Graphs

What is a Graph?

(in computer science, it's not a data plot)
General structure for representing positions with an arbitrary connectivity structure

- Collection of **vertices** (nodes) and **edges** (arcs)
  - Edge is a pair of vertices - it connects the two vertices, making them **adjacent**
  - A tree is a special type of graph!

What can graphs represent?

- City map
- Computer network
- Transportation system
- Electrical wiring
- etc.

What can we do with graphs?

- Find a **path** from one place to another
- Find the **shortest path** from one place to another
- Find the “weakest link”
  - check amount of redundancy in case of failures
- Draw them

Types of Graphs

- Undirected / directed
  - Edges are symmetric / one-way
- Acyclic
  - no path of unique edges starts and ends at same vertex
- Connected
  - There is a path between each pair of nodes
- Forest: acyclic graph
- Tree: connected forest (not necessarily rooted)

(undirected) Graph ADT

- **numVertices()**, **numEdges()**: return # of vertices or edges
- **vertices()**, **edges()**: return iterator of vertices or edges
- **degree(v)**: return # of incident edges on a vertex
- **incidentEdges(v)**: return iterator of incident edges on vertex
- **endVertices(e)**: return two vertices of edge
- **opposite(v, e)**: return endpoint of e that is not v
- **areAdjacent(v, w)**: return whether an edge connects v to w
- **insertEdge(v, w, o)**: create and return an edge between v and w storing object o
- **insertVertex(o)**: insert and return new vertex storing o
- **removeVertex(v)**: remove vertex v and its adjacent edges
- **removeEdge(e)**: remove edge
Concrete graph representations

Edge List: simple but inefficient in time
Adjacency List: moderately simple and efficient
Adjacency Matrix: simple but inefficient in space

Edge List

Container (list/vector/dictionary) of vertices
- Each vertex just has its object

Container (list/vector/dictionary) of edges
- Each edge has its object
- Edge also has references to its two endpoint vertices

Edge list (linked list) efficiency

vertices( ) : $O(n)$
edges( ) : $O(m)$
endVertices(e): $O(1)$
incidentEdges(v): $O(m)$
areAdjacent(v, w): $O(m)$
removeEdge(e): $O(1)$
removeVertex(v): $O(m)$

Adjacency List

Similar to Edge List
Each vertex also has container of references to incident edges

Adjacency list (linked list) efficiency

vertices( ) : $O(n)$
edges( ) : $O(m)$
endVertices(e): $O(1)$
incidentEdges(v): $O(deg(v))$
areAdjacent(v, w): $O(min(deg(v), deg(w)))$
removeEdge(e): $O(deg(u)+deg(v))$
removeVertex(v): $O(deg(v)+\sum deg(u))$

(note: the last two are incorrect in the textbook)

Adjacency Matrix

Extend edge list with $v \times v$ array
- each entry holds null reference or reference to edge connected vertex $i$ to vertex $j$
Adjacency Matrix efficiency

- vertices(): \( O(n) \)
- edges(): \( O(m) \)
- endVertices(e): \( O(1) \)
- incidentEdges(v): \( O(n) \)
- areAdjacent(v, w): \( O(1) \)
- removeEdge(e): \( O(1) \)
- removeVertex(v): \( O(n^2) \)
  - perhaps \( O(n) \) with amortization

Traversing Graphs

Traversal visits all nodes and edges of graph (preferably in linear time)

- Depth-first search
- Breadth-first search

Depth-first Search (DFS)

Basic approach
- Visit node, then recursively visit children
- Traverse a path all the way to deadend before traversing other paths

First, label all vertices and edges as unvisited

DFS(G, v)
for all edges, e, in G.incidentEdges(v) do
  if e is unvisited then
    w = G.opposite(v, e)
    if w is unvisited then
      label e as tree edge
      DFS(G, w)
    else
      label as back edge
  else
    continue

Performance of DFS

Each vertex is visited exactly once
Each edge is used exactly once
Each edge is considered exactly twice
Run time is \( O(n + m) \)

Uses for DFS

All \( n \) nodes and \( m \) edges are visited

- if graph is not connected, all nodes and edges in connected component are visited

Useful for:
- Find a spanning tree of a graph
- Find a path between two vertices
- Find all connected components of a graph
- Finding a cycle (if any) in a graph

Breadth-first search

Basic approach
- Visit a node, then put all its children on a queue to be visited
- Visit nodes in order of queue
  - visits “close” nodes first, then “farther” nodes

BFS(G, s)
mark all vertices and edges unvisited
Initialize queue, \( Q \) to contain vertex, \( s \)
while not \( Q \).isEmpty() do
  v = Q.dequeue(), mark v visited
  for each edge, e of v do
    if e is unvisited then
      w = G.other(v, e)
      if w is unvisited then
        label e as tree edge, \( Q\).enqueue(w)
      else
        label e as cross edge