CSE 250
Data Structures

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

Lec 22: Shortest Path

Warm Up Question

If $f(n) \in O(n^3)$, then:

(a) $f(n)$ must be in $\Theta(n^2)$
(b) $f(n)$ could be in $\Theta(n^2)$
(c) $f(n)$ can't be in $\Theta(n^2)$

If $f(n) \in \Theta(n^2)$, then:

(a) $f(n)$ must be in $O(n^3)$
(b) $f(n)$ could be in $O(n^3)$
(c) $f(n)$ can't be in $O(n^3)$

Warm Up Question

If $f(n) \in O(n^3)$, then:

(a) ~~$f(n)$ must be in $\Theta(n^2)$~~

(b) $f(n)$ could be in $\Theta(n^2)$

(c) ~~$f(n)$ can't be in $\Theta(n^2)$~~

If $f(n) \in \Theta(n^2)$, then:

(a) $f(n)$ must be in $O(n^3)$

(b) ~~$f(n)$ could be in $O(n^3)$~~

(c) ~~$f(n)$ can't be in $O(n^3)$~~

Announcements

- PA2 testing due on Sunday @ 11:59PM
  - No late submissions or grace days accepted
  - AutoLab now open
  - Can get a 6/5 with bonus point
- Mid Term grades posted (see Piazza @ @275)
- Mid Semester grades posted (see Piazza @280)

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?

```
public class TodoEntry {
  public Vertex vertex;
  public List<Edge> path;
}
```

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

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

Begin our search at the starting vertex with an empty path

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

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

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

What's the problem with this solution?

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

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

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

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

Create a map to store the necessary info

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

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

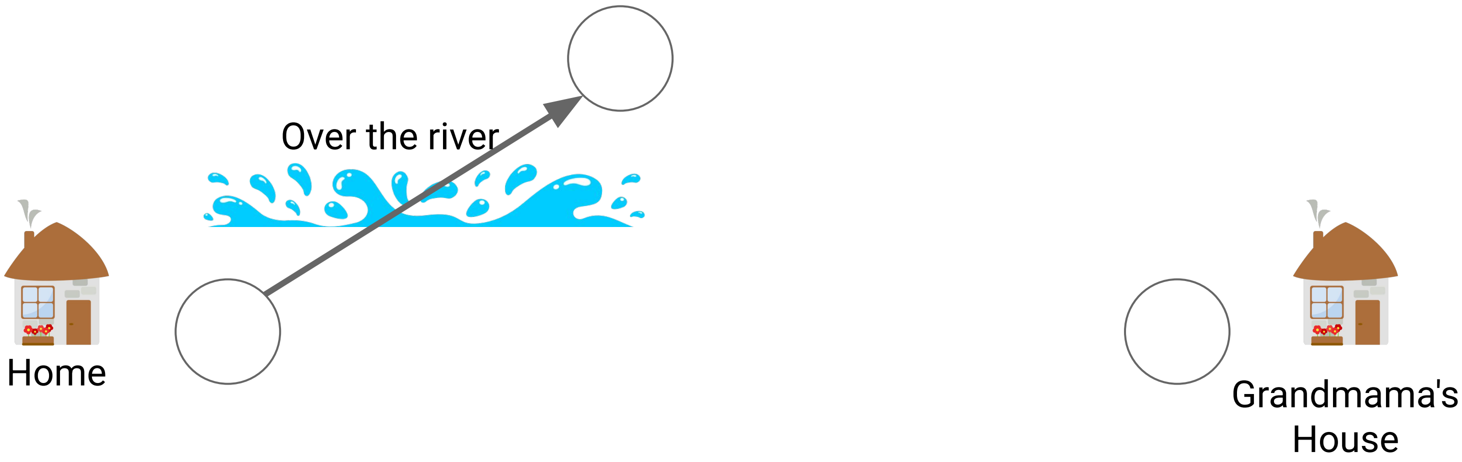
1. Set our **`current`** vertex to the vertex we want to find a path to
2. While the **`current`** vertex is not our starting vertex:
    a. Use our **`edgeTo`** map to lookup the edge that led to the **`current`** vertex
    b. Prepend that edge to our path
    c. Set **`current`** to the edges origin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

Quick Demo!

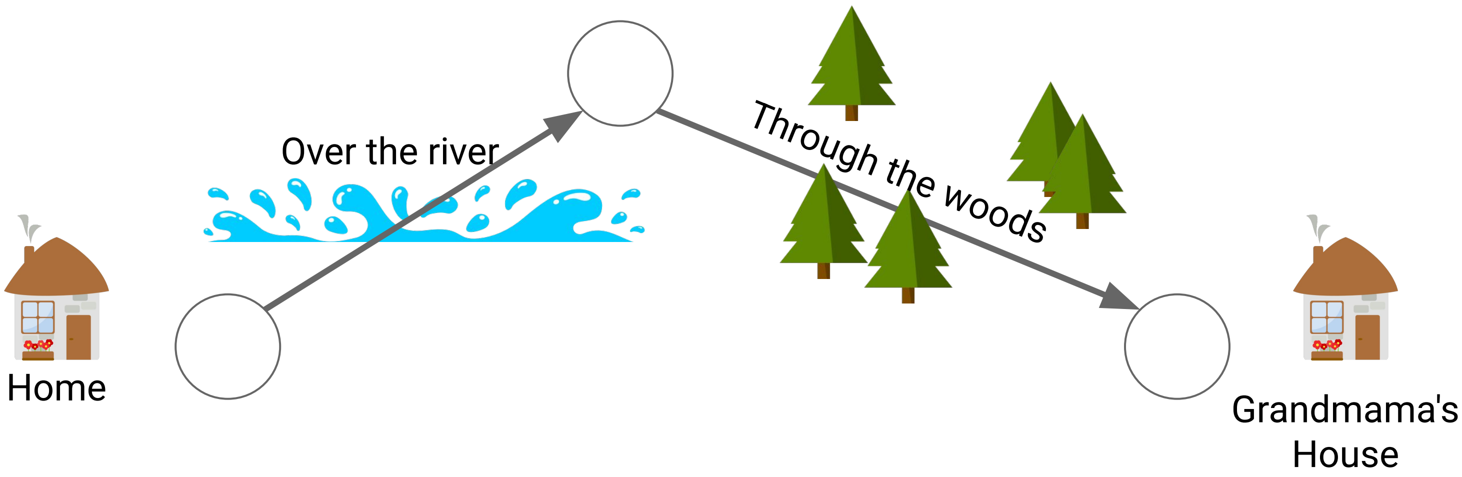
Now we can use BFS to find the "Shortest Paths" from any starting vertex…or can we???

Shortest Paths

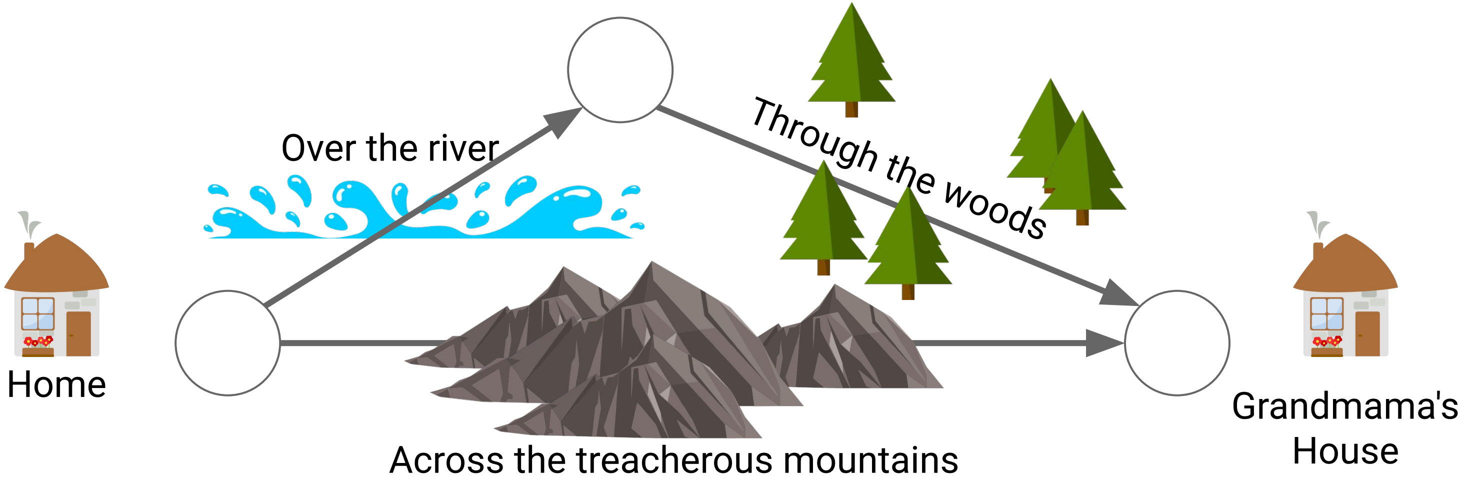
Shortest Paths



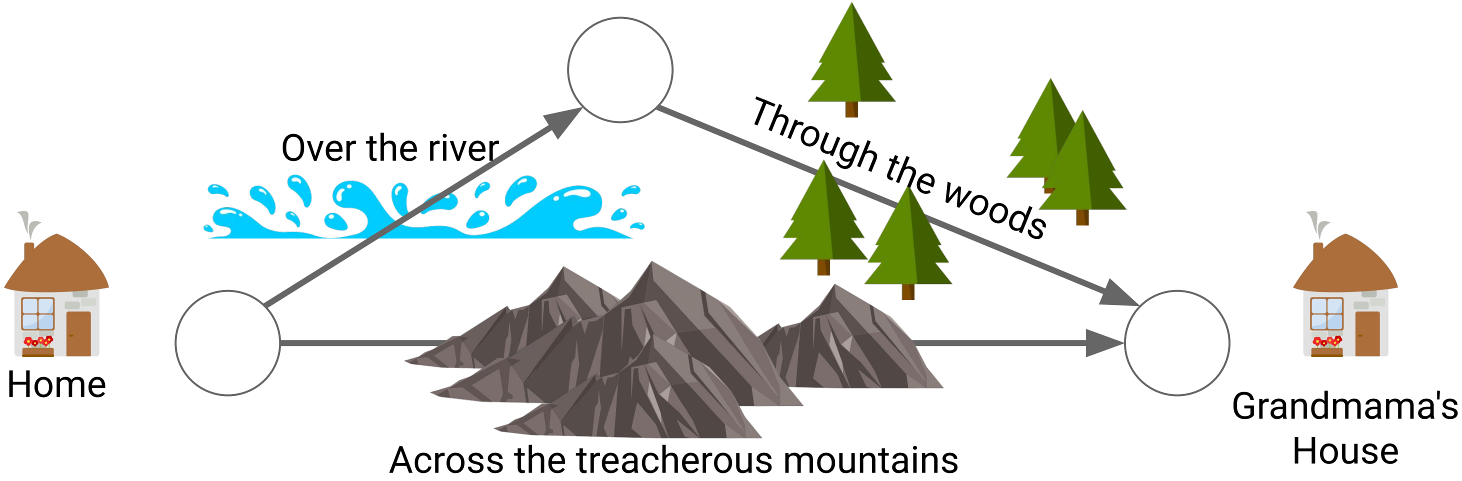
Shortest Paths



Shortest Paths

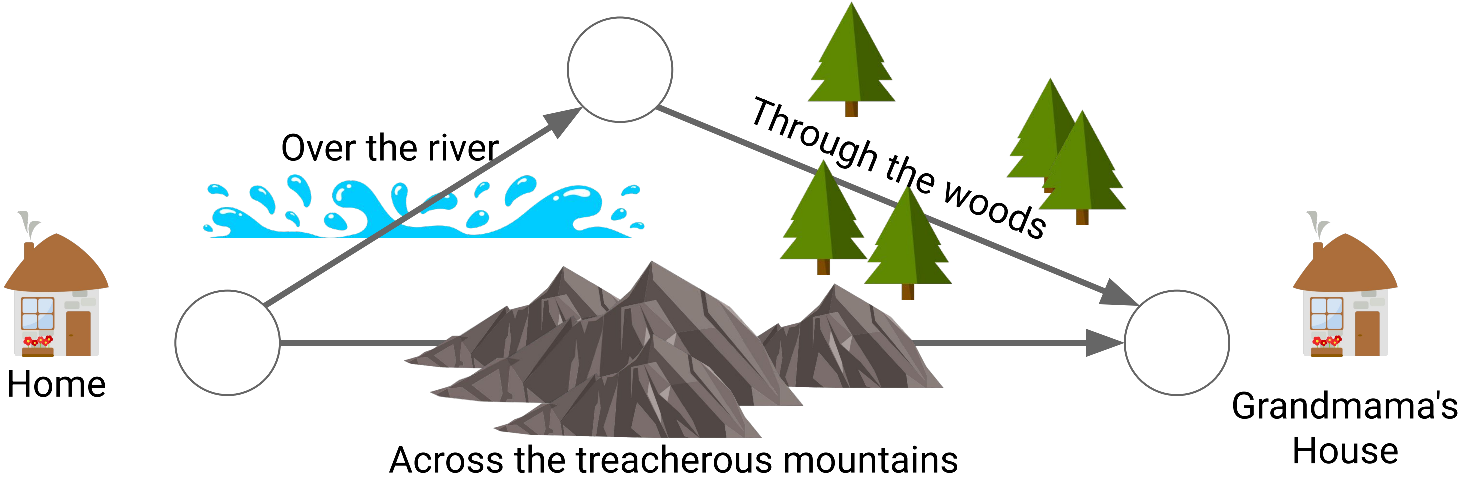


Shortest Paths



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 $\boldsymbol{G = (V,E)}$
- 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 $\boldsymbol{G = (V,E,\omega)}$
- A start vertex ***start*** in ***V***
- An end vertex ***end*** in ***V***

Shortest Path

Given:

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

Goal:

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

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)

```
public class TodoEntry {
  public Vertex vertex;
  public Integer level;
}
```

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.dequeue();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.level + 1));
        } else {
          e.setLabel(BACK);
}}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.dequeue();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.level + 1));
        } else {
          e.setLabel(BACK);
}}}}}
```

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

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.dequeue();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.level + 1));
        } else {
          e.setLabel(BACK);
}}}}}
```

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)

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.dequeue();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.level + 1));
        } else {
          e.setLabel(BACK);
}}}}}
```

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 $\boldsymbol{\omega(e) = 1}$ for all $\boldsymbol{e}$

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

Detailed Example

UNEXPLORED

START

TARGET

VISITED

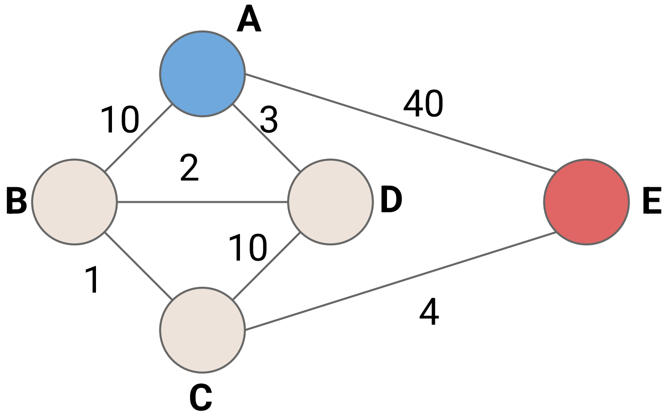
UNEXPLORED

SPANNING

CROSS

Call Stack

Work Queue



Detailed Example

UNEXPLORED

START

TARGET

VISITED

UNEXPLORED

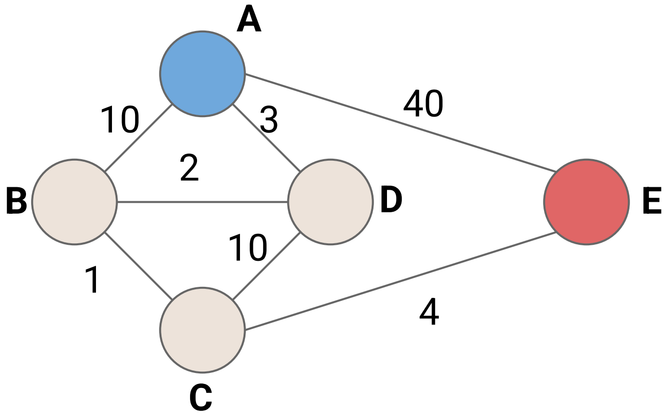
SPANNING

CROSS

<u>Call Stack</u>

`BFS(G)`

<u>Work Queue</u>



Detailed Example

UNEXPLORED

START

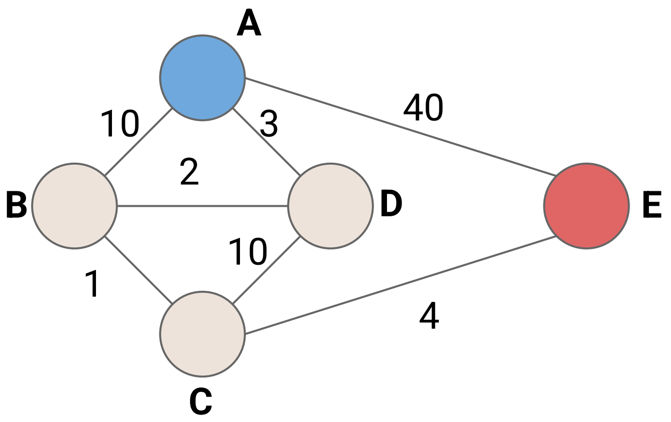
TARGET

✓ VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>

`BFS(G)`
`BFSOne(G,A)`

<u>Work Queue</u>

Detailed Example

UNEXPLORED

START

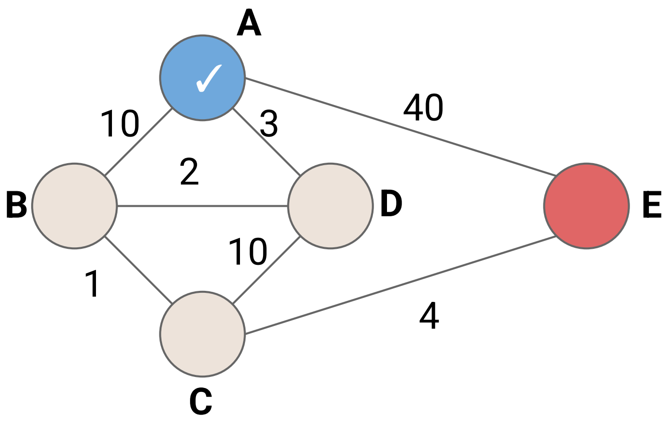
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>
`BFS(G)`
`BFSOne(G,A)`

<u>Work Queue</u>
A

Detailed Example

UNEXPLORED

START

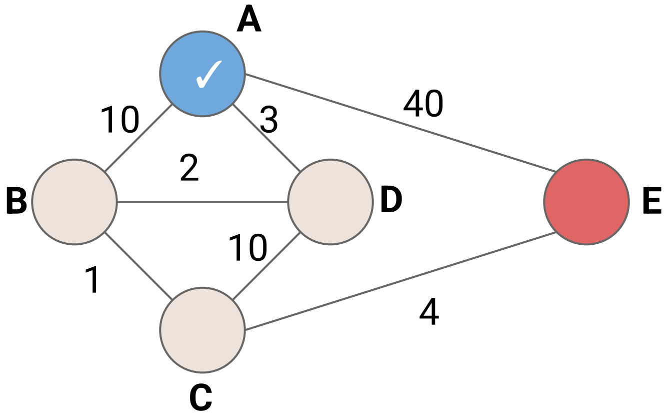
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



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

Work Queue
~~A~~

Detailed Example

UNEXPLORED

START

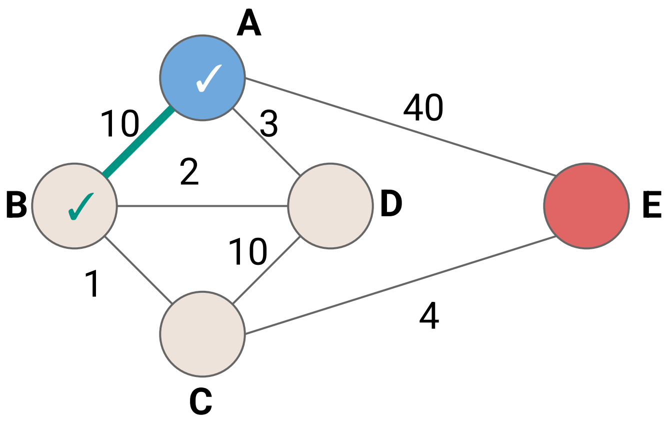
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>

BFS(G)

BFSOne(G,A)

<u>Work Queue</u>

~~A~~

B

Detailed Example

UNEXPLORED

START

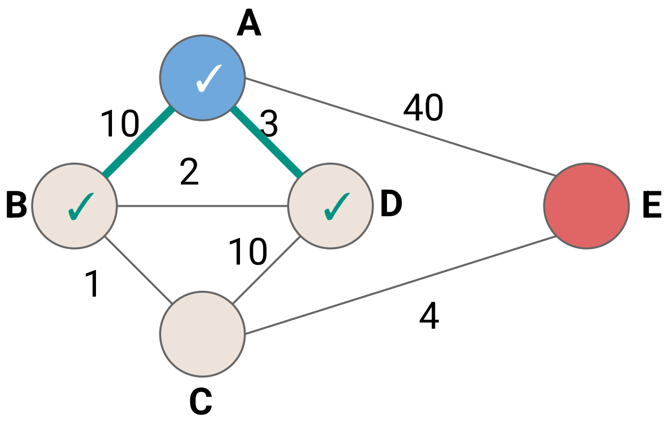
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



<u>Call Stack</u>

BFS(G)

BFSOne(G,A)

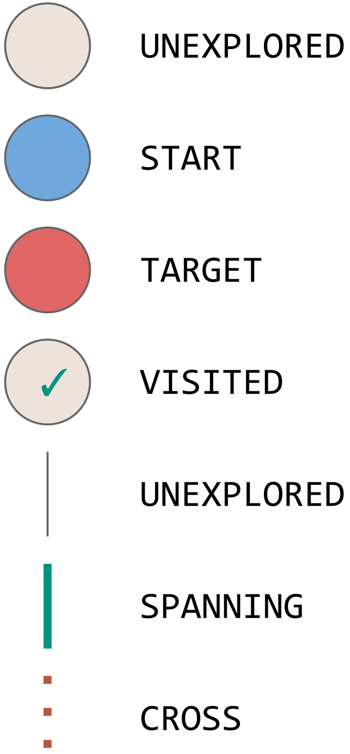
<u>Work Queue</u>

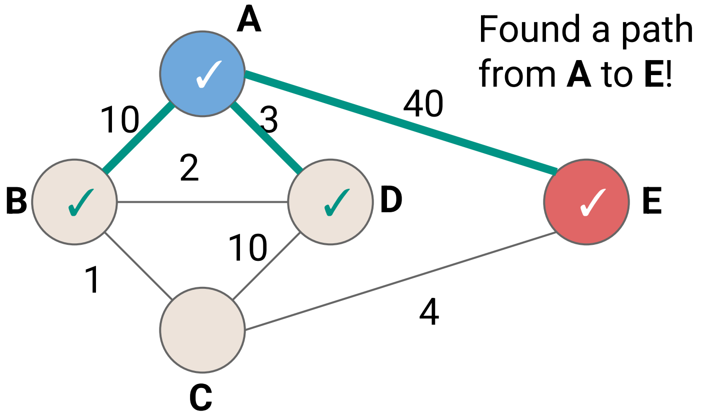
~~A~~

B

D

Detailed Example





<u>Call Stack</u>
BFS(G)
BFSOne(G,A)

<u>Work Queue</u>
~~**A**~~
B
D
E

Detailed Example - Desired Path

UNEXPLORED

START

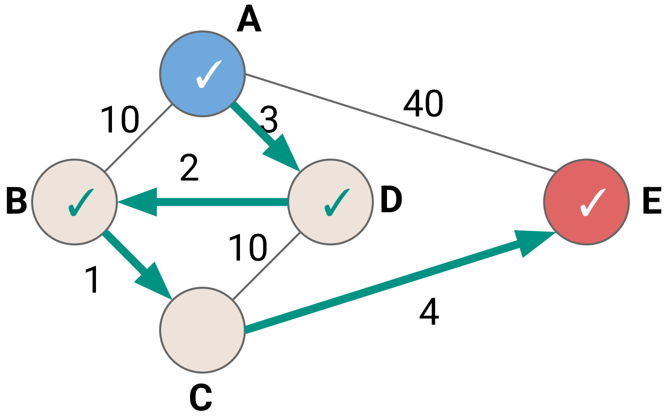
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



How do we find this path?

Shortest Path

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

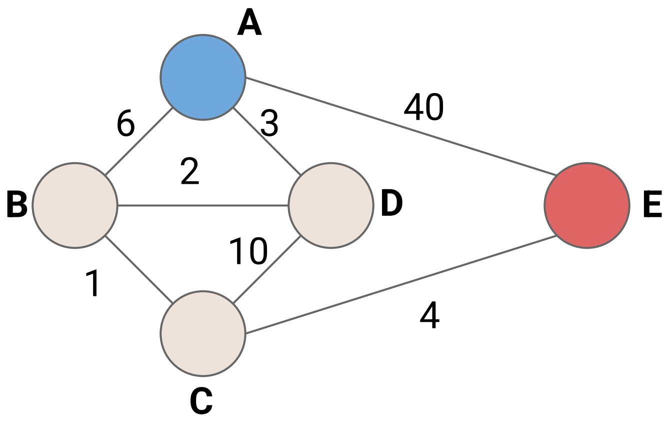
Shortest Path

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

- UNEXPLORED
- START
- TARGET
- VISITED
- UNEXPLORED
- SPANNING
- CROSS



Attempt #1: Explore Smallest Edge

UNEXPLORED

START

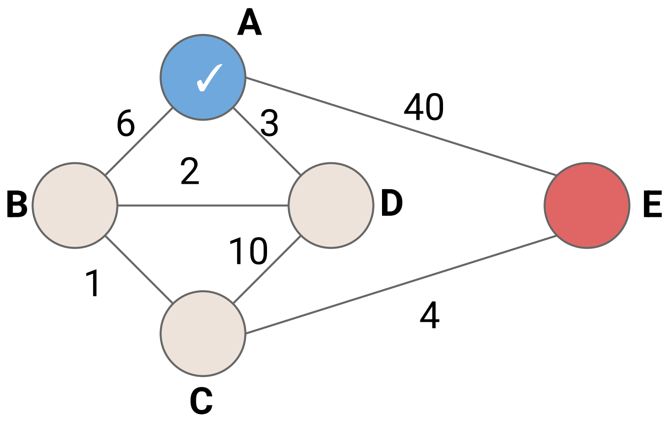
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Attempt #1: Explore Smallest Edge

UNEXPLORED

START

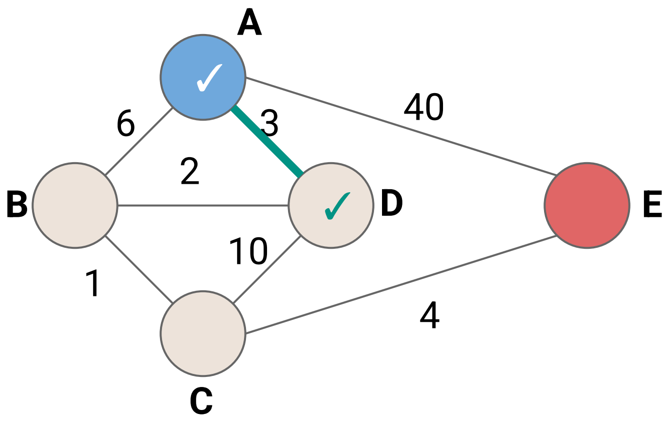
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Attempt #1: Explore Smallest Edge

UNEXPLORED

START

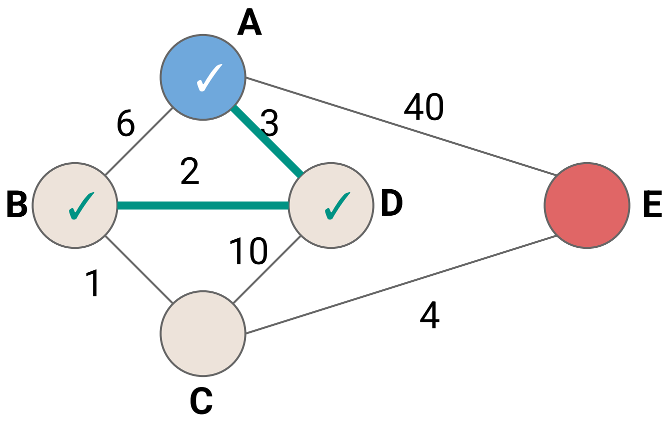
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Attempt #1: Explore Smallest Edge

UNEXPLORED

START

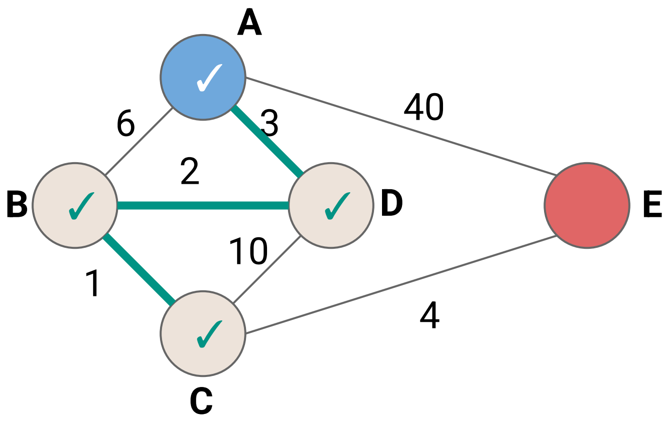
TARGET

VISITED

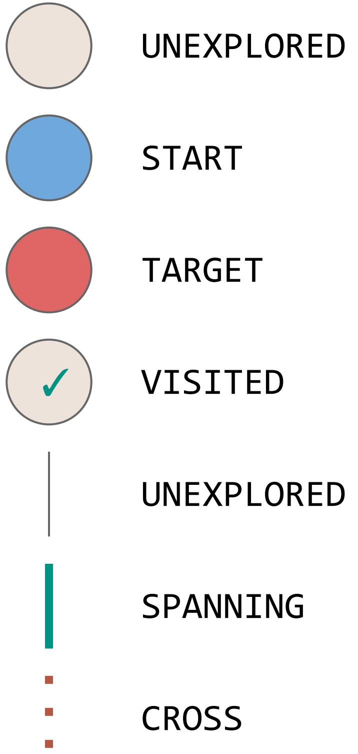
UNEXPLORED

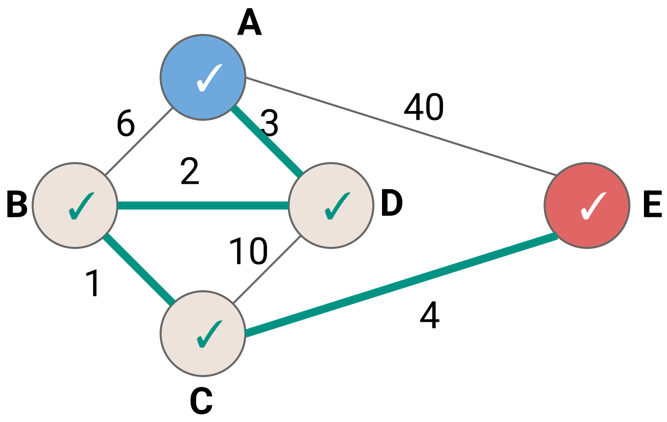
SPANNING

CROSS



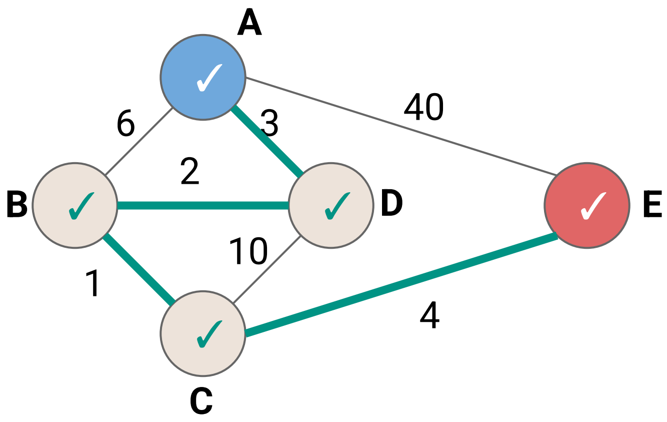
Attempt #1: Explore Smallest Edge





Attempt #1: Explore Smallest Edge

- UNEXPLORED
- START
- TARGET
- VISITED
- UNEXPLORED
- SPANNING
- CROSS



Attempt #1: Explore Smallest Edge

Will exploring the smallest available edge always work?

Desired Exploration Order

UNEXPLORED

START

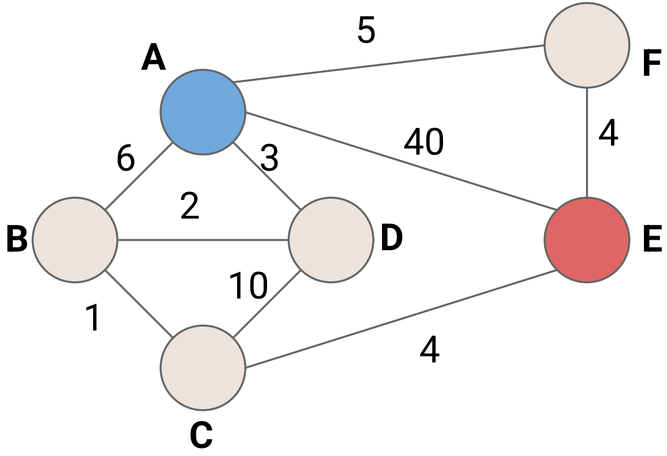
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Desired Exploration Order

UNEXPLORED

START

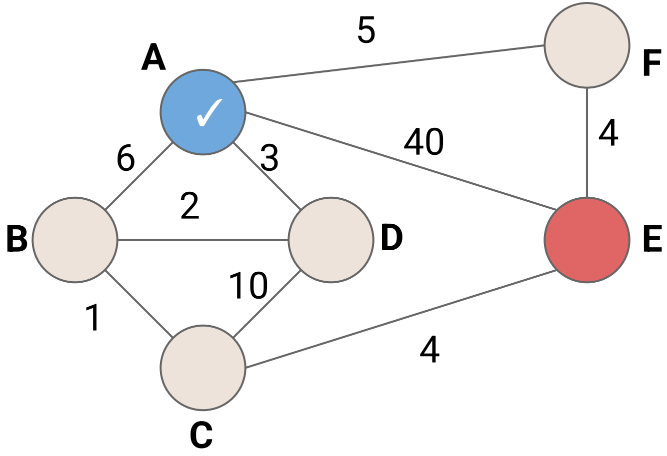
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Desired Exploration Order

UNEXPLORED

START

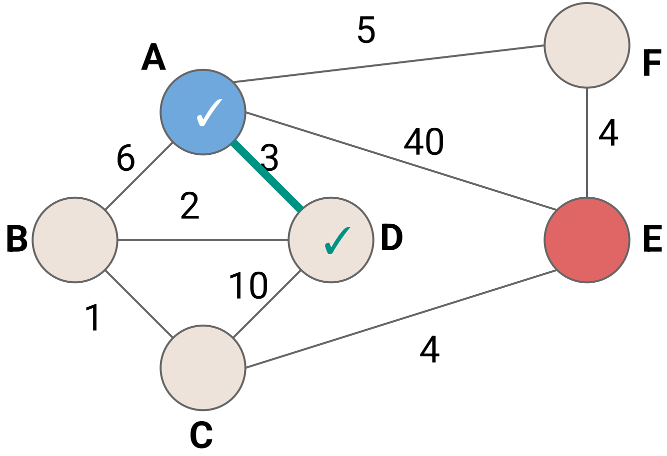
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Desired Exploration Order

UNEXPLORED

START

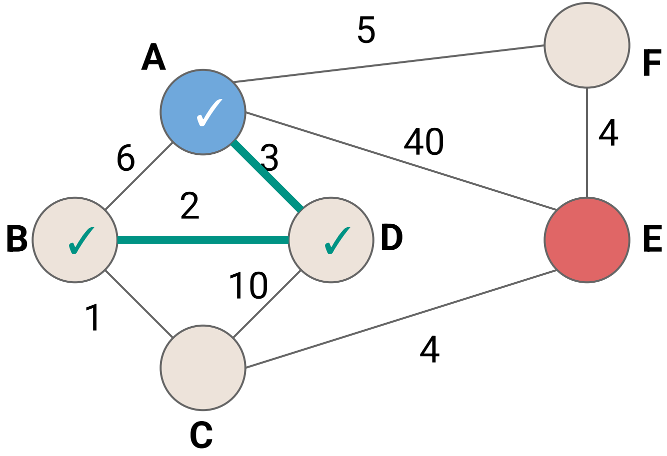
TARGET

VISITED

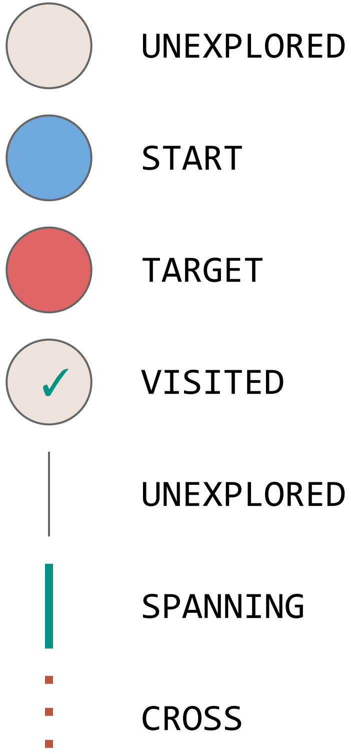
UNEXPLORED

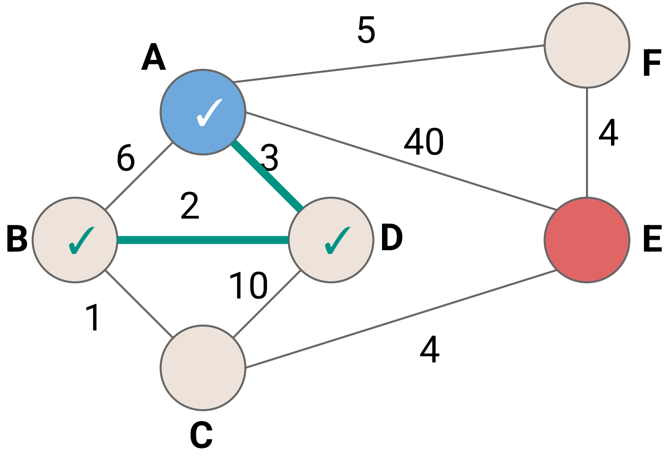
SPANNING

CROSS



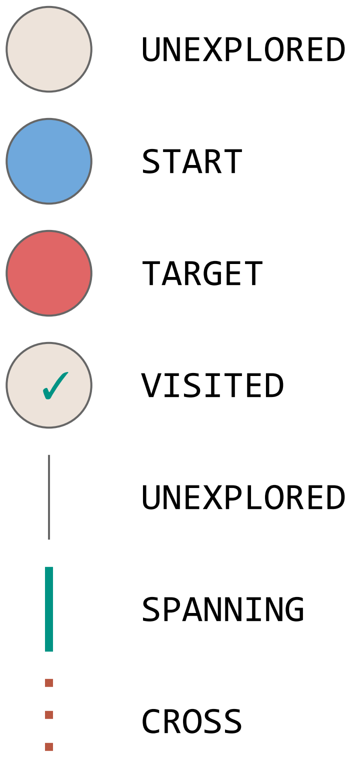
Desired Exploration Order

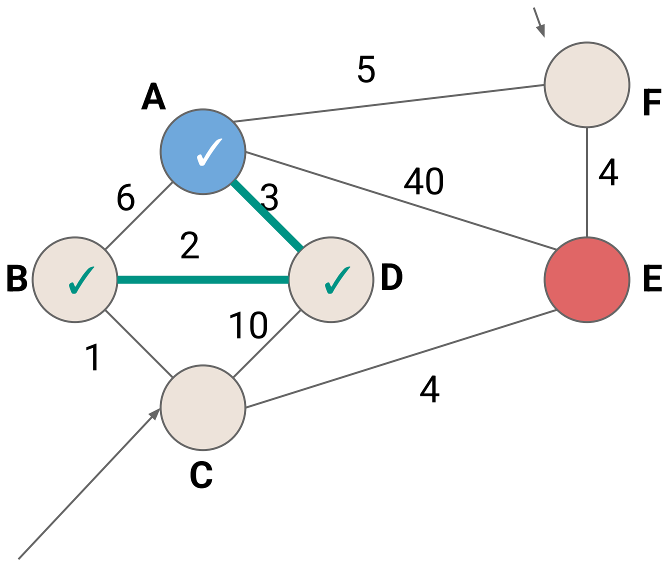




If we follow the smallest edge, we will explore **C** next.
But how far is **C** from **A**?
Are there any unexplored vertices that are closer to **A**?

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

UNEXPLORED

START

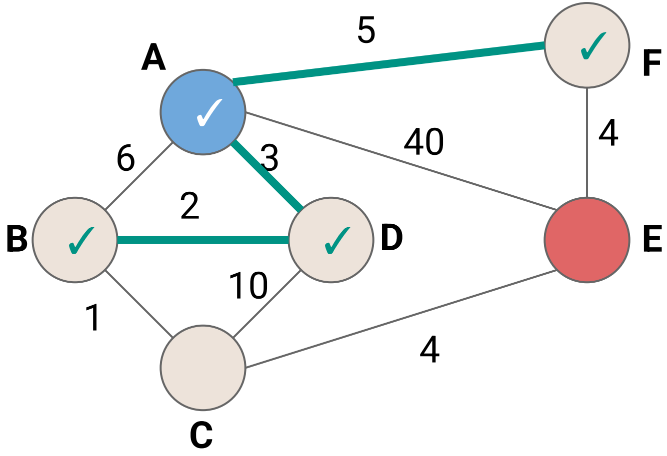
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Desired Exploration Order

UNEXPLORED

START

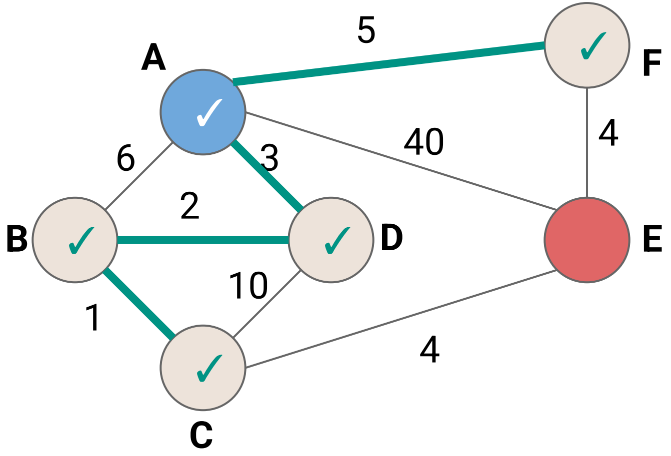
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Desired Exploration Order

UNEXPLORED

START

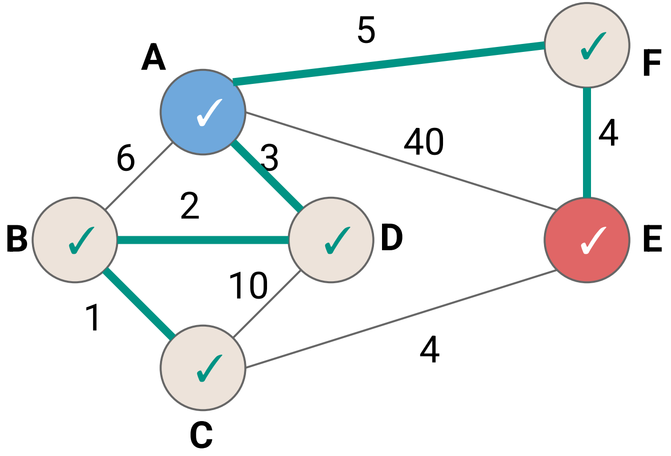
TARGET

VISITED

UNEXPLORED

SPANNING

CROSS



Path Found!

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.dequeue();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

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

`T poll()`

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

`T peek()`

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

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

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

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

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

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

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

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!**

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    curr.setLabel(VISITED);
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

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

This is called **Djikstra's Algorithm**

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    curr.setLabel(VISITED);
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

Step 1: Create a TODO list that prioritizes vertices based on their distance from the starting vertex

Step 2: Loop until we run out of things on our TODO list

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    curr.setLabel(VISITED);
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    curr.setLabel(VISITED);
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

Step 2a: When we take something off our TODO list, we know we found it via the shortest possible path, so mark it VISITED

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    curr.setLabel(VISITED);
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

Step 2b: Look through all of the outgoing edges of the vertex we just removed, and add any newly discovered vertices to our TODO list

Step 2b: Look through all of the outgoing edges of the vertex we just removed, and add any newly discovered vertices to our TODO list

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<Vertex> todo = new PriorityQueue<>();
  v.setLabel(VISITED);
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    curr.setLabel(VISITED);
    for (Edge e : curr.vertex.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          e.setLabel(SPANNING);
          todo.enqueue(new TodoEntry(w, curr.weight + e.weight));
        } else {
          e.setLabel(BACK);
}}}}}
```

Give them a priority equal to the total distance required to reach them!