CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 16
Introduction to Graphs
Textbook Ch. 15.3
Announcements

- Testing phase of PA2 due Monday 10/10
  - Autograder has been recently updated to address some issues
Recap

Mazes!
Formalizing Maze-Solving

Inputs:

- The map: an $n \times m$ grid of squares which are either filled or empty
- The O is at position start
- The X is at position dest

Goal: Compute $\text{steps}(\text{start}, \text{dest})$, the minimum number of steps from start to end.

How do we define the steps function?
Mazes
Mazes: Now with...Stacks!

steps(pos, dest, visited):
    if pos == dest then return visited.copy()
    elif pos ∈ visited then return no_path
    elif is_filled(pos) then return no_path
    else
        visited.push(pos)
        bestPath = 1 + min of all 4 steps
        visited.pop()
    return bestPath
Mazes: Now with...Stacks!

steps(pos, dest, visited):
    if pos == dest then return visited.copy()
    elif pos ∈ visited then return no_path
    elif is_filled(pos) then return no_path
    else
        visited.push(pos)
        bestPath = 1 + min of all 4 steps
        visited.pop()
        return bestPath

We can use stacks to track where we have been!
Queues?

**Thought Experiment:** Can we do something similar with queues?
Thought Experiment: Can we do something similar with queues?

Hold that thought!
Let's Talk About Graphs

A graph is a pair \((V,E)\) where:

- \(V\) is a set of vertices
- \(E\) is a set of vertex pairs called edges
- Edges and vertices may also store data (labels)
Example: A social network
(nodes store users, pictures, tweets, etc)
(edges store interactions)
**Example:** A computer network

(edges store ping, nodes store addresses)
Edge Types

Directed Edge (asymmetric relationship)
- Ordered pair of vertices \((u, v)\)
- origin \((u)\) → destination \((v)\)

Undirected Edge (symmetric relationship)
- Unordered pair of vertices \((u, v)\)

- Transmit bandwidth: 100 mb/s
- Round-trip latency: 7 ms
Edge Types

Directed Edge (asymmetric relationship)
- Ordered pair of vertices \((u, v)\)
- origin \((u)\) → destination \((v)\)

Undirected Edge (symmetric relationship)
- Unordered pair of vertices \((u,v)\)

Directed Graph: All edges are directed

Undirected Graph: All edges are undirected
Applications

- Transportation (flight/road/rail routing)
- Protein/Protein Interactions
- Computer Networks (ie the internet)
- Social Networks
- Dependency Tracking (ie make)
- Taxonomies
Terminology

Endpoints of an edge
$U, V$ are endpoints of a

Adjacent Vertices
$U, V$ are adjacent

Degree of a vertex
$X$ has degree 5
Terminology

Edges indecent on a vertex
$a, b, d$ are incident on $V$

Parallel Edges
$h, i$ are parallel

Self-Loop
$j$ is a self-loop

Simple Graph
A graph without parallel edges or self-loops
**Terminology**

**Path**
A sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

**Simple Path**
A path such that all of its vertices and edges are distinct
**Terminology**

**Path**
A sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

**Simple Path**
A path such that all of its vertices and edges are distinct

\[ U, c, W, e, X, g, Y, f, W, d, V \] is not simple

\[ V, b, X, h, Z \] is simple
**Terminology**

**Cycle**
A path that begins and ends with the same vertex. Must contain at least one edge.

**Simple Cycle**
A cycle such that all of its vertices and edges are distinct.
**Terminology**

**Cycle**
A path that begins and ends with the same vertex. Must contain at least one edge.

**Simple Cycle**
A cycle such that all of its vertices and edges are distinct.

$V, b, X, g, Y, f, W, c, U, a, V$ is a simple cycle.

$U, c, W, e, X, g, Y, f, W, d, V, a, U$ is a cycle that is not simple.
Notation

\( n \) The number of vertices

\( m \) The number of edges

\( \text{deg}(v) \) The degree of vertex \( v \)
Graph Properties

$$\sum_v \text{deg}(v) = 2m$$
Graph Properties

Proof: Each edge is counted twice
In a directed graph with no self-loops and no parallel edges:

\[ m \leq n (n - 1) \]
In a directed graph with no self-loops and no parallel edges:

\[ m \leq n \cdot (n - 1) \]

**No parallel edges:** each pair is connected at most once

**No self-loops:** pick each vertex only once
Graph Properties

In a directed graph with no self-loops and no parallel edges:

\[ m \leq n \ (n \ - \ 1) \]

No parallel edges: each pair is connected at most once

No self-loops: pick each vertex only once

\[ n \text{ choices for the first vertex; } (n \ - \ 1) \text{ choices for the second vertex.} \]

Therefore even if there was one edge between every possible pair, we still have at most \[ n(n \ - \ 1) \text{ edges.} \]
A (Directed Graph) ADT

Two type parameters (Graph[V, E])
  V: The vertex label type
  E: The edge label type

Vertices
  ...are elements (like Linked List Nodes)
  ...store a value of type V

Edges
  ...are also elements
  ...store a value of type E
trait Graph[V, E] {
    def vertices: Iterator[Vertex]
    def edges: Iterator[Edge]
    def addVertex(label: V): Vertex
    def addEdge(orig: Vertex, dest: Vertex, label: E): Edge
    def removeVertex(vertex: Vertex): Unit
    def removeEdge(edge: Edge): Unit
}
A (Directed) Graph ADT

trait Vertex[V, E] {  
def outEdges: Seq[Edge]  
def inEdges: Seq[Edge]  
def incidentEdges: Iterator[Edge] = outEdges ++ inEdges  
def edgeTo(v: Vertex): Boolean  
def label: V
}

trait Edge[V, E] {  
def origin: Vertex  
def destination: Vertex  
def label: E
}
Attempt 1: Edge List

Data Model:

A List of Edges
(ArrayBuffer)

A List of Vertices
(ArrayBuffer)
class DirectedGraphV1[V, E] extends Graph[V, E] {
    val vertices = mutable.Buffer[Vertex]()
    val edges = mutable.Buffer[Edge]()

    /* ... */
}
def addVertex(label: V): Vertex =
    vertices.append(new Vertex(label))

What's the complexity?
 Attempt 1: Edge List

```python
def addVertex(label: V): Vertex =
    vertices.append(new Vertex(label))
```

What's the complexity?

```python
def addEdge(orig: Vertex, dest: Vertex, label: E): Edge =
    edges.append(new Edge(orig, dest, label))
```

What's the complexity?
Attempt 1: Edge List

def addVertex(label: V): Vertex = 
    vertices.append(new Vertex(label))

What's the complexity? Amortized $O(1)$

def addEdge(orig: Vertex, dest: Vertex, label: E): Edge = 
    edges.append(new Edge(orig, dest, label))

What's the complexity? Amortized $O(1)$
def removeEdge(edge: Edge): Unit =
    edges.subtractOne(edge)

What's the complexity?
def removeEdge(edge: Edge): Unit =
    edges.subtractOne(edge)

What's the complexity? $O(n)$
Attempt 2: Linked Edge List

Data Model:

A List of Edges
(DoublyLinkedList)

A List of Vertices
(DoubleLinkedList)
class DoublyLinkedList[T] extends Seq[T] {
    def append(element: T): Node =
        /* O(1) with tail pointer */

    def remove(node: Node): Unit =
        /* O(1) */

    def iterator: Iterator[T]: Unit =
        /* O(1) + O(1) per call to next */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]()
    class Vertex(label: V) = {
        var node: DoublyLinkedList[Vertex].Node = null
        /* ... */
    }
    def addVertex(label: V): Vertex = {
        val vertex = new Vertex(label)
        val node = vertices.append(vertex)
        vertex.node = node
        return vertex
    }
    /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val vertices = DoublyLinkedList[Vertex]()
  class Vertex(label: V) = {
    var node: DoublyLinkedList[Vertex].Node = null
    /* ... */
  }
  def addVertex(label: V): Vertex = {
    val vertex = new Vertex(label)
    val node = vertices.append(vertex)
    vertex.node = node
    return vertex
  }
  /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]().
    class Vertex(label: V) = {
        var node: DoublyLinkedLIst[Vertex].Node = null
    
    
    }
    def addVertex(label: V): Vertex = {
        val vertex = new Vertex(label)
        val node = vertices.append(vertex)
        vertex.node = node
        return vertex
    }
    /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val vertices = DoublyLinkedList[Vertex]()
  class Vertex(label: V) = {
    var node: DoublyLinkedList[Vertex].Node = null
    /* ... */
  }
  def addVertex(label: V): Vertex = {
    val vertex = new Vertex(label)
    val node = vertices.append(vertex)
    vertex.node = node
    return vertex
  }
  /* ... */
}

Add our vertex to the linked list, and store a reference to the list node

What is the complexity? $\Theta(1)$
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val edges = DoublyLinkedList[Edge]()
    class Edge(orig: Vertex, dest: Vertex, label: E) = {
        var node: DoublyLinkedList[Edge].Node = null
        /* ... */
    }
    def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
        val edge = new Edge(orig, dest, label)
        val node = edges.append(vertex)
        edge.node = node
        return edge
    }
    /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val edges = DoublyLinkedList[Edge]()
  class Edge(orig: Vertex, dest: Vertex, label: E) = {
    var node: DoublyLinkedList[Edge].Node = null
    /* ... */
  }
  def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
    val edge = new Edge(orig, dest, label)
    val node = edges.append(vertex)
    edge.node = node
    return edge
  }
  /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val edges = DoublyLinkedList[Edge]()
  class Edge(orig: Vertex, dest: Vertex, label: E) = {
    var node: DoublyLinkedList[Edge].Node = null
    /* ... */
  }
  def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
    val edge = new Edge(orig, dest, label)
    val node = edges.append(vertex)
    edge.node = node
    return edge
  }
  /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val edges = DoublyLinkedList[Edge]()
    class Edge(orig: Vertex, dest: Vertex, label: E) = {
        var node: DoublyLinkedList[Edge].Node = null
/* ... */
    }
    def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
        val edge = new Edge(orig, dest, label)
        val node = edges.append(vertex)
        edge.node = node
        return edge
    }
/* ... */
}

What is the complexity? \( \Theta(1) \)
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val edges = DoublyLinkedList[Edge]()

    def removeEdge(edge: Edge): Unit = {
        edges.remove(edge.node)
    }

    /* ... */
}

class DirectedGraphV2[V, E] extends Graph[V, E] {
  val edges = DoublyLinkedList[Edge]()

  def removeEdge(edge: Edge): Unit = {
    edges.remove(edge.node)
    /* ... */
  }
  /* ... */
}

Remove the edge (by reference) from the linked list
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val edges = DoublyLinkedList[Edge]()

  def removeEdge(edge: Edge): Unit = {
    edges.remove(edge.node)
  }

  /* ... */
}

What is the complexity?

Remove the edge (by reference) from the linked list.
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val edges = DoublyLinkedList[Edge]()

    def removeEdge(edge: Edge): Unit = {
        edges.remove(edge.node)
    }

    /* ... */
}

What is the complexity? \( \Theta(1) \)

Remove the edge (by reference) from the linked list.
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val vertices = DoublyLinkedList[Vertex]()

  def removeVertex(vertex: Vertex): Unit = {
    vertices.remove(vertex.node)
  }

  /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]()

    def removeVertex(vertex: Vertex): Unit = {
        vertices.remove(vertex.node)
    }

    /* ... */
}

What if there's an edge to/from the vertex?
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList<Vertex]()

    def removeVertex(vertex: Vertex): Unit = {
        vertices.remove(vertex.node)
        for (edge <- vertex.incidentEdges){
            removeEdge(edge)
        }
    }

    /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]()

    def removeVertex(vertex: Vertex): Unit = {
        vertices.remove(vertex.node)
        for (edge <- vertex.incidentEdges) {
            removeEdge(edge)
        }
    }

    /* ... */
}
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]()

    def removeVertex(vertex: Vertex): Unit = {
        vertices.remove(vertex.node)
        for (edge <- vertex.incidentEdges) {
            removeEdge(edge)
        }
    }

    /* ... */
}

Remove the vertex (by reference) from the linked list, and then all remove all incident edges (by reference)

What is the complexity?
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val vertices = DoublyLinkedList[Vertex]()

  def removeVertex(vertex: Vertex): Unit = {
    vertices.remove(vertex.node)
    for (edge <- vertex.incidentEdges)
      removeEdge(edge)
  }
}

/* ... */

What is the complexity? $O(1) + O(T_{incidentEdges}(n,m))$
**Attempt 2: Linked Edge List**

```
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val vertices = DoublyLinkedList[Vertex]()

  def removeVertex(vertex: Vertex): Unit = {
    vertices.remove(vertex.node)
    for (edge <- vertex.incidentEdges) {
      removeEdge(edge)
    }
  }

  /* ... */
}
```

What is the complexity? $O(1) + O(T_{incidentEdges}(n,m))$

How do we figure out what edges are incident?
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]()
    val edges = DoublyLinkedList[Edge]()
    class Vertex(label: V) = {
        /* ... */
        def outEdges =
            edges.filter { _.orig = this }

        def inEdges =
            edges.filter { _.dest = this }
    }
    /* ... */
}"
class DirectedGraphV2[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]()
    val edges = DoublyLinkedList[Edge]()
    class Vertex(label: V) = {
        /* ... */
        def outEdges =
            edges.filter { _.orig = this }

        def inEdges =
            edges.filter { _.dest = this }
    }
    /* ... */
    What is the complexity?
class DirectedGraphV2[V, E] extends Graph[V, E] {
  val vertices = DoublyLinkedList[Vertex]()
  val edges = DoublyLinkedList[Edge]()
  class Vertex(label: V) = {
    /* ... */
    def outEdges =
      edges.filter { _.orig = this }

    def inEdges =
      edges.filter { _.dest = this }
  }
  /* ... */
}

What is the complexity? $O(m) = O(n^2)$
Edge List Summary

- addEdge, addVertex:
- removeEdge:
- removeVertex:
- vertex.incidentEdges:
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge:
- removeVertex:
- vertex.incidentEdges:
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex:
- vertex.incidentEdges:
Edge List Summary

- addEdge, addVertex: \(O(1)\)
- removeEdge: \(O(1)\)
- removeVertex: \(O(m)\)
- vertex.incidentEdges:
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(m)$
- vertex.incidentEdges: $O(m)$
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(m)$
- vertex.incidentEdges: $O(m)$
- vertex.edgeTo: $O(m)$
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(m)$
- vertex.incidentEdges: $O(m)$
- vertex.edgeTo: $O(m)$
- Space Used: $O(n) + O(m)$
Edge List Summary

**Vertex**

```
U  v_1
V  v_2
W  v_3
```

**Edge**

```
v_1 v_2 a
v_2 v_3 b
v_3 v_1 c
```

**LinkedList[Vertex]**

**Vertex**

```
U  v_1
V  v_2
W  v_3
```

**Edge**

```
v_1 v_2 a
v_2 v_3 b
v_3 v_1 c
```

**LinkedList[Edge]**
How can we improve?
How can we improve?

**Idea:** Store the in/out edges for each vertex!
class DirectedGraphV3[V, E] extends Graph[V, E] {
    val vertices = DoublyLinkedList[Vertex]()

    class Vertex(label: V) = {
        var node: DoublyLinkedList[Vertex].Node = null
        val inEdges = DoublyLinkedList[Edge]()
        val outEdges = DoublyLinkedList[Edge]()
        /* ... */
    }
    /* ... */
}
class DirectedGraphV3[V, E] extends Graph[V, E] {
  val vertices = DoublyLinkedList[Vertex]()

  class Vertex(label: V) = {
    var node: DoublyLinkedList[Vertex].Node = null
    val inEdges = DoublyLinkedList[Edge]()
    val outEdges = DoublyLinkedList[Edge]()
    /* ... */
  }
  /* ... */
}
Store linked lists of incident edges to this vertex
class DirectedGraphV3[V, E] extends Graph[V, E] {
  /* ... */
  def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
    val edge = new Edge(orig, dest, label)
    val node = edges.append(vertex)
    edge.node = node
    orig.outEdges.append(edge)
    dest.inEdges.append(edge)
    return edge
  }
  /* ... */
}
class DirectedGraphV3[V, E] extends Graph[V, E] {
    /* ... */
    def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
        val edge = new Edge(orig, dest, label)
        val node = edges.append(vertex)
        edge.node = node
        orig.outEdges.append(edge)
        dest.inEdges.append(edge)
        return edge
    }
    /* ... */
}
class DirectedGraphV3[V, E] extends Graph[V, E] {
    /* ... */
    def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
        val edge = new Edge(orig, dest, label)
        val node = edges.append(vertex)
        edge.node = node
        orig.outEdges.append(edge)
        dest.inEdges.append(edge)
        return edge
    }
    /* ... */
}
class DirectedGraphV3[\(V, E\)] extends Graph[\(V, E\)] {
  /* ... */
  def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
    val edge = new Edge(orig, dest, label)
    val node = edges.append(vertex)
    edge.node = node
    /* Highlighted line */
    orig.outEdges.append(edge)
    dest.inEdges.append(edge)
    return edge
  }
  /* ... */
}

What is the complexity? \(\Theta(1)\)
class DirectedGraphV3[V, E] extends Graph[V, E] {
  /* ... */
  def removeEdge(edge: Edge): Unit = {
    edges.remove(edge.node)
    edge.orig.outEdges.subtractOne(edge)
    edge.dest.inEdges.subtractOne(edge)
  }
  /* ... */
}
class DirectedGraphV3[V, E] extends Graph[V, E] {
    /* ... */
    def removeEdge(edge: Edge): Unit = {
        edges.remove(edge.node)
        edge.orig.outEdges.subtractOne(edge)
        edge.dest.inEdges.subtractOne(edge)
    }
    /* ... */
}
class DirectedGraphV3[V, E] extends Graph[V, E] { 
  /* ... */ 
  def removeEdge(edge: Edge): Unit = {
    edges.remove(edge.node)
    edge.orig.outEdges.subtractOne(edge)
    edge.dest.inEdges.subtractOne(edge)
  }
  /* ... */
}
class DirectedGraphV3[V, E] extends Graph[V, E] {
  /* ... */
  def removeEdge(edge: Edge): Unit = {
    edges.remove(edge.node)
    edge.orig.outEdges.subtractOne(edge)
    edge.dest.inEdges.subtractOne(edge)
  }
  /* ... */
}

What is the complexity? $O(\text{deg}(\text{orig})) + O(\text{deg}(\text{dest}))$

Remove the edges from our adjacency lists...?
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    class Edge(orig: Vertex, dest: Vertex, label: E) = {
        var node: DoublyLinkedList[Edge].Node = null
        var origNode: DoublyLinkedList[Edge].Node = null
        var destNode: DoublyLinkedList[Edge].Node = null
        /* ... */
    }
    /* ... */
}
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    
    class Edge(orig: Vertex, dest: Vertex, label: E) = {
        var node: DoublyLinkedList[Edge].Node = null
        var origNode: DoublyLinkedList[Edge].Node = null
        var destNode: DoublyLinkedList[Edge].Node = null
        /* ... */
    }
    /* ... */
}
Attempt 4: Adjacency List

class DirectedGraphV4[V, E] extends Graph[V, E] {
  /* ... */
  def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
    val edge = new Edge(orig, dest, label)
    val node = edges.append(vertex)
    edge.node = node
    edge.origNode = orig.outEdges.append(edge)
    edge.destNode = dest.inEdges.append(edge)
    return edge
  }
  /* ... */
}

class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
        val edge = new Edge(orig, dest, label)
        val node = edges.append(vertex)
        edge.node = node
        edge.origNode = orig.outEdges.append(edge)
        edge.destNode = dest.inEdges.append(edge)
        return edge
    }
    /* ... */
}
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
        val edge = new Edge(orig, dest, label)
        val node = edges.append(vertex)
        edge.node = node
        edge.origNode = orig.outEdges.append(edge)
        edge.destNode = dest.inEdges.append(edge)
        return edge
    }
    /* ... */
}

What is the complexity?
class DirectedGraphV4[V, E] extends Graph[V, E] {
  /* ... */
  def addEdge(orig: Vertex, dest: Vertex, label: E): Vertex = {
    val edge = new Edge(orig, dest, label)
    val node = edges.append(vertex)
    edge.node = node
    edge.origNode = orig.outEdges.append(edge)
    edge.destNode = dest.inEdges.append(edge)
    return edge
  }
  /* ... */
}

What is the complexity? Θ(1)

Save our references when adding an edge
class DirectedGraphV4[V, E] extends Graph[V, E] {
  /* ... */
  def removeEdge(edge: Edge): Unit = {
    edges.remove(edge.node)
    edge.orig.outEdges.remove(edge.origNode)
    edge.dest.inEdges.remove(edge.destNode)
  }
  /* ... */
}
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    def removeEdge(edge: Edge): Unit = {
        edges.remove(edge.node)
        edge.orig.outEdges.remove(edge.origNode)
        edge.dest.inEdges.remove(edge.destNode)
    }
    /* ... */
}
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    def removeEdge(edge: Edge): Unit = {
        edges.remove(edge.node)
        edge.orig.outEdges.remove(edge.origNode)
        edge.dest.inEdges.remove(edge.destNode)
    }
    /* ... */
}

What is the complexity?
Remove by reference!
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    def removeEdge(edge: Edge): Unit = {
        edges.remove(edge.node)
        edge.orig.outEdges.remove(edge.origNode)
        edge.dest.inEdges.remove(edge.destNode)
    }
    /* ... */
}

What is the complexity? Θ(1)
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    def removeVertex(vertex: Vertex): Unit = {
        vertices.remove(vertex.node)
        for (edge <- vertex.incidentEdges) {
            removeEdge(edge)
        }
    }
    /* ... */
}
class DirectedGraphV4[V, E] extends Graph[V, E] {
  /* ... */
  def removeVertex(vertex: Vertex): Unit = {
    vertices.remove(vertex.node)
    for(edge <- vertex.incidentEdges) {
      removeEdge(edge)
    }
  }
  /* ... */
}

What is the complexity?
class DirectedGraphV4[V, E] extends Graph[V, E] {
    /* ... */
    def removeVertex(vertex: Vertex): Unit = {
        vertices.remove(vertex.node)
        for (edge <- vertex.incidentEdges) {
            removeEdge(edge)
        }
    }
    /* ... */
}

What is the complexity? $O(\text{deg}(\text{vertex}))$
Adjacency List Summary

- `addEdge`, `addVertex`
- `removeEdge`
- `removeVertex`
- `vertex.incidentEdges`
- `vertex.edgeTo`
- `Space Used`
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge:
- removeVertex:
- vertex.incidentEdges:
- vertex.edgeTo:
- Space Used:
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex:
- vertex.incidentEdges:
- vertex.edgeTo:
- Space Used:
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(\text{deg}(\text{vertex}))$
- vertex.incidentEdges:
- vertex.edgeTo:
- Space Used:
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(\text{deg}(\text{vertex}))$
- vertex.incidentEdges: $O(\text{deg}(\text{vertex}))$
- vertex.edgeTo:
- Space Used:
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(\text{deg(vertex)})$
- vertex.incidentEdges: $O(\text{deg(vertex)})$
- vertex.edgeTo: $O(\text{deg(vertex)})$
- Space Used:
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(\text{deg}(\text{vertex}))$
- vertex.incidentEdges: $O(\text{deg}(\text{vertex}))$
- vertex.edgeTo: $O(\text{deg}(\text{vertex}))$
- Space Used: $O(n) + O(m)$
Adjacency Matrix

<table>
<thead>
<tr>
<th>Origin</th>
<th>U</th>
<th>V</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>-</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>-</td>
<td>b</td>
</tr>
<tr>
<td>W</td>
<td>c</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The adjacency matrix represents the connections in the graph: U is connected to V with edge a, V is connected to W with edge b, and U is connected to W with edge c.
Adjacency Matrix Summary

- addEdge, removeEdge:
- addVertex, removeVertex:
- vertex.incidentEdges:
- vertex.edgeTo:
- Space Used:
Adjacency Matrix Summary

- addEdge, removeEdge: $O(1)$
- addVertex, removeVertex:
- vertex.incidentEdges:
- vertex.edgeTo:
- Space Used:
Adjacency Matrix Summary

- addEdge, removeEdge: $O(1)$
- addVertex, removeVertex: $O(n^2)$
- vertex.incidentEdges:
- vertex.edgeTo:
- Space Used:
Adjacency Matrix Summary

- addEdge, removeEdge: $O(1)$
- addVertex, removeVertex: $O(n^2)$
- vertex.incidentEdges: $O(n)$
- vertex.edgeTo:
- Space Used:
Adjacency Matrix Summary

- addEdge, removeEdge: $O(1)$
- addVertex, removeVertex: $O(n^2)$
- vertex.incidentEdges: $O(n)$
- vertex.edgeTo: $O(1)$
- Space Used:
Adjacency Matrix Summary

- addEdge, removeEdge: $O(1)$
- addVertex, removeVertex: $O(n^2)$
- vertex.incidentEdges: $O(n)$
- vertex.edgeTo: $O(1)$
- Space Used: $O(n^2)$