Introduction to Graphs

- One of the most versatile data structures used in programming
  - Can be used to model real-world problems
  - Important algorithmic tool
  - However, not for storage problems
- It is much more general than a tree structure
  - **Components**: vertices (nodes) and edges
  - Allow arbitrary number of edges
Introduction to Graphs- Examples

- Northwest Airline Flight
Introduction to Graphs - Examples

- Internet connectivity (99)
Introduction to Graphs- Examples

- Social Network

- Brain?
Introduction to Graphs- Examples

- ECE Course prerequisites
Introduction to Graphs

- Come in different flavors (with all combinations possible)

- Non-connected vs connected graphs

- Non-directed vs directed graphs

- Unweighted vs Weighted graphs
Graphs: more definitions

- $G = (V, E)$, where $V$ is the vertex set, $E$ is the edge set
- Two vertices are adjacent one another if connected by a single edge (notion of neighbors)
  - The neighbors of $A$ are $B, C, D$
  - The neighbors of $B$ are $A, C$
  - etc.
- A path is a sequence of edges connecting two vertices anywhere in the graph
  - Simple path (traverse the node only once): $ABCD$
  - Cycle: path that starts and ends at the same point: $ABD$
- Notion of subgraph
Graph Representation
(for non-directed, unweighted graph)

- **Adjacency Matrix**
  - In dense storage, use a 2D array: A[0...N-1][0...N-1]
  - If vertex i and vertex j are adjacent in graph → A[i][j]=1
  - Otherwise → A[i][j]=0

<table>
<thead>
<tr>
<th>A[i][j]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

So, Matrix A =

\[
\begin{pmatrix}
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
\end{pmatrix}
\]

Non-directed graph: A is symmetric, A=A^T
Graph Representation
(for directed, unweighted graph)

- **Adjacency Matrix**
  - In dense storage, use a 2D array: \( A[0...N-1][0...N-1] \)
  - If vertices \( i \) and \( j \) adjacent in graph \( \rightarrow A[i][j]=1 \) (but may be different of \( A[j][i] \))
  - Otherwise \( \rightarrow A[i][j]=0 \)

\[
\begin{bmatrix}
0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

So, Matrix \( A = \) 

- directed graph: \( A \) is usually non-symmetric, \( A! = A^T \)
Graph Representation (for non-directed, weighted graph)

- **Adjacency Matrix**
  - In dense storage, use a 2D array: \( A[0...N-1][0...N-1] \)
  - If vertices \( i \) and \( j \) adjacent in graph \( \rightarrow A[i][j]=\text{weight} \)
  - Otherwise \( \rightarrow A[i][j]=0 \)

\[
\begin{array}{cccc}
A[i][j] & 0 & 1 & 2 & 3 \\
0 & 0 & 20 & 10 & 1 \\
1 & 20 & 0 & 0 & 5 \\
2 & 10 & 0 & 0 & 4 \\
3 & 1 & 5 & 4 & 0 \\
\end{array}
\]

So, Matrix \( A = \begin{pmatrix}
0 & 20 & 10 & 1 \\
20 & 0 & 0 & 5 \\
10 & 0 & 0 & 4 \\
1 & 5 & 4 & 0 \\
\end{pmatrix} \)
Adjacency List is the 'linked-list' representation of graphs.

In textbook and traditional course on 'data structure and algo', this approach (array of linked list LL) is presented as being somehow different from the adjacency matrix representation.

However, this is not the way it should be looked at. It is just a different storage/format for the matrix.

- Graph → Matrix Representation

  - Dense Storage – 2d array
  - Sparse Storage- array of LL
  - Sparse Storage- set of 1d arrays
  - etc.
Vertex are objects (only one letter here)
We need to keep track of the status of the vertex (visited or not) for performing searches

```java
public class Vertex {
    private char label; // label e.g. A, B
    public boolean visited;

    public Vertex (char c) {
        label = c;
        visited = false;
    }

    public void display() {
        System.out.println(label);
    }
}
```

```java
public class Graph {
    private int maxSize;
    private Vertex[] nodes; // array of nodes
    private int[][] mat; // adjacency matrix
    private int N;

    public Graph(int maxSize) {
        this.maxSize = maxSize;
        nodes = new Vertex[maxSize];
        mat = new int[maxSize][maxSize];
        for(int i=0; i<maxSize; i++)
            for(int j=0; j<maxSize; j++) mat[i][j] = 0;
    }

    public void addVertex(char c) {
        if (N >= maxSize) {
            System.out.println("Graph full");
            return;
        }
        nodes[N++] = new Vertex(c);
    }

    public void addEdge(int v1, int v2) {
        mat[v1][v2] = 1;
        mat[v2][v1] = 1;
    }
}
```
Graph Operations

- Graph has less structure than Trees, we need more general operations
- **Goal:** find which vertices can be reach from a particular vertex
- There are two main Graph Traversal algorithms (search algorithms)
  - **1- Depth-First Search (DFS)**- which uses a stack
  - **2- Breadth-First Search (BFS)**- which uses a queue
- Both algorithms will eventually reach all connected vertices (they are visiting the vertices in different order/using different strategies)
Example:

- Start with a starting point, for example A
- 3 things first: visit this vertex, push it on a stack (so you can remember it), mark it as visited (so you won't visit it again)
- Follow these simple rules:
  
  - **Rule 1**: If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack
  - **Rule 2**: If you can't follow rule 1, then if possible pop a vertex from stack
  - **Rule 3**: If you can't follow rules 1 and 2, then you are done
### Graph Operations - Depth-First Search (DFS)

<table>
<thead>
<tr>
<th>Visit</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Stack: A</td>
</tr>
<tr>
<td>B</td>
<td>Stack: AB</td>
</tr>
<tr>
<td>F</td>
<td>Stack: ABF</td>
</tr>
<tr>
<td>H</td>
<td>Stack: ABFH</td>
</tr>
<tr>
<td>C</td>
<td>Stack: AC</td>
</tr>
<tr>
<td>D</td>
<td>Stack: AD</td>
</tr>
<tr>
<td>G</td>
<td>Stack: ADG</td>
</tr>
<tr>
<td>I</td>
<td>Stack: ADGI</td>
</tr>
<tr>
<td>E</td>
<td>Stack: ADG</td>
</tr>
<tr>
<td></td>
<td>Stack: ADG</td>
</tr>
<tr>
<td></td>
<td>Stack: AD</td>
</tr>
<tr>
<td></td>
<td>Stack: A</td>
</tr>
<tr>
<td></td>
<td>Stack: AE</td>
</tr>
<tr>
<td></td>
<td>Stack A</td>
</tr>
<tr>
<td></td>
<td>Stack A</td>
</tr>
<tr>
<td></td>
<td>Stack (empty)</td>
</tr>
<tr>
<td></td>
<td>Done</td>
</tr>
</tbody>
</table>

#### Graph

```
A ——— B ——— F ——— H
|     |     |
| C   | D   |
| G   | I   |
| E   |     |
```

- **Visit: ABFHCDGIE**
- **Analogy:** Get as far away from the starting point as quickly as possible and returns only if you find a dead end → Maze game
Graph Operations - Depth-First Search (DFS)

- Todo: Java applet GraphN.html

```java
public class GraphN {
    public void dfs(){
        nodes[0].visited=true;
        nodes[0].display();
        theStack.push(0);

        while(!theStack.isEmpty()){
            int v=getAdjUnvisitedNode(theStack.peek());
            if (v == -1) // no such node
                theStack.pop();
            else {
                nodes[v].visited=true;
                nodes[v].display();
                theStack.push(v);
            }
        }
        // stack is empty so we can reset the flags
        for(int i=0; i<N;i++) nodes[i].visited=false;
    }

    public int getAdjUnvisitedNode(int v){
        for(int i=0; i<N;i++)
            if (mat[v][i]==1 && nodes[i].visited=false)
                return i; // found neighbor
        return -1; // no such node
    }
}
```