CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 19

BFS, Shortest Path, Priority Queues
object VertexLabel extends Enumeration
    { val UNEXPLORED, VISITED = Value }

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

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

Create a work list of "nodes to visit", and add our start node
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)
        }
      } else {
        e.setLabel(EdgeLabel.CROSS)
      }
    }
  }
}
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)
                }
            } else {
                e.setLabel(EdgeLabel.CROSS)
            }
        }
    }
}

If we find a new node, enqueue it to be explored, and label appropriately
def BFSOne(graph: Graph[...], start: Graph[...].#Vertex) {
  val work = mutable.Queue[Graph[...].#Vertex]()
  work.enqueue(start)
  start.setLabel(VertexLabel.VISITED)
  while (!work.isEmpty) {
    v = work.dequeue()
    for (e <- v.incident) {
      if (e.label == EdgeLabel.UNEXPLORED) {
        val w = e.getOpposite(v)
        if (w.label == VertexLabel.UNEXPLORED) {
          work.enqueue(w)
          w.setLabel(VertexLabel.VISITED)
          e.setLabel(EdgeLabel.SPANNING)
        } else {
          e.setLabel(EdgeLabel.CROSS)
        }
      }
    }
  }
}

...if we have already visited it, then don't add it to be explored
BFS - Complexity

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

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

Each vertex is enqueued exactly once
BFS - Complexity

```python
def BFSOne(graph: Graph[...,], start: Graph[...]#Vertex) {
    /* O(1) */
    while (!work.isEmpty) {
        /* O(1) */
        /* O(deg(v)) times */ {
            /* O(1) */{
                /* O(1) */
                /* O(1) */{
                /* O(1) */{
                    /* O(1) */{
                    /* O(1) */{
                    /* O(1) */{
                } else {
                    /* O(1) */{
                    /* O(1) */{
                    /* O(1) */{
                }
        }
    }
}
}
}
```
What is the sum over all iterations in **BFS**One?

\[
\sum_{v \in V} O(\text{deg}(v))
\]

\[
= O\left(\sum_{v \in V} \text{deg}(v)\right)
\]

\[
= O(2|E|)
\]

\[
= O(|E|)
\]
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 \textbf{UNVISITED} \quad \O(|V|)
2. Mark the edges \textbf{UNVISITED}
Breadth-First Search Complexity

In summary...

1. Mark the vertices **UNVISITED** $O(|V|)$
2. Mark the edges **UNVISITED** $O(|E|)$
Breadth-First Search Complexity

In summary...

1. Mark the vertices **UNVISITED** \( O(|V|) \)
2. Mark the edges **UNVISITED** \( O(|E|) \)
3. Add each vertex to the work queue
Breadth-First Search Complexity

In summary...

1. Mark the vertices UNVISITED \(O(|V|)\)
2. Mark the edges UNVISITED \(O(|E|)\)
3. Add each vertex to the work queue \(O(|V|)\)
Breadth-First Search Complexity

In summary...

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

In summary...

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

In summary...

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

\[ O(|V| + |E|) \]
Queues vs Stacks

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

Thought Experiment: How is the use of a Queue related to traversal order? What if we used a Stack instead?
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)
                }
            }
        }
    }
}
def DFSOneNoRecursion(graph: Graph[...], start: Graph[...].#Vertex) {
    val work = mutable.Stack[Graph[...].#Vertex]()
    work.push(start)
    start.setLabel(VertexLabel.VISITED)
    while (!work.isEmpty) {
        v = work.pop()
        for (e <- v.incident) {
            if (e.label == EdgeLabel.UNEXPLORED) {
                val w = e.getOpposite(v)
                if (w.label == VertexLabel.UNEXPLORED) {
                    work.push(w)
                    w.setLabel(VertexLabel.VISITED)
                    e.setLabel(EdgeLabel.SPANNING)
                } else {
                    e.setLabel(EdgeLabel.BACK)
                }
            } else {
                e.setLabel(EdgeLabel.BACK)
            }
        }
    }
}
Observation: The recursive version of DFS was using a Stack all along...the call stack!
## DFS Traversal vs BFS Traversal

<table>
<thead>
<tr>
<th>Application</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning Trees</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Connected Components</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Paths/Connectivity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cycles</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shortest Paths</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Articulation Points</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Shortest Paths

Home

Grandmama's House
Shortest Paths

Over the river

Home

Grandmama's House
Shortest Paths

Over the river

Through the woods

Home

Grandmama's House
Shortest Paths

Home — Over the river — Through the woods — Across the treacherous mountains — Grandmama's House
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 $\text{start}$ in $V$
- An end vertex $\text{end}$ in $V$
Given:
- A weighted graph $G = (V,E,\omega)$
- A start vertex $\text{start}$ in $V$
- An end vertex $\text{end}$ in $V$

Goal:
- Produce a simple path $P$ from $\text{start}$ to $\text{end}$...
- ...that minimizes the sum of edge weights in the $P$
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

```scala
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

```scala
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

```scala
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

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

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

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

UNEXPLORED

START

TARGET

VISITED

UNEXPLORED

SPANNING

CROSS
Detailed Example

Call Stack
- BFS(G)

Work Queue
- UNEXPLORED
- START
- TARGET
- VISITED
- UNEXPLORED
- SPANNING
- CROSS
Detailed Example

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

Work Queue

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

**Graph:**
- Nodes: A, B, C, D, E
- Edges: (A-B, 2), (A-C, 10), (B-D, 2), (C-D, 10), (D-E, 4)

**Colors:**
- UNEXPLORED
- START
- TARGET
- VISITED
- SPANNING
- CROSS
Detailed Example

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

Work Queue
A
B
D
E

UNEXPLORED
START
TARGET
VISITED
UNEXPLORED
SPANNING
CROSS

Found a path from A to E!
How do we find this path?
Thought Experiment: How can we find the shortest path when not all edges are created equal?
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

UNEXPLORED

START

TARGET

VISITED

UNEXPLORED

SPANNING

CROSS
Attempt #1: Explore Smallest Edge

![Graph Diagram]

- UNEXPLORED
- START
- TARGET
- VISITED
- UNEXPLORED
- SPANNING
- CROSS
Attempt #1: Explore Smallest Edge

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

The diagram shows a network with nodes A, B, C, D, and E. The edges and their weights are:
- A to B: 6
- A to D: 3
- B to C: 2
- B to D: 40
- B to E: 1
- C to D: 10
- C to E: 4
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

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

Diagram:

- A
- B
- C
- D
- E
- F

Edges with weights:
- A to B: 6
- B to D: 2
- D to C: 10
- D to E: 40
- F to E: 4
- A to C: 5

Selected nodes:
- A, B, C, D, F

Order:
- A
- B
- D
- E
- F
- C
**Desired Exploration Order**

- **START**: A
- **TARGET**: B
- **VISITED**: A, B, D, C, E
- **UNEXPLORED**: F
- **SPANNING**: C to D and B to E
- **CROSS**: B to A and C to F
Desired Exploration Order - Closest Vertex

Smallest edge? No

Closest Vertex!
Desired Exploration Order

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

The desired exploration order is shown in the graph with nodes A, B, C, D, E, and F. The edges and weights (1, 2, 3, 4, 5, 6, 10, 40) are indicated, and the nodes are marked as visited or unexplored.
 Desired Exploration Order

UNEXPLORED

START

TARGET

VISITED

UNEXPLORED

SPANNING

CROSS
Desired Exploration Order

UNEXPLORED
START
TARGET
VISITED
UNEXPLORED
SPANNING
CROSS

Path Found!
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 + \(\omega(e)\)))
                    w.setLabel(VertexLabel.VISITED)
                    e.setLabel(EdgeLabel.SPANNING)
                } else {
                    e.setLabel(EdgeLabel.CROSS)
                }
            } else {
                e.setLabel(EdgeLabel.CROSS)
            }
        }
    }
}

We want to be able to dequeue vertices in ascending order of distance...but how?
A New ADT... PriorityQueue

PriorityQueue[A:Ordering]

enqueue(v: A)
Insert value $v$ into the priority queue

head: A
Peek at the highest priority value in the priority queue

decqueue: A
Remove the highest priority value in the priority queue
A New ADT... PriorityQueue

PriorityQueue[A: Ordering]

enqueue(v: A)
   Insert value v into the priority queue

head: A
   Peek at the highest priority value in the priority queue

decqueue: A
   Remove the highest priority value in the priority queue