CSE 250
Data Structures

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

Announcements and Feedback

- PA2 testing target on Autolab is open
- Practice midterms on course website

Edge List Summary

Graph

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

Vertex

label: T
node: LinkedListNode

Edge

label: T
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

Adjacency List Summary

Graph

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

Vertex

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

Edge

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

Adjacency List Summary

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

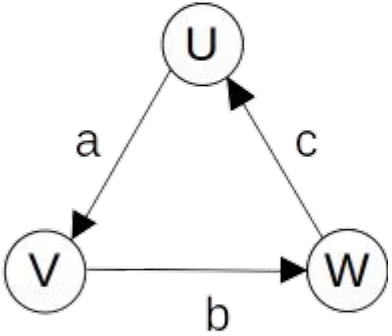
Now we already know what edges are incident without having to check them all

Adjacency Matrix

Destination

Origin

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



Adjacency Matrix Summary

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

Adjacency Matrix Summary

Just change a single entry of the matrix

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

Resize and copy the whole matrix

Check the row and column for that vertex

Check a single entry of the matrix

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

Adjacency Matrix Summary

Just change a single entry of the matrix

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

Resize and copy the whole matrix

Check the row and column for that vertex

Check a single entry of the matrix

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***:

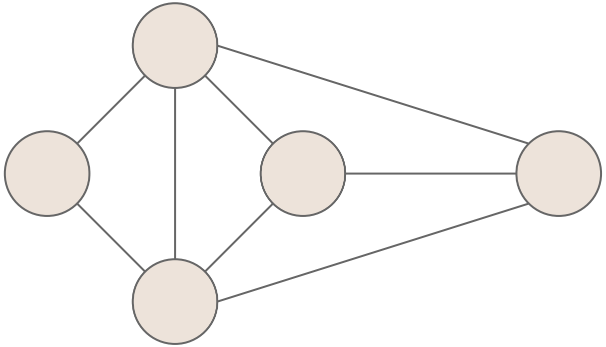
- Is vertex ***u*** **adjacent** to vertex ***v***?
- Is vertex ***u*** **connected** to vertex ***v*** via some path?
- Which vertices are **connected** to vertex ***v***?

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***?

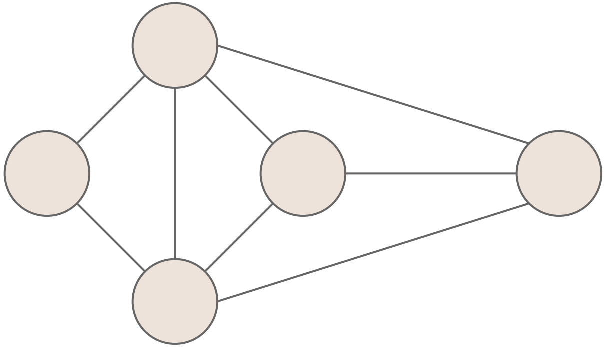
A few more definitions



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

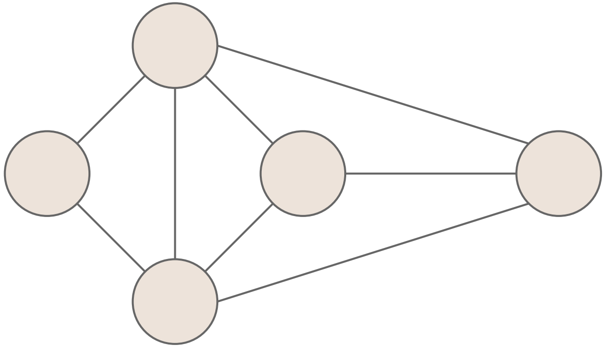


A few more definitions

A **subgraph, *S*,** of a graph ***G*** is a graph where:

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

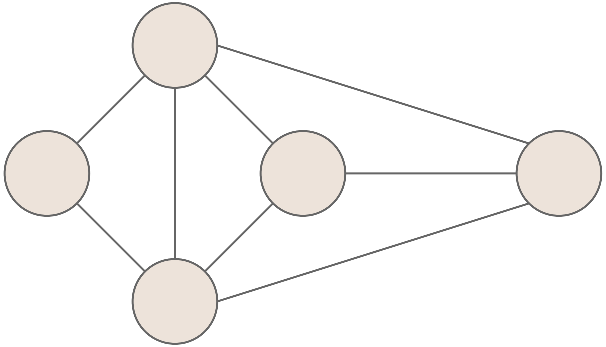
S's edges are a subset of ***G***'s edges



A **spanning subgraph** of ***G*…**

Is a subgraph of ***G***

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

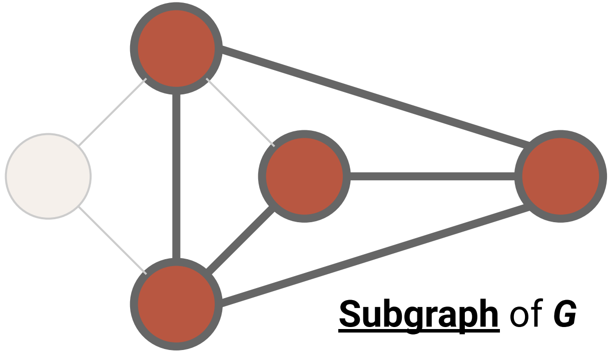


A few more definitions

A **subgraph, *S*,** of a graph ***G*** is a graph where:

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

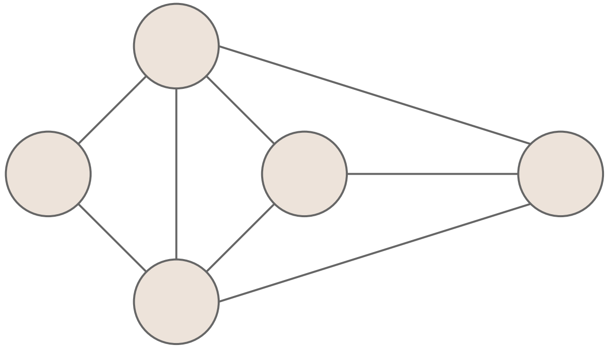
S's edges are a subset of ***G***'s edges



A **spanning subgraph** of ***G*...**

Is a subgraph of ***G***

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

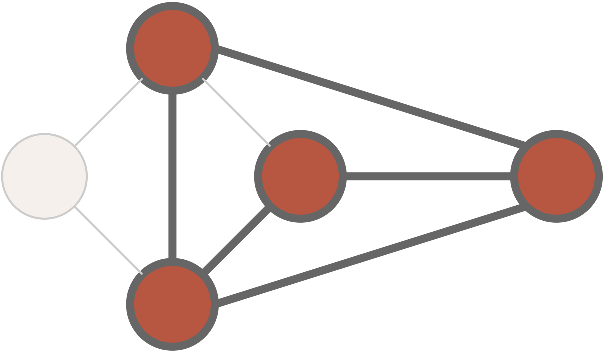


A few more definitions

A **subgraph, *S*,** of a graph ***G*** is a graph where:

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

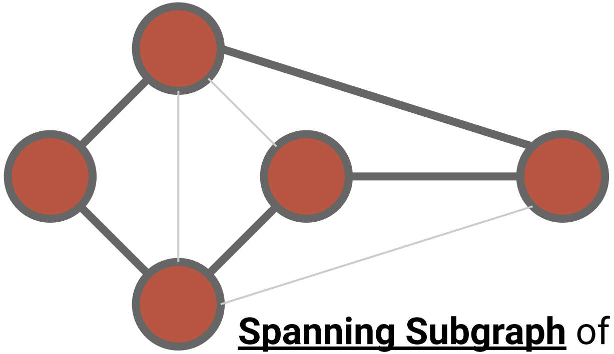
S's edges are a subset of ***G***'s edges



A **spanning subgraph** of ***G*...**

Is a subgraph of ***G***

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

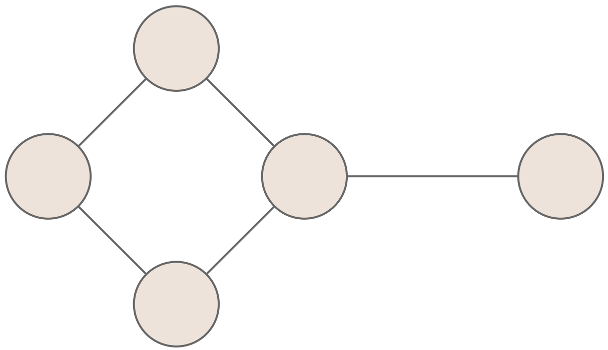


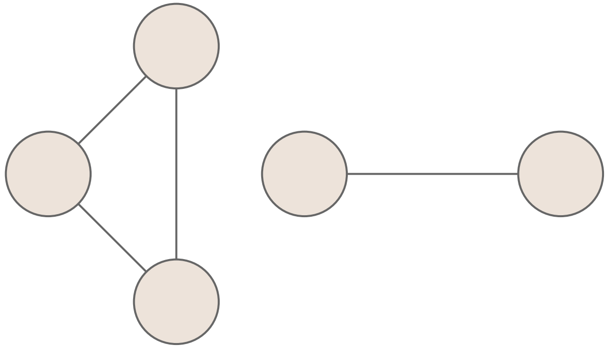
Spanning Subgraph of ***G***

A few more definitions

A graph is **connected**…

If there is a path between every pair of vertices

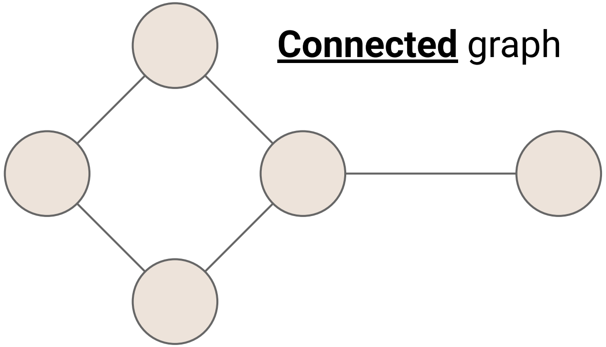




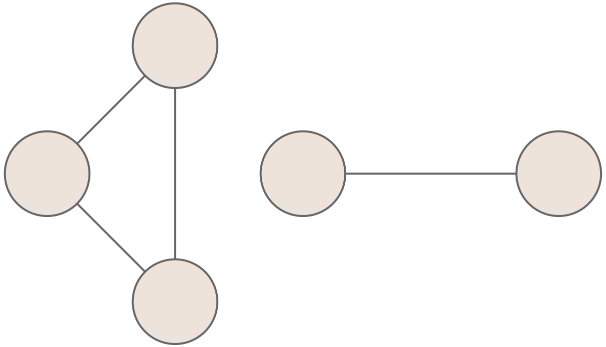
A few more definitions

A graph is **connected**…

If there is a path between every pair of vertices



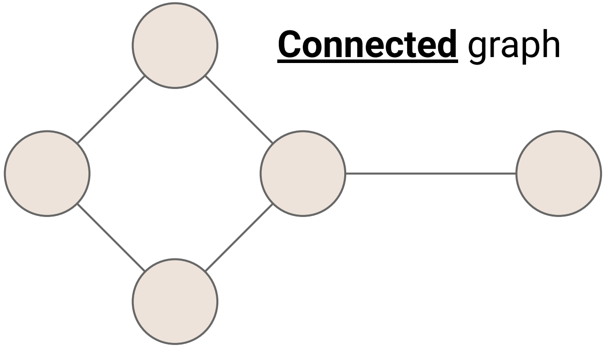
Connected graph



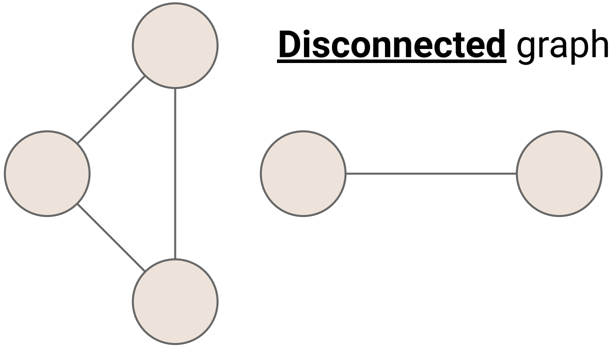
A few more definitions

A graph is **connected**…

If there is a path between every pair of vertices



Connected graph



Disconnected graph

A few more definitions

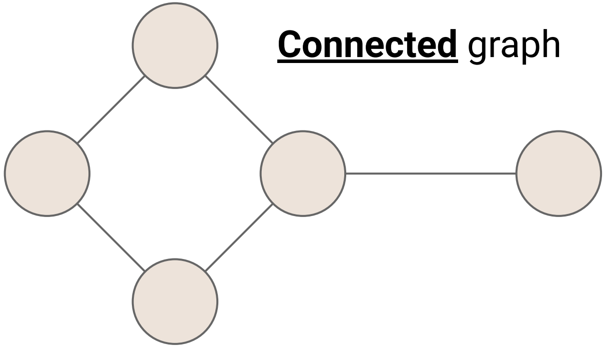
A graph is **connected**…

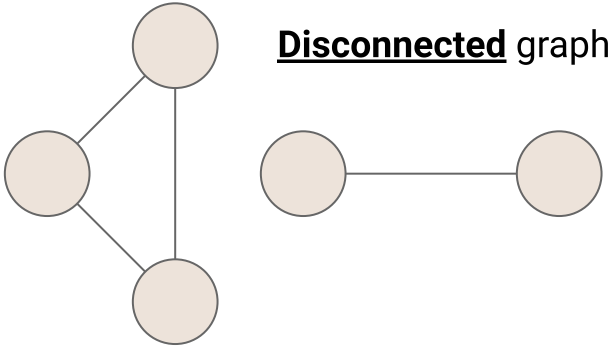
If there is a path between every pair of vertices

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

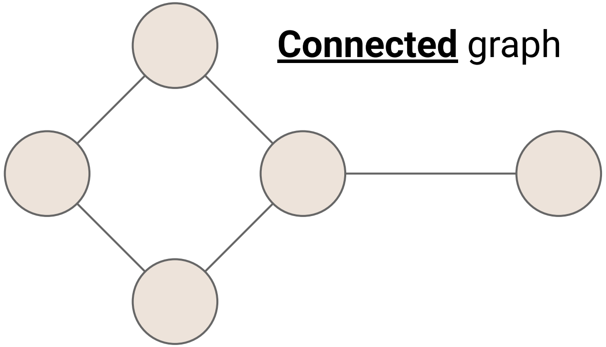
A graph is **connected**…

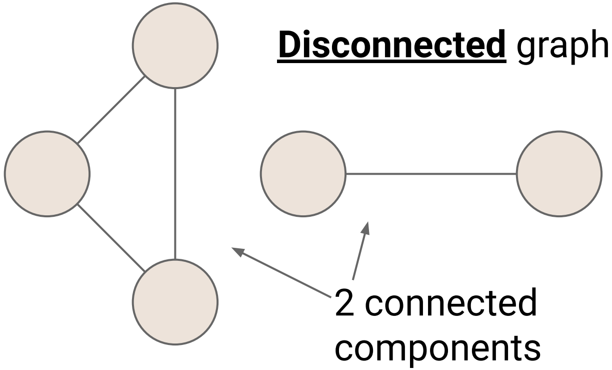
If there is a path between every pair of vertices

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 **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 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…

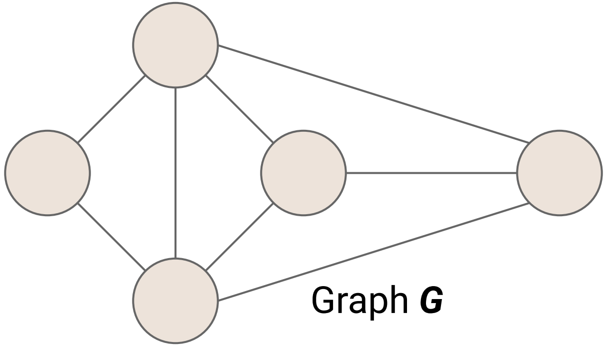
Every connected component is a tree

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



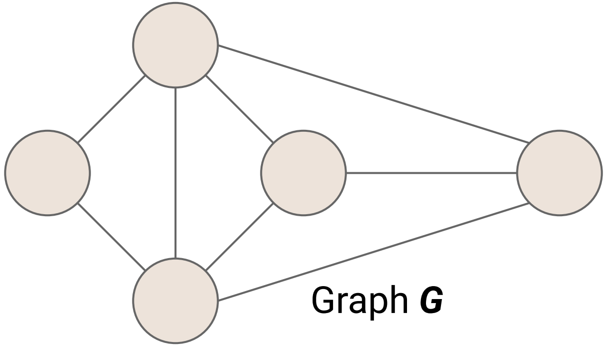
Graph ***G***

A few more definitions

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

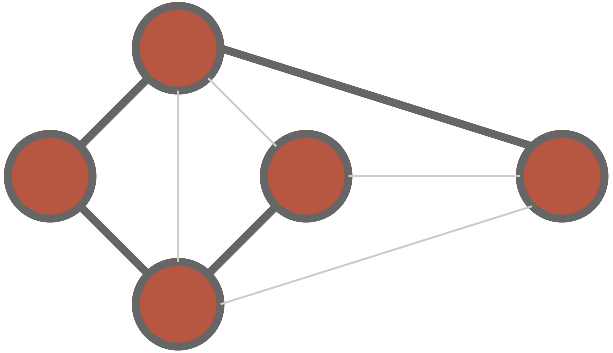
…Is a spanning subgraph that is a tree

…It is not unique unless the graph is a tree



Graph ***G***

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



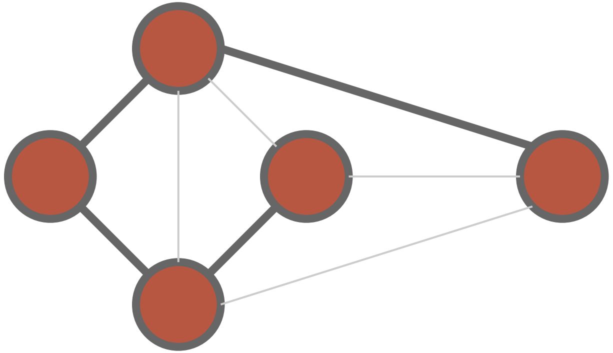
A few more definitions

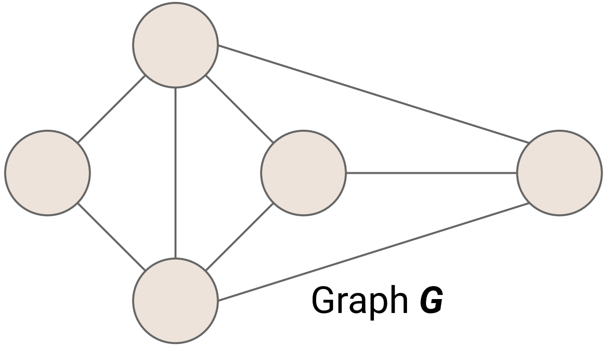
A **spanning tree** of a connected graph…

…Is a spanning subgraph that is a tree

…It is not unique unless the graph is a tree

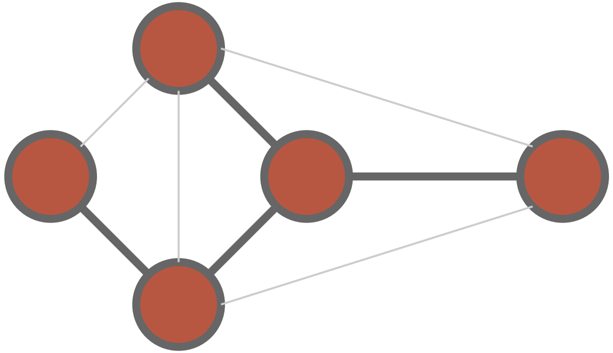
A **Spanning Tree** of ***G***





Graph ***G***

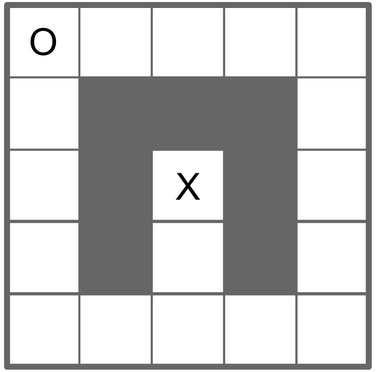
Another **Spanning Tree** of ***G***



Now back to the question…Connectivity

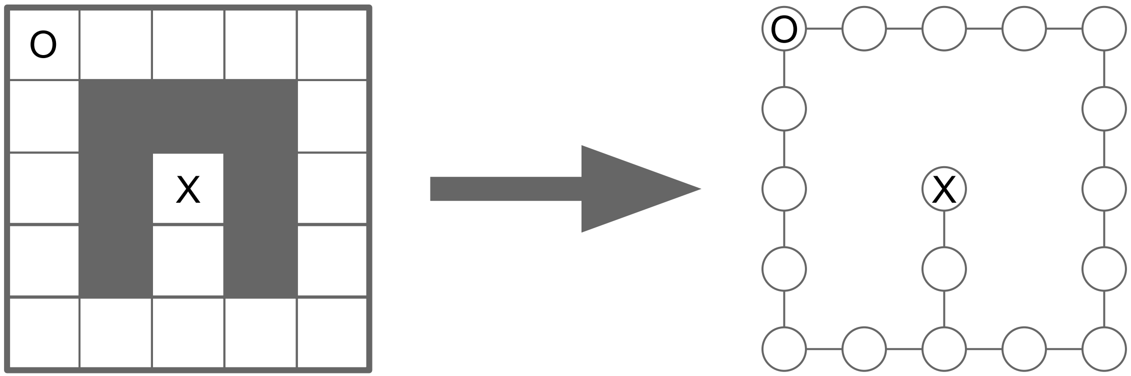
Back to Mazes

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

Primary Goals

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

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

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

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
- Complete in time $O(|V| + |E|)$

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:

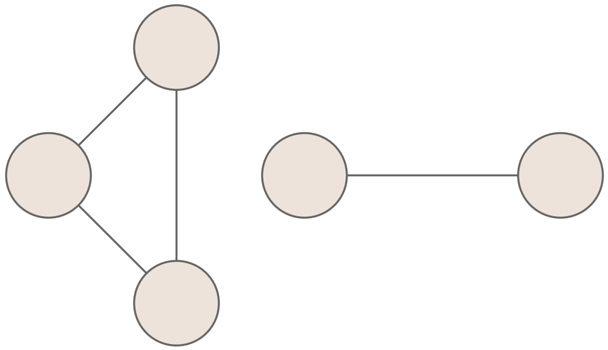
- Spanning Edge: Part of the spanning tree
- Back Edge: Part of a cycle

DFSOne

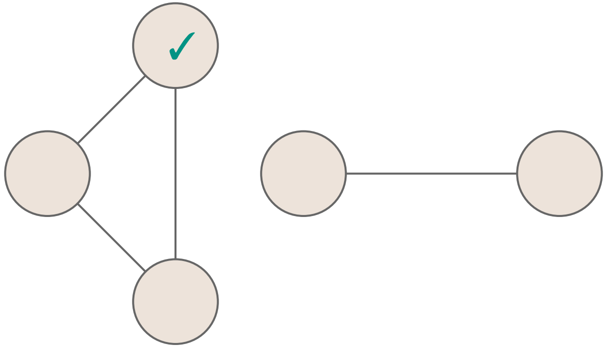
Input: Graph $G = (V,E)$, start vertex $v \in V$

Output: Label every edge in v's connected component

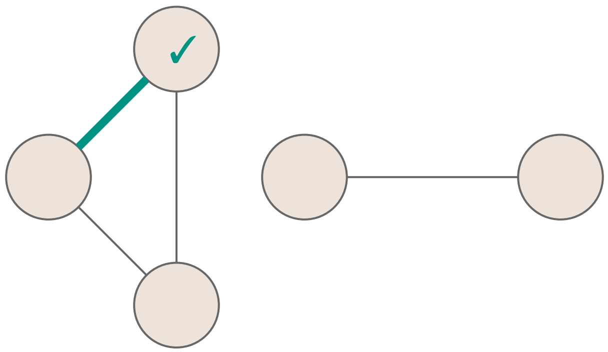
Depth-First Search



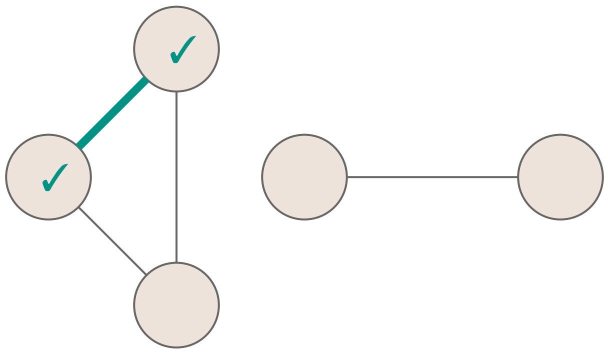
Depth-First Search



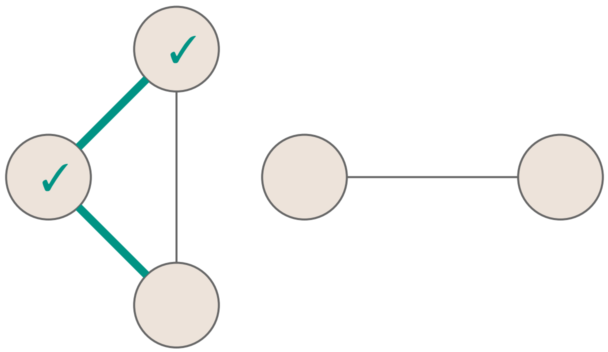
Depth-First Search



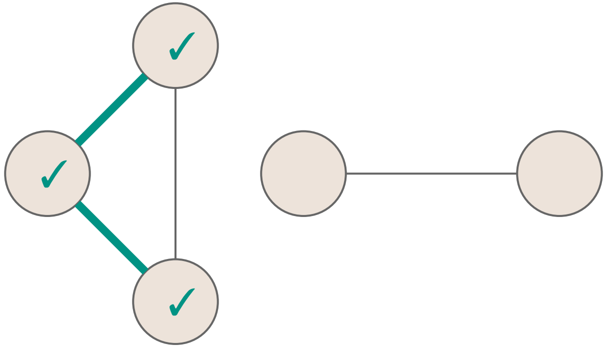
Depth-First Search



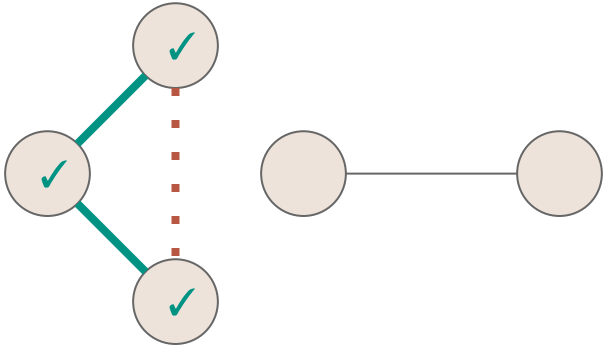
Depth-First Search



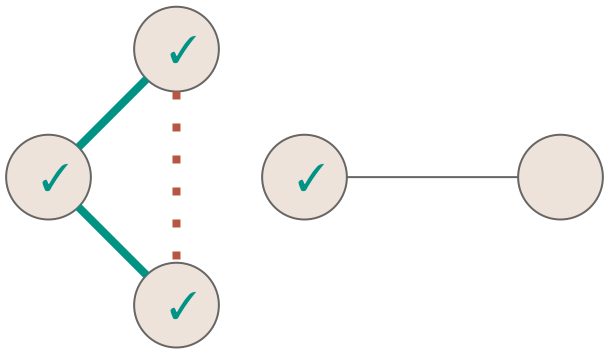
Depth-First Search



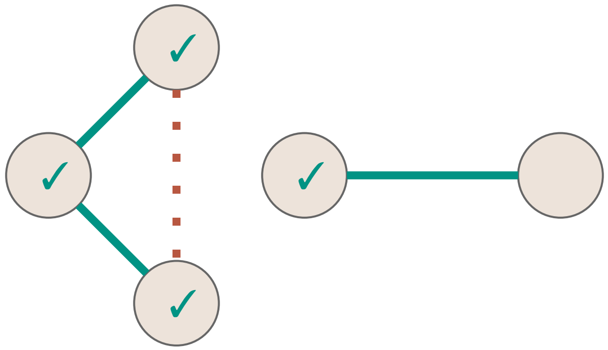
Depth-First Search



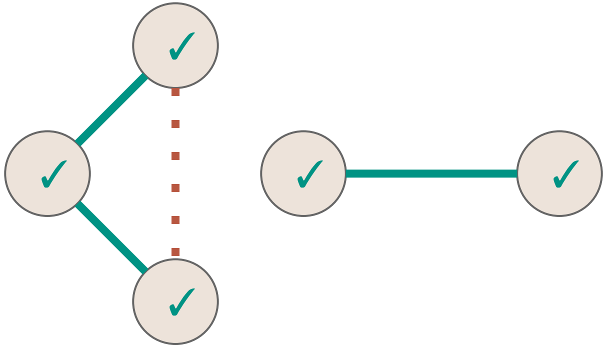
Depth-First Search



Depth-First Search



Depth-First Search



DFS

```
object VertexLabel extends Enumeration
  { val UNEXPLORED, VISITED = Value }

object EdgeLabel extends Enumeration
  { val UNEXPLORED, SPANNING, BACK = Value }

def DFS(graph: Graph[VertexLabel.Value, EdgeLabel.Value]) {
  for(v <- graph.vertices) { v.setLabel(VertexLabel.UNEXPLORED) }
  for(e <- graph.edges)    { e.setLabel(EdgeLabel.UNEXPLORED) }
  for(v <- graph.vertices) {
    if(v.label == VertexLabel.UNEXPLORED){
      DFSOne(graph, v)
    }
  }
}
```

DFSOne

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  v.setLabel(VertexLabel.VISITED)

  for(e <- v.incident) {
    if(e.label == EdgeLabel.UNEXPLORED){
      val w = e.getOpposite(v)
      if(w.label == VertexLabel.UNEXPLORED){
        e.setLabel(EdgeLabel.SPANNING)
        DFSOne(graph, w)
      } else {
        e.setLabel(EdgeLabel.BACK)
      }
    }
  }
}
```

DFSOne

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  v.setLabel(VertexLabel.VISITED)

  for(e <- v.incident) {
    if(e.label == EdgeLabel.UNEXPLORED){
      val w = e.getOpposite(v)
      if(w.label == VertexLabel.UNEXPLORED){
        e.setLabel(EdgeLabel.SPANNING)
        DFSOne(graph, w)
      } else {
        e.setLabel(EdgeLabel.BACK)
      }
    }
  }
}
```

If the edge is unexplored, explore it

DFSOne

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  v.setLabel(VertexLabel.VISITED)

  for(e <- v.incident) {
    if(e.label == EdgeLabel.UNEXPLORED){
      val w = e.getOpposite(v)
      if(w.label == VertexLabel.UNEXPLORED){
        e.setLabel(EdgeLabel.SPANNING)
        DFSOne(graph, w)
      } else {
        e.setLabel(EdgeLabel.BACK)
      }
    }
  }
}
```

If the edge is unexplored, explore it

If the other endpoint is unexplored, this is a spanning edge, explore that vertex

DFSOne

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  v.setLabel(VertexLabel.VISITED)

  for(e <- v.incident) {
    if(e.label == EdgeLabel.UNEXPLORED){
      val w = e.getOpposite(v)
      if(w.label == VertexLabel.UNEXPLORED){
        e.setLabel(EdgeLabel.SPANNING)
        DFSOne(graph, w)
      } else {
        e.setLabel(EdgeLabel.BACK)
      }
    }
  }
}
```

If the edge is unexplored, explore it

If the other endpoint is unexplored, this is a spanning edge, explore that vertex

If the other endpoint is already explored, this is a back edge

Detailed Example

UNEXPLORED

VISITED

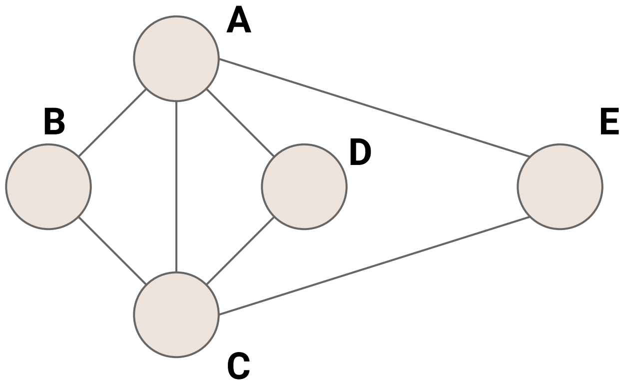
UNEXPLORED

SPANNING

BACK

Call Stack

(→ edges to list)



Detailed Example

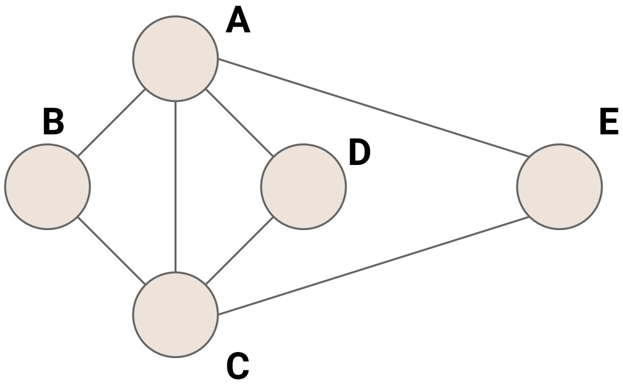
UNEXPLORED

✓ VISITED

UNEXPLORED

SPANNING

BACK



Call Stack

DFS(G)

(→ edges to list)

Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

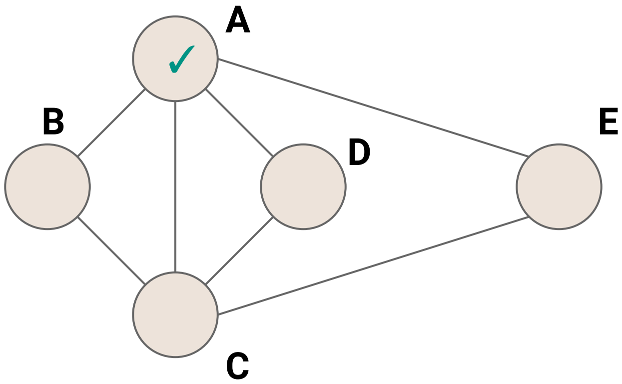
BACK

<u>Call Stack</u>

DFS(G)

DFSOne(G,A)

(→ edges to list)



Detailed Example

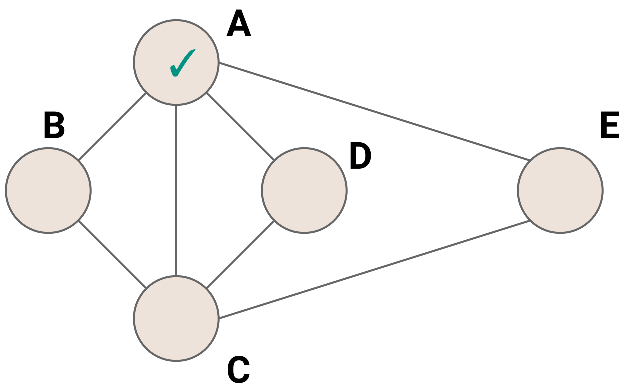
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

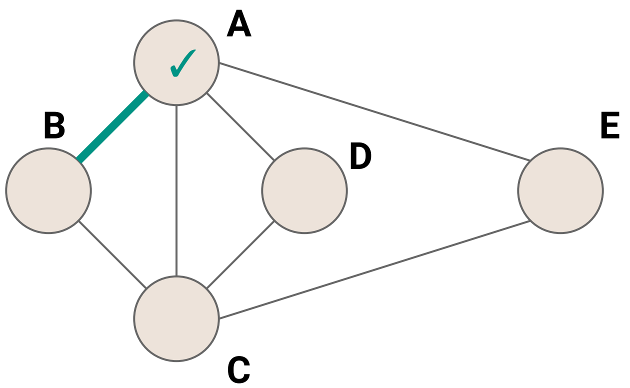
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

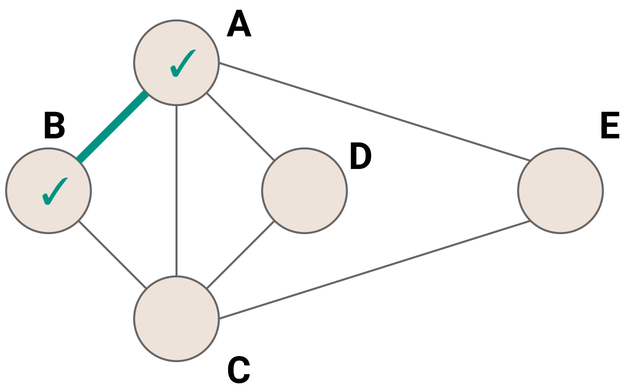
UNEXPLORED

VISITED

UNEXPLORED

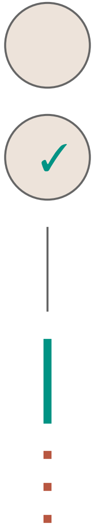
SPANNING

BACK



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

Detailed Example



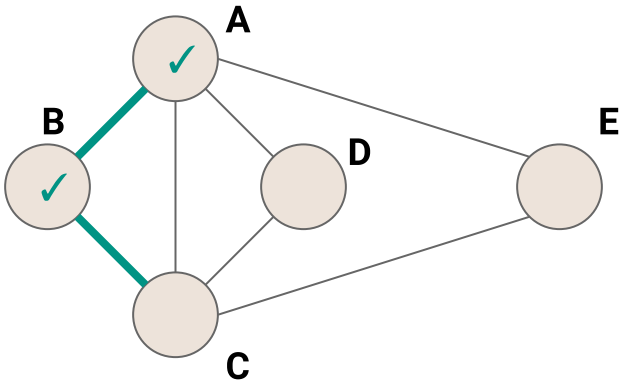
UNEXPLORED

VISITED

UNEXPLORED

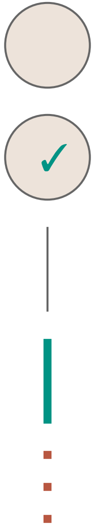
SPANNING

BACK



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

Detailed Example



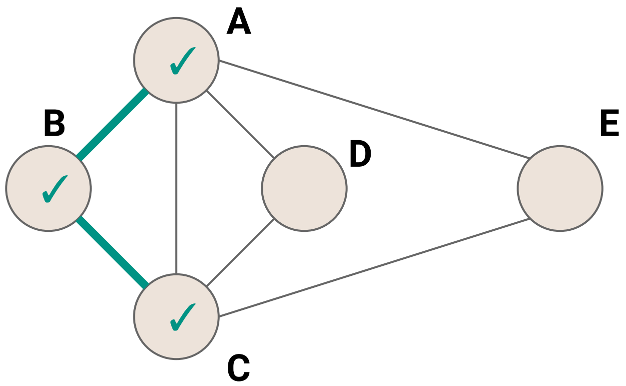
UNEXPLORED

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

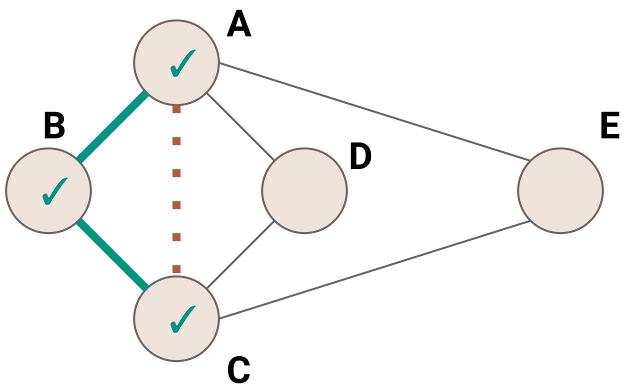
UNEXPLORED

VISITED

UNEXPLORED

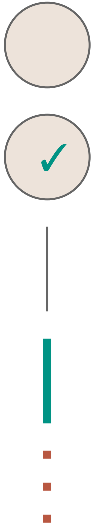
SPANNING

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



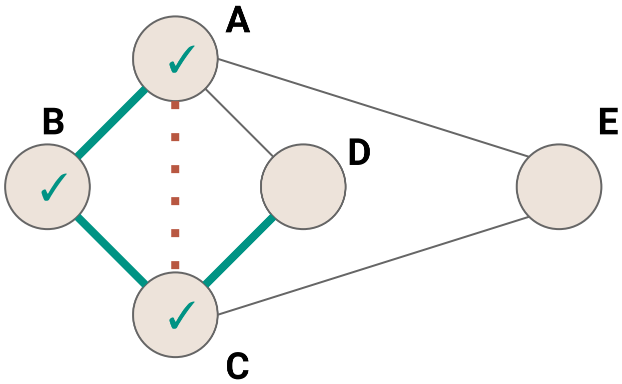
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

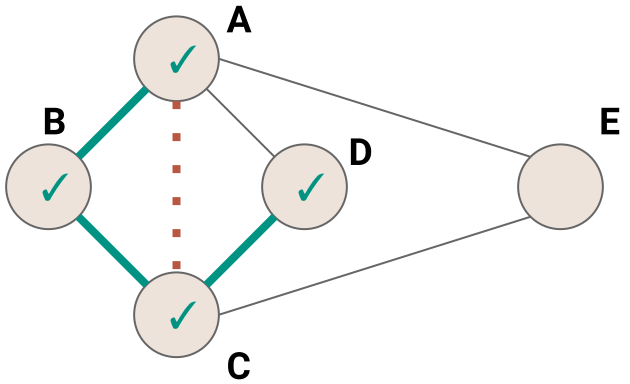
UNEXPLORED

VISITED

UNEXPLORED

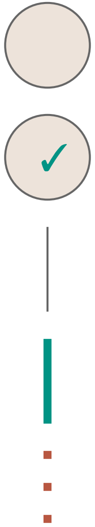
SPANNING

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)
`DFSOne(G,D)`	(→ A, C)

Detailed Example



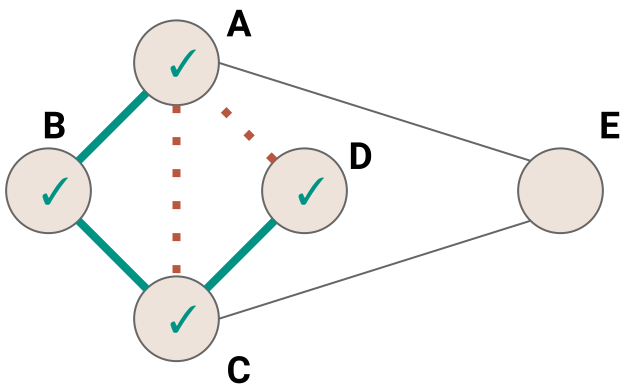
UNEXPLORED

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

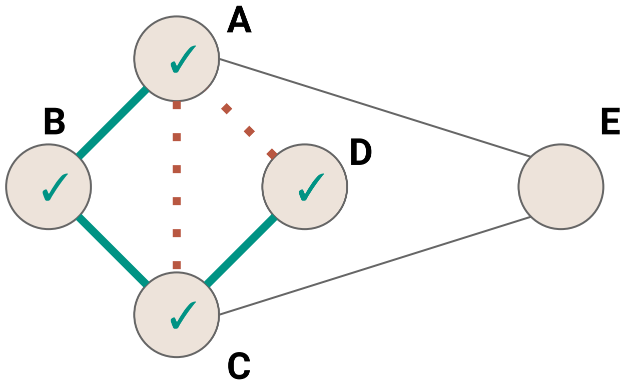
UNEXPLORED

VISITED

UNEXPLORED

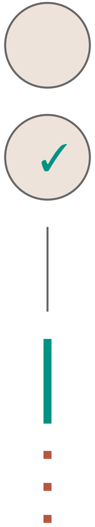
SPANNING

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



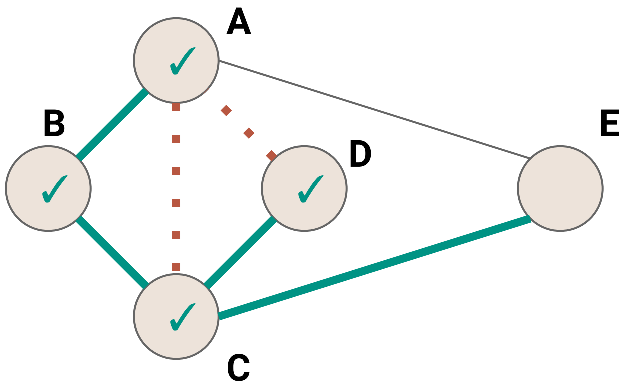
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



<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

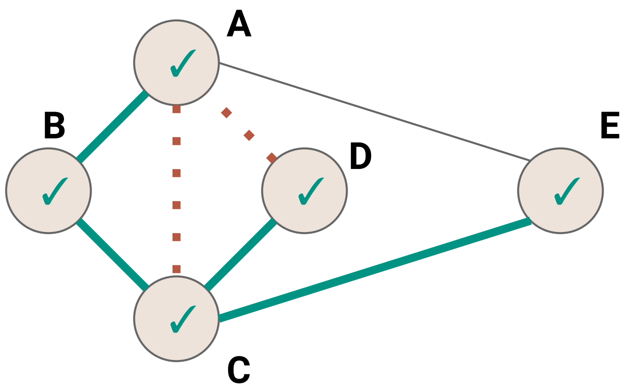
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

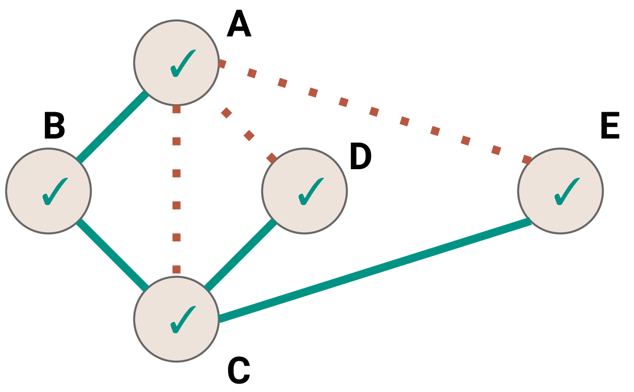
UNEXPLORED

✓ VISITED

UNEXPLORED

SPANNING

BACK



<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,E)	(→ A, C)

Detailed Example

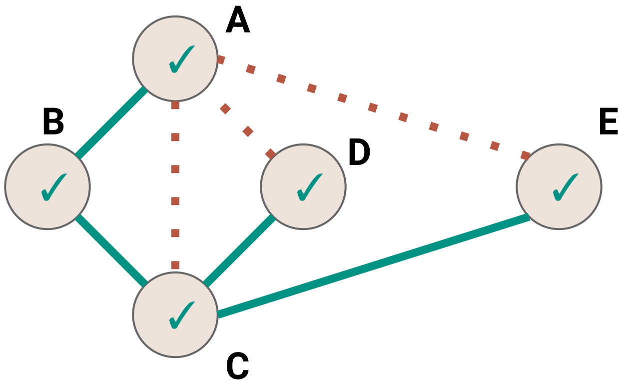
UNEXPLORED

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

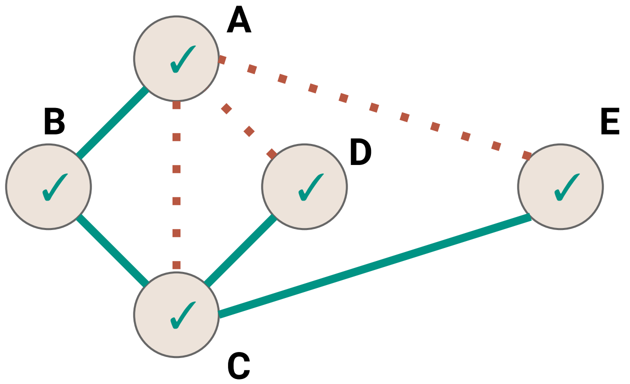
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack (→ edges to list)

DFS(G)
DFSOne(G,A) (→ B, C, D)
DFSOne(G,B) (→ A, C)

Detailed Example

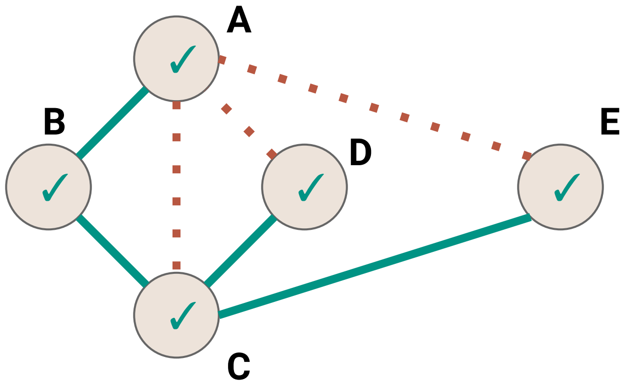
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

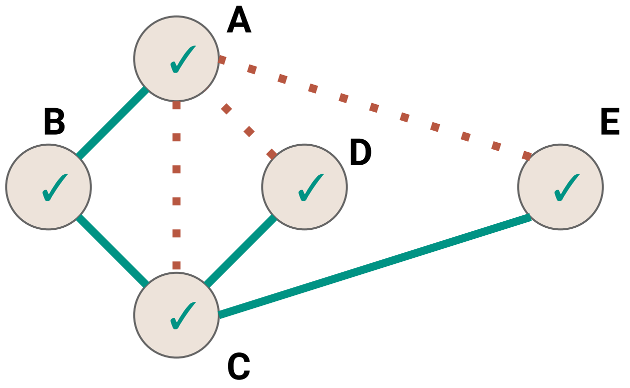
SPANNING

BACK

<u>Call Stack</u>

DFS(G)
DFSOne(G,B)

(→ edges to list)



Detailed Example

UNEXPLORED

VISITED

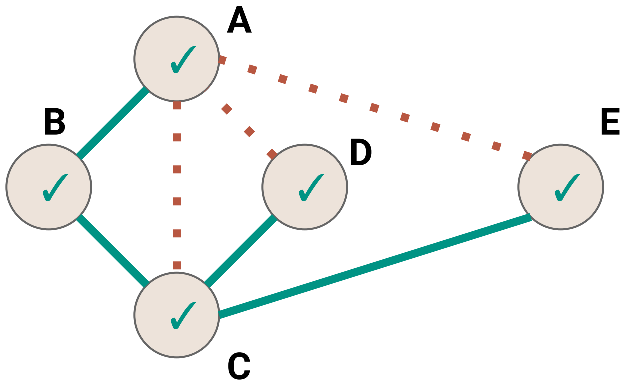
UNEXPLORED

SPANNING

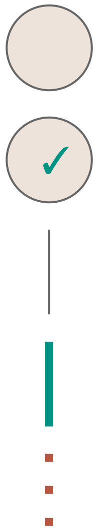
BACK

Call Stack
DFS(G)
DFSOne(G,C)

(→ edges to list)



Detailed Example



UNEXPLORED

VISITED

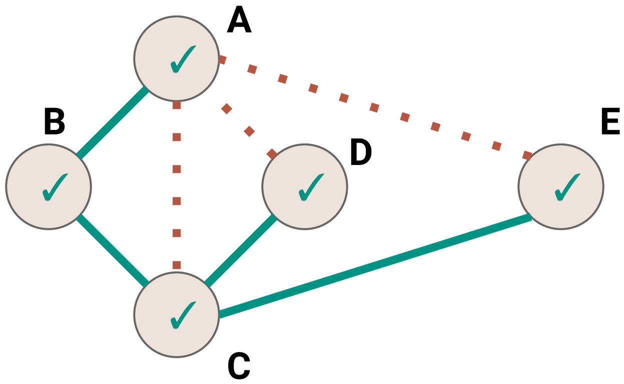
UNEXPLORED

SPANNING

BACK

<u>Call Stack</u>
DFS(G)
DFSOne(G,D)

(→ edges to list)



Detailed Example

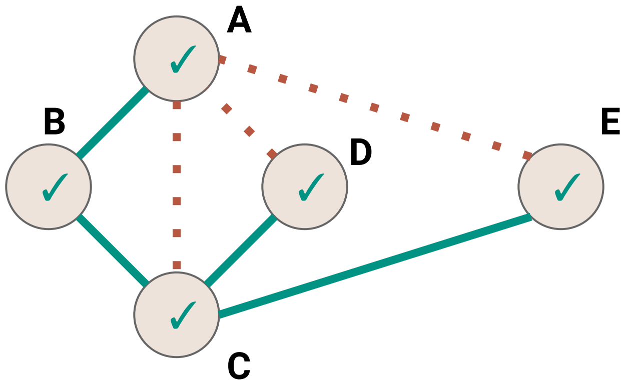
UNEXPLORED

✓ VISITED

UNEXPLORED

SPANNING

BACK



Call Stack

DFS(G)

DFSOne(G,E)

(→ edges to list)

Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

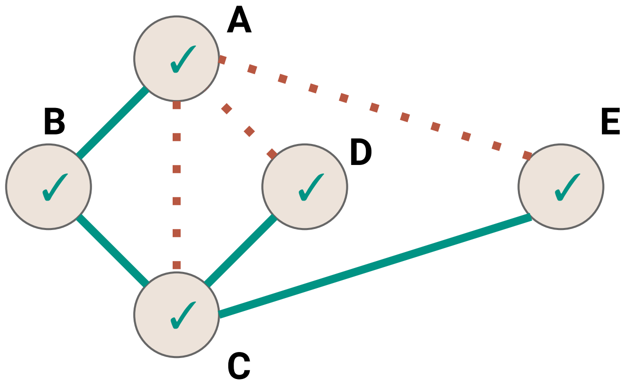
SPANNING

BACK

<u>Call Stack</u>

DFS(G)

<u>(→ edges to list)</u>



Detailed Example

UNEXPLORED

VISITED

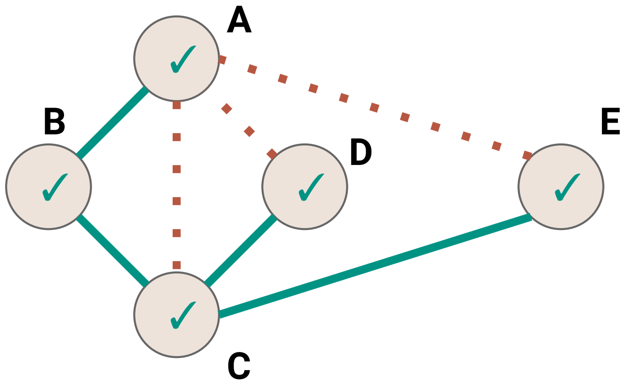
UNEXPLORED

SPANNING

BACK

Call Stack

(→ edges to list)



DFS vs Mazes

The DFS algorithm is like our stack-based maze solver

- Mark each grid square with **VISITED** as we explore it
- Mark each path with **SPANNING** or **BACK**
- Only visit each vertex once

DFS vs Mazes

The DFS algorithm is like our stack-based maze solver

- Mark each grid square with **VISITED** as we explore it
- Mark each path with **SPANNING** or **BACK**
- Only visit each vertex once
    - DFS will not necessarily find the shortest paths

Depth-First Search Complexity

What's the complexity?

Complexity

```
def DFS(graph: Graph[VertexLabel.Value, EdgeLabel.Value])
{
  for(v <- graph.vertices) { v.setLabel(VertexLabel.UNEXPLORED) }
  for(e <- graph.edges)    { e.setLabel(EdgeLabel.UNEXPLORED) }
  for(v <- graph.vertices) {
    if(v.label == VertexLabel.UNEXPLORED){
      DFSOne(graph, v)
    }
  }
}
```

Complexity

```
def DFS(graph: Graph[VertexLabel.Value, EdgeLabel.Value])
{
  /* O(|V|) */
  for(e <- graph.edges)     { e.setLabel(EdgeLabel.UNEXPLORED) }
  for(v <- graph.vertices) {
    if(v.label == VertexLabel.UNEXPLORED){
      DFSOne(graph, v)
    }
  }
}
```

Complexity

```
def DFS(graph: Graph[VertexLabel.Value, EdgeLabel.Value])
{
  /* O(|V|) */
  /* O(|E|) */
  for(v <- graph.vertices) {
    if(v.label == VertexLabel.UNEXPLORED){
      DFSOne(graph, v)
    }
  }
}
```

Complexity

```
def DFS(graph: Graph[VertexLabel.Value, EdgeLabel.Value])
{
  /* O(|V|) */
  /* O(|E|) */
  /* O(|V|) times */ {
    if(v.label == VertexLabel.UNEXPLORED){
      DFSOne(graph, v)
    }
  }
}
```

Complexity

```
def DFS(graph: Graph[VertexLabel.Value, EdgeLabel.Value])
{
  /* O(|V|) */
  /* O(|E|) */
  /* O(|V|) times */ {
    if(v.label == VertexLabel.UNEXPLORED){
      /* ??? */
    }
  }
}
```

Complexity

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  v.setLabel(VertexLabel.VISITED)
  for(e <- v.incident) {
    if(e.label == EdgeLabel.UNEXPLORED){
      val w = e.getOpposite(v)
      if(w.label == VertexLabel.UNEXPLORED){
        e.setLabel(EdgeLabel.SPANNING)
        DFSOne(graph, w)
      } else {
        e.setLabel(EdgeLabel.BACK)
      }
    }
  }
}
```

Complexity

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  /* O(1) */
  for(e <- v.incident) {
    if(e.label == EdgeLabel.UNEXPLORED){
      val w = e.getOpposite(v)
      if(w.label == VertexLabel.UNEXPLORED){
        e.setLabel(EdgeLabel.SPANNING)
        DFSOne(graph, w)
      } else {
        e.setLabel(EdgeLabel.BACK)
      }
    }
  }
}
```

Complexity

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  /* O(1) */
  /* O(deg(v)) times */ {
    if(e.label == EdgeLabel.UNEXPLORED){
      val w = e.getOpposite(v)
      if(w.label == VertexLabel.UNEXPLORED){
        e.setLabel(EdgeLabel.SPANNING)
        DFSOne(graph, w)
      } else {
        e.setLabel(EdgeLabel.BACK)
      }
    }
  }
}
```

Complexity

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  /* O(1) */
  /* O(deg(v)) times */ {
    /* O(1) */ {
      /* O(1) */
      /* O(1) */ {
        /* O(1) */
        DFSOne(graph, w)
      } else {
        /* O(1) */
      }
    }
  }
}
```

Complexity

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  /* O(1) */
  /* O(deg(v)) times */ {
    /* O(1) */ {
      /* O(1) */
      /* O(1) */ {
        /* O(1) */
        /* ??? */
      } else {
        /* O(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?***

Complexity

```
def DFSOne(graph: Graph[…], v: Graph[…]#Vertex) {
  /* O(1) */
  /* O(deg(v)) times */ {
    /* O(1) */ {
      /* O(1) */
      /* O(1) */ {
        /* O(1) */
        /* ??? */
      } else {
        /* O(1) */
      }
    }
  }
}
```

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

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

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

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`	$O(\lvert V \rvert)$
2.	Mark the edges `UNVISITED`	$O(\lvert E \rvert)$
3.	`DFS` vertex loop	$O(\lvert V \rvert)$

Depth-First Search Complexity

In summary...

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

Depth-First Search Complexity

In summary…

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