

CSE 250

Data Structures

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

Lec 20: Graph Traversals

Announcements

- PA2 released
 - Testing phase due **Tuesday** 10/22
 - Implementation due **Tuesday** 10/29
 - AutoLab opens tonight

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:

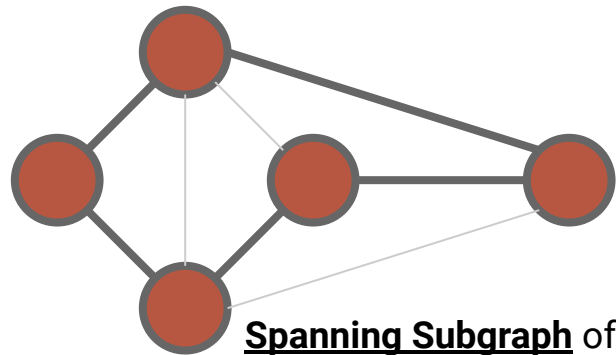
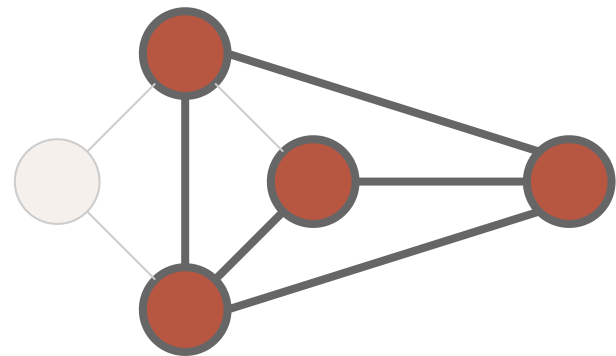
S 's vertices are a subset of G 's vertices

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

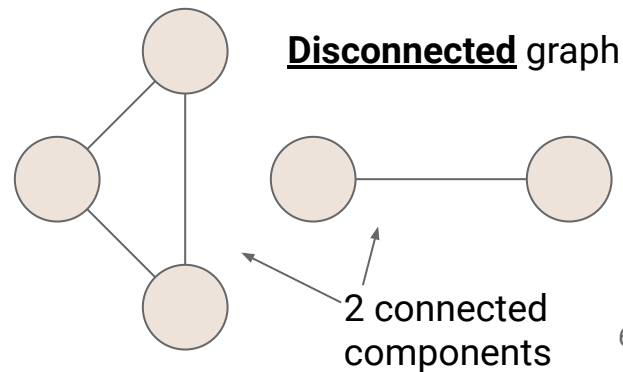
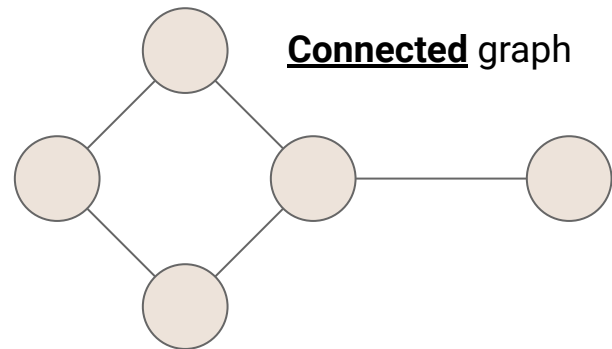
A graph is **connected**...

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 **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

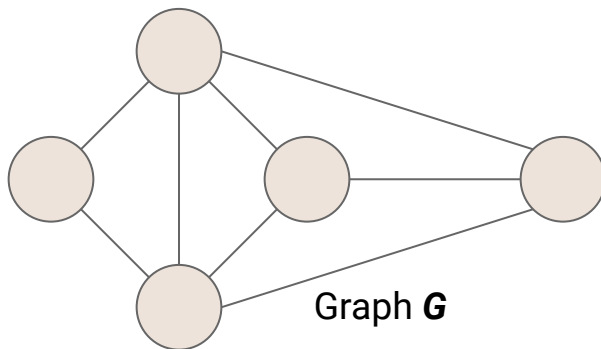
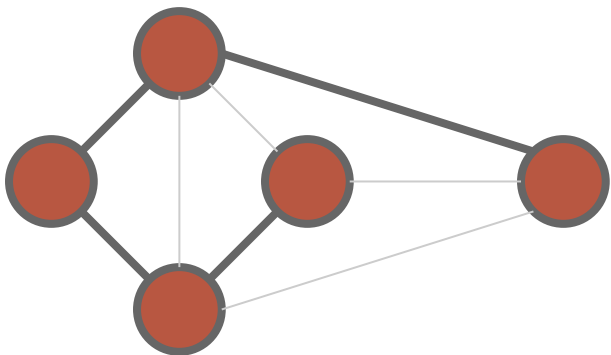
A few more definitions

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

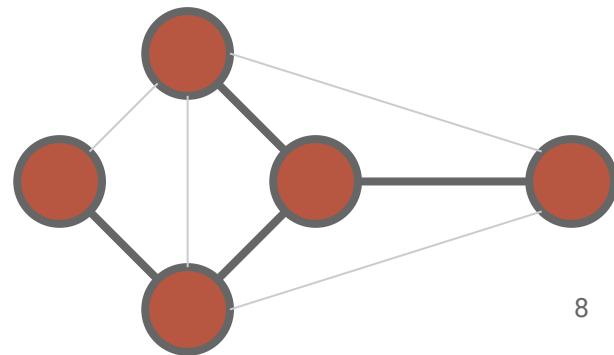
...Is a spanning subgraph that is a tree

...It is not unique unless the graph is a tree

A **Spanning Tree** of G



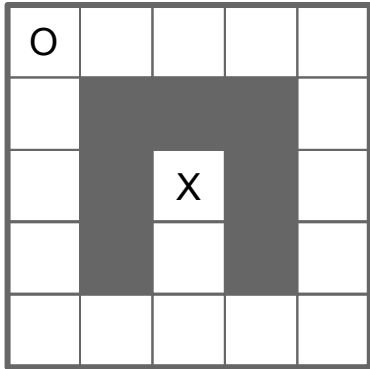
Another **Spanning Tree** of G



Now back to the question...Connectivity

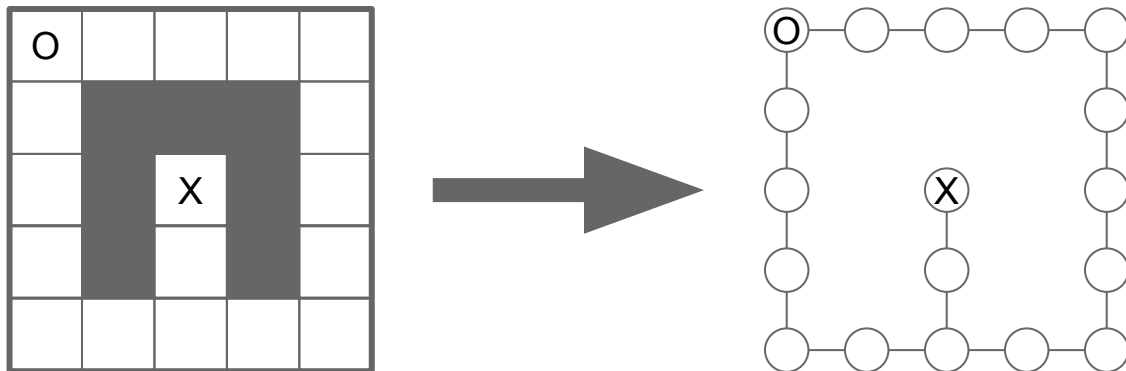
Back to Mazes

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 $G = (V, E)$
- 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:

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

DFS

```
1 public void DFS(Graph graph) {
2     for (Vertex v : graph.vertices) {
3         v.setLabel(UNEXPLORED);
4     }
5     for (Edge e : graph.edges) {
6         e.setLabel(UNEXPLORED);
7     }
8     for (Vertex v : graph.vertices) {
9         if (v.label == UNEXPLORED) {
10            DFSOne(graph, v);
11        }
12    }
13 }
```

DFS

```
1 public void DFS(Graph graph) {  
2     for (Vertex v : graph.vertices) {  
3         v.setLabel(UNEXPLORED);  
4     }  
5     for (Edge e : graph.edges) {  
6         e.setLabel(UNEXPLORED);  
7     }  
8     for (Vertex v : graph.vertices) {  
9         if (v.label == UNEXPLORED) {  
10            DFSOne(graph, v);  
11        }  
12    }  
13 }
```

Initialize all vertices and edges to UNEXPLORED

DFS

```
1 public void DFS(Graph graph) {  
2     for (Vertex v : graph.vertices) {  
3         v.setLabel(UNEXPLORED);  
4     }  
5     for (Edge e : graph.edges) {  
6         e.setLabel(UNEXPLORED);  
7     }  
8     for (Vertex v : graph.vertices) {  
9         if (v.label == UNEXPLORED) {  
10            DFSOne(graph, v);  
11        }  
12    }  
13 }
```

Call DFSOne to label the connected component of every unexplored vertex

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2     v.setLabel(VISITED);  
3     for (Edge e : v.outEdges) {  
4         if (e.label == UNEXPLORED) {  
5             Vertex w = e.to;  
6             if (w.label == UNEXPLORED) {  
7                 e.setLabel(SPANNING);  
8                 DFSOne(graph, w);  
9             } else {  
10                e.setLabel(BACK);  
11            }  
12        }  
13    }}
```

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {
2     v.setLabel(VISITED); ← Mark the vertex as VISITED (so we'll never try to visit it again)
3     for (Edge e : v.outEdges) {
4         if (e.label == UNEXPLORED) {
5             Vertex w = e.to;
6             if (w.label == UNEXPLORED) {
7                 e.setLabel(SPANNING);
8                 DFSOne(graph, w);
9             } else {
10                e.setLabel(BACK);
11            }
12        }
13    }}
```

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2     v.setLabel(VISITED);  
3     for (Edge e : v.outEdges) {  
4         if (e.label == UNEXPLORED) {  
5             Vertex w = e.to;  
6             if (w.label == UNEXPLORED) {  
7                 e.setLabel(SPANNING);  
8                 DFSOne(graph, w);  
9             } else {  
10                e.setLabel(BACK);  
11            }  
12        }  
13    }  
14 }
```

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

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2     v.setLabel(VISITED);  
3     for (Edge e : v.outEdges) {  
4         if (e.label == UNEXPLORED) {  
5             Vertex w = e.to;  
6             if (w.label == UNEXPLORED) {  
7                 e.setLabel(SPANNING);  
8                 DFSOne(graph, w);  
9             } else {  
10                e.setLabel(BACK);  
11            }  
12        }  
13    }  
14 }
```

Follow the unexplored edges

DFSOne

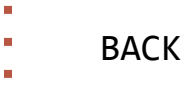
```
1 public void DFSOne(Graph graph, Vertex v) {
2     v.setLabel(VISITED);
3     for (Edge e : v.outEdges) {
4         if (e.label == UNEXPLORED) {
5             Vertex w = e.to;
6             if (w.label == UNEXPLORED) {
7                 e.setLabel(SPANNING);
8                 DFSOne(graph, w);
9             } else {
10                e.setLabel(BACK);
11            }
12        }
13    }}
```

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

DFSOne

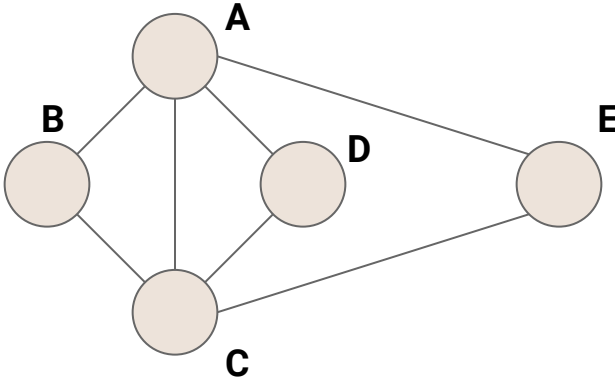
```
1 public void DFSOne(Graph graph, Vertex v) {
2     v.setLabel(VISITED);
3     for (Edge e : v.outEdges) {
4         if (e.label == UNEXPLORED) {
5             Vertex w = e.to;
6             if (w.label == UNEXPLORED) {
7                 e.setLabel(SPANNING);
8                 DFSOne(graph, w);
9             } else {
10                e.setLabel(BACK); Otherwise, we just found a cycle
11            }
12        }
13    }}
```

Detailed Example

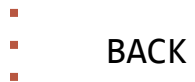


Call Stack

(→ edges to list)

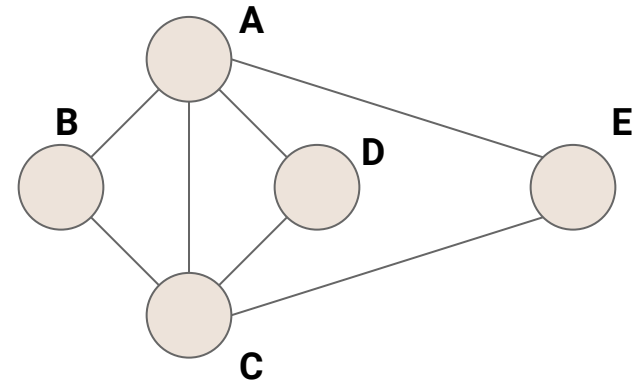


Detailed Example

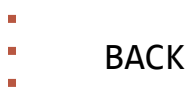


Call Stack
DFS(G)

(→ edges to list)



Detailed Example

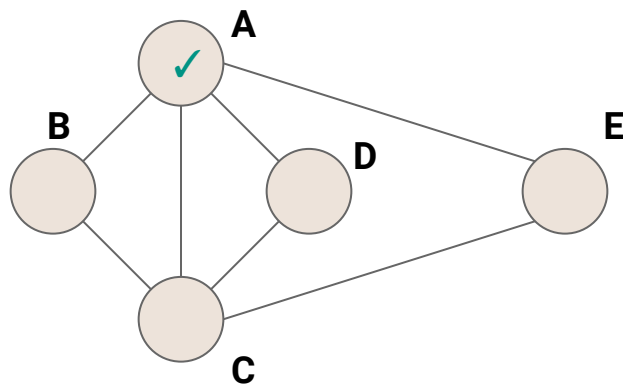


Call Stack

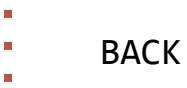
DFS(G)

DFSOne(G,A)

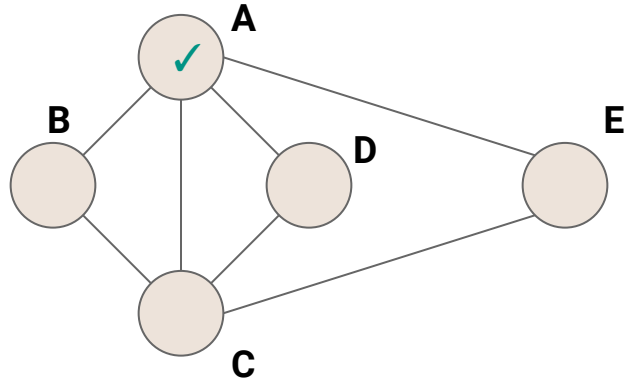
(→ edges to list)



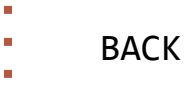
Detailed Example



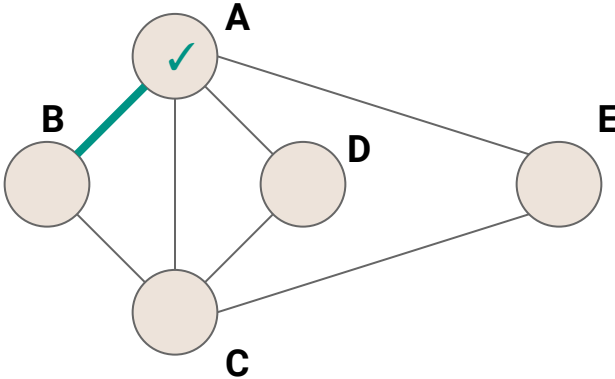
Call Stack (→ edges to list)
DFS(G)
DFSOne(G,A) (→ B, C, D, E)



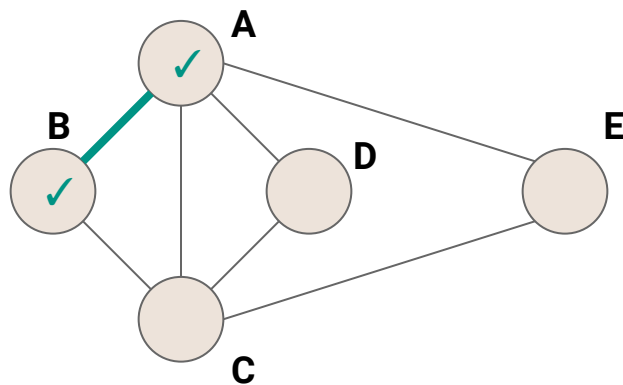
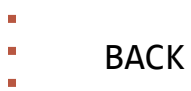
Detailed Example



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

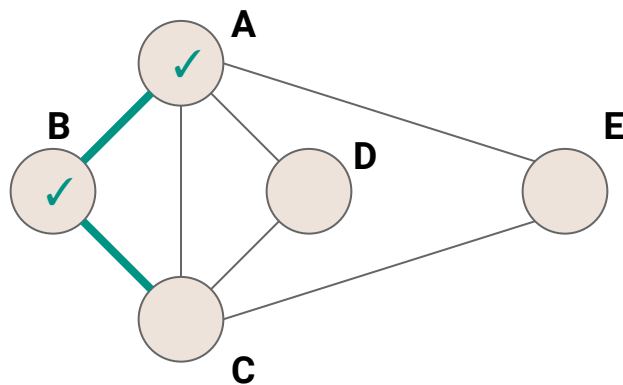
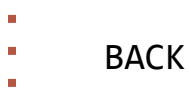


Detailed Example



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

Detailed Example



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

Detailed Example



UNEXPLORED



VISITED



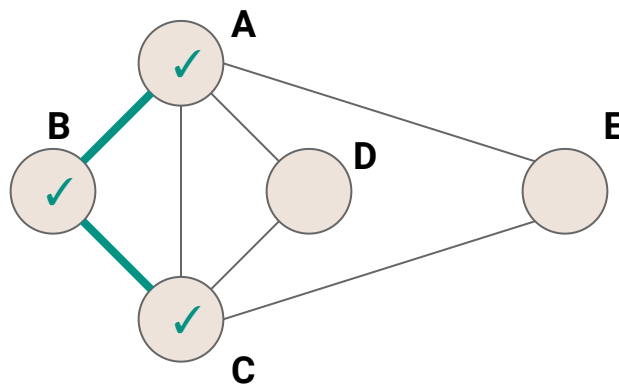
UNEXPLORED



SPANNING

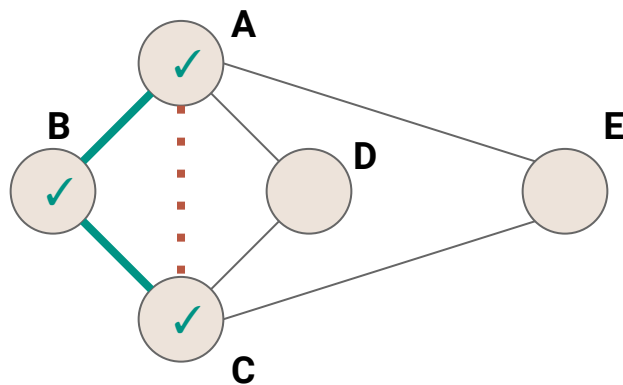
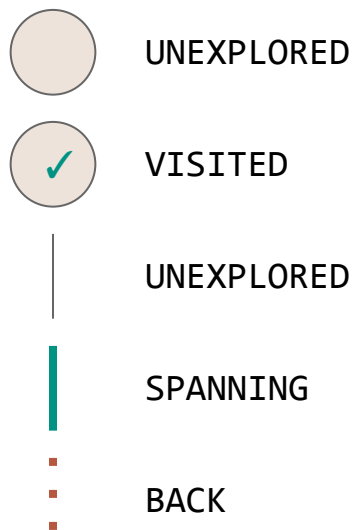


BACK



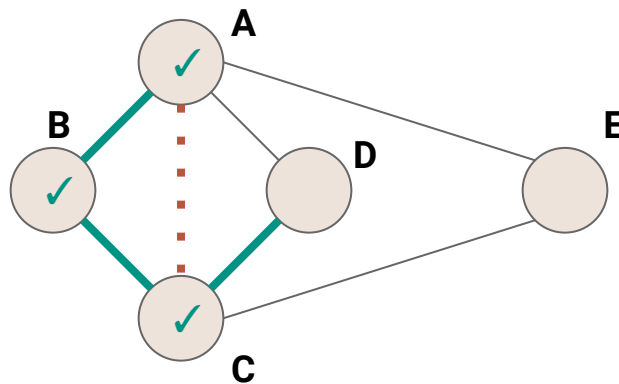
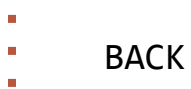
<u>Call Stack</u>	<u>(\rightarrow edges to list)</u>
DFS(G)	
DFSone(G,A)	(\rightarrow B, C, D, E)
DFSone(G,B)	(\rightarrow A, C)
DFSone(G,C)	(\rightarrow B, A, D, E)

Detailed Example



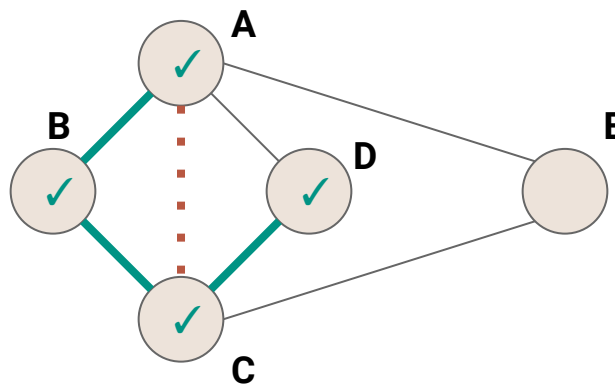
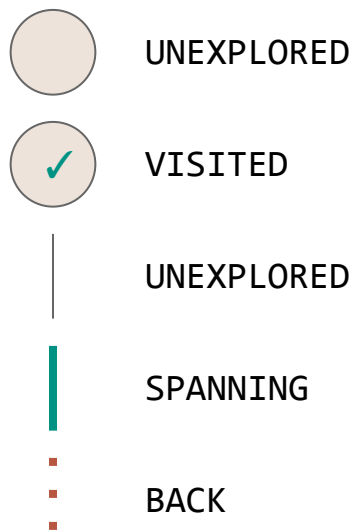
<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSone(G,A)	(→ B, C, D, E)
DFSone(G,B)	(→ A, C)
DFSone(G,C)	(→ B, A, D, E)

Detailed Example



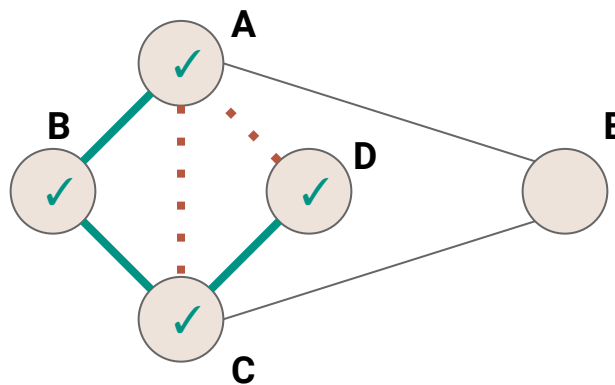
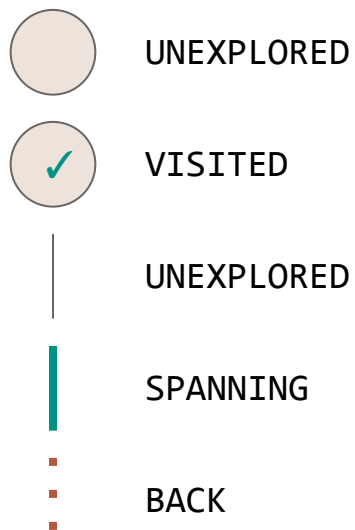
<u>Call Stack</u>	<u>(\rightarrow edges to list)</u>
DFS(G)	
DFSone(G,A)	(\rightarrow B, C, D, E)
DFSone(G,B)	(\rightarrow A, C)
DFSone(G,C)	(\rightarrow B, A, D, E)

Detailed Example



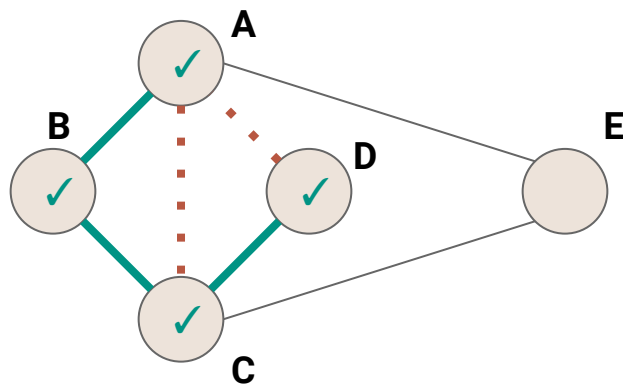
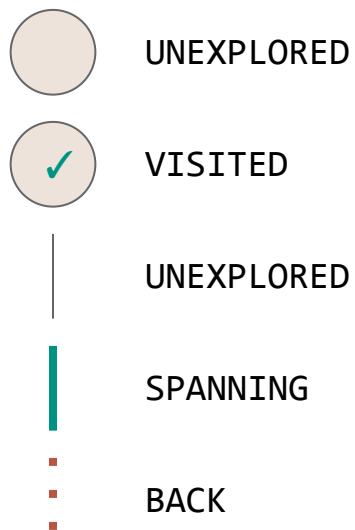
<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSOne(G,A)	(→ B, C, D, E)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)
DFSOne(G,D)	(→ A, C)

Detailed Example



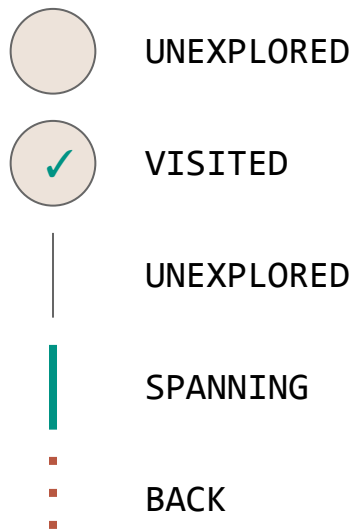
<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSOne(G,A)	(→ B, C, D, E)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)
DFSOne(G,D)	(→ A, C)

Detailed Example

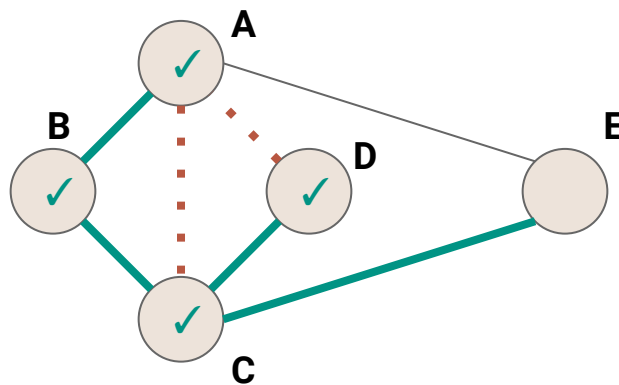


<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSone(G,A)	(→ B, C, D, E)
DFSone(G,B)	(→ A, C)
DFSone(G,C)	(→ B, A, D, E)

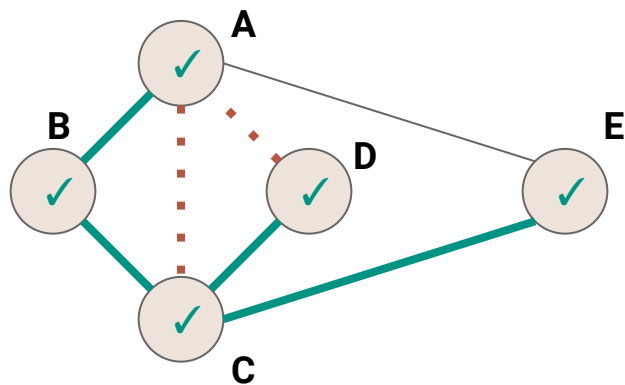
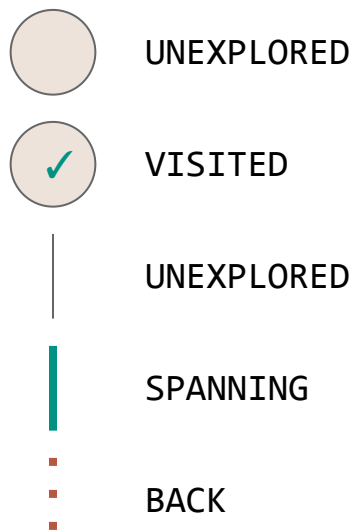
Detailed Example



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

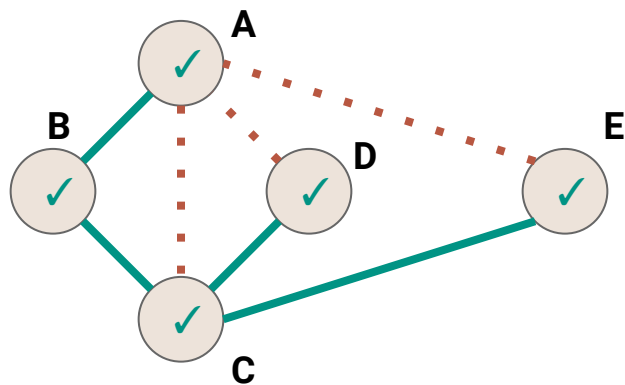
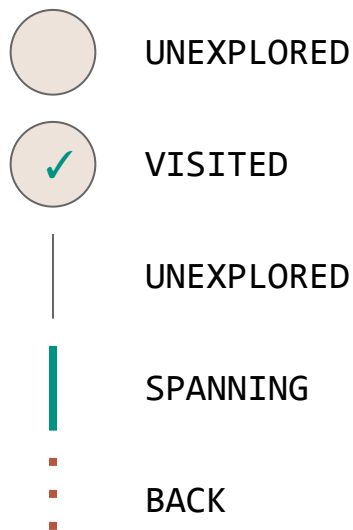


Detailed Example



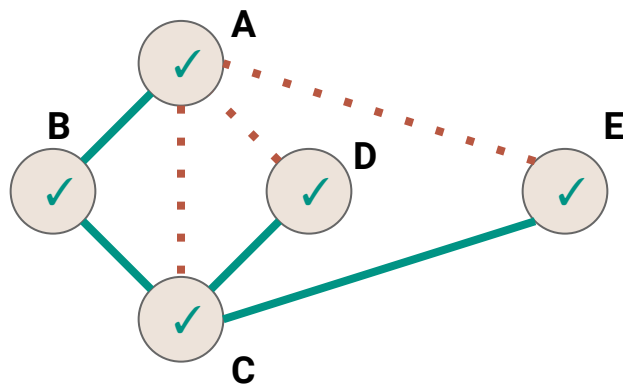
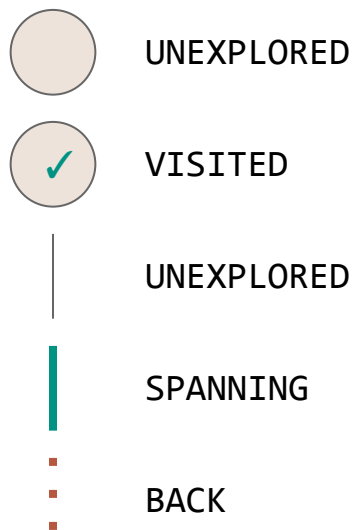
<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSOne(G,A)	(→ B, C, D, E)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)
DFSOne(G,E)	(→ A, C)

Detailed Example



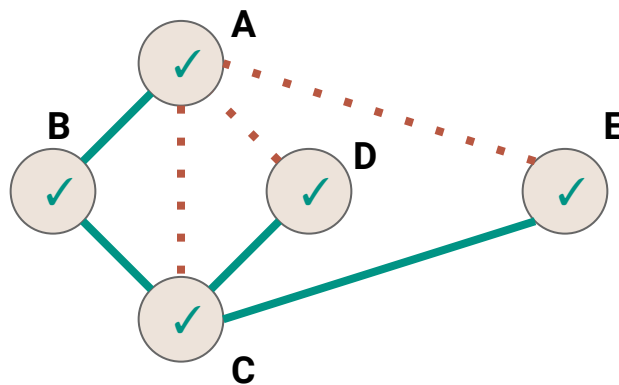
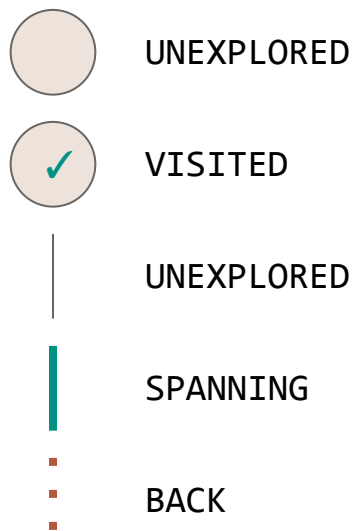
<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSOne(G,A)	(→ B, C, D, E)
DFSOne(G,B)	(→ A, C)
DFSOne(G,C)	(→ B, A, D, E)
DFSOne(G,E)	(→ A, C)

Detailed Example



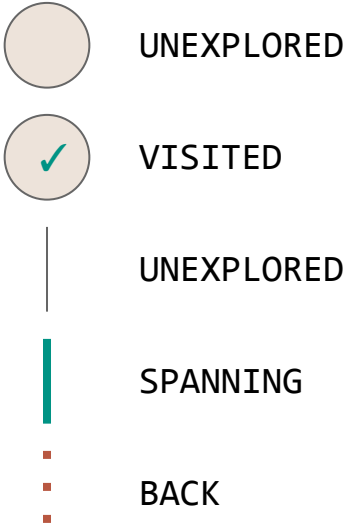
<u>Call Stack</u>	<u>(→ edges to list)</u>
DFS(G)	
DFSone(G,A)	(→ B, C, D, E)
DFSone(G,B)	(→ A, C)
DFSone(G,C)	(→ B, A, D, E)

Detailed Example

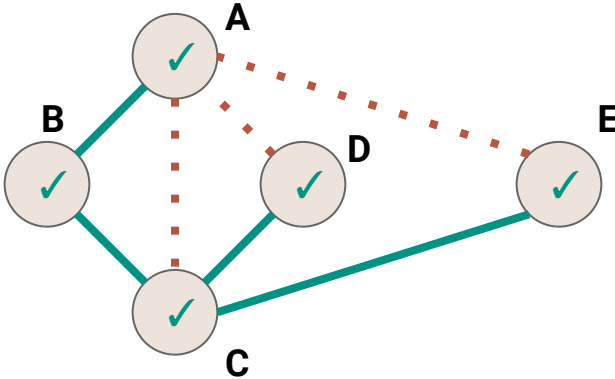


Call Stack (→ edges to list)
DFS(G)
DFSone(G,A) (→ B, C, D, E)
DFSone(G,B) (→ A, C)

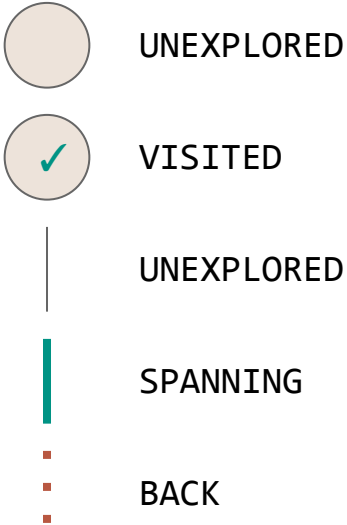
Detailed Example



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

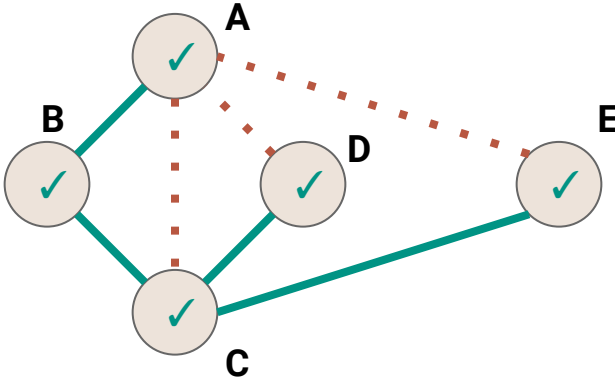


Detailed Example

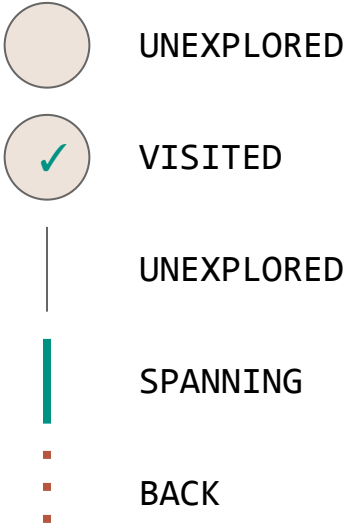


Call Stack
DFS(G)

(→ edges to list)

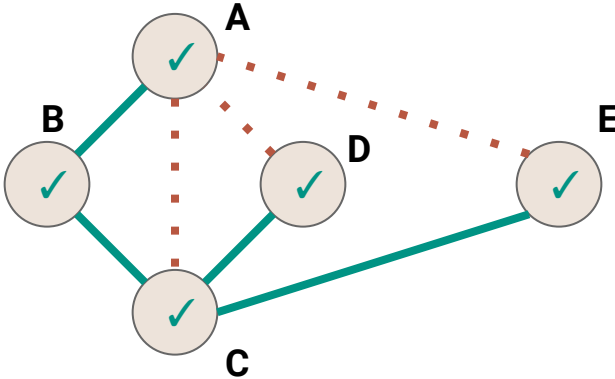


Detailed Example



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

```
1 public void DFS(Graph graph) {
2     for (Vertex v : graph.vertices) {
3         v.setLabel(UNEXPLORED);
4     }
5     for (Edge e : graph.edges) {
6         e.setLabel(UNEXPLORED);
7     }
8     for (Vertex v : graph.vertices) {
9         if (v.label == UNEXPLORED) {
10            DFSOne(graph, v);
11        }
12    }
13 }
```


DFS

```
1 public void DFS(Graph graph) {
2      $\Theta(|V|)$ 
3     for (Edge e : graph.edges) {
4         e.setLabel(UNEXPLORED);
5     }
6     for (Vertex v : graph.vertices) {
7         if (v.label == UNEXPLORED) {
8             DFSOne(graph, v);
9         }
10    }
11 }
```

DFS

```
1 public void DFS(Graph graph) {  
2      $\Theta(|V|)$   
3      $\Theta(|E|)$   
4     for (Vertex v : graph.vertices) {  
5         if (v.label == UNEXPLORED) {  
6             DFSOne(graph, v);  
7         }  
8     }  
9 }
```

DFS

```
1 public void DFS(Graph graph) {  
2      $\Theta(|V|)$   
3      $\Theta(|E|)$   
4     for (Vertex v : graph.vertices) {  
5         if (v.label == UNEXPLORED) {  
6              $\Theta(???)$   
7         }  
8     }  
9 }
```

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2     v.setLabel(VISITED);  
3     for (Edge e : v.outEdges) {  
4         if (e.label == UNEXPLORED) {  
5             Vertex w = e.to;  
6             if (w.label == UNEXPLORED) {  
7                 e.setLabel(SPANNING);  
8                 DFSOne(graph, w);  
9             } else {  
10                e.setLabel(BACK);  
11            }  
12        }  
13    }}
```

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     for (Edge e : v.outEdges) {  
4         if (e.label == UNEXPLORED) {  
5              $\Theta(1)$   
6             if (w.label == UNEXPLORED) {  
7                  $\Theta(1)$   
8                  $\Theta(???)$   
9             } else {  
10                 $\Theta(1)$   
11            }  
12        }  
13    }}
```

Depth-First Search Complexity

How many times do we call DFS 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

```
1 public void DFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     for (Edge e : v.outEdges) {  
4         if (e.label == UNEXPLORED) {  
5              $\Theta(1)$   
6             if (w.label == UNEXPLORED) {  
7                  $\Theta(1)$   
8                  $\Theta(???)$   
9             } else {  
10                 $\Theta(1)$   
11            }  
12        }  
13    }}
```

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     for (Edge e : v.outEdges) {  
4          $\Theta(1)$   
5     }  
6 }
```

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     for (Edge e : v.outEdges) {  
4          $\Theta(1)$  As long as we use an adjacency list this will be able  
5     } to iterate through the adjacent edges in  $\Theta(\text{deg}(v))$   
6 }
```

time

DFSOne

```
1 public void DFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3      $\Theta(\text{deg}(v))$   
4 }
```

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`?

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

Depth-First Search Complexity

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

$$\begin{aligned} & \sum_{v \in V} O(\text{deg}(v)) \\ &= O\left(\sum_{v \in V} \text{deg}(v)\right) \end{aligned}$$

Depth-First Search Complexity

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

$$\begin{aligned} & \sum_{v \in V} O(\text{deg}(v)) \\ &= O\left(\sum_{v \in V} \text{deg}(v)\right) \\ &= O(2|E|) \end{aligned}$$

Depth-First Search Complexity

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

$$\begin{aligned} & \sum_{v \in V} O(\text{deg}(v)) \\ &= O\left(\sum_{v \in V} \text{deg}(v)\right) \\ &= O(2|E|) \\ &= O(|E|) \end{aligned}$$

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 | $O(V)$ |
| 2. Mark the edges UNVISITED | $O(E)$ |
| 3. Sum of all calls to DFSOne | $O(E)$ total |
| | <hr/> |
| | $O(V + E)$ |

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?

```
1 public void DFSOneNoRecursion(Graph graph, Vertex v) {
2     Stack<Vertex> todo = new Stack<>();
3     v.setLabel(VISITED);
4     todo.push(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.pop();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.push(w);
14                } else {
15                    e.setLabel(BACK);
16                }
17            }
18        }
19    }
20 }
```

```
1 public void DFSOneNoRecursion(Graph graph, Vertex v) {  
2     Stack<Vertex> todo = new Stack<>();  
3     v.setLabel(VISITED);  
4     todo.push(v);  
5     while (!todo.isEmpty()) {  
6         Vertex curr = todo.pop();  
7         for (Edge e : curr.outEdges) {  
8             if (e.label == UNEXPLORED) {  
9                 Vertex w = e.to;  
10                if (w.label == UNEXPLORED) {  
11                    w.setLabel(VISITED);  
12                    e.setLabel(SPANNING);  
13                    todo.push(w);  
14                } else {  
15                    e.setLabel(BACK);  
16                }  
17            }  
18        }  
19    }  
20 }  
21 }  
22 }  
23 }
```

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

```
1 public void DFSOneNoRecursion(Graph graph, Vertex v) {
2     Stack<Vertex> todo = new Stack<>();
3     v.setLabel(VISITED);
4     todo.push(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.pop();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.push(w);
14                } else {
15                    e.setLabel(BACK);
16                }
17            }
18        }
19    }
20 }
```

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


```
1 public void DFSOneNoRecursion(Graph graph, Vertex v) {
2     Stack<Vertex> todo = new Stack<>();
3     v.setLabel(VISITED);
4     todo.push(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.pop();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.push(w);
14                } else {
15                    e.setLabel(BACK);
16                }
17            }
18        }
19    }
20 }
```

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

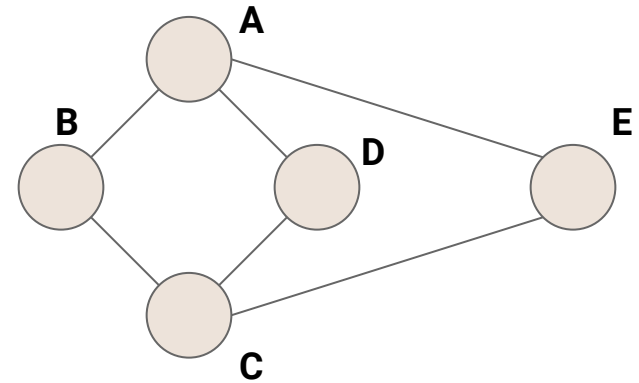


SPANNING

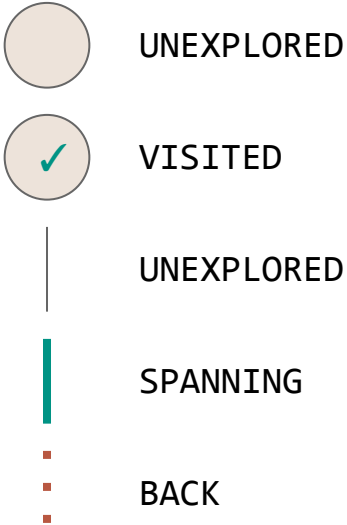


BACK

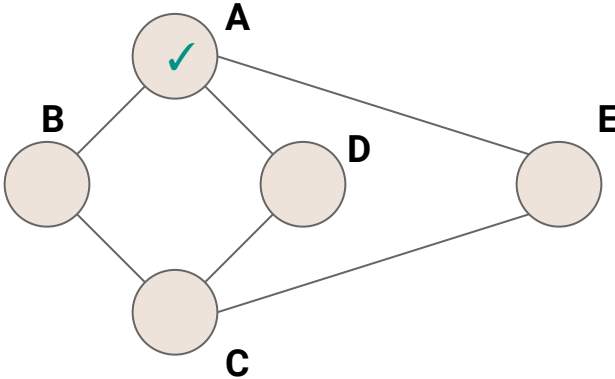
TODO Stack



Detailed Example



TODO Stack
A



Detailed Example



UNEXPLORED



VISITED



UNEXPLORED



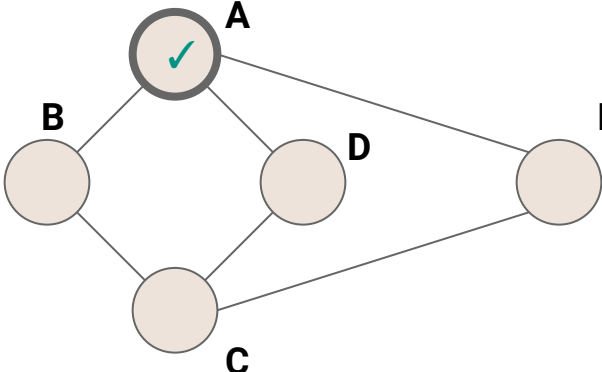
SPANNING



BACK

Current Vertex: A

TODO Stack



Detailed Example



UNEXPLORED



VISITED



UNEXPLORED



SPANNING



BACK

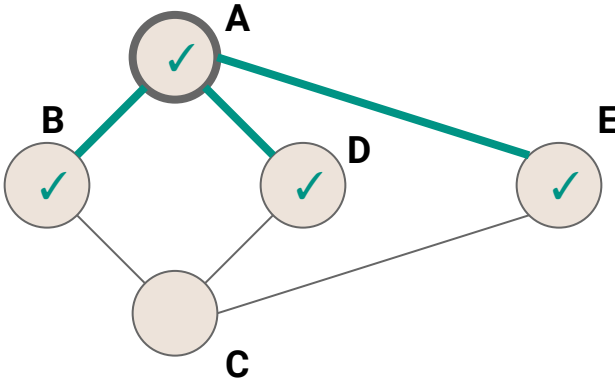
Current Vertex: A

TODO Stack

B

D

E



Detailed Example



UNEXPLORED



VISITED



UNEXPLORED



SPANNING



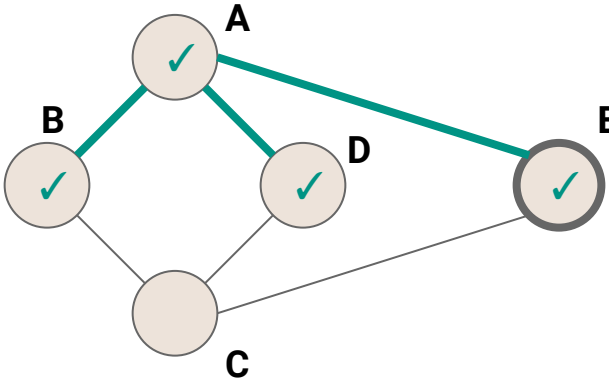
BACK

Current Vertex: E

TODO Stack

B

D



Detailed Example



UNEXPLORED



VISITED



UNEXPLORED



SPANNING

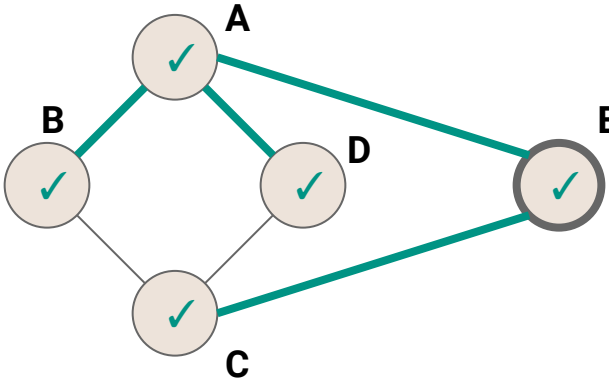


BACK

Current Vertex: E

TODO Stack

- B
- D
- C



Detailed Example



UNEXPLORED



VISITED



UNEXPLORED



SPANNING



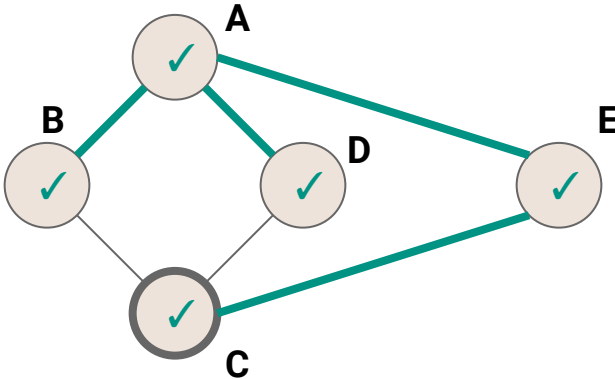
BACK

Current Vertex: C

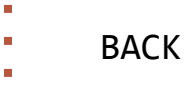
TODO Stack

B

D



Detailed Example

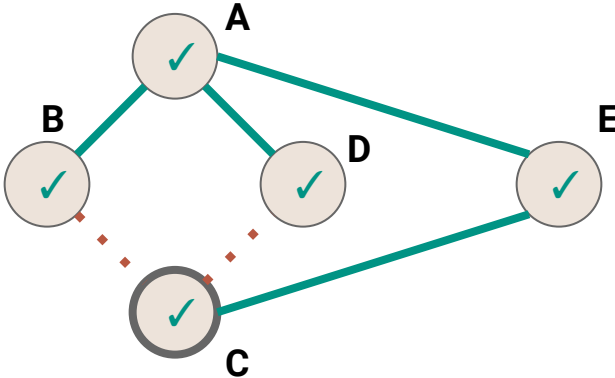


Current Vertex: C

TODO Stack

B

D



Detailed Example



UNEXPLORED



VISITED



UNEXPLORED



SPANNING

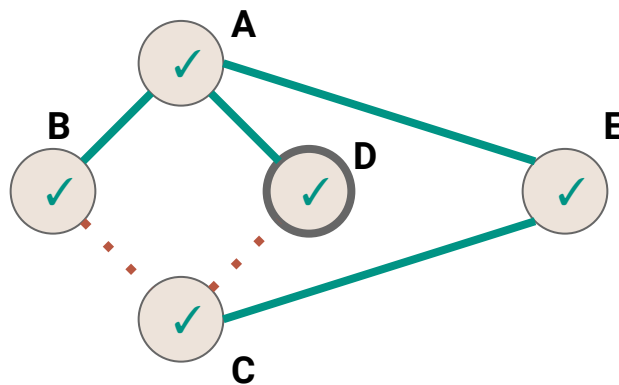


BACK

Current Vertex: D

TODO Stack

B



Detailed Example



UNEXPLORED



VISITED



UNEXPLORED



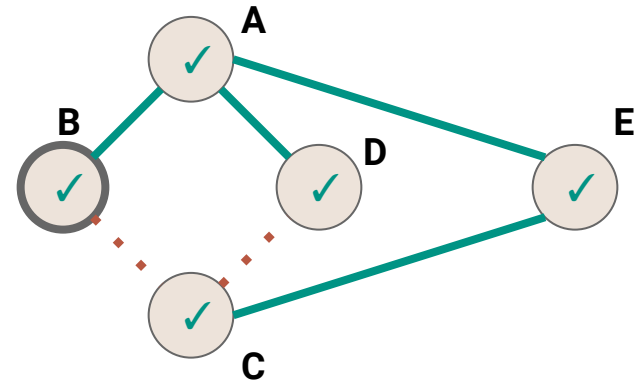
SPANNING



BACK

Current Vertex: B

TODO Stack



```
1 public void DFSOneNoRecursion(Graph graph, Vertex v) {
2     Stack<Vertex> todo = new Stack<>();
3     v.setLabel(VISITED);
4     todo.push(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.pop();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    curr.setLabel(VISITED);
12                    e.setLabel(SEARCHED);
13                    todo.push(w);
14                } else {
15                    e.setLabel(BACK);
16                }
17            }
18        }
19    }
20 }
```

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 $G = (V, E)$
- 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 $O(|V| + |E|)$, with memory overhead $O(|V|)$

Breadth-First Search

Primary Goals

- 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

```
1 public void BFS(Graph graph) {
2     for (Vertex v : graph.vertices) {
3         v.setLabel(UNEXPLORED);
4     }
5     for (Edge e : graph.edges) {
6         e.setLabel(UNEXPLORED);
7     }
8     for (Vertex v : graph.vertices) {
9         if (v.label == UNEXPLORED) {
10            BFSOne(graph, v);
11        }
12    }
13 }
```

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


```
1 public void BFSOne(Graph graph, Vertex v) {
2     Queue<Vertex> todo = new Queue<>();
3     v.setLabel(VISITED);
4     todo.enqueue(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.dequeue();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.enqueue(w);
14                } else {
15                    e.setLabel(CROSS);
16                }
17            }
18        }
19    }
20 }
```

```
1 public void BFSOne(Graph graph, Vertex v) {
2     Queue<Vertex> todo = new Queue<>();
3     v.setLabel(VISITED);
4     todo.enqueue(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.dequeue();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.enqueue(w);
14                } else {
15                    e.setLabel(CROSS);
16                }
17            }
18        }
19    }
20 }
```

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

```
1 public void BFSOne(Graph graph, Vertex v) {
2     Queue<Vertex> todo = new Queue<>();
3     v.setLabel(VISITED);
4     todo.enqueue(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.dequeue();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.enqueue(w);
14                } else {
15                    e.setLabel(CROSS);
16                }
17            }
18        }
19    }
20 }
```

Dequeue a vertex from the Queue and check all of its outgoing edges

```
1 public void BFSOne(Graph graph, Vertex v) {
2     Queue<Vertex> todo = new Queue<>();
3     v.setLabel(VISITED);
4     todo.enqueue(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.dequeue();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.enqueue(w);
14                } else {
15                    e.setLabel(CROSS);
16                }
17            }
18        }
19    }
20 }
```

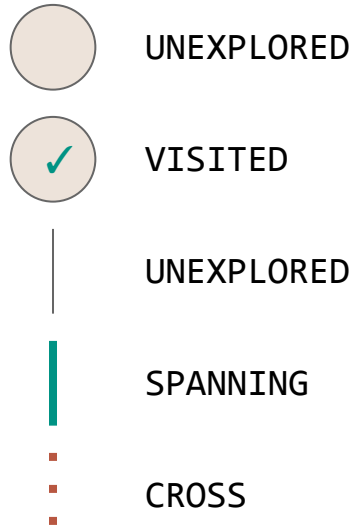
When we find a new vertex, mark it as VISITED, and add it to our TODO list.

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

```
1 public void BFSOne(Graph graph, Vertex v) {
2     Queue<Vertex> todo = new Queue<>();
3     v.setLabel(VISITED);
4     todo.enqueue(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.dequeue();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    w.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.enqueue(w);
14                } else {
15                    e.setLabel(CROSS);
16                }
17            }
18        }
19    }
20 }
```

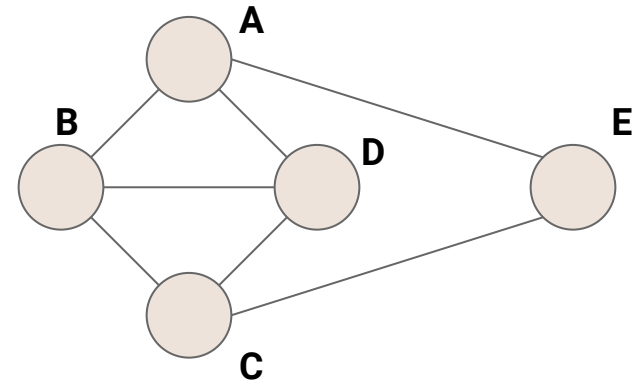
When doing BFS we label edges that return to visited vertices as CROSS edges

Detailed Example

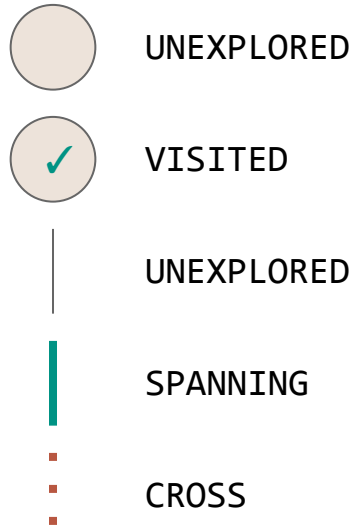


Call Stack

Work Queue

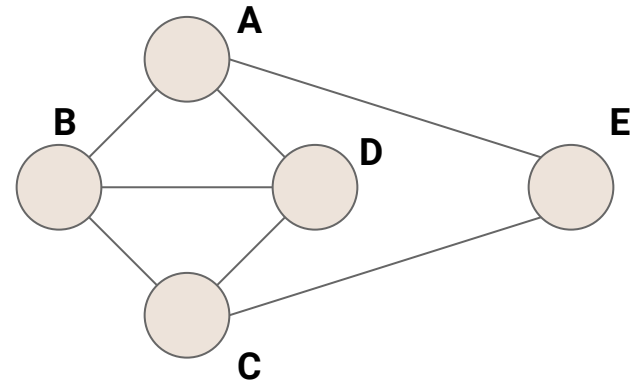


Detailed Example

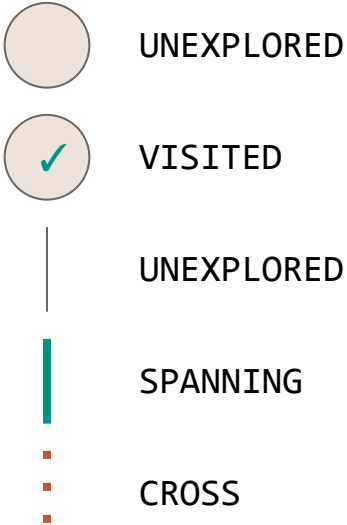


Call Stack
BFS(G)

Work Queue

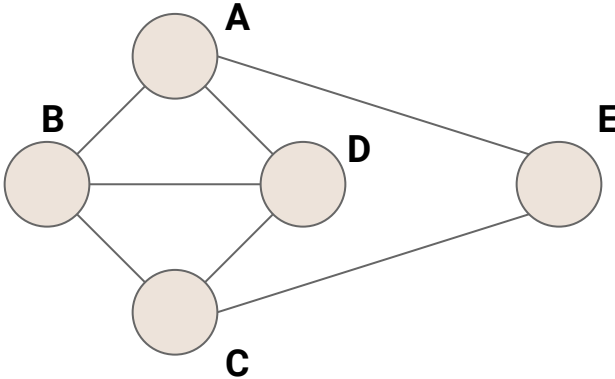


Detailed Example

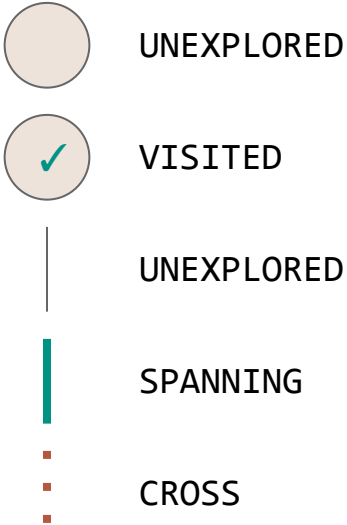


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue

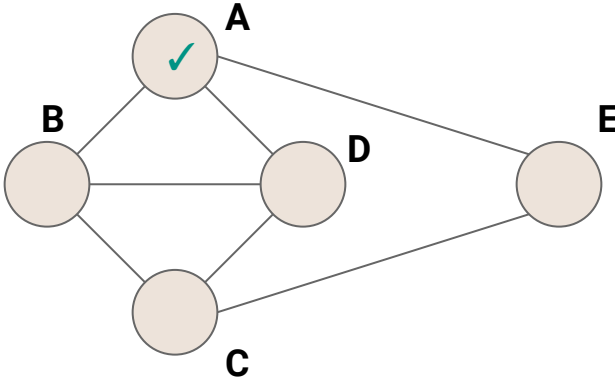


Detailed Example

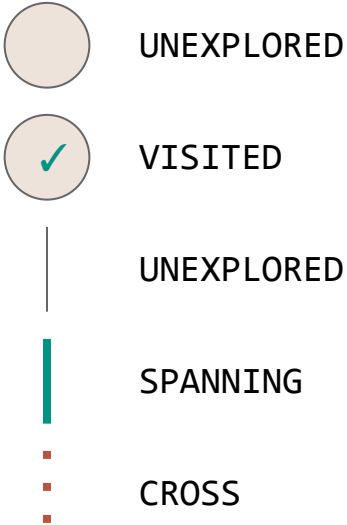


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A

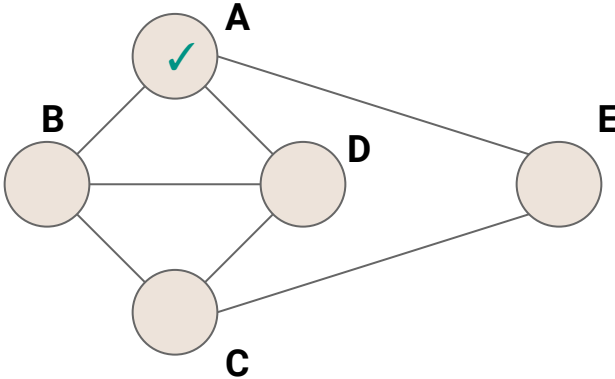


Detailed Example

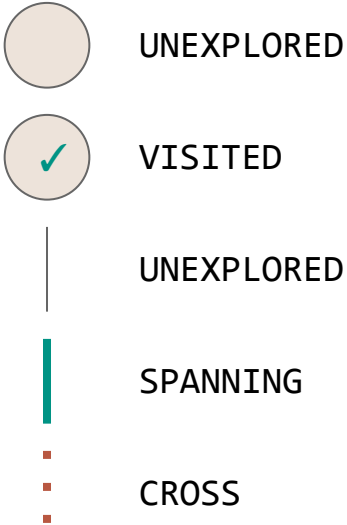


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
→ A



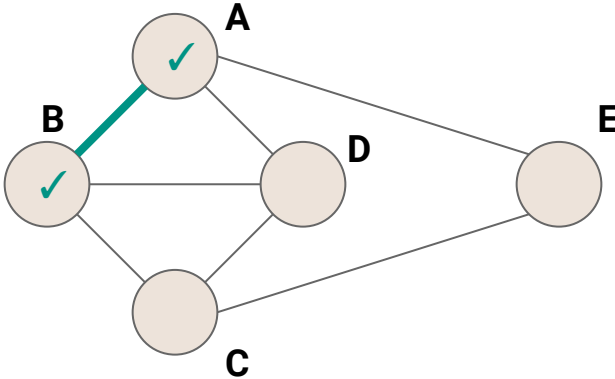
Detailed Example



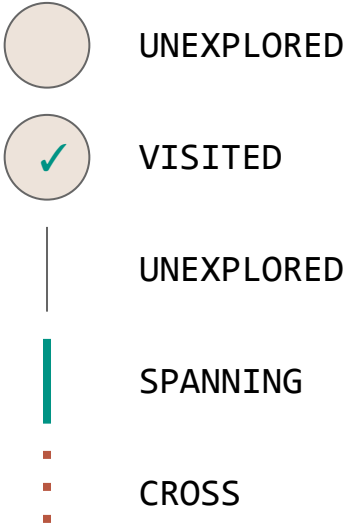
Call Stack
BFS(G)
BFSOne(G,A)



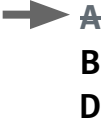
Work Queue
A
B



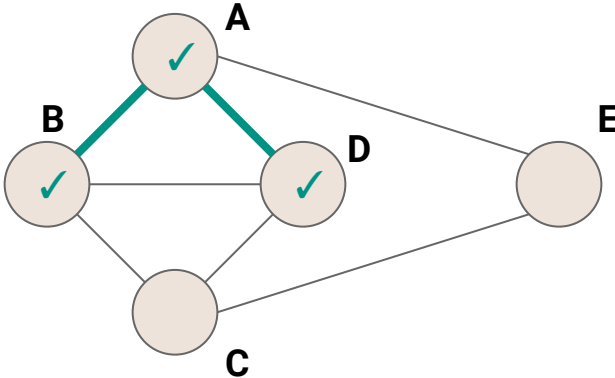
Detailed Example



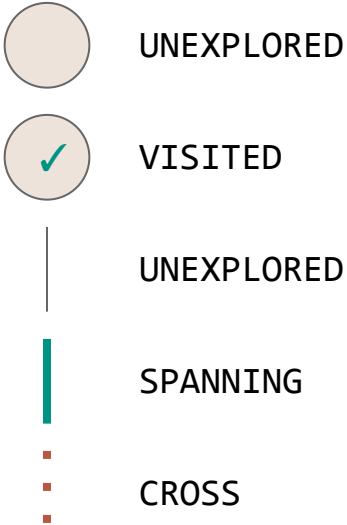
Call Stack
BFS(G)
BFSOne(G,A)



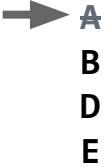
Work Queue
A
B
D



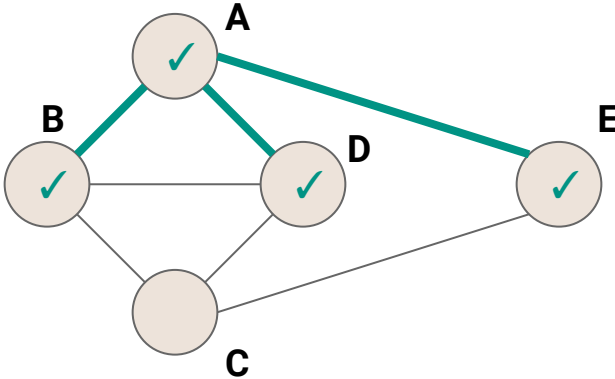
Detailed Example



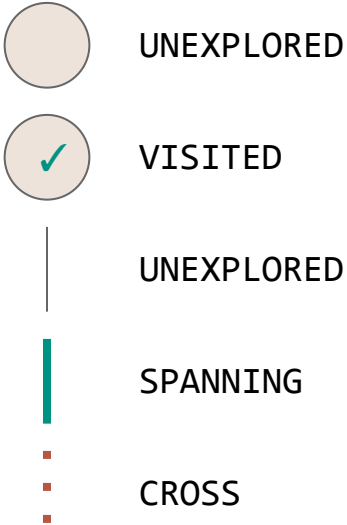
Call Stack
BFS(G)
BFSOne(G,A)



Work Queue
A
B
D
E



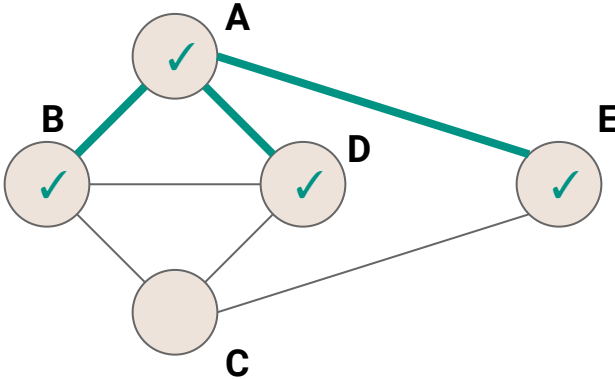
Detailed Example



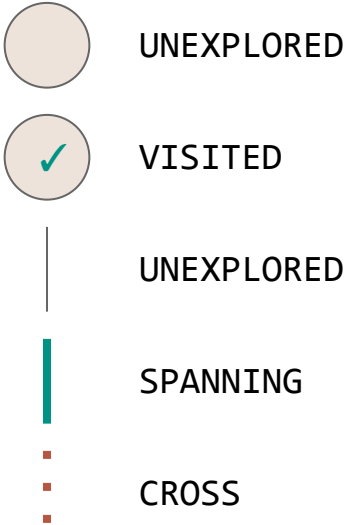
Call Stack
BFS(G)
BFSOne(G,A)



Work Queue
A
B
D
E



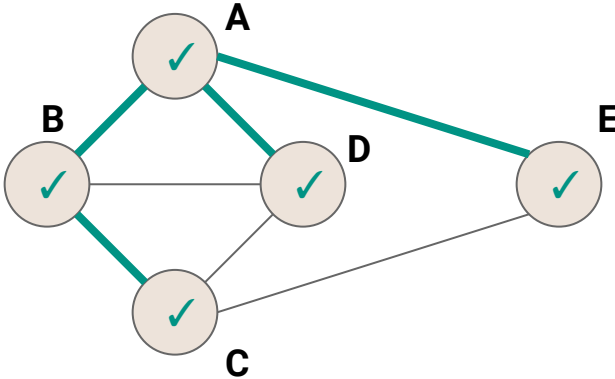
Detailed Example



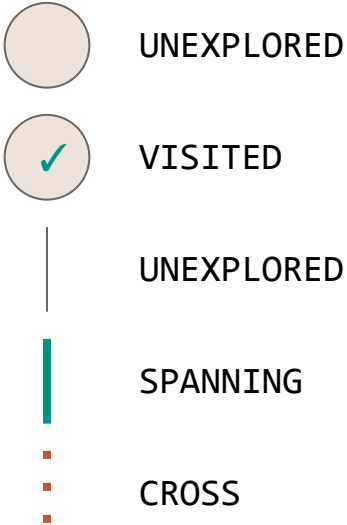
Call Stack
BFS(G)
BFSOne(G,A)



Work Queue
A
B
D
E
C



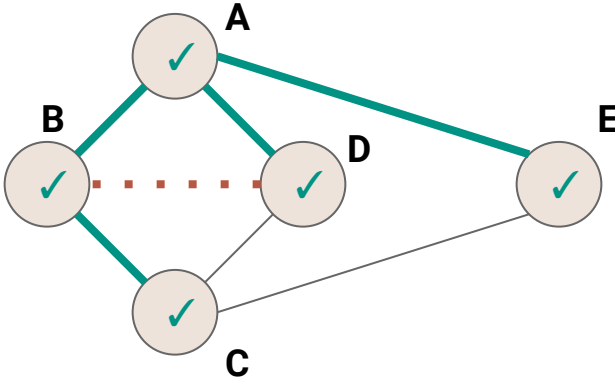
Detailed Example



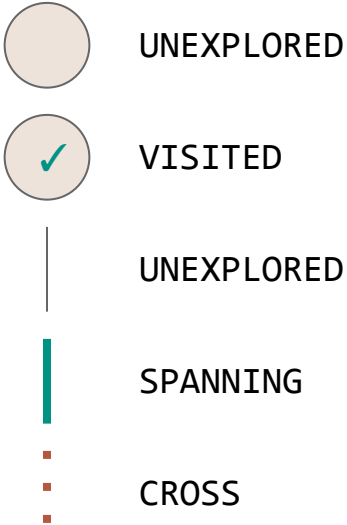
Call Stack
BFS(G)
BFSOne(G,A)



Work Queue
A
B
D
E
C

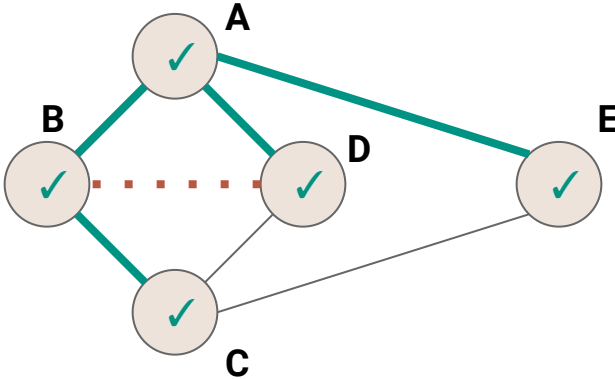


Detailed Example

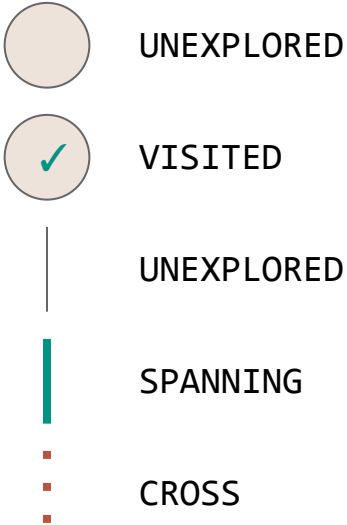


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A
B
~~D~~
E
C

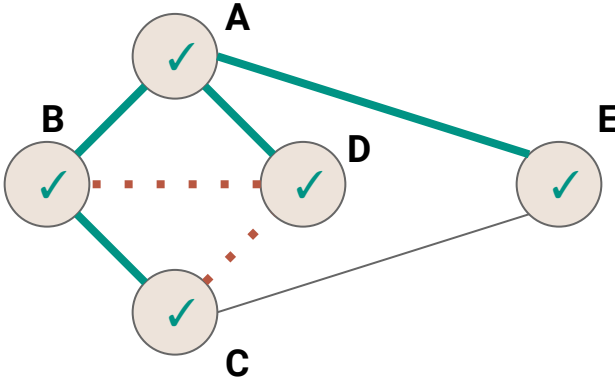


Detailed Example

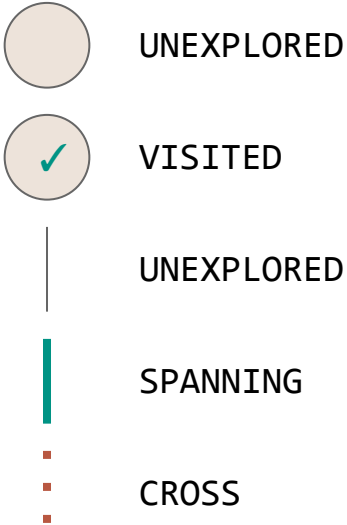


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A
B
~~D~~
E
C

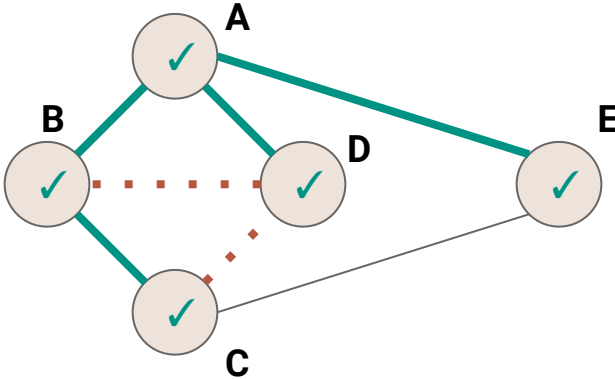


Detailed Example

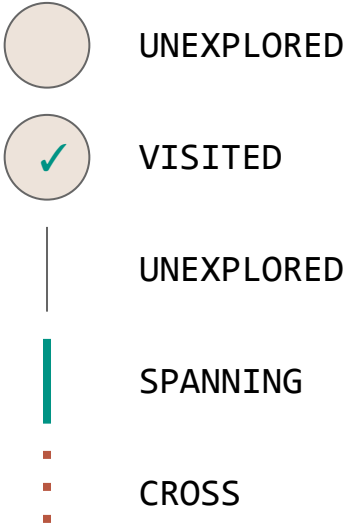


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A
B
D
E
C

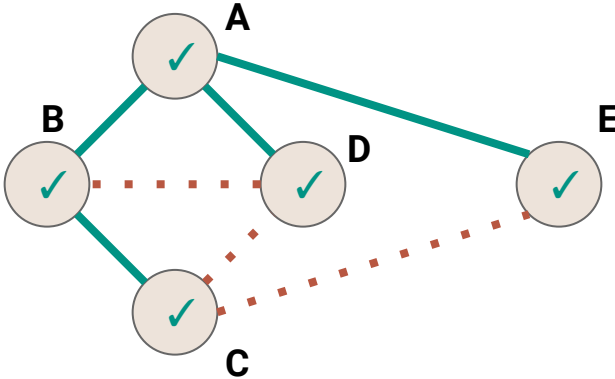


Detailed Example

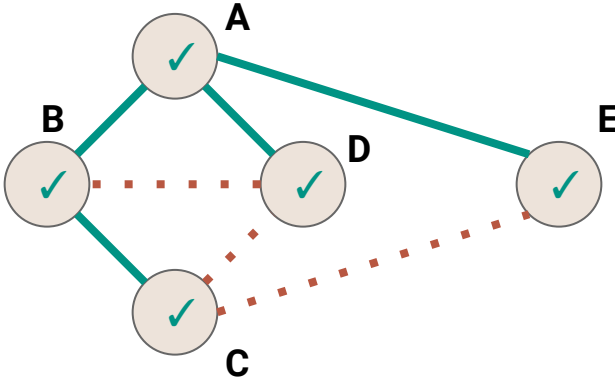
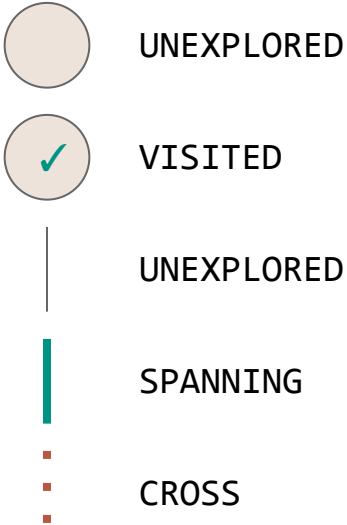


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A
B
D
E
C



Detailed Example



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

Work Queue
A
B
D
E
→ C

BFS Complexity

```
1 public void BFS(Graph graph) {  
2     for (Vertex v : graph.vertices) {  
3         v.setLabel(UNEXPLORED);  
4     }  
5     for (Edge e : graph.edges) {  
6         e.setLabel(UNEXPLORED);  
7     }  
8     for (Vertex v : graph.vertices) {  
9         if (v.label == UNEXPLORED) {  
10            BFSOne(graph, v);  
11        }  
12    }  
13 }
```

BFS Complexity

```
1 public void BFS(Graph graph) {  
2      $\Theta(|V|)$   
3      $\Theta(|E|)$   
4     for (Vertex v : graph.vertices) {  
5         if (v.label == UNEXPLORED) {  
6             BFSOne(graph, v);  
7         }  
8     }  
9 }
```

BFS Complexity

```
1 public void BFS(Graph graph) {  
2      $\Theta(|V|)$   
3      $\Theta(|E|)$   
4     for (Vertex v : graph.vertices) {  
5         if (v.label == UNEXPLORED) {  
6              $\Theta(???)$   
7         }  
8     }  
9 }
```



```
1 public void BFSOne(Graph graph, Vertex v) {
2     Queue<Vertex> todo = new Queue<>();
3     v.setLabel(VISITED);
4     todo.enqueue(v);
5     while (!todo.isEmpty()) {
6         Vertex curr = todo.dequeue();
7         for (Edge e : curr.outEdges) {
8             if (e.label == UNEXPLORED) {
9                 Vertex w = e.to;
10                if (w.label == UNEXPLORED) {
11                    curr.setLabel(VISITED);
12                    e.setLabel(SPANNING);
13                    todo.enqueue(w);
14                } else {
15                    e.setLabel(CROSS);
16                }
17            }
18        }
19    }
20 }
```

```
1 public void BFSOne(Graph graph, Vertex v) {
2    $\Theta(1)$ 
3   while (!todo.isEmpty()) {
4     Vertex curr = todo.dequeue();
5     for (Edge e : curr.outEdges) {
6       if (e.label == UNEXPLORED) {
7         Vertex w = e.to;
8         if (w.label == UNEXPLORED) {
9           curr.setLabel(VISITED);
10          e.setLabel(SPANNING);
11          todo.enqueue(w);
12        } else {
13          e.setLabel(CROSS);
14        }
15      }
16    }
17  }
```

```
1 public void BFSOne(Graph graph, Vertex v) {
2      $\Theta(1)$ 
3     while (!todo.isEmpty()) {
4          $\Theta(1)$ 
5         for (Edge e : curr.outEdges) {
6             if (e.label == UNEXPLORED) {
7                 Vertex w = e.to;
8                 if (w.label == UNEXPLORED) {
9                     curr.setLabel(VISITED);
10                    e.setLabel(SPANNING);
11                    todo.enqueue(w);
12                } else {
13                    e.setLabel(CROSS);
14                }
15            }
16        }
17    }
18 }
```

```
1 public void BFSOne(Graph graph, Vertex v) {
2      $\Theta(1)$ 
3     while (!todo.isEmpty()) {
4          $\Theta(1)$ 
5         for (Edge e : curr.outEdges) {
6             if (e.label == UNEXPLORED) {
7                  $\Theta(1)$ 
8                 if (w.label == UNEXPLORED) {
9                      $\Theta(1)$ 
10                    todo.enqueue(w);
11                } else {
12                     $\Theta(1)$ 
13                }
14            }
15        }
16    }
17 }
```


```
1 public void BFSOne(Graph graph, Vertex v) {
2      $\Theta(1)$ 
3     while (!todo.isEmpty()) {
4          $\Theta(1)$ 
5         for (Edge e : curr.outEdges) {
6             if (e.label == UNEXPLORED) {
7                  $\Theta(1)$ 
8                 if (w.label == UNEXPLORED) {
9                      $\Theta(1)$ 
10                     $\Theta(1)$ 
11                } else {
12                     $\Theta(1)$ 
13                }
14            }
15        }
16    }
17 }
```

```
1 public void BFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     while (!todo.isEmpty()) {  
4          $\Theta(1)$   
5         for (Edge e : curr.outEdges) {  
6              $\Theta(1)$   
7         }  
8     }  
9 }
```

```
1 public void BFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     while (!todo.isEmpty()) {  
4          $\Theta(1)$   
5          $\Theta(\text{deg}(v))$   
6     }  
7 }  
8  
9
```

```
1 public void BFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     while (!todo.isEmpty()) {  
4          $\Theta(1)$   
5          $\Theta(\text{deg}(v))$   
6     }  
7 }  
8  
9
```

How many iterations will this while loop run?




```
1 public void BFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     while (!todo.isEmpty()) {  
4          $\Theta(1)$   
5          $\Theta(\text{deg}(v))$   
6     }  
7 }  
8  
9
```

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

```
1 public void BFSOne(Graph graph, Vertex v) {  
2      $\Theta(1)$   
3     while (!todo.isEmpty()) {  
4          $\Theta(1)$   
5          $\Theta(\text{deg}(v))$   
6     }  
7 }  
8  
9
```

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

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**?

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

Breadth-First Search Complexity

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

$$\begin{aligned} & \sum_{v \in V} O(\text{deg}(v)) \\ &= O\left(\sum_{v \in V} \text{deg}(v)\right) \end{aligned}$$

Breadth-First Search Complexity

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

$$\begin{aligned} & \sum_{v \in V} O(\text{deg}(v)) \\ &= O\left(\sum_{v \in V} \text{deg}(v)\right) \\ &= O(2|E|) \end{aligned}$$

Breadth-First Search Complexity

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

$$\begin{aligned} & \sum_{v \in V} O(\text{deg}(v)) \\ &= O\left(\sum_{v \in V} \text{deg}(v)\right) \\ &= O(2|E|) \\ &= O(|E|) \end{aligned}$$

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**

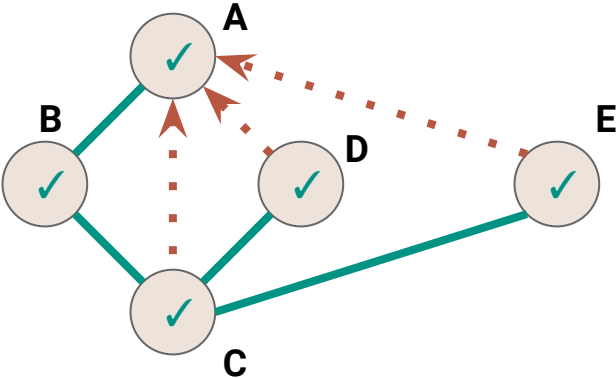
Breadth-First Search Complexity

In summary...

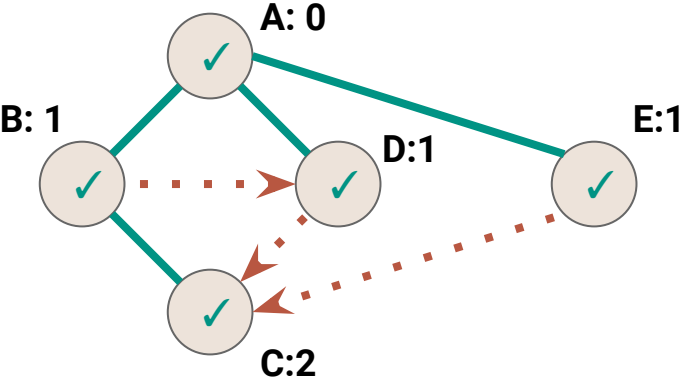
- | | |
|---------------------------------------|-----------------------|
| 1. Mark the vertices UNVISITED | $O(V)$ |
| 2. Mark the edges UNVISITED | $O(E)$ |
| 3. Add each vertex to the work queue | $O(V)$ |
| 4. Process each vertex | $O(E)$ total |
| | <hr/> |
| | $O(V + E)$ |

DFS vs BFS

DFS

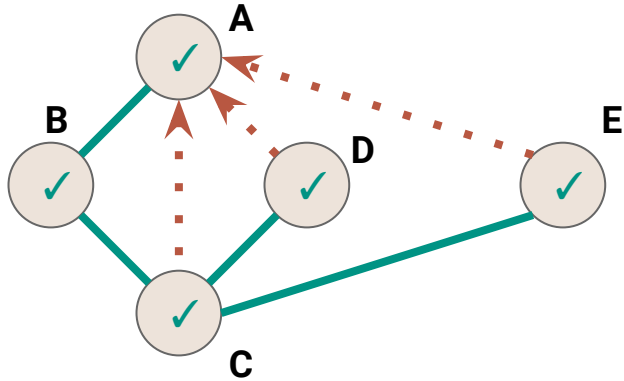


BFS

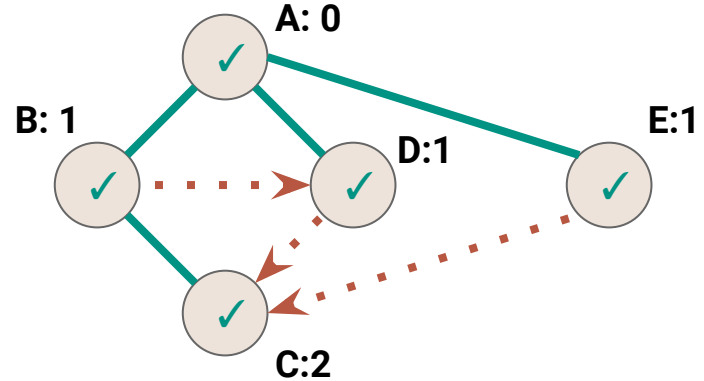


DFS vs BFS

DFS



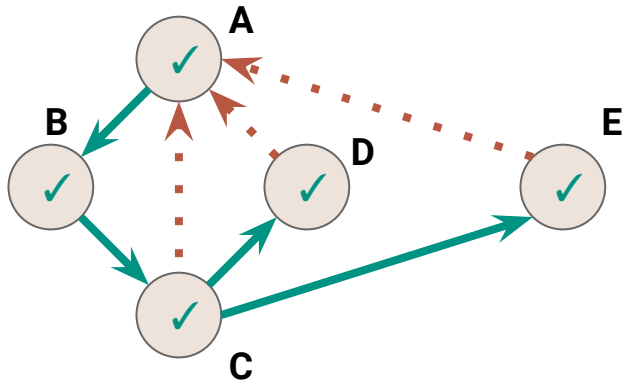
BFS



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

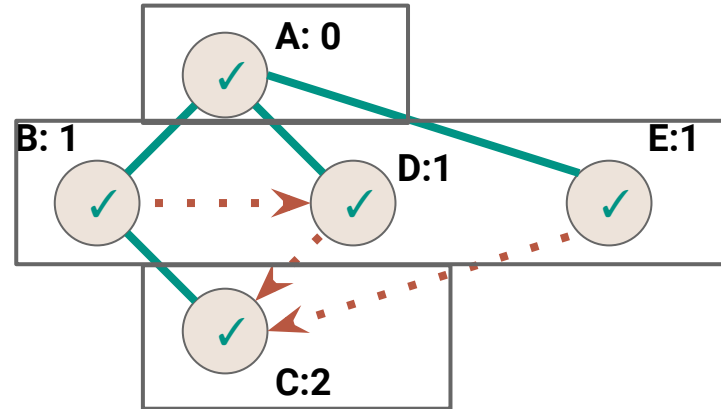
DFS vs BFS

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...