CSE 250
Data Structures

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

Shortest Path

Announcements

- WA2 has been posted. Due Sunday.
- Midterm exams should be returned by tonight.

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

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) */
        }
      }
    }
  }
}
```

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**?

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

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

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

$$= O(|E|)$$

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	$O(\|E\|)$
		$O(\|V\| + \|E\|)$

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

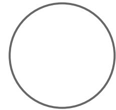
Observation: The recursive version of DFS was using a Stack all along…the call stack!

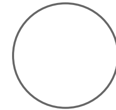
DFS Traversal vs BFS Traversal

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

Shortest Paths

Home

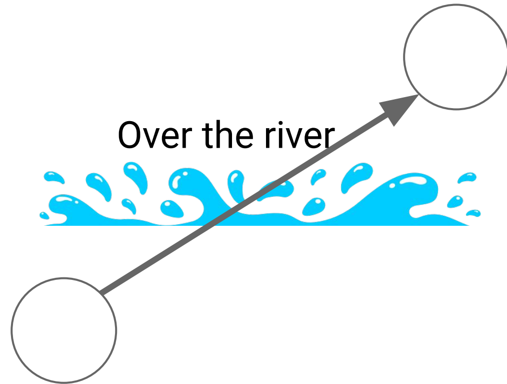


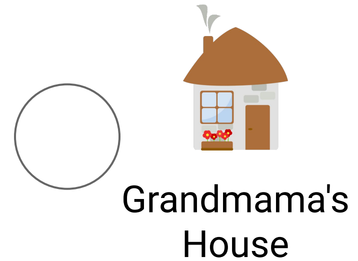


Grandmama's
House

Shortest Paths

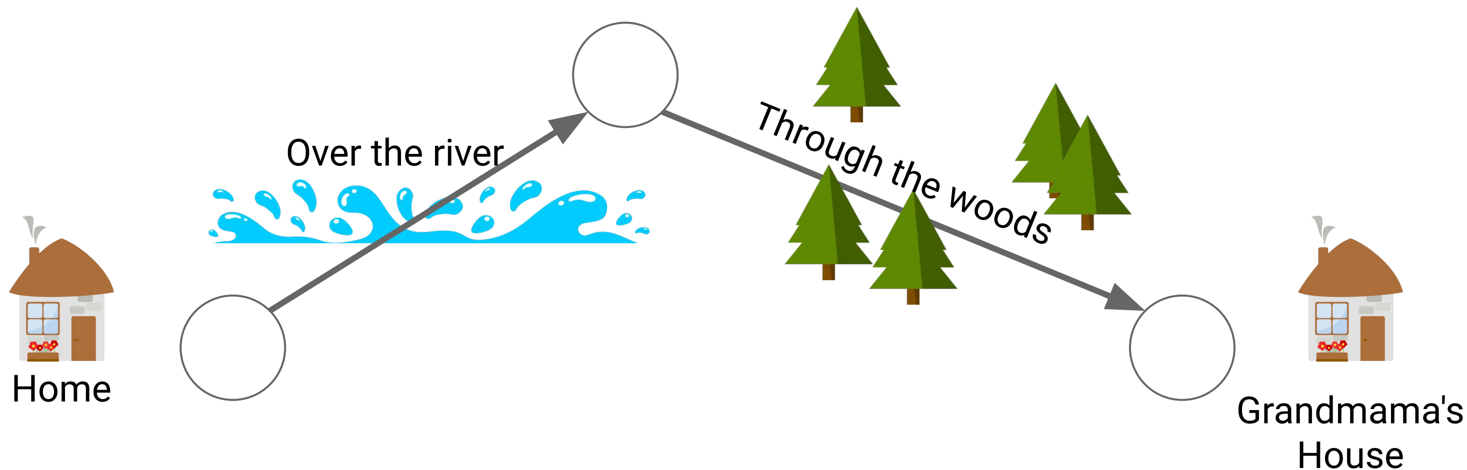
Home



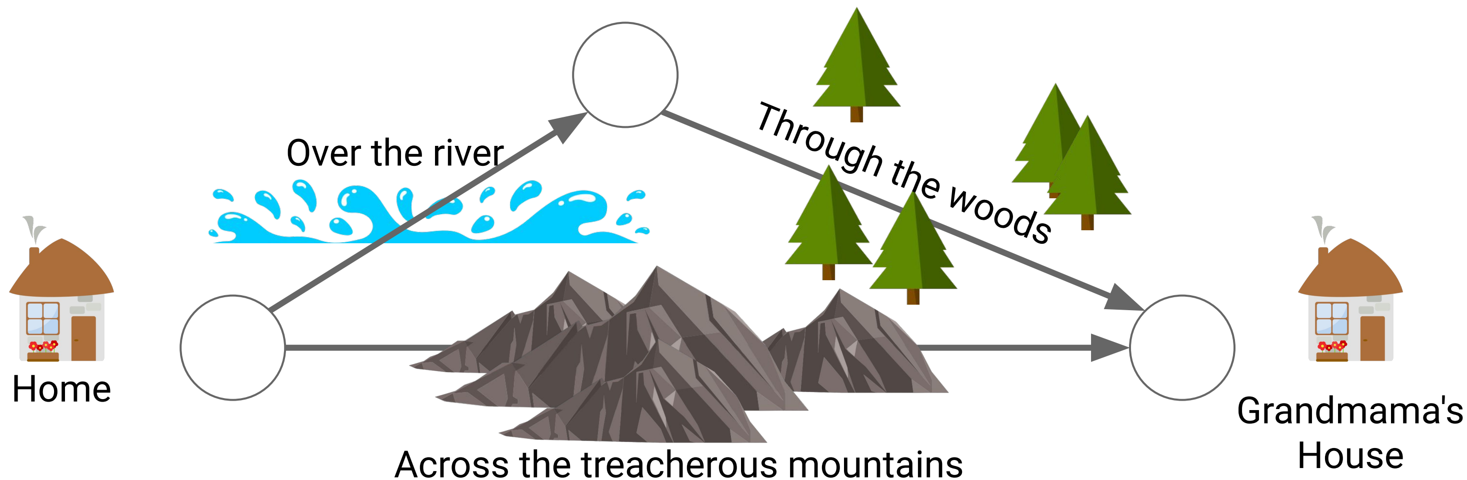


Grandmama's
House

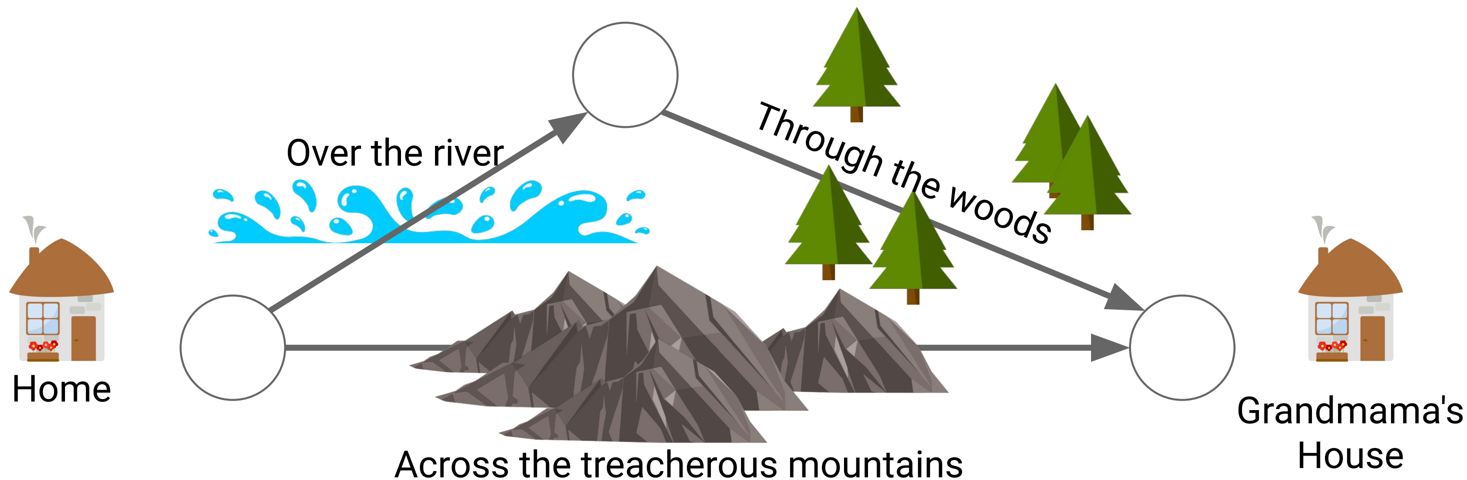
Shortest Paths



Shortest Paths

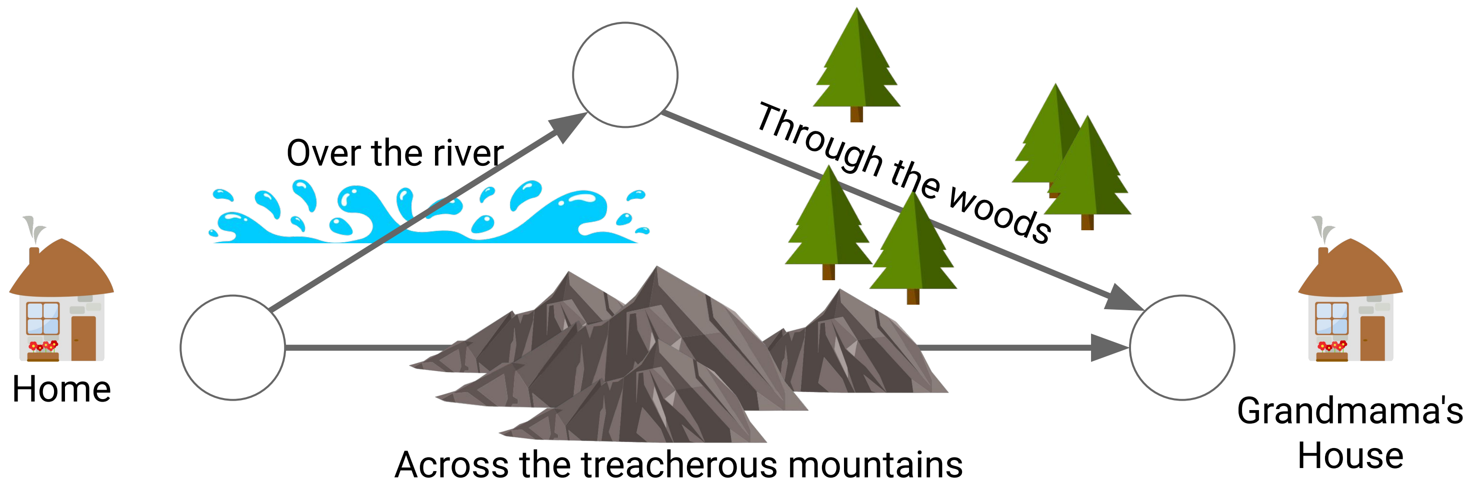


Shortest Paths



BFS will always find the path with the **fewest edges**...

Shortest Paths



BFS will always find the path with the **fewest edges**…

Not all edges in a real world graph are necessarily created equal!

Which path is actually the best/shortest?

Weighted Graphs

A **weighted graph** is a pair of:

- a graph $\boldsymbol{G = (V,E)}$
- A weight function $\boldsymbol{\omega(e)}$ that assigns a real number (called an **edge weight**) to each edge $\boldsymbol{e \in E}$

Examples of Weights:

- Latency of a network connection
- Distance between two cities
- Time between two metro stops
- Flow capacity between two points in a series of tubes

Shortest Path

Given:

- A weighted graph $G = (V,E,\omega)$
- A start vertex ***start*** in V
- An end vertex ***end*** in V

Shortest Path

Given:

- A weighted graph ***G* = (*V*,*E*,ω)**
- A start vertex ***start*** in ***V***
- An end vertex ***end*** in ***V***

Goal:

- Produce a simple path ***P*** from ***start*** to ***end***…
- …that minimizes the sum of edge weights in the ***P***

BFSOne - Adding Level

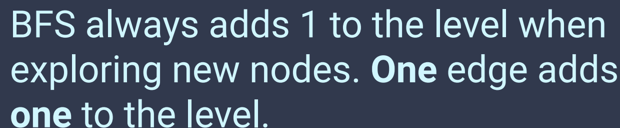
```
def BFSOne(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.Queue[Graph[...]#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for(e <- v.incident) {
      if(e.label == EdgeLabel.UNEXPLORED){
        val w =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

```
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)
        }
      }
    }
  }
}
```

BFSOne - Shortest Path

```
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)
        }
      }
    }
  }
}
```



BFSOne - Shortest Path

```
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)
        }
      }
    }
  }
}
```

Consequence: Dequeue reads vertices in ascending order of level. (FIFO)

BFSOne - Shortest Path

```
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)
        }
      }
    }
  }
}
```

Consequence: Dequeue reads vertices in ascending order of level. (FIFO)

Therefore: The first time we reach a vertex, it is via the fewest number of edges from start.

BFS and Shortest Path

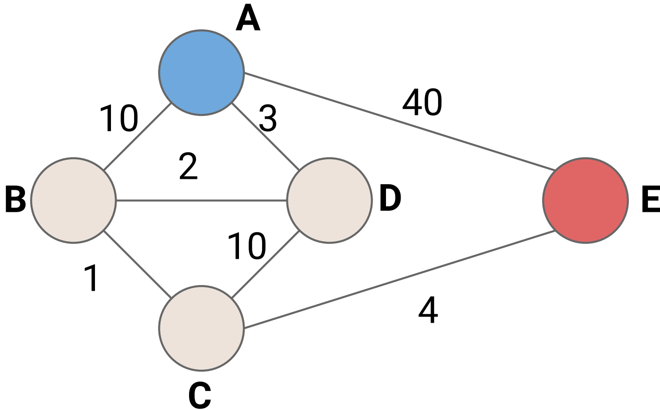
Observation: Breadth-First Search finds paths with the fewest number of edges. This is equivalent to finding the shortest path with $\boldsymbol{\omega(e) = 1}$ for all $\boldsymbol{e}$

What changes if we allow $\boldsymbol{\omega(e)}$ *to vary?*

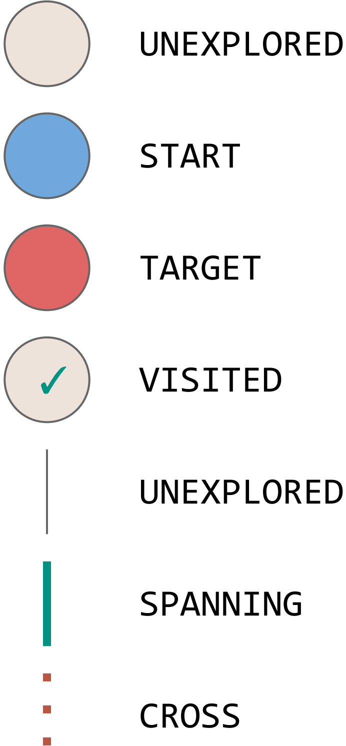
Detailed Example

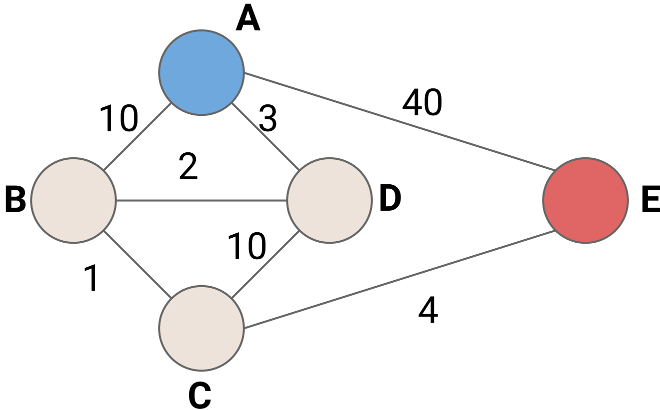
Call Stack

Work Queue



Detailed Example



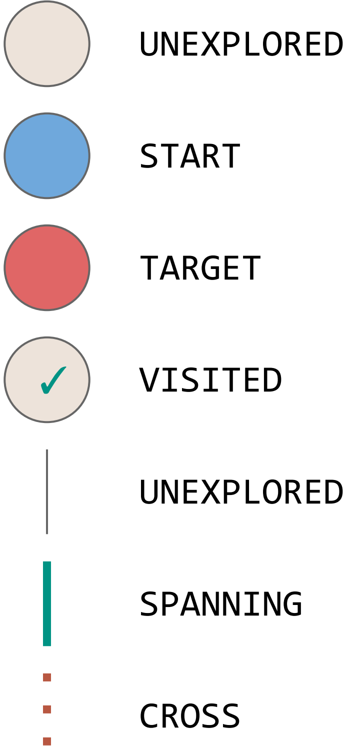


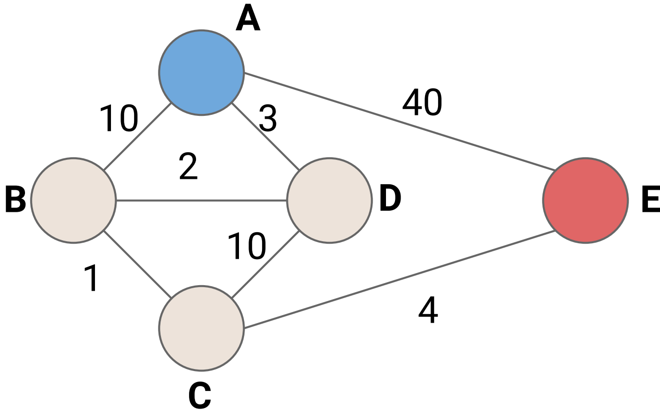
Call Stack

BFS(G)

Work Queue

Detailed Example

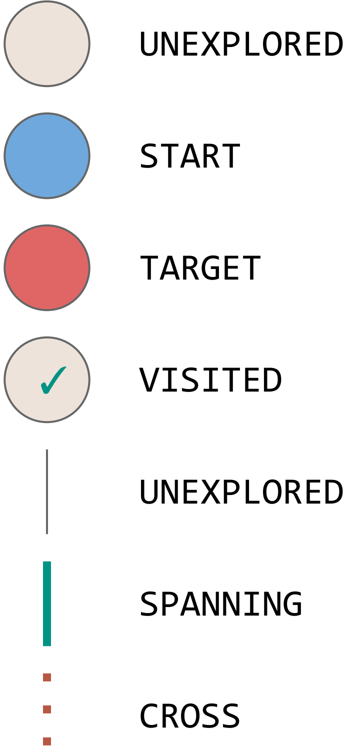


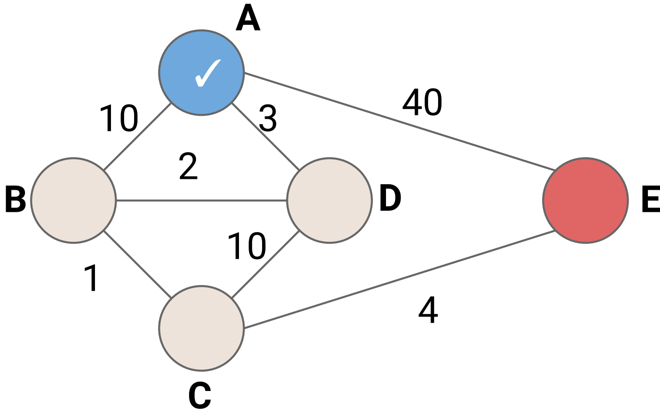


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue

Detailed Example

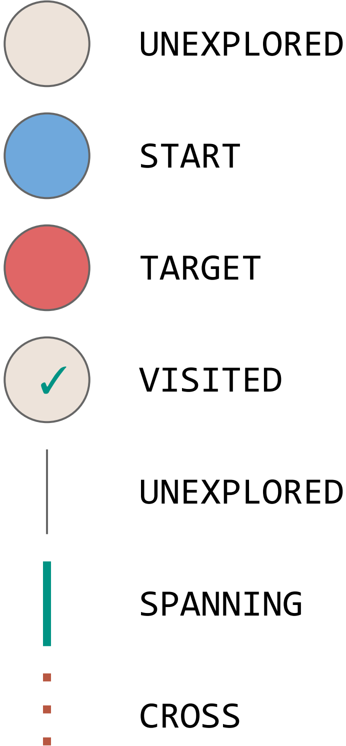




Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A

Detailed Example

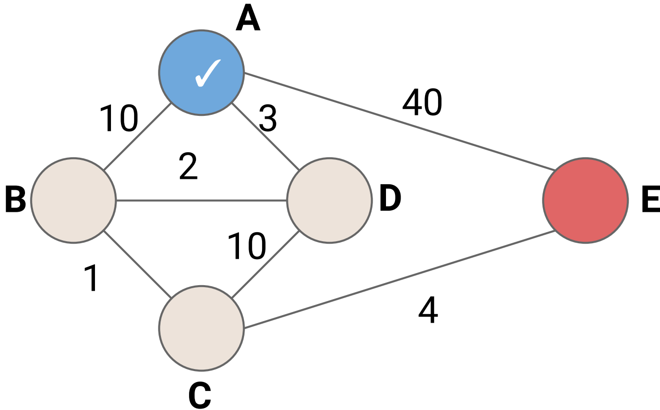


Call Stack

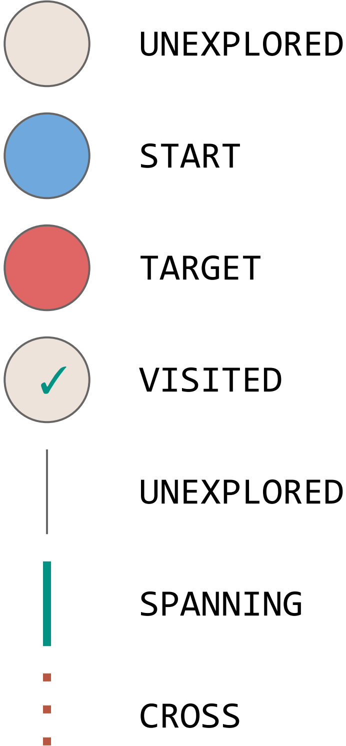
BFS(G)
BFSOne(G,A)

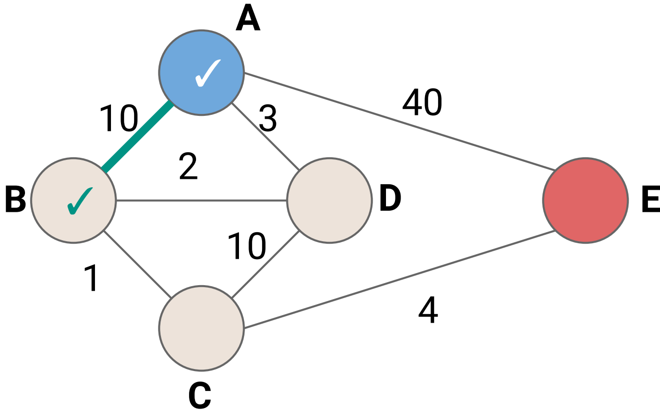
Work Queue

→ ~~A~~



Detailed Example





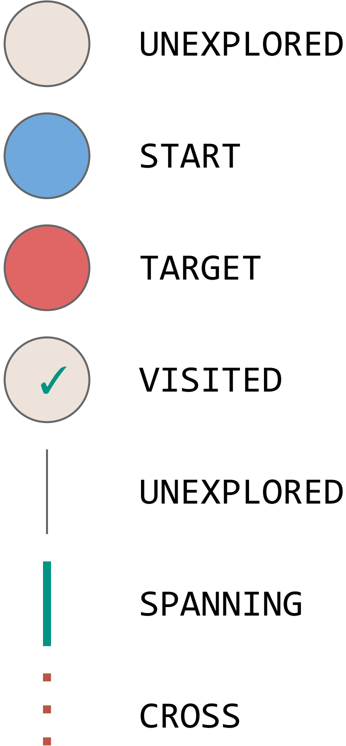
Call Stack

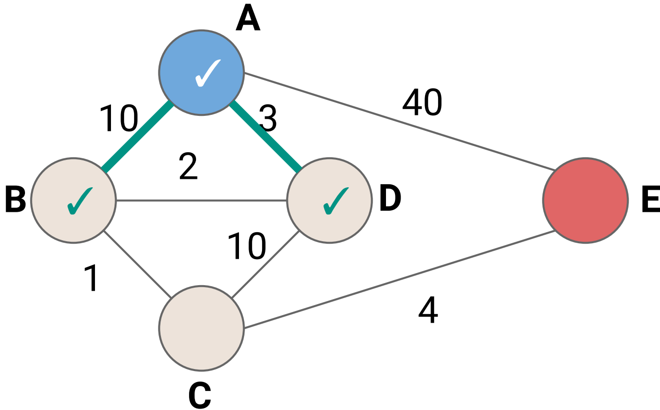
BFS(G)
BFSOne(G,A)

Work Queue

~~A~~
B

Detailed Example





<u>Call Stack</u>

BFS(G)

BFSOne(G,A)

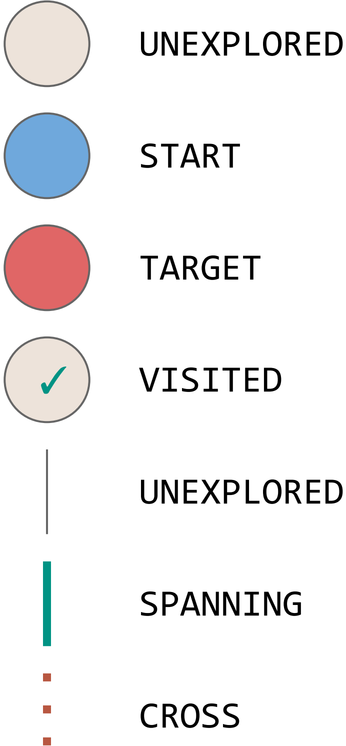
<u>Work Queue</u>

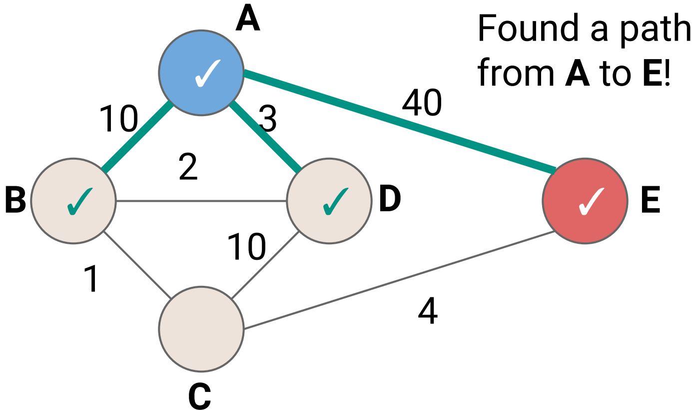
~~A~~

B

D

Detailed Example

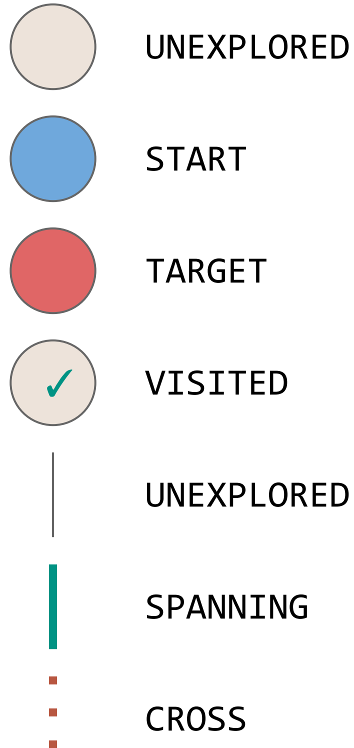


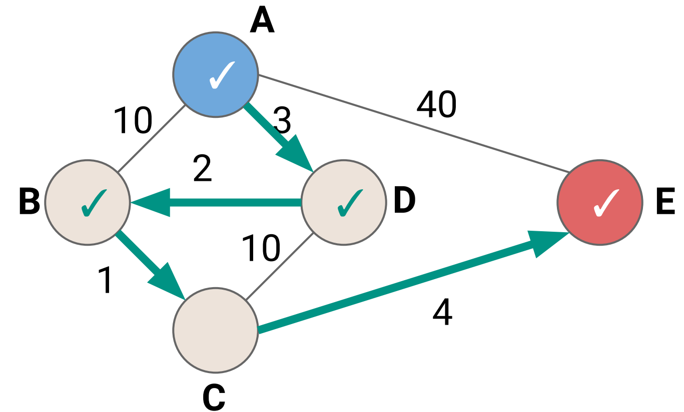


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B
D
E

Detailed Example - Desired Path





How do we find this path?

Shortest Path

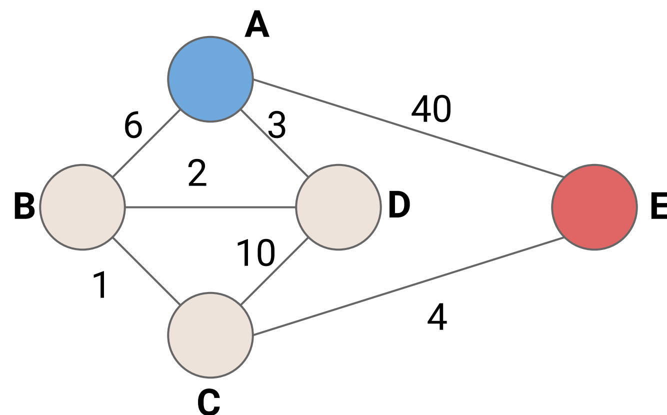
Thought Experiment: How can we find the shortest path when not all edges are created equal?

Shortest Path

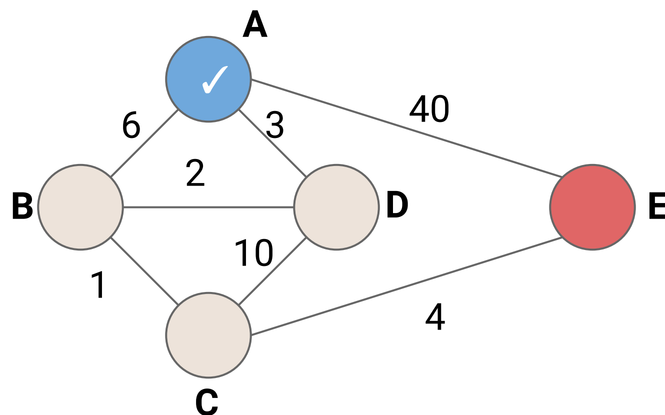
Thought Experiment: How can we find the shortest path when not all edges are created equal?

At any given point, what vertex should we explore next?

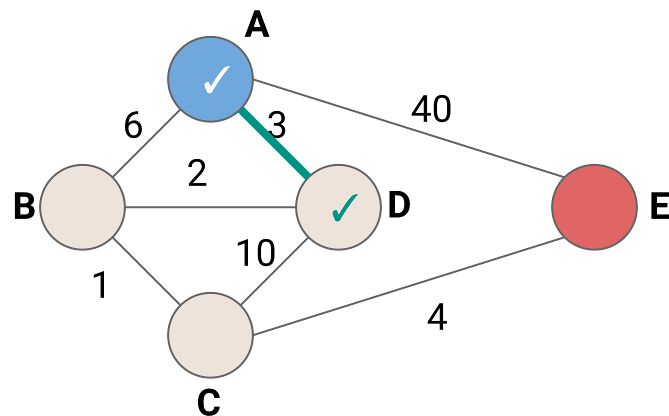
Attempt #1: Explore Smallest Edge



Attempt #1: Explore Smallest Edge

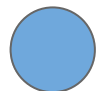


Attempt #1: Explore Smallest Edge

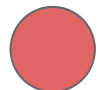


Attempt #1: Explore Smallest Edge

UNEXPLORED



START



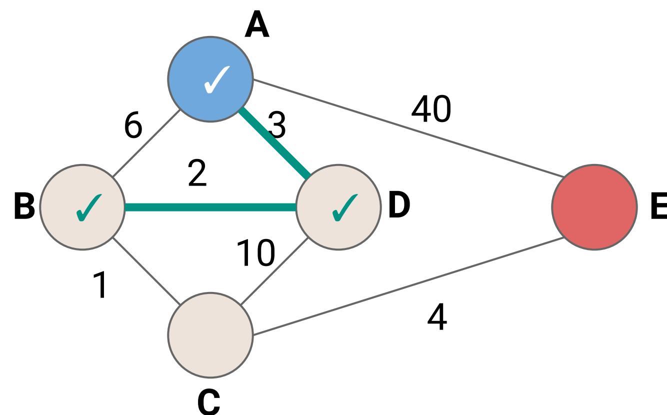
TARGET

VISITED

UNEXPLORED

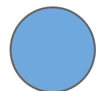
SPANNING

CROSS

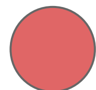


Attempt #1: Explore Smallest Edge

UNEXPLORED



START



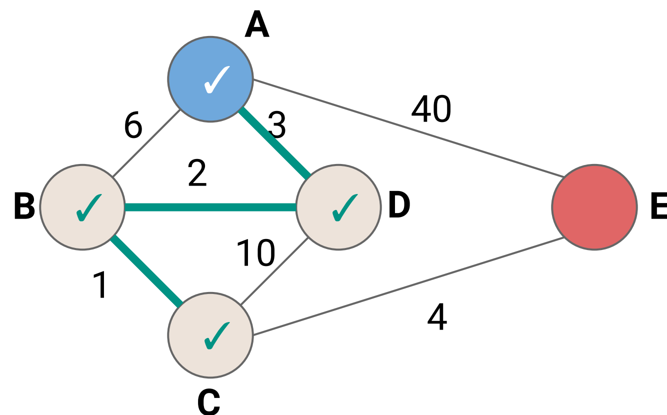
TARGET

VISITED

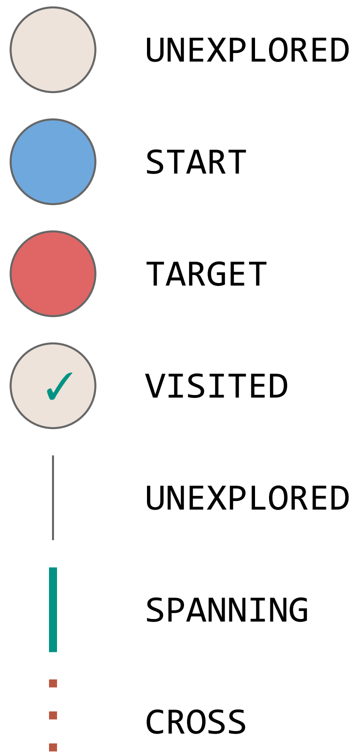
UNEXPLORED

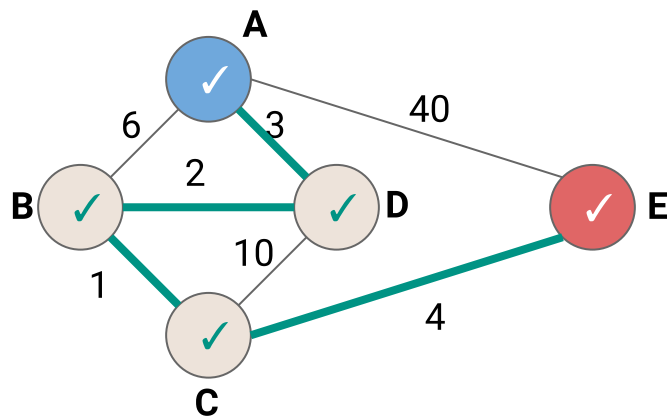
SPANNING

CROSS

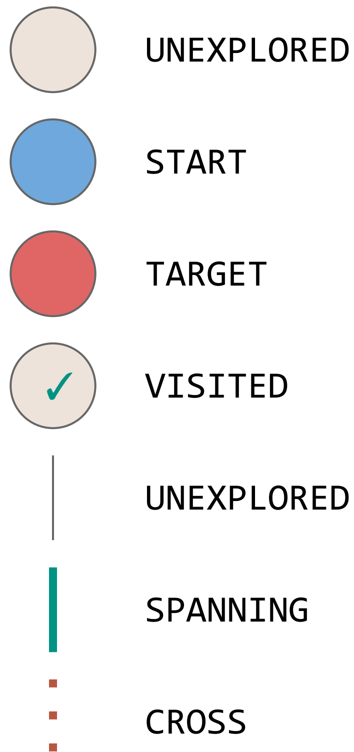


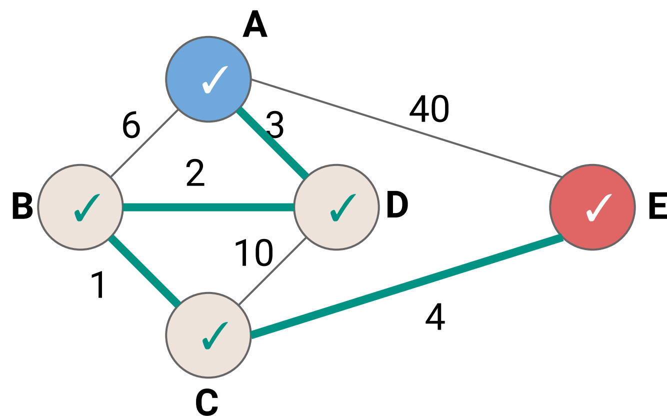
Attempt #1: Explore Smallest Edge





Attempt #1: Explore Smallest Edge

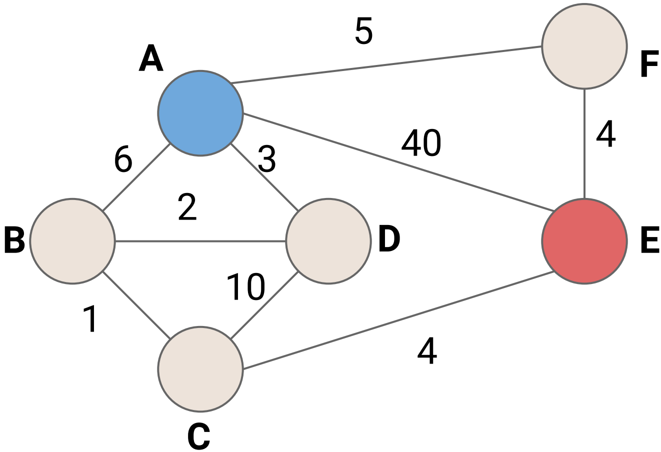




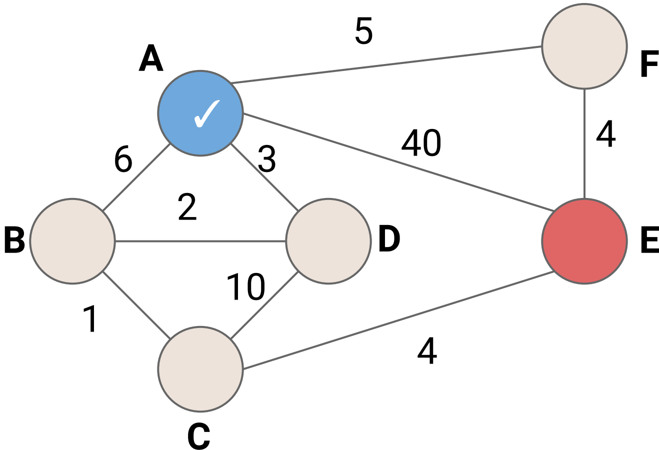
Attempt #1: Explore Smallest Edge

Will exploring the smallest available edge always work?

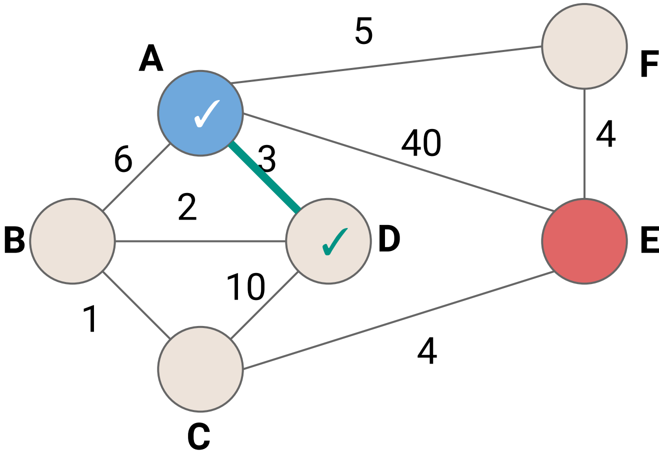
Desired Exploration Order



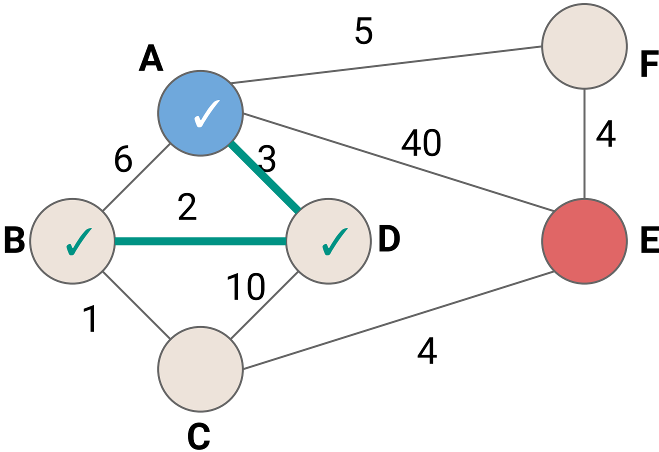
Desired Exploration Order



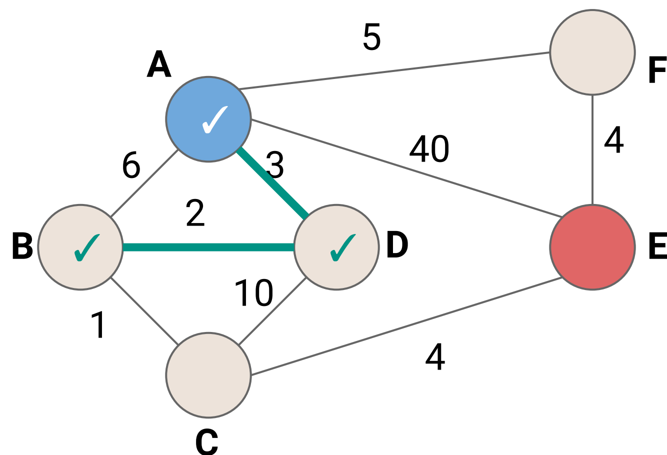
Desired Exploration Order



Desired Exploration Order



Desired Exploration Order



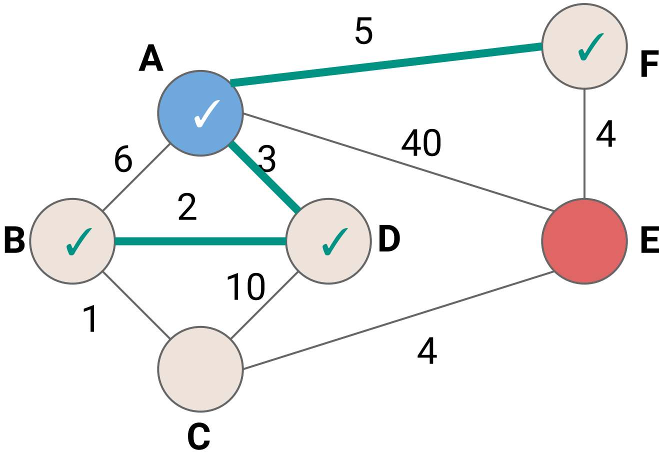
If we follow the smallest edge, we will explore **C** next.
But how far is **C** from **A**?
Are there any unexplored vertices that are closer to **A**?

Desired Exploration Order - Closest Vertex

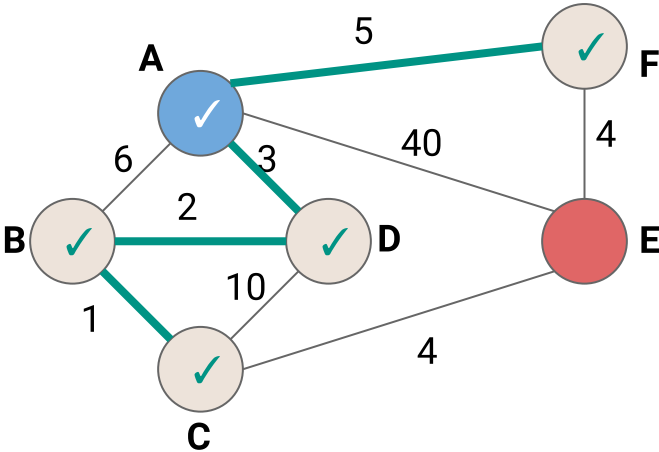
F is only 5 away...it is the closest unexplored vertex to **A**

C is 6 away from **A**

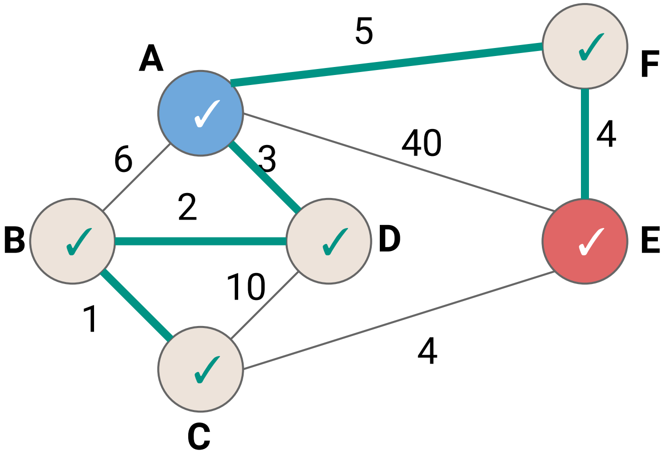
Desired Exploration Order



Desired Exploration Order



Desired Exploration Order



Path Found!

BFSOne - Shortest Path

```
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 + ω(e)))
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}
```

We want to be able to dequeue vertices in ascending order of distance…but how?

A New ADT...`PriorityQueue`

`PriorityQueue[A:Ordering]`

`enqueue(v: A)`

Insert value ***v*** into the priority queue

`head: A`

Peek at the highest priority value in the priority queue

`dequeue: A`

Remove the highest priority value in the priority queue

A New ADT...`PriorityQueue`

`PriorityQueue[A:Ordering]`

`enqueue(v: A)`

Insert value ***v*** into the priority queue

`head: A`

Peek at the **highest priority** value in the priority queue

`dequeue: A`

Remove the **highest priority** value in the priority queue