CHAPTER 13
GRAPH ALGORITHMS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM JORY DENNY AND MUKULIKA GHOSH
A graph is a pair $G = (V, E)$, where
- $V$ is a set of nodes, called vertices
- $E$ is a collection of pairs of vertices, called edges
- Vertices and edges can store arbitrary 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
EDGE & GRAPH TYPES

- **Edge Types**
  - Directed edge
    - ordered pair of vertices \((u, v)\)
    - first vertex \(u\) is the origin/source
    - second vertex \(v\) is the destination/target
    - e.g., a flight
  - Undirected edge
    - unordered pair of vertices \((u, v)\)
    - e.g., a flight route
  - Weighted edge

- **Graph Types**
  - Directed graph (Digraph)
    - all the edges are directed
    - e.g., route network
  - Undirected graph
    - all the edges are undirected
    - e.g., flight network
  - Weighted graph
    - all the edges are weighted
APPLICATIONS

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

- **End points** (or end vertices) 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 (multiple) edges**
  - $h$ and $i$ are parallel edges
- **Self-loop**
  - $j$ is a self-loop
- **Outgoing edges** of a vertex
  - \( h \) and \( b \) are the outgoing edges of \( X \)
- **Incoming edges** of a vertex
  - \( e, g, \) and \( i \) are incoming edges of \( X \)
- **In-degree** of a vertex
  - \( X \) has in-degree 3
- **Out-degree** of a vertex
  - \( X \) has out-degree 2
**TERMINOLOGY**

- **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
TERMINOLOGY

- **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
EXERCISE ON TERMINOLOGY

1. Number of vertices?
2. Number of edges?
3. What type of the graph is it?
4. Show the end vertices of the edge with largest weight
5. Show the vertices of smallest degree and largest degree
6. Show the edges incident to the vertices in the above question
7. Identify the shortest simple path from HNL to PVD
8. Identify the simple cycle with the most edges
EXERCISE
PROPERTIES OF UNDIRECTED GRAPHS

- Property 1 – Total degree
  \[ \Sigma_v \text{deg}(v) = ? \]

- Property 2 – Total number of edges
  - In an undirected graph with no self-loops and no multiple edges
    \[ m \leq \text{Upper Bound?} \]
    \[ \text{Lower Bound?} \leq m \]

- Notation
  - \( n \) number of vertices
  - \( m \) number of edges
  - \( \text{deg}(v) \) degree of vertex \( v \)

Example
- \( n = ? \)
- \( m = ? \)
- \( \text{deg}(v) = ? \)

A graph with given number of vertices (4) and maximum number of edges
Property 1 – Total degree

\[ \sum_v \text{deg}(v) = 2m \]

Property 2 – Total number of edges

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

\[ m \leq \frac{n(n-1)}{2} \]

\[ 0 \leq m \]

Proof: Each vertex can have degree at most \((n-1)\)

A graph with given number of vertices (4) and maximum number of edges

Example

- \(n = 4\)
- \(m = 6\)
- \(\text{deg}(v) = 3\)
EXERCISE
PROPERTIES OF DIRECTED GRAPHS

- Property 1 – Total in-degree and out-degree
  \[ \Sigma_v \text{in} - \deg(v) = \? \]
  \[ \Sigma_v \text{out} - \deg(v) = \? \]

- Property 2 – Total number of edges
  - In an directed graph with no self-loops and no multiple edges
    \[ m \leq \text{UpperBound}\? \]
    \[ \text{LowerBound}\? \leq m \]

- Notation
  - \( n \) number of vertices
  - \( m \) number of edges
  - \( \deg(v) \) degree of vertex \( v \)

Example
- \( n = \? \)
- \( m = \? \)
- \( \deg(v) = \? \)

A graph with given number of vertices (4) and maximum number of edges
EXERCISE
PROPERTIES OF DIRECTED GRAPHS

- Property 1 – Total in-degree and out-degree
  \( \sum_v \text{in} - \deg(v) = m \)
  \( \sum_v \text{out} - \deg(v) = m \)

- Property 2 – Total number of edges
  - In an directed graph with no self-loops and no multiple edges
    \( m \leq n(n - 1) \)
    \( 0 \leq m \)

- Notation
  - \( n \) number of vertices
  - \( m \) number of edges
  - \( \deg(v) \) degree of vertex \( v \)

Example
- \( n = 4 \)
- \( m = 12 \)
- \( \deg(v) = 6 \)

A graph with given number of vertices (4) and maximum number of edges
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 $G$
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
A (free) tree is an undirected graph $T$ such that
- $T$ is connected
- $T$ has no cycles
- This definition of tree is different from the one of a rooted tree

A forest is an undirected graph without cycles
The connected components of a forest are 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.
GRAPH ADT

- Vertices and edges are positions and store elements
- **Vertex ADT**
  - operator *( )
  - incidentEdges( )
  - isAdjacentTo(ν)
- **Edge ADT**
  - operator *( )
  - endVertices( )
  - opposite(ν)
  - isAdjacentTo(f)
  - isIncidentOn(ν)
  - isDirected( )
  - origin( )
  - dest( )
- **Graph ADT**
  - vertices( )
  - edges( )
  - insertVertex(χ)
  - insertEdge(ν, w, χ)
  - insertDirectedEdge(ν, w, χ)
  - eraseVertex(ν)
  - eraseEdge(e)
- Many more generic/accessor methods
- Lists of entities provide iterators
EXERCISE ON ADT

1. `ord.incidentEdges()`
2. `ord.adjacentVertices()`
3. `ord.degree()`
4. `(lga, mia).endVertices()`
5. `(dfw, lga).opposite(dfw)`
6. `dfw.isAdjacentTo(sfo)`
7. `insertVertex(iah)`
8. `insertEdge(mia, pvd, 1200)`
9. `eraseVertex(ord)`
10. `eraseEdge(dfw, ord)`
11. `(dfw, lga).isDirected()`
12. `(dfw, lga).origin()`
13. `(dfw, lga).dest()`
An edge list can be stored in a sequence, a vector, a list or a dictionary such as a hash table.

**Edge List**

- (ORD, PVD) 849
- (ORD, DFW) 802
- (LGA, PVD) 142
- (LGA, MIA) 1099
- (DFW, LGA) 1387
- (DFW, MIA) 1120

**Vertex Sequence**

- ORD
- LGA
- PVD
- DFW
- MIA
Construct the edge list for the following graph.
ASYMPTOTIC PERFORMANCE
EDGE LIST STRUCTURE

- $n$ vertices, $m$ edges
- No parallel edges
- No self-loops

<table>
<thead>
<tr>
<th>Space</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>endVertices(), opposite(), isIncidentOn($v$)</td>
<td>?</td>
</tr>
<tr>
<td>$v$.incidentEdges(), $v$.isAdjacentTo($w$)</td>
<td>?</td>
</tr>
<tr>
<td>insertVertex($x$), insertEdge($u, v, w$), eraseEdge($e$)</td>
<td>?</td>
</tr>
<tr>
<td>eraseVertex($v$)</td>
<td>?</td>
</tr>
</tbody>
</table>

### Edge List

<table>
<thead>
<tr>
<th>Edge List</th>
<th>Weight</th>
<th>Directed</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ORD, PVD)</td>
<td>849</td>
<td>False</td>
<td>ORD</td>
</tr>
<tr>
<td>(ORD, DFW)</td>
<td>802</td>
<td>False</td>
<td>LGA</td>
</tr>
<tr>
<td>(LGA, PVD)</td>
<td>142</td>
<td>False</td>
<td>PVD</td>
</tr>
<tr>
<td>(LGA, MIA)</td>
<td>1099</td>
<td>False</td>
<td>DFW</td>
</tr>
<tr>
<td>(DFW, LGA)</td>
<td>1387</td>
<td>False</td>
<td>MIA</td>
</tr>
<tr>
<td>(DFW, MIA)</td>
<td>1120</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

### Vertex Sequence

<table>
<thead>
<tr>
<th>Vertex Sequence</th>
<th>Weight</th>
<th>Directed</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>False</td>
<td>ORD</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>LGA</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>PVD</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>DFW</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>MIA</td>
<td>2</td>
</tr>
</tbody>
</table>
ASYMPTOTIC PERFORMANCE
EDGE LIST STRUCTURE

- \( n \) vertices, \( m \) edges
- No parallel edges
- No self-loops

<table>
<thead>
<tr>
<th>Method</th>
<th>Edge List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>( O(n + m) )</td>
</tr>
<tr>
<td>endVertices(), opposite(), isIncidentOn(( v ))</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( v ).incidentEdges(), ( v ).isAdjacentTo(( w ))</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>insertVertex(( x )), insertEdge(( u, v, w )), eraseEdge(( e ))</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>eraseVertex(( v ))</td>
<td>( O(m) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge List</th>
<th>Weight</th>
<th>Directed</th>
<th>Vertex Sequence</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ORD, PVD)</td>
<td>849</td>
<td>False</td>
<td>ORD</td>
<td>2</td>
</tr>
<tr>
<td>(ORD, DFW)</td>
<td>802</td>
<td>False</td>
<td>LGA</td>
<td>3</td>
</tr>
<tr>
<td>(LGA, PVD)</td>
<td>142</td>
<td>False</td>
<td>PVD</td>
<td>2</td>
</tr>
<tr>
<td>(LGA, MIA)</td>
<td>1099</td>
<td>False</td>
<td>DFW</td>
<td>3</td>
</tr>
<tr>
<td>(DFW, LGA)</td>
<td>1387</td>
<td>False</td>
<td>MIA</td>
<td>2</td>
</tr>
<tr>
<td>(DFW, MIA)</td>
<td>1120</td>
<td>False</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- **Vertex object**
  - element
  - reference to position in vertex sequence
- **Edge object**
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- **Vertex sequence**
  - sequence of vertex objects
- **Edge sequence**
  - sequence of edge objects
Adjacency Lists associate edges with their end vertices

- Each vertex stores a list of incident edges

Adjacency List

- **ORD**: (ORD, PVD), (ORD, DFW)
- **LGA**: (LGA, PVD), (LGA, MIA), (LGA, DFW)
- **PVD**: (PVD, ORD), (PVD, LGA)
- **DFW**: (DFW, ORD), (DFW, LGA), (DFW, MIA)
- **MIA**: (MIA, LGA), (MIA, DFW)
Construct the adjacency list for the following graph.

```
 a → u → v → y → z → x
```

EXERCISE
ADJACENCY LIST STRUCTURE
## Asymptotic Performance

### Adjacency List Structure

- $n$ vertices, $m$ edges
- No parallel edges
- No self-loops

<table>
<thead>
<tr>
<th>Method/Operation</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>?</td>
</tr>
<tr>
<td>endVertices(), opposite(), isIncidentOn($v$)</td>
<td>?</td>
</tr>
<tr>
<td>$v$.incidentEdges($v$), $v$.isAdjacentTo($w$)</td>
<td>?</td>
</tr>
<tr>
<td>insertVertex($x$), insertEdge($u$, $v$, $w$), eraseEdge($e$)</td>
<td>?</td>
</tr>
<tr>
<td>eraseVertex($v$)</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORD</td>
<td>(ORD, PVD) (ORD, DFW)</td>
</tr>
<tr>
<td>LGA</td>
<td>(LGA, PVD) (LGA, MIA) (LGA, DFW)</td>
</tr>
<tr>
<td>PVD</td>
<td>(PVD, ORD) (PVD, LGA)</td>
</tr>
<tr>
<td>DFW</td>
<td>(DFW, ORD) (DFW, LGA) (DFW, MIA)</td>
</tr>
<tr>
<td>MIA</td>
<td>(MIA, LGA) (MIA, DFW)</td>
</tr>
</tbody>
</table>
**Asymptotic Performance**

**Adjacency List Structure**

- \( n \) vertices, \( m \) edges
- No parallel edges
- No self-loops

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>( O(n + m) )</td>
</tr>
<tr>
<td><code>endVertices()</code></td>
<td>( O(1) )</td>
</tr>
<tr>
<td><code>opposite()</code></td>
<td>( O(1) )</td>
</tr>
<tr>
<td><code>isIncidentOn(v)</code></td>
<td>( O(1) )</td>
</tr>
<tr>
<td><code>v.incidentEdges()</code></td>
<td>( O(\deg(v)) )</td>
</tr>
<tr>
<td><code>v.isAdjacentTo(w)</code></td>
<td>( O(\min(\deg(v), \deg(w))) )</td>
</tr>
<tr>
<td><code>insertVertex(x)</code></td>
<td>( O(1) )</td>
</tr>
<tr>
<td><code>insertEdge(u, v, w)</code></td>
<td>( O(1) )</td>
</tr>
<tr>
<td><code>eraseEdge(e)</code></td>
<td>( O(\deg(v)) )</td>
</tr>
<tr>
<td><code>eraseVertex(v)</code></td>
<td>( O(\deg(v)) )</td>
</tr>
</tbody>
</table>

**Adjacency List**

- \( \text{ORD} \Rightarrow \{\text{ORD, PVD} \}, \{\text{ORD, DFW} \} \)
- \( \text{LGA} \Rightarrow \{\text{LGA, PVD} \}, \{\text{LGA, MIA} \}, \{\text{LGA, DFW} \} \)
- \( \text{PVD} \Rightarrow \{\text{PVD, ORD} \}, \{\text{PVD, LGA} \} \)
- \( \text{DFW} \Rightarrow \{\text{DFW, ORD} \}, \{\text{DFW, LGA} \}, \{\text{DFW, MIA} \} \)
- \( \text{MIA} \Rightarrow \{\text{MIA, LGA} \}, \{\text{MIA, DFW} \} \)
ADJACENCY LIST STRUCTURE

- Store vertex sequence and edge sequence
- Each vertex stores a sequence of incident edges
  - Sequence of references to edge objects of incident edges
- Augmented edge objects
  - References to associated positions in incidence sequences of end vertices
### Adjacency Matrix Structure

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Adjacency matrices store edges in a table, indexed by the vertex.
Construct the adjacency matrix for the following graph.
ADJACENCY MATRIX STRUCTURE IN A WEIGHTED GRAPH

<table>
<thead>
<tr>
<th></th>
<th>0 ORD</th>
<th>1 LGA</th>
<th>2 PVD</th>
<th>3 DFW</th>
<th>4 MIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ORD</td>
<td>0</td>
<td>0</td>
<td>849</td>
<td>802</td>
<td>0</td>
</tr>
<tr>
<td>1 LGA</td>
<td>0</td>
<td>0</td>
<td>142</td>
<td>1387</td>
<td>1099</td>
</tr>
<tr>
<td>2 PVD</td>
<td>849</td>
<td>142</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3 DFW</td>
<td>802</td>
<td>138</td>
<td>0</td>
<td>0</td>
<td>1120</td>
</tr>
<tr>
<td>4 MIA</td>
<td>0</td>
<td>1099</td>
<td>0</td>
<td>1120</td>
<td>0</td>
</tr>
</tbody>
</table>

- Store edge object/property in table, or include a pointer to it inside of the table.
EXERCISE

ADJACENCY MATRIX STRUCTURE: WEIGHTED DIGRAPH

<table>
<thead>
<tr>
<th></th>
<th>ORD</th>
<th>LGA</th>
<th>PVD</th>
<th>DFW</th>
<th>MIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image-url)
EXERCISE
ADJACENCY MATRIX STRUCTURE: WEIGHTED DIGRAPHEXERCISE
ADJACENCY MATRIX STRUCTURE: WEIGHTED DIGRAPH

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ORD</td>
<td>0</td>
<td>0</td>
<td>849</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 LGA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1387</td>
<td>1099</td>
</tr>
<tr>
<td>2 PVD</td>
<td>0</td>
<td>142</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3 DFW</td>
<td>802</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4 MIA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1120</td>
<td>0</td>
</tr>
</tbody>
</table>
**ASYMPTOTIC PERFORMANCE OF ADJACENCY MATRIX STRUCTURE**

<table>
<thead>
<tr>
<th>• $n$ vertices, $m$ edges</th>
<th><strong>Adjacency Matrix</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• No parallel edges</td>
<td></td>
</tr>
<tr>
<td>• No self-loops</td>
<td></td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>?</td>
</tr>
<tr>
<td><code>endVertices()</code>, <code>opposite()</code>, <code>isIncidentOn(v)</code>, <code>v.isAdjacentTo(w)</code></td>
<td>?</td>
</tr>
<tr>
<td><code>v.incidentEdges()</code></td>
<td>?</td>
</tr>
<tr>
<td><code>insertEdge(u, v, w)</code>, <code>eraseEdge(e)</code></td>
<td>?</td>
</tr>
<tr>
<td><code>insertVertex(x)</code>, <code>eraseVertex(v)</code></td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
# Asymptotic Performance of Adjacency Matrix Structure

- $n$ vertices, $m$ edges
- No parallel edges
- No self-loops

<table>
<thead>
<tr>
<th>Space</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n^2)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>endVertices(), opposite(), isIncidentOn($v$), $v$.isAdjacentTo($w$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$v$.incidentEdges()</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insertEdge($u$, $v$, $w$), eraseEdge($e$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertVertex($x$), eraseVertex($v$)</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

## Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non-adjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge
### ASYMPTOTIC PERFORMANCE

- **n** vertices, **m** edges
- No parallel edges
- No self-loops

<table>
<thead>
<tr>
<th>Operation</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$O(n + m)$</td>
<td>$O(n + m)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>endVertices(), opposite(), isIncidentOn($v$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$v$. incidentEdges()</td>
<td>$O(m)$</td>
<td>$O(\text{deg}(v))$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$v$. isAdjacentTo($w$)</td>
<td>$O(m)$</td>
<td>$O(\min(\text{deg}(v), \text{deg}(w)))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertEdge($u$, $v$, $w$), eraseEdge($e$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertVertex($x$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>eraseVertex($v$)</td>
<td>$O(m)$</td>
<td>$O(\text{deg}(v))$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
DEPTH-FIRST SEARCH
Depth-first search (DFS) is a general technique for traversing a graph.

A DFS traversal of a graph $G$:
- Visits all the vertices and edges of $G$
- Determines whether $G$ is connected
- Computes the connected components of $G$
- Computes a spanning forest of $G$

DFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time.

DFS can be further extended to solve other graph problems:
- Find and report a path between two given vertices
- Find a cycle in the graph

Depth-first search is to graphs what Euler tour is to binary trees.
EXAMPLE

unexplored vertex
visited vertex
unexplored edge
discovery edge
back edge

\[ A(A) = \{B, C, D, E\} \]

\[ A(B) = \{A, C, F\} \]
\[ A(B) = \{A, C, F\} \]

\[ A(C) = \{A, B, D, E\} \]
\[ A(C) = \{A, B, D, E\} \]

\[ A(C) = \{A, B, D, E\} \]
\[ A(C) = \{A, B, D, E\} \]
**EXAMPLE**

\[ A(C) = \{A, B, D, E\} \]

\[ A(D) = \{A, C\} \]
\[ A(D) = \{A, C\} \]

\[ A(E) = \{A, C\} \]
\[ A(E) = \{A, C\} \]

\[ A(E) = \{A, C\} \]
\[ A(E) = \{A, C\} \]
EXAMPLE

\[ A(C) = \{A, B, D, E\} \]
\[ A(B) = \{A, C, F\} \]
\[ A(G) = \emptyset \]

\[ A(F) = \{B\} \]
\[ A(B) = \{A, C, F\} \]
\[ A(A) = \{A, B, C, D\} \]
DFS AND MAZE TRAVERSAL

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)
The algorithm uses a mechanism for setting and getting “labels” of vertices and edges.

**Algorithm DFS(G)**

**Input**: Graph $G$

**Output**: Labeling of the edges of $G$ as discovery edges and back edges

1. **for each** $v \in G$.vertices() **do**
2. $v$.setLabel(UNEXPLORED)
3. **for each** $e \in G$.edges() **do**
4. $e$.setLabel(UNEXPLORED)
5. **for each** $v \in G$.vertices() **do**
6. **if** $v$.getLabel() = UNEXPLORED
7. DFS($G,v$)

**Algorithm DFS($G,v$)**

**Input**: Graph $G$ and a start vertex $v$

**Output**: Labeling of the edges of $G$ in the connected component of $v$ as discovery edges and back edges

1. $v$.setLabel(VISITED)
2. **for each** $e \in v$.incidentEdges() **do**
3. **if** $e$.getLabel() = UNEXPLORED
4. $w \leftarrow e$.opposite($v$)
5. **if** $w$.getLabel() = UNEXPLORED
6. $e$.setLabel(DISCOVERY)
7. DFS($G,w$)
8. **else**
9. $e$.setLabel(BACK)
EXERCISE
DFS ALGORITHM

- Perform DFS of the following graph, start from vertex A
  - Assume adjacent edges are processed in alphabetical order
  - Number vertices in the order they are visited
  - Label edges as discovery or back edges

```
A ——— B ——— C ——— D
|      |      |      |
E ——— F
```
PROPERTIES OF DFS

- Property 1
  - $\text{DFS}(G, v)$ visits all the vertices and edges in the connected component of $v$

- Property 2
  - The discovery edges labeled by $\text{DFS}(G, v)$ form a spanning tree of the connected component of $v$
ANALYSIS OF DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Function DFS($G, v$) and the method incidentEdges() are called once for each vertex
ANO\LYSIS OF DFS

- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Algorithm DFS($G$)
\begin{itemize}
  \item Input: Graph $G$
  \item Output: Labeling of the edges of $G$ as discovery edges and back edges
  \begin{enumerate}
    \item for each $v \in G.\text{vertices()}$ do $O(n)$
    \item $v.\text{setLabel}(UNEXPLORED)$
    \item for each $e \in G.\text{edges()}$ do $O(m)$
    \item $e.\text{setLabel}(UNEXPLORED)$
    \item for each $v \in G.\text{vertices()}$ do $O(n + m)$
    \item if $v.\text{setLabel()} = UNEXPLORED$
    \item DFS($G, v$)
  \end{enumerate}
\end{itemize}

Algorithm DFS($G, v$)
\begin{itemize}
  \item Input: Graph $G$ and a start vertex $v$
  \item Output: Labeling of the edges of $G$ in the connected component of $v$ as discovery edges and back edges
  \begin{enumerate}
    \item $v.\text{setLabel}(VISITED)$
    \item for each $e \in v.\text{incidentEdges()}$ do $O(\deg(v))$
    \item if $e.\text{setLabel()} = UNEXPLORED$
    \item $w \leftarrow e.\text{opposite}(v)$
    \item if $w.\text{setLabel()} = UNEXPLORED$
    \item $e.\text{setLabel}(DISCOVERY)$
    \item DFS($G, w$)
    \item else
    \item $e.\text{setLabel}(BACK)$
  \end{enumerate}
\end{itemize}
We can specialize the DFS algorithm to find a path between two given vertices \( u \) and \( z \) using the template method pattern.

We call \( \text{DFS}(G, u) \) with \( u \) as the start vertex.

We use a stack \( S \) to keep track of the path between the start vertex and the current vertex.

As soon as destination vertex \( z \) is encountered, we return the path as the contents of the stack.

---

**Algorithm** pathDFS\((G, v, z)\)

1. \( v \).setLabel(\text{VISITED})
2. \( S \).push\((v)\)
3. if \( v = z \)
   4. return \( S \).elements()
5. for each \( e \in v \).incidentEdges() do
   6. if \( e \).getLabel() = \text{UNEXPLORED}
      7. \( w \leftarrow e \).opposite\((v)\)
      8. if \( w \).getLabel() = \text{UNEXPLORED}
         9. \( e \).setLabel(\text{DISCOVERY})
      10. \( S \).push\((e)\)
      11. pathDFS\((G, w)\)
      12. \( S \).pop()
   13. else
      14. \( e \).setLabel(\text{BACK})
      15. \( S \).pop()
We can specialize the DFS algorithm to find a simple cycle using the template method pattern.

We use a stack $S$ to keep track of the path between the start vertex and the current vertex.

As soon as a back edge $(v, w)$ is encountered, we return the cycle as the portion of the stack from the top to vertex $w$.

**Algorithm** cycleDFS($G, v, z$)

1. $v$.setLabel($VISITED$)
2. $S$.push($v$)
3. for each $e \in v$.incidentEdges() do
   4. if $e$.getLabel() = $UNEXPLORED$
   5. $w \leftarrow e$.opposite($v$)
   6. $S$.push($e$)
   7. if $w$.getLabel() = $UNEXPLORED$
   8. $e$.setLabel($DISCOVERY$
   9. cycleDFS($G, w$)
10. $S$.pop()
11. else
12. $T \leftarrow$ empty stack
13. repeat
14. $T$.push($S$.top($))$
15. $S$.pop($)
16. until $T$.top($) = w$
17. return $T$.elements($)
18. $S$.pop($)
BREADTH-FIRST SEARCH
BREADTH-FIRST SEARCH

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph $G$
  - Visits all the vertices and edges of $G$
  - Determines whether $G$ is connected
  - Computes the connected components of $G$
  - Computes a spanning forest of $G$
- BFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one
EXAMPLE

- **A** unexplored vertex
- **A** visited vertex
- - - unexplored edge
- → discovery edge
- →→ cross edge
EXAMPLE

DISCOVERY EDGE
CROSS EDGE

VISITED VERTEX
UNEXPLORED VERTEX
UNEXPLORED EDGE

55
EXAMPLE

A unexplored vertex
A visited vertex
unexplored edge
discovery edge
cross edge

unexplored vertex
unexplored edge
discovery edge
cross edge

unexplored vertex
unexplored edge
discovery edge
cross edge

unexplored vertex
unexplored edge
discovery edge
cross edge
BFS ALGORITHM

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

**Algorithm BFS(G)**

**Input:** Graph G

**Output:** Labeling of the edges and partition of the vertices of G

1. for each $v \in G$.vertices() do
2.   $v$.setLabel(UNEXPLORED)
3. for each $e \in G$.edges() do
4.   $e$.setLabel(UNEXPLORED)
5. for each $v \in G$.vertices() do
6.   if $v$.getLabel() = UNEXPLORED
7.     BFS($G, v$)

**Algorithm BFS($G, s$)**

1. $L_0 \leftarrow \{s\}$
2. $s$.setLabel(VISITED)
3. $i \leftarrow 0$
4. while $\neg L_i$.empty() do
5.   $L_{i+1} \leftarrow \emptyset$
6.   for each $v \in L_i$ do
7.     for each $e \in v$.incidentEdges() do
8.       if $e$.setLabel() = UNEXPLORED
9.         $w \leftarrow e$.opposite($v$)
10.        if $w$.setLabel() = UNEXPLORED
11.           $e$.setLabel(DISCOVERY)
12.          $w$.setLabel(VISITED)
13.         $L_{i+1} \leftarrow L_{i+1} \cup \{w\}$
14.     else
15.         $e$.setLabel(CROSS)
16.     $i \leftarrow i + 1$
 Perform BFS of the following graph, start from vertex A

- Assume adjacent edges are processed in alphabetical order
- Number vertices in the order they are visited and note the level they are in
- Label edges as discovery or cross edges
PROPERTIES

- Notation
  - $G_s$: connected component of $s$
- Property 1
  - $\text{BFS}(G, s)$ visits all the vertices and edges of $G_s$
- Property 2
  - The discovery edges labeled by $\text{BFS}(G, s)$ form a spanning tree $T_s$ of $G_s$
- Property 3
  - For each vertex $v \in L_i$
    - The path of $T_s$ from $s$ to $v$ has $i$ edges
    - Every path from $s$ to $v$ in $G_s$ has at least $i$ edges
ANALYSIS

- Setting/getting a vertex/edge label takes $O(1)$ time.
- Each vertex is labeled twice:
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice:
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence $L_i$.
- Method `incidentEdges()` is called once for each vertex.
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure.
  - Recall that $\sum_{v} \deg(v) = 2m$. 
ANALYSIS OF BFS

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges.

**Algorithm BFS(G)**

**Input:** Graph G

**Output:** Labeling of the edges and partition of the vertices of G

1. for each \( v \in G \).vertices() do \( O(n) \)
2. \( v \).setLabel(UNEXPLORERED)
3. for each \( e \in G \).edges() do \( O(m) \)
4. \( e \).setLabel(UNEXPLORERED)
5. for each \( v \in G \).vertices() do \( O(n + m) \)
6. if \( v \).getLabel() = UNEXPLORERED
7. BFS(\( G, v \)

**Algorithm BFS(G, s)**

1. \( L_0 \leftarrow \{s\} \)
2. \( s \).setLabel(VISITED)
3. \( i \leftarrow 0 \)
4. while \( \neg L_i \).empty() do
5. \( L_{i+1} \leftarrow \emptyset \)
6. for each \( v \in L_i \) do \( O(deg(v)) \)
7. for each \( e \in v \).incidentEdges() do
8. if \( e \).getLabel() = UNEXPLORERED
9. \( w \leftarrow e \).opposite(\( v \)
10. if \( w \).getLabel() = UNEXPLORERED
11. \( e \).setLabel(DISCOVERY)
12. \( w \).setLabel(VISITED)
13. \( L_{i+1} \leftarrow L_{i+1} \cup \{w\} \)
14. else
15. \( e \).setLabel(CROSS)
16. \( i \leftarrow i + 1 \)
APPLICATIONS

- Using the template method pattern, we can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time
  - Compute the connected components of $G$
  - Compute a spanning forest of $G$
  - Find a simple cycle in $G$, or report that $G$ is a forest
  - Given two vertices of $G$, find a path in $G$ between them with the minimum number of edges, or report that no such path exists
### DFS VS. BFS

<table>
<thead>
<tr>
<th>Applications</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning forest, connected components, paths, cycles</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Shortest paths</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Biconnected components</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

#### Applications
- Spanning forest, connected components, paths, cycles
- Shortest paths
- Biconnected components
DFS VS. BFS

Back edge \((v, w)\)
- \(w\) is an ancestor of \(v\) in the tree of discovery edges

Cross edge \((v, w)\)
- \(w\) is in the same level as \(v\) or in the next level in the tree of discovery edges

DFS

BFS