Graphs

Graphs

ADTs (Abstract Data Structure)

- Unstructured Structures: Sets
- Sequential, linear structures: Arrays, linked lists
  - iterating over all elements
  - accessing via index
- Hierarchical Structures: Trees
  - prefix relationship

Graphs

Graph ADT

- Create a graph
- Are two vertices adjacent?
- Is the graph dense? sparse?
- How far are two vertices in the graph?
- How many components are there in the graph?
- Can we find a vertex with particular key value?

Graph Definitions

- **Basic objects : V**: vertices, nodes
- **Relationships between them : E**: edges, arcs, links

- Symmetric/ Undirected
- Directed graphs

- **Size of graph:** |V|+|E|
- **u is a neighbor of v** : there is an edge from u to v / from v to u
- **Path** : sequence of vertices and edges that depicts hopping along graph *(considering direction of edges)*

Implementing Graphs in Java

```java
public abstract class Graph {

        private int numVertices;
        private int numEdges;
        //optional association of String labels to vertices
        private Map<Integer,String> vertexLabels;

        public Graph() {
                numVertices = 0;
                numEdges = 0;
                vertexLabels = null;
        }

        public int getNumVertices() {
                return numVertices;
        }

        public int getNumEdges() {
                return numEdges;
        }

        public int addVertex() {
                implementAddVertex();
                numVertices ++;
                return (numVertices-1);
        }

        public abstract void implementAddVertex();
```

```
        public abstract List<Integer> getNeighbors(int v);
        public abstract List<Integer> getInNeighbors(int v);
}
```

Ajacency Matrix

- Directed: 5 edges in graph, 5 nonzero entries
- Undirected: 5 edges in graph, 10 nonzero entries

```
public class GraphAdjMatrix extends Graph {

        private final int defaultNumVertices = 5;
        private int[][] adjMatrix;

        /** Create a new empty Graph */
        public GraphAdjMatrix () {
                adjMatrix = new int[defaultNumVertices][defaultNumVertices];
        }

        public void implementAddVertex() {
                int v = getNumVertices();
                if (v >= adjMatrix.length) {
                        int[][] newAdjMatrix = new int[v*2][v*2];
                        for (int i = 0; i < adjMatrix.length; i ++) {
                                for (int j = 0; j < adjMatrix.length; j ++) {
                                        newAdjMatrix[i][j] = adjMatrix[i][j];
                                }
                        }
                        adjMatrix = newAdjMatrix;
                }
        }

        public void implementAddEdge(int v, int w) {
                adjMatrix[v][w] += 1;
        }
```

- Algebraic representation of graph structure.
- Fast to test for edges.
- Fast to add/ remove edges.
- Slow to add/ remove vertices.
- Requires a lot of menmory : |V|^2

Adjacency List

Want to avoid storing information on edges that aren't in the graph.

- Edges connect a vertex to its neighbors

```
public class GraphAdjList extends Graph {
    // vertex -> { neighbors }
        private Map<Integer,ArrayList<Integer>> adjListsMap;
        public GraphAdjList () {
                adjListsMap = new HashMap<Integer,ArrayList<Integer>>();
        }

        /**
         * Implement the abstract method for adding a vertex.
         */
        public void implementAddVertex() {
                int v = getNumVertices();
                ArrayList<Integer> neighbors = new ArrayList<Integer>();
                adjListsMap.put(v,  neighbors);
        }

        /**
         * Implement the abstract method for adding an edge.
         * @param v the index of the start point for the edge.
         * @param w the index of the end point for the edge.
         */
        public void implementAddEdge(int v, int w) {
                (adjListsMap.get(v)).add(w);
```

```
        }
}
```

- Storage : |V| + |E| (in dense graph |E| -> |V|^2)
- Easy to add vertices
- Easy to add/ remove edges
- May use a lot less memory than adjacency matrices
- **Sparse graph: O(1) edges for each vertex** : more efficient

Neighbors (vertices that are adjacent)

- In degree: number of incoming edges
- Out degree: number of outgoing egges

Matrix Multiplication for Finding Two-hop Neighbours

Class design and simple graph search

Basic Graph Search

DFS - Depth-First Search

- How to keep track of where to search next?
  - **Stack:** List where you add and remove from one end only: push - > add an element; pop -> remove an element
  - **Last In, First Out (LIFO)**
- How to keep track of what's been visited?
  - **HashSet**: Constant time add, remove, and find

- How to keep track of the path from start to goal?
    - **HashMap**: Link each node to the node from which it was discovered

Algorithm

```
Initialize: stack, visited HashSet and parent HashMap
Push S onto the stack and add to visited
while stack is not empty:
    pop node curr from top of stack
    if curr == G return parent map
    for each of curr's neighbors, n, not in visited set:
        add n to visited set
        add curr as n's parent in parent map
        push n onto the stack
// if we get here then there's no path
```

```
DFS(n, G, visited, parents):
if S == G return;
for each of S's neighbors, n, not in visited set:
    add n to visited set
    add S as n's parent in parents map
    DFS(n, G, visited, parents)
```

BFS - Breadth-First Search

- How to keep track of where to search next?
    - **Queue:** List where you add element to one end and remove them from the other: enqueue - > add an element; dequeue -> remove an element
    - **First In, First Out (FIFO)**

```
Initialize: queue, visited HashSet and parent HashMap
Enqueue S onto the queue and add to visited
while queue is not empty:
```

```
    dequeue node curr from front of queue
    if curr == G return parent map
    for each of curr's neighbors, n, not in visited set:
        add n to visited set
        add curr as n's parent in parent map
        enqueue n onto the queue
// if we get here then there's no path
```

From problem Specification to Class Design

Class Design

- What do I want to do with the graph?
    - Goal: Design classes to support path finding through a maze
- What is the ratio of edges to nodes? (Adj. list or matrix)?
    - Maze(Adj list representation: this graph is *sparse*)
- How do I need to access to nodes/ edges
    - MezeNode[][] nodes
- What properties do nodes and edges need to store?

Shortest Path Algorithms

Dijkstra's Algorithm

- BFS doesn't account for edge wights, only number of edges
- How to keep track of where to search nexxt?
    - **Priority Queue - Heaps** : List where you add an {element, priority} to one end and remove highest priority item from the other
    - **enqueue** : add an {element, priority}
    - **dequeue** : remove the highest priority element - larger distances are lower priority

- Dequeing and enqueueing from a priority queue with N elementsL O(|logN);

O(|V| + |E| * log|E|)

```
Initialize Priority queue(PQ), visited HashSet, parent HashMap, and distances to infinity
Enqueue {S,0} onto the PQ
while PQ is not empty:
    dequeue node curr from front of queue
    if (curr is not visited)
        add curr to visited set
        if curr == G return parent map
        for each of curr's neighbors, n, not in visited set:
            if path through curr to n is shorter
                update n's distance
                update curr as n's parent in parent map
                enqueue {n, distance} into the PQ
// if we get here then there is no path
```

A* Search Algorithm

Dijstra's Algorithm:

- Priority Queue ordering is based on:
  - g(n): the distance from start vertex to vertex n

A Algoritim: (only change the priority function)

- Priority Queue ordering is based on:
  - g(n): the distance from start vertex to vertex n
  - h(n): the *heuristic estimated* (guaranteed to find shortest path IF estimate is never an overestimate) cost from vertex n to goal vertex
- f(n) = g(n) + h(n)

The Travelling Salesperson Problem (TSP)

TSP

- Goal: Travel to other cities
  - minimize cost
  - visit each exactly once
  - end at start point
- The graph is fully connected
- **Adjacency Matrix**
- Problem: Given n cities with one Hometown and all pairwise distances, plan a tour starting and ending at Hometown that visits every city exactly once and has minimum distance

Greedy Algorithm

pick best next choice

Brute - Force Algorithm

Try all paths and choose the shortest.

Running Time

Brute-Force

```
bestPath = null, bestDist = +infinity
for each permutation of cities, starting and ending in Hometown:
    calculate distance of current pwemutation
    if (diatance < bestDist)
        bestPath = currnt permutation.bestDist = distance
return bestPath
```

O(n!) - (n-1)!*n

Greedy Algorithm (faster)

```
bestPath = []
current = Hometown
cities to visit = all other cities
while (more cities to visit)
    select city closest to current and add to bestPath
    remove current city from cities to visit
    current = selected city
return bestPath
```

O(n^2) - (n-1)*n

Solving TSP using Heuristics

NP Hard

Complexity Theory

- Classifies problem by their inherent difficulty
- P: polynomial time - O(n^k)
- NP: nondeterministic polynomial time. *Some problems seem harder to find solutions, but its still easy to verify solution correctness*
- NP-Hard: Problems are at *least* as sifficult to solve as hardest problem in NP
- NP-Complete: No known polynomial time algorithm to find a solution, but can check a solution in polynomial time