Graphs

- Graph is a non-linear data structure (like a tree).

*Note: nonlinear data structures do not have their elements in a sequence.*
- But trees are somewhat orderly. Root is connected to its children. The children are connected to their children…all the way to the leaves.
- Graph can be linked in any pattern.

- A graph consists of vertices (nodes) and edges (arcs).

  - Name the vertices: \{A,B,C,D,E\}
  - Name the edges: \{(A,B), (A,E), (A,D),…\}

*Applications:*
  - roads between cities
  - communication networks
  - oil pipelines
  - personal relationships between people
  - interval graphs
  - rooms in caves with passages between them

*Directed vs. Undirected*
  - Graphs can be directed or undirected.
  - In undirected graphs, edges have no specific direction. Edges are always “two-way”
  - In directed graphs (also called digraphs), edges have a direction
- A self-edge a.k.a. a loop is an edge of the form (a,a)

- Graph Connectivity
  - A graph is connected when there is a path between every pair of vertices.
  - In a connected graph, there are no unreachable vertices.
  - A graph that is not connected is disconnected.
  - A graph G is said to be disconnected if there exist two nodes in G such that no path in G has those nodes as endpoints.

- Weighted graphs
  - In a weighed graph, each edge has a weight a.k.a. cost
  - Some graphs allow negative weights; some not

- Paths and Cycles
Path from one vertex to another is a sequence of vertices, each a neighbor of the previous one.
- Path: [0, 1, 3]

A cycle is a path that begins and ends at the same node
- Cycle: [1, 3, 4, 1]

Path length: Number of edges in a path
- Path: [0, 1, 3]
  - Path length: 3
  - Path cost: 4

- Graphs and Trees
  - A tree is a graph that is acyclic (graph with no cycles) and connected
  - So all trees are graphs, but not all graphs are trees

*Note: A directed acyclic graph is called a DAG*

- More terminologies
  - Degree of a vertex: number of edges containing that vertex i.e. the number of adjacent vertices
  - In-degree of a vertex: number of in-bound edges, i.e., edges where the vertex is the destination
  - Out-degree of a vertex: number of out-bound edges i.e., edges where the vertex is the source

- Dense vs. Sparse
  - A dense graph is a graph in which the number of edges is close to the maximal number of edges.
  - A graph with only a few edges, is a sparse graph
Representing Graphs

1. Adjacency matrix

- An adjacency matrix is a square matrix in which each element stores one edge of the graph.
- The matrix is represented in memory as a two-dimensional array consisting of $n$ rows and $n$ columns, where $n$ is the number of vertices in the graph.
- This scheme can be used when the vertices are represented using an array.
- An entry of 1 in element $[i][j]$ of the matrix indicates that the edge from vertex $i$ to vertex $j$ is present in the graph. Otherwise, the element is set to 0.

Undirected Graph:

Since, in an undirected graph, the travel along the edges is bidirectional (2-way), if element $[i][j]$ is 1, element $[j][i]$ is also 1. Consequently, the adjacency matrix for an undirected graph is always symmetric.

This fact can save some time in algorithms that process undirected graphs because only half the array elements need to be fetched from memory. The values of the other half of the elements can be determined from those by reversing the indices (i.e., edge $[i][j] = \text{edge } [j][i]$).

Digraph:

Travel along the edges of a digraph is not bidirectional and, therefore, the adjacency matrix of a directed graph is usually not symmetric. The matrix of a digraph is symmetric only when there are two edges between pairs of vertices.
Advantages of the adjacency matrix:
Constant time lookup of an edge. Using the linked representation of graphs, you need to look through the list of neighbors of the node to find whether the other node is a neighbor.

Disadvantages of the adjacency matrix:
See below.

Aside from the four basic operations (Insert, Fetch, Delete, and Update) a common operation performed on graphs is to determine which vertices are adjacent to a given vertex, for example, vertex \( v_i \).

To accomplish this in an undirected graph, we simply examine the elements of row \( i \) of the adjacency matrix. If there is a nonzero entry stored in element \([i][j]\) of the array, then the graph contains an edge from \( v_i \) to \( v_j \), and the two vertices are adjacent.

Although the algorithm is straightforward, it does perform \( n \) memory accesses for a graph with \( n \) vertices; the algorithm is \( O(n) \).

Thus, even if there were only one edge emanating from a vertex in a graph that contained 1000 vertices, the algorithm would perform 1000 memory accesses to locate the adjacent vertex.

From a space complexity viewpoint an adjacency matrix can be an inefficient way to represent a graph.

Consider the case when each of the 1000 vertices in a directed graph has two adjacent edges. In this case only 2000 of the 1,000,000 elements of the matrix would contain a 1. The remaining 998,000 elements would store a 0. A matrix such as this, in which most of its elements contain a default value (in our case 0, to represent no edge) is called a sparse matrix.

For that reason, the adjacency matrix implementation should only be used if you know in advance that your graph is going to have either a small number of vertices or a lot of edges.

From a space (and time) complexity viewpoint, sparse matrices are better represented as a set of linked lists. Enter the adjacency list.

2. Adjacency lists
- An adjacency list is a set of \( n \) linked lists, one list per vertex (the neighbors of this vertex).
- The first linked list stores the edges emanating from vertex 0, the second linked list stores edges emanating from vertex 1, etc.
- Each node on the linked list contains at least two pieces of information, the vertex number of the edge it is incident upon and, of course, the location of the next node in the linked list.

Undirected Graph:
What is its disadvantage?  
[Very slow to lookup a neighbor if the graph is nearly complete. Time is O(n).] This implementation is good if, instead of searching whether two nodes are connected by an edge, you are interested in traversing the graph.

This is a good OO implementation since each GraphNode object is responsible for telling you its neighbors instead of the graph itself storing all that information.

This implementation is also much more like the implementation of binary trees in that each node keeps track of the neighbor or children pointers.

However, in the case of a graph, you need an list (or set) instead of just two pointers, since a node can have arbitrarily many children or neighbors.

A Graph corresponds in this way to a BinaryTree object and the nodes correspond to nodes. But there is one more difference: Why does a Graph maintain a list of nodes, instead of just one starting "root" node, like a BinaryTree does? [A graph might be disconnected]

Graph operations are as follows:
- initialize a graph by specifying its vertices, edges and (any) weighting factors;
- addition, deletion of vertices and edges, and modification of weighting factors;
- visiting vertices by ‘traversal’ of a graph’s edges;
- finding a path between two vertices.
Optional Exercises:

1. Give the adjacency matrix for the graphs A and B shown below.

![Graph A and B](image)

2. Give the adjacency list representation of the graphs A and B

3. Draw the graph whose edges are represented by the following matrices. Assume the vertices are named $V_0$ through $V_4$.

   \[
   \begin{array}{cccccc}
   0 & 1 & 2 & 3 & 4 \\
   0 & 1 & 0 & 1 & 1 \\
   1 & 0 & 0 & 0 & 1 \\
   2 & 0 & 1 & 0 & 1 \\
   3 & 1 & 0 & 0 & 0 \\
   4 & 0 & 1 & 0 & 1 \\
   \end{array}
   \]

   \[
   \begin{array}{cccccc}
   0 & 1 & 2 & 3 & 4 \\
   0 & 5 & 0 & 0 & 9 \\
   1 & 3 & 0 & 0 & 1 \\
   2 & 0 & 0 & 0 & 0 \\
   3 & 0 & 0 & 4 & 0 \\
   4 & 0 & 0 & 7 & 0 \\
   \end{array}
   \]

   \[
   \begin{array}{cccccc}
   0 & 1 & 2 & 3 & 4 \\
   0 & 1 & 0 & 1 & 1 \\
   1 & 1 & 0 & 1 & 0 \\
   2 & 0 & 1 & 0 & 0 \\
   3 & 1 & 0 & 0 & 0 \\
   4 & 1 & 0 & 0 & 0 \\
   \end{array}
   \]

4. Consider the graphs in the previous exercise. Which of them are:
   - Disjoint?
   - Directed?
   - Weighted?
   - Undirected?

5. Here is an adjacency list representation of a directed graph where there are no weights assigned to the edges)
a. Draw a picture of the directed graph that has the above adjacency list representation

b. Draw the adjacency matrix for this graph

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Graph problems**

Now, what do people do with graphs?
1. Is there a path that uses each edge exactly once? [Eulerian Path Problem]
2. Is there a path that goes through each node exactly once? [Hamilton Path Problem]
3. Is there a circuit that uses each edge exactly once? [Eulerian Circuit Problem]
4. Is there a circuit that goes through each node exactly once? [Hamilton Circuit Problem]
5. Does it have particular data in it? [a search problem]
6. Is it connected?
7. Are there any cycles?
8. Is there a path between two given nodes? If so, what is it?
9. What is the shortest path (least number of edges), if any, between two given nodes? [Shortest Path Problem]
10. What is the shortest path (sum of weights of edges) between two given nodes?
11. What are the lengths of the shortest paths between all pairs of nodes? [All-pairs Shortest Path Problem]
12. What is the shortest tour of all nodes in a complete graph (where “shortest” means the least sum of the weights of edges)? [Travelling Salesperson Problem]
13. Can the nodes be colored red and blue so that red nodes have edges only to blue nodes and vice versa? [Bipartite Graph Problem]
14. What is the fewest number of colors needed to color the nodes so that adjacent nodes have different colors? [Chromatic Number of a graph]
15. Can the graph be drawn in the plane so that no edges cross? [Planarity Problem]
16. How many paths of a given length are there between two given nodes?
17. Can each node be represented by an interval on the real line so that adjacent nodes correspond to overlapping intervals? [Interval Graph Problem]

**Graph Traversals**

Starting at node v, find all nodes reachable from v (i.e., there exists a path from v)

1. **Depth First Search (DFS)**
   - DFS starts at a node (often called the root and it can be any node) and goes down as far as it can down a branch (until you reach a dead end) before backtracking (back up) and follow another branch.
   - DFS generally uses a stack in order to keep track of visited nodes, as the last node seen is the next one to be visited and the rest are stored to be visited later.

**DFS Steps:**
- Visit a vertex v
- Mark v as being visited.
- Explore each adjacent vertex that is unvisited.

**Complexity:** Depth-first search visits every vertex (V) once and checks every edge (E) in the graph once. Therefore, DFS complexity is O(V+E).

**Applications:**
- Finding a Spanning Tree
- Finding Paths
- Finding a cycle

**Example - board**

2. **Breadth-First Search (BFS)**
   - Breadth-first search starts by searching a start vertex s, then visits all vertices that are k edges away from the source vertex s before visiting any vertex k+1 edges away. This is
done until no more vertices are reachable from s.

- For breadth-first search, we choose the vertex from the unvisited set that was least recently encountered. This corresponds using a queue to hold vertices on the unvisited set.

Complexity: Breadth-first search has a running time of $O(V+E)$ since every vertex and every edge will be checked once. Depending on the input to the graph, $O(E)$ could be between $O(1)$ and $O(V^2)$.

Example - board

BFS vs. DFS:
- BFS is better at: finding shortest path
- DFS is better at finding connectivity

Visualization: hfp://visualgo.net/dfsdfs.html

Spanning Trees

- A graph's spanning tree is a tree that contains (possibly) all of the vertices of the graph connected by a subset of the graph's edges.
- The edges are chosen such that is there is a path from each vertex to every other vertex, and (since it is a tree) there are no cycles.
- Most graphs have more than one spanning tree.

Applications
Spanning trees help provide solutions to a range of problems. These include efficient design of data networks, and the routes chosen to reduce mileage and costs in the distribution of goods from a central warehouse to various retail outlets.

![Graph with spanning trees](image)

has the following spanning trees
3. Dijkstra’s Algorithm (Shortest Path Algorithm)

- The task is to find the shortest (least-weight) path between a starting point and a destination. It runs on a weighted graph.
- You start with an initial node and find the least cost path to the goal node.

Note: ‘weight’ does not always mean distance

Applications

Establishing a shortest route through data represented as a graph is necessary for many purposes. Here are three examples:

1. A journey between two locations usually requires finding the most efficient path. In the case of a road journey, this may simply involve finding the least distance to minimize use of fuel; however, some routes may be subject to toll charges, so that least cost may not always be shortest distance. Navigation and online mapping services offer alternatives to take these factors into account.

2. Planning an air trip means investigating routes, services and also the fares offered by competing airlines; taking a longer route of two or more flights through a ‘hub’ airport may be cheaper than a direct flight. Here, the shortest route means the least expensive one, if the weighting factors are financial.

3. Routers transmit data packets using strategies based upon shortest paths. They attempt to send data across a network using the least costly routes available.

[Material for Graphs lectures adapted from Data Structures And Algorithms using Java by William McAllister]