CSE 250: Graphs

Lecture 18

Oct 9, 2023

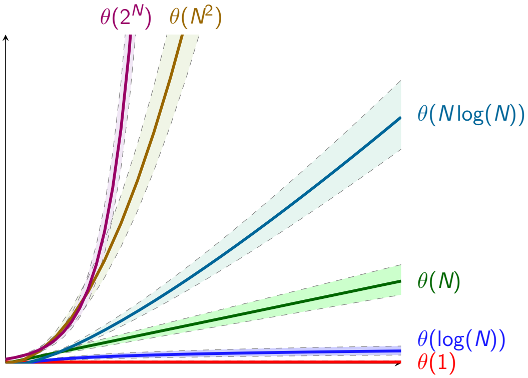
Reminders

- WA3 due Sun, Oct 13 at 11:59 PM
  - See Piazza for a hint.

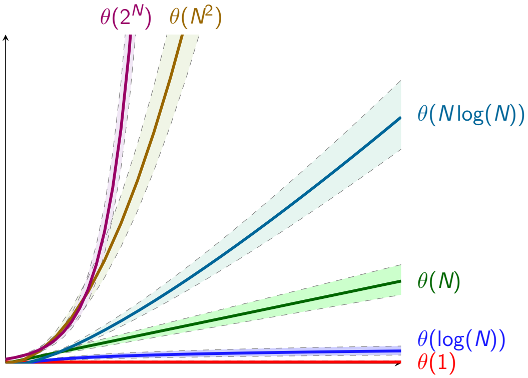
Graphs

What is a graph?

Graphs

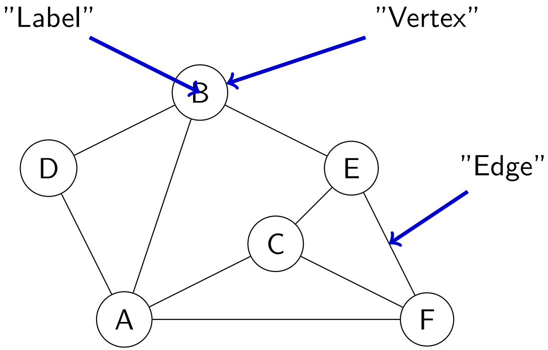


Graphs



Not this kind of graph.

Graphs



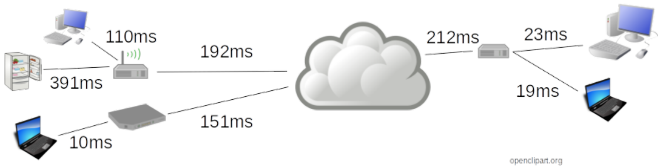
Graphs

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

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

Graph Examples

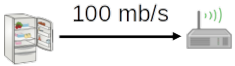
Example: A computer network
(edges store ping, nodes store IP addresses)



Edge Types

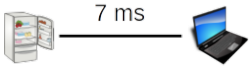
Directed Edge (e.g., transmit bandwidth)

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



Undirected Edge (e.g., latency)

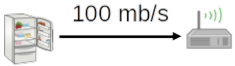
- Unordered pair of vertices (u, v)



Edge Types

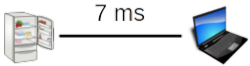
Directed Edge (e.g., transmit bandwidth)

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



Undirected Edge (e.g., latency)

- Unordered pair of vertices (u, v)



Directed Graph

- All edges are directed

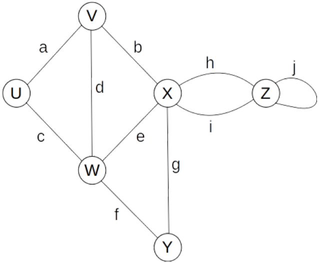
Undirected Graph

- All edges are undirected

Other Applications of Graphs

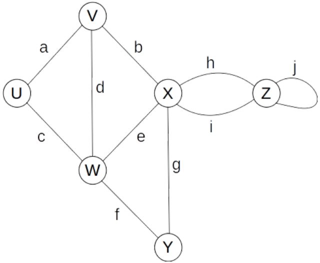
- Transportation (Flight/Road/Rail Routing)
- Protein/Protein interactions
- Social Networks
- Dependency Tracking
- Game Development (Planning, Routefinding)
- Taxonomies

Graph Terminology



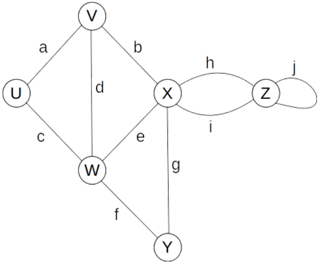
- **Endpoints of an edge**
  U, V are the endpoints of a.
- **Edges incident on a vertex**
  a, b, d are incident on V.
- **Adjacent Vertices**
  U, V are adjacent.
- **Degree of a vertex (# of incident edges)**
  X has degree 5.
- **Parallel Edges** (same endpoints)
  h, i are parallel.
- **Self-loop** (same vertex is start and end)
  j is a self-loop.
- **Simple Graph**
  A graph with no parallel edges or self-loops.

Paths



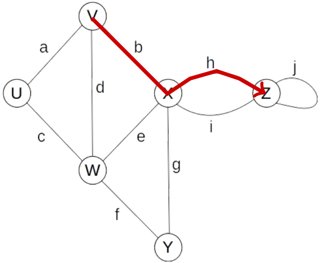
- **Path**
  A sequence of alternating vertices and edges
  - Begins with a vertex
  - Ends with a vertex
  - Each edge is preceded/followed by its endpoints

Paths



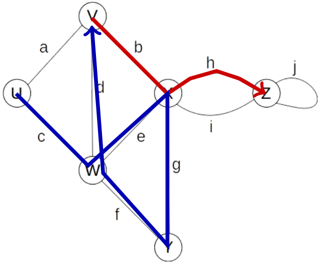
- **Path**
  A sequence of alternating vertices and edges
  - Begins with a vertex
  - Ends with a vertex
  - Each edge is preceded/followed by its endpoints
- **Simple Path**
  A path that never crosses the same vertex/edge twice

Paths



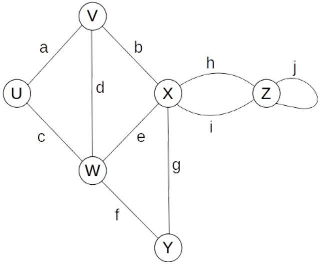
- **Path**
  A sequence of alternating vertices and edges
  - Begins with a vertex
  - Ends with a vertex
  - Each edge is preceded/followed by its endpoints
- **Simple Path**
  A path that never crosses the same vertex/edge twice
- **Examples**
  V, b, X, h, Z is a simple path.

Paths



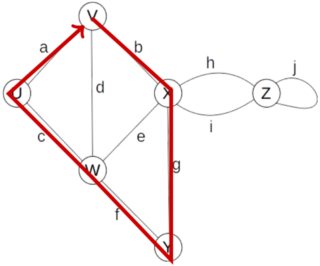
- **Path**
  A sequence of alternating vertices and edges
  - Begins with a vertex
  - Ends with a vertex
  - Each edge is preceded/followed by its endpoints
- **Simple Path**
  A path that never crosses the same vertex/edge twice
- **Examples**
  V, b, X, h, Z is a simple path.
  U, c, W, e, X, g, Y, f, W, d, V is a path that is not simple.

Cycles



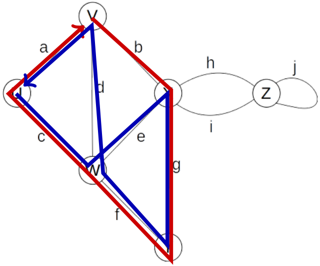
- **Cycle**
  A path that starts and ends on the same vertex.
  - Must contain at least one edge
- **Simple Cycle**
  A cycle where all of the edges and vertices are distinct (except the start/end vertex).

Cycles



- **Cycle**
  A path that starts and ends on the same vertex.
  - Must contain at least one edge
- **Simple Cycle**
  A cycle where all of the edges and vertices are distinct (except the start/end vertex).
- **Examples**
  $V, b, X, g, Y, f, W, c, U, a, V$ is a simple cycle.

Cycles



- **Cycle**
  A path that starts and ends on the same vertex.
  - Must contain at least one edge
- **Simple Cycle**
  A cycle where all of the edges and vertices are distinct (except the start/end vertex).
- **Examples**
  $V, b, X, g, Y, f, W, c, U, a, V$ 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
- $\texttt{deg}(v)$: The degree of a vertex

Handshake Theorem

$$\sum_{v \in V} \texttt{deg}(v) = 2M$$

Handshake Theorem

$$\sum_{v \in V} \texttt{deg}(v) = 2M$$

Proof (sketch): Each edge adds 1 to the degree of 2 vertices.

Edge Limit

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

$$M \leq N \cdot (N - 1)$$

Edge Limit

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

$$M \leq N \cdot (N - 1)$$

Proof (sketch):

- Each pair is connected at most once (no parallel edges)
- N possible start vertices
- $(N - 1)$ possible end vertices (no self-loops)
- $N \cdot (N - 1)$ distinct combinations possible

Hey...

Isn't this supposed to be a data structures class?

The Directed Graph ADT

Interfaces

- `Graph<V, E>`
  - `V`: The vertex label type.
  - `E`: The edge label type.
- `Vertex<V, E>`
  - ... represents a single element (like a LinkedListNode)
  - ... stores a single value of type `V`
- `Edge<V, E>`
  - ... represents an edge (a pair of vertices)
  - ... stores a single value of type `E`

The Directed Graph ADT

What can we do with a (directed) graph?

The Directed Graph ADT

What can we do with a (directed) graph?

- Iterate over its vertices
- Iterate over its edges
- Add a vertex
- Add an edge
- Remove a vertex
- Remove an edge

The Directed Graph ADT

```
interface Graph<V, E> {
  public Iterator<Vertex> vertices()
  public Iterator<Edge> edges()
  public Vertex addVertex(V label): Vertex
  public Edge addEdge(Vertex orig, Vertex dest, E label): Edge
  public void removeVertex(Vertex vertex)
  public void removeEdge(Edge edge)
}
```

The Directed Graph ADT

What can we do with a vertex?

The Directed Graph ADT

What can we do with a vertex?

- What is the vertex's label?
- What edges are incident on the vertex?
- What edges is the vertex the origin of?
- What edges is the vertex the destination of?
- Is there an edge to another vertex?

The Directed Graph ADT

```
interface Vertex<V, E> {
  public V getLabel();
  public Iterator<Edge> getOutEdges();
  public Iterator<Edge> getInEdges();
  public Iterator<Edge> getIncidentEdges();
  public boolean hasEdgeTo(Vertex v);
}
```

The Directed Graph ADT

What can we do with an edge?

The Directed Graph ADT

What can we do with an edge?

- What is the edge's label?
- What is the edge's origin?
- What is the edge's destination?

The Directed Graph ADT

```
interface Edge<V, E> {
  public E getLabel();
  public Vertex getOrigin();
  public Vertex getDestination();
}
```

Graph ADT Guarantees

What guarantees does the ADT provide?

Graph ADT Guarantees

What guarantees does the ADT provide?

- The origin and destination of each edge are in the graph.

Graph Data Structures

What do we need to store for a graph (V, E)?

Graph Data Structures

What do we need to store for a graph $((V, E))$?

- A collection of vertices
- A collection of edges

Edge List

```
class EdgeList<V, E> implements Graph<V, E>
{
  List<Vertex> vertices = new ArrayList<Vertex>();
  List<Edge>   edges    = new ArrayList<Edge>();

  /*...*/
}
```

addVertex

addVertex

```
public Vertex addVertex(V label)
{
  Vertex v = new Vertex(label);
  vertices.append(v);
  return v;
}
```

addVertex

```
public Vertex addVertex(V label)
{
  Vertex v = new Vertex(label);
  vertices.append(v);
  return v;
}
```

What's the complexity?

addVertex

```
public Vertex addVertex(V label)
{
  Vertex v = new Vertex(label);
  vertices.append(v);
  return v;
}
```

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

addEdge

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  Edge e = new Edge(origin, dest, label);
  edges.append(e);
  return e;
}
```

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  Edge e = new Edge(origin, dest, label);
  edges.append(e);
  return e;
}
```

What's the complexity?

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  Edge e = new Edge(origin, dest, label);
  edges.append(e);
  return e;
}
```

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

removeEdge

removeEdge

```
public void removeEdge(Edge edge)
{
  Iterator<Edge> i = edges.iterator();
  while(i.hasNext())
  {
    if(i.next().equals(edge)){
      i.remove();
      return;
    }
  }
}
```

removeEdge

```
public void removeEdge(Edge edge)
{
  Iterator<Edge> i = edges.iterator();
  while(i.hasNext())
  {
    if(i.next().equals(edge)){
      i.remove();
      return;
    }
  }
}
```

What's the complexity?

removeEdge

```
public void removeEdge(Edge edge)
{
  Iterator<Edge> i = edges.iterator();
  while(i.hasNext())
  {
    if(i.next().equals(edge)){
      i.remove();
      return;
    }
  }
}
```

What's the complexity? $O(M)$

Edge List

Could we do better?

Edge List

```
class EdgeListV2<V, E> implements Graph<V, E>
{
  List<Vertex> vertices = new LinkedList<Vertex>();
  List<Edge>   edges    = new LinkedList<Edge>();

  /*...*/
}
```

removeEdge

removeEdge

```
public void removeEdge(Edge edge)
{
  Iterator<Edge> i = edges.iterator();
  while(i.hasNext())
  {
    if(i.next().equals(edge)){
      i.remove();
      return;
    }
  }
}
```

removeEdge

```
public void removeEdge(Edge edge)
{
  Iterator<Edge> i = edges.iterator();
  while(i.hasNext())
  {
    if(i.next().equals(edge)){
      i.remove();
      return;
    }
  }
}
```

What's the complexity?

removeEdge

```java
public void removeEdge(Edge edge)
{
  Iterator<Edge> i = edges.iterator();
  while(i.hasNext())
  {
    if(i.next().equals(edge)){
      i.remove();
      return;
    }
  }
}
```

What's the complexity? $O(M)$

removeEdge

```
public void removeEdge(Edge edge)
{
  Iterator<Edge> i = edges.iterator();
  while(i.hasNext())
  {
    if(i.next().equals(edge)){
      i.remove();
      return;
    }
  }
}
```

What's the complexity? $O(M)$

Problem: Finding the edge index/node is expensive

Edge List

Idea: Store a reference to the linked list node

Edge List

```
class EdgeListV2<V, E> implements Graph<V, E>
{
  List<Vertex> vertices = new BetterLinkedList<Vertex>();
  List<Edge>   edges    = new BetterLinkedList<Edge>();

  /*...*/
}
```

Edge List

```
class BetterLinkedList<T> implements List<T>
{
  public Node<T> add(T element);
    /* O(1) with tail pointer */

  public void remove(Node<T> node)
    /* O(1) */

  public ListIterator<T> iterator
    /* O(1) + O(1) per call to next */

  /*...*/
}
```

addVertex

```
class Vertex<V, E>
{
  Node<Vertex> node = null;
  /*...*/
}
```

```
public Vertex addVertex(V label)
{
  Vertex vertex = new Vertex(label);
  Node<Vertex> node = vertices.append(vertex);
  vertex.node = node;
  return vertex;
}
```

addVertex

```
class Vertex<V, E>
{
  Node<Vertex> node = null;
  /*...*/
}
```

```
public Vertex addVertex(V label)
{
  Vertex vertex = new Vertex(label);
  Node<Vertex> node = vertices.append(vertex);
  vertex.node = node;                          ←
  return vertex;
}
```

What's the complexity?

addVertex

```
class Vertex<V, E>
{
  Node<Vertex> node = null;
  /*...*/
}
```

```
public Vertex addVertex(V label)
{
  Vertex vertex = new Vertex(label);
  Node<Vertex> node = vertices.append(vertex);
  vertex.node = node;
  return vertex;
}
```

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

addEdge

```
class Edge<V, E>
{
  Node<Edge> node = null;
  /*...*/
}
```

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  Edge edge = new Edge(origin, dest, label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;            ←
  return edge;
}
```

addEdge

```
class Edge<V, E>
{
  Node<Edge> node = null;
  /*...*/
}
```

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  Edge edge = new Edge(origin, dest, label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;                          ←
  return edge;
}
```

What's the complexity?

addEdge

```
class Edge<V, E>
{
  Node<Edge> node = null;
  /*...*/
}
```

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  Edge edge = new Edge(origin, dest, label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;                              ←
  return edge;
}
```

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

removeEdge

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  edge.node = null;
}
```

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  edge.node = null;
}
```

What's the complexity?

removeEdge

```java
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  edge.node = null;
}
```

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

removeVertex

removeVertex

```
public void removeVertex(Vertex vertex)
{
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

removeVertex

```
public void removeVertex(Vertex vertex)
{
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

What's the complexity?

removeVertex

```
public void removeVertex(Vertex vertex)
{
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

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

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

- `addVertex`:

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

- `addVertex`: No violations possible

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

- `addVertex`: No violations possible
- `addEdge`:

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

- `addVertex`: No violations possible
- `addEdge`: The edge's incident vertices might not be in the graph

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

- `addVertex`: No violations possible
- `addEdge`: The edge's incident vertices might not be in the graph
- `removeEdge`:

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

- `addVertex`: No violations possible
- `addEdge`: The edge's incident vertices might not be in the graph
- `removeEdge`: No violations possible

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?

Which operations might violate this guarantee?

- `addVertex`: No violations possible
- `addEdge`: The edge's incident vertices might not be in the graph
- `removeEdge`: No violations possible
- `removeVertex`:

Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?
Which operations might violate this guarantee?

- `addVertex`: No violations possible
- `addEdge`: The edge's incident vertices might not be in the graph
- `removeEdge`: No violations possible
- `removeVertex`: The vertex might be incident on an edge.

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);    ←
  assert(dest.node != null);      ←
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  return edge;
}
```

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);        ←
  assert(dest.node != null);          ←
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  return edge;
}
```

What's the complexity?

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);
  assert(dest.node != null);
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  return edge;
}
```

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

removeVertex

removeVertex

```
public void removeVertex(Vertex vertex)
{
  for(edge : vertex.getIncidentEdges)
  {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

removeVertex

```
public void removeVertex(Vertex vertex)
{
  for(edge : vertex.getIncidentEdges)
  {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

What's the complexity?

removeVertex

```
public void removeVertex(Vertex vertex)
{
  for(edge : vertex.getIncidentEdges)
  {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

What's the complexity?

$O(|\text{deg}(\texttt{vertex}) + T_{\texttt{getIncidentEdges}}(N, M))$

getIncidentEdges

getIncidentEdges

```
public Iterator<Edge> getIncidentEdges(Vertex vertex)
{
  ArrayList<Edge> incidentEdges = new ArrayList<>();
  for(edge : edges)
  {
    if(edge.origin.equals(vertex)
       edge.dest.equals(vertex))
    {
      incidentEdges.add(edge);
    }
  }
  return incidentEdges.iterator();
}
```

getIncidentEdges

```
public Iterator<Edge> getIncidentEdges(Vertex vertex)
{
  ArrayList<Edge> incidentEdges = new ArrayList<>();
  for(edge : edges)
  {
    if(edge.origin.equals(vertex)
       edge.dest.equals(vertex))
    {
      incidentEdges.add(edge);
    }
  }
  return incidentEdges.iterator();
}
```

What's the complexity?

getIncidentEdges

```
public Iterator<Edge> getIncidentEdges(Vertex vertex)
{
  ArrayList<Edge> incidentEdges = new ArrayList<>();
  for(edge : edges)
  {
    if(edge.origin.equals(vertex)
       edge.dest.equals(vertex))
    {
      incidentEdges.add(edge);
    }
  }
  return incidentEdges.iterator();
}
```

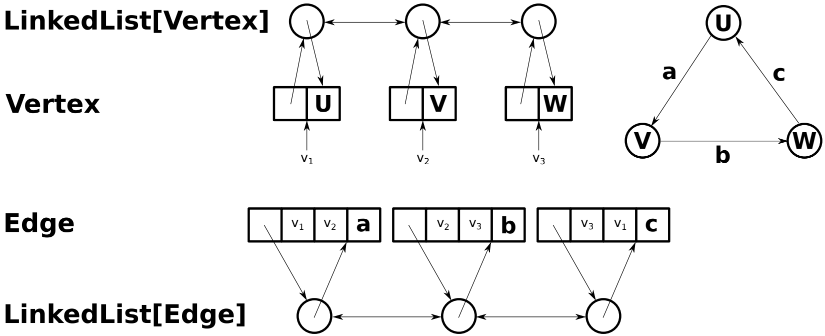
What's the complexity? $O(M)$

getIncidentEdges

```
public Iterator<Edge> getIncidentEdges(Vertex vertex)
{
  ArrayList<Edge> incidentEdges = new ArrayList<>();
  for(edge : edges)
  {
    if(edge.origin.equals(vertex)
       edge.dest.equals(vertex))
    {
      incidentEdges.add(edge);
    }
  }
  return incidentEdges.iterator();
}
```

What's the complexity? $O(M) = O(N^2)$

Edge List Summary



Edge List Summary

- `addEdge`, `addVertex`:
- `removeEdge`:
- `removeVertex`:
- `incidentEdges`:

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`:
- `removeVertex`:
- `incidentEdges`:

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`:
- `incidentEdges`:

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`:
- `incidentEdges`: $O(M)$

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`:

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

Space Used:

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

Space Used: $O(N + M)$
(constant space per vertex, edge)

Improving on the Edge List

How can we avoid searching every edge in the edge list to find the incident edges?

Improving on the Edge List

How can we avoid searching every edge in the edge list to find the incident edges?

Idea: Store each edges in/out edge list.

Adjacency List

```
public class Vertex<V, E>
{
  Node<Vertex> node = null;
  List<Edge> inEdges = new BetterLinkedList<Edge>();
  List<Edge> outEdges = new BetterLinkedList<Edge>();
  /*...*/
}
```

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);
  assert(dest.node != null);
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  origin.outEdges.add(edge);        ←
  dest.inEdges.add(edge);           ←
  return edge;
}
```

addEdge

```java
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);
  assert(dest.node != null);
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  origin.outEdges.add(edge);   ←
  dest.inEdges.add(edge);      ←
  return edge;
}
```

What's the complexity?

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);
  assert(dest.node != null);
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  origin.outEdges.add(edge);          ←
  dest.inEdges.add(edge);             ←
  return edge;
}
```

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

removeEdge

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  for(iter : edge.orig.outEdges)
  {
    if(iter.next().equals(edge)){ iter.remove(); }
  }
  for(iter : edge.dest.inEdges)
  {
    if(iter.next().equals(edge)){ iter.remove(); }
  }
  edge.node = null;
}
```

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  for(iter : edge.orig.outEdges)
  {
    if(iter.next().equals(edge)){ iter.remove(); }
  }
  for(iter : edge.dest.inEdges)
  {
    if(iter.next().equals(edge)){ iter.remove(); }
  }
  edge.node = null;
}
```

What's the complexity?

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  for(iter : edge.orig.outEdges)
  {
    if(iter.next().equals(edge)){ iter.remove(); }
  }
  for(iter : edge.dest.inEdges)
  {
    if(iter.next().equals(edge)){ iter.remove(); }
  }
  edge.node = null;
}
```

What's the complexity? $O(M)$

removeEdge

Idea: Store the outEdge/inEdge node with the Edge.

removeEdge

Idea: Store the outEdge/inEdge node with the Edge.

```
public class Edge<V, E>
{
  Node<Edge> node = null;
  Node<Edge> inNode = null;
  Node<Edge> outNode = null;
  /*...*/
}
```

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);
  assert(dest.node != null);
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  edge.outNode = origin.outEdges.add(edge);    ←
  edge.inNode = dest.inEdges.add(edge);        ←
  return edge;
}
```

addEdge

```
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);
  assert(dest.node != null);
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  edge.outNode = origin.outEdges.add(edge);
  edge.inNode = dest.inEdges.add(edge);
  return edge;
}
```

What's the complexity?

addEdge

```java
public Edge addEdge(Vertex origin, Vertex dest, E label)
{
  assert(origin.node != null);
  assert(dest.node != null);
  Edge edge = new Edge(label);
  Node<Edge> node = edges.append(edge);
  edge.node = node;
  edge.outNode = origin.outEdges.add(edge);
  edge.inNode = dest.inEdges.add(edge);
  return edge;
}
```

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

removeEdge

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  edge.orig.outEdges.remove(edge.outNode);
  edge.dest.inEdges.remove(edge.inNode);
  edge.node = null;
  edge.inNode = null;
  edge.outNode = null;
}
```

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  edge.orig.outEdges.remove(edge.outNode);
  edge.dest.inEdges.remove(edge.inNode);
  edge.node = null;
  edge.inNode = null;
  edge.outNode = null;
}
```

What's the complexity?

removeEdge

```
public void removeEdge(Edge edge)
{
  edges.remove(edge.node);
  edge.orig.outEdges.remove(edge.outNode);
  edge.dest.inEdges.remove(edge.inNode);
  edge.node = null;
  edge.inNode = null;
  edge.outNode = null;
}
```

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

removeVertex

removeVertex

```
public void removeVertex(Vertex vertex)
{
  for(edge : vertex.inEdges)
  {
    removeEdge(edge.node)
  }
  for(edge : vertex.outEdges)
  {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

removeVertex

```
public void removeVertex(Vertex vertex)
{
  for(edge : vertex.inEdges)
  {
    removeEdge(edge.node)
  }
  for(edge : vertex.outEdges)
  {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

What's the complexity?

removeVertex

```
public void removeVertex(Vertex vertex)
{
  for(edge : vertex.inEdges)
  {
    removeEdge(edge.node)
  }
  for(edge : vertex.outEdges)
  {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
  vertex.node = null;
}
```

What's the complexity? $O(\texttt{deg}(\texttt{vertex}))$

Adjacency List Summary

Starting with an edge list:

- Store a linked list of in/out edges with each vertex
- Store the linked list node for the in/out lists with each edge

Adjacency List Summary

- `addEdge`, `addVertex`:
- `removeEdge`:
- `removeVertex`:
- `incidentEdges`:
- `hasEdgeTo`:

Adjacency List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`:
- `removeVertex`:
- `incidentEdges`:
- `hasEdgeTo`:

Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex:
- incidentEdges:
- hasEdgeTo:

Adjacency List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`:
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`:

Adjacency List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\deg(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`:

Adjacency List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\deg(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`: $O(\deg(v))$

Adjacency List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\deg(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`: $O(\deg(v))$

Space Used:

Adjacency List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\texttt{deg}(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`: $O(\texttt{deg}(v))$

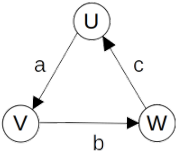
Space Used: $O(N + M)$
(constant space per vertex, edge)

hasEdgeTo

Can we cut `hasEdgeTo` down to $O(1)$?

The Adjacency Matrix Data Structure

Origin \ Destination	U	V	W
U	-	a	-
V	-	-	b
W	c	-	-



Adjacency Matrix Summary

- `addEdge`, `removeEdge`:
- `addVertex`, `removeVertex`:
- `incidentEdges`:
- `hasEdgeTo`:

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`:
- `incidentEdges`:
- `hasEdgeTo`:

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`:
- `hasEdgeTo`:

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`:

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used:

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used:

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used:

Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used: $O(N^2)$