depth First Search (DFS)





Call Stack

`DFS(G)`

`DFSOne(G, A)` (→ B, C, D)

`DFSOne(G, B)` (→ A, C)

`DFSOne(G, C)` (→ A, B, D, E)

`DFSOne(G, D)` (→ A, C)

`DFSOne(G, E)` (→ A, C, D)

DFS on Mazes

The DFS algorithm is like our stack-based maze solver

- Mark each grid square with `VISITED`
- Mark each path with `SPANNING`
- Only visit each vertex once.

DFS won't necessarily find the shortest paths.

DFS

What's the complexity?

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     for(Vertex v : graph.vertices)
7                    { v.setLabel(VertexLabel.UNEXPLORED); }
8     for(Edge e   : graph.edges)
9                    { e.setLabel(EdgeLabel.UNEXPLORED); }
10    for(Vertex v : graph.vertices)
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        DFSOne(graph, v)
15      }
16    }
17  }
```

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     for(Edge e    : graph.edges)
9                    { e.setLabel(EdgeLabel.UNEXPLORED); }
10    for(Vertex v : graph.vertices)
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        DFSOne(graph, v)
15      }
16    }
17  }
```

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     O(M)
9
10    for(Vertex v : graph.vertices)
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        DFSOne(graph, v)
15      }
16    }
17  }
```

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     O(M)
9
10    O(N) times
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        DFSOne(graph, v)
15      }
16    }
17  }
```

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     O(M)
9
10    O(N) times
11    {
12      O(1)
13      {
14        DFSOne(graph, v)
15      } else {
16        O(1)
17      }
18    }
19  }
```

Depth First Search (DFS)

Observation: `DFSOne` is called on each vertex *at most* once.

- If `v`'s label is `VISITED`, both `DFSOne` and `DFS` skip it

$$O(N) \text{ calls to } \texttt{DFSOne}$$

What's the runtime of `DFSOne` **excluding recursive calls**?

DFSOne

```
1  public void DFSOne(Graph<...> graph, Vertex start)
2  {
3    start.setLabel(VertexLabel.VISITED)
4
5    for( edge : start.getIncident() ) {
6      if(edge.getLabel == EdgeLabel.UNEXPLORED){
7        Vertex otherVertex = edge.getOppositeVertex(start);
8
9        if(otherVertex.getLabel == VertexLabel.UNEXPLORED){
10         edge.setLabel(EdgeLabel.SPANNING);
11         DFSOne(graph, otherVertex);
12       } else {
13         edge.setLabel(EdgeLabel.BACK);
14       }
15     }
16   }
17 }
```

DFSOne

```
1  public void DFSOne(Graph<...> graph, Vertex start)
2  {
3    O(1)
4
5    O(deg(start)) times {
6      O(1) {
7        O(1)
8
9        O(1) {
10         O(1)
11         DFSOne(graph, otherVertex);
12       } else {
13         O(1)
14       }
15     }
16   }
17 }
```

Depth First Search (DFS)

What's the runtime of `DFSOne` **excluding recursive calls**?

$$O(\texttt{deg}(\texttt{start}))$$

Depth First Search (DFS)

What's the sum over all calls to `DFSOne`

$$\sum_{v \in V} O(\deg(v))$$

$$= O\left(\sum_{v \in V} \deg(v)\right)$$

$$= O(2M) = O(M)$$

To summarize...

Mark Vertices `UNVISITED`	$O(N)$
Mark Edges `UNVISITED`	$O(M)$
`DFS` Vertex Loop	$O(N)$
All calls to `DFSOne`	$O(M)$
	$O(N+M)$