CSE 250
Data Structures

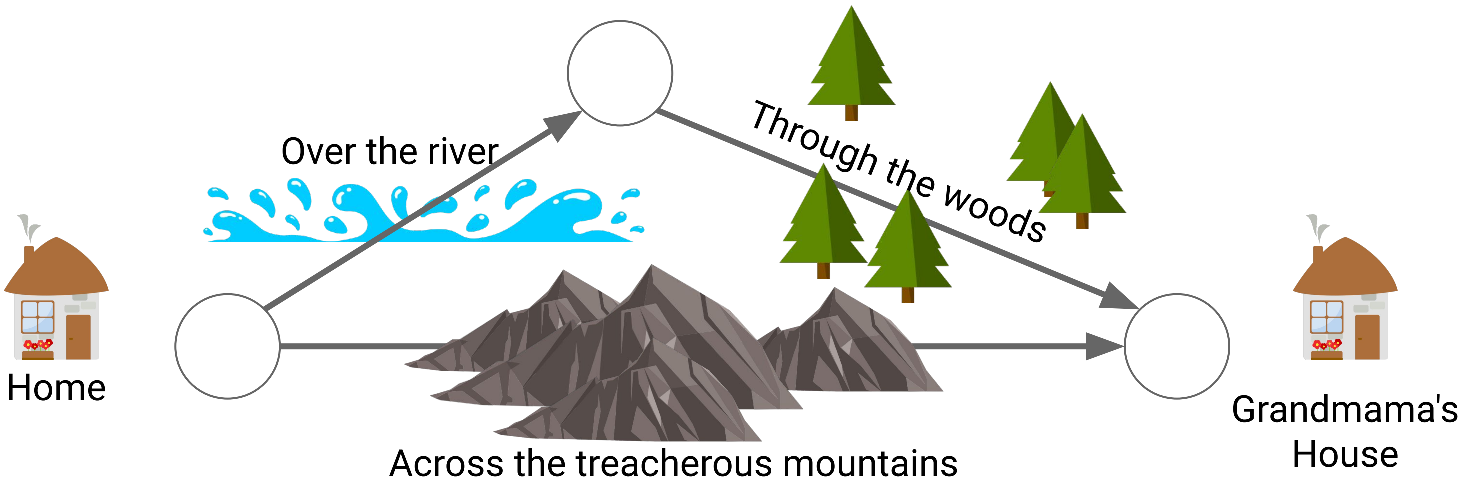
Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Shortest Path Revisited

Announcements

- PA3 has been released
  - Autograder for testing should be open by tonight
  - **START EARLY**
  - **READ THE ENTIRE HANDOUT!!**

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?

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)
        }
      }
    }
  }
}
```

BFS always adds 1 to the level when exploring new nodes. **One** edge adds **one** to the level.

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)

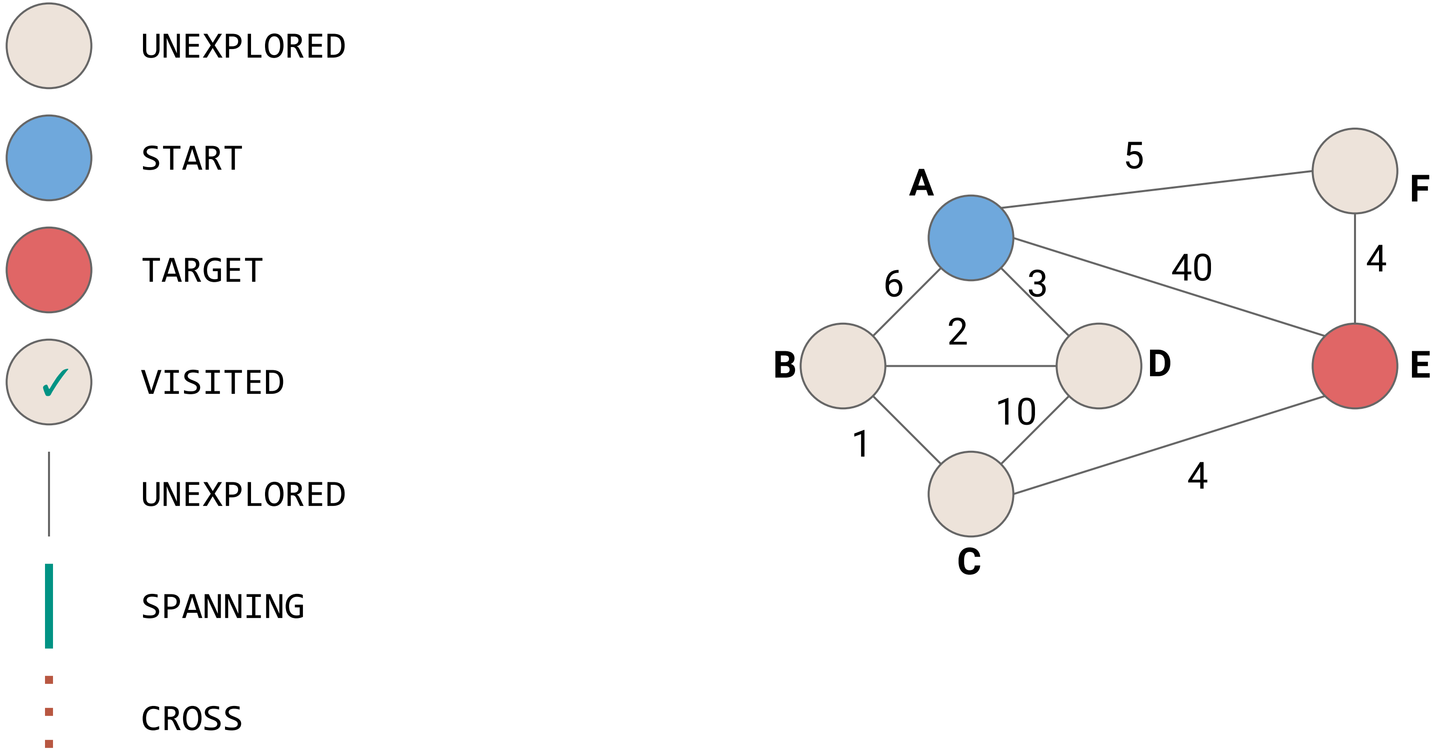
BFSOne - Shortest Path

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

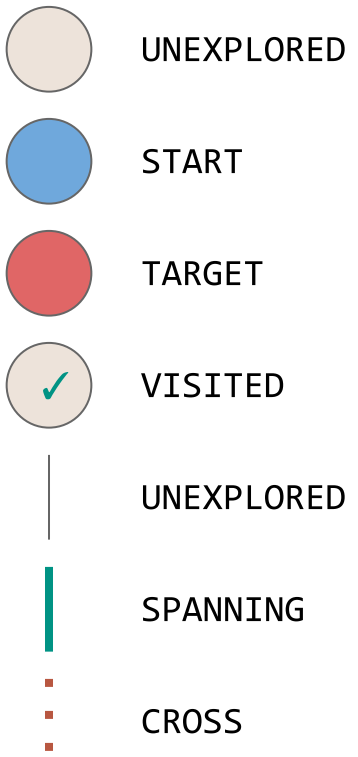
Consequence: Dequeue reads vertices in ascending order of level. (FIFO)

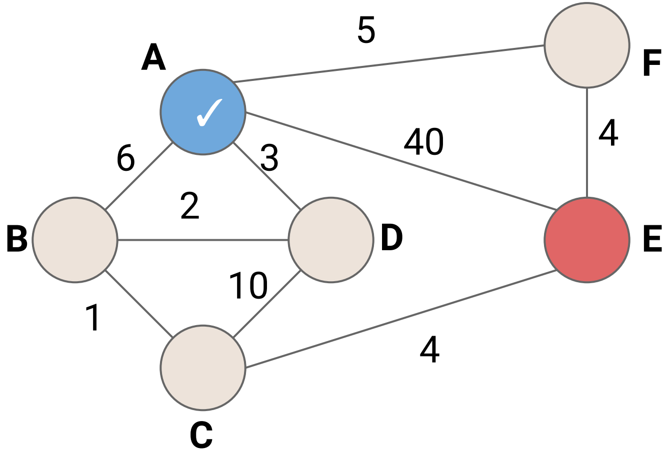
Therefore: The first time we visit a vertex, it is via the fewest number of edges from start.

Desired Exploration Order - Closest Vertex



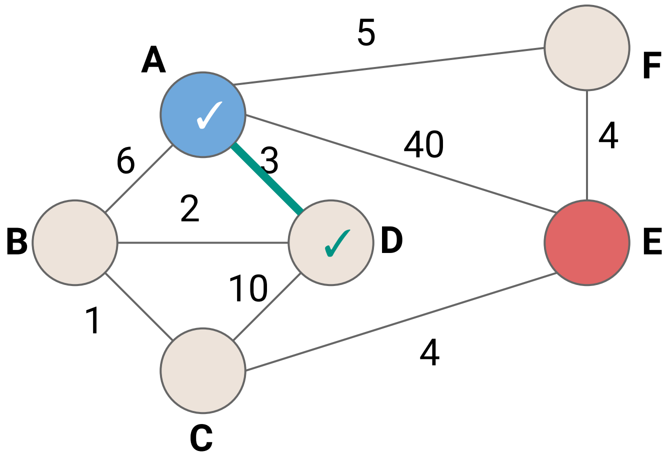
Desired Exploration Order - Closest Vertex



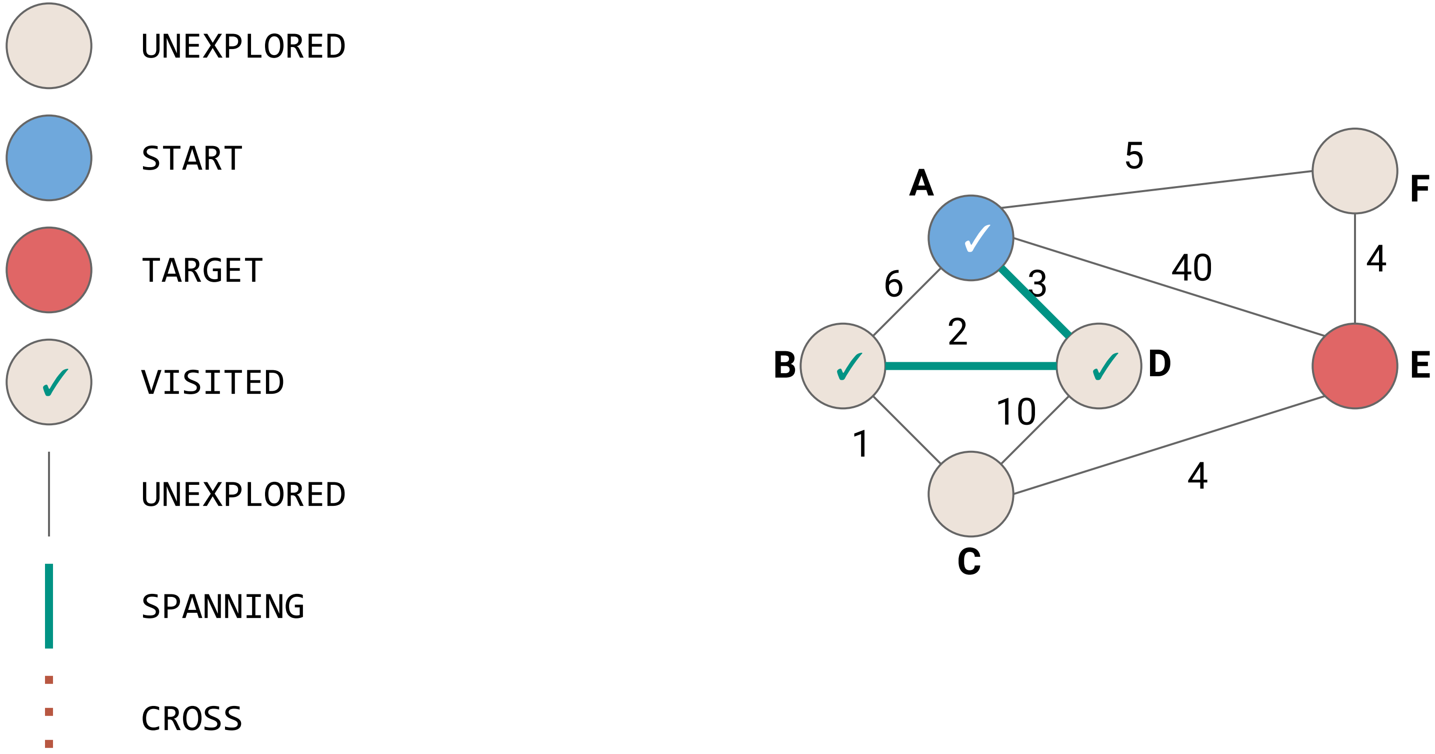


Desired Exploration Order - Closest Vertex

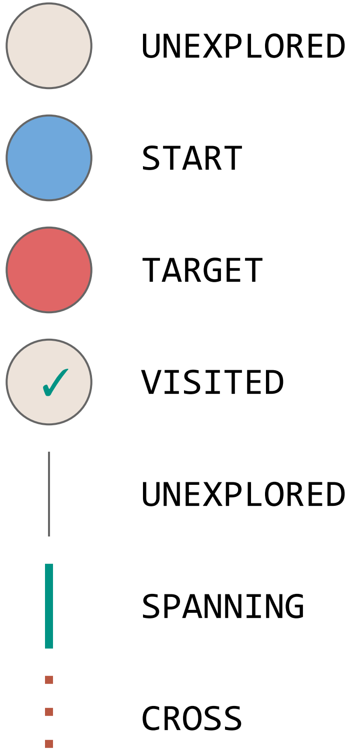
- UNEXPLORED
- START
- TARGET
- VISITED
- UNEXPLORED
- SPANNING
- CROSS



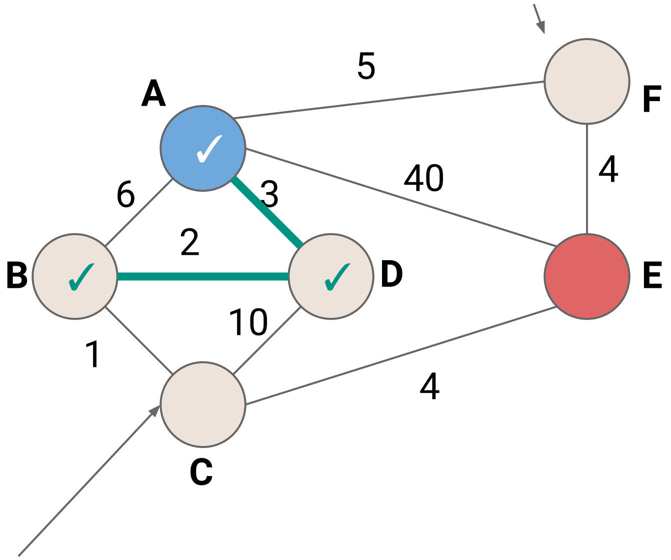
Desired Exploration Order - Closest Vertex



Desired Exploration Order - Closest Vertex

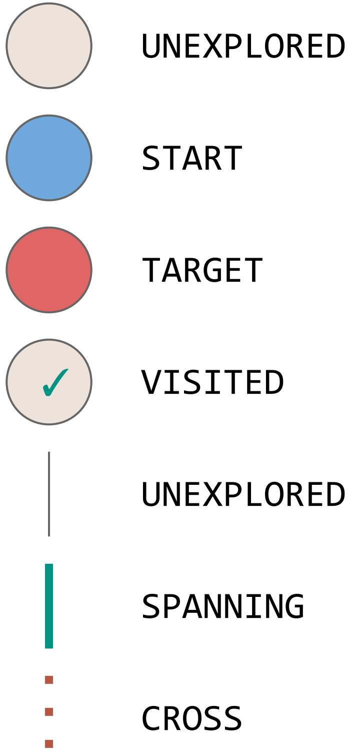


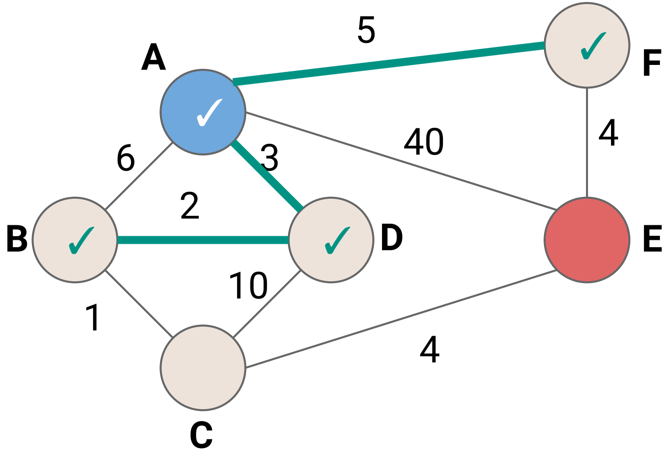
F is only 5 away…it is the closest unexplored vertex to **A**



C is 6 away from **A**

Desired Exploration Order - Closest Vertex





Desired Exploration Order - Closest Vertex

UNEXPLORED

START

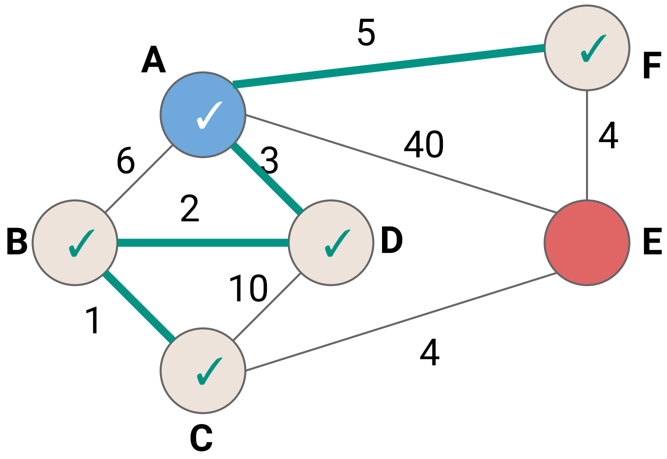
TARGET

VISITED

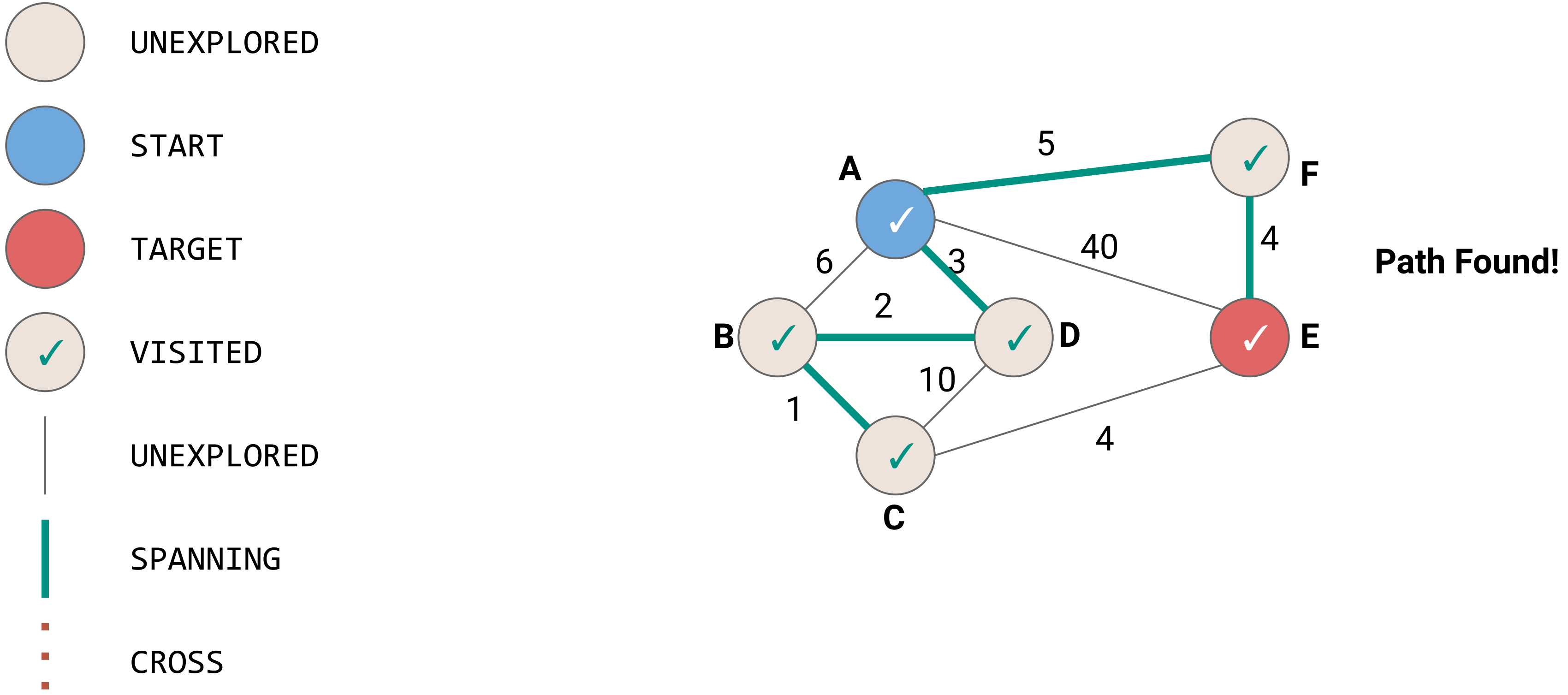
UNEXPLORED

SPANNING

CROSS



Desired Exploration Order - Closest Vertex



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?

PQSearch - Shortest Path Attempt #1

```
def PQSearch(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  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)
        }
      }
    }
  }
}
```

Use a PriorityQueue instead (ordered by distance)

PriorityQueue Attempt #1

UNEXPLORED

START

TARGET

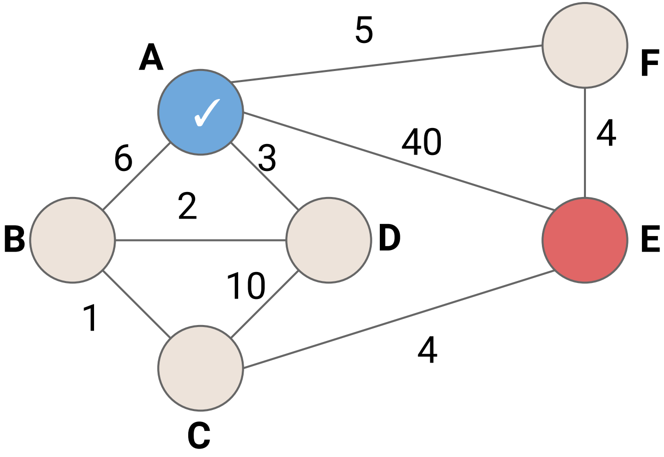
VISITED

UNEXPLORED

SPANNING

CROSS

PriorityQueue
(A, 0)



PriorityQueue Attempt #1

UNEXPLORED

START

TARGET

VISITED

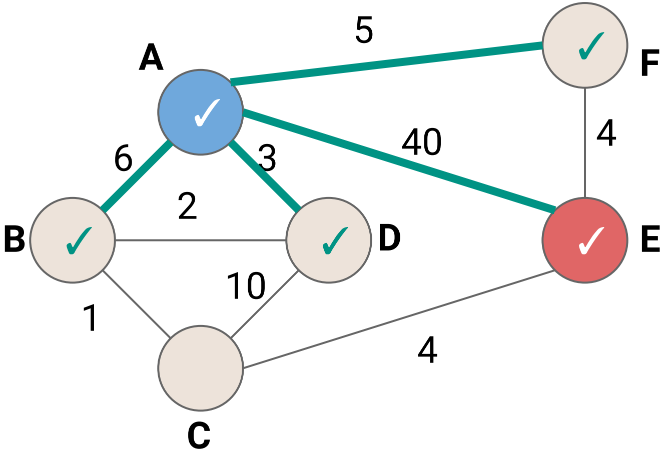
UNEXPLORED

SPANNING

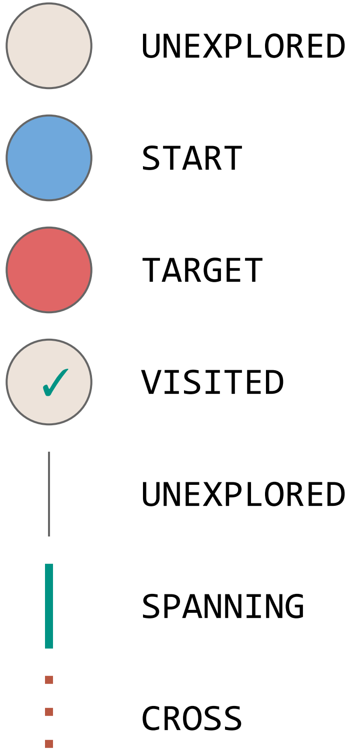
CROSS

PriorityQueue

~~(A, 0)~~
(B,6)
(D,3)
(F,5)
(E,40)

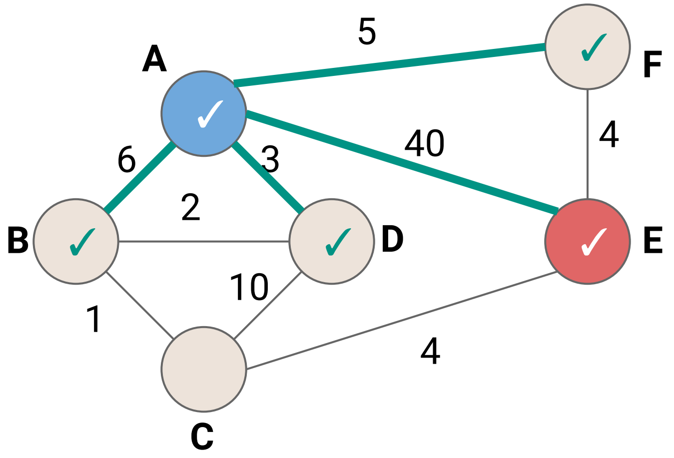


PriorityQueue Attempt #1



<u>PriorityQueue</u>

~~(A, 0)~~
(B,6)
(D,3)
(F,5)
(E,40)



We've just marked E as visited! Have we found the shortest path to E?
When should we consider something VISITED?

PQSearch - Shortest Path Attempt #1

```
def PQSearch(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  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)
        }
      }
    }
  }
}
```

When we enqueue, we don't know if there are better paths that could exist. We cannot consider it VISITED yet. When should we consider it visited?

PQSearch - Shortest Path Attempt #2

```
def PQSearch(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    (v, level) = work.dequeue()
    v.setLabel(VertexLabel.VISITED)
    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)))
        }
      }
    }
  }
}
```

What about when we dequeue?

PriorityQueue Attempt #2

UNEXPLORED

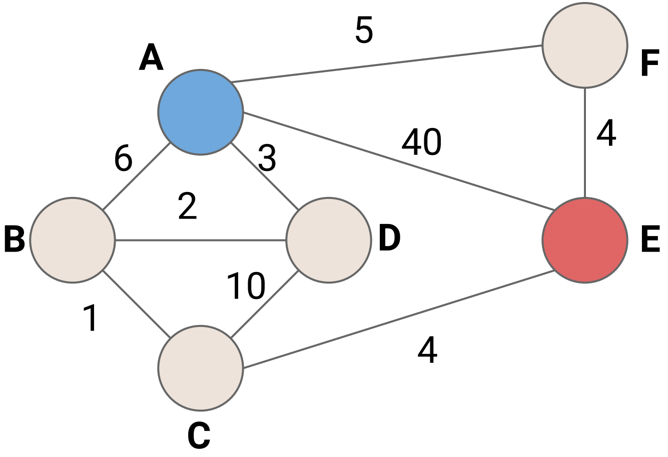
START

TARGET

✓ VISITED

UNEXPLORED

<u>PriorityQueue</u>
(A, 0)



PriorityQueue Attempt #2

UNEXPLORED

START

TARGET

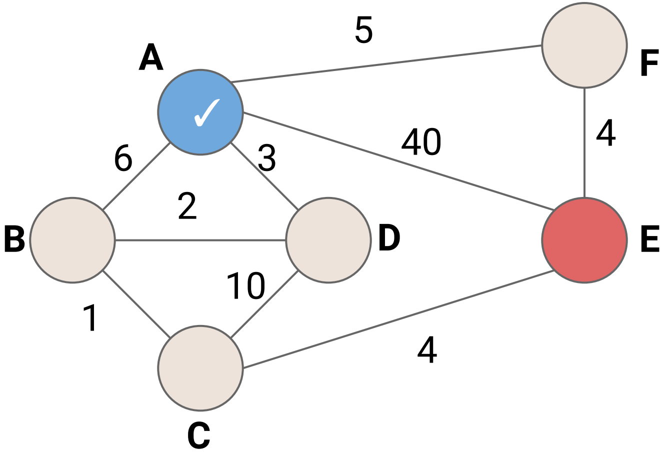
VISITED

UNEXPLORED

Mark A as visited because we just dequeued it. We now know there's no better path to A

PriorityQueue

~~(A, 0)~~
(B,6)
(D,3)
(F,5)
(E,40)



PriorityQueue Attempt #2

UNEXPLORED

START

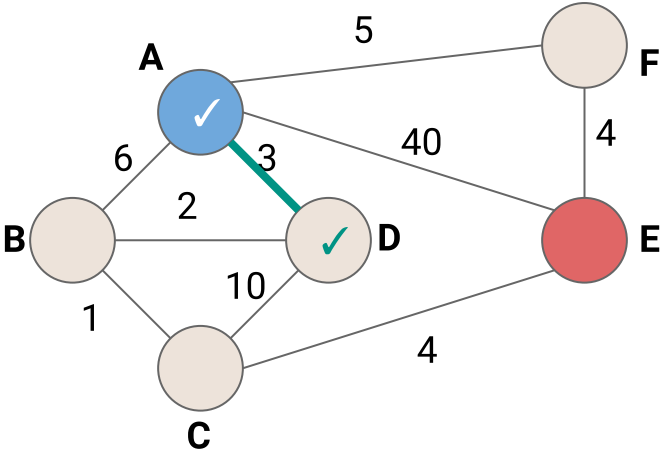
TARGET

VISITED

UNEXPLORED

PriorityQueue

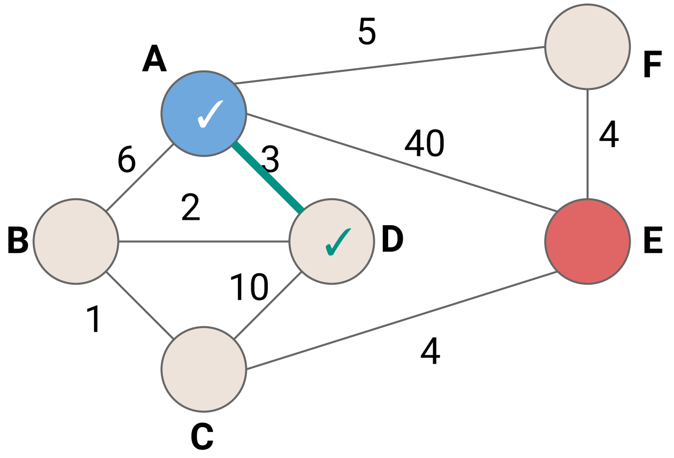
~~(A, 0)~~
(B,6)
~~(D,3)~~
(F,5)
(E,40)



Mark D as visited because we've found the shortest path to D. What should we add to the PQ now?

PriorityQueue Attempt #2

<u>PriorityQueue</u>
~~(A, 0)~~
(B,6)
~~(D,3)~~
(F,5)
(E,40)
(B,5)
(C,13)



We know now we could get to B in at most 5, and C in at most 13.

PriorityQueue Attempt #2

UNEXPLORED

START

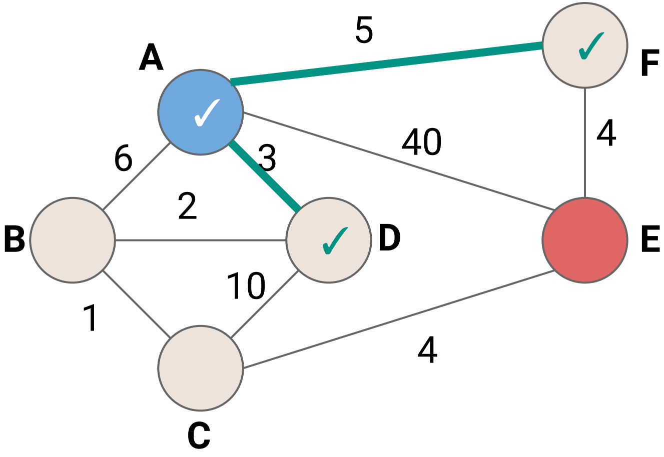
TARGET

VISITED

UNEXPLORED

<u>PriorityQueue</u>

~~(A, 0)~~
(B,6)
~~(D,3)~~
~~(F,5)~~
(E,40)
(B,5)
(C,13)
(E,9)



PriorityQueue Attempt #2

UNEXPLORED

START

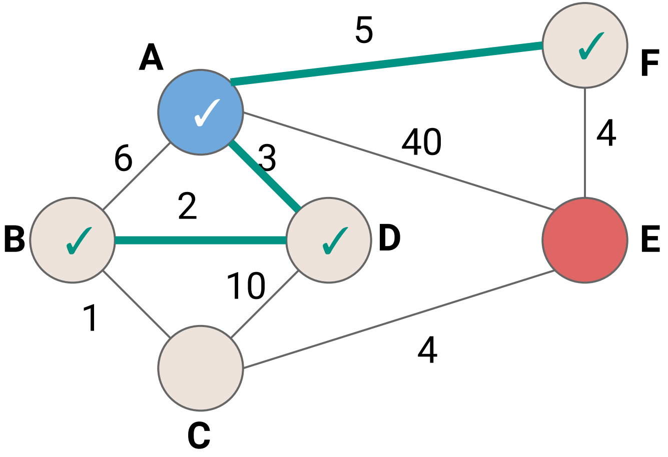
TARGET

VISITED

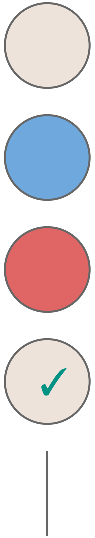
UNEXPLORED

PriorityQueue

~~(A, 0)~~
(B,6)
~~(D,3)~~
~~(F,5)~~
(E,40)
~~(B,5)~~
(C,13)
(E,9)
(C,6)



PriorityQueue Attempt #2



UNEXPLORED

START

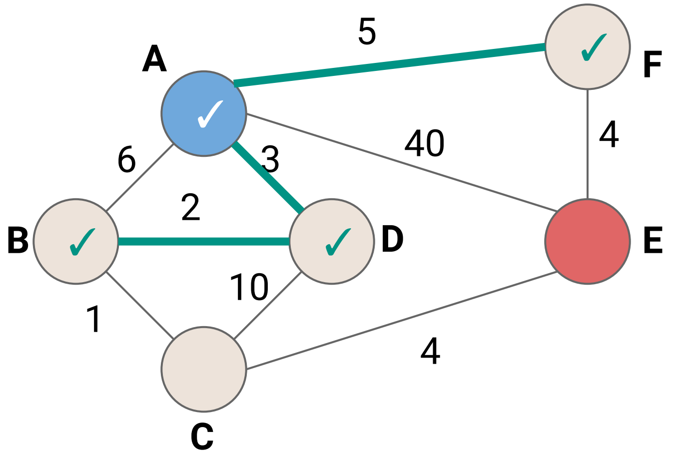
TARGET

VISITED

UNEXPLORED

<u>PriorityQueue</u>

~~(A, 0)~~
~~(B,6)~~
~~(D,3)~~
~~(F,5)~~
(E,40)
~~(B,5)~~
(C,13)
(E,9)
(C,6)



We've already visited B so we can ignore this

PriorityQueue Attempt #2

UNEXPLORED

START

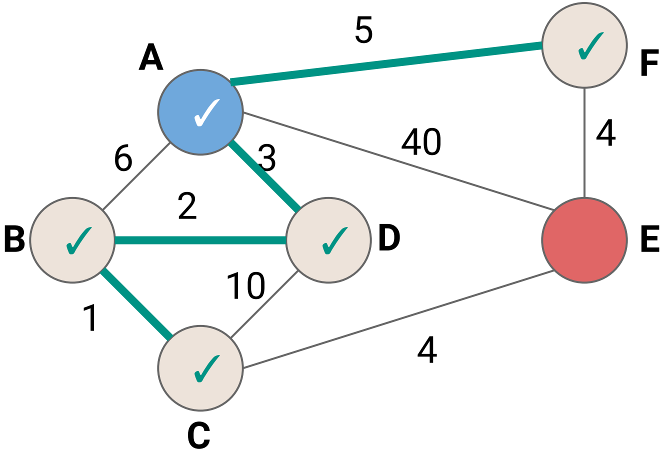
TARGET

VISITED

UNEXPLORED

PriorityQueue

~~(A, 0)~~
~~(B,6)~~
~~(D,3)~~
~~(F,5)~~
(E,40)
~~(B,5)~~
(C,13)
(E,9)
~~(C,6)~~
(E,10)



PriorityQueue Attempt #2

UNEXPLORED

START

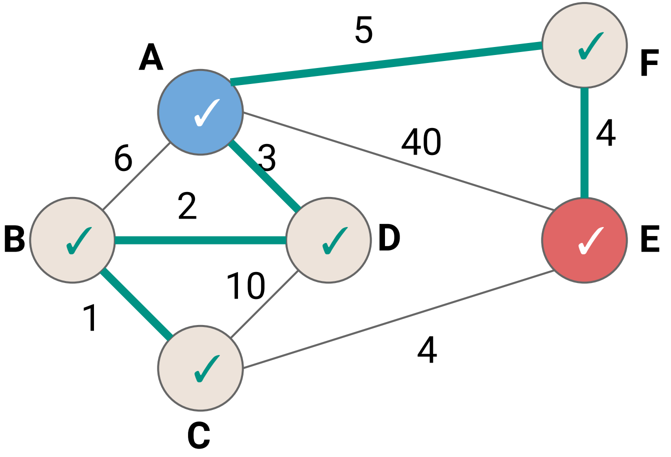
TARGET

VISITED

UNEXPLORED

<u>PriorityQueue</u>

~~(A, 0)~~
~~(B,6)~~
~~(D,3)~~
~~(F,5)~~
(E,40)
~~(B,5)~~
(C,13)
~~(E,9)~~
~~(C,6)~~
(E,10)



We've dequeued E, so we've found the shortest possible path to get there! (Anything else still left in the PriorityQueue is a longer path)

Djikstra's Algorithm

```
def Djikstras(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  while (!work.isEmpty) {
    (v, dist) = work.dequeue()
    if(v.label == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED)
      for(e <- v.incident) {
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, dist + ω(e)))
        }
      }
    }
  }
}
```

Djikstra's Algorithm

```
def Djikstras(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  while (!work.isEmpty) {
    (v, dist) = work.dequeue()
    if(v.label == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED)
      for(e <- v.incident) {
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, dist + ω(e)))
        }
      }
    }
  }
}
```

Create a PriorityQueue ordered by distance from start, and insert (start,0)

Djkstra's Algorithm

```
def Djikstras(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  while (!work.isEmpty) {
    (v, dist) = work.dequeue()
    if(v.label == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED)
      for(e <- v.incident) {
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, dist + ω(e)))
        }
      }
    }
  }
}
```

As long as there is still work, dequeue the next vertex (which will be the closest vertex since we are using a PQ)

Djikstra's Algorithm

```
def Djikstras(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  while (!work.isEmpty) {
    (v, dist) = work.dequeue()
    if(v.label == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED)
      for(e <- v.incident) {
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, dist + ω(e)))
        }
      }
    }
  }
}
```

If that vertex is UNEXPLORED, then we've just found the shortest path to that vertex

Add all of the neighboring UNEXPLORED vertices to the PQ, with their v's distance, plus the distance of the next edge

Djikstra's Algorithm

```
def Djikstras(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  while (!work.isEmpty) {
    (v, dist) = work.dequeue()
    if(v.label == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED)
      for(e <- v.incident) {
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, dist + ω(e)))
        }
      }
    }
  }
}
```

What is the complexity?

Djikstra's Algorithm

```
def Djikstras(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  while (!work.isEmpty) {
    (v, dist) = work.dequeue()
    if(v.label == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED)
      for(e <- v.incident) {
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, dist + ω(e)))
        }
      }
    }
  }
}
```

Each dequeue is O(log(|V|)

Each vertex is visited once

What is the complexity?

Djikstra's Algorithm

```
def Djikstras(graph: Graph[…], start: Graph[…]#Vertex) {
  val work = mutable.PriorityQueue[Graph[...]#Vertex, Int].empty(
      Ordering.by[(Graph[...]#Vertex, Int), Int](_._2))
  work.enqueue((start,0))
  while (!work.isEmpty) {
    (v, dist) = work.dequeue()
    if(v.label == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED)
      for(e <- v.incident) {
        if(w.label == VertexLabel.UNEXPLORED){
          work.enqueue((w, dist + ω(e)))
        }
      }
    }
  }
}
```

Each dequeue is O(log(|V|)

Each vertex is visited once

What is the complexity? O(|V| log |V|) (roughly…)

Djikstra's Algorithm

- Many tweaks can be made
  - What if instead of enqueuing a vertex we've already seen we just update the existing value in our heap?
  - How can we track the actual path?
    - Store it in the heap as well
    - Build a map of reverse lookups