CSE 250
Data Structures

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

Lec 21: Shortest Path

Announcements

- PA2 testing due on Sunday @ 11:59PM
  - No late submissions or grace days accepted

Depth-First Search Complexity

In summary…

1. Mark the vertices `UNVISITED`	$O(\|V\|)$
2. Mark the edges `UNVISITED`	$O(\|E\|)$
3. All calls to `DFSOne`	$O(\|E\|)$ **total**
	$O(\|V\| + \|E\|)$

```
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 == U
11           curr.setLabel(
12           e.setLabel(SPANNING);
13           todo.push(w);
14         } else {
15           e.setLabel(BACK);
16         }
17 }}}}
```

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 }}}}
```

```
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 }}}}
```

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 }}}}
```

Dequeue a vertex from the Queue and check all of it's 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 }}}}
```

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 enqueud 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 }}}}
```

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

Detailed Example

UNEXPLORED

VISITED

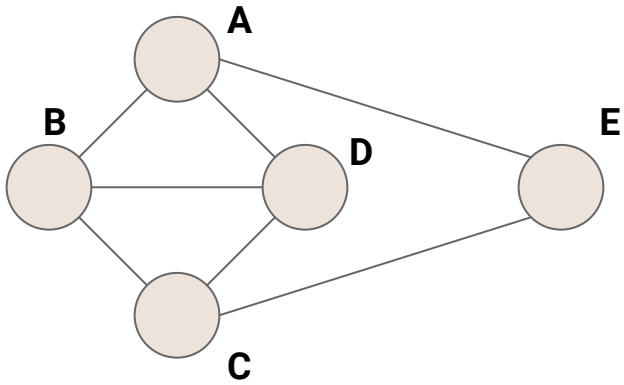
UNEXPLORED

SPANNING

CROSS

Call Stack

Work Queue



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

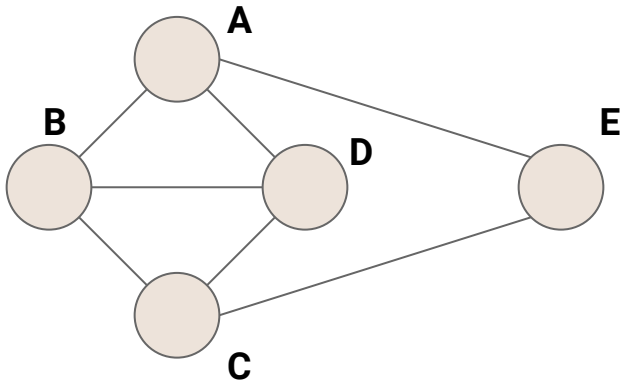
SPANNING

CROSS

Call Stack

BFS(G)

Work Queue



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

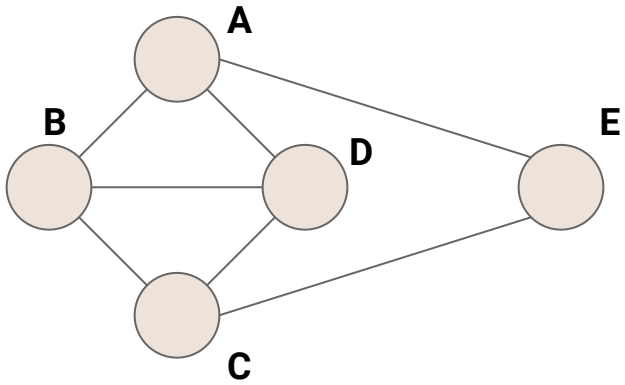
SPANNING

CROSS

Call Stack

BFS(G)
BFSOne(G,A)

Work Queue



Detailed Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

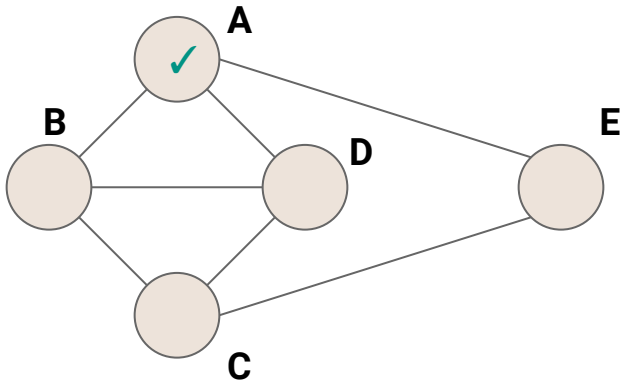
CROSS

Call Stack

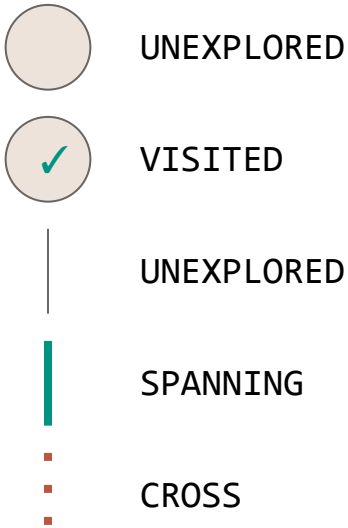
BFS(G)
BFSOne(G,A)

Work Queue

A

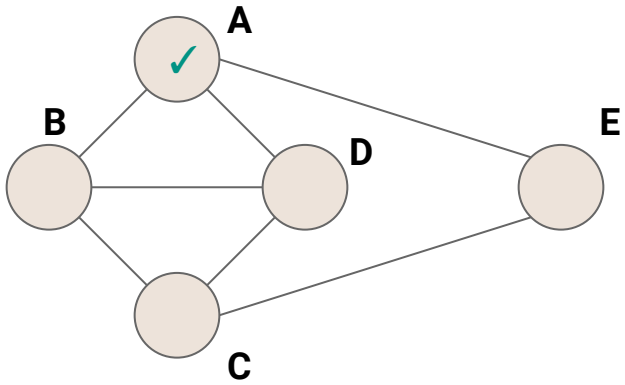


Detailed Example

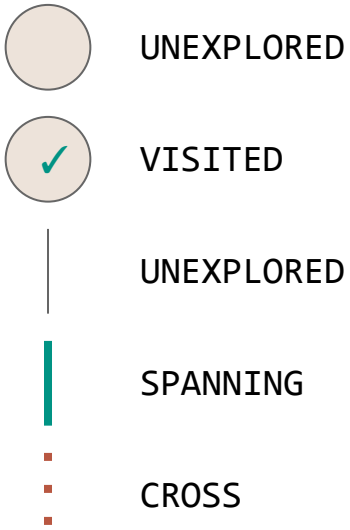


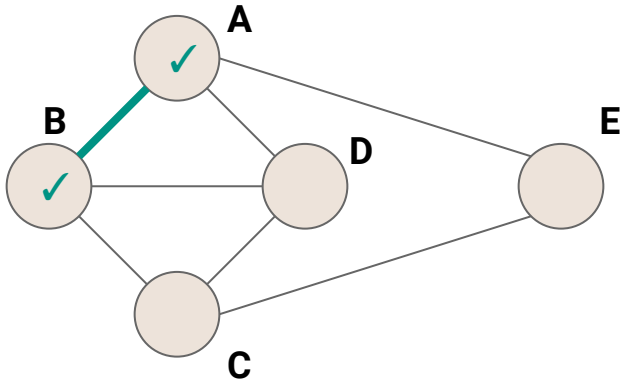
Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~



Detailed Example

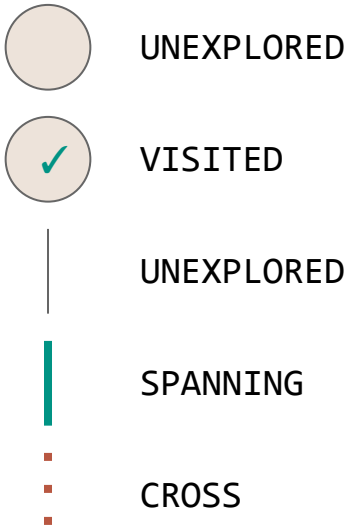


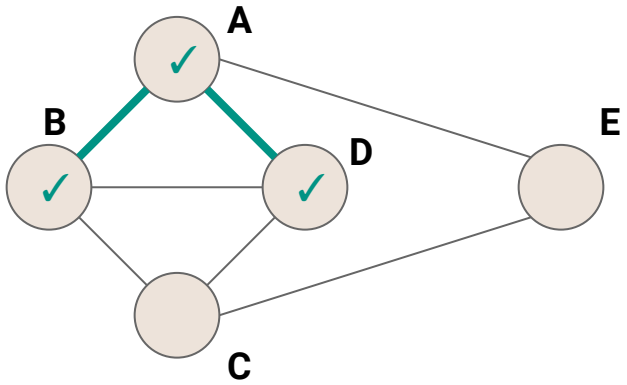


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B

Detailed Example

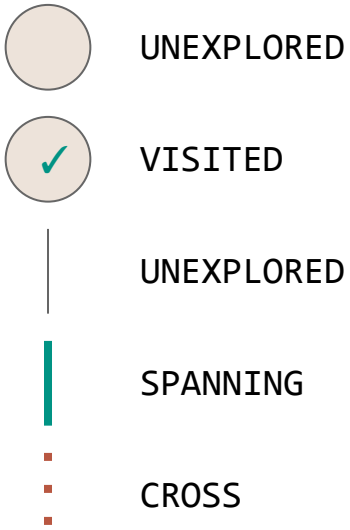


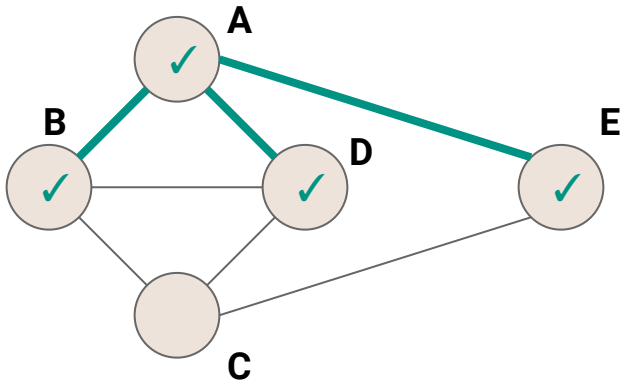


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B
D

Detailed Example

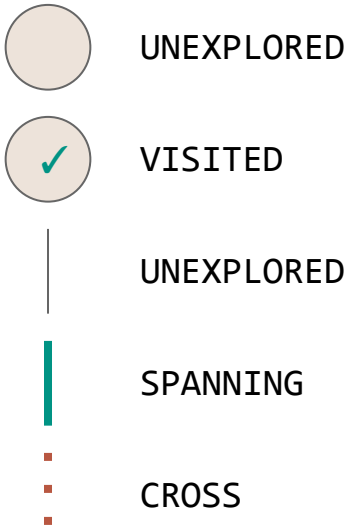


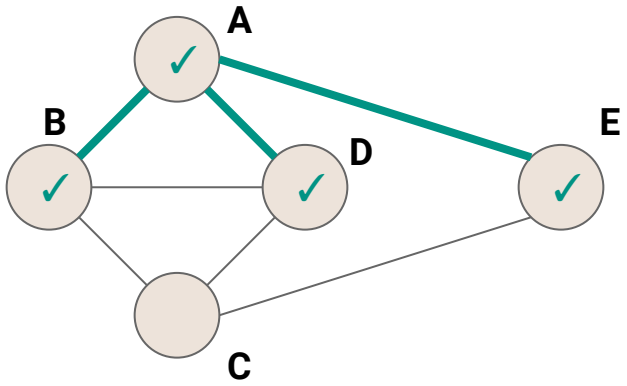


Call Stack
BFS(G)
BFSOne(G,A)

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

Detailed Example

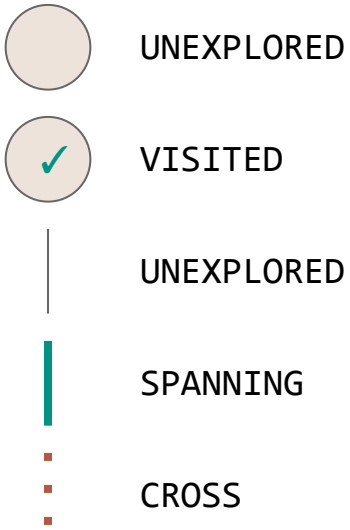


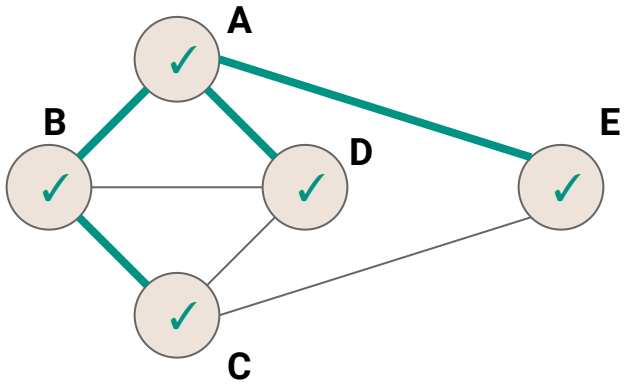


Call Stack
BFS(G)
BFSOne(G,A)

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

Detailed Example





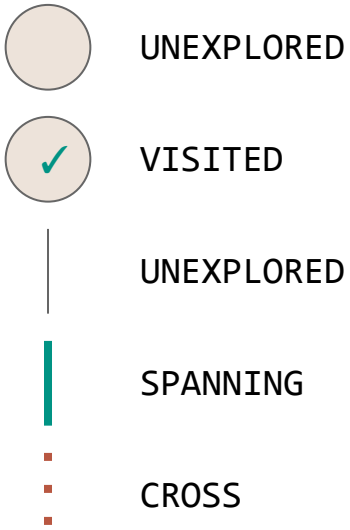
Call Stack

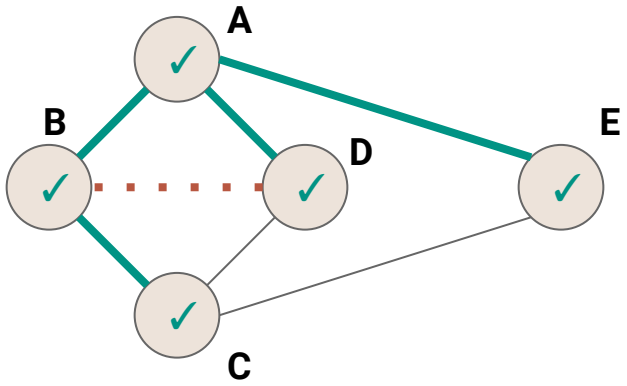
BFS(G)
BFSOne(G,A)

Work Queue

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

Detailed Example

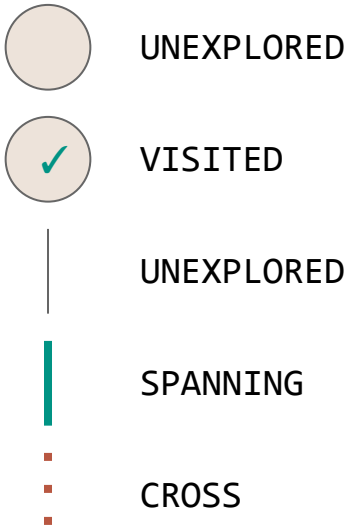


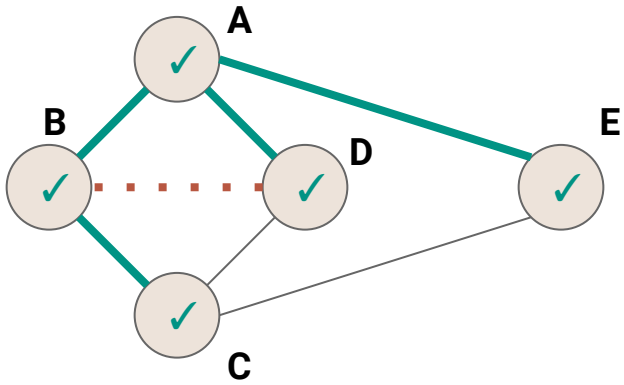


Call Stack
BFS(G)
BFSOne(G,A)

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

Detailed Example

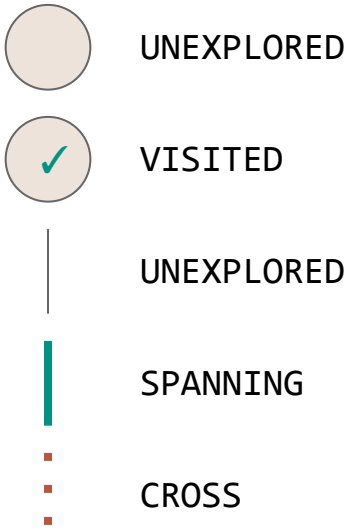




Call Stack
BFS(G)
BFSOne(G,A)

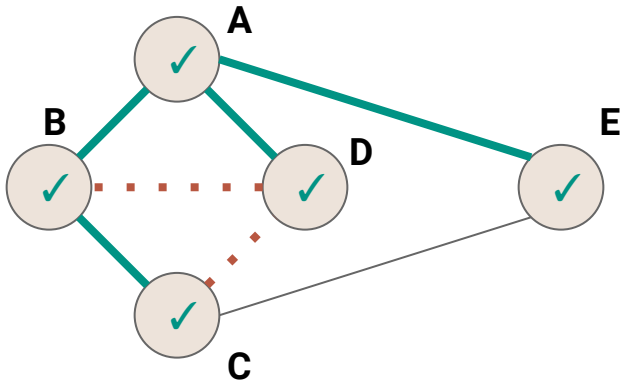
Work Queue
~~**A**~~
~~**B**~~
~~**D**~~
E
C

Detailed Example

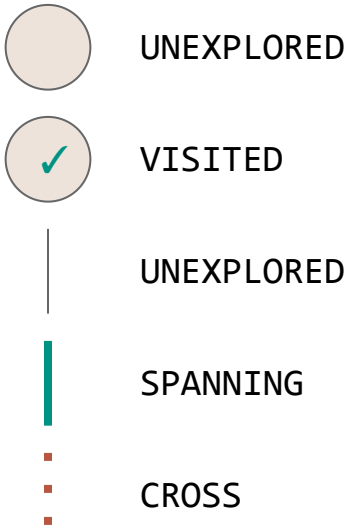


Call Stack
BFS(G)
BFSOne(G,A)

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

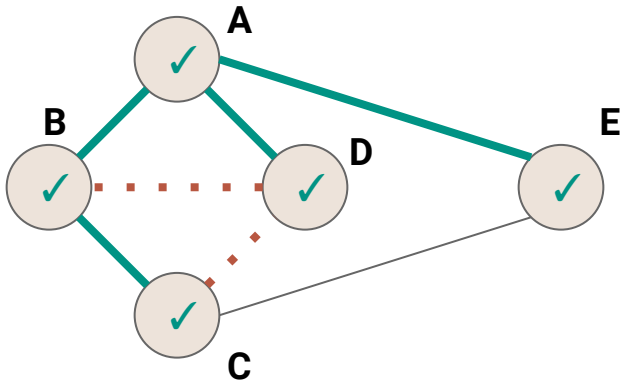


Detailed Example

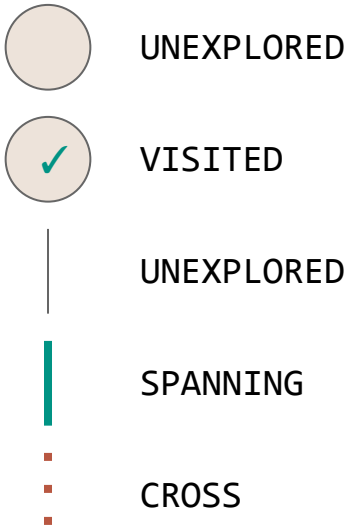


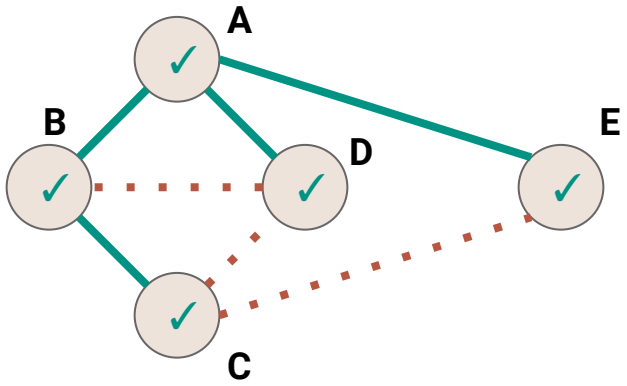
Call Stack
BFS(G)
BFSOne(G,A)

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



Detailed Example

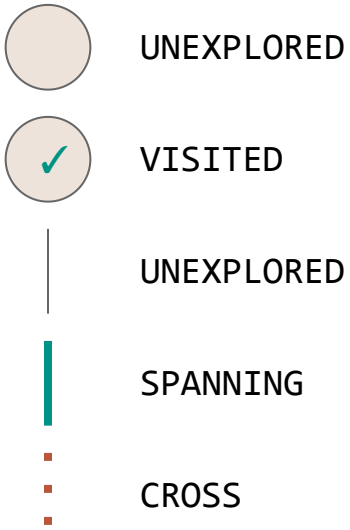


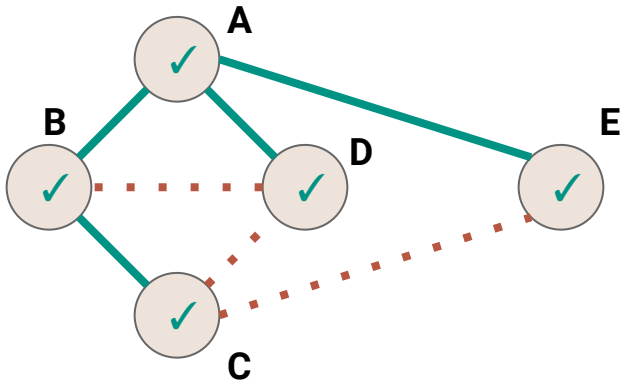


Call Stack
BFS(G)
BFSOne(G,A)

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    Θ(|V|)
3    Θ(|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    Θ(|V|)
3    Θ(|E|)
4    for (Vertex v : graph.vertices) {
5      if (v.label == UNEXPLORED) {
6        Θ(???)
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 }}}}
```

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Θ(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 }}}}
```

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

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

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

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

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

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

How many iterations will this while loop run?

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

Breadth-First Search Complexity

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

Breadth-First Search Complexity

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

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

Breadth-First Search Complexity

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

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

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

Breadth-First Search Complexity

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

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

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

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

Breadth-First Search Complexity

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

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

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

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

$$= O(|E|)$$

Breadth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`**

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary…

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

Breadth-First Search Complexity

In summary…

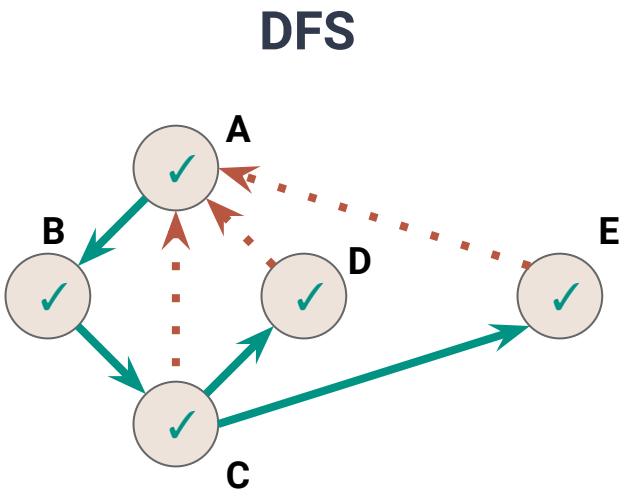
1.	Mark the vertices `UNVISITED`	$O(\|V\|)$
2.	Mark the edges `UNVISITED`	$O(\|E\|)$
3.	Add each vertex to the work queue	$O(\|V\|)$
4.	Process each vertex	$O(\|E\|)$ **total**

Breadth-First Search Complexity

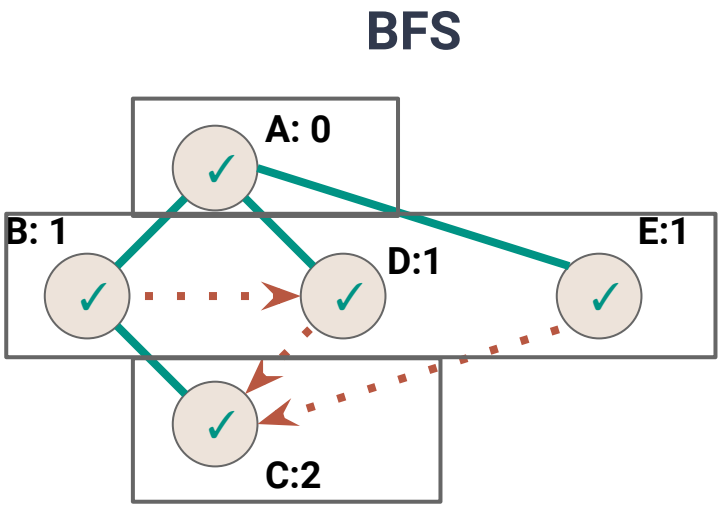
In summary…

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

DFS vs 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***

DFS Traversal vs BFS Traversal

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

** sort of…*

Getting the Actual Paths

So far we can label the whole graph…but what if we want the actual paths?

Option #1 - Store the Paths as We Go

What if we store the paths in addition to the Vertices?

```
1  public class TodoEntry {
2    public Vertex vertex;
3    public List<Edge> path;
4  }
```

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v, new LinkedList<Edge>()));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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           List path = curr.path.clone();
14           path.add(e);
15           todo.enqueue(new TodoEntry(w, path));
16         } else {
17           e.setLabel(BACK);
18 }}}}}
```

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v, new LinkedList<Edge>()));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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           List path = curr.path.clone();
14           path.add(e);
15           todo.enqueue(new TodoEntry(w, path));
16         } else {
17           e.setLabel(BACK);
18 }}}}}
```

Begin our search at the starting vertex with an empty path

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v, new LinkedList<Edge>()));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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           List path = curr.path.clone();
14           path.add(e);
15           todo.enqueue(new TodoEntry(w, path));
16         } else {
17           e.setLabel(BACK);
18 }}}}}
```

When we find an unexplored node, copy our current path and add the edge we took to get to the new node

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v, new LinkedList<Edge>()));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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           List path = curr.path.clone();
14           path.add(e);
15           todo.enqueue(new TodoEntry(w, path));
16         } else {
17           e.setLabel(BACK);
18 }}}}}
```

What's the problem with this solution?

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v, new LinkedList<Edge>()));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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           List path = curr.path.clone();
14           path.add(e);
15           todo.enqueue(new TodoEntry(w, path));
16         } else {
17           e.setLabel(BACK);
18 }}}}}
```

We have to copy the path every time!
This can get expensive

What's the problem with this solution?

Option #2 - Create an "Edge To" Map

Let's store information about which edge we took to get to each vertex

If we know how we got to each vertex, we can rebuild paths in reverse

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

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

Create a map to store the necessary info

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

When we find an edge that goes to vertex w, put that information into our map

Rebuilding the Paths

Now that we have our edgeTo map, we can rebuild a path from our starting vertex to any other vertex

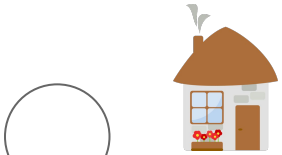
1. Set our **`current`** vertex to the vertex we want to find a path to
2. While the **`current`** vertex is not our starting vertex:
    a. Use our **`edgeTo`** map to lookup the edge that led to the **`current`** vertex
    b. Prepend that edge to our path
    c. Set **`current`** to the edges origin vertex

Quick Demo!

Now we can use BFS to find the "Shortest Paths" from any starting vertex…or can we???

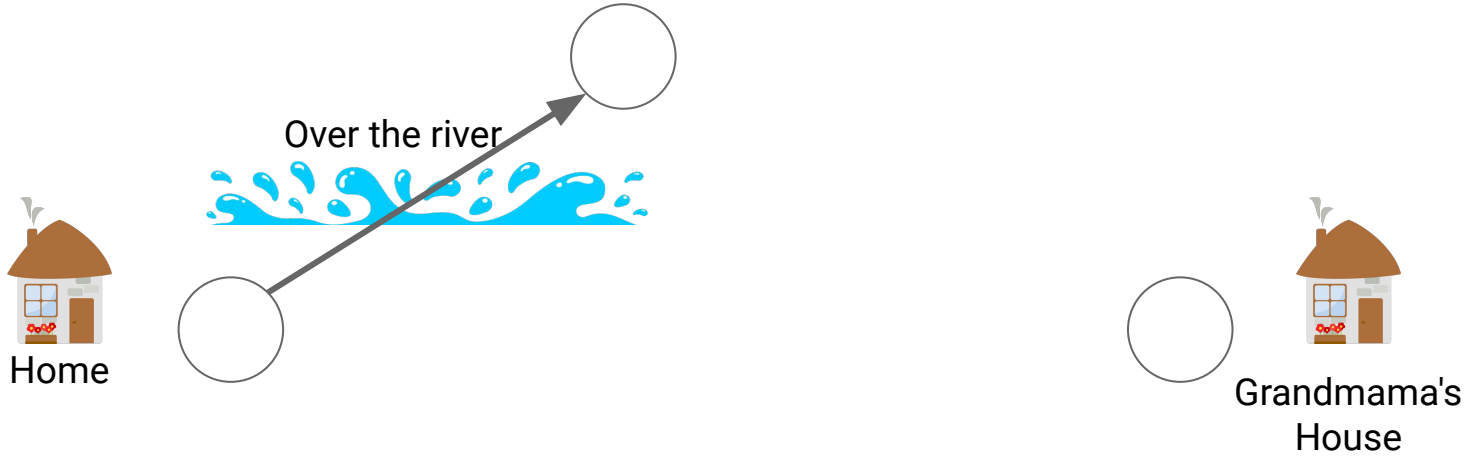
Shortest Paths

Home

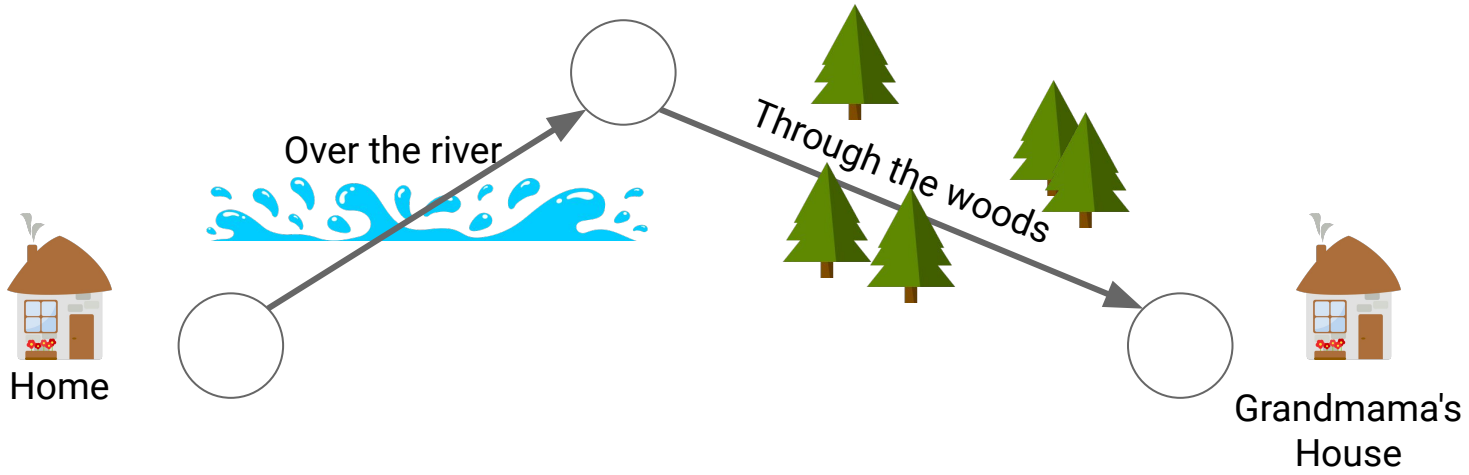


Grandmama's House

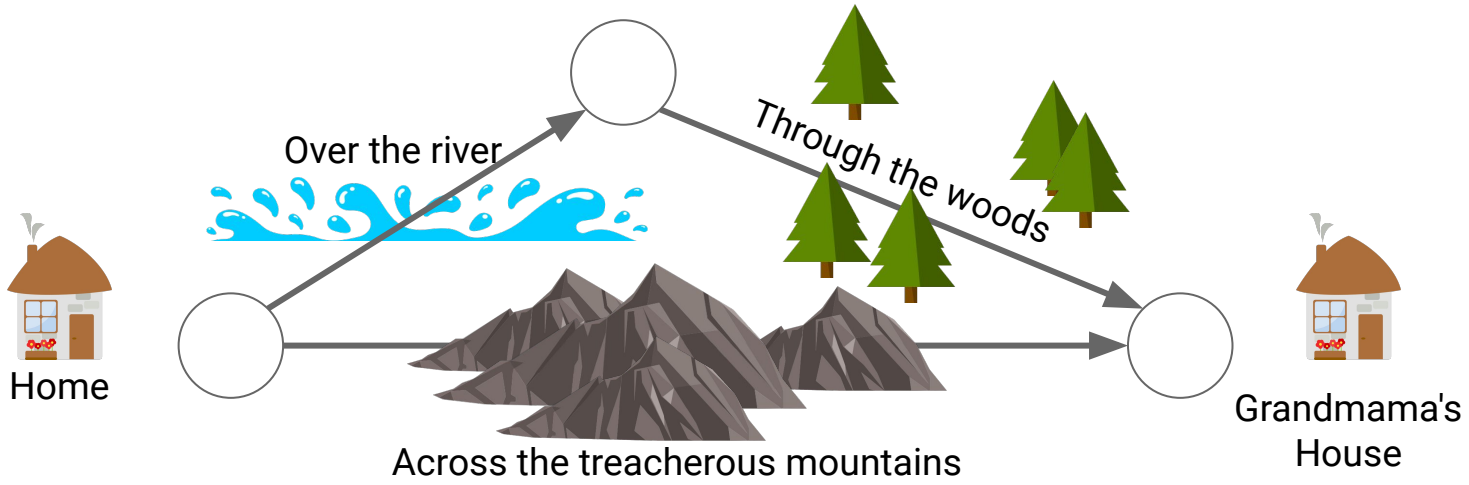
Shortest Paths



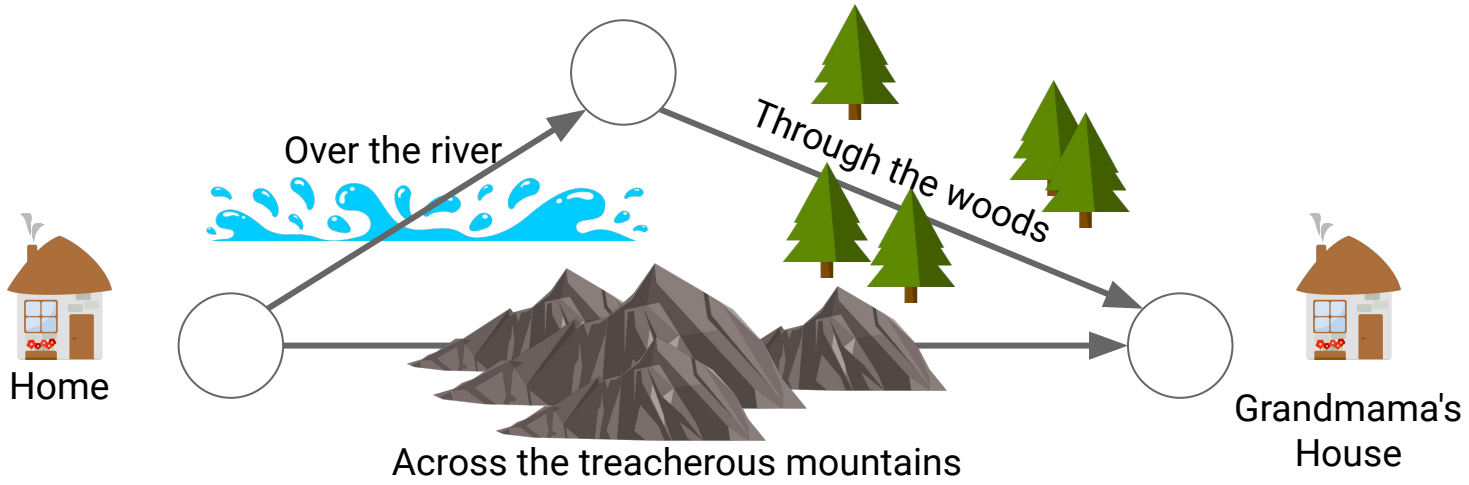
Shortest Paths



Shortest Paths

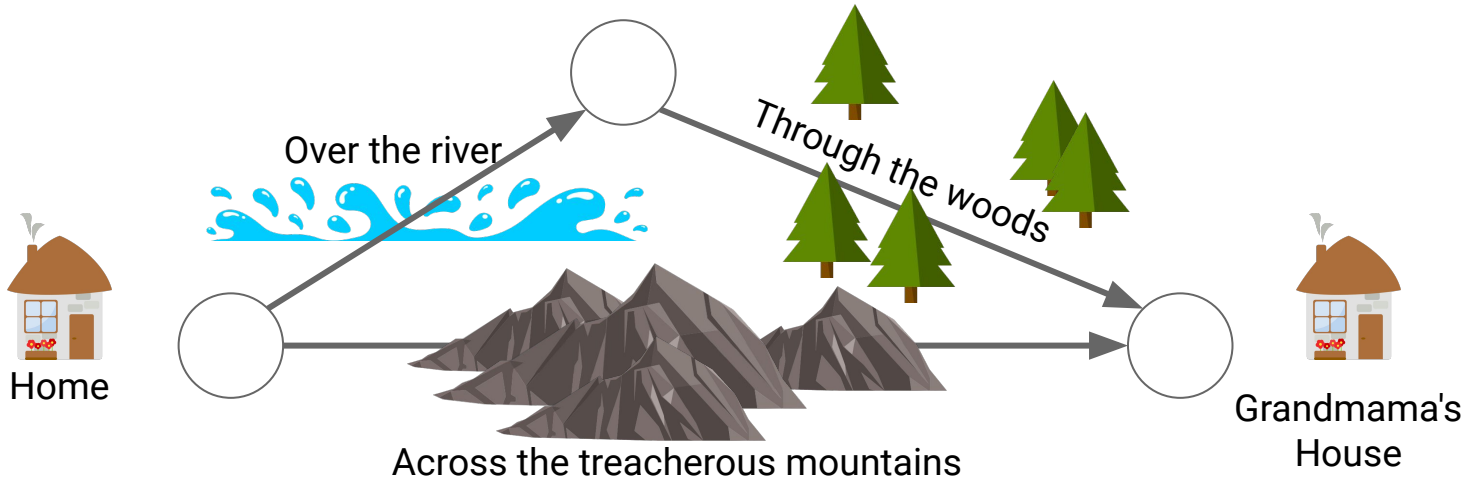


Shortest Paths



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

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 $G = (V,E,\omega)$
- 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

What if we track the "level" of each vertex in BFS?

(level in this case means the number of edges from our start vertex)

```
1 public class TodoEntry {
2   public Vertex vertex;
3   public Integer level;
4 }
```

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.level + 1));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.level + 1));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

Our starting vertex is at level 0 (0 edges from itself)

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.level + 1));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

Our starting vertex is at level 0 (0 edges from itself)

When we discover a new vertex, we add 1 to the level (because we just went one more edge)

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.level + 1));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

Because Queue is FIFO, we will dequeue in ascending order of level

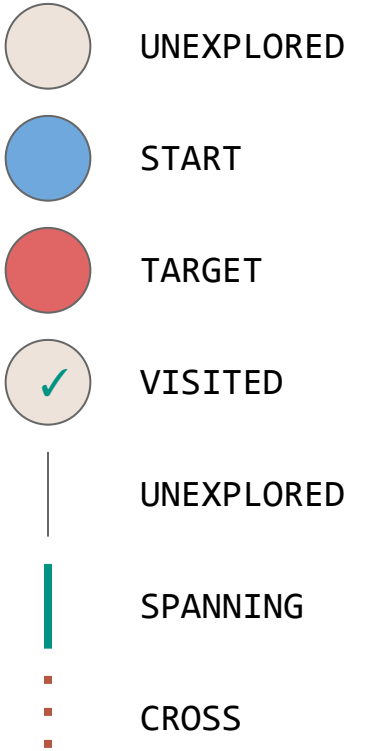
Therefore, when we discover a new vertex, it is by the shortest number of edges

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 $\omega(e) = 1$ for all e

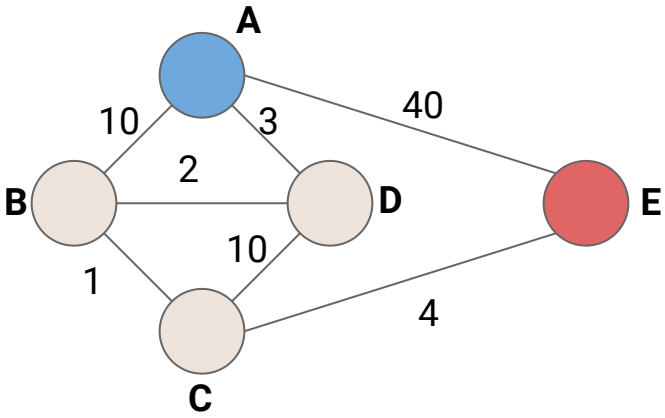
What changes if we allow $\omega(e)$ to vary?

Detailed Example

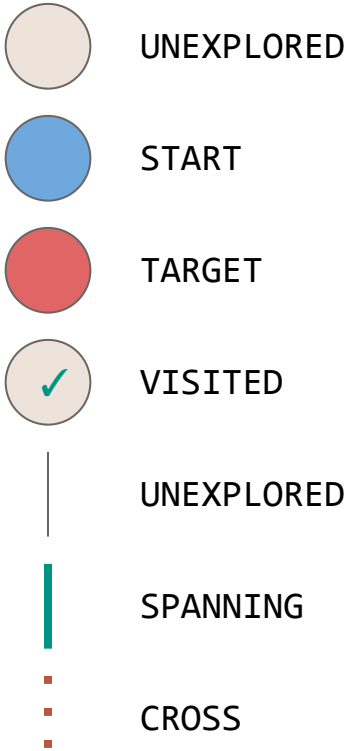


Call Stack

Work Queue



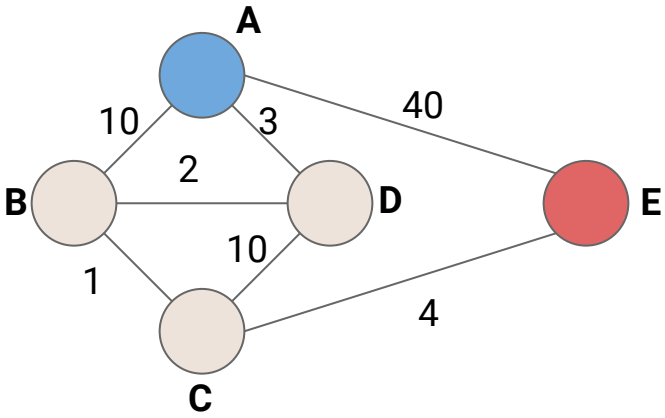
Detailed Example



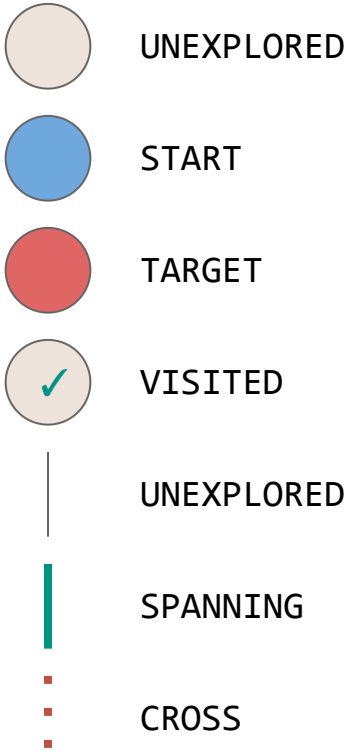
Call Stack

BFS(G)

Work Queue

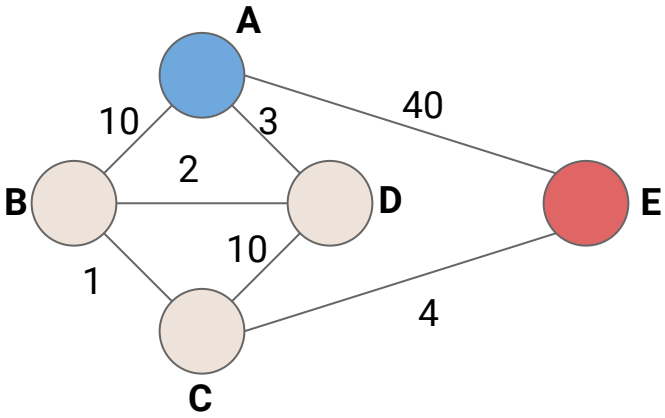


Detailed Example

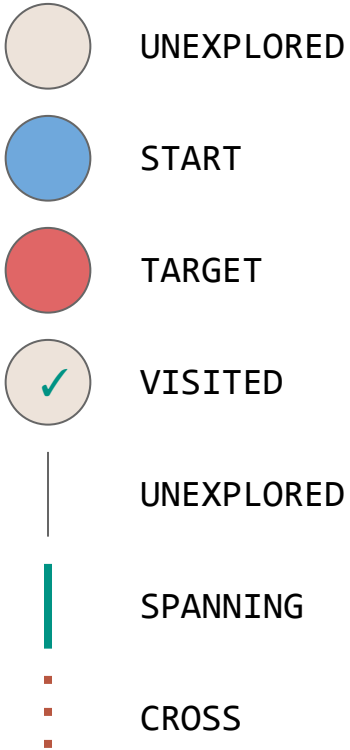


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue

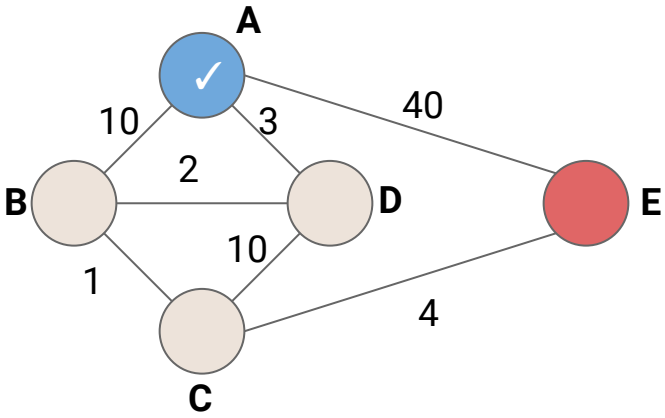


Detailed Example

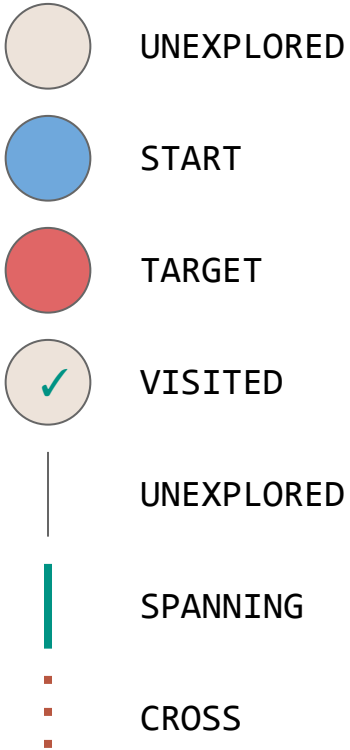


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
A

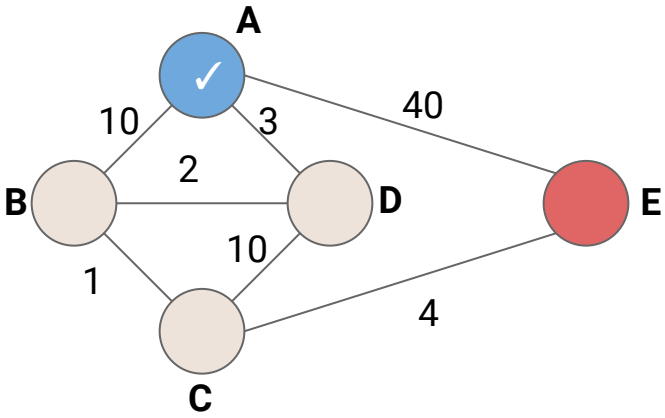


Detailed Example

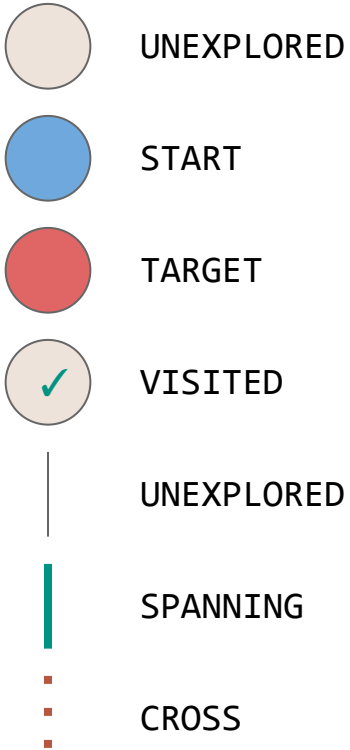


Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~

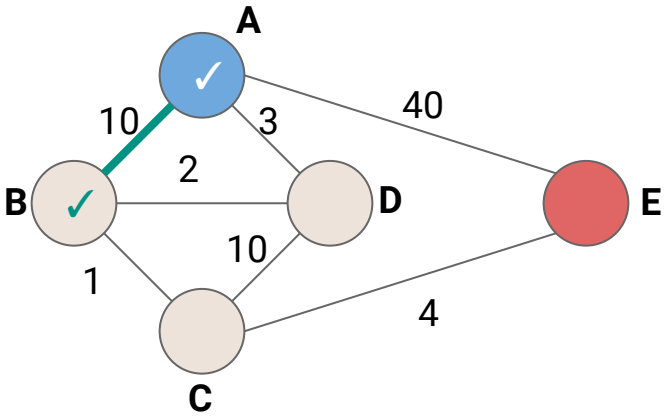


Detailed Example

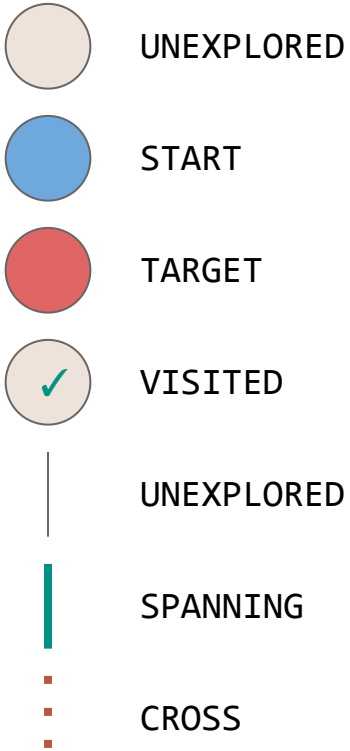


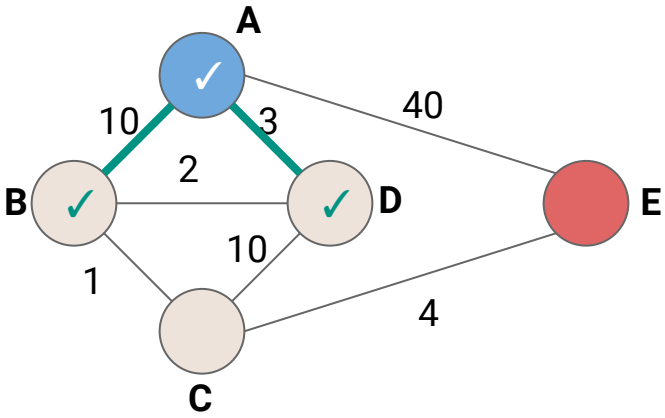
Call Stack
BFS(G)
BFSOne(G,A)

Work Queue
~~A~~
B



Detailed Example

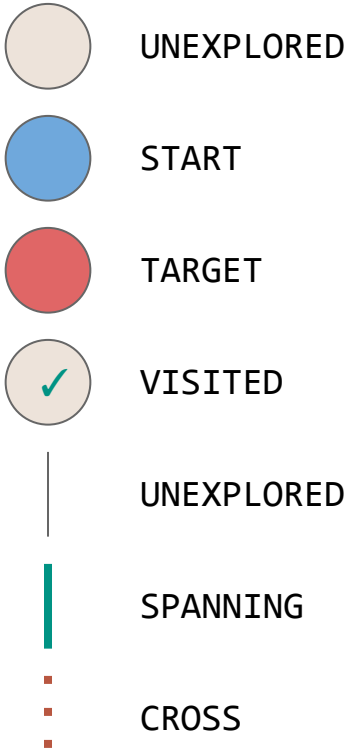




Call Stack
BFS(G)
BFSOne(G,A)

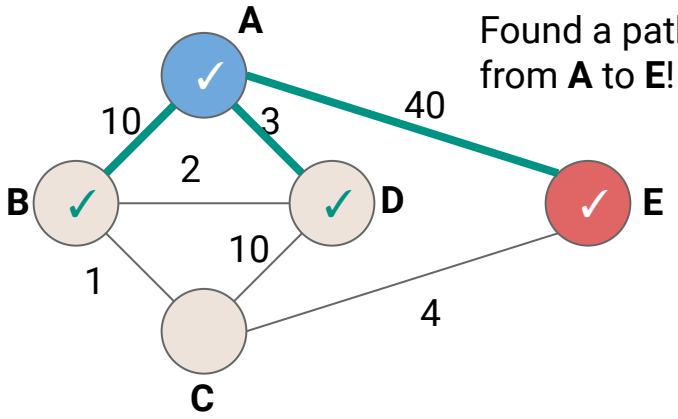
Work Queue
~~**A**~~
B
D

Detailed Example



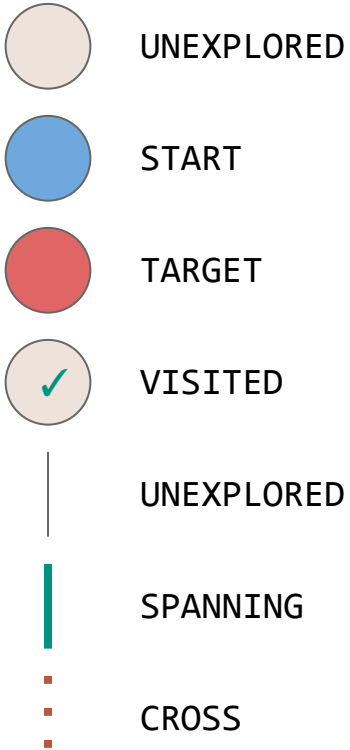
Call Stack
BFS(G)
BFSOne(G,A)

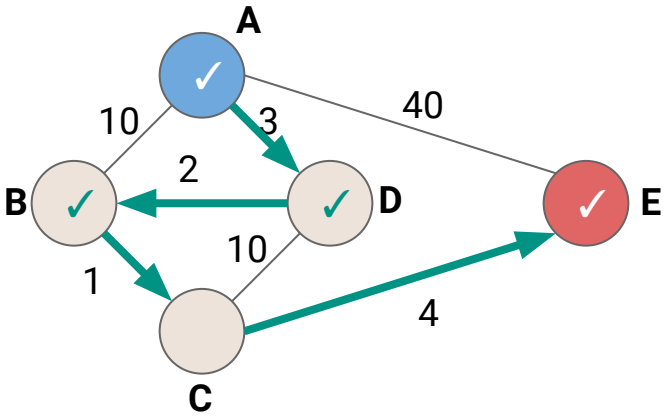
Work Queue
~~**A**~~
B
D
E



Found a path from **A** to **E**!

Detailed Example - Desired Path





How do we find this path?

Shortest Path

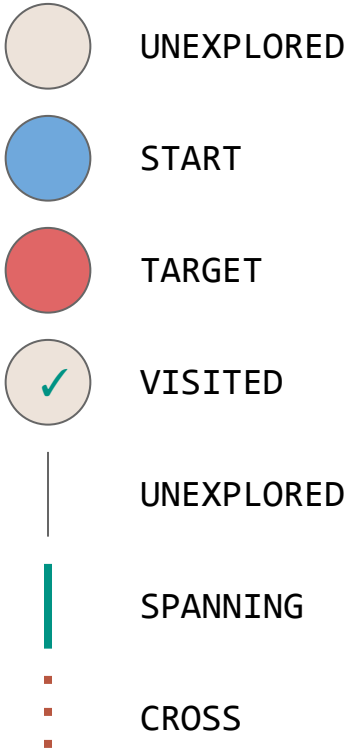
Thought Experiment: How can we find the shortest path when not all edges are created equal?

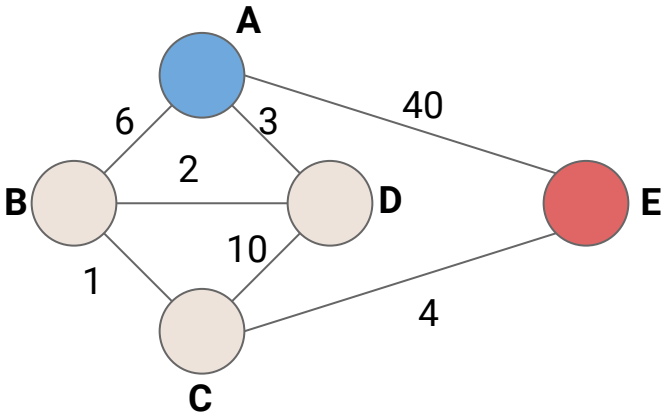
Shortest Path

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

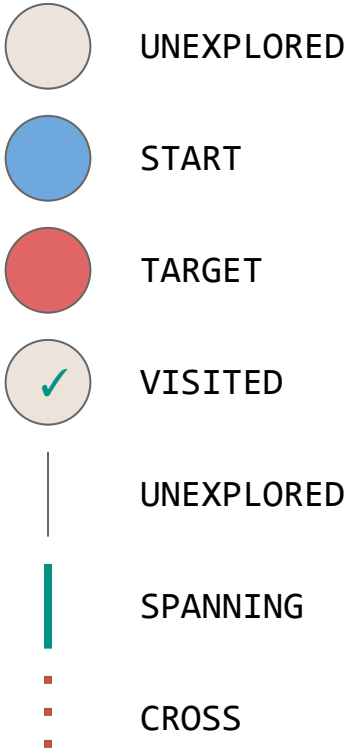
At any given point, what vertex should we explore next?

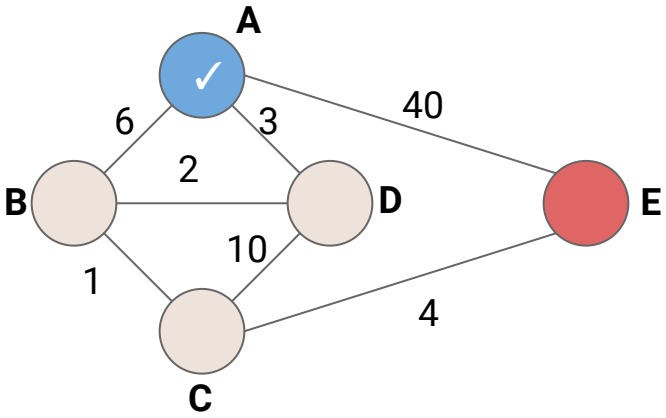
Attempt #1: Explore Smallest Edge



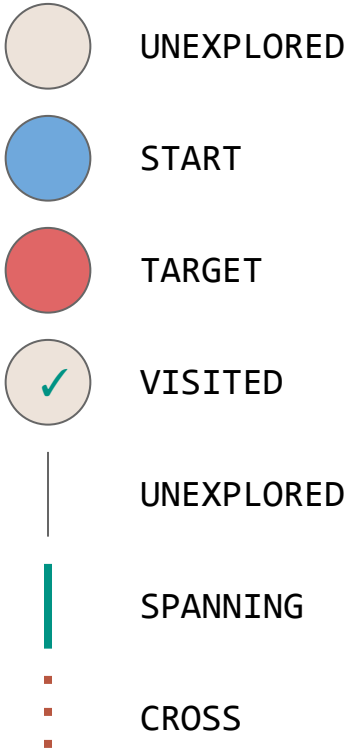


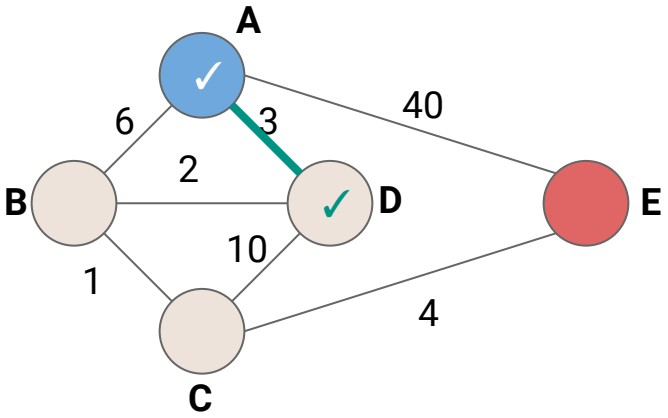
Attempt #1: Explore Smallest Edge



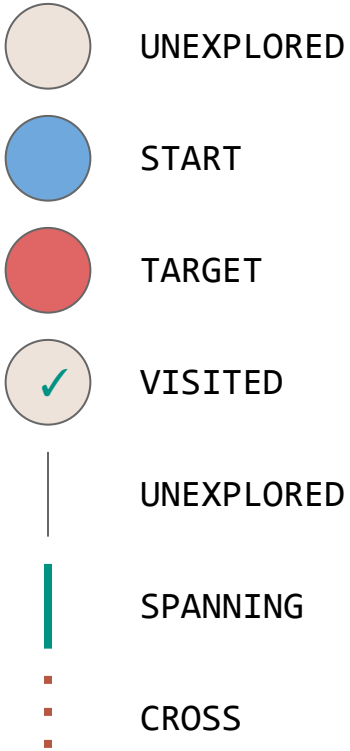


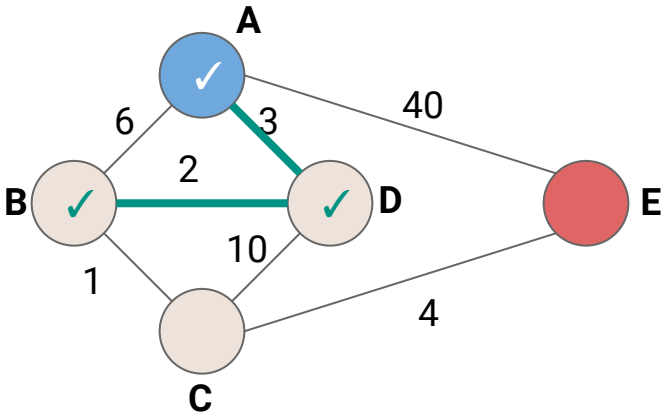
Attempt #1: Explore Smallest Edge



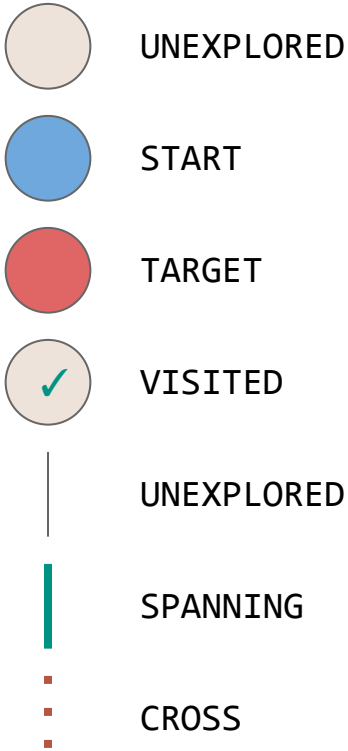


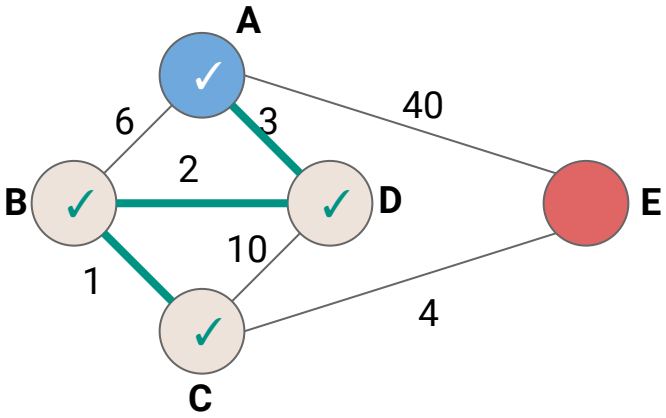
Attempt #1: Explore Smallest Edge



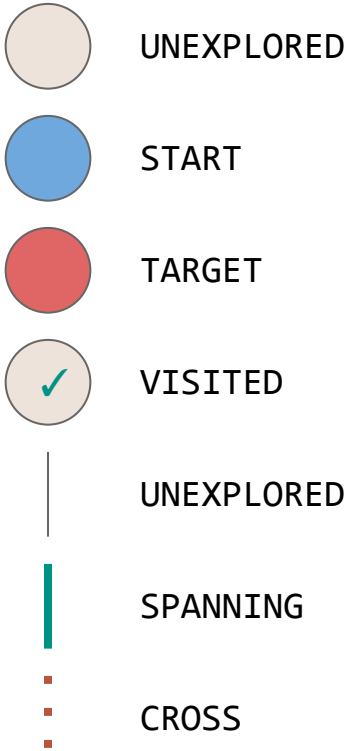


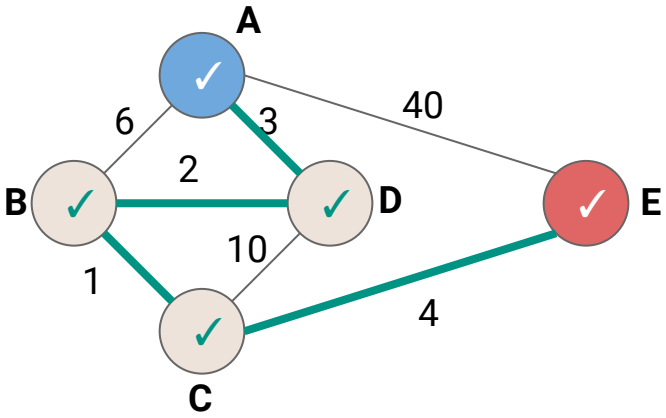
Attempt #1: Explore Smallest Edge



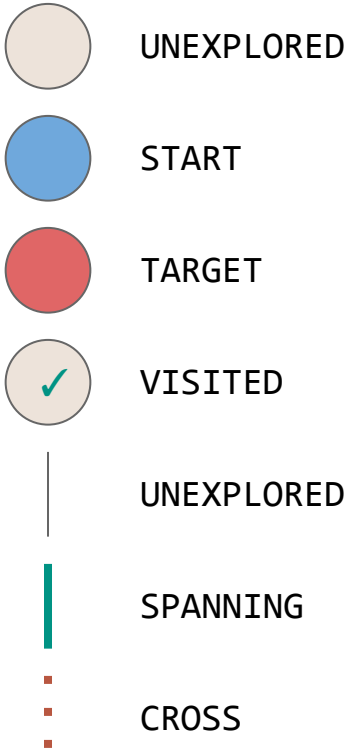


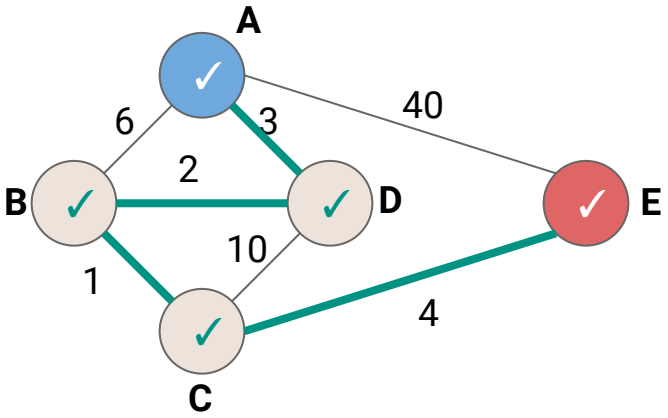
Attempt #1: Explore Smallest Edge





Attempt #1: Explore Smallest Edge





Attempt #1: Explore Smallest Edge

Will exploring the smallest available edge always work?

Desired Exploration Order

UNEXPLORED

START

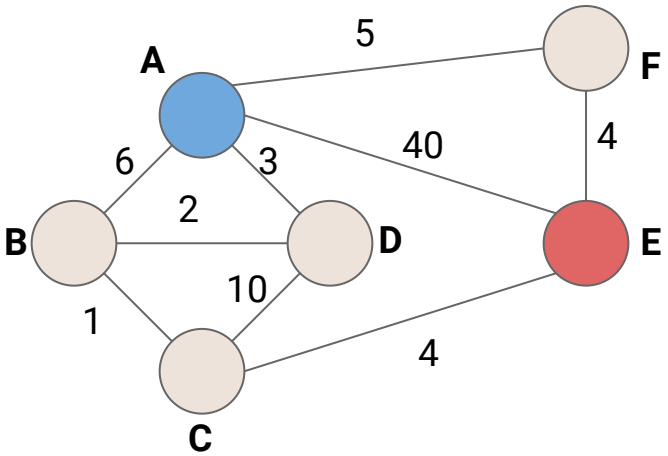
TARGET

✓ VISITED

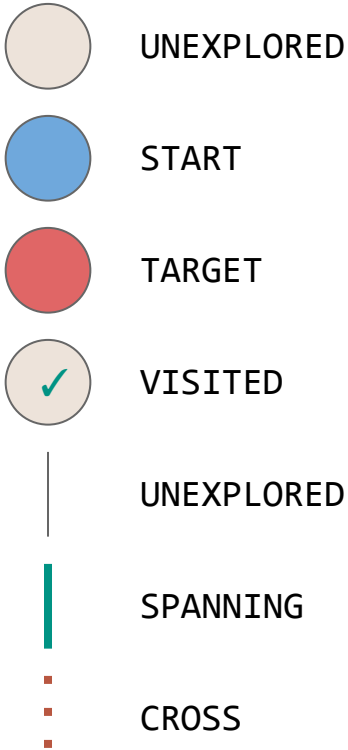
UNEXPLORED

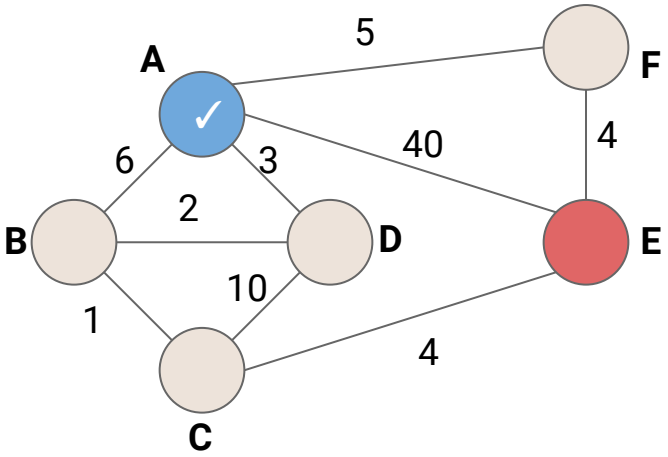
SPANNING

CROSS



Desired Exploration Order





Desired Exploration Order

UNEXPLORED

START

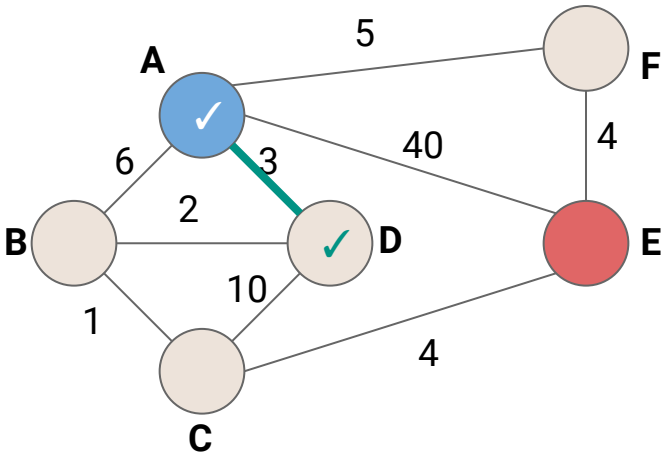
TARGET

VISITED

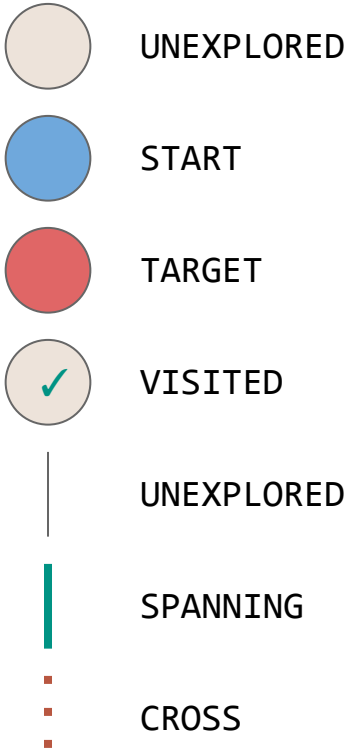
UNEXPLORED

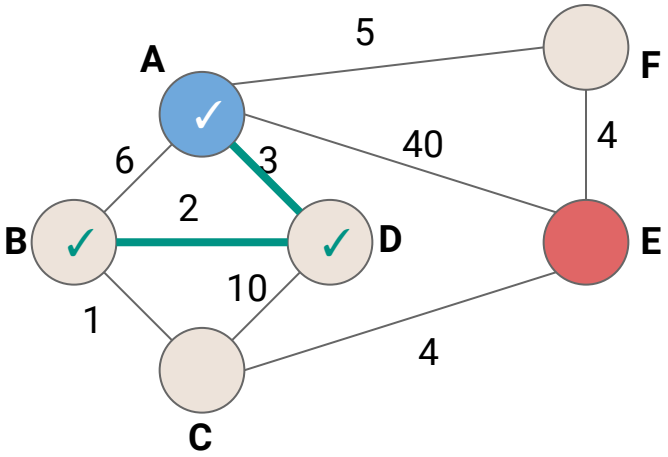
SPANNING

CROSS

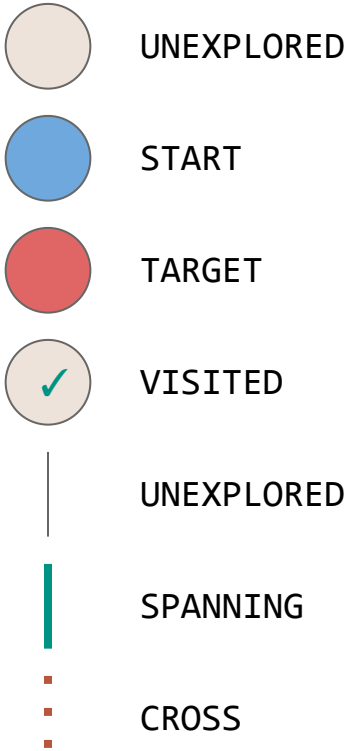


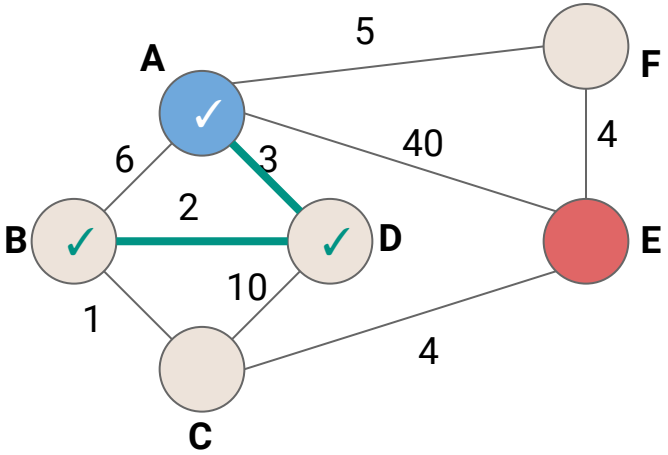
Desired Exploration Order





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

UNEXPLORED

START

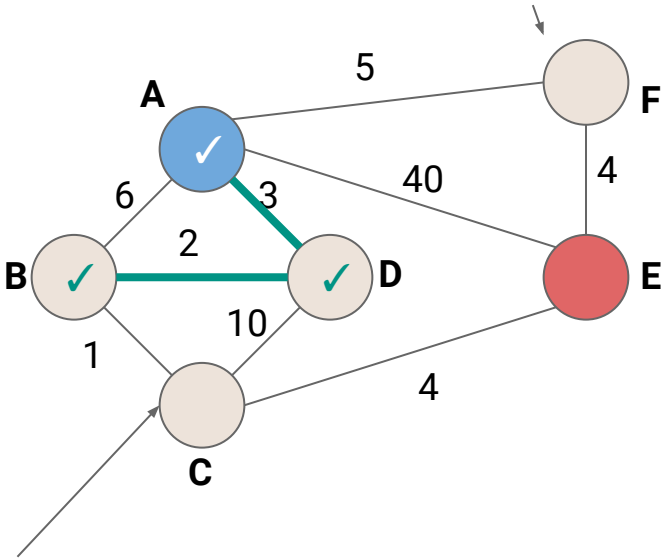
TARGET

VISITED

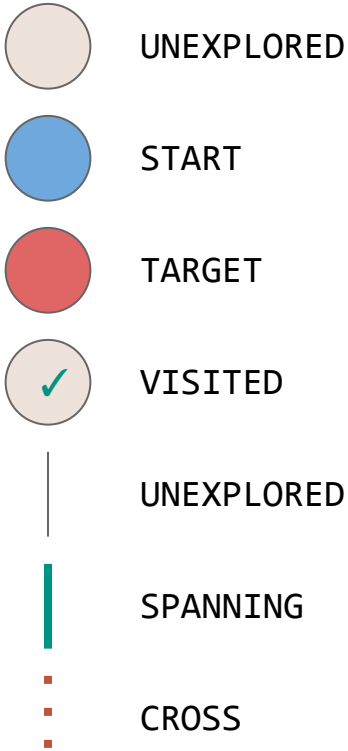
UNEXPLORED

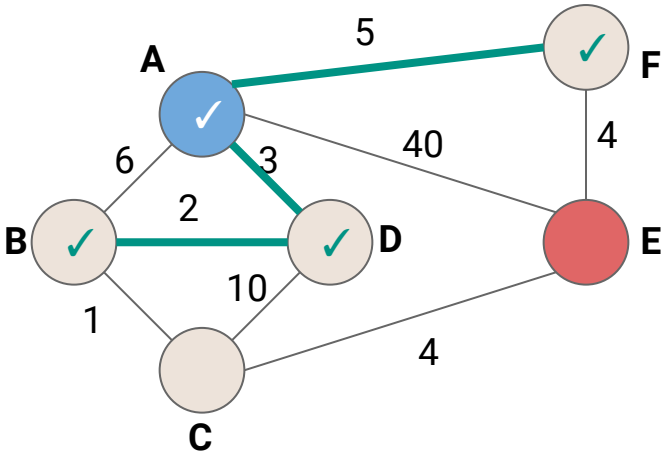
SPANNING

CROSS

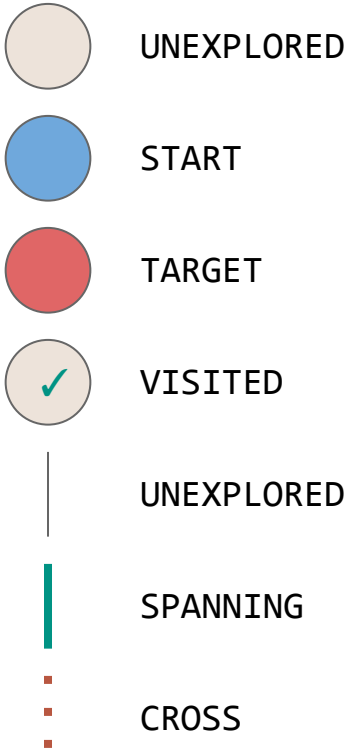


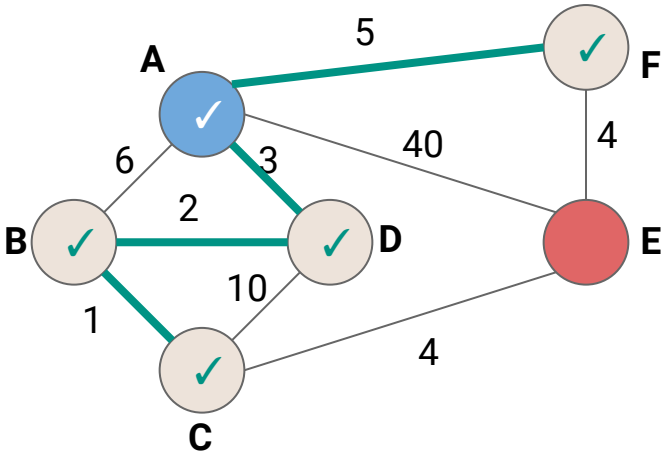
Desired Exploration Order



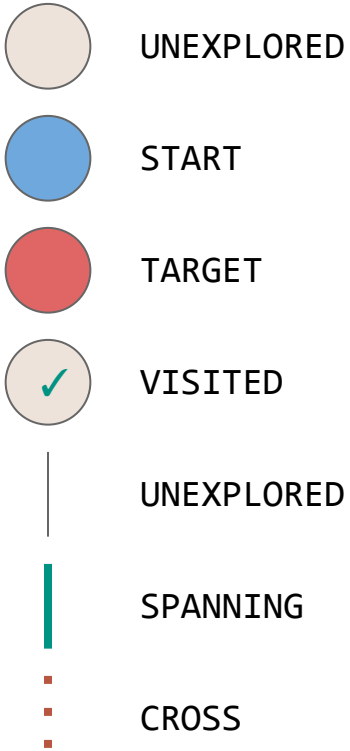


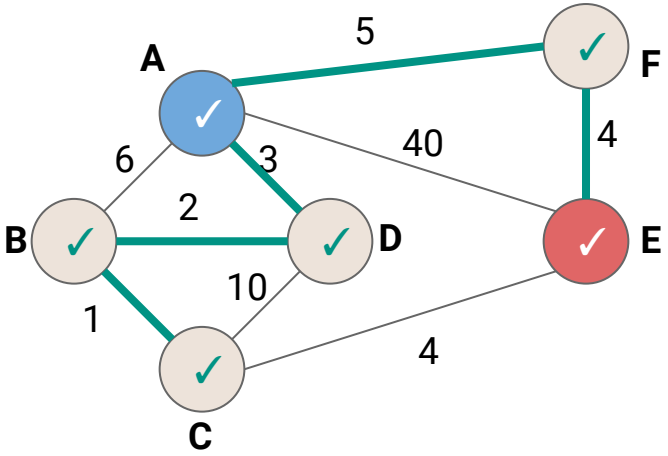
Desired Exploration Order





Desired Exploration Order





Path Found!

```
1  public void BFSOne(Graph graph, Vertex v) {
2    Queue<Vertex> todo = new Queue<>();
3    v.setLabel(VISITED);
4    todo.enqueue(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.dequeue();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.weight + e.weight));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

We want to be able to dequeue in order of weight…but how?

A New ADT…`PriorityQueue`

`PriorityQueue<T>`

`void add(T value)`

Insert **`value`** into the priority queue

`T poll()`

Remove the highest priority value in the priority queue

`T peek()`

Peek at the highest priority value in the priority queue

A New ADT...`PriorityQueue`

`PriorityQueue<T>`

`void add(T value)`

Insert `value` into the priority queue

Next class: What is priority? How do we define it?

`T poll()`

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

`T peek()`

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

```
1  public void Djikstras(Graph graph, Vertex v) {
2    PriorityQueue<Vertex> todo = new PriorityQueue<>();
3    v.setLabel(VISITED);
4    todo.add(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.poll();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.weight + e.weight));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

```
1  public void Djikstras(Graph graph, Vertex v) {
2    PriorityQueue<Vertex> todo = new PriorityQueue<>();
3    v.setLabel(VISITED);
4    todo.add(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.poll();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.weight + e.weight));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

We can use a PriorityQueue for our TODO list to explore vertices in order from closest to furthest from our starting point!

```
1  public void Djikstras(Graph graph, Vertex v) {
2    PriorityQueue<Vertex> todo = new PriorityQueue<>();
3    v.setLabel(VISITED);
4    todo.add(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.poll();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.weight + e.weight));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

Is this an OK time to consider a vertex VISITED? Remember when we mark something VISITED we will never consider it again…

```
1  public void Djikstras(Graph graph, Vertex v) {
2    PriorityQueue<Vertex> todo = new PriorityQueue<>();
3    v.setLabel(VISITED);
4    todo.add(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.poll();
7      for (Edge e : curr.vertex.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(new TodoEntry(w, curr.weight + e.weight));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

Is this an OK time to consider a vertex VISITED? Remember when we mark something VISITED we will never consider it again…**No We may find it via a shorter path later in our search!**

```
1  public void Djikstras(Graph graph, Vertex v) {
2    PriorityQueue<Vertex> todo = new PriorityQueue<>();
3    v.setLabel(VISITED);
4    todo.add(new TodoEntry(v,0));
5    while (!todo.isEmpty()) {
6      TodoEntry curr = todo.poll();
7      curr.setLabel(VISITED);
8      for (Edge e : curr.vertex.outEdges) {
9        if (e.label == UNEXPLORED) {
10         Vertex w = e.to;
11         if (w.label == UNEXPLORED) {
12           e.setLabel(SPANNING);
13           todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
14         } else {
15           e.setLabel(BACK);
16 }}}}}
```

Instead, we can only consider something visited when we remove if from our TODO list!

This is called **Djikstra's Algorithm**