Graphs

- Definitions
- The Graph ADT
- Data structures for graphs
What is a Graph?

- A graph $G = (V,E)$ is composed of:
  - $V$: set of vertices
  - $E$: set of edges connecting the vertices in $V$
- An edge $e = (u,v)$ is a pair of vertices
- Example:

  \[ V = \{a, b, c, d, e\} \]
  \[ E = \{(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)\} \]
Applications

• electronic circuits

find the path of least resistance to CS16

• networks (roads, flights, communications)
mo’ better examples
A Spike Lee Joint Production

- scheduling (project planning)

A typical student day

wake up → cs16 meditation → work → more cs16 → play → cs16 program → cxhextris → make cookies for cs16 HTA → sleep → dream of cs16 → eat
Graph Terminology

- **adjacent vertices**: connected by an edge
- **degree (of a vertex)**: # of adjacent vertices

\[ \sum_{v \in V} \deg(v) = 2(\# \text{ edges}) \]

- Since adjacent vertices each count the adjoining edge, it will be counted twice

**path**: sequence of vertices \( v_1, v_2, \ldots, v_k \) such that consecutive vertices \( v_i \) and \( v_{i+1} \) are adjacent.
More Graph Terminology

- **simple path:** no repeated vertices

![Graph Example](image)

- **cycle:** simple path, except that the last vertex is the same as the first vertex

![Cycle Example](image)
Even More Terminology

- **connected graph**: any two vertices are connected by some path

- **subgraph**: subset of vertices and edges forming a graph

- **connected component**: maximal connected subgraph. E.g., the graph below has 3 connected components.
¡Caramba! Another Terminology Slide!

- (free) tree - connected graph without cycles
- forest - collection of trees
Connectivity

Let \( n = \# \text{vertices} \)
\( m = \# \text{edges} \)

- **complete graph** - all pairs of vertices are adjacent

\[
m = \frac{1}{2} \sum_{v \in V} \deg(v) = \frac{1}{2} \sum_{v \in V} (n - 1) = \frac{n(n-1)}{2}
\]

• Each of the \( n \) vertices is incident to \( n - 1 \) edges, however, we would have counted each edge twice!!! Therefore, intuitively, \( m = \frac{n(n-1)}{2} \).

\[
\begin{align*}
n &= 5 \\
m &= \frac{(5 \times 4)}{2} = 10
\end{align*}
\]

• Therefore, if a graph is *not* complete, \( m < \frac{n(n-1)}{2} \)
More Connectivity

\[ n = \#\text{vertices} \]
\[ m = \#\text{edges} \]

- For a tree \( m = n - 1 \)

\[
\begin{aligned}
\text{Graph 1:} & \quad n = 5 \\
\text{Graph 2:} & \quad m = 4
\end{aligned}
\]

- If \( m < n - 1 \), \( G \) is not connected

\[
\begin{aligned}
\text{Graph 3:} & \quad n = 5 \\
\text{Graph 4:} & \quad m = 3
\end{aligned}
\]
Spanning Tree

- A **spanning tree** of $G$ is a subgraph which
  - is a tree
  - contains all vertices of $G$

- Failure on any edge disconnects system (least fault tolerant)
AT&T vs. RT&T

(Roberto Tamassia & Telephone)

• Roberto wants to call the TA’s to suggest an extension for the next program...

But Plant-Ops ‘accidentally’ cuts a phone cable!!

• One fault will disconnect part of graph!!

• A cycle would be more fault tolerant and only requires $n$ edges
Euler and the Bridges of Koenigsberg

Can one walk across each bridge exactly once and return at the starting point?

- Consider if you were a UPS driver, and you didn’t want to retrace your steps.
- In 1736, Euler proved that this is not possible
Graph Model (with parallel edges)

- Eulerian Tour: path that traverses every edge exactly once and returns to the first vertex
- Euler’s Theorem: A graph has a Eulerian Tour if and only if all vertices have even degree
- Do you find such ideas interesting?
- Would you enjoy spending a whole semester doing such proofs?

Well, look into CS22! if you dare...
The Graph ADT

- The **Graph ADT** is a positional container whose positions are the vertices and the edges of the graph.

  - `size()` Return the number of vertices plus the number of edges of $G$.
  - `isEmpty()`
  - `elements()`
  - `positions()`
  - `swap()`
  - `replaceElement()`

Notation: Graph $G$; Vertices $v$, $w$; Edge $e$; Object $o$

  - `numVertices()` Return the number of vertices of $G$.
  - `numEdges()` Return the number of edges of $G$.
  - `vertices()` Return an enumeration of the vertices of $G$.
  - `edges()` Return an enumeration of the edges of $G$. 
The Graph ADT (contd.)

- directedEdges()
  Return an enumeration of all directed edges in $G$.

- undirectedEdges()
  Return an enumeration of all undirected edges in $G$.

- incidentEdges($v$)
  Return an enumeration of all edges incident on $v$.

- inIncidentEdges($v$)
  Return an enumeration of all the incoming edges to $v$.

- outIncidentEdges($v$)
  Return an enumeration of all the outgoing edges from $v$.

- opposite($v$, $e$)
  Return an endpoint of $e$ distinct from $v$.

- degree($v$)
  Return the degree of $v$.

- inDegree($v$)
  Return the in-degree of $v$.

- outDegree($v$)
  Return the out-degree of $v$. 
- adjacentVertices($v$)
  Return an enumeration of the vertices adjacent to $v$.

- inAdjacentVertices($v$)
  Return an enumeration of the vertices adjacent to $v$ along incoming edges.

- outAdjacentVertices($v$)
  Return an enumeration of the vertices adjacent to $v$ along outgoing edges.

- areAdjacent($v,w$)
  Return whether vertices $v$ and $w$ are adjacent.

- endVertices($e$)
  Return an array of size 2 storing the end vertices of $e$.

- origin($e$)
  Return the end vertex from which $e$ leaves.

- destination($e$)
  Return the end vertex at which $e$ arrives.

- isDirected($e$)
  Return true iff $e$ is directed.
Update Methods

- **makeUndirected**(\(e\))
  Set \(e\) to be an undirected edge.

- **reverseDirection**(\(e\))
  Switch the origin and destination vertices of \(e\).

- **setDirectionFrom**(\(e, v\))
  Sets the direction of \(e\) away from \(v\), one of its end vertices.

- **setDirectionTo**(\(e, v\))
  Sets the direction of \(e\) toward \(v\), one of its end vertices.

- **insertEdge**(\(v, w, o\))
  Insert and return an undirected edge between \(v\) and \(w\), storing \(o\) at this position.

- **insertDirectedEdge**(\(v, w, o\))
  Insert and return a directed edge between \(v\) and \(w\), storing \(o\) at this position.

- **insertVertex**(\(o\))
  Insert and return a new (isolated) vertex storing \(o\) at this position.

- **removeEdge**(\(e\))
  Remove edge \(e\).
Data Structures for Graphs

• A Graph! How can we represent it?
• To start with, we store the vertices and the edges into two containers, and we store with each edge object references to its endvertices.

• Additional structures can be used to perform efficiently the methods of the Graph ADT.
Edge List

- The **edge list** structure simply stores the vertices and the edges into unsorted sequences.

- Easy to implement.

- Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence.
## Performance of the Edge List Structure

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty, replaceElement, swap</td>
<td>O(1)</td>
</tr>
<tr>
<td>numVertices, numEdges</td>
<td>O(1)</td>
</tr>
<tr>
<td>vertices</td>
<td>O(n)</td>
</tr>
<tr>
<td>edges, directedEdges, undirectedEdges</td>
<td>O(m)</td>
</tr>
<tr>
<td>elements, positions</td>
<td>O(n+m)</td>
</tr>
<tr>
<td>endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree</td>
<td>O(1)</td>
</tr>
<tr>
<td>incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices, areAdjacent</td>
<td>O(m)</td>
</tr>
<tr>
<td>insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeVertex</td>
<td>O(m)</td>
</tr>
</tbody>
</table>
Adjacency List (traditional)

- adjacency list of a vertex v:
  sequence of vertices adjacent to v

- represent the graph by the adjacency lists of all the vertices

Space = $\Theta(N + \sum \text{deg}(v)) = \Theta(N + M)$
Adjacency List (modern)

- The **adjacency list** structure extends the edge list structure by adding **incidence containers** to each vertex.

- The space requirement is $O(n + m)$.
# Performance of the Adjacency List Structure

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty, replaceElement, swap</td>
<td>O(1)</td>
</tr>
<tr>
<td>numVertices, numEdges</td>
<td>O(1)</td>
</tr>
<tr>
<td>vertices</td>
<td>O(n)</td>
</tr>
<tr>
<td>edges, directedEdges, undirectedEdges</td>
<td>O(m)</td>
</tr>
<tr>
<td>elements, positions</td>
<td>O(n+m)</td>
</tr>
<tr>
<td>endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree</td>
<td>O(1)</td>
</tr>
<tr>
<td>incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v)</td>
<td>O(deg(v))</td>
</tr>
<tr>
<td>areAdjacent(u, v)</td>
<td>O(min(deg(u), deg(v)))</td>
</tr>
<tr>
<td>insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>O(deg(v))</td>
</tr>
</tbody>
</table>
Adjacency Matrix (traditional)

- matrix \( M \) with entries for all pairs of vertices
- \( M[i,j] = \text{true} \) means that there is an edge \((i,j)\) in the graph.
- \( M[i,j] = \text{false} \) means that there is no edge \((i,j)\) in the graph.
- There is an entry for every possible edge, therefore:
  \[
  \text{Space} = \Theta(N^2)
  \]
## Adjacency Matrix (modern)

- The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ø</td>
<td>Ø</td>
<td>NW 35</td>
<td>Ø</td>
<td>DL 247</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>AA 49</td>
<td>Ø</td>
<td>DL 335</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>Ø</td>
<td>AA 1387</td>
<td>Ø</td>
<td>Ø</td>
<td>AA 903</td>
<td>Ø</td>
<td>TW 45</td>
</tr>
<tr>
<td>3</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>UA 120</td>
<td>Ø</td>
</tr>
<tr>
<td>4</td>
<td>Ø</td>
<td>AA 523</td>
<td>Ø</td>
<td>AA 411</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>5</td>
<td>Ø</td>
<td>UA 877</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>6</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

- The space requirement is $O(n^2 + m)$
## Performance of the Adjacency Matrix Structure

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size</code>, <code>isEmpty</code>, <code>replaceElement</code>, <code>swap</code></td>
<td><strong>O(1)</strong></td>
</tr>
<tr>
<td><code>numVertices</code>, <code>numEdges</code></td>
<td><strong>O(1)</strong></td>
</tr>
<tr>
<td><code>vertices</code></td>
<td><strong>O(n)</strong></td>
</tr>
<tr>
<td><code>edges</code>, <code>directedEdges</code>, <code>undirectedEdges</code></td>
<td><strong>O(m)</strong></td>
</tr>
<tr>
<td><code>elements</code>, <code>positions</code></td>
<td><strong>O(n+m)</strong></td>
</tr>
<tr>
<td><code>endVertices</code>, <code>opposite</code>, <code>origin</code>, <code>destination</code>, <code>isDirected</code>, <code>degree</code>, <code>inDegree</code>, <code>outDegree</code></td>
<td><strong>O(1)</strong></td>
</tr>
<tr>
<td><code>incidentEdges</code>, <code>inIncidentEdges</code>, <code>outIncidentEdges</code>, <code>adjacentVertices</code>, <code>inAdjacentVertices</code>, <code>outAdjacentVertices</code>, <code>areAdjacent</code></td>
<td><strong>O(n)</strong></td>
</tr>
<tr>
<td><code>insertEdge</code>, <code>insertDirectedEdge</code>, <code>removeEdge</code>, <code>makeUndirected</code>, <code>reverseDirection</code>, <code>setDirectionFrom</code>, <code>setDirectionTo</code></td>
<td><strong>O(1)</strong></td>
</tr>
<tr>
<td><code>insertVertex</code>, <code>removeVertex</code></td>
<td><strong>O(n^2)</strong></td>
</tr>
</tbody>
</table>