Graphs
Definition, Implementation and Traversal
Graphs

- Formally speaking, 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
Graphs

- 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 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
  - all the edges are directed
  - e.g., route network

- Undirected graph
  - all the edges are undirected
  - e.g., flight network
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 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
Terminology (cont.)

- **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 (cont.)

- **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**
  - $C_1 = (V,b,X,g,Y,f,W,c,U,a,e)$ is a simple cycle
  - $C_2 = (U,c,W,e,X,g,Y,f,W,d,V,a,e)$ is a cycle that is not simple
Properties

Notation

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

Example

- \( n = 4 \)
- \( m = 6 \)
- \( \text{deg}(v) = 3 \)
Properties

- Property 1
  - \( \sum_v \deg(v) = 2m \)
  - Proof: each edge is counted twice

- Property 2
  - In an undirected graph with no self-loops and no multiple edges
    - \( m \leq n (n - 1)/2 \)
    - Proof: each vertex has degree at most \( n - 1 \)

- What is the bound for a directed graph?
Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements

- Accessor methods
  - endVertices(e): an array of the two endvertices of e
  - opposite(v, e): the vertex opposite of v on e
  - areAdjacent(v, w): true iff v and w are adjacent
  - replace(v, x): replace element at vertex v with x
  - replace(e, x): replace element at edge e with x
Main Methods of the Graph ADT

- Update methods
  - insertVertex(o): insert a vertex storing element o
  - insertEdge(v, w, o): insert an edge (v,w) storing element o
  - removeVertex(v): remove vertex v (and its incident edges)
  - removeEdge(e): remove edge e

- Iterable collection methods
  - incidentEdges(v): edges incident to v
  - vertices(): all vertices in the graph
  - edges(): all edges in the graph
Edge List Structure

- **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 List Structure

- Edge list structure +
- Incidence sequence for each vertex
  - 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

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge
### Performance

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

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>incidentEdges(( v ))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>areAdjacent (( v, w ))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertVertex(( o ))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertEdge(( v, w, o ))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeVertex(( v ))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeEdge(( e ))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 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>( n + m )</td>
<td>( n + m )</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>incidentEdges((v))</td>
<td>( m )</td>
<td>( \text{deg}(v) )</td>
<td>( n )</td>
</tr>
<tr>
<td>areAdjacent ((v, w))</td>
<td>( m )</td>
<td>( \min(\text{deg}(v), \text{deg}(w)) )</td>
<td>1</td>
</tr>
<tr>
<td>insertVertex((o))</td>
<td>1</td>
<td>1</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>insertEdge((v, w, o))</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removeVertex((v))</td>
<td>( m )</td>
<td>( \text{deg}(v) )</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>removeEdge((e))</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Graph Traversal

- How to visit all vertices?

Depth-First Search
Subgraphs

- 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 and Forests

- 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.
Spanning Trees and Forests

- 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.
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
DFS Algorithm

- 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

**for all** $u \in G\text{.vertices()}$

```
setLabel(u, UNEXPLORED)
```

**for all** $e \in G\text{.edges()}$

```
setLabel(e, UNEXPLORED)
```

**for all** $v \in G\text{.vertices()}$

**if** `getLabel(v) = UNEXPLORED`

```
DFS(G, v)
```

**else**

```
setLabel(e, BACK)
```
Example

- **unexplored vertex**
- **visited vertex**
- **unexplored edge**
- **discovery edge**
- **back edge**
Example (cont.)
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)
Properties of DFS

Property 1

\(\text{DFS}(G, \nu)\) visits all the vertices and edges in the connected component of \(\nu\)

Property 2

The discovery edges labeled by \(\text{DFS}(G, \nu)\) form a spanning tree of the connected component of \(\nu\)
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
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices \( u \) and \( z \) using the template method pattern.
- We call \( DFS(G, u) \) 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)
    setLabel(v, VISITED)
    S.push(v)
    if \( v = z \)
        return S.elements()
    for all \( e \in G.incidentEdges(v) \)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                S.push(e)
                pathDFS(G, w, z)
                S.pop(e)
            else
                setLabel(e, BACK)
        S.pop(v)
```
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 $\text{cycleDFS}(G, v, z)$

- $\text{setLabel}(v, \text{VISITED})$
- $S.push(v)$
- for all $e \in G.\text{incidentEdges}(v)$
  - if $\text{getLabel}(e) = \text{UNEXPLORED}$
    - $w \leftarrow \text{opposite}(v, e)$
    - $S.push(e)$
    - if $\text{getLabel}(w) = \text{UNEXPLORED}$
      - $\text{setLabel}(e, \text{DISCOVERY})$
      - $\text{pathDFS}(G, w, z)$
      - $S.pop(e)$
    - else
      - $T \leftarrow \text{new empty stack}$
      - repeat
        - $o \leftarrow S.pop()$
        - $T.push(o)$
      - until $o = w$
      - return $T.elements()$
  - $S.pop(v)$

Breadth-First Search

- Traverse the graph level by level
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
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 $G$

for all $u \in G.\text{vertices}()$

setLabel($u$, UNEXPLORED)

for all $e \in G.\text{edges}()$

setLabel($e$, UNEXPLORED)

for all $v \in G.\text{vertices}()$

if $\text{getLabel}(v) = \text{UNEXPLORED}$

$BFS(G, v)$
Example

- **unexplored vertex**
- **visited vertex**
- **unexplored edge**
- **discovery edge**
- **cross edge**
Example (cont.)
Example (cont.)
Properties

Notation

$G_s$: connected component of $s$

**Property 1**

$BFS(G, s)$ visits all the vertices and edges of $G_s$

**Property 2**

The discovery edges labeled by $BFS(G, s)$ 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 $\sum_v \deg(v) = 2m$
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>

DFS

BFS

![DFS Diagram](image1.png)

![BFS Diagram](image2.png)
DFS vs. BFS (cont.)

**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

DFS

BFS
Graphs II

Digraphs, Strongly Connective Component, Topological Sorting, and Minimum Spanning Tree
Java Graph Library

- No standard library
- JGraphT
  - An open source library
  - [http://www.jgrapht.org/](http://www.jgrapht.org/)
  - Supports most mentioned Graph functions
  - You can simply download the file and use the library to create your graph
<table>
<thead>
<tr>
<th>Time</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00~10:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:00~11:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:00~12:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Schedule on Jan. 11

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00-3:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:00-5:00</td>
<td>MAKEUP Exam</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Makeup exam on Jan. 11

- Friday 3:10-5:00. College of Commerce 312
- Maximal 80 points
- Dynamic programing on LCS
- Binary Search Tree (AVL)
- Hash Table
- Cycle Detection