Introduction to Graphs

My favorite data structure ;}
A familiar place...
From Networks to Graphs

Network Science: Graph Theory 2012

N=4
L=4
Undirected Graphs

You can use an edge in either direction
(Undirected) Graph Definition

- Our first non-linear data structure!
- An **undirected graph** $G$ consists of two sets $G = \{V, E\}$
  - A set of $V$ **vertices**, or nodes
  - A set of $E$ **edges**, relationships between nodes
- A **subgraph** $G'$ consists of a subset of the vertices and edges of $G$
- **Adjacent** vertices are two vertices joined by an edge
Paths and Cycles

- A **path** between two vertices
  - A sequence of edges that begins at the first vertex and ends at the other vertex

- A **simple path**
  - A path that passes through a vertex at most once

- A **cycle**
  - A path that begins and ends at the same vertex

- A **simple cycle**
  - A cycle that does not pass through a vertex more than once
Complete Graph

- A **complete** graph
  - A graph that has an edge between each pair of distinct vertices

- How many edges does a complete graph with **n** nodes have?
Connectivity

- A **connected** graph
  - A graph that has a path between each pair of vertices

- A **connected component** is a connected subgraph

- A **disconnected** graph
  - A graph that has at least one pair of vertices without a path between them
Tree: A Special Graph

A tree is a connected graph in which there is exactly one simple path connecting any two nodes.

How many edges does a tree with \( n \) nodes have?
Directed Graphs

Think of one-way streets
WC Campus Directed Graph

[Diagram showing a network of nodes and arrows connecting various places such as lulu, jewett, pendleton, founders, chapel, tower, clapp, and science.]

club
Directed Graphs and DAGs

- **Directed** graph $G = \{V, A\}$
  - **Arcs** (or **links**) are directed edges between vertices
    - We may have 2 arcs between any pair of vertices, one in each direction
  - Vertex $y$ is **adjacent** to vertex $x$ **iff** (if and only if) there is an arc (directed edge) from $x$ to $y$

- **Directed path**
  - A sequence of directed edges between two vertices

- **Directed Acyclic Graph (DAG)**
  - Directed graph with no cycles

How few arcs can you remove to make this graph a DAG?
Visualizing Graphs with yEd

- yEd: A great and simple graph visualization
- Download it: https://www.yworks.com/products/yed
- You can create any graph by clicking (for vertices) and clicking-and-dragging (for edges)
- Lots of graph formats supported. Use .tgf
- TGF format: a text file listing lines of:
  - `vertexID vertexName` (for vertices)
  - `#`
  - `vertexID vertexID` (for arcs)
- Once you upload a file, choose Layout > Circular to see it laid out nicely.
Strong Connectivity

- A **strongly connected** graph
  - A graph that has a directed path between any pair of vertices

- A **strongly connected component**
  - a maximally strongly connected subgraph

- How many strongly connected components do you see?
Implementing Graphs

An **undirected** graph $G$ consists of two sets $G = \{V, E\}$, a set $V$ of vertices and a set $E$ of edges.

A **directed** graph $G$ consists of two sets $G = \{V, A\}$, a set $V$ of vertices and a set $A$ of arcs (directed edges).

If a directed graph contains between every pair of vertices **either both arcs or none**, then it can be considered undirected.
public interface Graph<T> { // Partial interface

public int numVert() // Returns number of vertices
public int numArcs() // Returns the number of arcs

public void addVertex(T v) // Insert a vertex in a graph
public void removeVertex(T v) // Delete a vertex along with any arcs between v and other vertices

public void addArc(T v1, T v2) // Inserts an arc between two given vertices in a graph
public void removeArc(T v1, T v2) // Deletes the arc between two given vertices in a graph

public T findVertex(String key) // Returns the vertex that contains a given search key

public boolean isArc(T v1, T v2) // Returns true iff an arc exists between vertices v1 and v2

public LinkedList<T> getNeighbors(T v) // Returns a list of the vertices adjacent to vertex v

public boolean isEmpty() // Returns true iff a graph is empty
Adjacency Matrix implementation

- **Adjacency matrix for (directed) graph with**
  - *n vertices:* numbered 0, 1, ..., *n* – 1
  - *arcs:* boolean *n* × *n* array where *arcs*[i][j] =
    - 1 (true) if there is an arc from vertex *i* to vertex *j*
    - 0 (false) if there is no arc from vertex *i* to vertex *j*

What do you need to add to turn this into an undirected graph? What property does the matrix of an undirected graph have?
import java.util.*; import java.io.*;

public class AdjMatGraph<T> implements Graph<T> {
    private final int NOT_FOUND = -1;
    private final int DEFAULT_CAPACITY = 10;

    private int n;  // number of vertices in the graph
    private boolean[][] arcs;  // adjacency matrix of arcs
    private T[] vertices;  // values of vertices

    public AdjMatGraph() {
        n = 0;
        this.arcs = new boolean[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
        this.vertices = (T[])(new Object[DEFAULT_CAPACITY]);
    }

    public boolean isEmpty() { ... }  // returns true if a graph is empty
    public int numVert() { ... }  // (n) returns the number of vertices
    public int numEdges() { ... }  // (m) returns the number of edges
    etc...

Adjacency Lists implementation

- An adjacency list for a (directed) graph with
  - *n* vertices numbered 0, 1, ..., *n* – 1
  - *arcs*: array of *n* linked lists
    - The *i*\(^{th}\) linked list has a list entry for vertex *j*
      if the graph contains an arc from vertex *i* to vertex *j*
Adjacency Matrix vs. Adjacency Lists

• Which representation supports better these two frequent operations on graphs?
  • \texttt{isArc}(v, w)
    Return true iff there is an arc from vertex \(v\) to vertex \(w\)
  • \texttt{getNeighbors}(v)
    Return list of all vertices adjacent to a given vertex \(v\)

<table>
<thead>
<tr>
<th>Arcs</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\begin{tikzpicture}[auto, node distance=1.5cm, >=stealth]
  \node (a) {a};
  \node (b) [below of=a] {b};
  \node (c) [below of=b] {c};
  \node (d) [below of=c] {d};

  \foreach \x in {a, b, c, d} {
    \draw[->] (\x) -- (\x |- a) node [midway, fill=white] {\texttt{Arcs}};
    \draw[->] (\x) -- (\x |- c) node [midway, fill=white] {\texttt{Arcs}};
  }
\end{tikzpicture}
Weighted Graphs
Weighted Graph

- **Weighted** graph
  - A graph whose edges have **weights**
  - Weight is the “cost” or “magnitude” of the relationship represented by the edge
  - It can be directed or undirected
Weighted Adjacency Matrix

- Adjacency matrix for a weighted graph with
  - \( n \) vertices numbered 0, 1, ..., \( n - 1 \)
  - An \( n \times n \) array matrix \( \text{EdgeW} \) such that
    \[
    \text{EdgeW}[i][j] =
    \]
    - The weight of the arc from vertex \( i \) to vertex \( j \)
      if there is an edge from \( i \) to \( j \)
    - \( \infty \) if there is no edge from vertex \( i \) to vertex \( j \)
Weighted Adjacency List

- Adjacency list for a weighted undirected graph
  - Each list entry contains the edge label and weight
  - Treats each edge as if it were two arcs in opposite directions
Graph Traversal: Depth-First Search

- Produce a list of all the vertices that can be reached starting at some vertex
Searching a Graph

- **High Planes Airline Company (HPAir)**
  - Maintains a flight map (a directed graph)
  - For each customer request \( oC \rightarrow dC \), find a directed path of HPAir flights from the origin city \( oC \) to the destination city \( dC \)
  - E.g. \( P \rightarrow Z \) finds \([(P,W), (W,Y),(Y,Z)]\)

- **The flight map is a directed graph**
  - Arc \((a,b)\) between vertices means
    - There is a flight from city \( a \) to city \( b \)
  - Directed path means
    - There is a sequence of flight connections
Draw your Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Draw your Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

![Diagram](image-url)
The solution performs an **exhaustive** search
- Beginning at the origin city, tries every possible sequence of flights until **either**
  - Finds a sequence that gets to the destination city
  - Determines that no such sequence exists

**Keep going deep** in the graph as much as you can selecting among adjacent vertices
**Backtrack** (if you must) to recover from a choice that did not reach the city
**Repeat** the above steps until successful in finding the sequence or failure to reach destination

Which data structure is useful in backtracking?
- It should help you remember how you got to the current point
DFS(origin, destination): Search the Map

stk = new Stack<E>();

stk.push(origin);

while (stk is not empty) {
    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)
        stk.push(anotherCity);
    else // cannot find an unvisited neighbor
        stk.pop(); // backtrack
... and remember where you’ve been

```java
stk = new Stack<E>();

Mark all nodes as not visited yet;
stk.push(origin);
Mark(origin) as visited;
while (stk is not empty) {
    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)
        stk.push(anotherCity);
        Mark(anotherCity) as visited;
    else // cannot find an unvisited neighbor
        stk.pop(); // backtrack
}
```
Depth-First-Search Example: From P -> Z

```
stk = new Stack<E>();
Mark all nodes as not visited yet;
stk.push(origin);
Mark(origin) as visited;
while (stk is not empty) {
    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)
        stk.push(anotherCity);
        Mark(anotherCity) as visited;
    else // cannot find an unvisited neighbor
        stk.pop(); // backtrack
```

P R X W S T Y Z
Marked (visited)

Stack stk

P R X R R W W S S T T Z Y Y
Would DFS work for undirected graphs?

```
stk = new Stack<E>();
Mark all nodes as not visited yet;
stk.push(origin);
Mark(origin) as visited;
while (stk is not empty) {
    if (you can find an unvisited neighbor, 
        anotherCity, from the city on top of stack)
        stk.push(anotherCity);
        Mark(anotherCity) as visited;
    else // cannot find an unvisited neighbor
        stk.pop(); // backtrack
}
```
Searching a Maze

Try rooms in some order (e.g. left, down, right, up)

```
C     B     A     H
E     D     F     G
```

```
A
B
C
D
E
F
G
H
```

```
A
B
C
D
E
F
G
H
```
“To find the way out of a labyrinth there is only one means. At every new junction never seen before, the path we have taken will be marked with three signs. If you see that the junction has already been visited, you will make only one mark on the path you have taken. If all the apertures have already been marked, then you must retrace your steps. But if one or two apertures of the junction are still without signs, you will choose any one, making another sign on it. Proceeding through an aperture that bears only one sign, you will make two more, so that now the aperture bears three.”
Testing for Connectivity using DFS(oC)

Connected: An undirected graph for which there is a path from any node to any other node

Is this graph connected?

Connected component: A connected sub-graph

Can we use DFS to find all connected components?
Strong Connectivity

- Strongly Connected: A graph for which there is a directed path from any node to any other node

- Is this graph strongly connected?

- Strongly connected component: A strongly connected sub-graph

- Can you find the strongly connected components of this graph?
LinkedStack<Integer> traversalStack = new LinkedStack<Integer>();

ArrayIterator<T> dfsIter = new ArrayIterator<T>();

boolean[] visited = new boolean[getNumVertices()];

for (int vID= 0; vID < getNumVertices(); vID++)
    visited[vID] = false;

traversalStack.push(startIndex);
dfsIter.add(vertices.get(startIndex));
visited[startIndex] = true;

while (!traversalStack.isEmpty()){
    currentVertex = traversalStack.peek(); found = false;

    for (int vID = 0; vID < getNumVertices() && !found; vID++)
        if (isArc(currentVertex,vID) && !visited[vID]) {
            traversalStack.push(vID);
            dfsIter.add(vertices.get(vID));
            visited[vID] = true;  found = true; } 

    if (!found && !traversalStack.isEmpty()){
        traversalStack.pop(); } } }

return dfsIter;
Assuming you began at node **a**, give the order of traversal if you visited every node.

**DFS:**
From Parking to Parking
WC Campus Directed Graph
WC Campus DAG