

CSE 250

Data Structures

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

Lec 19: Intro to Graphs

Announcements

- WA3 due this Sunday @ 11:59PM

Let's Talk About Graphs

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

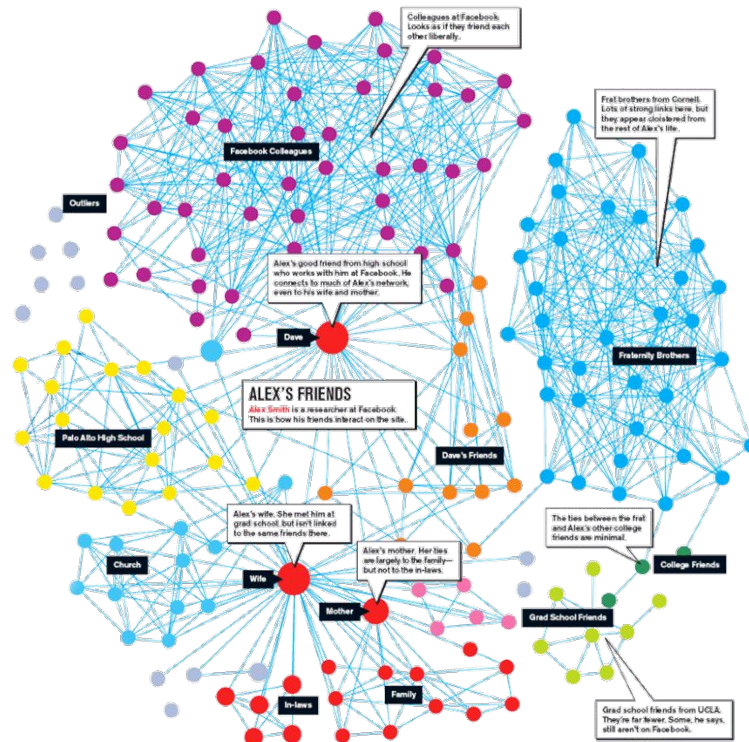
- V is a set of **vertices**
- E is a set of vertex pairs called **edges**
- Edges and vertices may also store data (**labels**)

Graphs

Example: A social network

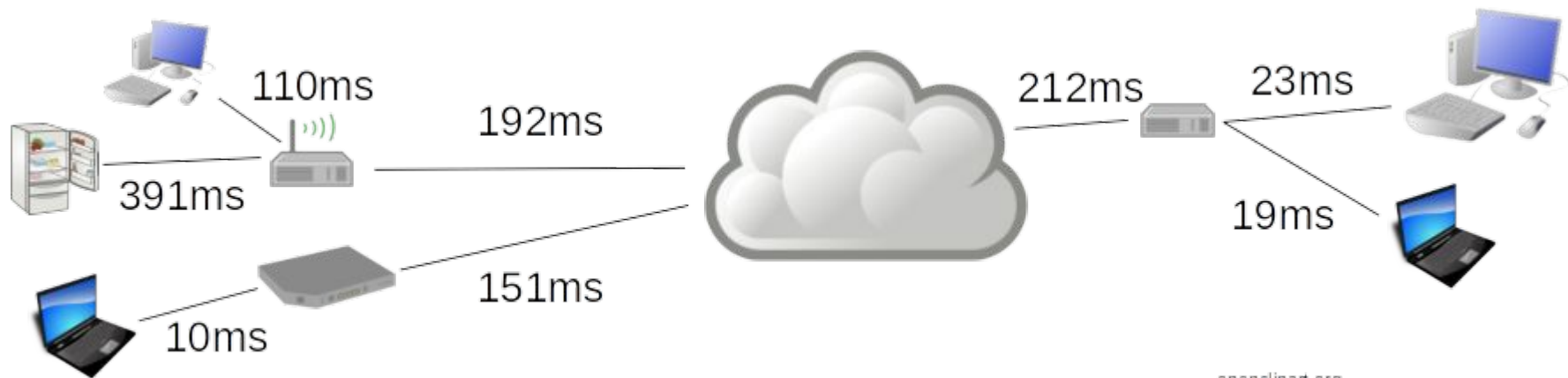
(nodes store users, pictures, tweets, etc)

(edges store interactions)



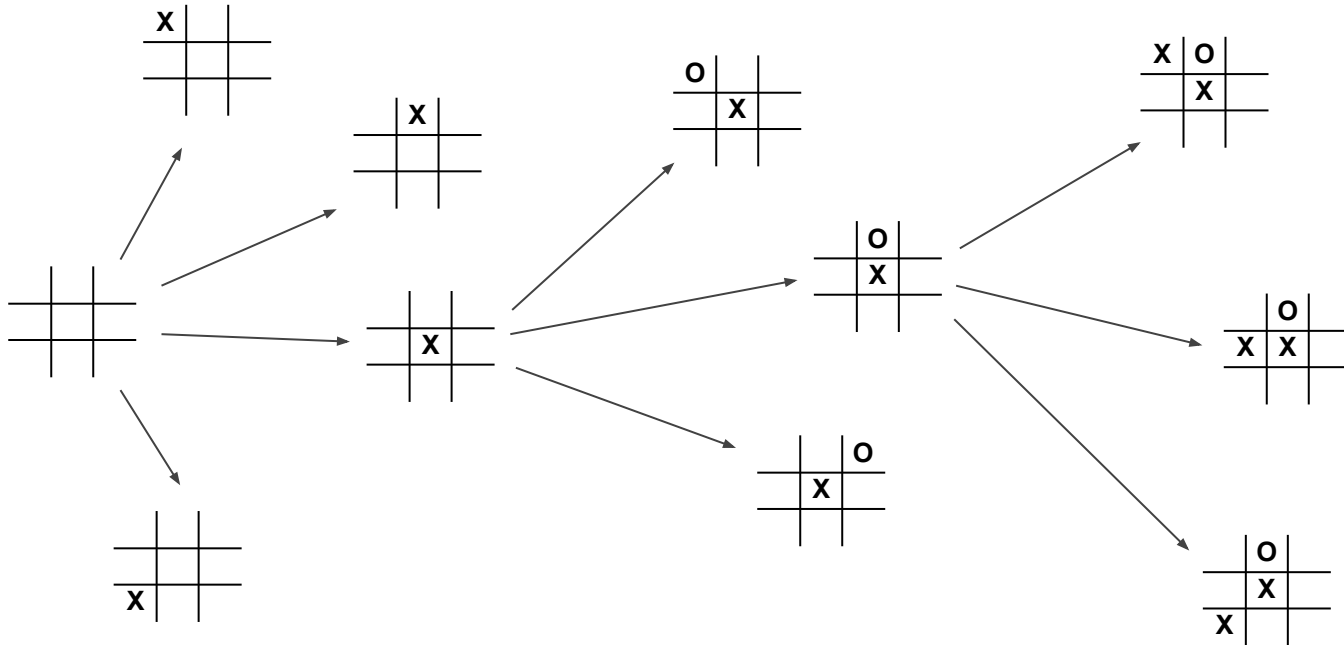
Graphs

Example: A computer network
(edges store ping, nodes store addresses)



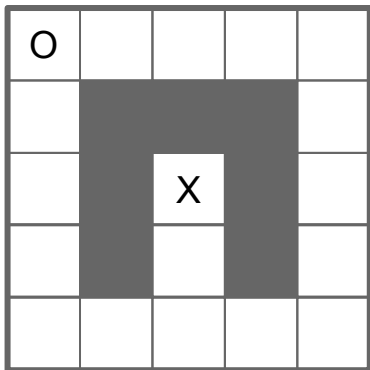
Graphs

Example: Moves in a game



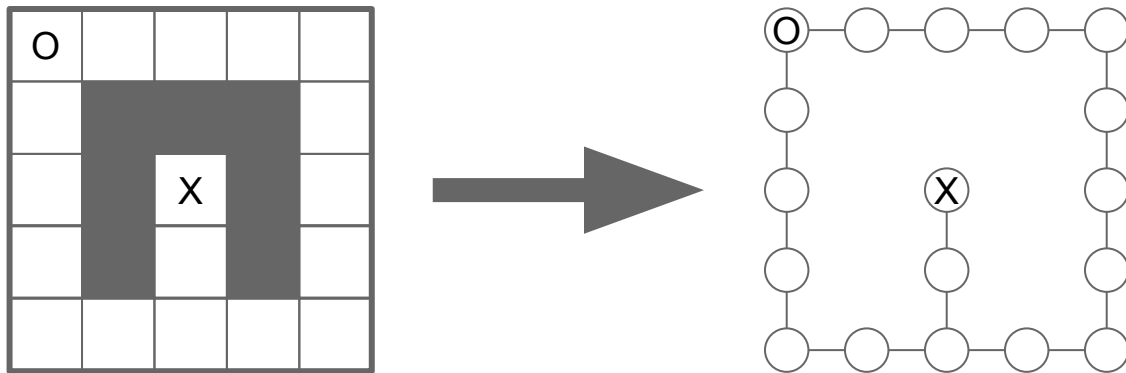
Back to Mazes

How could we represent our maze as a graph?



Back to Mazes

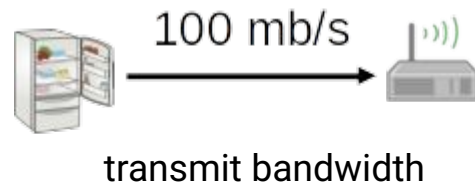
How could we represent our maze as a graph?



Edge Types

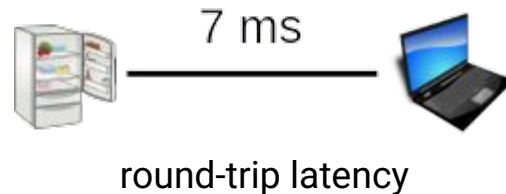
Directed Edge (asymmetric relationship)

- Ordered pair of vertices (u, v)
- origin (u) \rightarrow destination (v)



Undirected Edge (symmetric relationship)

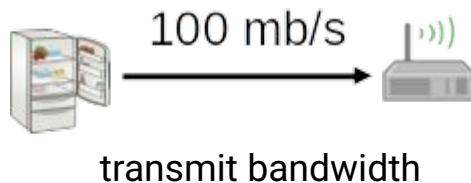
- Unordered pair of vertices (u, v)



Edge Types

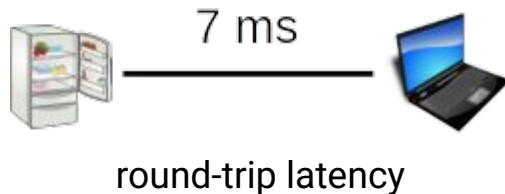
Directed Edge (asymmetric relationship)

- Ordered pair of vertices (u, v)
- origin (u) \rightarrow destination (v)



Undirected Edge (symmetric relationship)

- Unordered pair of vertices (u, v)



Directed Graph: All edges are directed

Undirected Graph: All edges are undirected

Terminology

Endpoints of an edge

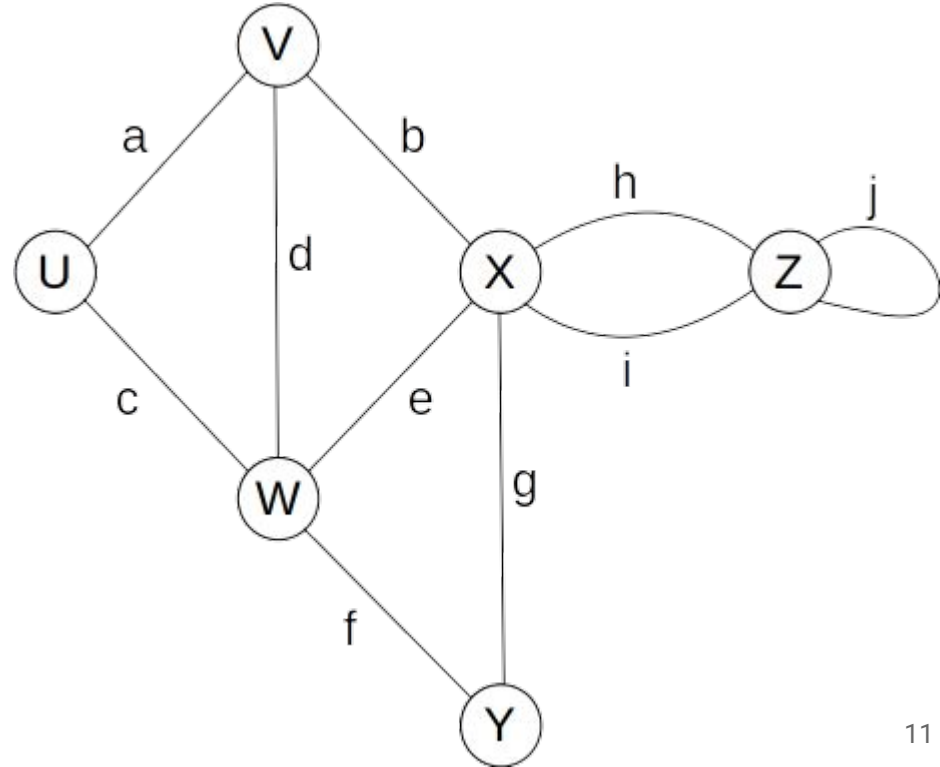
U, V are endpoints of a

Adjacent Vertices

U, V are adjacent

Degree of a vertex

X has degree 5



Terminology

Edges incident on a vertex

a, b, d are incident on V

Parallel Edges

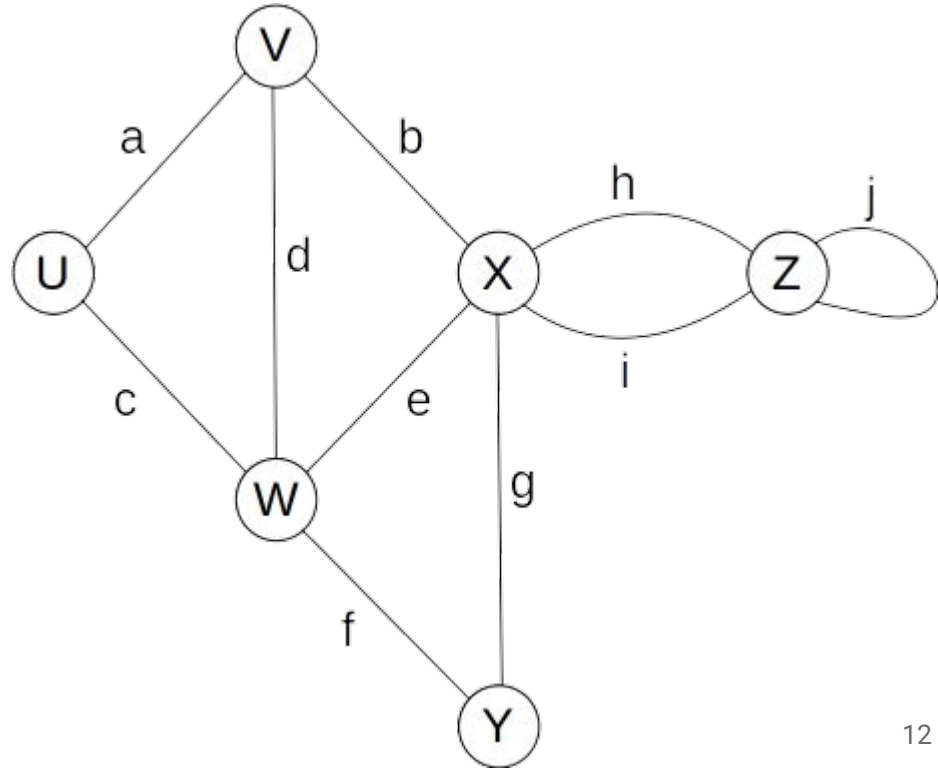
h, i are parallel

Self-Loop

j is a self-loop

Simple Graph

A graph without parallel edges or self-loops



Terminology

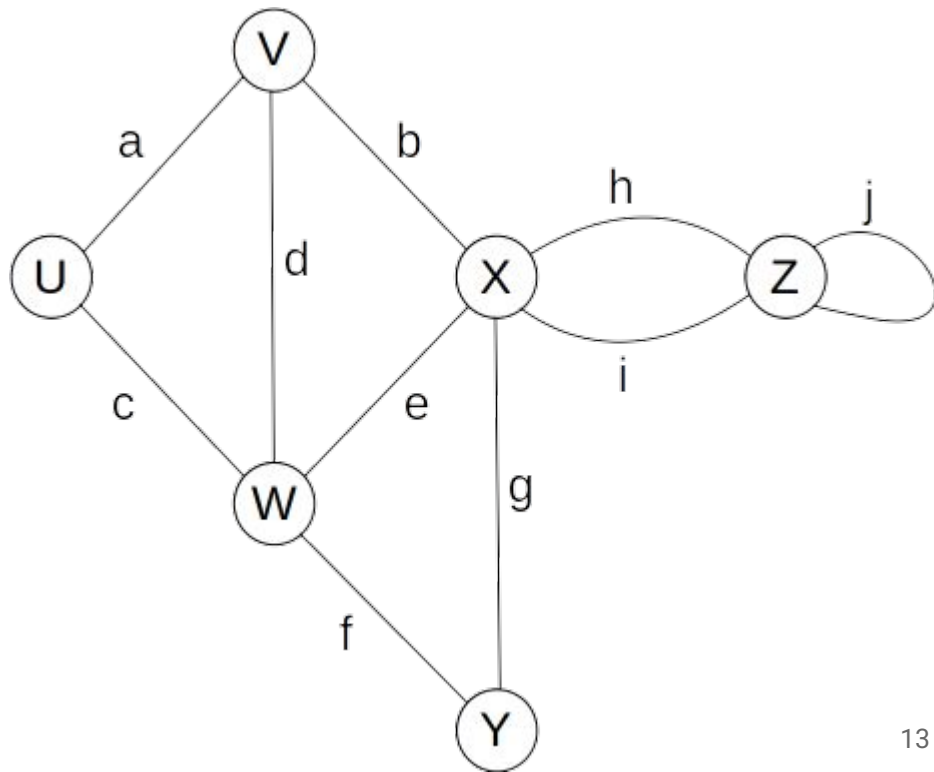
Path

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

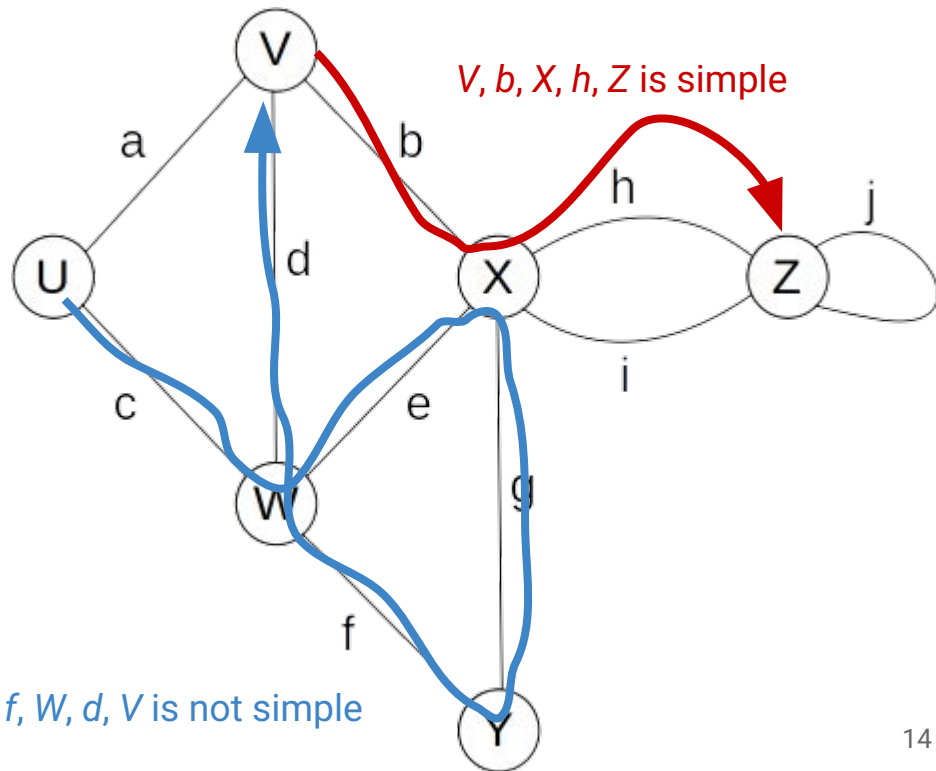
Path

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

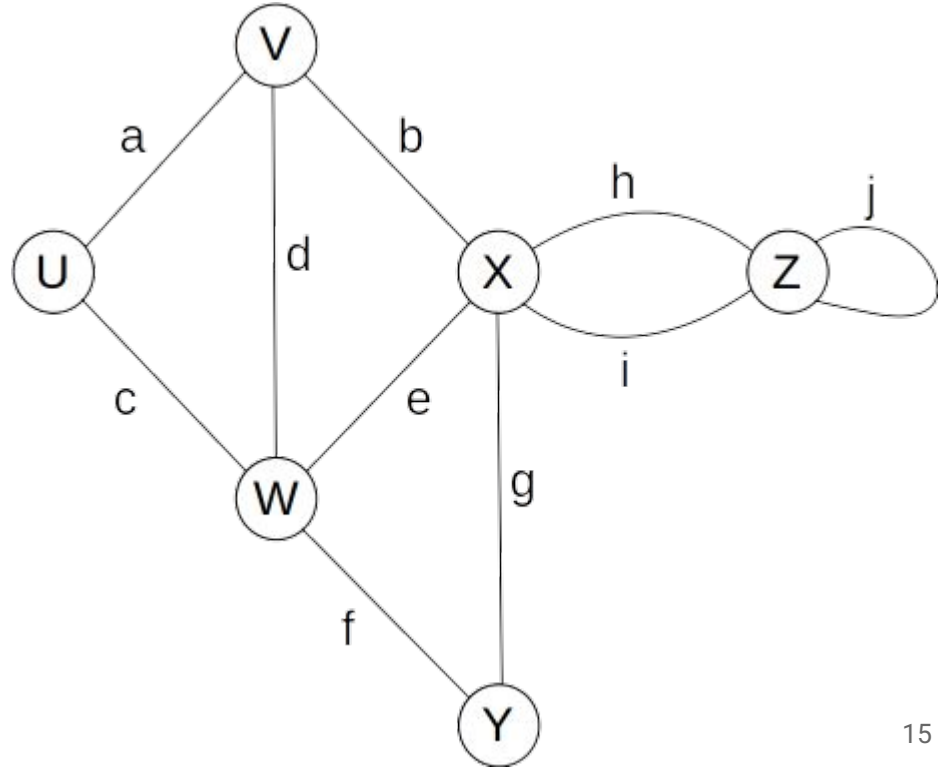
Terminology

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



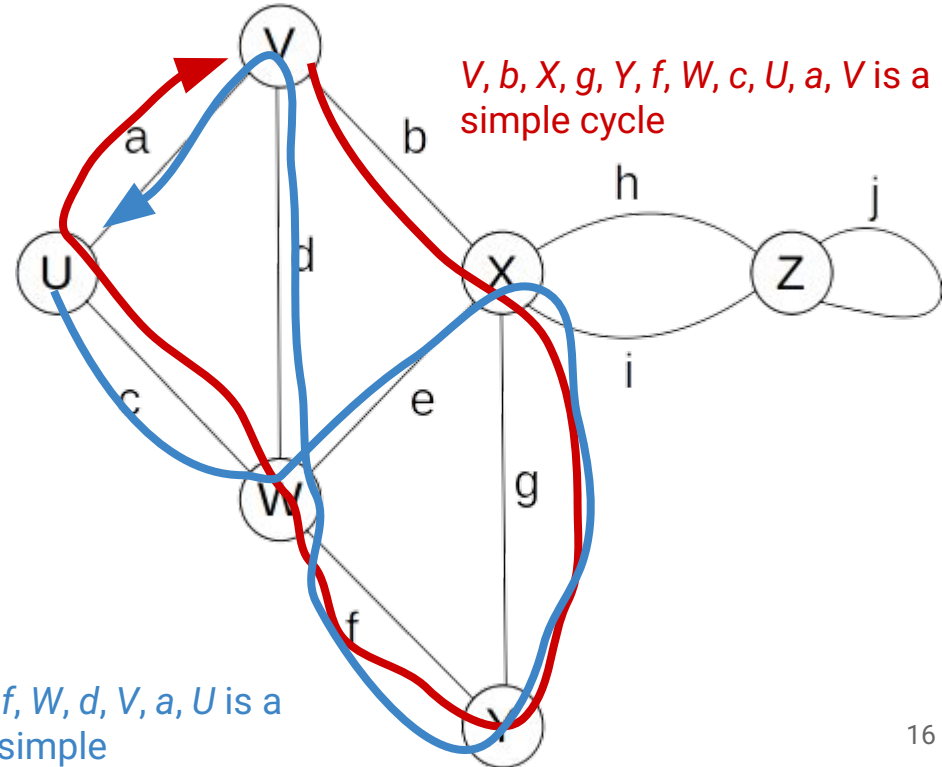
Terminology

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



Notation

n The number of vertices

m The number of edges

$\text{deg}(v)$ 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

What can we do with a Graph?

A (Directed) Graph ADT

What can we do with a Graph?

- Iterate through the vertices
- Iterate through the edges
- Add a vertex
- Add an edge
- Remove a vertex
- Remove an edge

A (Directed) Graph ADT

```
1 public interface Graph<V, E> {  
2     public Iterator<Vertex> vertices();  
3     public Iterator<Edge> edges();  
4     public Vertex addVertex(V label);  
5     public Edge addEdge(Vertex orig, Vertex dest, E label);  
6     public void removeVertex(Vertex vertex);  
7     public void removeEdge(Edge edge);  
8 }
```

A (Directed) Graph ADT

What can we do with a Vertex?

A (Directed) Graph ADT

What can we do with a Vertex?

- Get it's label
- Get the outgoing edges
- Get the incoming edges
- Get all incident edges
- Check if it's adjacent to another Vertex

A (Directed) Graph ADT

What can we do with an Edge?

- Get it's label
- Get the incident vertices

A (Directed) Graph ADT

```
1 public interface Vertex<V,E> {
2     public V getLabel();
3     public Iterator<Edge> getOutEdges();
4     public Iterator<Edge> getInEdges();
5     public Iterator<Edge> getIncidentEdges();
6     public boolean hasEdgeTo(Vertex v);
7 }
8
9 public interface Edge<V,E> {
10    public Vertex getOrigin();
11    public Vertex getDestination();
12    public E getLabel();
13 }
```

Implementation Attempt 1: Edge List

Data Model:

A List of Edges

(ArrayList)

A List of Vertices

(ArrayList)

Implementation Attempt 1: Edge List

```
1 public class EdgeList<V,E> implements Graph<V,E> {  
2     List<Vertex> vertices = new ArrayList<Vertex>();  
3     List<Edge> edges = new ArrayList<Edge>();  
4     /*...*/  
5 }
```


Implementation Attempt 1: Edge List

```
public Vertex addVertex(V label) {  
    Vertex v = new Vertex(label);  
    vertices.add(v);  
    return v;  
}  
  
public Edge addEdge(Vertex orig, Vertex dest, E label) {  
    Edge e = new Edge(orig, dest, label);  
    edges.add(e);  
    return e;  
}
```

What's the complexity?

Implementation Attempt 1: Edge List

```
public Vertex addVertex(V label) {  
    Vertex v = new Vertex(label);           Amortized  $\Theta(1)$   
    vertices.add(v);  
    return v;  
}  
  
public Edge addEdge(Vertex orig, Vertex dest, E label) {  
    Edge e = new Edge(orig, dest, label);  
    edges.add(e);  
    return e;                               Amortized  $\Theta(1)$   
}
```

What's the complexity?

Implementation Attempt 1: Edge List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge);  
3 }
```

What's the complexity?

Implementation Attempt 1: Edge List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge);  
3 }
```

What's the complexity?

We have to search for edge by value in an unsorted list! $O(m)$

Attempt 2: Linked Edge List

Data Model:

A List of Edges
(LinkedList)

A List of Vertices
(LinkedList)

Attempt 2: Linked Edge List

```
1 public class LinkedList<V,E> implements Graph<V,E> {  
2     List<Vertex> vertices = new LinkedList<Vertex>();  
3     List<Edge> edges = new LinkedList<Edge>();  
4     /*...*/  
5 }
```

Attempt 2: Linked Edge List

```
public Vertex addVertex(V label) {  
    Vertex v = new Vertex(label);  
    vertices.add(v);  
    return v;  
}  
  
public Edge addEdge(Vertex orig, Vertex dest, E label) {  
    Edge e = new Edge(orig, dest, label);  
    edges.add(e);  
    return e;  
}
```

What's the complexity?

Attempt 2: Linked Edge List

```
public Vertex addVertex(V label) {  
    Vertex v = new Vertex(label);            $\Theta(1)$   
    vertices.add(v);  
    return v;  
}  
  
public Edge addEdge(Vertex orig, Vertex dest, E label) {  
    Edge e = new Edge(orig, dest, label);  
    edges.add(e);  
    return e;                                $\Theta(1)$   
}
```

What's the complexity?

Attempt 2: Linked Edge List

```
1 public void removeEdge(Edge<V,E> edge) {  
2     edges.remove(edge);  
3 }
```

What's the complexity?

Attempt 2: Linked Edge List

```
1 public void removeEdge(Edge<V,E> edge) {  
2     edges.remove(edge);  
3 }
```

What's the complexity?

We have to search for edge by value in an unsorted list! $O(m)$

Attempt 2: Linked Edge List

```
1 public void removeEdge(Edge<V,E> edge) {  
2     edges.remove(edge);  
3 }
```

What's the complexity?

We have to search for edge by value in an unsorted list! $O(m)$

Solution: What if we stored a reference to the node?

Attempt 2: Linked Edge List

```
1 public class LinkedList<V,E> implements Graph<V,E> {  
2     List<Vertex> vertices = new CustomLinkedList<Vertex>();  
3     List<Edge> edges = new CustomLinkedList<Edge>();  
4     /*...*/  
5 }
```

```
1 public class Vertex<V,E> {  
2     private Node<Vertex> node;  
3     /*...*/  
4 }  
5 public class Edge<V,E> {  
6     private Node<Edge> node;  
7     /*...*/  
8 }
```

Attempt 2: Linked Edge List

```
1 public Vertex addVertex(V label) {
2     Vertex v = new Vertex(label);
3     Node<Vertex> node = vertices.add(v);
4     v.node = node;
5     return v;
6 }
7
8 public Edge addEdge(Vertex orig, Vertex dest, E label) {
9     Edge e = new Edge(orig, dest, label);
10    Node<Edge> node = edges.add(e);
11    e.node = node;
12    return e;
13 }
```

Attempt 2: Linked Edge List

```
1 public Vertex addVertex(V label) {
2     Vertex v = new Vertex(label);
3     Node<Vertex> node = vertices.add(v);
4     v.node = node;
5     return v;
6 }
7
8 public Edge addEdge(Vertex orig, Vertex dest, E label) {
9     Edge e = new Edge(orig, dest, label);
10    Node<Edge> node = edges.add(e);
11    e.node = node;
12    return e;
13 }
```

Both add methods still $\Theta(1)$

Attempt 2: Linked Edge List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge.node);  
3 }
```

What's the complexity?

Attempt 2: Linked Edge List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge.node);  
3 }
```

What's the complexity? $\Theta(1)$

Attempt 2: Linked Edge List

```
1 public void removeVertex(Vertex vertex) {  
2     vertices.remove(vertex.node);  
3 }
```

What's the complexity?

Attempt 2: Linked Edge List

```
1 public void removeVertex(Vertex vertex) {  
2     vertices.remove(vertex.node);  
3 }
```

What's the complexity? $\Theta(1)$

What's the problem?

Attempt 2: Linked Edge List

```
1 public void removeVertex(Vertex vertex) {  
2     vertices.remove(vertex.node);  
3 }
```

What's the complexity? $\Theta(1)$

What's the problem? The removed vertex may be incident to edges, which now have an endpoint that is not in the graph!

Attempt 2: Linked Edge List

```
1 public void removeVertex(Vertex vertex) {  
2     for(edge : vertex.getIncidentEdges()) {  
3         removeEdge(edge.node)  
4     }  
5     vertices.remove(vertex.node);  
6 }
```

What's the complexity?

Attempt 2: Linked Edge List

```
1 public void removeVertex(Vertex vertex) {  
2     for(edge : vertex.getIncidentEdges()) {  
3         removeEdge(edge.node)  
4     }  
5     vertices.remove(vertex.node);  
6 }
```

How do we get the incident edges
with our current model?

What's the complexity?

Attempt 2: Linked Edge List

```
1 public Iterator<Edge> getIncidentEdges(Vertex vertex) {  
2     ArrayList<Edge> incidentEdges = new ArrayList<>();  
3     for(edge : edges) {  
4         if(edge.origin.equals(vertex) || edge.dest.equals(vertex)) {  
5             incidentEdges.add(edge);  
6         }  
7     }  
8     return incidentEdges.iterator();  
9 }
```

What is the complexity?

Attempt 2: Linked Edge List

```
1 public Iterator<Edge> getIncidentEdges(Vertex vertex) {
2   ArrayList<Edge> incidentEdges = new ArrayList<>();
3   for(edge : edges) {
4     if(edge.origin.equals(vertex) || edge.dest.equals(vertex)) {
5       incidentEdges.add(edge);
6     }
7   }
8   return incidentEdges.iterator();
9 }
```

What is the complexity? $O(m)$

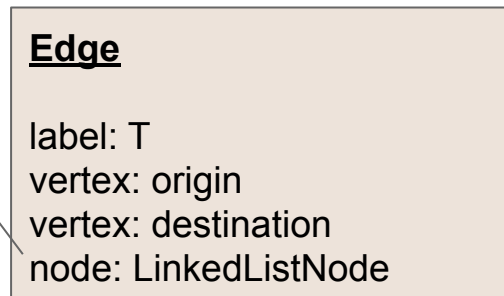
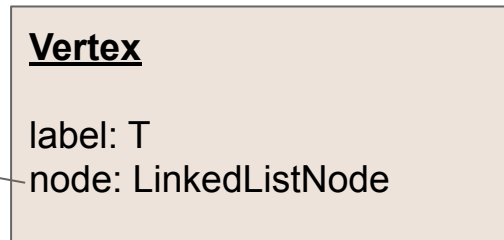
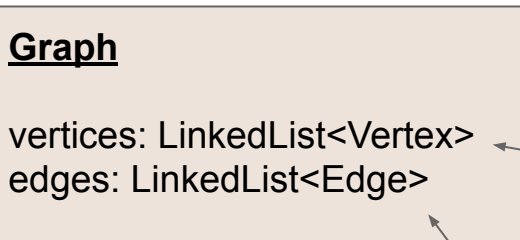
Attempt 2: Linked Edge List

```
1 public void removeVertex(Vertex vertex) {  
2     for(edge : vertex.getIncidentEdges()) {  
3         removeEdge(edge.node)  
4     }  
5     vertices.remove(vertex.node);  
6 }
```

How do we get the incident edges
with our current model?

What's the complexity? $O(m) = O(n^2)$

Edge List Summary



Storing the list nodes in the edges/vertices allows us to remove by reference in $\Theta(1)$ time

Edge List Summary

- `addEdge`, `addVertex`:
- `removeEdge`:
- `removeVertex`:
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Edge List Summary

- `addEdge`, `addVertex`: $\Theta(1)$
- `removeEdge`: $\Theta(1)$
- `removeVertex`: $O(m)$
- `vertex.incidentEdges`: $O(m)$
- `vertex.edgeTo`: $O(m)$
- **Space Used**: $O(n) + O(m)$

Edge List Summary

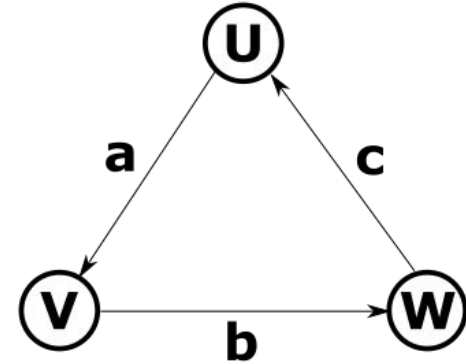
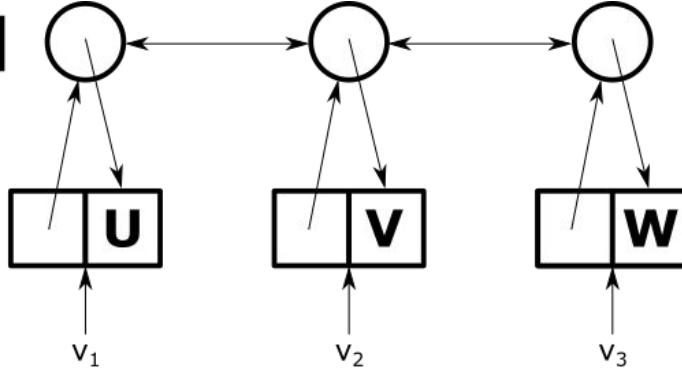
- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(m)$
- `vertex.incidentEdges`: $O(m)$
- `vertex.edgeTo`: $O(m)$
- **Space Used: $O(n) + O(m)$**



Involves checking every edge in the graph

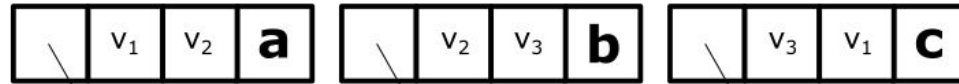
Edge List Summary

LinkedList[Vertex]



Vertex

Edge



LinkedList[Edge]



How can we improve?

How can we improve?

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

(Called an adjacency list)

Adjacency List

```
1 public class Vertex<V,E> {  
2     public Node<Vertex> node;  
3     public List<Edge> inEdges = new CustomLinkedList<Edge>();  
4     public List<Edge> outEdges = new CustomLinkedList<Edge>();  
5     /*...*/  
6 }
```

Each vertex stores a list of **inEdges** and **outEdges**, which are maintained as the graph is modified...

What functions need to change to maintain these lists?

Adjacency List

```
1 public Edge addEdge(Vertex orig, Vertex dest, E label) {  
2     Edge e = new Edge(orig, dest, label);  
3     e.node = edges.add(e);  
4     orig.outEdges.add(e);  
5     dest.inEdges.add(e);  
6     return e;  
7 }
```

← When we add an edge to the graph, also add it to the appropriate adjacency lists

What is the complexity of **addEdge** now?

Adjacency List

```
1 public Edge addEdge(Vertex orig, Vertex dest, E label) {  
2     Edge e = new Edge(orig, dest, label);  
3     e.node = edges.add(e);  
4     orig.outEdges.add(e);  
5     dest.inEdges.add(e);  
6     return e;  
7 }
```

← When we add an edge to the graph, also add it to the appropriate adjacency lists

What is the complexity of **addEdge** now? Still $\Theta(1)$

Adjacency List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge.node);  
3     edge.orig.outEdges.remove(edge);  
4     edge.dest.inEdges.remove(edge);  
5 }
```

← When we remove an edge from the graph, also remove it from the adjacency lists

What is the complexity of **removeEdge** now?

Adjacency List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge.node);  
3     edge.orig.outEdges.remove(edge);  
4     edge.dest.inEdges.remove(edge);  
5 }
```

← When we remove an edge from the graph, also remove it from the adjacency lists

What is the complexity of `removeEdge` now? $O(\text{deg}(\text{orig}) + \text{deg}(\text{dest}))$:(

But how can we fix this?

Adjacency List

```
1 public class Edge<V,E> {  
2     public Node<Edge> node;  
3     public Node<Edge> inNode;  
4     public Node<Edge> outNode;  
5     /*...*/  
6 }
```

Each Edge now also stores a reference to the nodes in each adjacency list

Adjacency List

```
1 public Edge addEdge(Vertex orig, Vertex dest, E label) {  
2     Edge e = new Edge(orig, dest, label);  
3     e.node = edges.add(e);  
4     e.outNode = orig.outEdges.add(e);  
5     e.inNode = dest.inEdges.add(e);  
6     return e;  
7 }
```

← When we add an edge to the graph, also add it to the appropriate adjacency lists AND store the node refs in the Edge object

What is the complexity of **addEdge** now? Still $\Theta(1)$

Adjacency List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge.node);  
3     edge.orig.outEdges.remove(edge.outNode);  
4     edge.dest.inEdges.remove(edge.inNode);  
5 }
```

← When we remove an edge from the graph, also remove it from the adjacency lists (remove by reference)

What is the complexity of **removeEdge** now?

Adjacency List

```
1 public void removeEdge(Edge edge) {  
2     edges.remove(edge.node);  
3     edge.orig.outEdges.remove(edge.outNode);  
4     edge.dest.inEdges.remove(edge.inNode);  
5 }
```

← When we remove an edge from the graph, also remove it from the adjacency lists (remove by reference)

What is the complexity of **removeEdge** now? $\Theta(1)$

Adjacency List

So, we are able to store and maintain adjacency lists in each vertex while still keeping a $\Theta(1)$ runtime for `addVertex`, `addEdge`, and `removeEdge`

How much extra space is used?

Adjacency List

So, we are able to store and maintain adjacency lists in each vertex while still keeping a $\Theta(1)$ runtime for `addVertex`, `addEdge`, and `removeEdge`

How much extra space is used? $\Theta(1)$ per edge

Each edge only appears in 3 lists:

- The edge list
- One vertices inList
- One vertices outList

Adjacency List

So, we are able to store and maintain adjacency lists in each vertex while still keeping a $\Theta(1)$ runtime for `addVertex`, `addEdge`, and `removeEdge`

How much extra space is used? $\Theta(1)$ per edge

Each edge only appears in 3 lists:

- The edge list
- One vertices inList
- One vertices outList

But now what have we gained?

Adjacency List

```
1 public void removeVertex(Vertex v) {  
2     for(edge : v.getIncidentEdges()) {  
3         removeEdge(edge.node)  
4     }  
5     vertices.remove(v.node);  
6 }
```

What is the complexity of **removeVertex** now?

Adjacency List

```
1 public void removeVertex(Vertex v) {  
2     for(edge : v.getIncidentEdges()) {  
3          $\Theta(1)$   
4     }  
5      $\Theta(1)$   
6 }
```

What is the complexity of `removeVertex` now?

Adjacency List

```
1 public void removeVertex(Vertex v) {  
2     for(edge : v.getIncidentEdges()) {  
3          $\Theta(1)$   
4     }  
5      $\Theta(1)$   
6 }
```

We now have a reference to the list of edges in $\Theta(1)$ time, and there are **deg(v)** edge in the list

What is the complexity of **removeVertex** now?

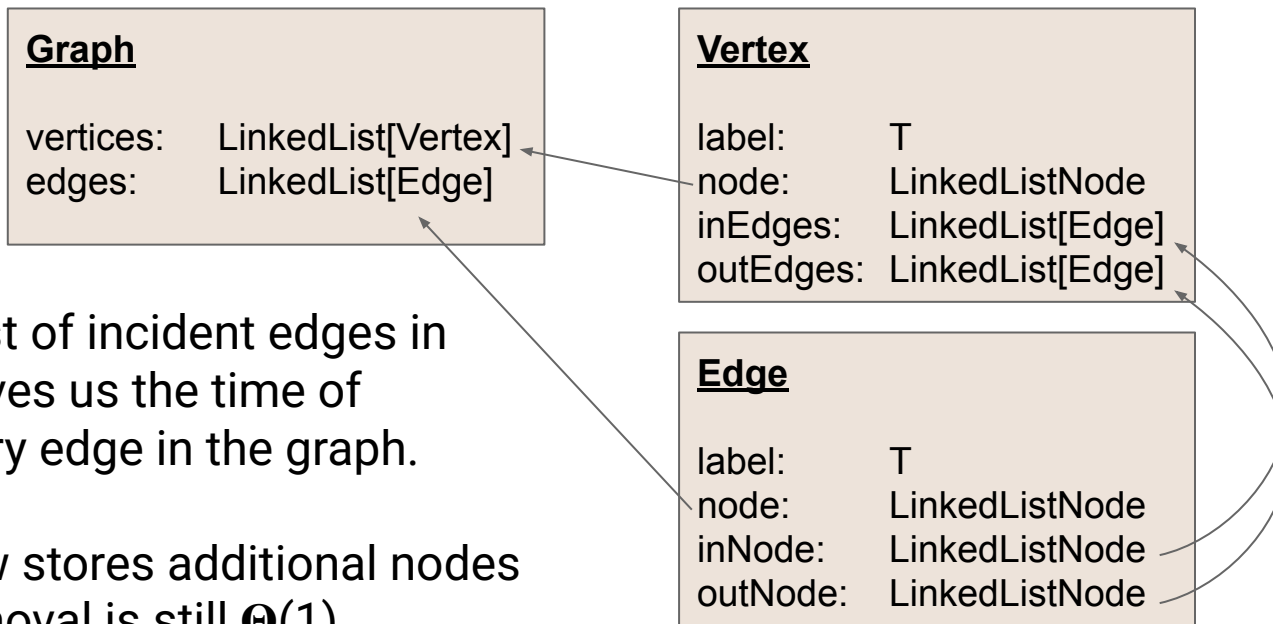
Adjacency List

```
1 public void removeVertex(Vertex v) {  
2     for(edge : v.getIncidentEdges()) {  
3          $\Theta(1)$   
4     }  
5      $\Theta(1)$   
6 }
```

We now have a reference to the list of edges in $\Theta(1)$ time, and there are **deg(v)** edge in the list

What is the complexity of **removeVertex** now? $\Theta(\mathit{deg}(v))$

Adjacency List Summary



Storing the list of incident edges in the vertex saves us the time of checking every edge in the graph.

The edge now stores additional nodes to ensure removal is still $\Theta(1)$

Adjacency List Summary

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

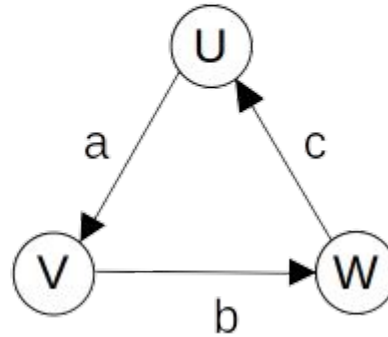
Adjacency List Summary

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

Now we already know what edges are incident without having to check them all

Adjacency Matrix

		<u>Destination</u>		
		U	V	W
<u>Origin</u>	U	-	<i>a</i>	-
	V	-	-	<i>b</i>
	W	<i>c</i>	-	-



Adjacency Matrix Summary

- `addEdge`, `removeEdge`:
- `addVertex`, `removeVertex`:
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Adjacency Matrix Summary


Just change a single entry of the matrix

- `addEdge`, `removeEdge`: $\Theta(1)$
- `addVertex`, `removeVertex`:
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Adjacency Matrix Summary

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


Resize and copy the whole matrix



Adjacency Matrix Summary

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

Check the row and column for that vertex



Adjacency Matrix Summary

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


Check a single entry of the matrix



Adjacency Matrix Summary

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

How does this relate to space of
edge/adjacency lists?



Adjacency Matrix Summary

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

How does this relate to space of edge/adjacency lists? **If the matrix is "dense" it's about the same**