CSE 250
Data Structures

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

Introduction to Graphs
Textbook Ch. 15.3

Recap

Mazes!

Formalizing Maze-Solving

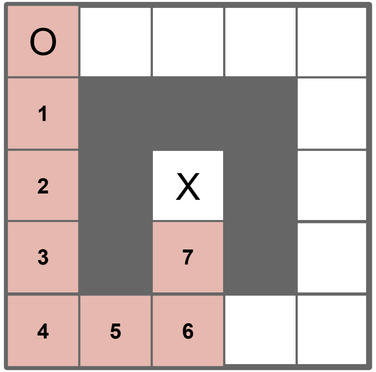
Inputs:

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

Goal: Compute `steps(start, 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?

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:

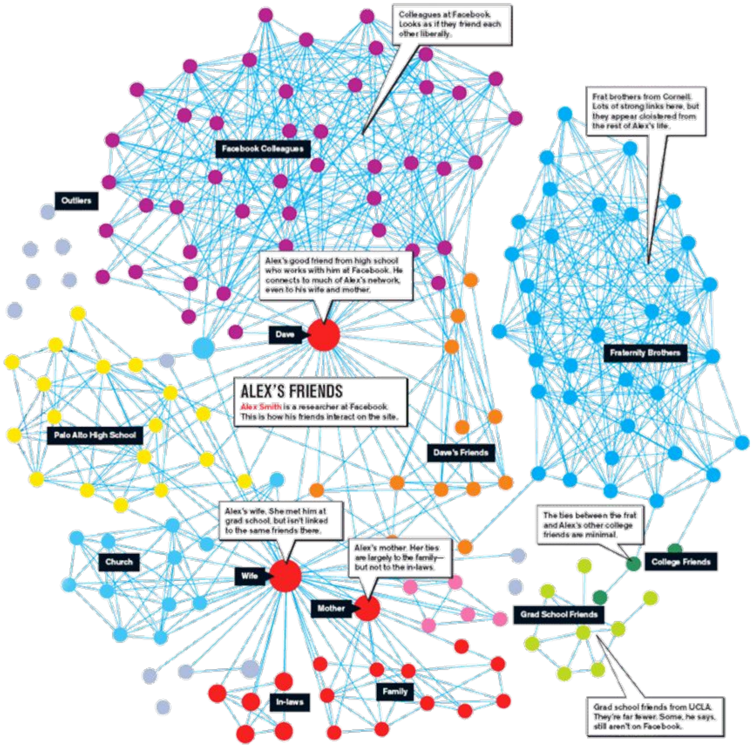
- ***V*** is a set of **vertices**
- ***E*** is a set of vertex pairs called **edges**
- Edges and vertices may also store data **(labels)**

Graphs

Example: A social network

(nodes store users, pictures, tweets, etc)

(edges store interactions)

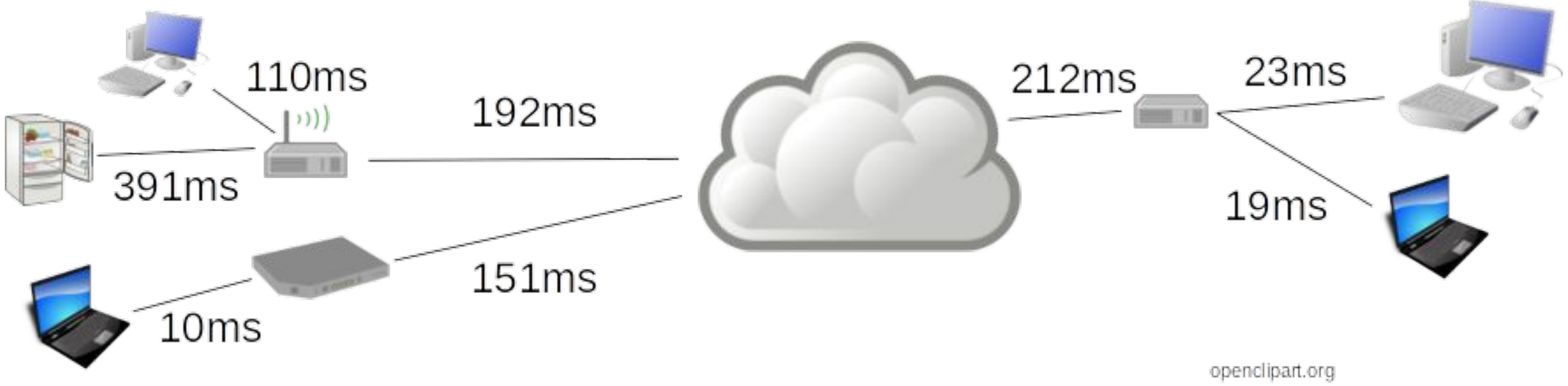


Ref :https://www.pinterest.com/pin/490470215639647556/

Graphs

Example: A computer network

(edges store ping, nodes store addresses)



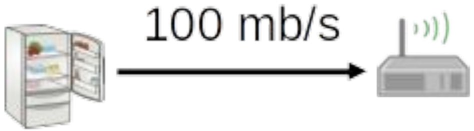
Edge Types

Directed Edge (asymmetric relationship)

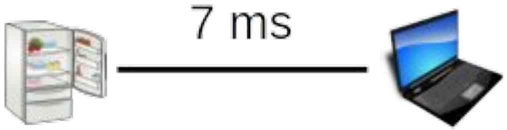
- Ordered pair of vertices (***u*, *v***)
- origin (***u***) →destination (***v***)

Undirected Edge (symmetric relationship)

- Unordered pair of vertices (***u*,*v***)



transmit bandwidth



round-trip latency

Edge Types

Directed Edge (asymmetric relationship)

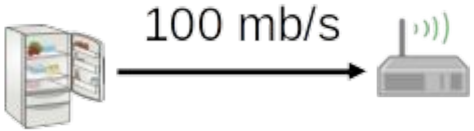
- Ordered pair of vertices (***u*, *v***)
- origin (***u***) →destination (***v***)

Undirected Edge (symmetric relationship)

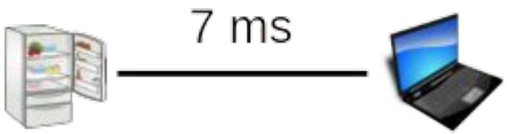
- Unordered pair of vertices (***u*,*v***)

Directed Graph: All edges are directed

Undirected Graph: All edges are undirected



transmit bandwidth



round-trip latency

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

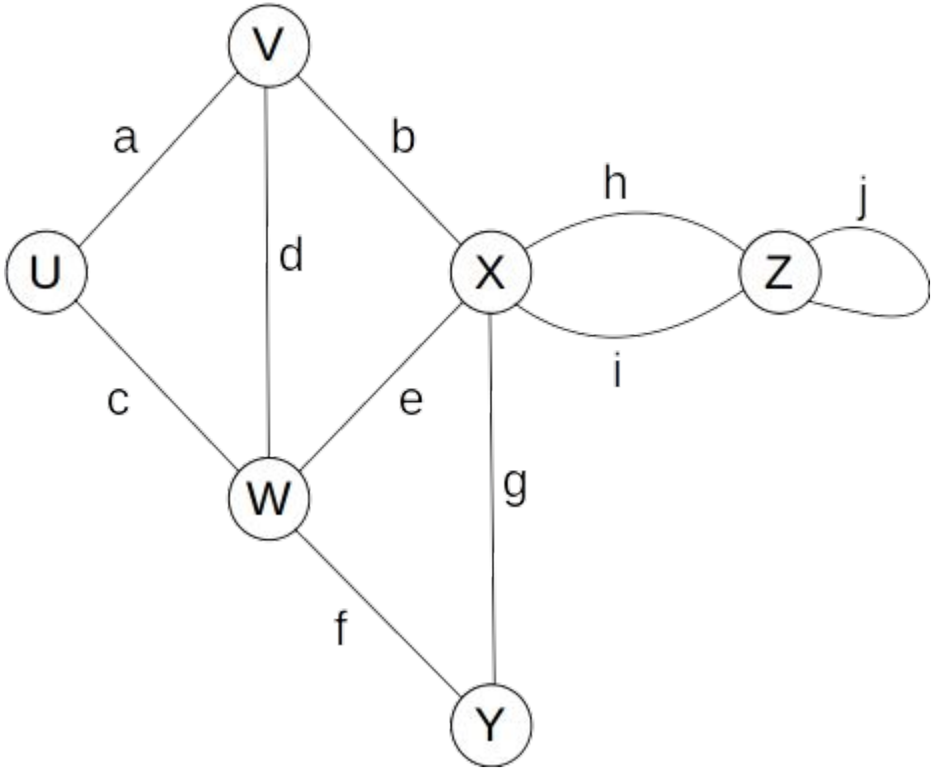
U, *V* are endpoints of *a*

Adjacent Vertices

U, *V* are adjacent

Degree of a vertex

X has degree 5



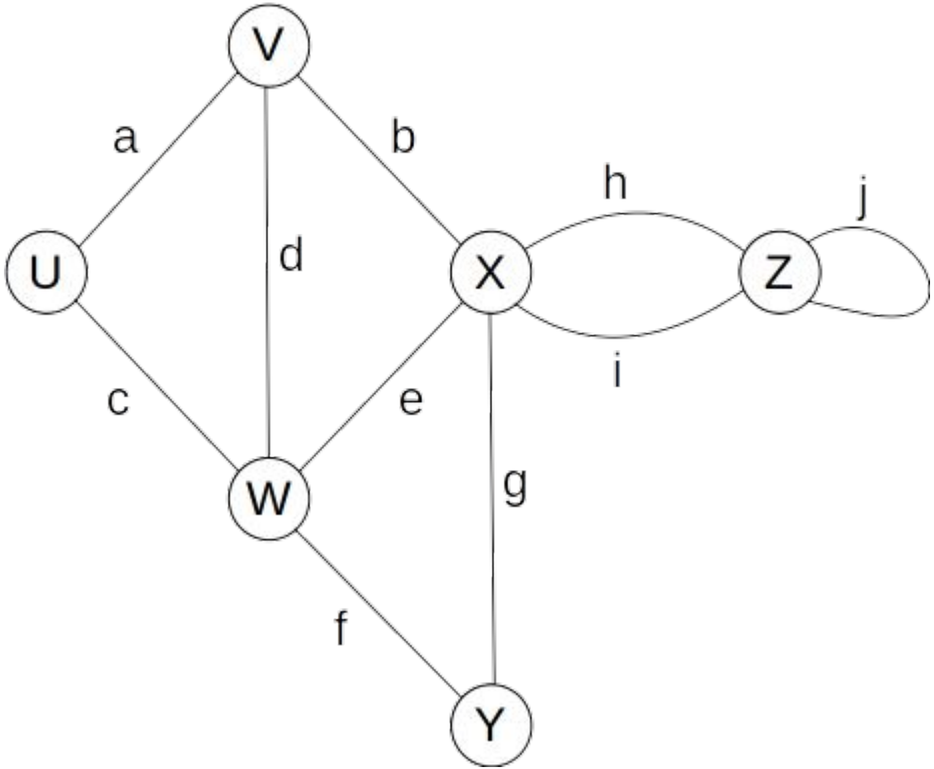
Terminology

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

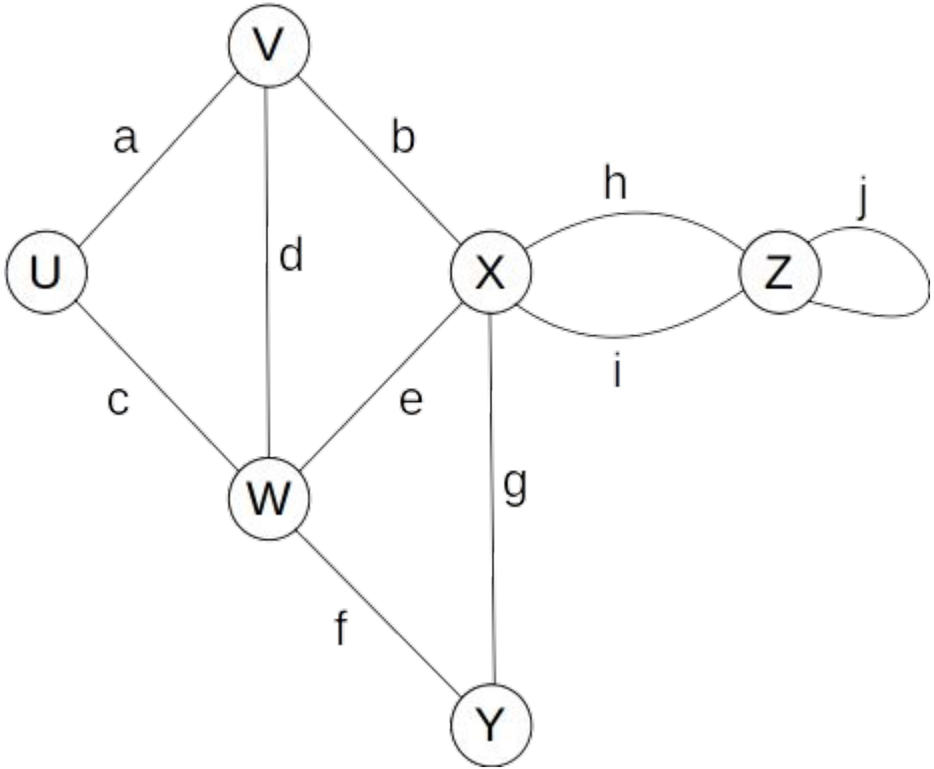


Terminology

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

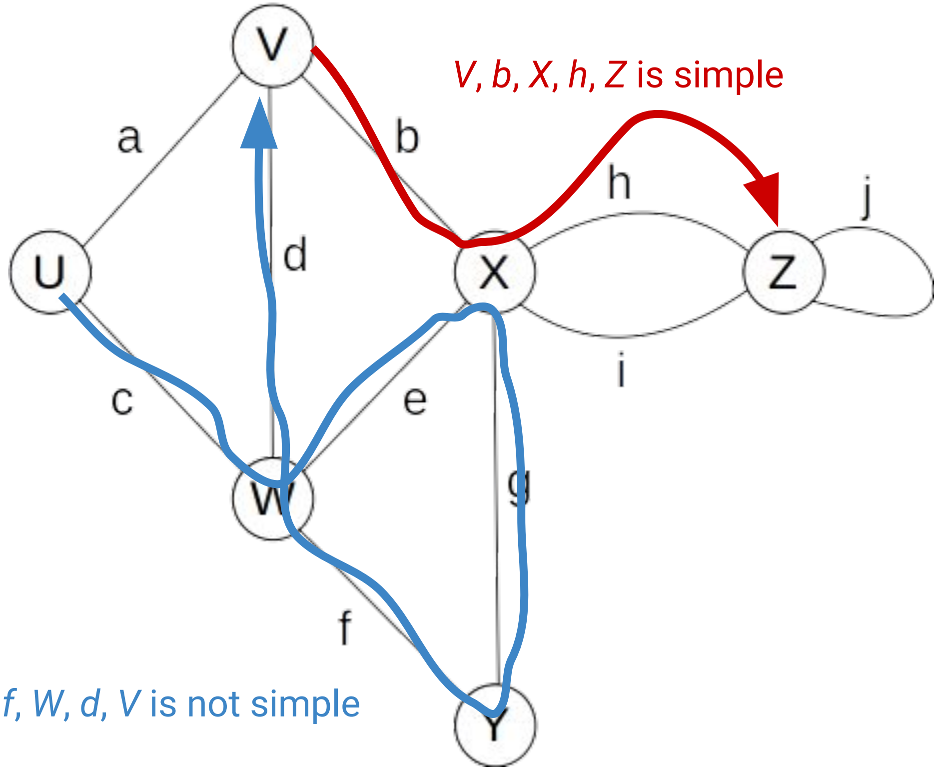


Terminology

Path
A sequence of alternating vertices and edges

- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

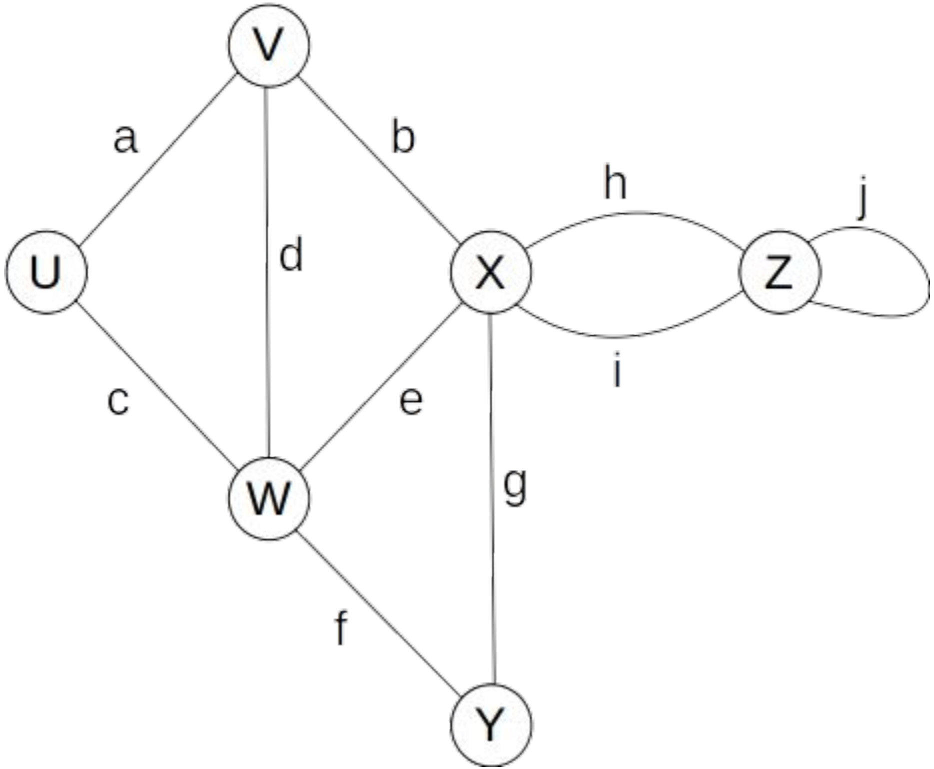
Simple Path
A path such that all of its vertices and edges are distinct



Terminology

Cycle
A path the 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



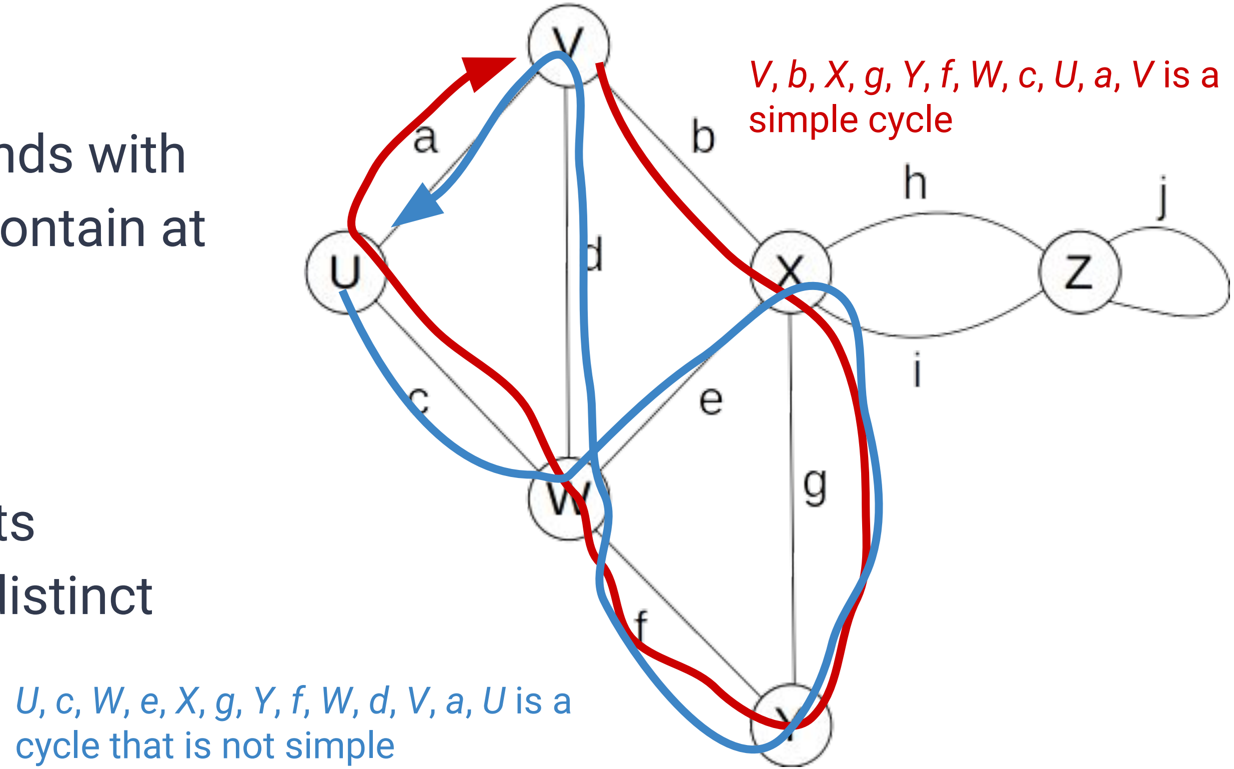
Terminology

Cycle

A path the 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



Notation

n The number of vertices

m The number of edges

deg(*v*) The degree of vertex ***v***

Graph Properties

$$\sum_{v} deg(v) = 2m$$

Graph Properties

$$\sum_{v} deg(v) = 2m$$

Proof: Each edge is counted twice

Graph Properties

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

$$m \leq n\,(n - 1)$$

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

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 choices for the first vertex; **(*n* - 1)** choices for the second vertex. Therefore even if there was one edge between every possible pair, we still have at most ***n*(*n* - 1)** 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`

A (Directed) Graph ADT

```
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)

Attempt 1: Edge List

```
class DirectedGraphV1[V, E] extends Graph[V, E] {
    val vertices = mutable.Buffer[Vertex]()
    val edges    = mutable.Buffer[Edge]()

    /* ... */
}
```

Attempt 1: Edge List

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

What's the complexity?

Attempt 1: Edge List

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

What's the complexity?

```
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 $O(1)$

Attempt 1: Edge List

```
def removeEdge(edge: Edge): Unit =
      edges.subtractOne(edge)
```

What's the complexity?

Attempt 1: Edge List

```
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)

Attempt 2: Linked Edge List

```
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 */
}
```

Attempt 2: Linked Edge List

```
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
      }
      /* ... */
}
```

Attempt 2: Linked Edge List

```
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

Attempt 2: Linked Edge List

```
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?

Attempt 2: Linked Edge List

```
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? Θ(1)

Attempt 2: Linked Edge List

```
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
      }
      /* ... */
    }
```

Attempt 2: Linked Edge List

```
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
      }
      /* ... */
    }
```

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

Attempt 2: Linked Edge List

```
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
      }
      /* ... */
    }
```

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

What is the complexity?

Attempt 2: Linked Edge List

```
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
      }
      /* ... */
    }
```

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

What is the complexity? Θ(1)

Attempt 2: Linked Edge List

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

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

Attempt 2: Linked Edge List

```
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

Attempt 2: Linked Edge List

```
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

What is the complexity?

Attempt 2: Linked Edge List

```
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

What is the complexity? Θ(1)

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)
    }
    /* ... */
}
```

Attempt 2: Linked Edge List

```scala
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?

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)
        }
    }
    /* ... */
}
```

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)
        }
    }
    /* ... */
}
```

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

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)
        }
    }
    /* ... */
}
```

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

What is the complexity?

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)
        }
    }
    /* ... */
}
```

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

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)
        }
    }
    /* ... */
}
```

How do we figure out what edges are incident?

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]()
    val edges = DoublyLinkedList[Edge]()
    class Vertex(label: V) = {
        /* ... */
        def outEdges =
            edges.filter { _.orig = this }

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

Attempt 2: Linked Edge List

```
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?

Attempt 2: Linked Edge List

```
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`: $O(m)$

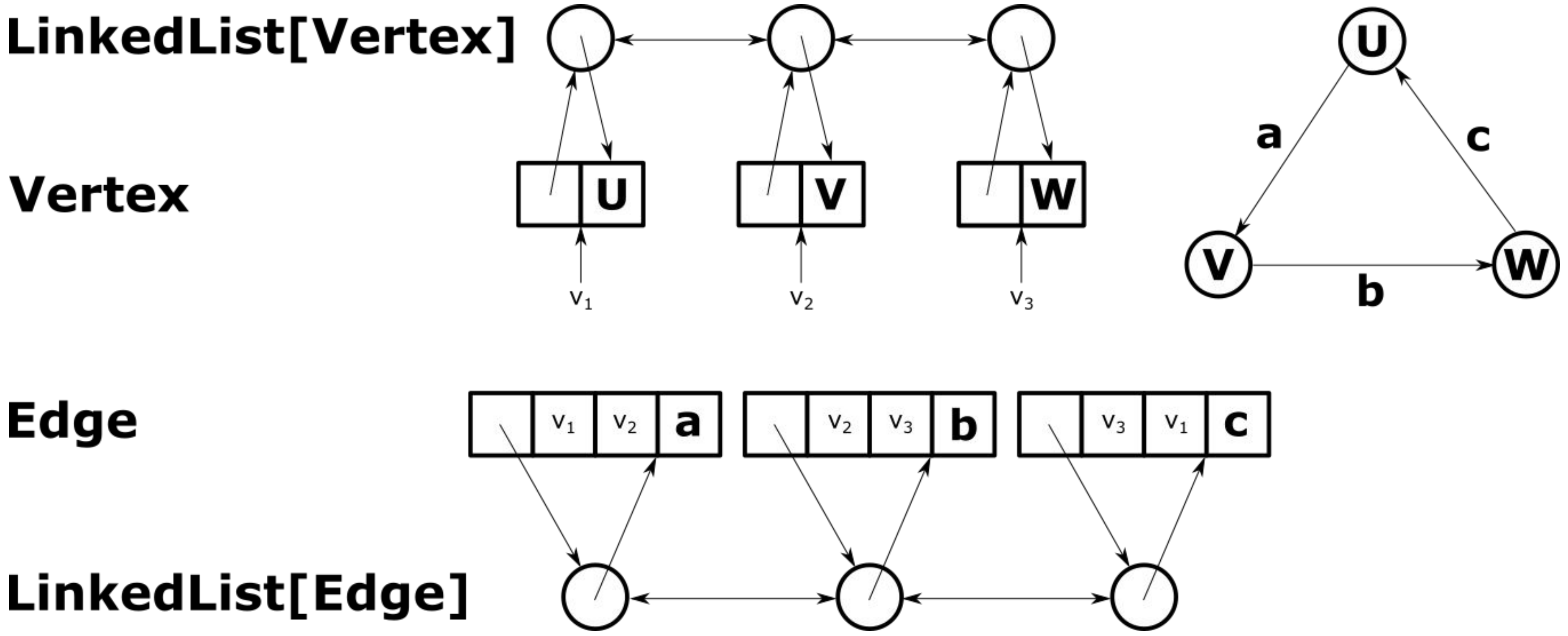
Edge List Summary

- `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



How can we improve?

How can we improve?

Idea: Store the in/out edges for each vertex!

Attempt 3: Adjacency List

```
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]()
        /* ... */
    }
    /* ... */
}
```

Attempt 3: Adjacency List

```
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

Attempt 3: Adjacency List

```
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
    }
    /* ... */
}
```

Attempt 3: Adjacency List

```
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
    }
    /* ... */
}
```

Also add our edge to the adjacency lists

Attempt 3: Adjacency List

```
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
    }
    /* ... */
}
```

Also add our edge to the adjacency lists

What is the complexity?

Attempt 3: Adjacency List

```
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
    }
    /* ... */
}
```

Also add our edge to the adjacency lists

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

Attempt 3: Adjacency List

```
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)
      }
      /* ... */
    }
```

Attempt 3: Adjacency List

```
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)
      }
      /* ... */
    }
```

Remove the edges from our adjacency lists…?

Attempt 3: Adjacency List

```scala
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)
    }
    /* ... */
  }
```

Remove the edges from our adjacency lists…?

What is the complexity?

Attempt 3: Adjacency List

```
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)
    }
    /* ... */
  }
```

Remove the edges from our adjacency lists...?

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

Attempt 4: Adjacency List

```
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

```scala
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
        /* ... */
    }
    /* ... */
}
```

Let's save references for each adjacency list!

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
    }
    /* ... */
}
```

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
    }
    /* ... */
}
```

Save our references when adding an edge

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
    }
    /* ... */
}
```

Save our references when adding an edge

What is the complexity?

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
    }
    /* ... */
}
```

Save our references when adding an edge

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

Attempt 4: Adjacency List

```
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)
    }
    /* ... */
}
```

Attempt 4: Adjacency List

```
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)
    }
    /* ... */
}
```

Remove by reference!

Attempt 4: Adjacency List

```
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)
    }
    /* ... */
}
```

Remove by reference!

What is the complexity?

Attempt 4: Adjacency List

```
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)
    }
    /* ... */
}
```

Remove by reference!

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

Attempt 4: Adjacency List

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

Attempt 4: Adjacency List

```
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?

Attempt 4: Adjacency List

```
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? $O(\deg(\mathit{vertex}))$

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`: $O(\deg(vertex))$
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Adjacency List Summary

- `addEdge`, `addVertex`: *O*(1)
- `removeEdge`: *O*(1)
- `removeVertex`: ***O*(deg(vertex))**
- `vertex.incidentEdges`: ***O*(deg(vertex))**
- `vertex.edgeTo`:
- **Space Used:**

Adjacency List Summary

- `addEdge, addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\mathrm{deg}(\mathrm{vertex}))$
- `vertex.incidentEdges`: $O(\mathrm{deg}(\mathrm{vertex}))$
- `vertex.edgeTo`: $O(\mathrm{deg}(\mathrm{vertex}))$
- **Space Used:**

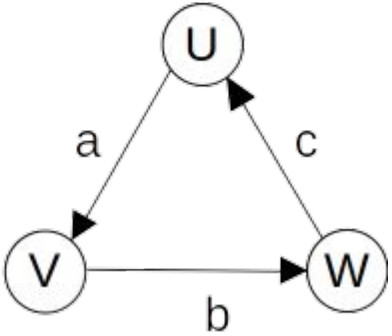
Adjacency List Summary

- `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

Destination

Origin	U	V	W
U	*-*	***a***	*-*
V	*-*	*-*	***b***
W	***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:** $O(n^2)$