CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 19: Adjacency Lists and DFS

Announcements

- WA3 was due yesterday
  - Submissions today will have a 50% penalty (no grace days allowed)
  - Submissions close tonight at midnight
- PA2 released
  - Testing phase due Sunday 10/22
  - Implementation due Sunday 11/5

Edge List Summary

Graph

vertices: LinkedList<Vertex>
edges: LinkedList<Edge>

Vertex

label: T
node: LinkedListNode

Edge

label: T
vertex: origin
vertex: destination
node: LinkedListNode

Storing the list nodes in the edges/vertices allows us to remove by reference in $\Theta(1)$ time

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(m)$
- `vertex.incidentEdges`: $O(m)$
- `vertex.edgeTo`: $O(m)$
- **Space Used:** $O(n) + O(m)$

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(m)$
- `vertex.incidentEdges`: $O(m)$
- `vertex.edgeTo`: $O(m)$
- **Space Used: $O(n) + O(m)$**

← Involves checking every edge in the graph

How can we improve?

How can we improve?

Idea: Store the in/out edges for each vertex!

(Called an adjacency list)

Adjacency List

```java
public class Vertex<V,E> {
  public Node<Vertex> node;
  public List<Edge> inEdges = new CustomLinkedList<Edge>();
  public List<Edge> outEdges = new CustomLinkedList<Edge>();
  /*...*/
}
```

Each vertex stores a list of **`inEdges`** and **`outEdges`**, which are maintained as the graph is modified…

What functions need to change to maintain these lists?

Adjacency List

```java
public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  e.node = edges.add(e);
  orig.outEdges.add(e);
  dest.inEdges.add(e);
  return e;
}
```

← When we add an edge to the graph, also add it to the appropriate adjacency lists

What is the complexity of **addEdge** now?

Adjacency List

```java
public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  e.node = edges.add(e);
  orig.outEdges.add(e);
  dest.inEdges.add(e);
  return e;
}
```

← When we add an edge to the graph, also add it to the appropriate adjacency lists

What is the complexity of **addEdge** now? Still $\Theta(1)$

Adjacency List

```java
public void removeEdge(Edge edge) {
  edges.remove(edge.node);
  edge.orig.outEdges.remove(edge);
  edge.dest.inEdges.remove(edge);
}
```

← When we remove an edge from the graph, also remove it from the adjacency lists

What is the complexity of **`removeEdge`** now?

Adjacency List

```java
public void removeEdge(Edge edge) {
  edges.remove(edge.node);
  edge.orig.outEdges.remove(edge);
  edge.dest.inEdges.remove(edge);
}
```

← When we remove an edge from the graph, also remove it from the adjacency lists

What is the complexity of **`removeEdge`** now? ***O*(deg(orig) + deg(dest)) :(**

But how can we fix this?

Adjacency List

```
public class Edge<V,E> {
  public Node<Edge> node;
  public Node<Edge> inNode;
  public Node<Edge> outNode;
  /*...*/
}
```

Each Edge now also stores a reference to the nodes in each adjacency list

Adjacency List

```java
public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  e.node = edges.add(e);
  e.outNode = orig.outEdges.add(e);
  e.inNode = dest.inEdges.add(e);
  return e;
}
```

← When we add an edge to the graph, also add it to the appropriate adjacency lists AND store the node refs in the Edge object

What is the complexity of **addEdge** now? Still $\Theta(1)$

Adjacency List

```
public void removeEdge(Edge edge) {
  edges.remove(edge.node);
  edge.orig.outEdges.remove(edge.outNode);
  edge.dest.inEdges.remove(edge.inNode);
}
```

← When we remove an edge from the graph, also remove it from the adjacency lists (remove by reference)

What is the complexity of **`removeEdge`** now?

Adjacency List

```
public void removeEdge(Edge edge) {
  edges.remove(edge.node);
  edge.orig.outEdges.remove(edge.outNode);
  edge.dest.inEdges.remove(edge.inNode);
}
```

← When we remove an edge from the graph, also remove it from the adjacency lists (remove by reference)

What is the complexity of `removeEdge` now? $\Theta(1)$

Adjacency List

So, we are able to store and maintain adjacency lists in each vertex while still keeping a $\Theta(1)$ runtime for **`addVertex`**, **`addEdge`**, and **`removeEdge`**

How much extra space is used?

Adjacency List

So, we are able to store and maintain adjacency lists in each vertex while still keeping a $\Theta(1)$ runtime for **`addVertex`**, **`addEdge`**, and **`removeEdge`**

How much extra space is used? $\Theta(1)$ per edge

Each edge only appears in 3 lists:

- The edge list
- One vertices inList
- One vertices outList

Adjacency List

So, we are able to store and maintain adjacency lists in each vertex while still keeping a $\Theta(1)$ runtime for **`addVertex`**, **`addEdge`**, and **`removeEdge`**

How much extra space is used? $\Theta(1)$ per edge

Each edge only appears in 3 lists:

- The edge list
- One vertices inList
- One vertices outList

But now what have we gained?

Adjacency List

```
public void removeVertex(Vertex v) {
  for(edge : v.getIncidentEdges()) {
    removeEdge(edge.node)
  }
  vertices.remove(v.node);
}
```

What is the complexity of **`removeVertex`** now?

Adjacency List

```
public void removeVertex(Vertex v) {
  for(edge : v.getIncidentEdges()) {
    Θ(1)
  }
  Θ(1)
}
```

What is the complexity of **`removeVertex`** now?

Adjacency List

```
public void removeVertex(Vertex v) {
  for(edge : v.getIncidentEdges()) {
    Θ(1)
  }
  Θ(1)
}
```

We now have a reference to the list of edges in $\Theta(1)$ time, and there are **deg(v)** edge in the list

What is the complexity of **`removeVertex`** now?

Adjacency List

```
public void removeVertex(Vertex v) {
  for(edge : v.getIncidentEdges()) {
    Θ(1)
  }
  Θ(1)
}
```

We now have a reference to the list of edges in $\Theta(1)$ time, and there are **deg(v)** edge in the list

What is the complexity of **`removeVertex`** now? $\boldsymbol{\Theta(\mathit{deg}(v))}$

Adjacency List Summary

<u>Graph</u>

vertices: LinkedList[Vertex]
edges: LinkedList[Edge]

<u>Vertex</u>

label: T
node: LinkedListNode
inEdges: LinkedList[Edge]
outEdges: LinkedList[Edge]

<u>Edge</u>

label: T
node: LinkedListNode
inNode: LinkedListNode
outNode: LinkedListNode

Storing the list of incident edges in the vertex saves us the time of checking every edge in the graph.

The edge now stores additional nodes to ensure removal is still $\Theta(1)$

Adjacency List Summary

- `addEdge, addVertex`: $\Theta(1)$
- `removeEdge`: $\Theta(1)$
- `removeVertex`: $\Theta(\text{deg}(\text{vertex}))$
- `vertex.incidentEdges`: $\Theta(\text{deg}(\text{vertex}))$
- `vertex.edgeTo`: $\Theta(\text{deg}(\text{vertex}))$
- **Space Used:** $\Theta(n) + \Theta(m)$

Adjacency List Summary

- `addEdge, addVertex`: $\Theta(1)$
- `removeEdge`: $\Theta(1)$
- `removeVertex`: $\Theta(\text{deg}(\text{vertex}))$
- `vertex.incidentEdges`: $\Theta(\text{deg}(\text{vertex}))$
- `vertex.edgeTo`: $\Theta(\text{deg}(\text{vertex}))$
- **Space Used:** $\Theta(n) + \Theta(m)$

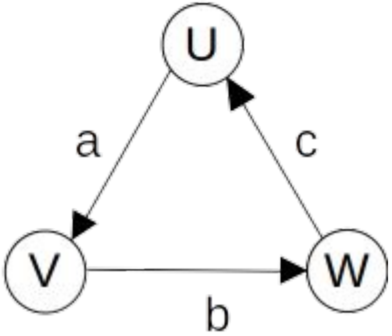
Now we already know what edges are incident without having to check them all

Adjacency Matrix

Destination

Origin

	U	V	W
U	***-***	***a***	***-***
V	***-***	***-***	***b***
W	***c***	***-***	***-***



Adjacency Matrix Summary

- `addEdge, removeEdge`:
- `addVertex, removeVertex`:
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Adjacency Matrix Summary

Just change a single entry of the matrix

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`:
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Adjacency Matrix Summary

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`: $\Theta(n^2)$
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Resize and copy the whole matrix

Adjacency Matrix Summary

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`: $\Theta(n^2)$
- `vertex.incidentEdges`: $\Theta(n)$
- `vertex.edgeTo`:
- **Space Used:**

Check the row and column for that vertex

Adjacency Matrix Summary

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`: $\Theta(n^2)$
- `vertex.incidentEdges`: $\Theta(n)$
- `vertex.edgeTo`: $\Theta(1)$
- **Space Used:**

Check a single entry of the matrix

Adjacency Matrix Summary

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`: $\Theta(n^2)$
- `vertex.incidentEdges`: $\Theta(n)$
- `vertex.edgeTo`: $\Theta(1)$
- **Space Used: $\Theta(n^2)$**

How does this relate to space of edge/adjacency lists?

Adjacency Matrix Summary

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`: $\Theta(n^2)$
- `vertex.incidentEdges`: $\Theta(n)$
- `vertex.edgeTo`: $\Theta(1)$
- **Space Used:** $\Theta(n^2)$

How does this relate to space of edge/adjacency lists? **If the matrix is "dense" it's about the same**

So…what do we do with our graphs?

Connectivity Problems

Given graph ***G***:

Connectivity Problems

Given graph ***G***:

- Is vertex ***u*** **adjacent** to vertex ***v***?

Connectivity Problems

Given graph ***G***:

- Is vertex ***u*** **adjacent** to vertex ***v***?
- Is vertex ***u*** **connected** to vertex ***v*** via some path?

Connectivity Problems

Given graph ***G***:

- Is vertex ***u*** **adjacent** to vertex ***v***?
- Is vertex ***u*** **connected** to vertex ***v*** via some path?
- Which vertices are **connected** to vertex ***v***?

Connectivity Problems

Given graph ***G***:

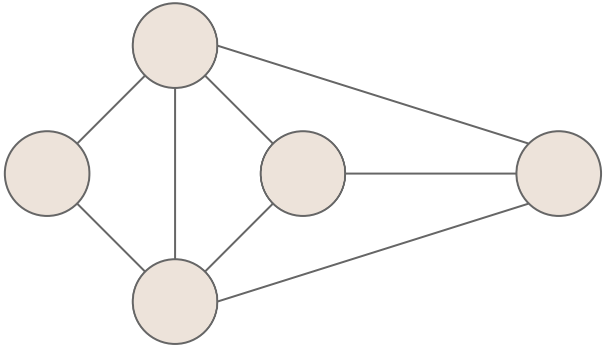
- Is vertex ***u*** **adjacent** to vertex ***v***?
- Is vertex ***u*** **connected** to vertex ***v*** via some path?
- Which vertices are **connected** to vertex ***v***?
- What is the **shortest path** from vertex ***u*** to vertex ***v***?

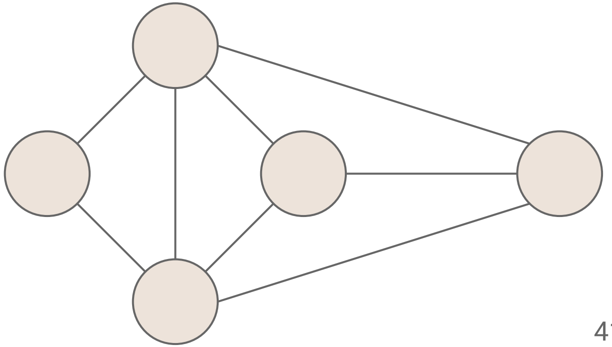
A few more definitions

A **<u>subgraph</u>, *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



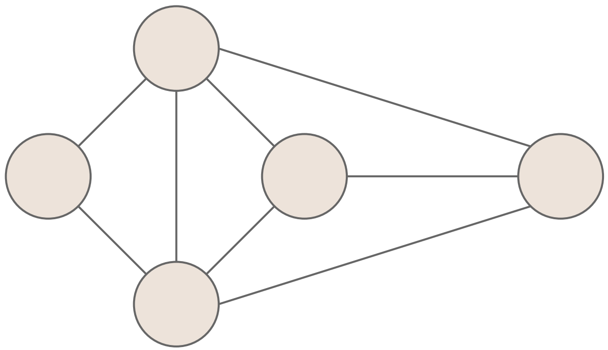


A few more definitions

A **<u>subgraph</u>, *S*,** of a graph ***G*** is a graph where:

S's vertices are a subset of ***G***'s vertices

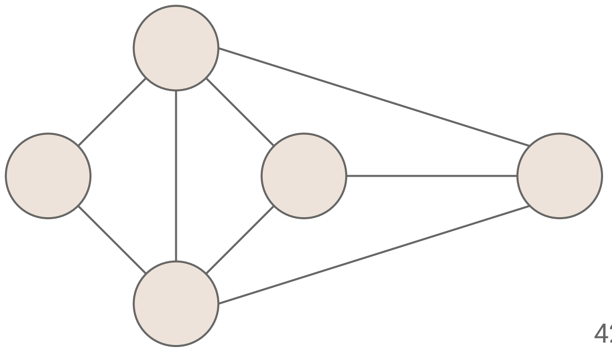
S's edges are a subset of ***G***'s edges



A **<u>spanning subgraph</u>** of ***G*…**

Is a subgraph of ***G***

Contains all of ***G***'s vertices

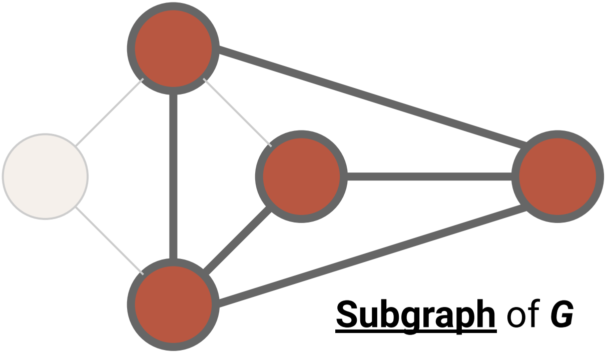


A few more definitions

A **<u>subgraph</u>, *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

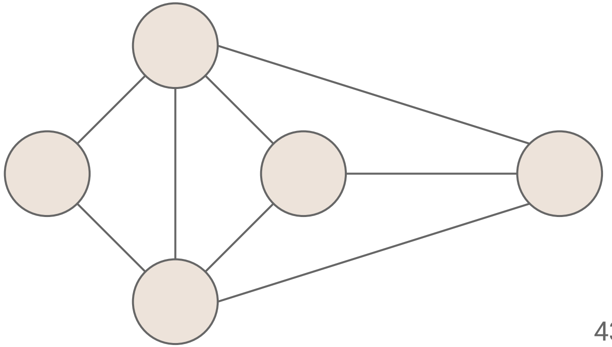


<u>Subgraph</u> of ***G***

A **<u>spanning subgraph</u>** of ***G*…**

Is a subgraph of ***G***

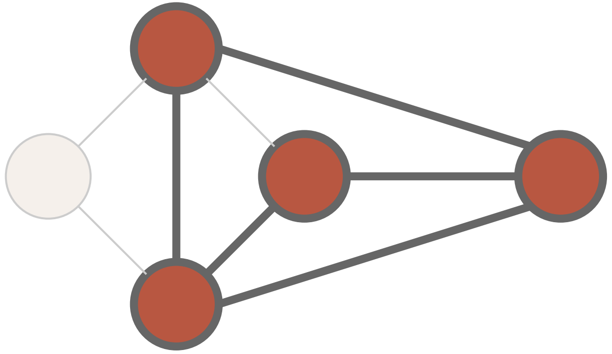
Contains all of ***G***'s vertices



A few more definitions

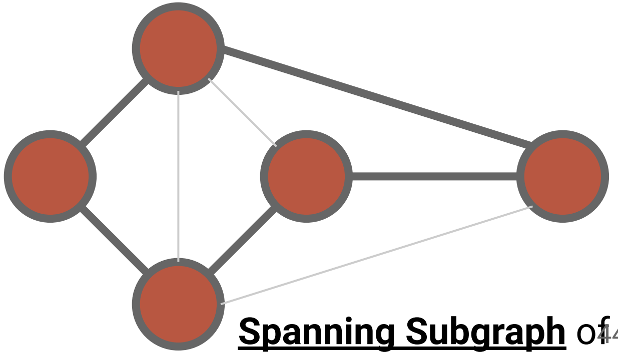
A **<u>subgraph</u>, *S*,** of a graph ***G*** is a graph where:

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



A **<u>spanning subgraph</u>** of ***G*…**

- Is a subgraph of ***G***
- Contains all of ***G***'s vertices

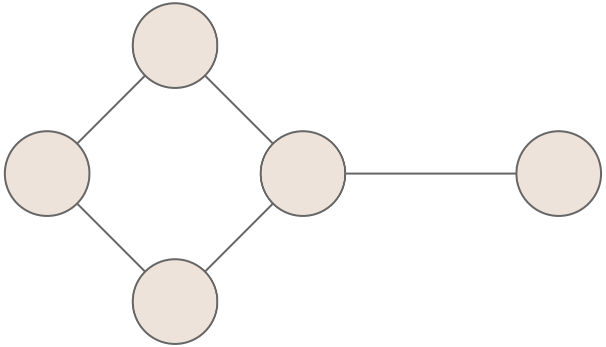


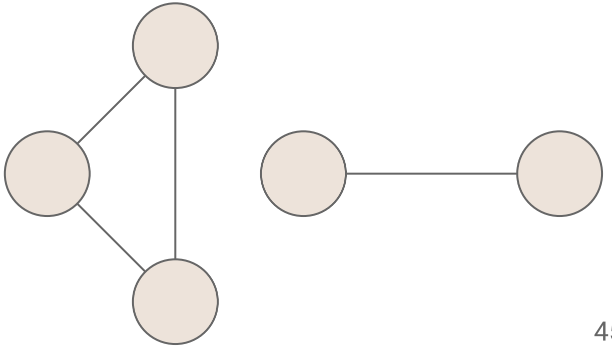
<u>Spanning Subgraph</u> of ***G***

A few more definitions

A graph is **<u>connected</u>**…

If there is a path between every pair of vertices

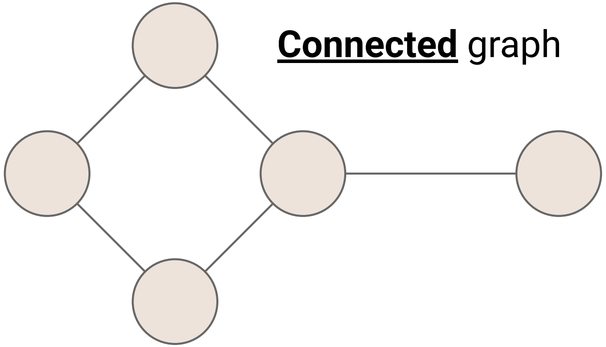




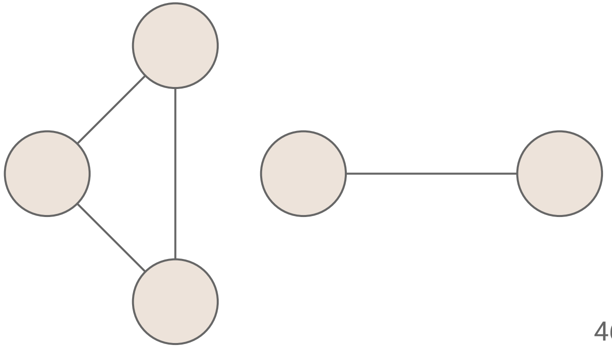
A few more definitions

A graph is **connected**…

If there is a path between every pair of vertices



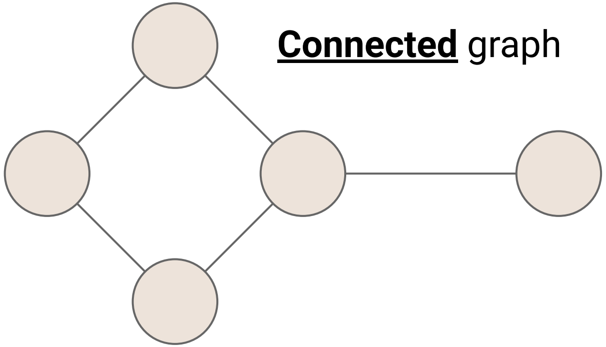
Connected graph

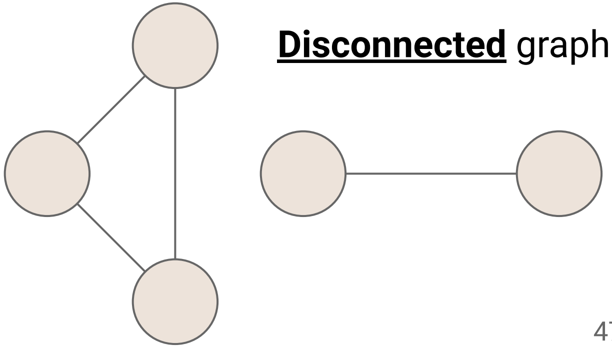


A few more definitions

A graph is **<u>connected</u>**…

If there is a path between every pair of vertices





A few more definitions

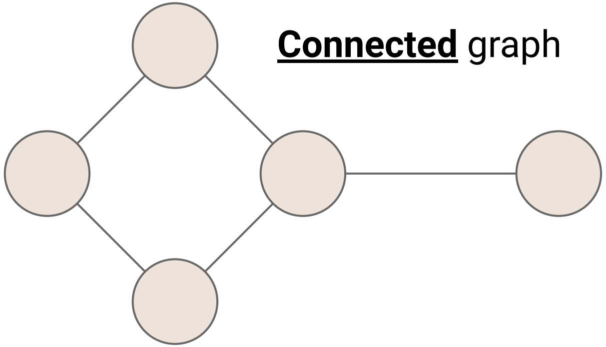
A graph is **connected**…

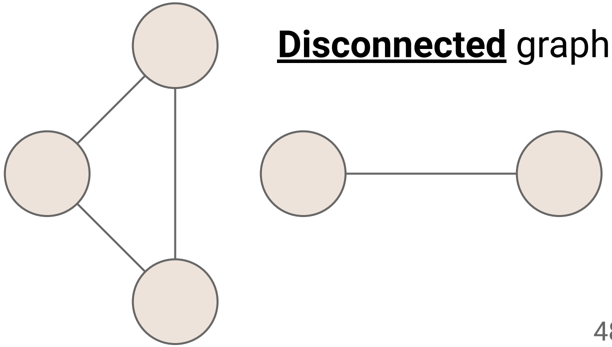
If there is a path between every pair of vertices

A **connected component** of ***G***…

Is a maximal connected subgraph of ***G***

- "maximal" means you can't add a new vertex without breaking the property
- Any subset of ***G***'s edges that connect the subgraph are fine





A few more definitions

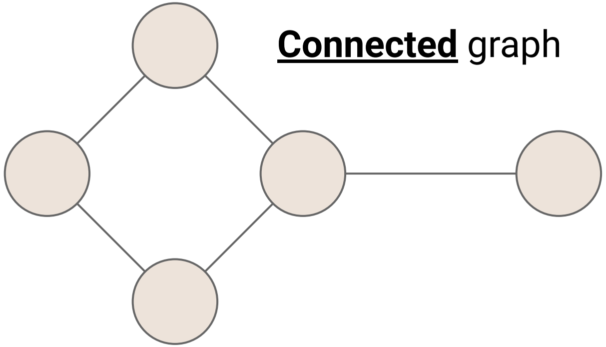
A graph is **connected**…

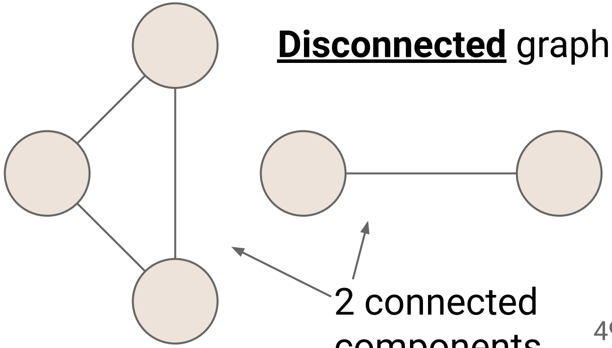
If there is a path between every pair of vertices

A **connected component** of ***G*…**

Is a maximal connected subgraph of ***G***

- "maximal" means you can't add a new vertex without breaking the property
- Any subset of ***G***'s edges that connect the subgraph are fine





A few more definitions

A **free tree** is an undirected graph ***T*** such that…

There is exactly one simple path between any two nodes

- ***T*** is connected
- ***T*** has no cycles

A few more definitions

A <u>**free tree**</u> is an undirected graph ***T*** such that…

There is exactly one simple path between any two nodes

- ***T*** is connected
- ***T*** has no cycles

A <u>**rooted tree**</u> is a directed graph ***T*** such that…

One vertex of ***T*** is the <u>**root**</u>

There is exactly one simple path from the root to every other vertex in the graph

A few more definitions

A **free tree** is an undirected 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 the graph

A (free/rooted) **forest** is a graph ***F*** such that…

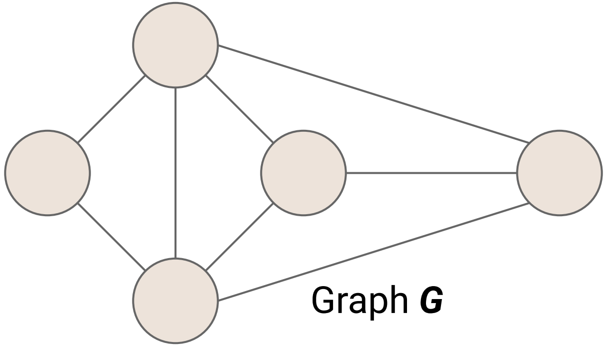
Every connected component is a tree

A few more definitions

A **spanning tree** of a connected graph…

…Is a spanning subgraph that is a tree

…It is not unique unless the graph is a tree


Graph ***G***

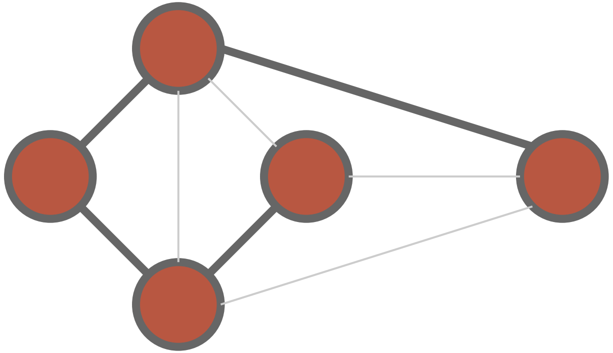
A few more definitions

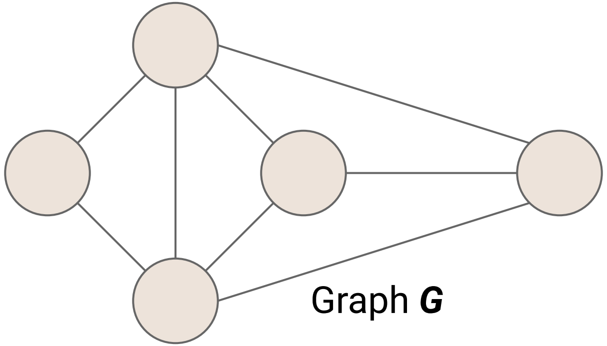
A **<u>spanning tree</u>** of a connected graph…

…Is a spanning subgraph that is a tree

…It is not unique unless the graph is a tree

A **<u>Spanning Tree</u>** of ***G***





Graph ***G***

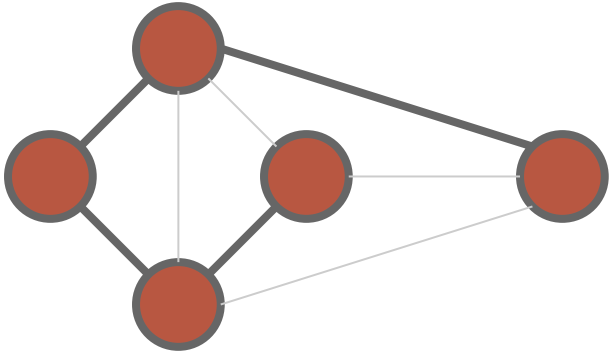
A few more definitions

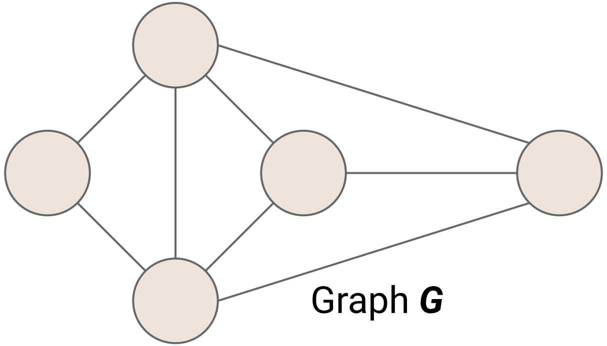
A **spanning tree** of a connected graph…

…Is a spanning subgraph that is a tree

…It is not unique unless the graph is a tree

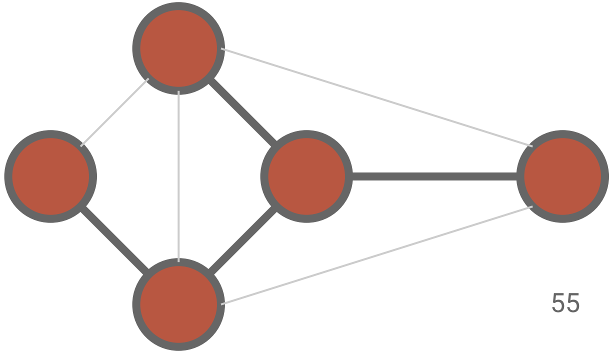
A **Spanning Tree** of ***G***





Graph ***G***

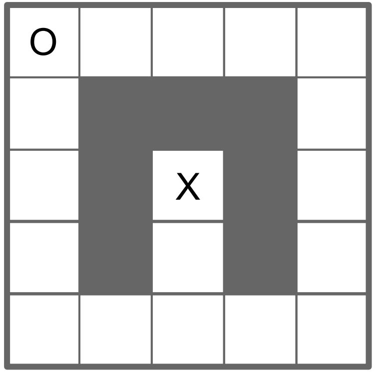
Another **Spanning Tree** of ***G***



Now back to the question…Connectivity

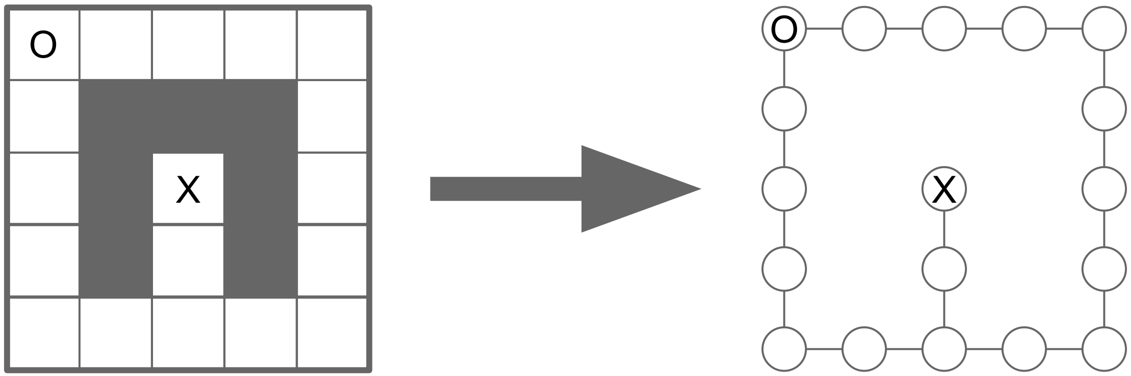
Back to Mazes

How could we represent our maze as a graph?



Back to Mazes

How could we represent our maze as a graph?



Recall

Searching the maze with a stack

We try every path, one at a time, following it as far as we can

…then backtrack and try another

Recall

Searching the maze with a stack (Depth-First Search)

We try every path, one at a time, following it as far as we can

…then backtrack and try another

Recall

Searching the maze with a stack (Depth-First Search)

We try every path, one at a time, following it as far as we can

…then backtrack and try another

Searching with a queue?

TBD…

Depth-First Search

<u>Primary Goals</u>

- Visit every vertex in graph $\boldsymbol{G = (V,E)}$
- Construct a spanning tree for every connected component

Depth-First Search

<u>Primary Goals</u>

- Visit every vertex in graph ***G* = (*V*,*E*)**
- Construct a spanning tree for every connected component
  - **Side Effect:** Compute connected components

Depth-First Search

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

Depth-First Search

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

Depth-First Search

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 cycles

Depth-First Search

<u>Primary Goals</u>

- Visit every vertex in graph $\boldsymbol{G = (V,E)}$
- 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 cycles
- Complete in time $\boldsymbol{O(|V| + |E|)}$

Depth-First Search

DFS

Input: Graph ***G* = (*V*,*E*)**

Output: Label every edge as:

- <u>Spanning Edge</u>: Part of the spanning tree
- <u>Back Edge</u>: Part of a cycle

Depth-First Search

DFS

Input: Graph $G = (V,E)$

Output: Label every edge as:

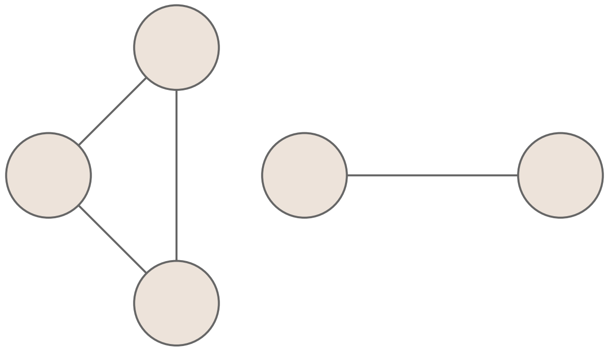
- Spanning Edge: Part of the spanning tree
- Back Edge: Part of a cycle

DFSOne

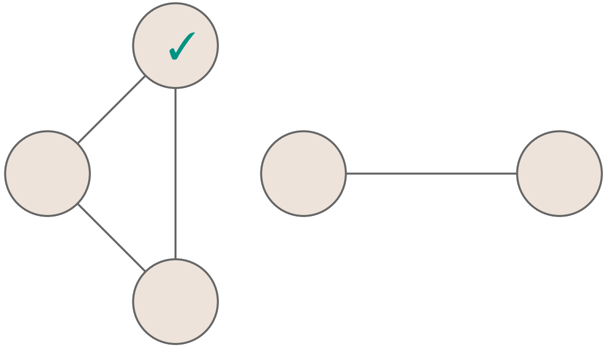
Input: Graph $G = (V,E)$, start vertex $v \in V$

Output: Label every edge in v's connected component

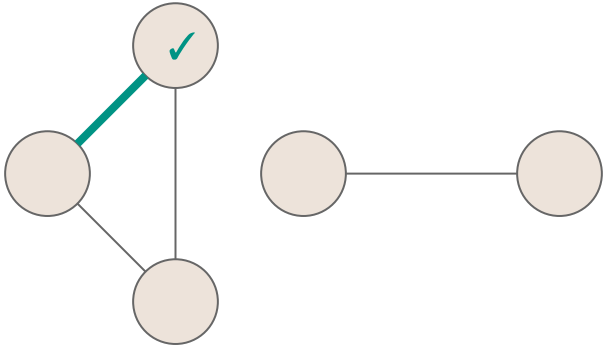
Depth-First Search



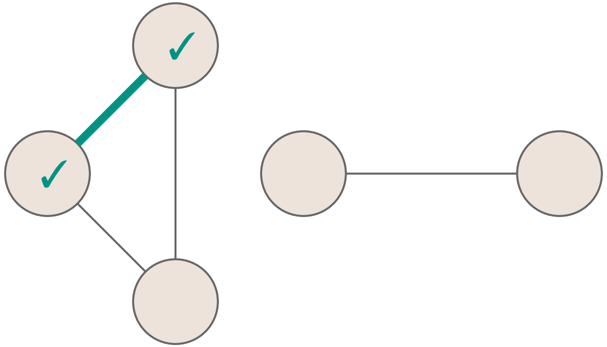
Depth-First Search



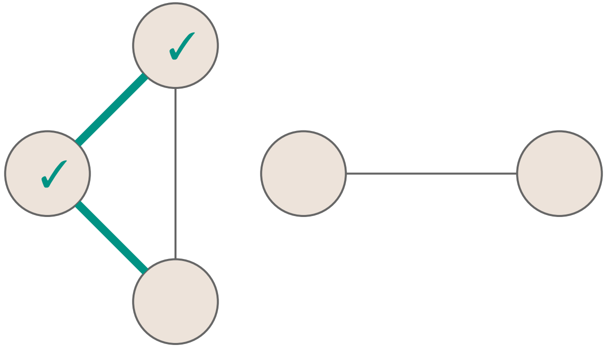
Depth-First Search



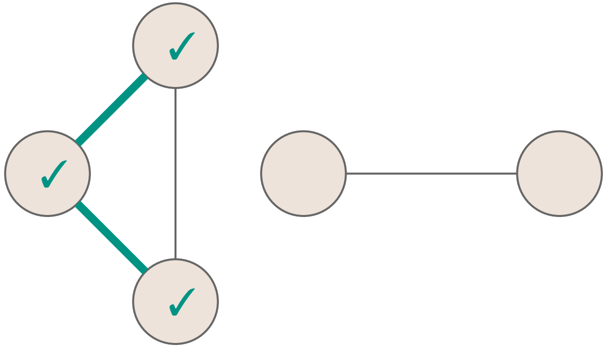
Depth-First Search



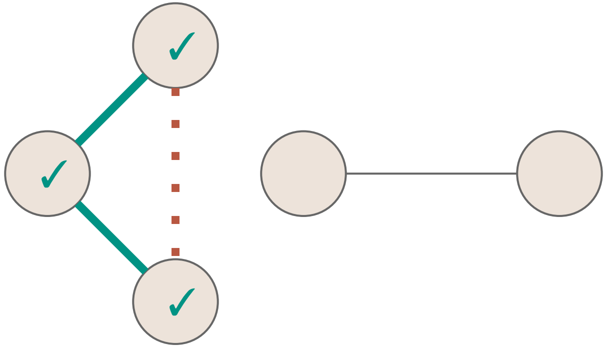
Depth-First Search



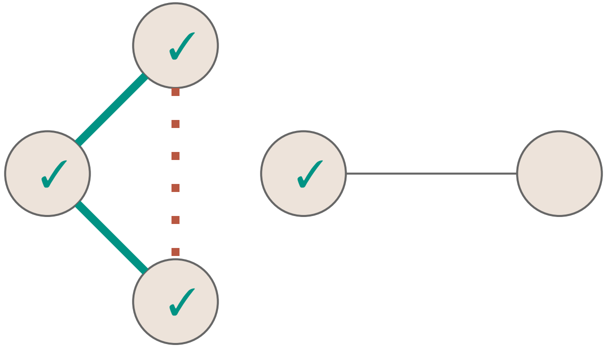
Depth-First Search



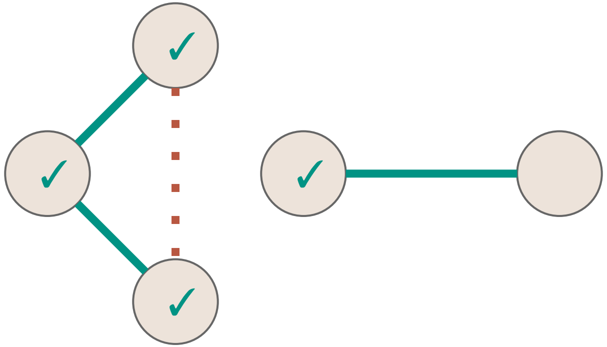
Depth-First Search



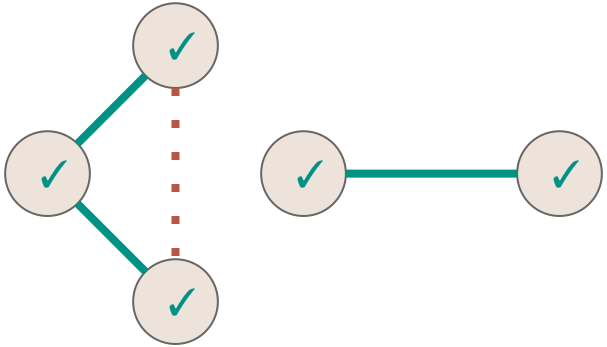
Depth-First Search



Depth-First Search



Depth-First Search



DFS

```
public void DFS(Graph graph) {
  for (Vertex v : graph.vertices) {
    v.setLabel(UNEXPLORED);
  }
  for (Edge e : graph.edges) {
    e.setLabel(UNEXPLORED);
  }
  for (Vertex v : graph.vertices) {
    if (v.label == UNEXPLORED) {
      DFSOne(graph, v);
    }
  }
}
```

DFS

```java
public void DFS(Graph graph) {
  for (Vertex v : graph.vertices) {
    v.setLabel(UNEXPLORED);
  }
  for (Edge e : graph.edges) {
    e.setLabel(UNEXPLORED);
  }
  for (Vertex v : graph.vertices) {
    if (v.label == UNEXPLORED) {
      DFSOne(graph, v);
    }
  }
}
```

Initialize all vertices and edges to UNEXPLORED

DFS

```
public void DFS(Graph graph) {
  for (Vertex v : graph.vertices) {
    v.setLabel(UNEXPLORED);
  }
  for (Edge e : graph.edges) {
    e.setLabel(UNEXPLORED);
  }
  for (Vertex v : graph.vertices) {
    if (v.label == UNEXPLORED) {
      DFSOne(graph, v);
    }
  }
}
```

Call DFSOne to label the connected component of every unexplored vertex

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED); ← Mark the vertex as VISITED (so we'll never try to visit it again)
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

DFSOne

```java
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
  }
}
```

Check every outgoing edge (every possible way we could leave the current vertex)

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Follow the unexplored edges

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

If it leads to an unexplored vertex, then it is a spanning edge. Recursively explore that vertex.

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Otherwise, we just found a cycle

Detailed Example

UNEXPLORED

VISITED

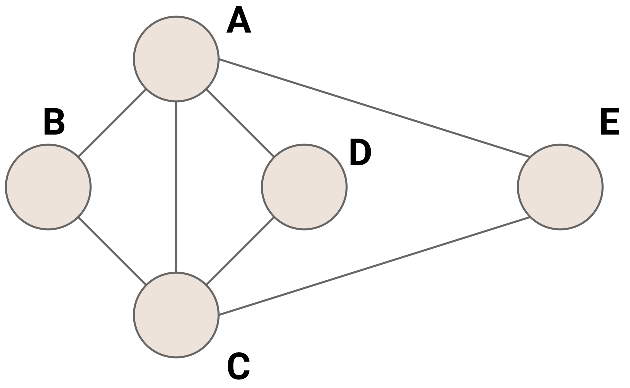
UNEXPLORED

SPANNING

BACK

Call Stack

(→ edges to list)



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

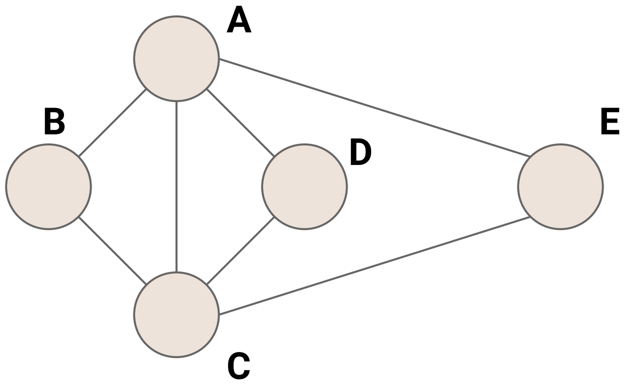
SPANNING

BACK

Call Stack

DFS(G)

(→ edges to list)



Detailed Example

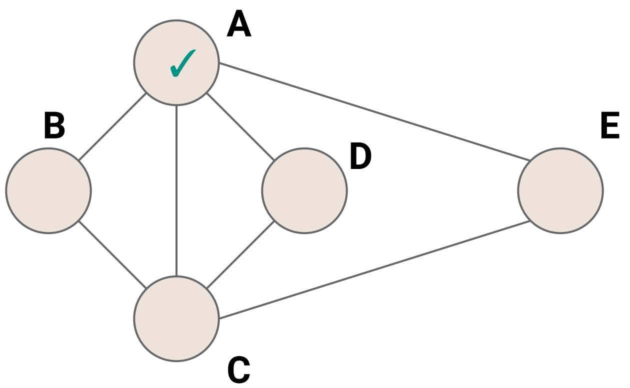
UNEXPLORED

✓ VISITED

UNEXPLORED

SPANNING

BACK



Call Stack

(→ edges to list)

DFS(G)
DFSOne(G,A)

Detailed Example

UNEXPLORED

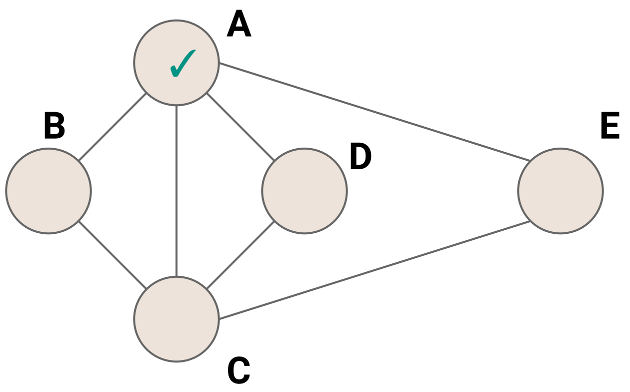
VISITED

UNEXPLORED

SPANNING

BACK

Call Stack	(→ edges to list)
DFS(G)	
DFSOne(G,A)	(→ B, C, D)



Detailed Example

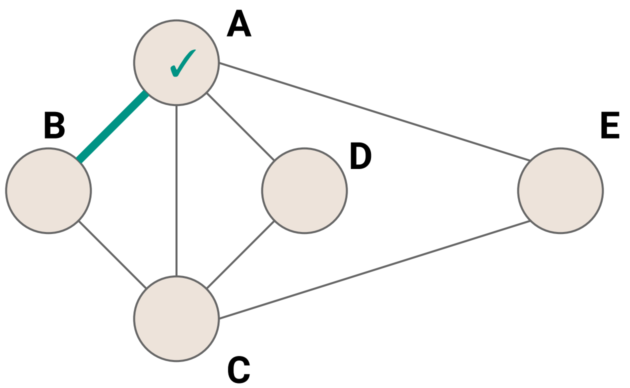
UNEXPLORED

✓ VISITED

UNEXPLORED

SPANNING

BACK



<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSOne(G,A)	(→ B, C, D)

Detailed Example

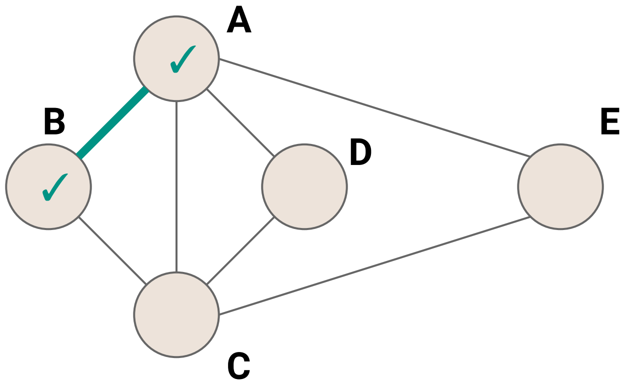
UNEXPLORED

✓ VISITED

UNEXPLORED

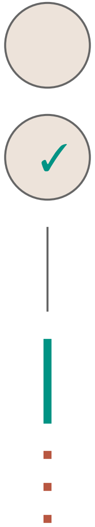
SPANNING

BACK



<u>Call Stack</u>	<u>(→ edges to list)</u>
`DFS(G)`	
`DFSOne(G,A)`	(→ B, C, D)
`DFSOne(G,B)`	(→ A, C)

Detailed Example



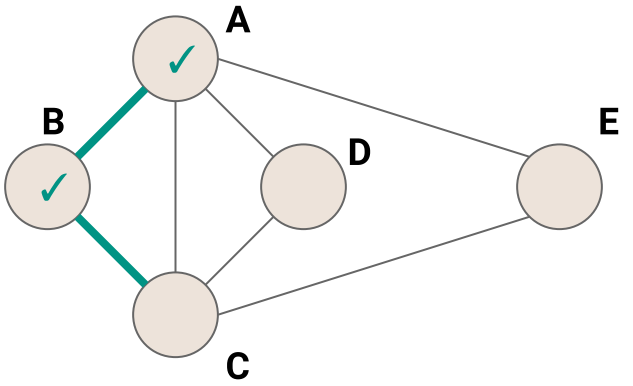
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack	(→ edges to list)
`DFS(G)`	
`DFSOne(G,A)`	(→ B, C, D)
`DFSOne(G,B)`	(→ A, C)

Detailed Example

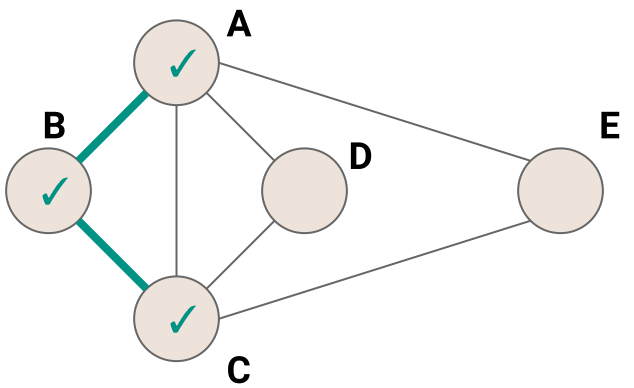
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack	(→ edges to list)
DFS(G)	
DFSOne(G,A)	(→ B, C, D)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)

Detailed Example

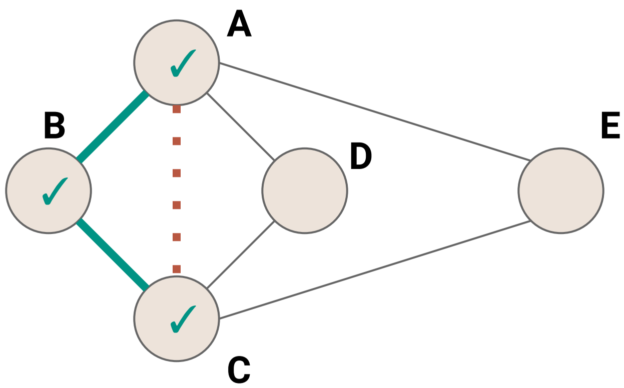
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



<u>Call Stack</u> <u>(→ edges to list)</u>

DFS(G)

DFSOne(G,A) (→ B, C, D)

DFSOne(G,B) (→ A, C)

DFSOne(G,C) (→ B, A, D, E)

Detailed Example

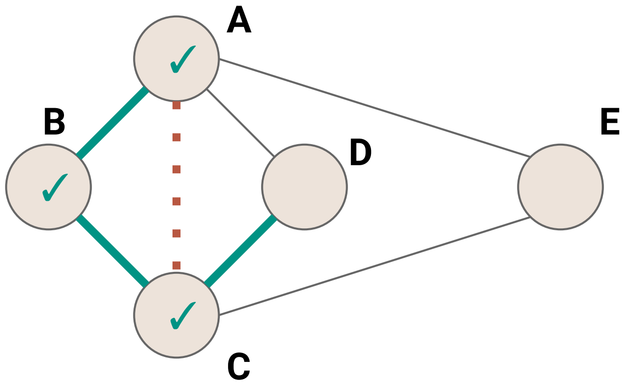
UNEXPLORED

VISITED

UNEXPLORED

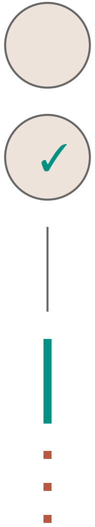
SPANNING

BACK



<u>Call Stack</u>	<u>(→ edges to list)</u>
`DFS(G)`	
`DFSOne(G,A)`	(→ B, C, D)
`DFSOne(G,B)`	(→ A, C)
`DFSOne(G,C)`	(→ B, A, D, E)

Detailed Example



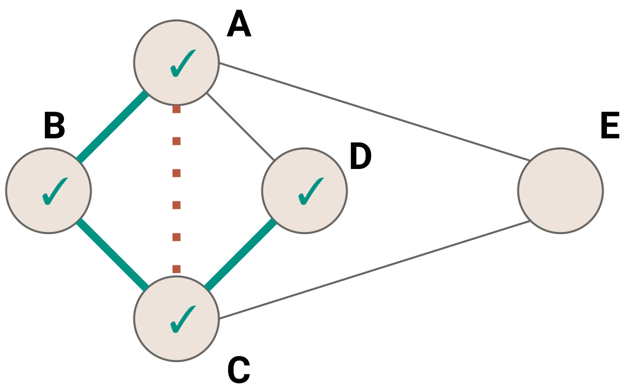
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



<u>Call Stack</u>	<u>(→ edges to list)</u>
`DFS(G)`	
`DFSOne(G,A)`	(→ B, C, D)
`DFSOne(G,B)`	(→ A, C)
`DFSOne(G,C)`	(→ B, A, D, E)
`DFSOne(G,D)`	(→ A, C)

Detailed Example

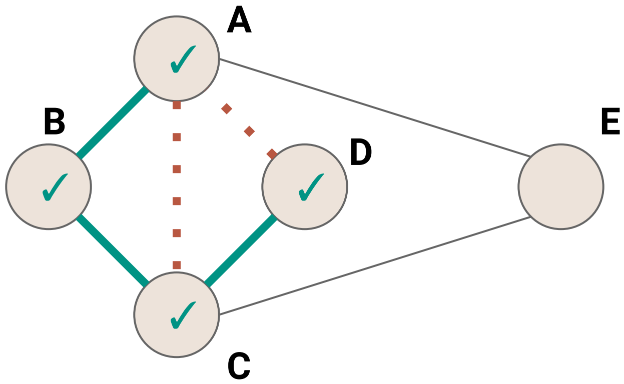
UNEXPLORED

VISITED

UNEXPLORED

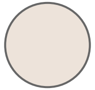
SPANNING

BACK



Call Stack	(→ edges to list)
`DFS(G)`	
`DFSOne(G,A)`	(→ B, C, D)
`DFSOne(G,B)`	(→ A, C)
`DFSOne(G,C)`	(→ B, A, D, E)
`DFSOne(G,D)`	(→ A, C)

Detailed Example



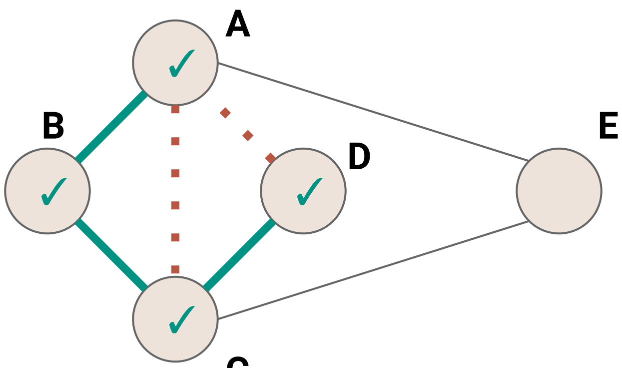
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack	(→ edges to list)
DFS(G)	
DFSOne(G,A)	(→ B, C, D)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)

Detailed Example

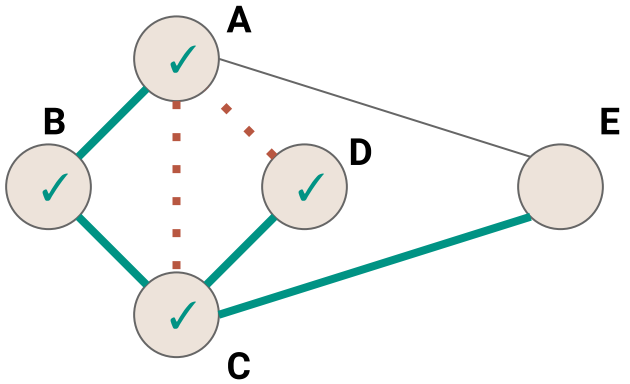
UNEXPLORED

VISITED

UNEXPLORED

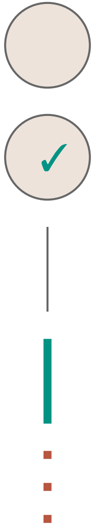
SPANNING

BACK



Call Stack	(→ edges to list)
DFS(G)	
DFSOne(G,A)	(→ B, C, D)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)

Detailed Example



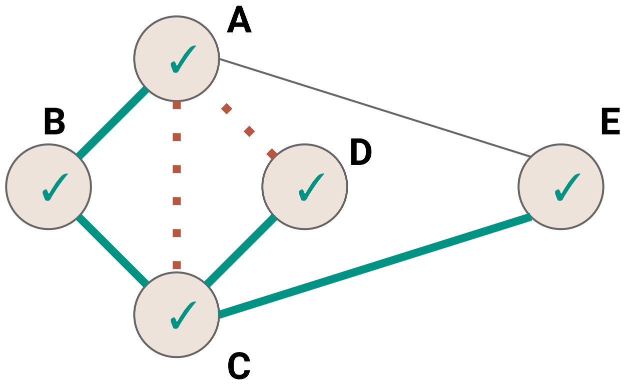
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack	(→ edges to list)
`DFS(G)`	
`DFSOne(G,A)`	(→ B, C, D)
`DFSOne(G,B)`	(→ A, C)
`DFSOne(G,C)`	(→ B, A, D, E)
`DFSOne(G,E)`	(→ A, C)

Detailed Example

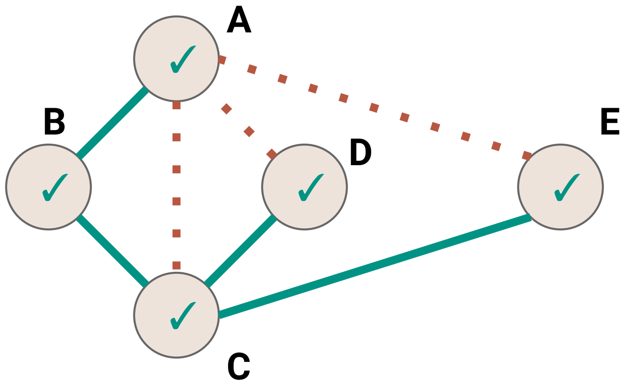
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack	(→ edges to list)
DFS(G)	
DFSOne(G,A)	(→ B, C, D)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)
DFSOne(G,E)	(→ A, C)

Detailed Example

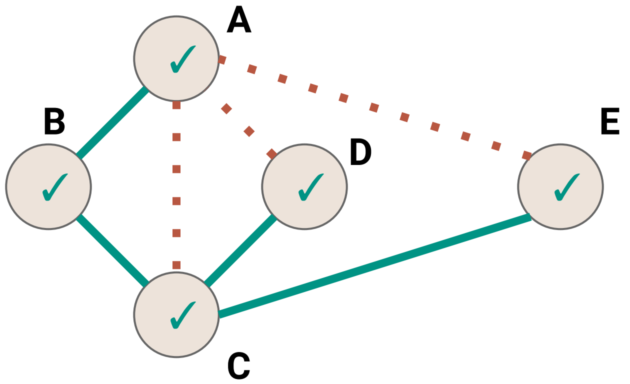
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack	(→ edges to list)
DFS(G)	
DFSOne(G,A)	(→ B, C, D)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)

Detailed Example

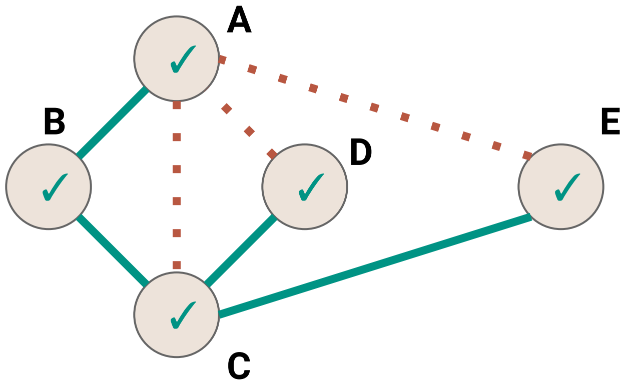
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack	(→ edges to list)
DFS(G)	
DFSOne(G,A)	(→ B, C, D)
DFSOne(G,B)	(→ A, C)

Detailed Example

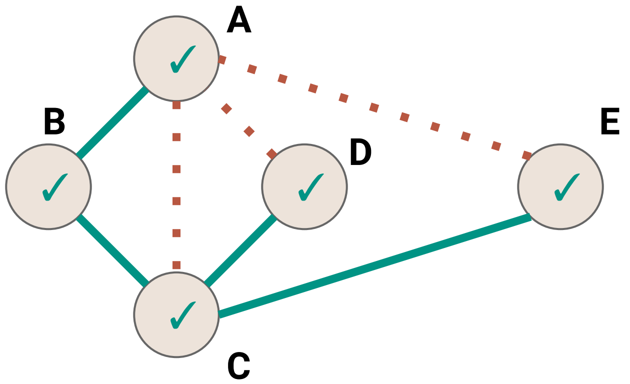
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack | (→ edges to list)
DFS(G)
DFSOne(G,A) (→ B, C, D)

Detailed Example

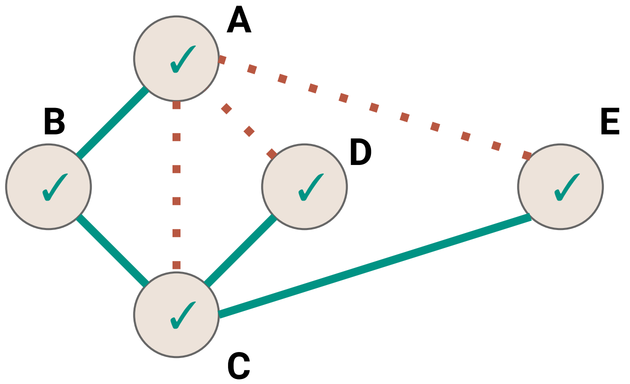
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK

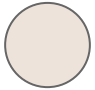


<u>Call Stack</u>

```
DFS(G)
DFSOne(G,B)
```

<u>(→ edges to list)</u>

Detailed Example



UNEXPLORED

VISITED

UNEXPLORED

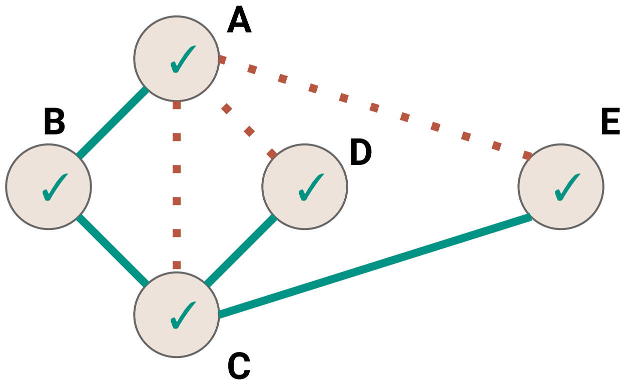
SPANNING

BACK

Call Stack

DFS(G)
DFSOne(G,C)

(→ edges to list)



Detailed Example

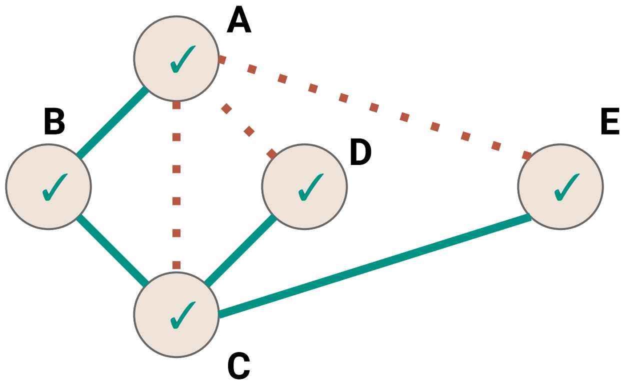
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack

DFS(G)
DFSOne(G,D)

(→ edges to list)

Detailed Example

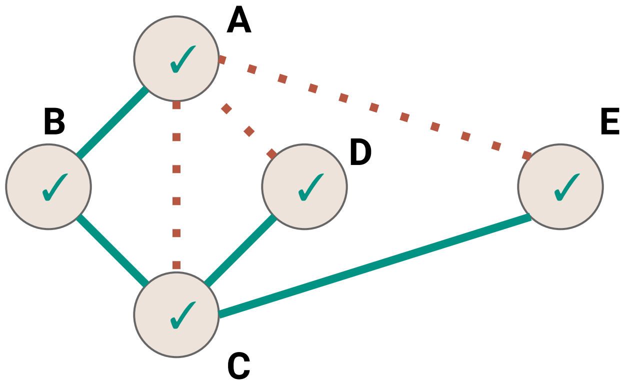
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



<u>Call Stack</u>

(→ edges to list)

DFS(G)
DFSOne(G,E)

Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

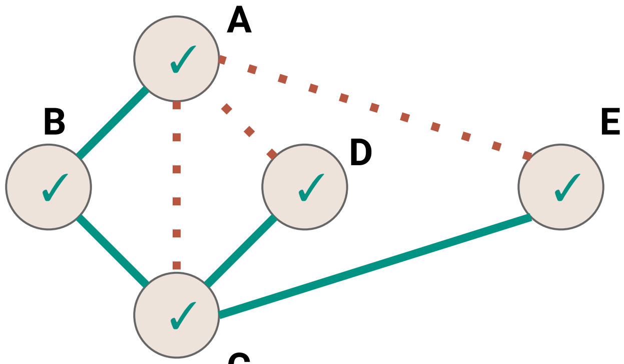
SPANNING

BACK

<u>Call Stack</u>

DFS(G)

(→ edges to list)



Detailed Example

UNEXPLORED

VISITED

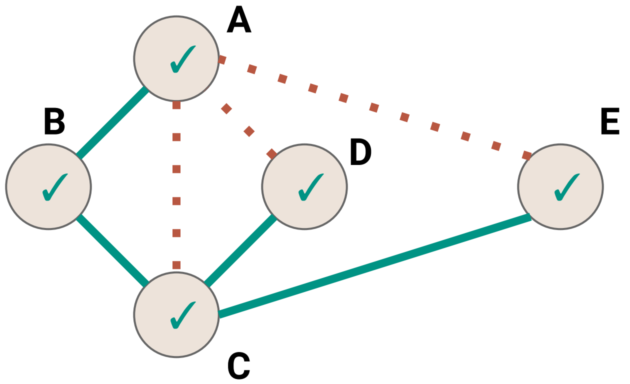
UNEXPLORED

SPANNING

BACK

Call Stack

(→ edges to list)



DFS vs Mazes

The DFS algorithm is like our stack-based maze solver (kind of)

- Mark each grid square with **VISITED** as we explore it
- Mark each path with **SPANNING** or **BACK**
- Only visit each vertex once (this differs from our maze search)

DFS vs Mazes

The DFS algorithm is like our stack-based maze solver (kind of)

- Mark each grid square with **VISITED** as we explore it
- Mark each path with **SPANNING** or **BACK**
- Only visit each vertex once (this differs from our maze search)
    - DFS will not necessarily find the shortest paths

Depth-First Search Complexity

What's the complexity?

DFS

```java
public void DFS(Graph graph) {
  for (Vertex v : graph.vertices) {
    v.setLabel(UNEXPLORED);
  }
  for (Edge e : graph.edges) {
    e.setLabel(UNEXPLORED);
  }
  for (Vertex v : graph.vertices) {
    if (v.label == UNEXPLORED) {
      DFSOne(graph, v);
    }
  }
}
```

DFS

```
public void DFS(Graph graph) {
  Θ(|V|)
  for (Edge e : graph.edges) {
    e.setLabel(UNEXPLORED);
  }
  for (Vertex v : graph.vertices) {
    if (v.label == UNEXPLORED) {
      DFSOne(graph, v);
    }
  }
}
```

DFS

```
public void DFS(Graph graph) {
  Θ(|V|)
  Θ(|E|)
  for (Vertex v : graph.vertices) {
    if (v.label == UNEXPLORED) {
      DFSOne(graph, v);
    }
  }
}
```

DFS

```
public void DFS(Graph graph) {
  Θ(|V|)
  Θ(|E|)
  for (Vertex v : graph.vertices) {
    if (v.label == UNEXPLORED) {
      Θ(???)
    }
  }
}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  Θ(1)
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Θ(1)
      if (w.label == UNEXPLORED) {
        Θ(1)
        Θ(???)
      } else {
        Θ(1)
      }
    }
}}
```

Depth-First Search Complexity

How many times do we call **`DFSOne`** *on each vertex?*

Depth-First Search Complexity

How many times do we call **`DFSOne`** *on each vertex?*

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

If **`v.label == VISITED`**, both **`DFS`**, and **`DFSOne`** skip it

Depth-First Search Complexity

How many times do we call `DFSOne` *on each vertex?*

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

If `v.label == VISITED`, both `DFS`, and `DFSOne` skip it

$O(|V|)$ calls to `DFSOne`

Depth-First Search Complexity

How many times do we call `DFSOne` *on each vertex?*

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

If `v.label == VISITED`, both `DFS`, and `DFSOne` skip it

$O(|V|)$ calls to `DFSOne`

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

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  Θ(1)
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Θ(1)
      if (w.label == UNEXPLORED) {
        Θ(1)
        Θ(???)
      } else {
        Θ(1)
      }
    }
}}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  Θ(1)
  for (Edge e : v.outEdges) {
    Θ(1)
  }
}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  Θ(1)
  Θ(deg(v))
}
```

Depth-First Search Complexity

How many times do we call **`DFSOne`** *on each vertex?*

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

If **`v.label == VISITED`**, both **`DFS`**, and **`DFSOne`** skip it

$O(|V|)$ calls to **`DFSOne`**

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

Depth-First Search Complexity

How many times do we call **`DFSOne`** *on each vertex?*

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

If **`v.label == VISITED`**, both **`DFS`**, and **`DFSOne`** skip it

$\boldsymbol{O}(|V|)$ calls to **`DFSOne`**

What's the runtime of **`DFSOne`** ***excluding the recursive calls?*** $\boldsymbol{O}(\mathbf{deg}(v))$

Depth-First Search Complexity

What is the sum over all calls to **`DFSOne`**?

Depth-First Search Complexity

What is the sum over all calls to **`DFSOne`**?

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

Depth-First Search Complexity

What is the sum over all calls to **`DFSOne`**?

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

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

Depth-First Search Complexity

What is the sum over all calls to **`DFSOne`**?

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

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

$$= O(2|E|)$$

Depth-First Search Complexity

What is the sum over all calls to **`DFSOne`**?

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

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

$$= O(2|E|)$$

$$= O(|E|)$$

Depth-First Search Complexity

In summary…

Depth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`**

Depth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`** $O(|V|)$

Depth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`** $O(|V|)$
2. Mark the edges **`UNVISITED`**

Depth-First Search Complexity

In summary…

1. Mark the vertices `UNVISITED` $O(|V|)$
2. Mark the edges `UNVISITED` $O(|E|)$

Depth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`** $O(|V|)$
2. Mark the edges **`UNVISITED`** $O(|E|)$
3. **`DFS`** vertex loop

Depth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`** — ***O***(|*V*|)
2. Mark the edges **`UNVISITED`** — ***O***(|*E*|)
3. **`DFS`** vertex loop — ***O***(|*V*|) **iterations**

Depth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`** — ***O***(|***V***|)
2. Mark the edges **`UNVISITED`** — ***O***(|***E***|)
3. **`DFS`** vertex loop — ***O***(|***V***|) **iterations**
4. All calls to **`DFSOne`**

Depth-First Search Complexity

In summary...

1. Mark the vertices `UNVISITED`	$O(\|V\|)$
2. Mark the edges `UNVISITED`	$O(\|E\|)$
3. `DFS` vertex loop	**$O(\|V\|)$ iterations**
4. All calls to `DFSOne`	**$O(\|E\|)$ total**

Depth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`**	$O(\|V\|)$
2. Mark the edges **`UNVISITED`**	$O(\|E\|)$
3. **`DFS`** vertex loop	**$O(\|V\|)$ iterations**
4. All calls to **`DFSOne`**	**$O(\|E\|)$ total**
	$O(\|V\| + \|E\|)$