CSE 250
Data Structures

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

Graph Exploration (BFS)
Textbook Ch. 15.3

Announcements and Feedback

- PA2 Testing Phase due Sunday (remember, no grace days)
- Midterm Review Monday
- Midterm Wednesday

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

Depth-First Search Complexity

In summary…

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

Specializing DFS for Path-Finding

- Like with mazes, we can add a stack parameter to keep track of our path as we search
  - As soon as the target is reached, return the current stack
- DFSOne now returns either:
  - No Path, or…
  - Edge, Vertex, Edge, Vertex, …, Vertex

Specializing DFS for Cycle-Finding

- We can also utilize this extra stack parameter to find cycles
  - Return as soon as we attempt to visit an already visited node
- Our stack now contains a simple path from **S** to **V** concatenated with a simple cycle from **V** to **V**

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

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

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

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

BFSOne

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

BFSOne

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

Create a work list of "nodes to visit", and add our start node

BFSOne

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

Loop as long as we have more work to do, dequeueing an item each iteration

BFSOne

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

If we find a new node, enqueue it to be explored, and label appropriately

BFSOne

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

…if we have already visited it, then don't add it to be explored

Detailed Example

UNEXPLORED

VISITED

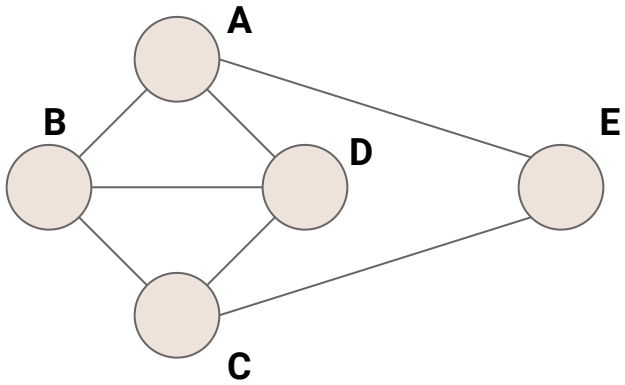
UNEXPLORED

SPANNING

CROSS

Call Stack

Work Queue



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

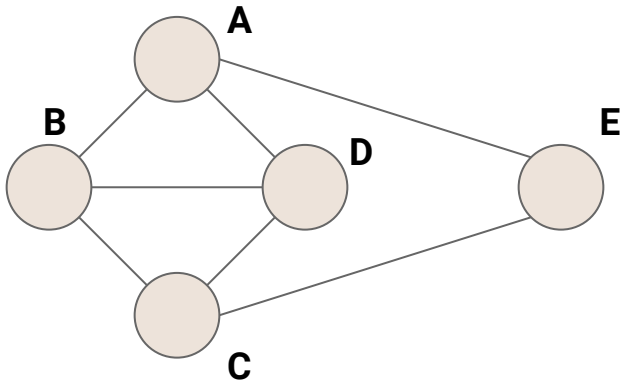
SPANNING

CROSS

Call Stack

BFS(G)

Work Queue



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

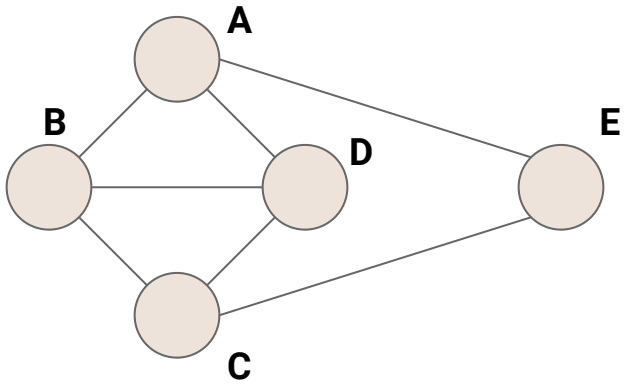
SPANNING

CROSS

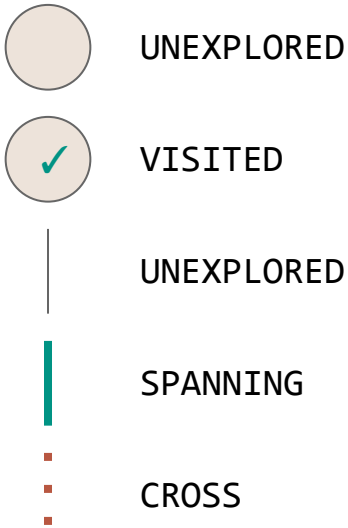
Call Stack

BFS(G)
BFSOne(G,A)

Work Queue

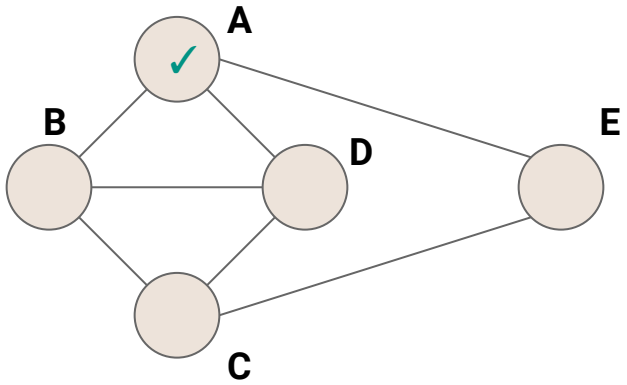


Detailed Example

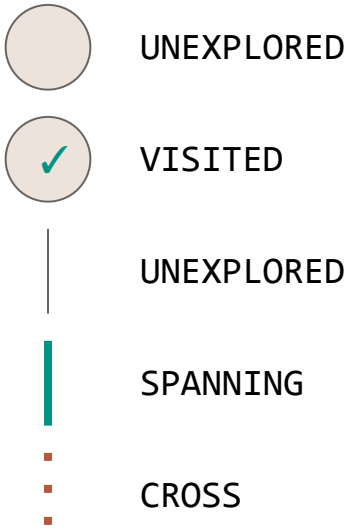


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A

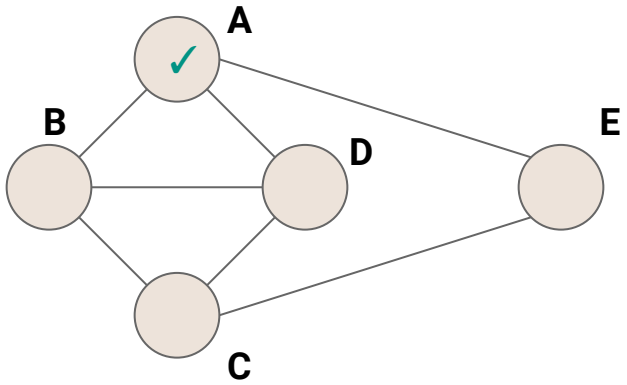


Detailed Example

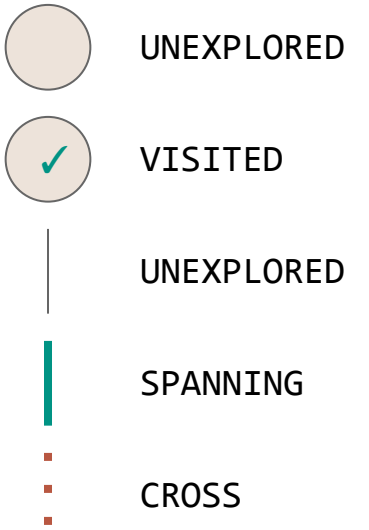


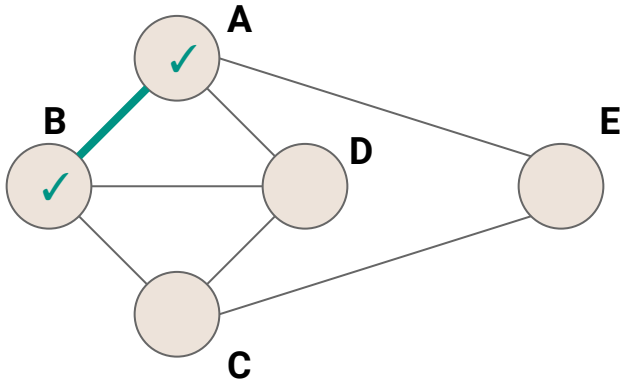
Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~



Detailed Example

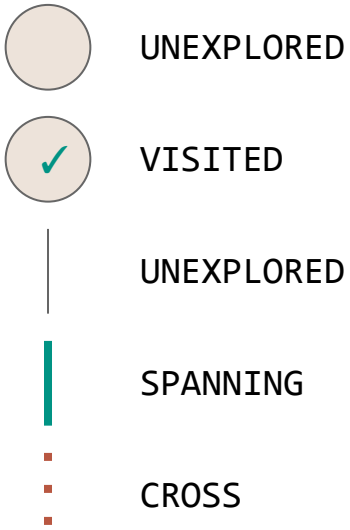


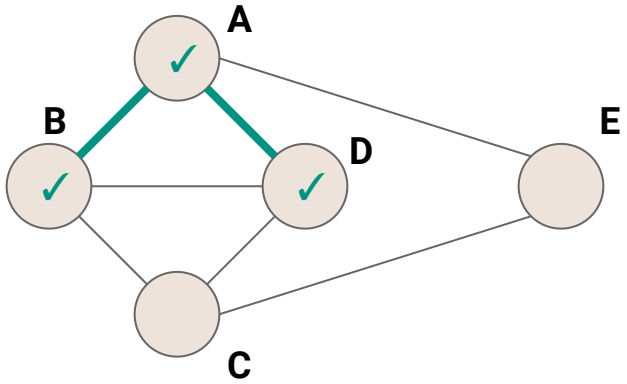


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B

Detailed Example

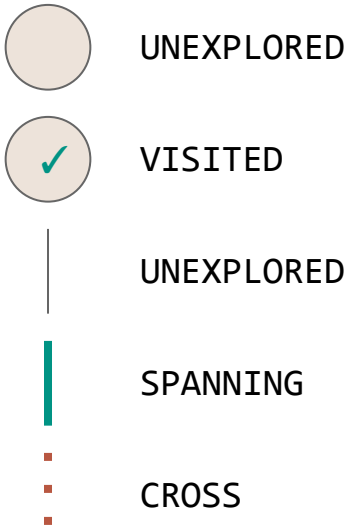


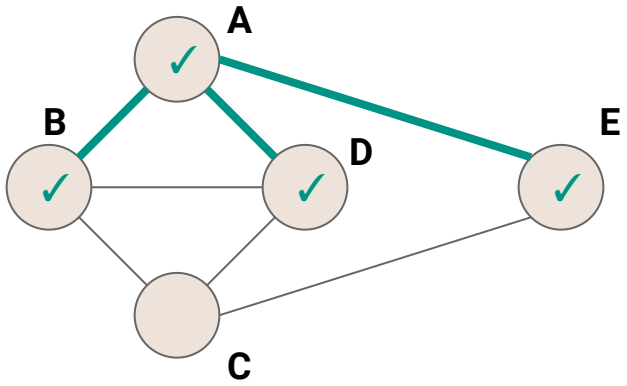


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~**A**~~
B
D

Detailed Example

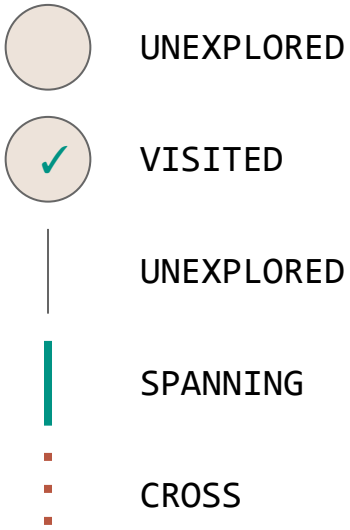


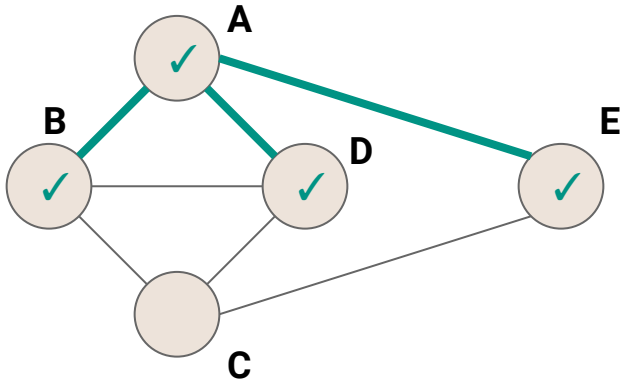


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B
D
E

Detailed Example

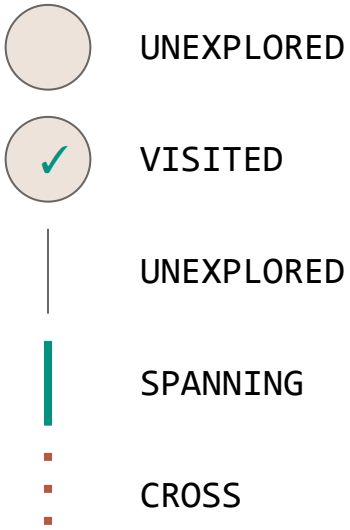


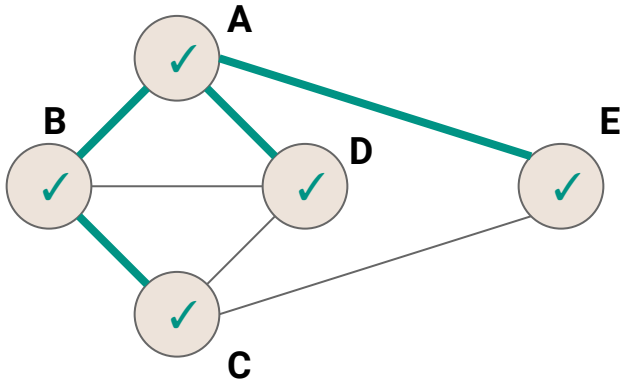


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~**A**~~
~~**B**~~
D
E

Detailed Example

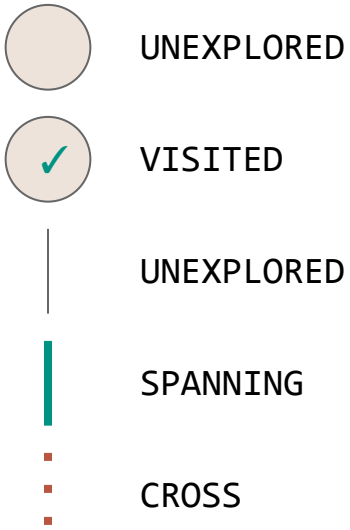


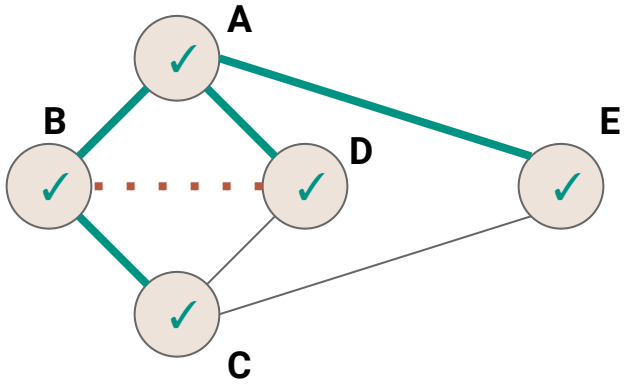


Call Stack
BFS(G)
BFSOne(G,A)

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

Detailed Example





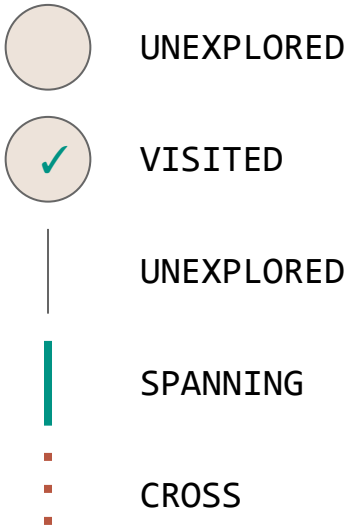
Call Stack

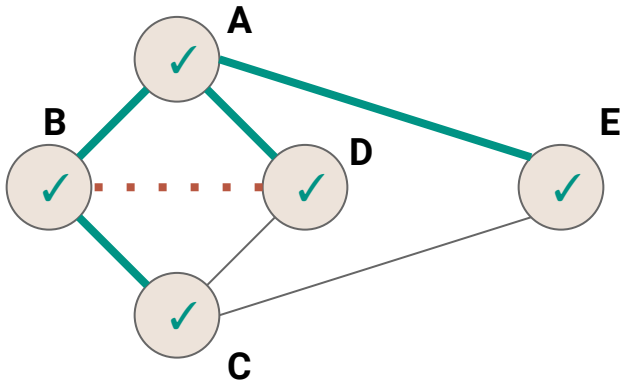
BFS(G)
BFSOne(G,A)

Work Queue

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

Detailed Example

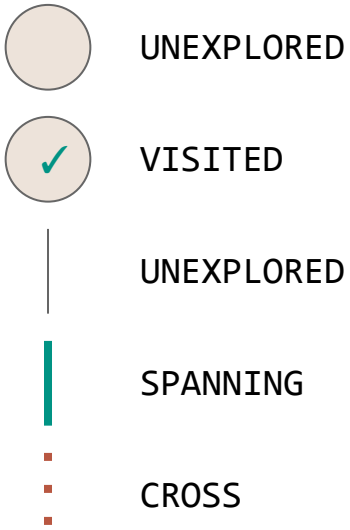


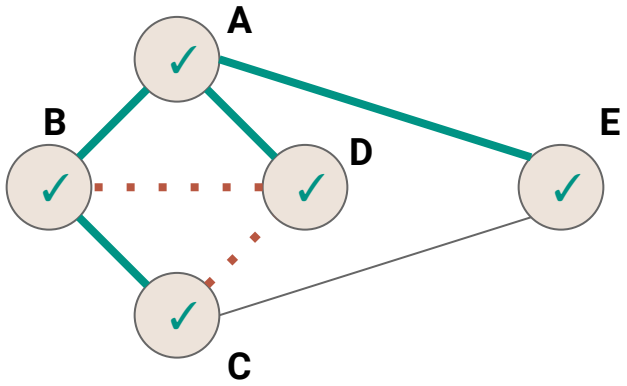


Call Stack
BFS(G)
BFSOne(G,A)

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

Detailed Example





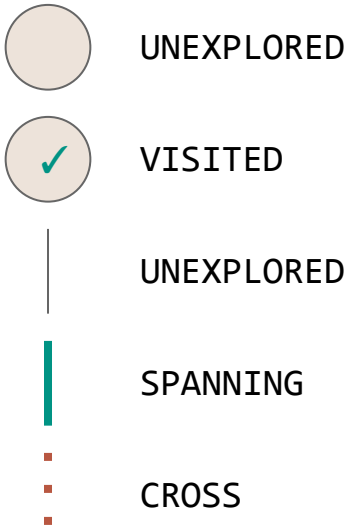
Call Stack

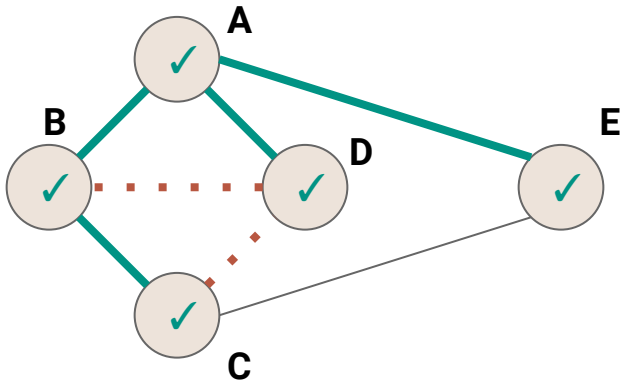
BFS(G)
BFSOne(G,A)

Work Queue

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

Detailed Example

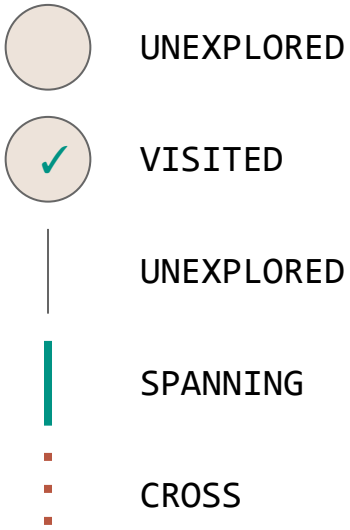


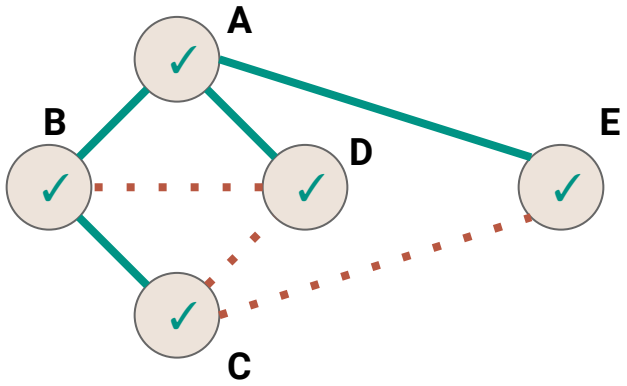


Call Stack
BFS(G)
BFSOne(G,A)

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

Detailed Example

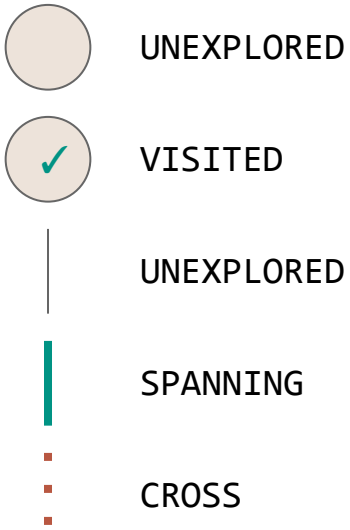


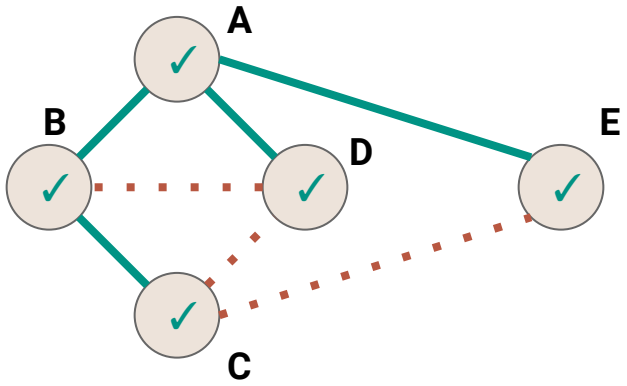


Call Stack
BFS(G)
BFSOne(G,A)

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

Detailed Example





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

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

Detailed Example


UNEXPLORED


VISITED

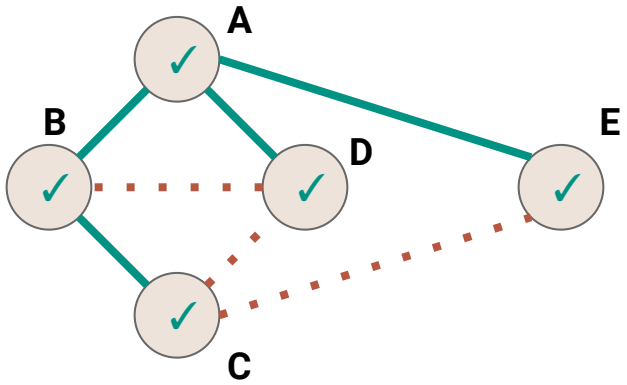
UNEXPLORED

SPANNING

CROSS

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

Work Queue



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

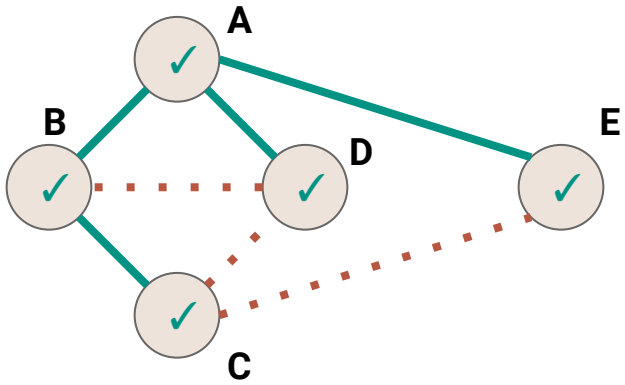
SPANNING

CROSS

Call Stack

BFS(G)
BFSOne(G,C)

Work Queue



Detailed Example


UNEXPLORED


VISITED

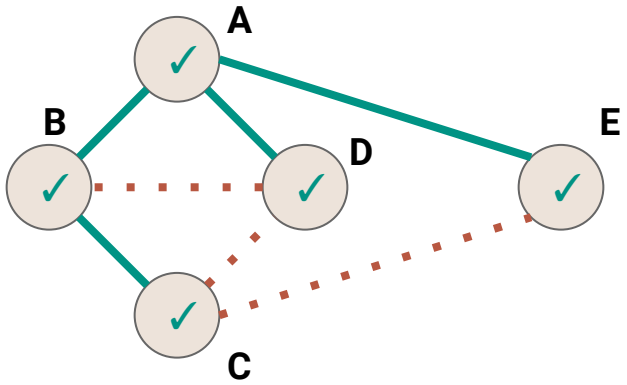
UNEXPLORED

SPANNING

CROSS

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

Work Queue



Detailed Example


UNEXPLORED


VISITED

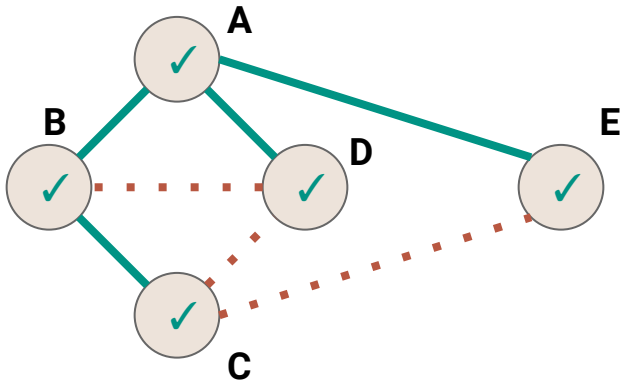
UNEXPLORED

SPANNING

CROSS

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

Work Queue



BFSOne - Adding Distance

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

BFSOne - Adding Distance

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex, Int]()
  work.enqueue((start,0))
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    (v, level) = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, level + 1))
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

Detailed Example


UNEXPLORED


VISITED

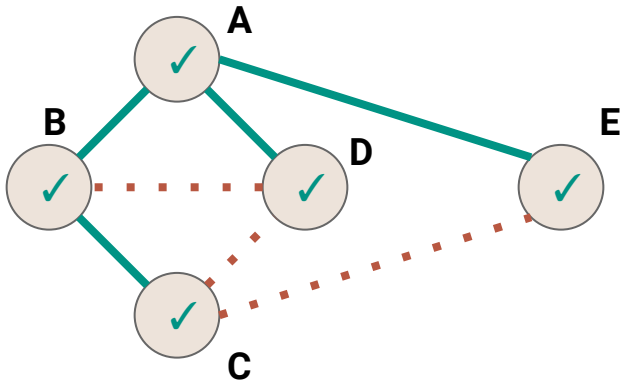
UNEXPLORED

SPANNING

CROSS

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

Work Queue



Detailed Example


UNEXPLORED


VISITED

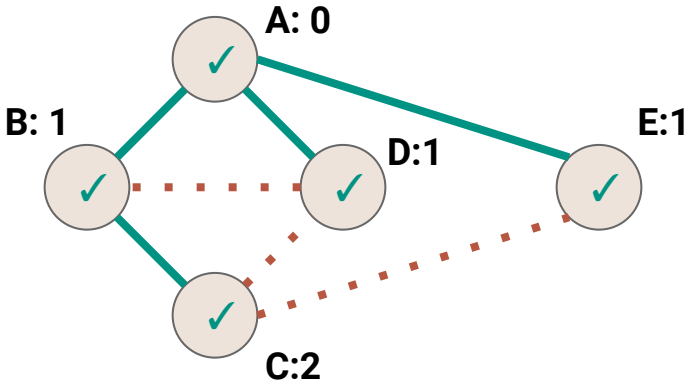
UNEXPLORED

SPANNING

CROSS

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

Work Queue



BFS - Complexity

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

BFS - Complexity

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

BFS - Complexity

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

BFS - Complexity

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

BFS - Complexity

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

BFS - Complexity

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex, Int]()
  work.enqueue((start,0))
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    (v, level) = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, level + 1))
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

BFS - Complexity

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  /* O(1) */
  while (!work.isEmpty) {
    (v, level) = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, level + 1))
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

BFS - Complexity

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  /* O(1) */
  while (!work.isEmpty) {
    /* O(1) */
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, level + 1))
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

BFS - Complexity

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  /* O(1) */
  while (!work.isEmpty) {
    /* O(1) */
    /* O(deg(v)) times */ {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, level + 1))
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

BFS - Complexity

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  /* O(1) */
  while (!work.isEmpty) {
    /* O(1) */
    /* O(deg(v)) times */ {
      /* O(1) */{
        /* O(1) */
        /* O(1) */{
          /* O(1) */
          /* O(1) */
          /* O(1) */
        } else {
          /* O(1) */
        }
      }
    }
  }
}
```

BFS - Complexity

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  /* O(1) */
  while (!work.isEmpty) {
    /* O(1) */
    /* O(deg(v)) times */ {
      /* O(1) */{
        /* O(1) */
        /* O(1) */{
          /* O(1) */
          /* O(1) */
          /* O(1) */
        } else {
          /* O(1) */
        }
      }
    }
  }
}
```

Each vertex is enqueued exactly once

BFS - Complexity

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  /* O(1) */
  while (!work.isEmpty) {
    /* O(1) */
    /* O(deg(v)) times */ {
      /* O(1) */{
        /* O(1) */
        /* O(1) */{
          /* O(1) */
          /* O(1) */
          /* O(1) */
        } else {
          /* O(1) */
        }
      }
    }
  }
}
```

Each vertex is enqueued exactly once

The cost of each vertex, ***v***, is ***O*(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…

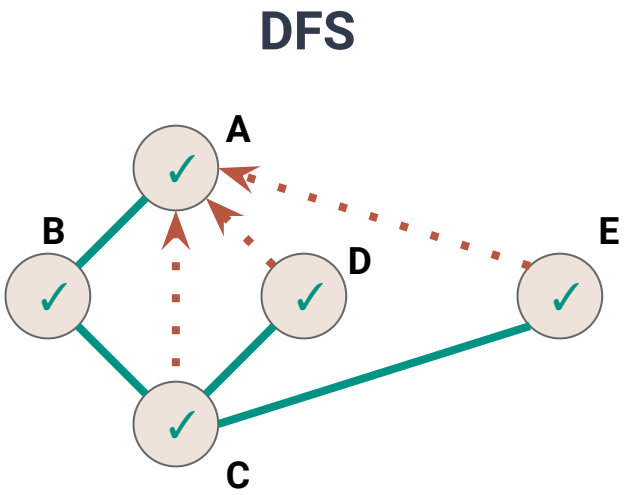
1. Mark the vertices `UNVISITED` $O(|V|)$
2. Mark the edges `UNVISITED` $O(|E|)$
3. Add each vertex to the work queue $O(|V|)$
4. Process each vertex $O(|E|)$

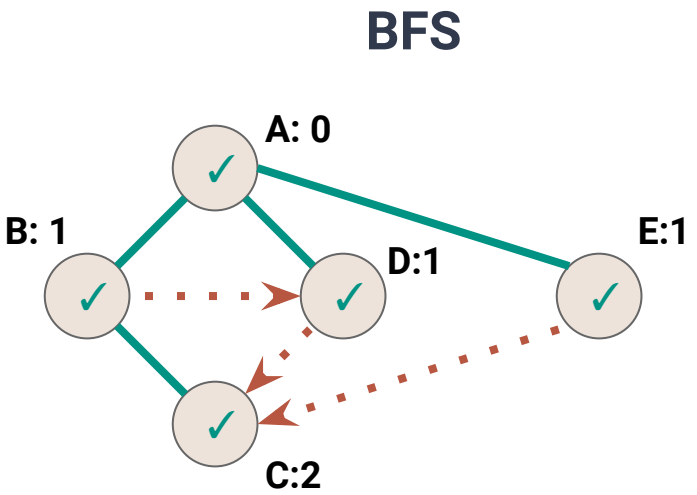
Breadth-First Search Complexity

In summary…

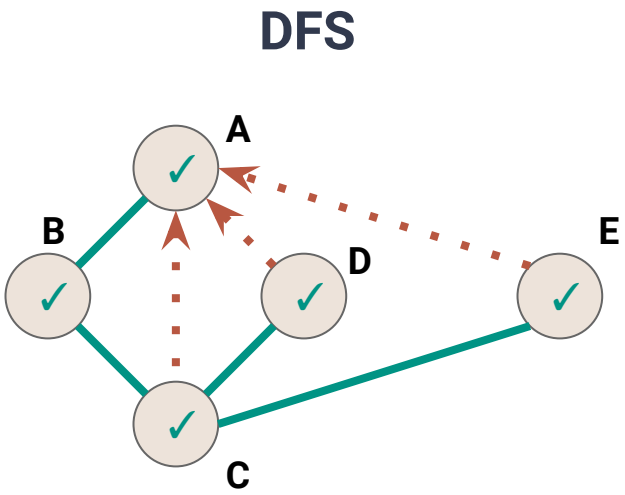
1.	Mark the vertices `UNVISITED`	$O(\|V\|)$
2.	Mark the edges `UNVISITED`	$O(\|E\|)$
3.	Add each vertex to the work queue	$O(\|V\|)$
4.	Process each vertex	$O(\|E\|)$
		$O(\|V\| + \|E\|)$

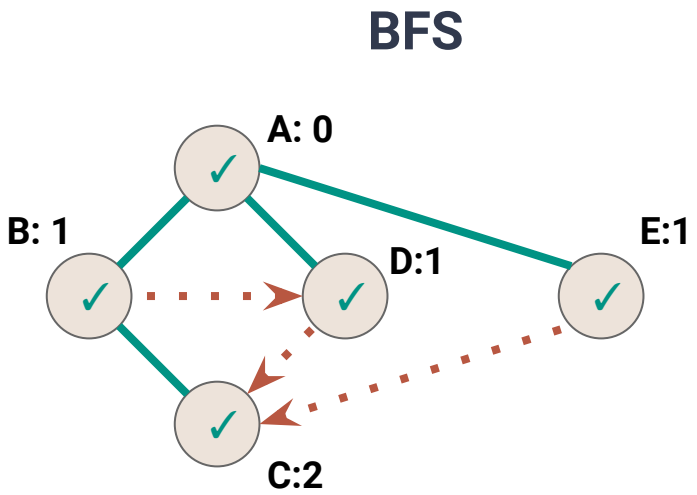
DFS vs BFS





DFS vs BFS

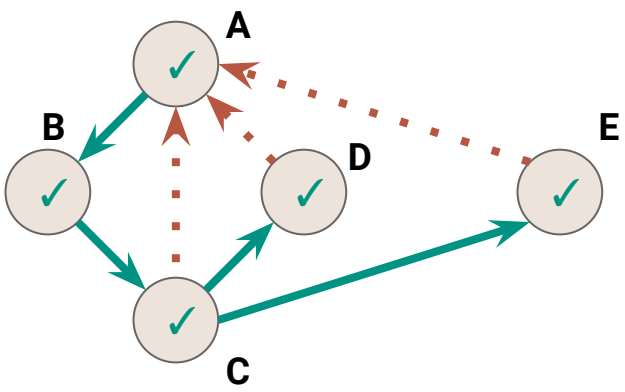




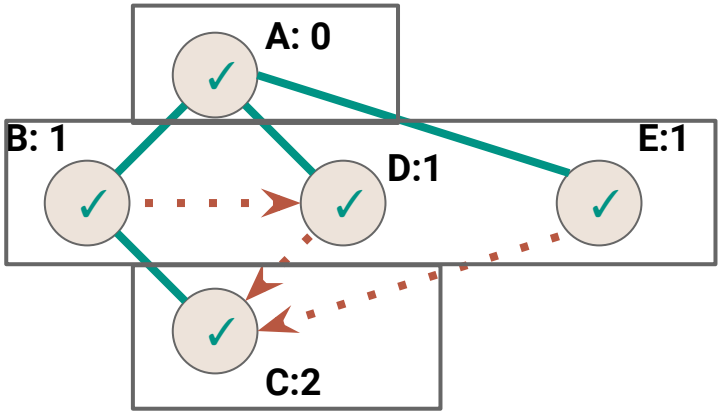
BACK Edge(*v*,*w*): ***w*** is an ancestor of ***v*** in the discovery tree

DFS vs BFS

DFS



BFS



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

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

Queues vs Stacks

Thought Experiment: How is the use of a Queue related to traversal order?

Queues vs Stacks

Thought Experiment: How is the use of a Queue related to traversal order? What if we used a Stack instead?

BFSOne

```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

DFSOneNoRecursion

```
def DFSOneNoRecursion(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Stack[Graph[...]#Vertex]()
  work.push(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.pop()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w = e.getOpposite(v)
        if(w.label == VertexLabel.UNEXPLORED){
          work.push(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.BACK)
        }
      }
    }
  }
}
```

Queues vs Stacks

Observation: The recursive version of DFS was using a Stack all along…the call stack!

DFS Traversal vs BFS Traversal

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