CSE 250
Data Structures

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

Lec 20: Graph Traversals

Announcements

- PA2 released
  - Testing phase due Sunday 3/16
  - Implementation due Sunday 3/30
  - AutoLab opens tonight

1 week for testing

Spring Break

1 week for implemenation

Warm Up Question

sorted array `a = [1,1,2,2,2,2,5,5,...,x,x,y,y]`, total size = n

```
int i = 0
while i < n:
  val = a[i], count = 0
  while a[i] == val and i < n:
    count++, i++
  print("Found %i elems of value %i", count, val)
```

1. How many iterations does this loop run?
2. How many iterations does this loop run?
3. Total runtime?

Warm Up Question

sorted array `a = [1,1,2,2,2,2,5,5,...,x,x,y,y]`, total size = n

```
int i = 0
while i < n:
  val = a[i], count = 0
  while a[i] == val and i < n:
    count++, i++
  print("Found %i elems of value %i", count, val)
```

1. How many iterations does this loop run? $\Omega(1)$, $O(n)$
2. How many iterations does this loop run? $\Omega(1)$, $O(n)$
3. Total runtime? $\Theta(n)$

So…what do we do with our graphs?

Connectivity Problems

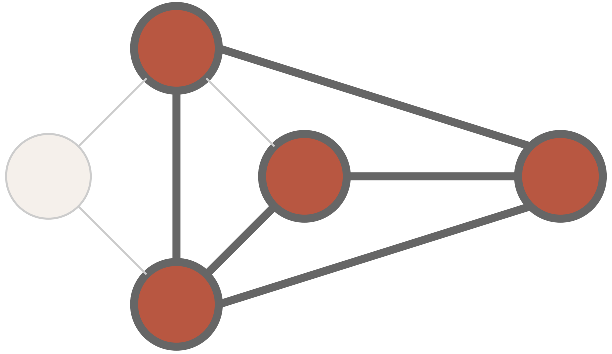
Given graph ***G***:

- 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

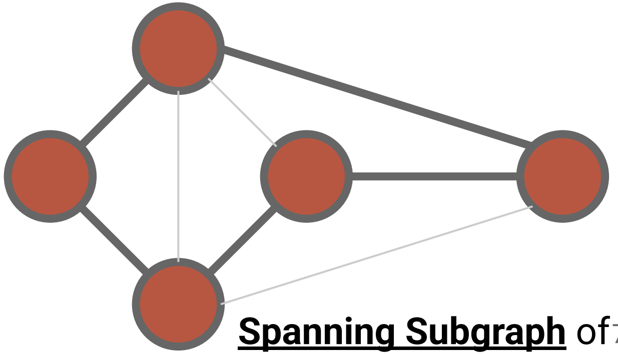
A **subgraph**, ***S***, of a graph ***G*** is a graph where:

- ***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

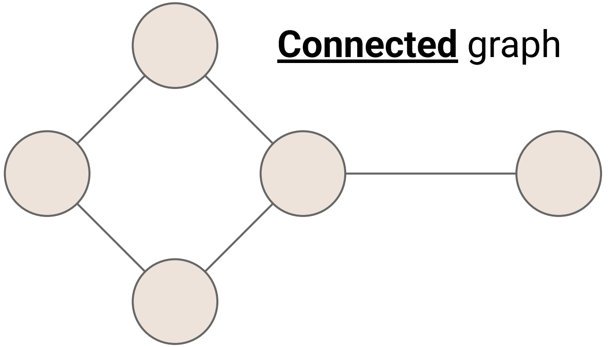
A graph is **connected**…

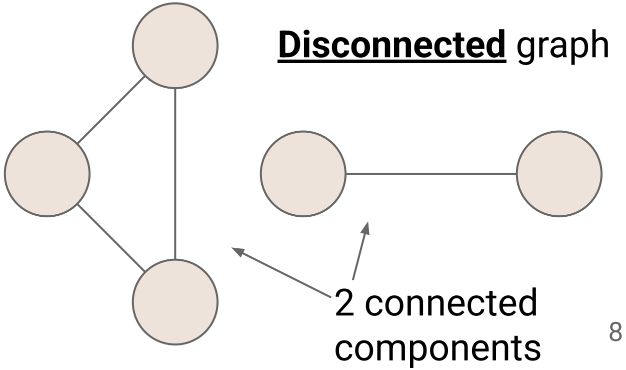
If there is a path between every pair of vertices

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

Every connected component is a tree

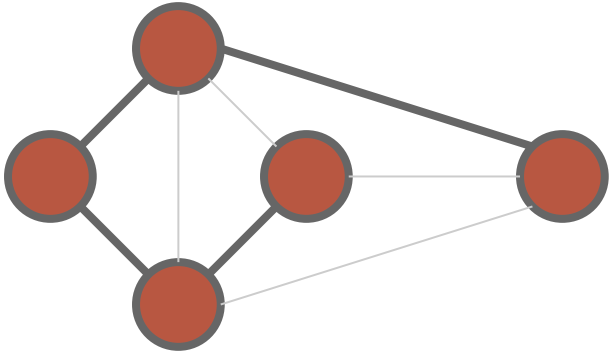
A few more definitions

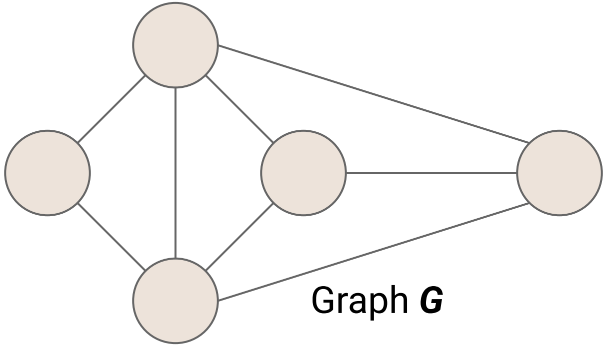
A **spanning tree** of a connected graph…

…Is a spanning subgraph that is a tree

…It is not unique unless the graph is a tree

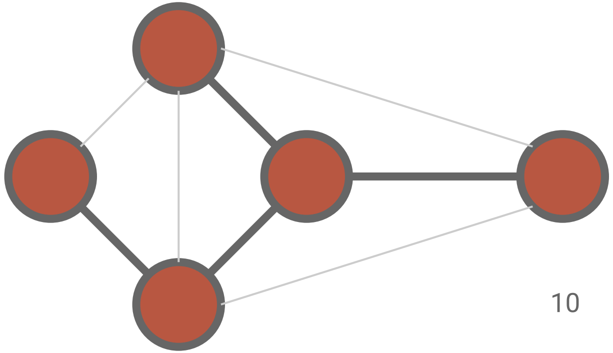
A **Spanning Tree** of ***G***





Graph ***G***

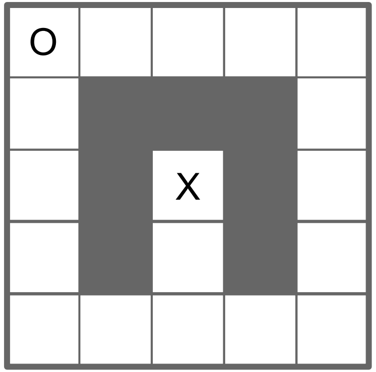
Another **Spanning Tree** of ***G***



Now back to the question…Connectivity

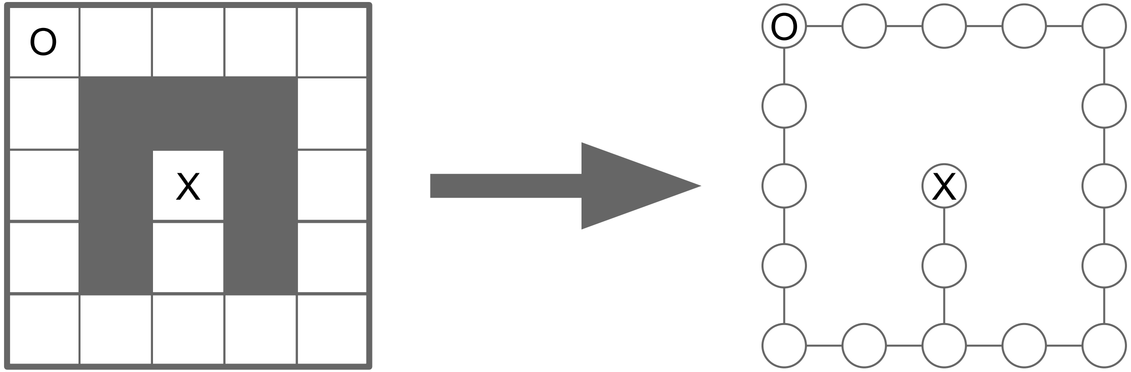
Back to Mazes

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 - kind of…)

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 - kind of…)

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 $\boldsymbol{G = (V,E)}$
- 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

<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

Depth-First Search

Primary Goals

- 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

Depth-First Search

Primary Goals

- 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:

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

Depth-First Search

DFS

Input: Graph $\boldsymbol{G = (V,E)}$

Output: Label every edge as:

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

DFSOne

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

Output: Label every edge in $\boldsymbol{v}$'s connected component

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

```
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

```
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

```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);
      }
    }
}}
```

Otherwise, we just found a cycle

Detailed Example

UNEXPLORED

VISITED

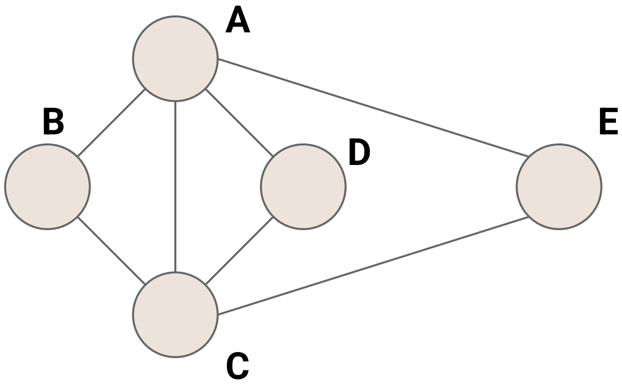
UNEXPLORED

SPANNING

BACK

Call Stack

(→ edges to list)



Detailed Example

UNEXPLORED

✓ VISITED

UNEXPLORED

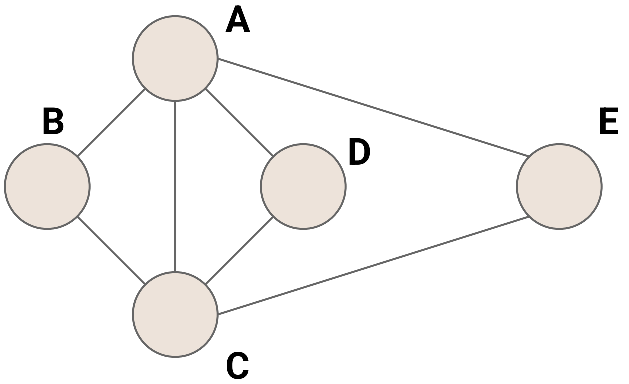
SPANNING

BACK

<u>Call Stack</u>

DFS(G)

(→ edges to list)



Detailed Example

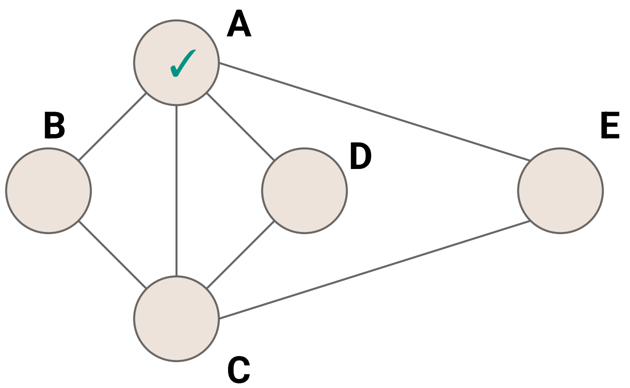
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



Call Stack

DFS(G)
DFSOne(G,A)

(→ edges to list)

Detailed Example

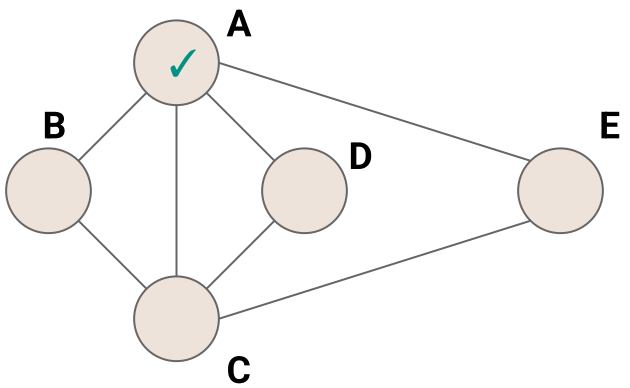
UNEXPLORED

VISITED

UNEXPLORED

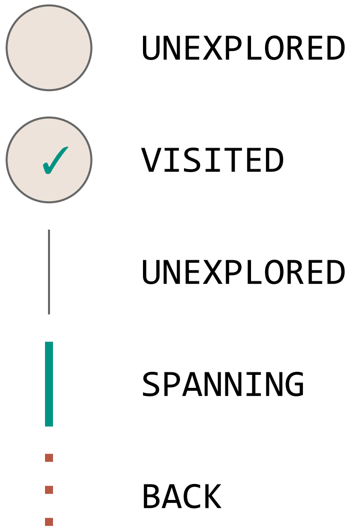
SPANNING

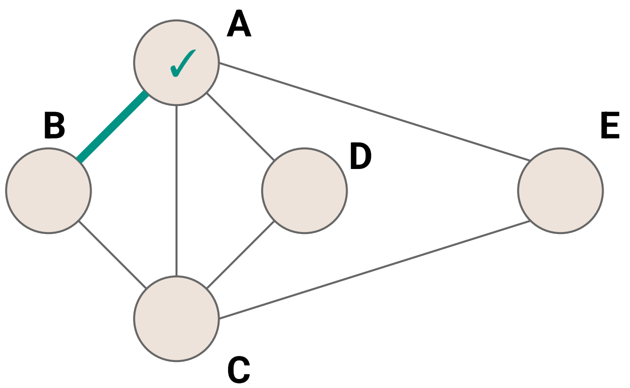
BACK



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

Detailed Example





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

(→ edges to list)
(→ B, C, D, E)

Detailed Example

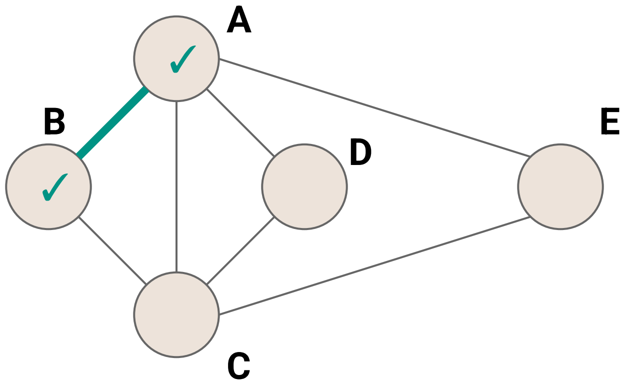
UNEXPLORED

VISITED

UNEXPLORED

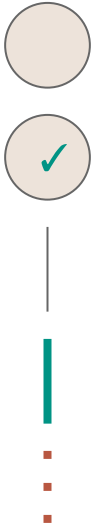
SPANNING

BACK



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

Detailed Example



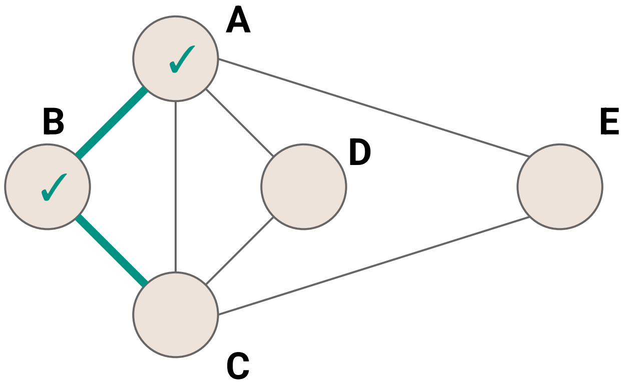
UNEXPLORED

VISITED

UNEXPLORED

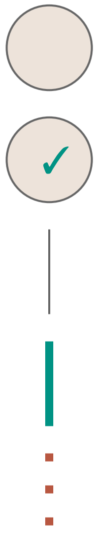
SPANNING

BACK



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

Detailed Example



UNEXPLORED

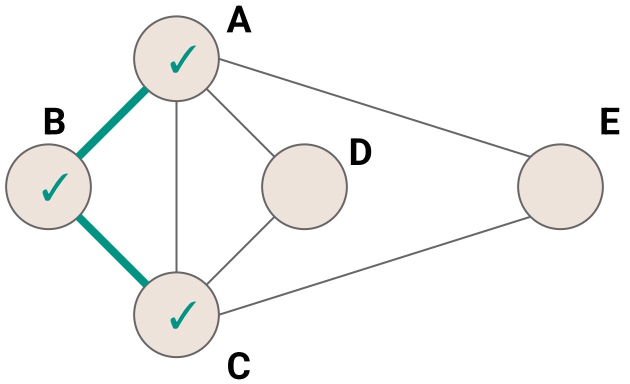
VISITED

UNEXPLORED

SPANNING

BACK

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



Detailed Example

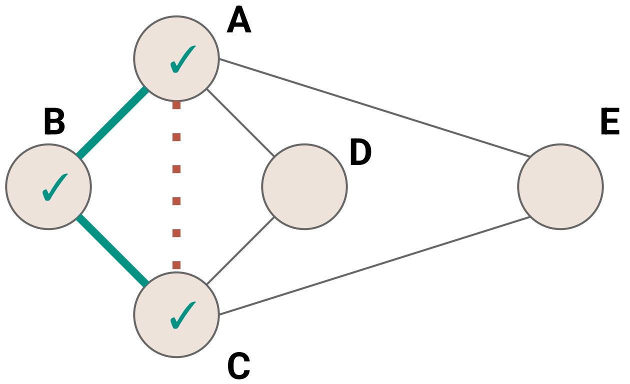
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

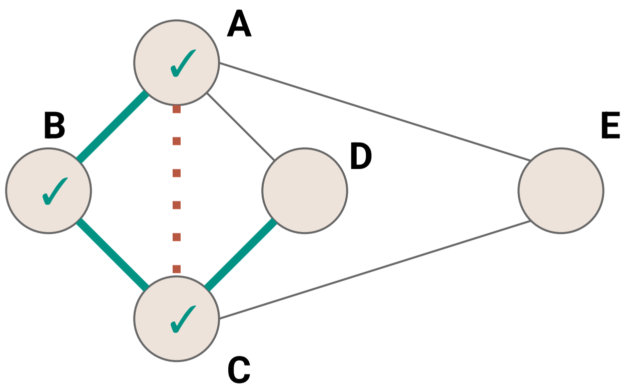
UNEXPLORED

✓ VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

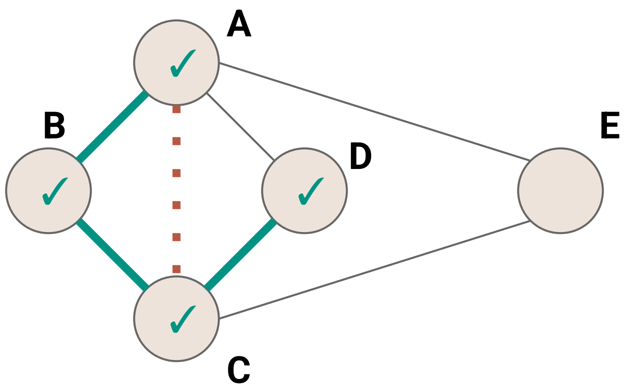
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

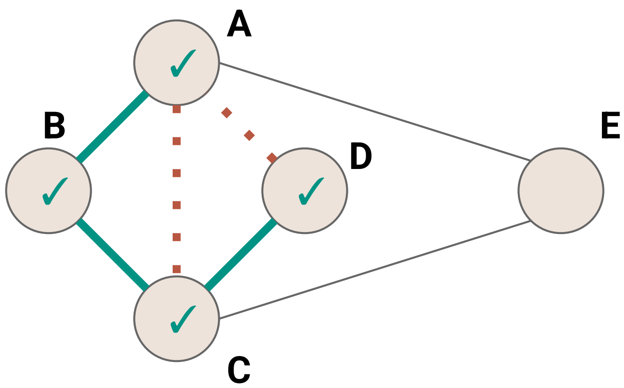
UNEXPLORED

VISITED

UNEXPLORED

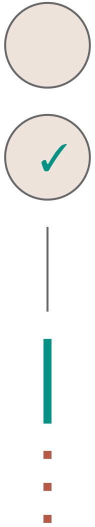
SPANNING

BACK



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

Detailed Example



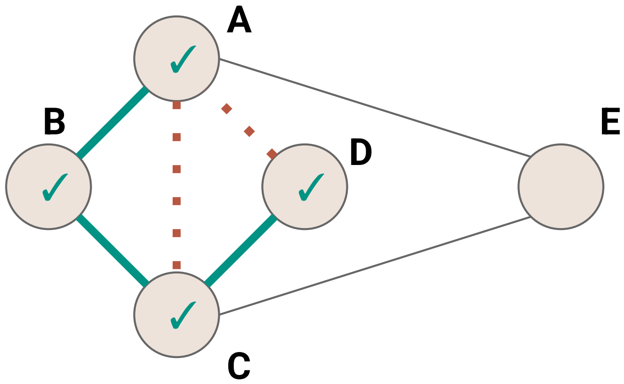
UNEXPLORED

VISITED

UNEXPLORED

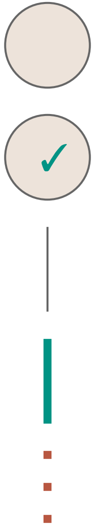
SPANNING

BACK



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

Detailed Example



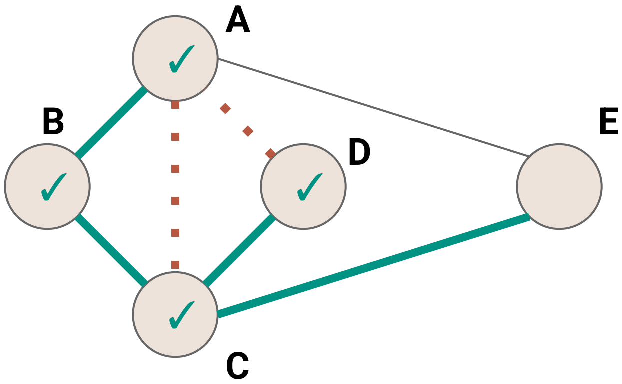
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

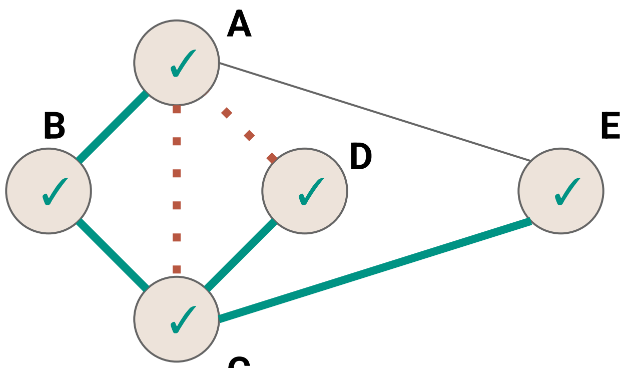
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

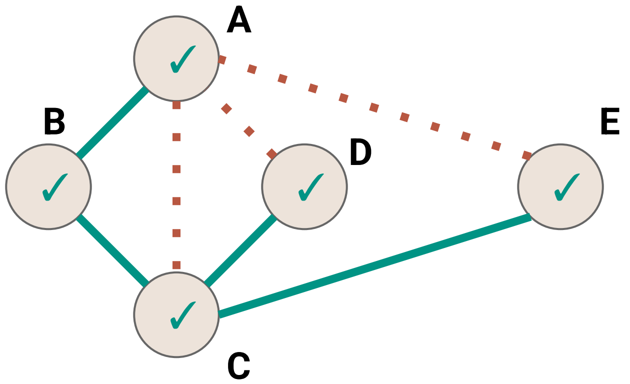
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

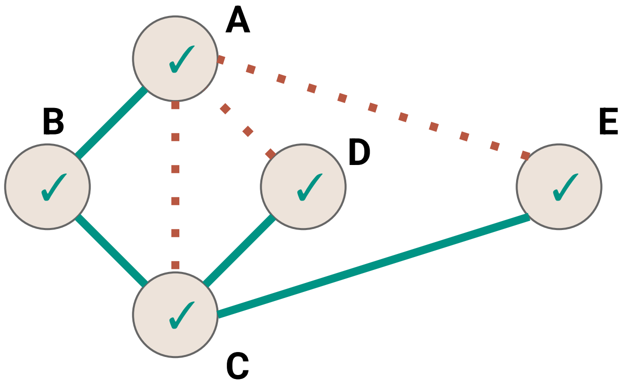
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

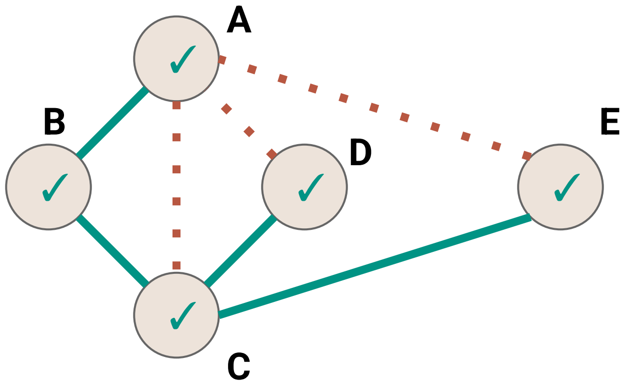
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



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

Detailed Example

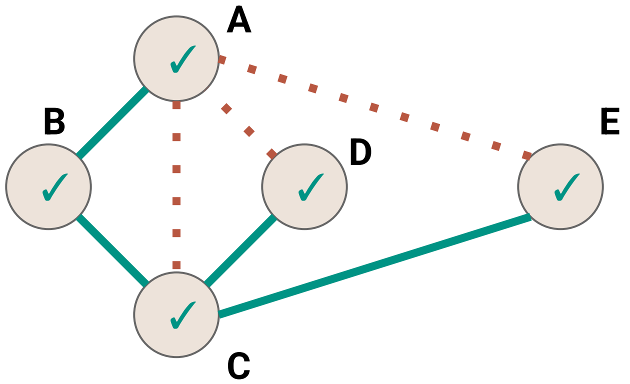
UNEXPLORED

VISITED

UNEXPLORED

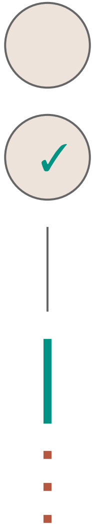
SPANNING

BACK



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

Detailed Example



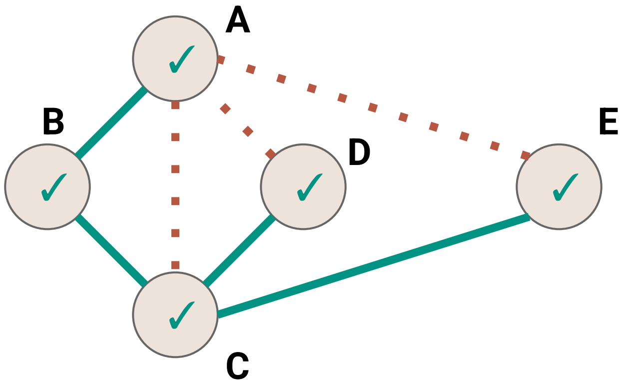
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK



<u>Call Stack</u>
DFS(G)

<u>(→ edges to list)</u>

Detailed Example

UNEXPLORED

VISITED

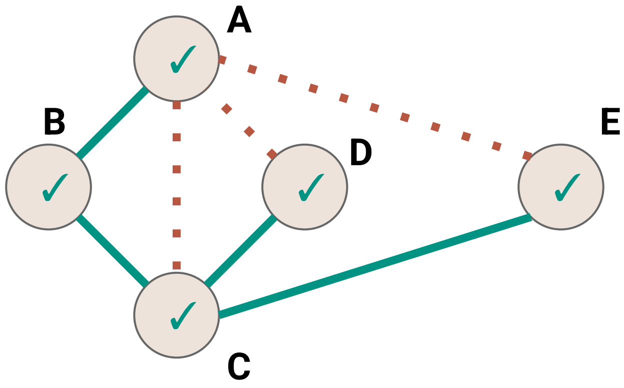
UNEXPLORED

SPANNING

BACK

Call Stack

(→ edges to list)



DFS vs Mazes

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

```
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)
  for (Edge e : v.outEdges) {
    Θ(1)
  }
}
```

As long as we use an adjacency list this will be able to iterate through the adjacent edges in $\Theta(\text{deg}(v))$ time

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

$O(|V|)$ 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. Sum of 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. Sum of all calls to **`DFSOne`**	***O***$(\|E\|)$ **total**

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. Sum of all calls to **`DFSOne`**	$O(\lvert E \rvert)$ **total**
	$O(\lvert V \rvert + \lvert E \rvert)$

DFS without Recursion

Our DFSOne implementation uses recursion for the search…

The recursive calls form a Stack…

Can we make a non-recursive implementation using a Stack explicitly?

```java
public void DFSOneNoRecursion(Graph graph, Vertex v) {
  Stack<Vertex> todo = new Stack<>();
  v.setLabel(VISITED);
  todo.push(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.pop();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.push(w);
        } else {
          e.setLabel(BACK);
        }
}}}}
```

```
public void DFSOneNoRecursion(Graph graph, Vertex v) {
  Stack<Vertex> todo = new Stack<>();
  v.setLabel(VISITED);
  todo.push(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.pop();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.push(w);
        } else {
          e.setLabel(BACK);
        }
}}}}
```

Use a stack to keep track of what vertices we want to visit (basically a running TODO list)

```
public void DFSOneNoRecursion(Graph graph, Vertex v) {
  Stack<Vertex> todo = new Stack<>();
  v.setLabel(VISITED);
  todo.push(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.pop();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.push(w);
        } else {
          e.setLabel(BACK);
        }
}}}}
```

Pop a vertex from the Stack and check all of it's outgoing edges

```java
public void DFSOneNoRecursion(Graph graph, Vertex v) {
  Stack<Vertex> todo = new Stack<>();
  v.setLabel(VISITED);
  todo.push(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.pop();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.push(w);
        } else {
          e.setLabel(BACK);
        }
}}}}
```

When we find a new vertex, mark it as VISITED, and add it to our TODO list.

Remember, our TODO list is a stack (LIFO) so whatever we push last will be the next thing we pop (and explore)

Detailed Example

UNEXPLORED

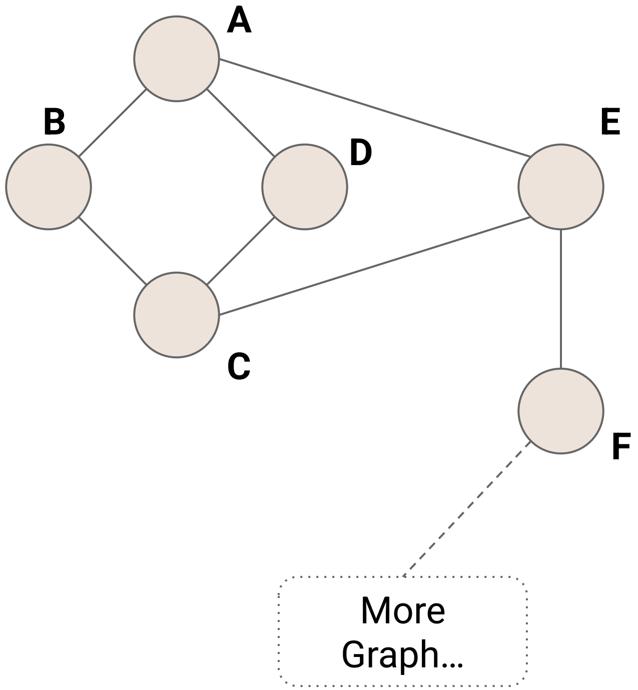
VISITED

UNEXPLORED

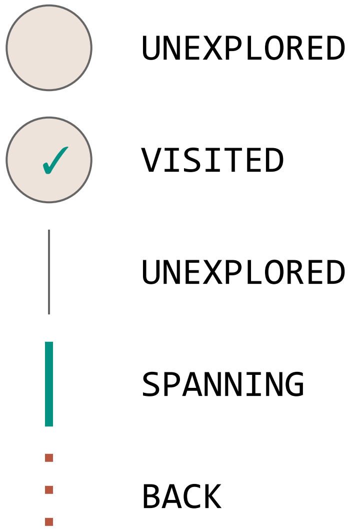
SPANNING

BACK

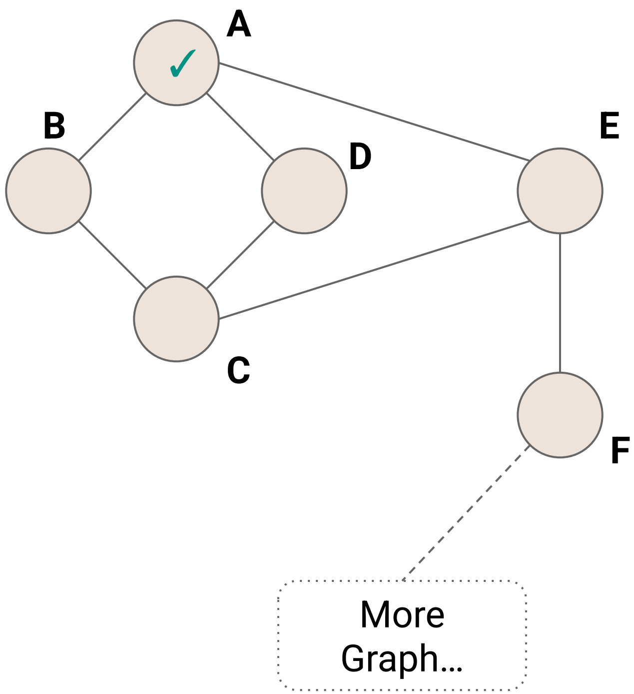
TODO Stack



Detailed Example



TODO Stack
A



Detailed Example

UNEXPLORED

VISITED

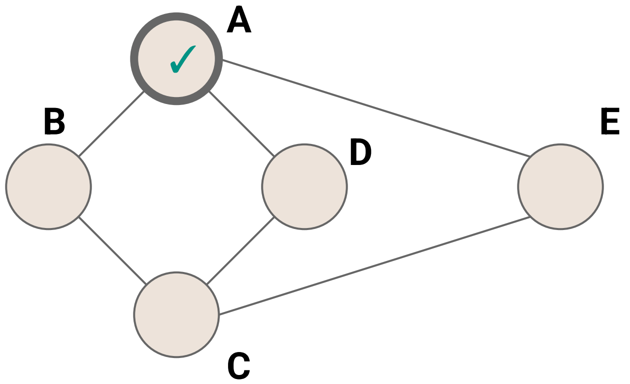
UNEXPLORED

SPANNING

BACK

Current Vertex: **A**

<u>TODO Stack</u>



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

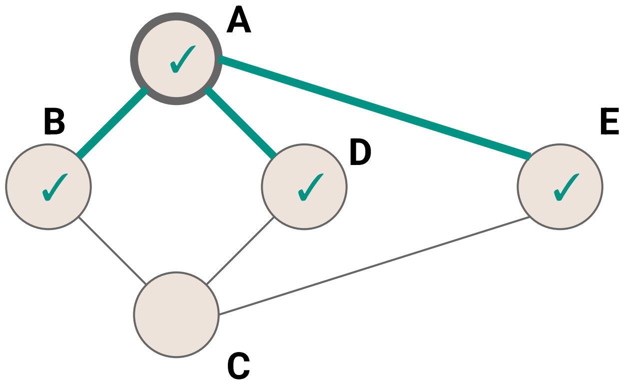
SPANNING

BACK

Current Vertex: **A**

<u>TODO Stack</u>

B
D
E



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

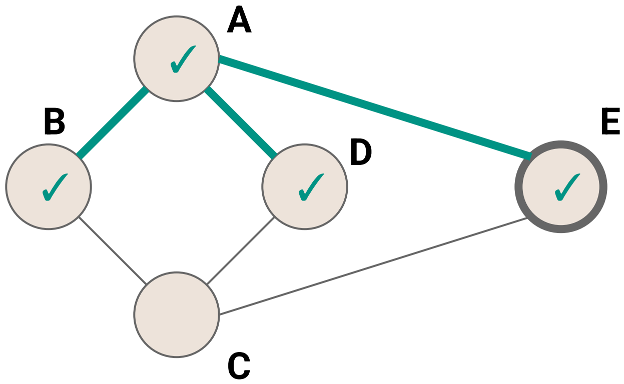
BACK

Current Vertex: **E**

<u>TODO Stack</u>

B

D



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK

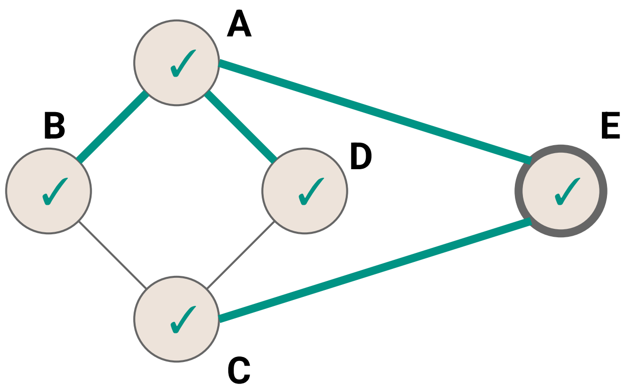
Current Vertex: **E**

<u>TODO Stack</u>

B

D

C



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

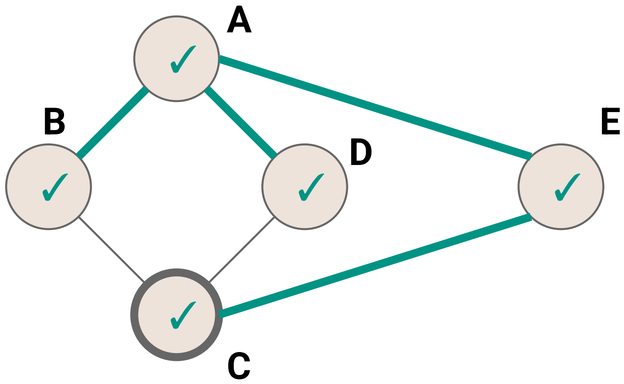
BACK

Current Vertex: **C**

<u>TODO Stack</u>

B

D



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

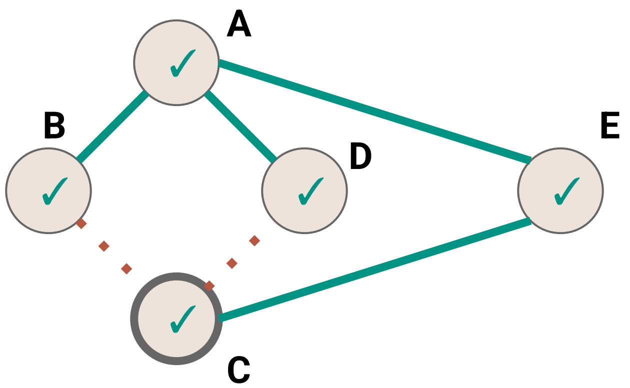
BACK

Current Vertex: **C**

<u>TODO Stack</u>

B

D



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

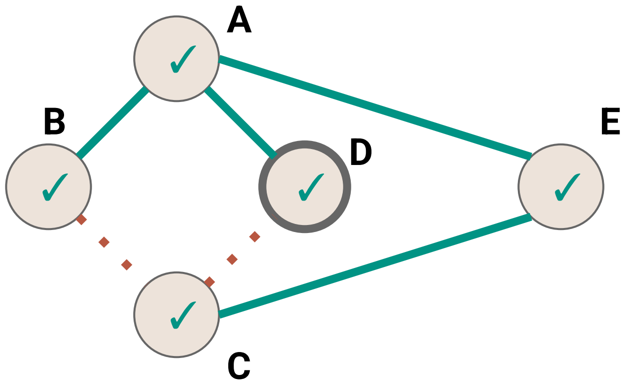
SPANNING

BACK

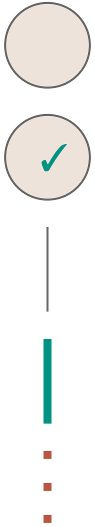
Current Vertex: **D**

<u>TODO Stack</u>

B



Detailed Example



UNEXPLORED

VISITED

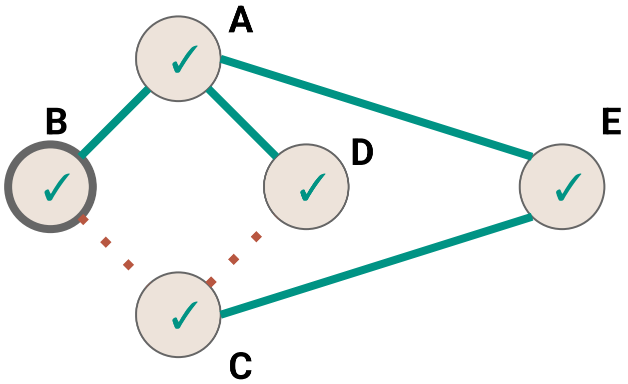
UNEXPLORED

SPANNING

BACK

Current Vertex: **B**

TODO Stack



Wait that didn't look the same...

Let's see it with a more complex graph

Detailed Example

UNEXPLORED

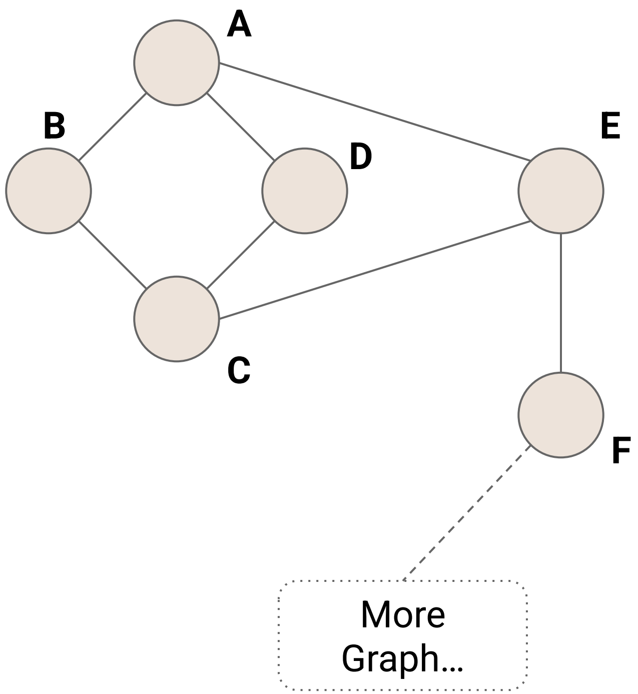
VISITED

UNEXPLORED

SPANNING

BACK

TODO Stack



Detailed Example

UNEXPLORED

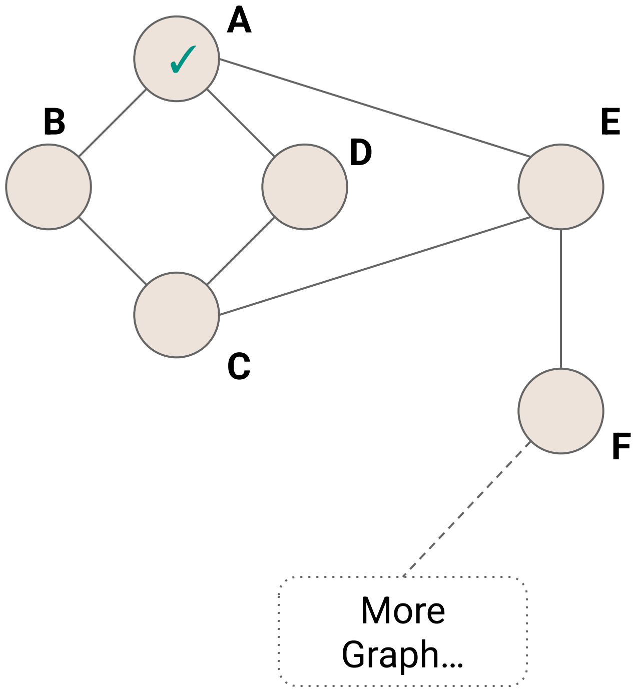
VISITED

UNEXPLORED

SPANNING

BACK

TODO Stack
A



Detailed Example

UNEXPLORED

VISITED

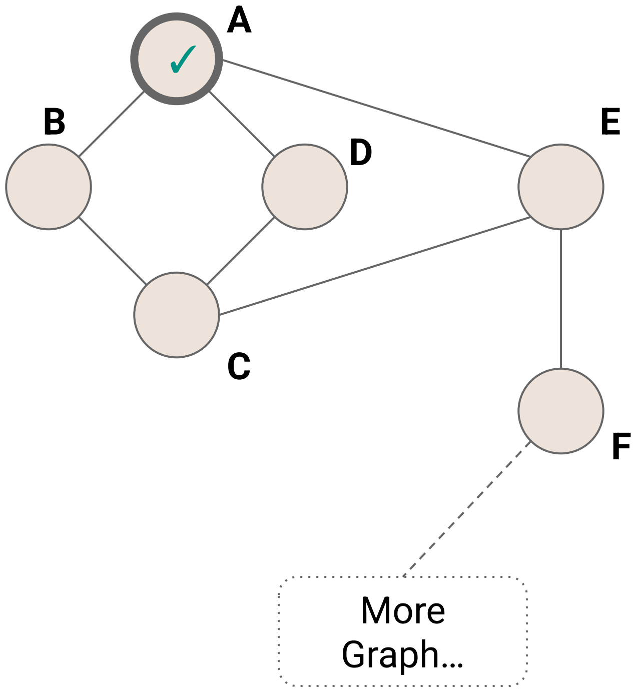
UNEXPLORED

SPANNING

BACK

Current Vertex: **A**

<u>TODO Stack</u>



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

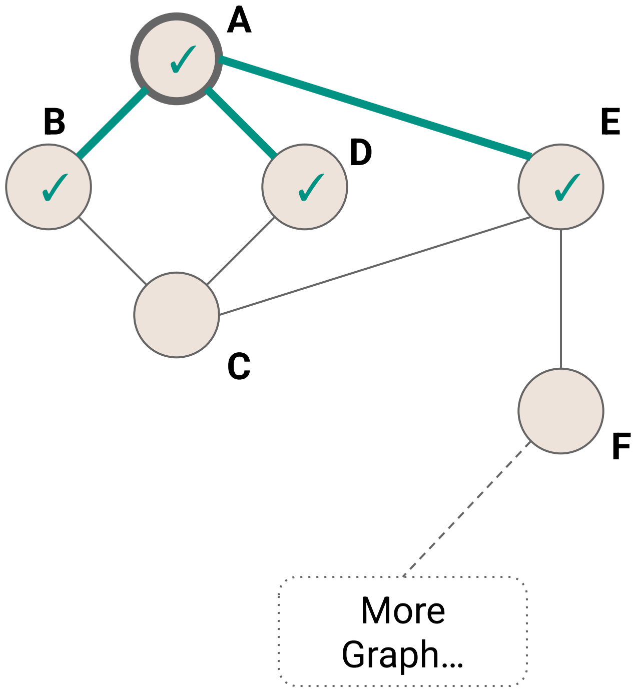
SPANNING

BACK

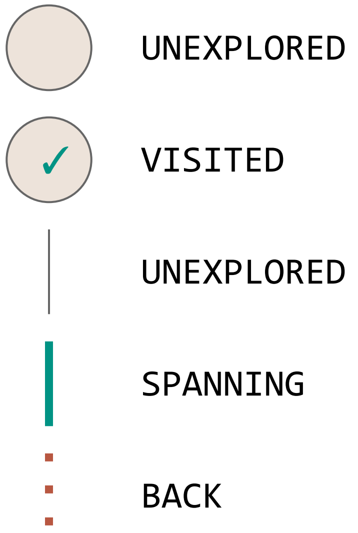
Current Vertex: **A**

<u>TODO Stack</u>

B
D
E

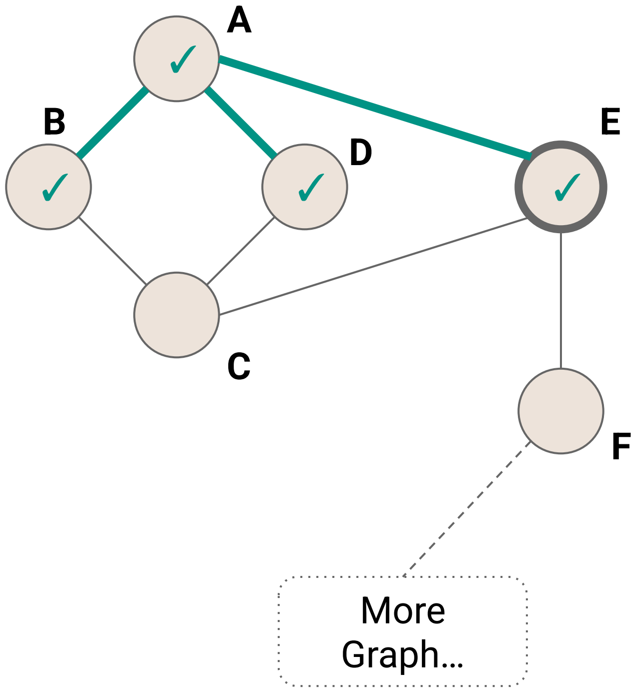


Detailed Example

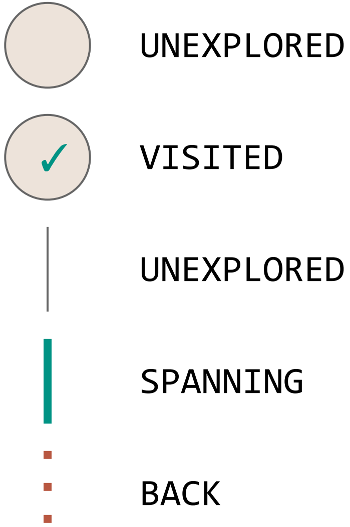


Current Vertex: **E**

<u>TODO Stack</u>
B
D



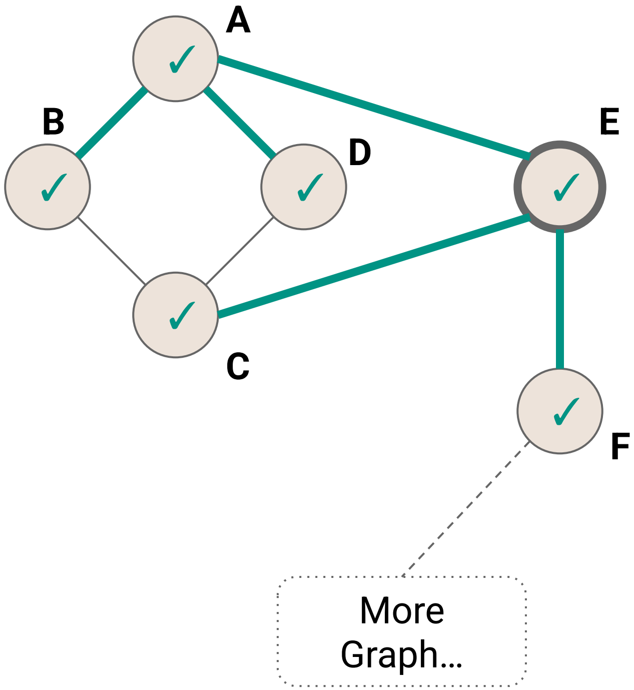
Detailed Example



Current Vertex: **E**

<u>TODO Stack</u>

B
D
C
F



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

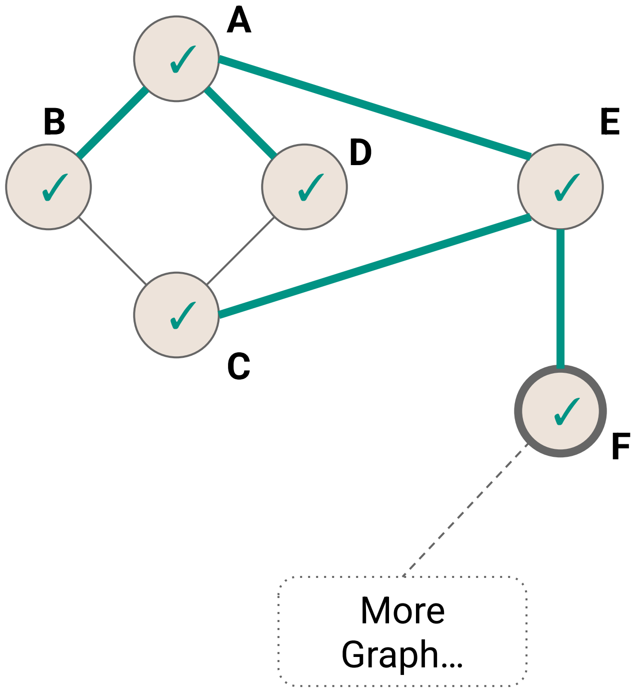
SPANNING

BACK

Current Vertex: **F**

<u>TODO Stack</u>

B
D
C



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

BACK

Current Vertex: **F**

<u>TODO Stack</u>

B

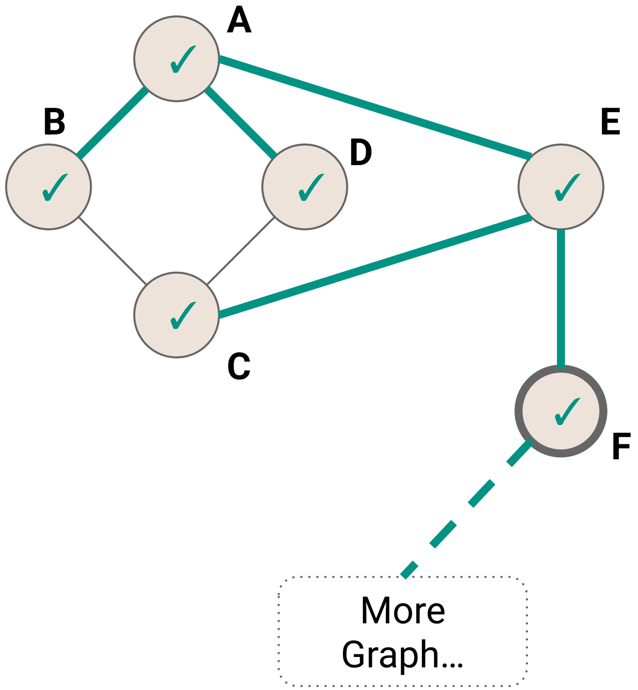
D

C

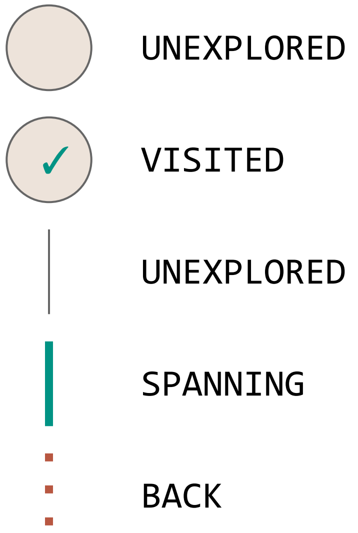
...

...

...



Detailed Example

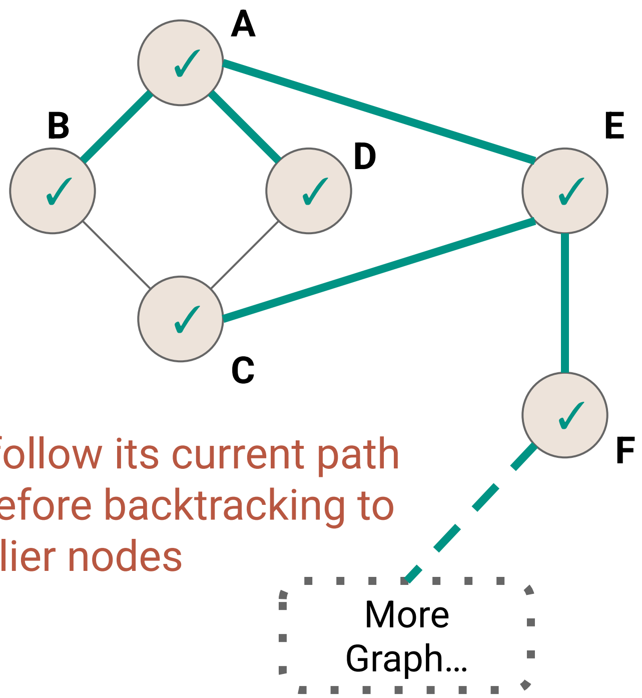


Current Vertex: **???**

<u>TODO Stack</u>

B
D
C
...
...
...

The algorithm will follow its current path as **deep** as it can before backtracking to finish exploring earlier nodes



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

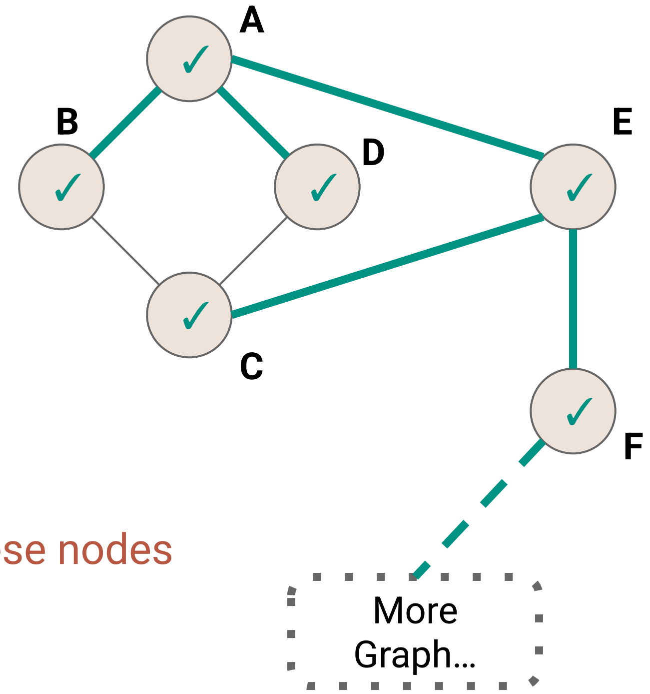
BACK

Current Vertex: **???**

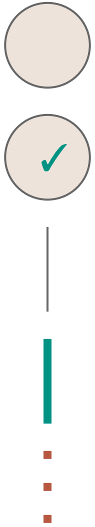
<u>TODO Stack</u>

B
D
C
...
...
...

Eventually all of these nodes will be popped



Detailed Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

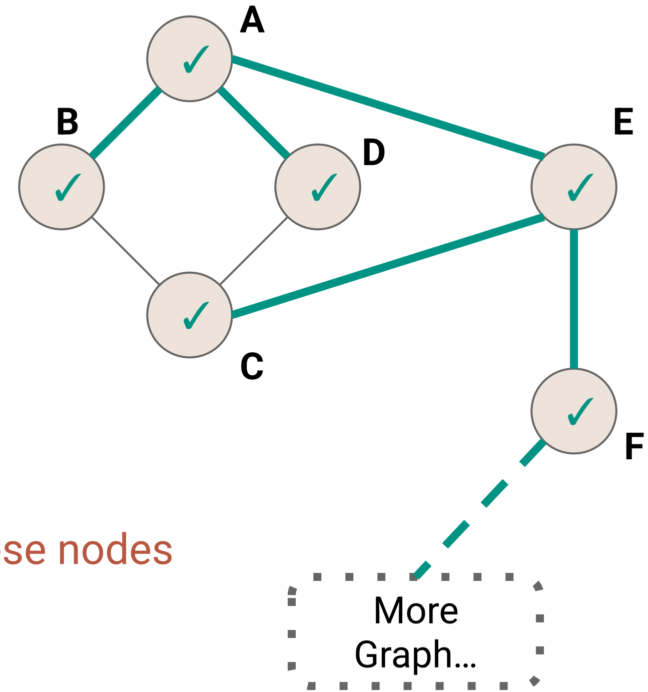
BACK

Current Vertex: **???**

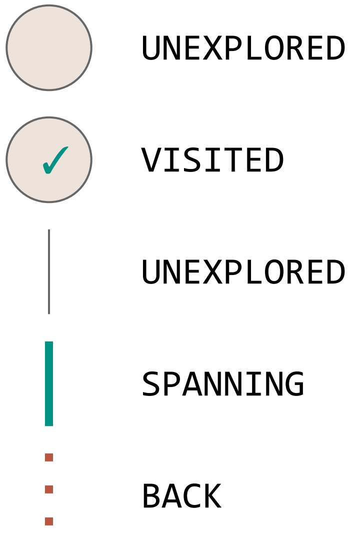
<u>TODO Stack</u>

B
D
C
...
...
...

Eventually all of these nodes will be popped



Detailed Example

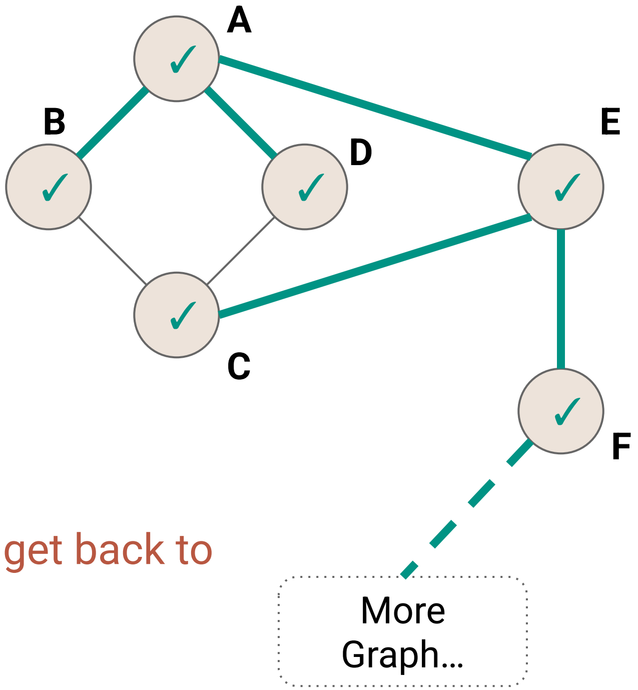


Current Vertex: **???**

<u>TODO Stack</u>

B
D
C

And **NOW** we finally get back to exploring C



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

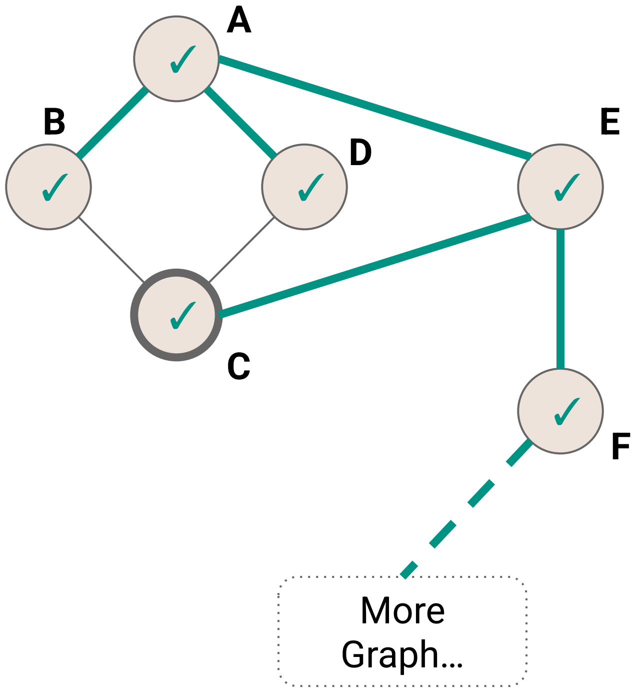
BACK

Current Vertex: **C**

<u>TODO Stack</u>

B

D



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

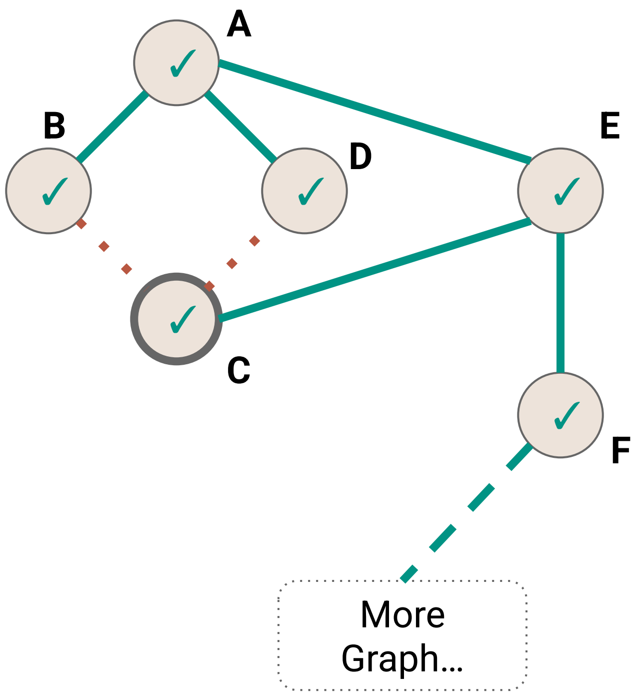
BACK

Current Vertex: **C**

<u>TODO Stack</u>

B

D



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

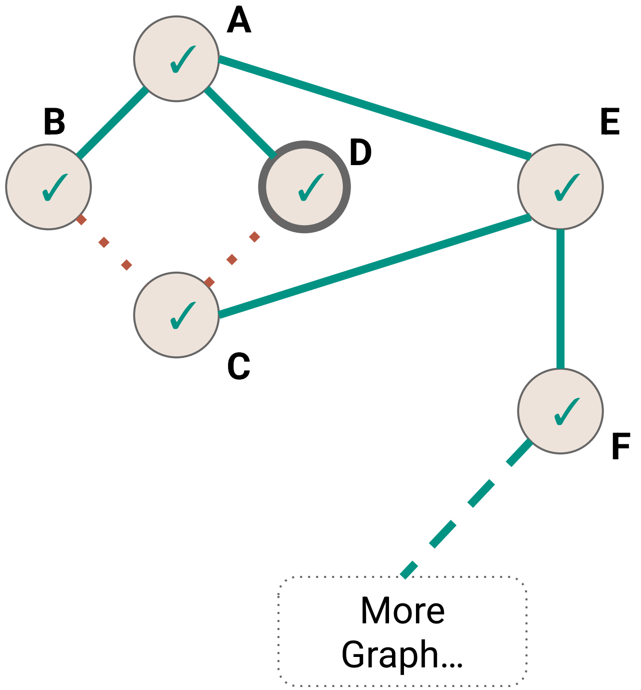
SPANNING

BACK

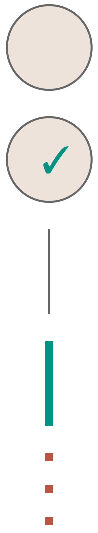
Current Vertex: **D**

<u>TODO Stack</u>

B



Detailed Example



UNEXPLORED

VISITED

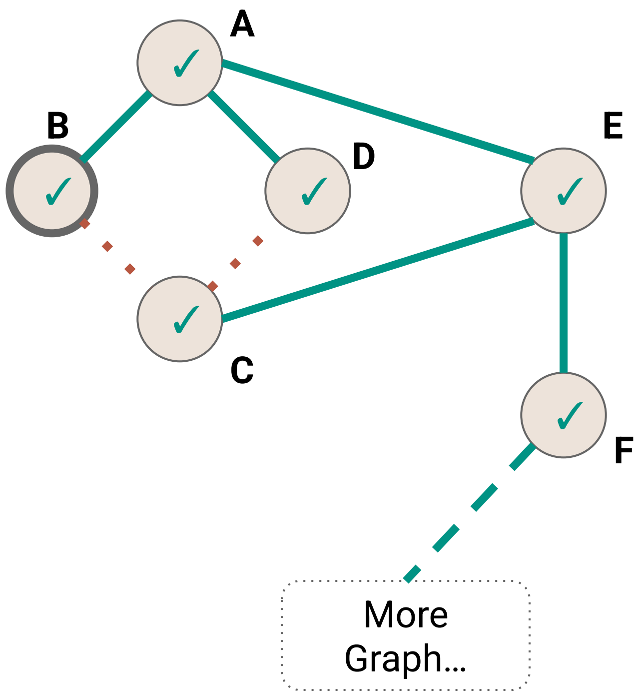
UNEXPLORED

SPANNING

BACK

Current Vertex: **B**

<u>TODO Stack</u>



```java
public void DFSOneNoRecursion(Graph graph, Vertex v) {
  Stack<Vertex> todo = new Stack<>();
  v.setLabel(VISITED);
  todo.push(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.pop();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == U
          curr.setLabel(
          e.setLabel(SPA
          todo.push(w);
        } else {
          e.setLabel(BACK);
        }
}}}}
```

Now back to our burning question…

What happens if we use a Queue to do our search instead of a Stack?

Breadth-First Search

Breadth-First Search

Primary Goals

- 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 $\boldsymbol{O(|V| + |E|)}$, with memory overhead $\boldsymbol{O(|V|)}$

Breadth-First Search

<u>Primary Goals</u>

- Visit every vertex in graph $G = (V,E)$ **in increasing order of distance from the start**
- 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
    - **Side Effect: Identify shortest paths to the starting vertex**
- Complete in time $O(|V| + |E|)$, with memory overhead $O(|V|)$

BFS

```
public void BFS(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) {
      BFSOne(graph, v);
    }
  }
}
```

Same as DFS driver function…just make sure that we explore EVERY vertex, even if the graph is disconnected

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

Use a queue to keep track of what vertices we want to visit (basically a running TODO list)

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

Dequeue a vertex from the Queue and check all of it's outgoing edges

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

When we find a new vertex, mark it as VISITED, and add it to our TODO list.

Remember, our TODO list is a Queue (FIFO) so whatever we enqueued first will be the next thing we dequeue (and explore)

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

When doing BFS we label edges that return to visited vertices as CROSS edges

Detailed Example

UNEXPLORED

VISITED

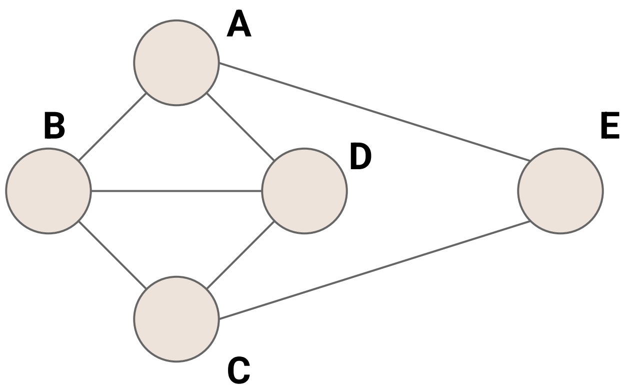
UNEXPLORED

SPANNING

CROSS

<u>Call Stack</u>

<u>Work Queue</u>



Detailed Example

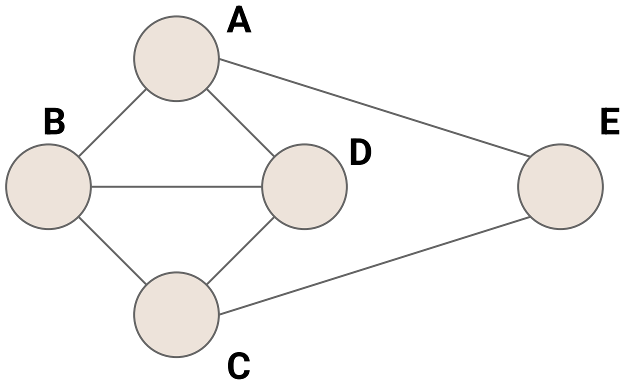
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



Call Stack

`BFS(G)`

Work Queue

Detailed Example

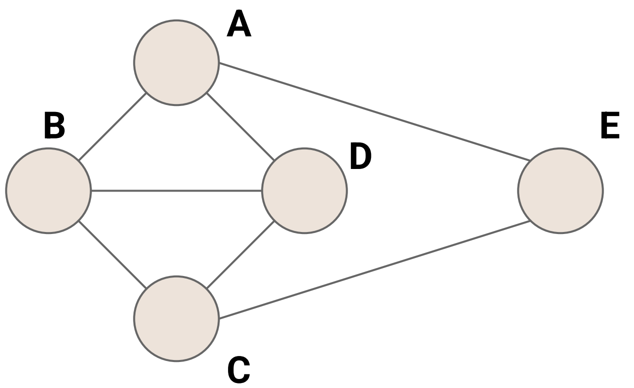
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



Call Stack

BFS(G)
BFSOne(G,A)

Work Queue

Detailed Example

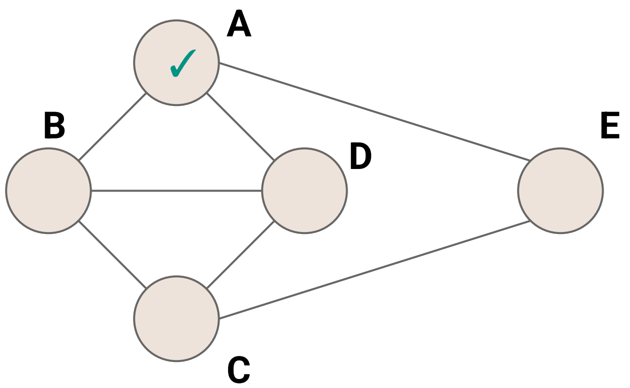
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

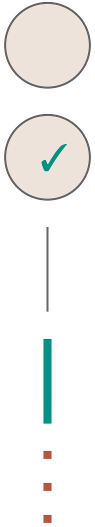
CROSS



Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A

Detailed Example



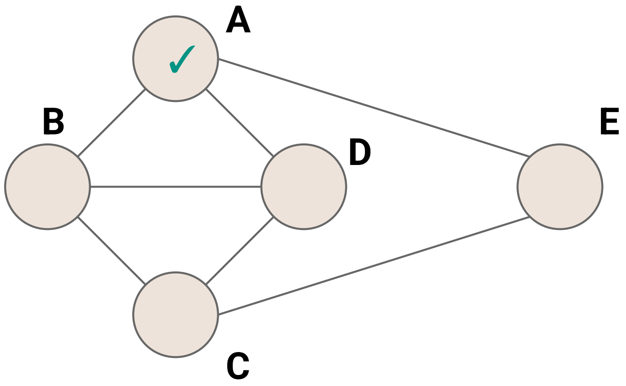
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>

BFS(G)
BFSOne(G,A)

<u>Work Queue</u>

~~A~~

Detailed Example

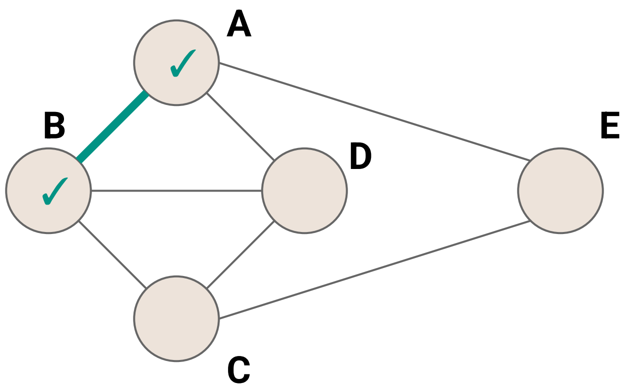
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B

Detailed Example

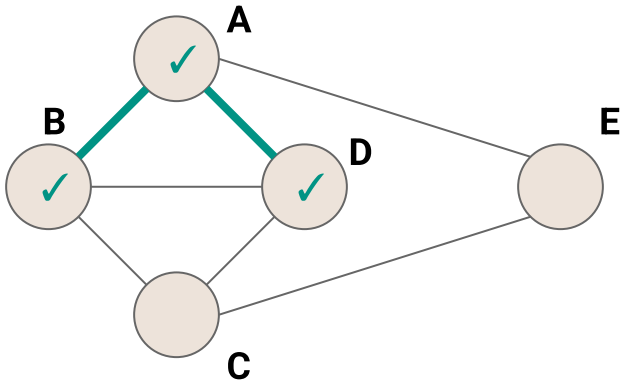
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>

BFS(G)
BFSOne(G,A)

<u>Work Queue</u>

~~A~~
B
D

Detailed Example

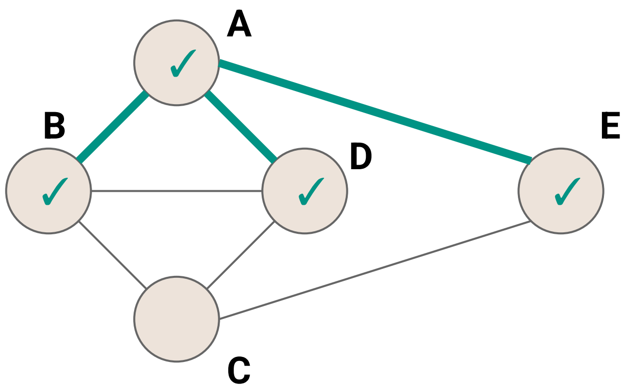
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

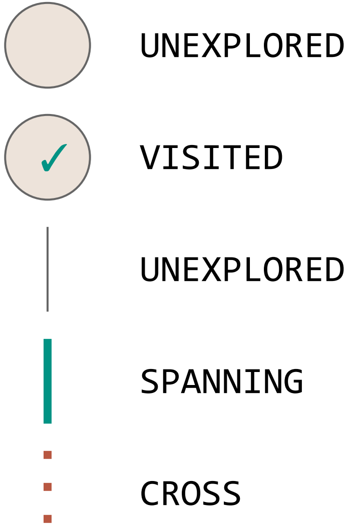
CROSS

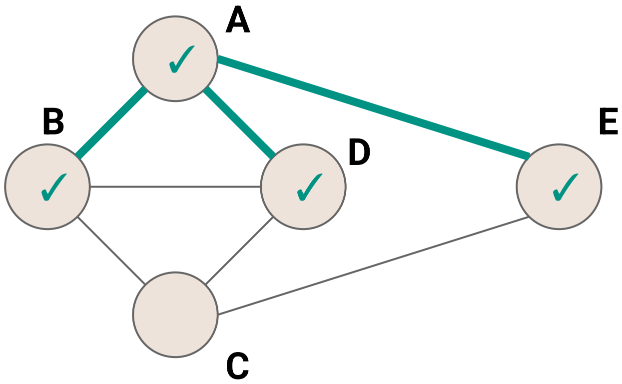


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B
D
E

Detailed Example





Call Stack

BFS(G)
BFSOne(G,A)

Work Queue

~~A~~
~~B~~
D
E

Detailed Example

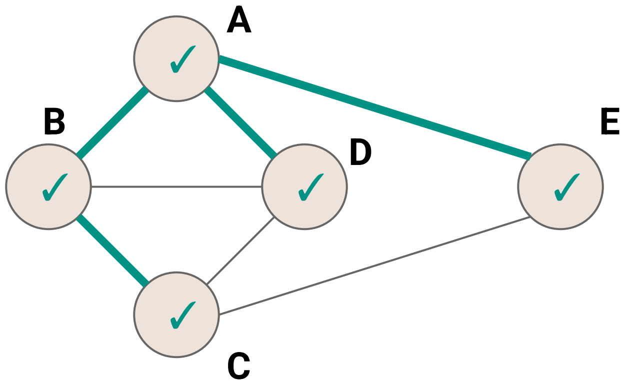
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
~~B~~
D
E
C

Detailed Example

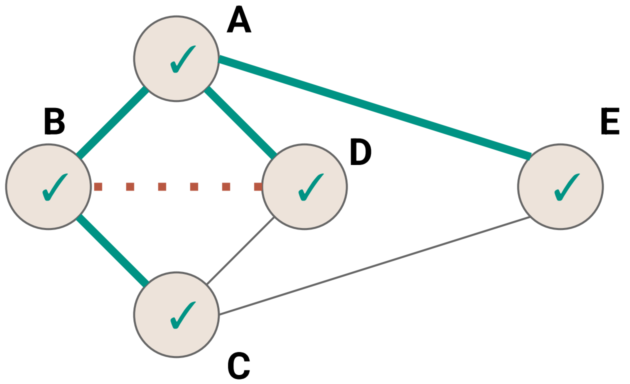
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>

BFS(G)
BFSOne(G,A)

<u>Work Queue</u>

~~A~~
~~B~~
D
E
C

Detailed Example

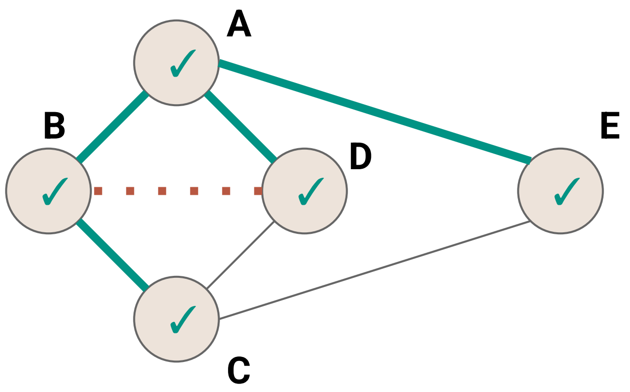
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

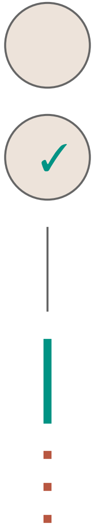
CROSS



Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
~~B~~
~~D~~
E
C

Detailed Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

<u>Call Stack</u>

BFS(G)

BFSOne(G,A)

<u>Work Queue</u>

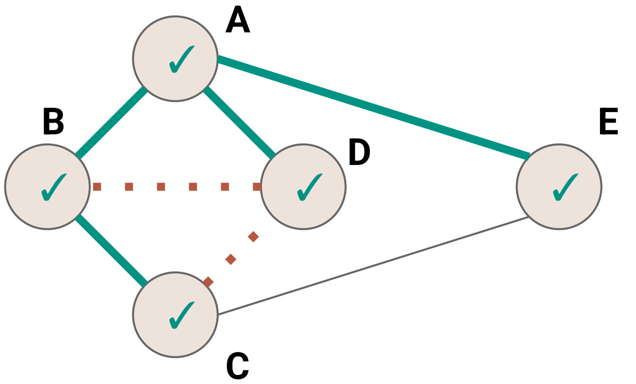
~~**A**~~

~~**B**~~

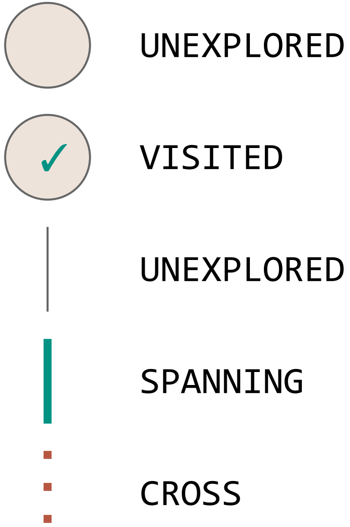
~~**D**~~

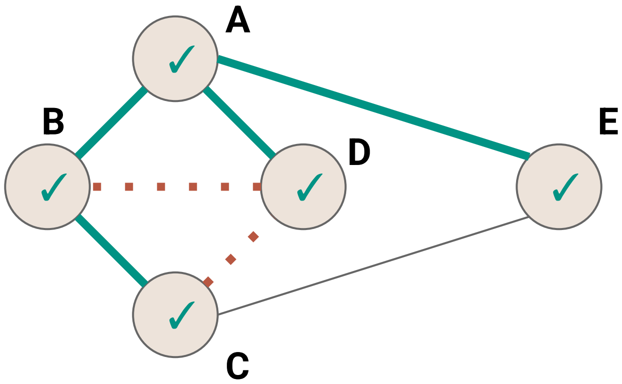
E

C



Detailed Example





__Call Stack__
BFS(G)
BFSOne(G,A)

__Work Queue__
~~A~~
~~B~~
~~D~~
~~E~~
C

Detailed Example

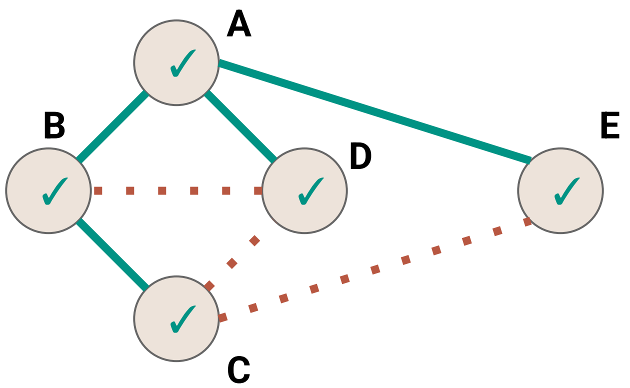
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>

BFS(G)

BFSOne(G,A)

<u>Work Queue</u>

~~A~~

~~B~~

~~D~~

~~E~~

C

Detailed Example

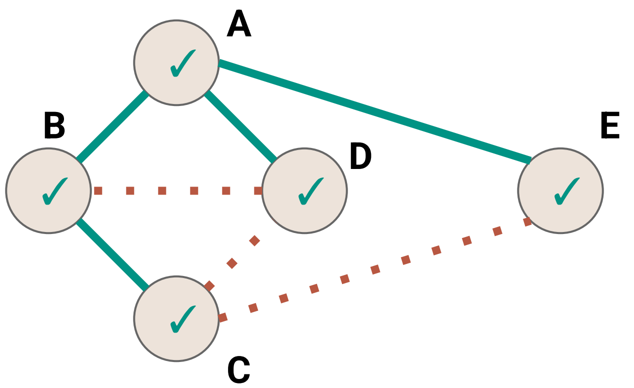
UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS



Call Stack

BFS(G)
BFSOne(G,A)

Work Queue

~~A~~
~~B~~
~~D~~
~~E~~
~~C~~

BFS Complexity

```java
public void BFS(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) {
      BFSOne(graph, v);
    }
  }
}
```

BFS Complexity

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

BFS Complexity

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

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          curr.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          curr.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          curr.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Θ(1)
        if (w.label == UNEXPLORED) {
          Θ(1)
          todo.enqueue(w);
        } else {
          Θ(1)
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Θ(1)
        if (w.label == UNEXPLORED) {
          Θ(1)
          Θ(1)
        } else {
          Θ(1)
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    for (Edge e : curr.outEdges) {
      Θ(1)
    }
  }
}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    Θ(deg(v))
  }
}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    Θ(deg(v))
  }
}
```

How many iterations will this while loop run?

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    Θ(deg(v))
  }
}
```

How many iterations will this while loop run?
Each vertex will be enqueued exactly ONCE

```
public void BFSOne(Graph graph, Vertex v) {
  Θ(1)
  while (!todo.isEmpty()) {
    Θ(1)
    Θ(deg(v))
  }
}
```

How many iterations will this while loop run?
Each vertex will be enqueued exactly ONCE
The cost to process each vertex is deg(v)

Breadth-First Search Complexity

What is the sum over all iterations in `BFSOne`?

Breadth-First Search Complexity

What is the sum over all iterations in **BFSOne**?

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

Breadth-First Search Complexity

What is the sum over all iterations in **`BFSOne`**?

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

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

Breadth-First Search Complexity

What is the sum over all iterations in **BFSOne**?

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

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

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

Breadth-First Search Complexity

What is the sum over all iterations in **BFSOne**?

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

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

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

$$= O(|E|)$$

Breadth-First Search Complexity

In summary…

1. Mark the vertices `UNVISITED`

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary...

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

Breadth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`** $O(|V|)$
2. Mark the edges **`UNVISITED`** $O(|E|)$
3. Add each vertex to the work queue

Breadth-First Search Complexity

In summary…

1.	Mark the vertices **`UNVISITED`**	$O(\|V\|)$
2.	Mark the edges **`UNVISITED`**	$O(\|E\|)$
3.	Add each vertex to the work queue	$O(\|V\|)$

Breadth-First Search Complexity

In summary…

1. Mark the vertices `UNVISITED` $O(|V|)$
2. Mark the edges `UNVISITED` $O(|E|)$
3. Add each vertex to the work queue $O(|V|)$
4. Process each vertex

Breadth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`**	***O***(|***V***|)
2. Mark the edges **`UNVISITED`**	***O***(|***E***|)
3. Add each vertex to the work queue	***O***(|***V***|)
4. Process each vertex	***O***(|***E***|) **total**

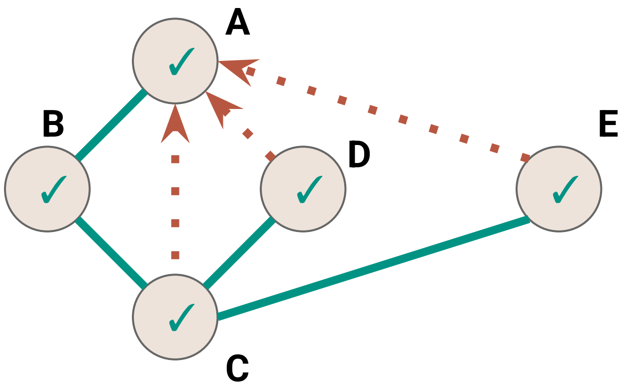
Breadth-First Search Complexity

In summary…

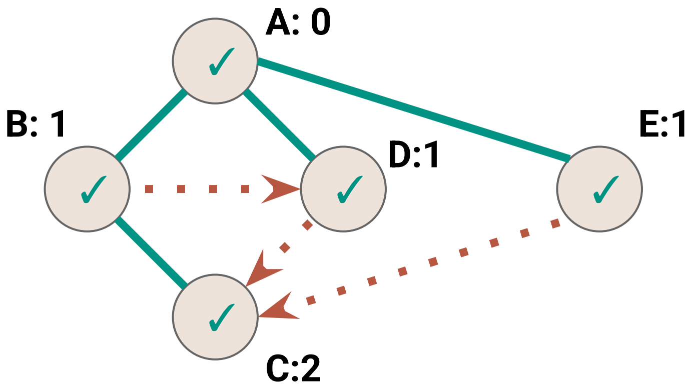
1.	Mark the vertices **`UNVISITED`**	$O(\lvert V \rvert)$
2.	Mark the edges **`UNVISITED`**	$O(\lvert E \rvert)$
3.	Add each vertex to the work queue	$O(\lvert V \rvert)$
4.	Process each vertex	$O(\lvert E \rvert)$ **total**
		$O(\lvert V \rvert + \lvert E \rvert)$

DFS vs BFS

DFS

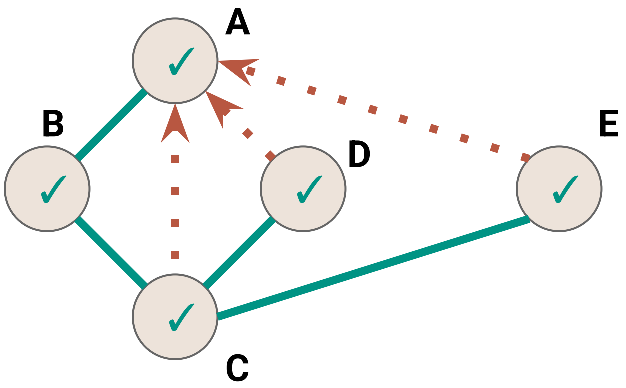


BFS



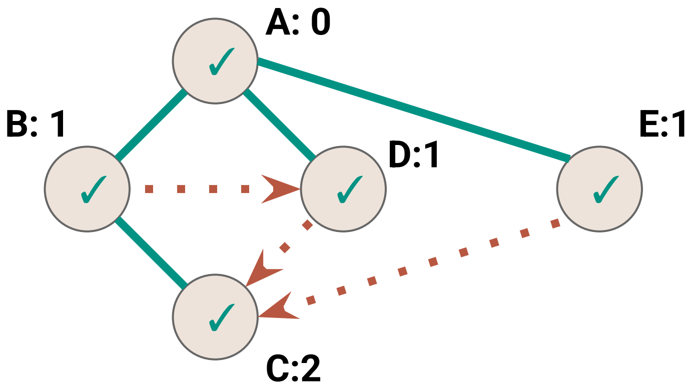
DFS vs BFS

DFS



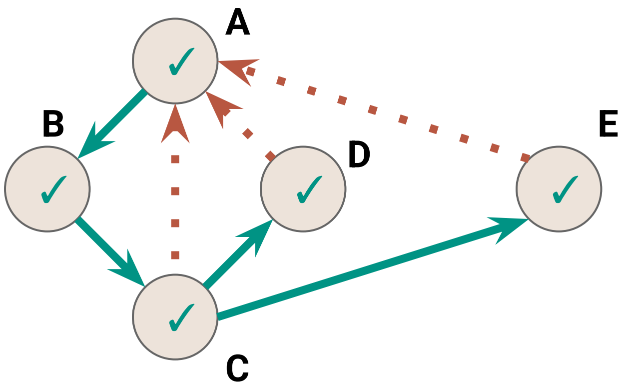
BACK Edge(*v*,*w*): ***w*** is an ancestor of ***v*** in the discovery tree

BFS



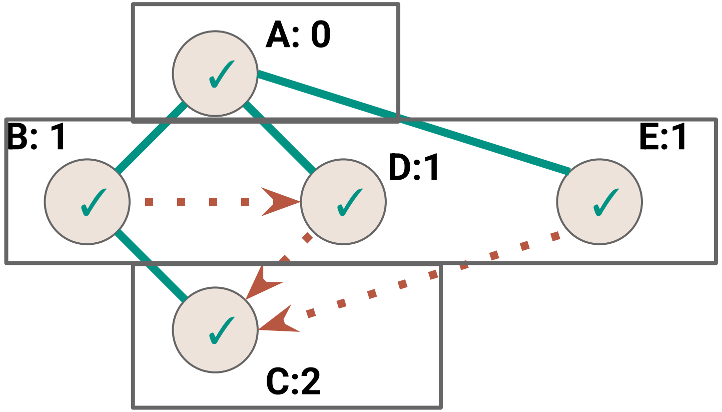
DFS vs BFS

DFS



BACK Edge(*v*,*w*): ***w*** is an ancestor of ***v*** in the discovery tree

BFS



CROSS Edge(*v*,*w*): ***w*** is at the same or next level as ***v***

DFS Traversal vs BFS Traversal

Application	DFS	BFS
Spanning Trees	✓	✓
Connected Components	✓	✓
Paths/Connectivity	✓	✓
Cycles	✓	✓
Shortest Paths*		✓
Articulation Points	✓	

** we'll come back to this...*