Introduction to Graphs

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

Network Science: Graph Theory
2012
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
Directed Graphs and DAGs

- **Directed** graph
  - An arc (or link) is a directed edge
  - 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 a 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?
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?
Weighted Graph

- **Weighted** graph
  - A graph whose edges have **weights**
  - Weight is the “cost” or “magnitude” of the relationship represented by the edge
Implementing Graphs
public interface Graph<T> { // Partial interface

public int numVert() // (n) return number of vertices
public int numEdges() // (m) returns the number of edges

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

public void addEdge(T v1, T v2) // Insert an edge between
two given vertices in a graph
public void removeEdge(T v1, T v2) // Deletes the edge
between two given vertices in a graph

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

public boolean isEdge(T v1, T v2)
// returns true if an edge exists between two given vertices

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

public boolean isEmpty() // return true if 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 property does the matrix of an undirected graph have?
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?
  - **isEdge**($v, w$)
    Determine whether there is an edge from vertex $v$ to vertex $w$
  - **getNeighbors**($v$)
    Return list of all vertices linked to from a given vertex $v$
**Weighted Adjacency Matrix**

- **Adjacency matrix** for a weighted graph with
  - *n vertices* numbered 0, 1, ..., *n* – 1
  - An *n* × *n* array matrix *EdgeW* such that
    - *EdgeW[i][j]* =
      - The weight of the arc from vertex *i* to vertex *j* if there is an edge from *i* to *j*
      - ∞ 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
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.
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?
Graph Traversal: Depth-First Search

- Visits all the vertices that it can reach starting at some vertex
- Visits all vertices of the graph \textit{iff} the graph is connected
- Must not loop forever, if a graph contains a cycle
- Must never visit a vertex more than once
A Search Problem

• High Planes Airline Company (HPAir) Problem
  – For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city

• The flight map for HPAir is a directed graph
  – Arc between vertices (a,b) means
    • There is a flight from city a to city b
  – Directed path means
    • There is a sequence of flight connections
Depth-First Search (Non-recursive solution)

• 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 moving forward** as much as you can selecting among adjacent vertices
• **Backtrack** (if you have to) to recover from a choice that did not reach final city
• Repeat the above steps until the end (success 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 (a sequence of flights from origin to destination has not yet been found) {
    if (you cannot go anywhere from the city on top of stack)
        stk.pop();  // backtrack
    else select a neighbor, anotherCity, from the city on top of stack;
        stk.push(anotherCity);
}

... and remember where you’ve been

```java
stk = new Stack<E>(); Clear Marks;
stk.push(origin);
Mark(origin) as visited;
while (a sequence of flights from \texttt{origin} to \texttt{destination} has not been found) {
    if (you cannot find an unvisited city from the city on top of stack)
        stk.pop(); // backtrack
    else select an unvisited neighbor, \texttt{anotherCity}, from the city on top of stack;
    stk.push(anotherCity);
        Mark(anotherCity) as visited;
}
```
Depth-First-Search Example: From P→Z

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

Stack stk
Would DFS(oC) work for undirected graphs?

```java
stk = new Stack<E>(); Clear Marks;
stk.push(originCity);
Mark(originCity) as visited;
while (a sequence of flights from originCity to destinCity has not been found) {
    if (you cannot find an unvisited city from the city on top of stack)
        stk.pop(); // backtrack
    else select an unvisited neighbor, anotherCity,
        from the city on top of stack;
        stk.push(anotherCity);
        Mark(anotherCity) as visited;
}
```
Searching a Maze

![Maze Diagram]

A maze is a type of puzzle consisting of a network of paths and alleys. The goal is to find a path from the start to the finish. In this example, the start is in the bottom left corner, and the finish is in the top right corner. The maze is solved by moving from one cell to an adjacent cell, following the path indicated by the arrows.
“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?
Graph Traversal: Breadth First Search

How does Google collect Web pages?
How Search Engines Work

1. **Crawl the Web**
   - Document IDs

2. **Create doc index**
   - Search engine servers

3. **Rank results**
   - Search engine servers

4. **Word Index & Frequencies**

5. **User query**

6. **User**

7. **THE WEB**
The Web is a Graph

- Directed Graph of Nodes and Arcs
  - Nodes = web pages
  - Arcs = hyperlinks from a page to another
- A graph can be explored
- A graph can be indexed

URL: http://cs.wellesley.edu/~cs230/slides/BFS.pdf
Traversing the Webgraph

- The web can be considered a graph “the Webgraph”
- Web pages are the graph nodes
- Hyperlinks on pages are graph arcs
- The web graph is huge (way over one million billions nodes) - maybe infinite (pages are created on the fly)
- For traversing the web graph, Depth-first-search (DFS) is not a good strategy. (Why?)

The shape of the Web is ... a “bow-tie”(!)
Breadth First Example: BFS(9)

Queue:

9 6 7 8 3 4 5 1 2

Iterator:

Mark all vertices as unvisited
enQueue \( v \) onto a new queue \( Q \)
Mark \( v \) as visited

// Procedure:
While (\( Q \) is not empty)
    deQueue a vertex \( w \) from \( Q \)
    For each unvisited vertex \( u \) adjacent to \( w \)
        enQueue \( u \) onto \( Q \)
        Mark \( u \) as visited
BFS(v) pseudocode

// BFS traversal starting at v

// Initialization:
Mark all vertices as unvisited
enQueue v onto a new queue Q
Mark v as visited

// Procedure:
While (Q is not empty)
    deQueue a vertex w from Q
    For each unvisited vertex u adjacent to w:
        enQueue u onto Q
        Mark u as visited

BFS from S to G:
How do you remember the path?

Initialization: enqueue path [S] in Q

While you have not reached G

- dequeue a path from BFS queue and
- consider the last node x in the path
- extend the path to unvisited neighbors of x and
- enqueue extended paths to back of Q

... in the next step it will reach the goal G
Assuming you began at node a, give the order of traversal if you visited every node.

**DFS:**

**BFS:**