Graphs – ADTs and Implementations
Applications of Graphs

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram
Outcomes

- By understanding this lecture, you should be able to:
  - Define basic terminology of graphs.
  - Use a graph ADT for appropriate applications.
  - Program standard implementations of the graph ADT.
  - Understand advantages and disadvantages of these implementations, in terms of space and run time.
Outline

- Definitions
- Graph ADT
- Implementations
Outline

- Definitions
- Graph ADT
- Implementations
Edge Types

- **Directed edge**
  - ordered pair of vertices \((u,v)\)
  - first vertex \(u\) is the origin
  - second vertex \(v\) is the destination
  - e.g., a flight

- **Undirected edge**
  - unordered pair of vertices \((u,v)\)
  - e.g., a flight route

- **Directed graph (Digraph)**
  - all the edges are directed
  - e.g., route network

- **Undirected graph**
  - all the edges are undirected
  - e.g., flight network
Vertices and Edges

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a

- Edges incident on a vertex
  - a, d, and b are incident on V

- Adjacent vertices
  - U and V are adjacent

- Degree of a vertex
  - X has degree 5

- Parallel edges
  - h and i are parallel edges

- Self-loop
  - j is a self-loop
Graphs

- A graph is a pair \((V, E)\), where
  - \(V\) is a set of nodes, called vertices
  - \(E\) is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements

- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route
Paths

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

- **Simple path**
  - path such that all its vertices and edges are distinct

- **Examples**
  - $P_1=(V,b,X,h,Z)$ is a simple path
  - $P_2=(U,c,W,e,X,g,Y,f,W,d,V)$ is a path that is not simple
Cycles

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints

- **Simple cycle**
  - cycle such that all its vertices and edges are distinct

- **Examples**
  - $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ is a simple cycle
  - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ is a cycle that is not simple
A subgraph $S$ of a graph $G$ is a graph such that
- The vertices of $S$ are a subset of the vertices of $G$
- The edges of $S$ are a subset of the edges of $G$

A spanning subgraph of $G$ is a subgraph that contains all the vertices of $G$
Connectivity

- A graph is connected if there is a path between every pair of vertices.
- A connected component of a graph $G$ is a maximal connected subgraph of $G$.

Connected graph

Non-connected graph with two connected components
Trees

A tree is a connected, acyclic, undirected graph.

A forest is a set of trees (not necessarily connected)
Spanning Trees

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest
Reachability in Directed Graphs

A node $w$ is **reachable** from $v$ if there is a directed path originating at $v$ and terminating at $w$.

- E is reachable from B
- B is not reachable from E
Properties

Property 1

\[ \sum_v \deg(v) = 2|E| \]

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

\[ |E| \leq |V| (|V| - 1)/2 \]

Proof: each vertex has degree at most \(|V| - 1\)

Q: What is the bound for a digraph?

A: \(|E| \leq |V|(|V| - 1)\)
Outline

- Definitions
- Graph ADT
- Implementations
Main Methods of the Graph ADT

- **Accessor methods**
  - `numVertices()`: Returns the number of vertices in the graph
  - `numEdges()`: Returns the number of vertices in the graph
  - `getEdge(u, v)`: Returns edge from u to v
  - `endVertices(e)`: an array of the two endvertices of e
  - `opposite(v, e)`: the vertex opposite to v on e
  - `outDegree(v)`: Returns number of outgoing edges
  - `inDegree(v)`: Returns number of incoming edges
Main Methods of the Graph ADT

- **Update methods**
  - `insertVertex(x)`: insert a vertex storing element x
  - `insertEdge(u, v, x)`: insert an edge (u,v) storing element x
  - `removeVertex(v)`: remove vertex v (and its incident edges)
  - `removeEdge(e)`: remove edge e
Main Methods of the Graph ADT

- **Iterator methods**
  - `incomingEdges(v)`: Incoming edges to v
  - `outgoingEdges(v)`: Outgoing edges from v
  - `vertices()`: all vertices in the graph
  - `edges()`: all edges in the graph
Outline

- Definitions
- Graph ADT
- Implementations
GTG Implementation (net.datastructures)

- There are many ways to implement the Graph ADT.
- We will follow the textbook implementation.
A graph consists of a collection of vertices V and a collection of edges E.

Each of these will be represented as a Positional List (Ch.7.3).

In net.datastructures, Positional Lists are implemented as doubly-linked lists.
Vertices and Edges

- Each vertex $v$ stores an element containing information about the vertex.
  - For example, if the graph represents course dependencies, the vertex element might store the course number.

- Each edge $e$ stores an element containing information about the edge.
  - e.g., pre-requisite, co-requisite.

- In addition, each edge must store references to the vertices it connects.

![Diagram of vertex and edge relationships]
Vertices and Edges

To facilitate efficient removal of vertices and edges, we will make both location aware:

- A reference to the Position in the Positional List will be stored in the element.

```
Vertex Position / Node
prev  next
```

```
Edge Position / Node
prev  next
```

```
Vertex   u
```

```
Edge     e
```
Edge List Implementation

- This organization yields an Edge List Structure

[Diagram showing vertex and edge lists]
Performance of Edge List Implementation

- Edge List implementation does not provide efficient access to edge information from vertex list.

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$n + m$</td>
</tr>
<tr>
<td><code>incomingEdges(v)</code></td>
<td>$m$</td>
</tr>
<tr>
<td><code>outgoingEdges(v)</code></td>
<td>$m$</td>
</tr>
<tr>
<td><code>getEdge(u, v)</code></td>
<td>$m$</td>
</tr>
<tr>
<td><code>insertVertex(x)</code></td>
<td>1</td>
</tr>
<tr>
<td><code>insertEdge(u, v, x)</code></td>
<td>1</td>
</tr>
<tr>
<td><code>removeVertex(v)</code></td>
<td>$m$</td>
</tr>
<tr>
<td><code>removeEdge(e)</code></td>
<td>1</td>
</tr>
</tbody>
</table>
Other Graph Implementations

- Can we come up with a graph implementation that improves the efficiency of these basic operations?
  - Adjacency List
  - Adjacency Map
  - Adjacency Matrix
Other Graph Implementations

Can we come up with a graph implementation that improves the efficiency of these basic operations?

- Adjacency List
- Adjacency Map
- Adjacency Matrix
Adjacency List Implementation

- An Adjacency List implementation augments each vertex element with Positional Lists of incoming and outgoing edges.
An Adjacency List implementation augments each vertex element with lists of incoming and outgoing edges.
Performance of Adjacency List Implementation

- Adjacency List implementation improves efficiency without increasing space requirements.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Edge List</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$n + m$</td>
<td>$n + m$</td>
</tr>
<tr>
<td>incomingEdges($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
</tr>
<tr>
<td>outgoingEdges($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
</tr>
<tr>
<td>getEdge($u, v$)</td>
<td>$m$</td>
<td>$\min(\text{deg}(u), \text{deg}(v))$</td>
</tr>
<tr>
<td>insertVertex($x$)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insertEdge($u, v, x$)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
</tr>
<tr>
<td>removeEdge($e$)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Other Graph Implementations

Can we come up with a graph implementation that improves the efficiency of these basic operations?

- Adjacency List
- Adjacency Map
- Adjacency Matrix
Adjacency Map Implementation

- An Adjacency Map implementation augments each vertex element with an Adjacency Map of edges
  - Each entry consists of:
    - Key = opposite vertex
    - Value = edge
  - Implemented as a hash table.

![Graph diagram]

![Adjacency Maps]

Vertex List

Adjacency Maps
Performance of Adjacency Map Implementation

- Adjacency Map implementation improves expected run time of `getEdge(u, v)`:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n + m$</td>
</tr>
<tr>
<td>incomingEdges($v$), outgoingEdges($v$)</td>
<td>$m$</td>
<td>deg($v$)</td>
<td>deg($v$)</td>
</tr>
<tr>
<td>getEdge($u, v$)</td>
<td>$m$</td>
<td>min(deg($u$), deg($v$))</td>
<td>1 (exp.)</td>
</tr>
<tr>
<td>insertVertex($x$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insertEdge($u, v, x$)</td>
<td>1</td>
<td>1</td>
<td>1 (exp.)</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$m$</td>
<td>deg($v$)</td>
<td>deg($v$)</td>
</tr>
<tr>
<td>removeEdge($e$)</td>
<td>1</td>
<td>1</td>
<td>1 (exp.)</td>
</tr>
</tbody>
</table>
Other Graph Implementations

- Can we come up with a graph implementation that improves the efficiency of these basic operations?
  - Adjacency List
  - Adjacency Map
  - **Adjacency Matrix**
Adjacency Matrix Implementation

- In an Adjacency Matrix implementation we map each of the $n$ vertices to an integer index from $[0 \ldots n-1]$.

- Then a 2D $n \times n$ array $A$ is maintained:
  - If edge $(i, j)$ exists, $A[i, j]$ stores a reference to the edge.
  - If edge $(i, j)$ does not exist, $A[i, j]$ is set to null.
Adjacency Matrix Structure
Performance of Adjacency Matrix Implementation

- Requires more space.
- Slow to get incoming / outgoing edges
- Very slow to insert or remove a vertex (array must be resized)

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Map</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>incomingEdges($v$), outgoingEdges($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td>$\text{deg}(v)$</td>
<td>$n$</td>
</tr>
<tr>
<td>getEdge($u$, $v$)</td>
<td>$m$</td>
<td>$\min(\text{deg}(u), \text{deg}(v))$</td>
<td>1 (exp.)</td>
<td>1</td>
</tr>
<tr>
<td>insertVertex($x$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$n^2$</td>
</tr>
<tr>
<td>insertEdge($u$, $v$, $x$)</td>
<td>1</td>
<td>1</td>
<td>1 (exp.)</td>
<td>1</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td>$\text{deg}(v)$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>removeEdge($e$)</td>
<td>1</td>
<td>1</td>
<td>1 (exp.)</td>
<td>1</td>
</tr>
</tbody>
</table>
A4Q2: Course Prerequisites

- In most post-secondary programs, courses have prerequisites.
- For example, you cannot take EECS 3101 until you have passed EECS 2011.
- How can we represent such a system of dependencies?
- A natural choice is a **directed graph**.

- Each vertex represents a course
- Each directed edge represents a prerequisite

- A directed edge from Course U to Course V means that Course U must be taken before Course V.
A4Q2: Course Prerequisites

- We also want to be able to find the information for a particular course quickly.

- The course number provides a convenient key that can be used to organize course records in a sorted map, implemented as a binary search tree (cf. A3Q1).

- Thus it makes sense to represent courses using both a sorted map (for efficient access) and a directed graph (to represent dependencies).

- By storing a reference to the directed graph vertex for a course in the sorted map, we can efficiently access course dependencies.
A4Q2: Course Prerequisites

Key: 2011
Value:
- Number: 2011
- Name: “Data Structures”
- Vertex:

(K₁, V₁)
(K₂, V₂)
(K₃, V₃)
A4Q2: Course Prerequisites

It is important that the course prerequisite graph be a directed acyclic graph (DAG). Why?
In this question, you are provided with a basic implementation of a system to represent courses and dependencies.

Methods for adding courses and getting prerequisites are provided.

You need only write the method for adding a prerequisite.

This method will use a depth-first-search algorithm (also provided) that can be used to prevent the addition of prerequisites that introduce cycles.
A4Q2: Implementation using net.datastructures

- We use the TreeMap class to represent the sorted map (cf. A3Q1).

Key: 2011
Value:
- Number: 2011
- Name: “Data Structures”
- Vertex:

Sorted Map

(K1, V1)
(K2, V2)
(K3, V3)
A4Q2: Implementation using net.datastructures

- We use the **AdjacencyMapGraph** class to represent the directed graph.
- This implementation uses **ProbeHashMap**, a linear probe hash table, to represent the incoming and outgoing edges for each vertex.
Outline

- Definitions
- Graph ADT
- Implementations
Outcomes

By understanding this lecture, you should be able to:

- Define basic terminology of graphs.
- Use a graph ADT for appropriate applications.
- Program standard implementations of the graph ADT.
- Understand advantages and disadvantages of these implementations, in terms of space and run time.