Undirected Graphs

Graph = set of vertices connected pairwise by edges

Path = sequence of vertices connected by edges

Cycle = path whose first and last vertices are the same

* Two vertices are **connected** if there is a path between them

Problems:

1) Is there a path between s and t ?
2) What is the shortest path between s and t ?
3) Is there a cycle in the graph?
4) Euler tour - Is there a cycle that uses each edge exactly once?

5) Hamilton tour - Is there a cycle that uses each vertex exactly once?

6) Connectivity - Is there a way to connect all of the vertices?

7) MST - What is the best way to connect all of the vertices?

8) Biconnectivity - is there a vertex whose removal disconnects the graph?

9) Planarity - can you draw the graph in the plane with no edges crossing?

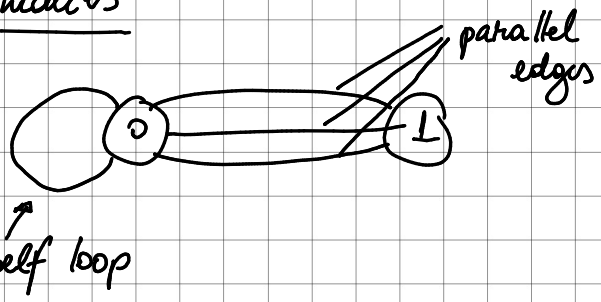
10) Graph isomorphism - do two adjacency lists represent the same graph?

Graph API

Vertex Representation

- use integers between 0 and (V-1)
- convert between names and integers with symbol table

Anomalies



class Graph

- Graph (int v)
- void addEdge (int v, int w)
- Iterable < Integer > adjacent (int v)
- int V() - number of vertices
- int E() - number of edges
- String toString ()

Set-of-edges representation

- maintain a list of edges { linked list / array

Adjacency-matrix graph representation

- 2D V by V boolean array

- for each edge v-w in graph

  adj[v][w] = adj[w][v] = true

- space is a concern

Adjacency-list graph representation

- maintain a vertex-indexed array of lists

* In practice - use adjacency-list graph representation

  - algorithms are based on iterating over vertices adjacent to v

  - real-world graphs tend to be sparse = huge number of vertices, small average vertex degree

Depth-First Search

* Maze exploration → maze graph

  vertex = intersection

  edge = passage

* Goal = explore every intersection (vertex) in the maze

* avoid going to the same place twice

Depth-First Search

Goal : systematically search through a graph

Idea : Mimic maze exploration

DFS (to visit a vertex V)

- mark V as visited
- recursively visit all unmarked vertices w adjacent to V

Typical Applications

- find a path between 2 vertices
- find all vertices connected to a given source vertex

* Design Pattern

- decouple graph data type from graph processing
- create a graph object
- pass graph through a graph-processing routine for information
- query the graph processing routine for information

public class Paths

- Paths(Graph g, int s) -> s = source vertex
- boolean hasPathTo(int v)
- Iterable<Integer> pathTo(int v)

- marked[] – array of booleans => visited
- edgeTo[v] – the vertex we are coming from

keeps the path

* edgeTo[w] == v means that edge (v-w) was taken to visit (w) for the first time

Properties

- DFS marks all vertices connected to s in time proportional to the sum of their degrees
- After (DFS) => client can find vertices connected to s in constant time and can find a path to (s) (if exists) in time proportional to its length.

(*) (edgeTo[]) = parent-link representation of a tree rooted at s

DFS = expand the next thing you come to

Breadth-First Search

- Repeat until queue is empty
  - remove vertex (v) from queue
  - add to queue all unmarked vertices adjacent to v and mark them
  - edgeTo[] - same as for DFS
  - distTo[] - size of path

* Depth-first search - put unvisited vertices on a STACK
* Breadth-first search - put unvisited vertices on a QUEUE
* Shortest path - path from s to t that uses fewest number of edges

BFS (from source vertex s)

- put s onto a FIFO queue and mark s as visited
- Repeat until the queue is empty
  - remove the least recently added vertex v
  - add each of v's unvisited neighbors to the queue and mark them as visited

* BFS examines vertices in increasing distance from s
* BFS computes shortest paths (fewest number of edges) from s to all other vertices in graph in time proportional to $E+V$

BFS : routing

- fewest number of hops in a communication network
- oracle of bacon

Connected Components

Def vertices v and w are connected if there is a path between them

Goal Preprocess graph to answer questions like is v connected to w in constant time

class Connected Components

- boolean connected (int v, int w)
- int count() - no. of connected comp.
- int id (int v) - comm. comp. identifier

- union-find? not quite
- depth-first search

* The relation "is connected" is an equivalence relation ← reflexive (v conn. v), symmetric, transitive

Def A connected component is a maximal set of connected vertices

id[ ] — component identifier

Goal Partition the vertices into connected components.

- initialize all vertices v as unmarked
- for each unmarked vertex v, run dfs to identify all vertices discovered as part of the same component

Graph Challenges

1) Is a given graph bipartite?

Bipartite = divide the vertices into 2 subsets

with the property that every edge connects a vertex from one set with a vertex from the other subset.

- simple dfs solution → textbook

2) does a graph contain a cycle?

DFS

3) Bridges of königsberg

- is there a cycle that uses each edge exactly once?
- a connected graph is Eulerian iff all vertices have even degree.

4) Find a cycle that uses every edge exactly once

5) Find a cycle that visits every vertex exactly once

* this is called the travelling salesman

* Hamiltonian-cycle / classical NP complete problem

6) Are two graphs identical except the vertex names?

- graph isomorphism - no one knows

7) Lay out a graph in the plane without crossing edges

- linear-time DFS-based planarity algorithm (Tarjan)