Data Structures/Graphs

Data Structures
Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs

Graphs

A **graph** is a structure consisting of a set of vertices and a set of edges . An edge is a pair of vertices . The two vertices are called the edge *endpoints*. Graphs are ubiquitous in computer science. They are used to model real-world systems such as the Internet (each node represents a router and each edge represents a connection between routers); airline connections (each node is an airport and each edge is a flight); or a city road network (each node represents an intersection and each edge represents a block). The wireframe drawings in computer graphics are another example of graphs.

A graph may be either *undirected* or *directed*. Intuitively, an undirected edge models a "two-way" or "duplex" connection between its endpoints, while a directed edge is a one-way connection, and is typically drawn as an arrow. A directed edge is often called an *arc*. Mathematically, an undirected edge is an unordered pair of vertices, and an arc is an ordered pair. For example, a road network might be modeled as a directed graph, with one-way streets indicated by an arrow between endpoints in the appropriate direction, and two-way streets shown by a pair of parallel directed edges going both directions between the endpoints. You might ask, why not use a single *undirected* edge for a two-way street. There's no theoretical problem with this, but from a practical programming standpoint, it's generally simpler and less error-prone to stick with all directed or all undirected edges.

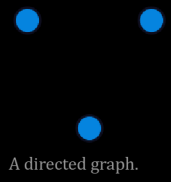
An undirected graph can have at most edges (one for each unordered pair), while a directed graph can have at most edges (one per ordered pair). A graph is called *sparse* if it has many fewer than this many edges (typically edges), and *dense* if it has closer to edges. A **multigraph** can have more than one edge between the same two vertices. For example, if one were modeling airline flights, there might be multiple flights between two cities, occurring at different times of the day.

A **path** in a graph is a sequence of vertices such that there exists an edge or arc between consecutive vertices. The path is called a **cycle** if . An undirected acyclic graph is equivalent to an undirected tree. A directed acyclic graph is called a **DAG**. It is not necessarily a tree.

Nodes and edges often have associated information, such as *labels* or *weights*. For example, in a graph of airline flights, a node might be labeled with the name of the corresponding airport, and an edge might have a weight equal to the flight time. The popular game "Six Degrees of Kevin Bacon" can be modeled by a labeled undirected graph. Each actor becomes a node, labeled by the actor's name. Nodes are connected by an edge when the two actors appeared together in some movie. We can label this edge by the name of the movie. Deciding if an actor is separated from Kevin Bacon by six or fewer steps is equivalent to finding a path of length at most six in the graph between Bacon's vertex and the other actors vertex. (This can be done with the breadth-first search algorithm found in the companion Algorithms book. The Oracle of Bacon at the University of Virginia has actually implemented this algorithm and can tell you the path from any actor to Kevin Bacon in a few clicks_____.)

Directed Graphs

The number of edges with one endpoint on a given vertex is called that vertex's **degree**. In a directed graph, the number of edges that point *to* a given vertex is called its **in-degree**, and the number that point *from* it is called its **out-degree**. Often, we may want to be able to distinguish between different nodes and edges. We can associate labels with either. We call such a graph **labeled**.



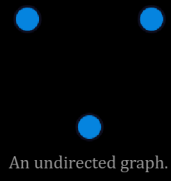
A directed graph.

Directed Graph Operations

`make-graph(): graph`
Create a new graph, initially with no nodes or edges.
`make-vertex(graph G, element value): vertex`
Create a new vertex, with the given value.
`make-edge(vertex u, vertex v): edge`
Create an edge between u and v. In a directed graph, the edge will flow from u to v.
`get-edges(vertex v): edge-set`
Returns the set of edges flowing from v
`get-neighbors(vertex v): vertex-set`
Returns the set of vertices connected to v

Undirected Graphs

In a directed graph, the edges point from one vertex to another, while in an **undirected graph**, they merely connect two vertices. we can travel forward or backward.It is a bidirectional graph.



An undirected graph.

Weighted Graphs

We may also want to associate some cost or weight to the traversal of an edge. When we add this information, the graph is called **weighted**. An example of a weighted graph would be the distance between the capitals of a set of countries.

Directed and undirected graphs may both be weighted. The operations on a weighted graph are the same with addition of a weight parameter during edge creation:

Weighted Graph Operations (an extension of undirected/directed graph operations)

`make-edge(vertex u, vertex v, weight w): edge`
Create an edge between u and v with weight w. In a directed graph, the edge will flow from u to v.

Graph Representations

Adjacency Matrix Representation

An **adjacency matrix** is one of the two common ways to represent a graph. The adjacency matrix shows which nodes are **adjacent** to one another. Two nodes are adjacent if there is an edge connecting them. In the case of a directed graph, if node is adjacent to node , there is an edge from to . In other words, if is adjacent to , you can get from to by traversing one edge. For a given graph with nodes, the adjacency matrix will have dimensions of . For an unweighted graph, the adjacency matrix will be populated with boolean values.

For any given node , you can determine its adjacent nodes by looking at row of the adjacency matrix. A value of true at indicates that there is an edge from node to node , and false indicating no edge. In an undirected graph, the values of and will be equal. In a weighted graph, the boolean values will be replaced by the weight of the edge connecting the two nodes, with a special value that indicates the absence of an edge.

The memory use of an adjacency matrix is .

Adjacency List Representation

The adjacency list is another common representation of a graph. There are many ways to implement this adjacency representation. One way is to have the graph maintain a list of lists, in which the first list is a list of indices corresponding to each node in the graph. Each of these refer to another list that stores the index of each adjacent node to this one. It might also be useful to associate the weight of each link with the adjacent node in this list.

Example: An undirected graph contains four nodes 1, 2, 3 and 4. 1 is linked to 2 and 3. 2 is linked to 3. 3 is linked to 4.

1 - [2, 3]

2 - [1, 3]

3 - [1, 2, 4]

4 - [3]

It might be useful to store the list of all the nodes in the graph in a hash table. The keys then would correspond to the indices of each node and the value would be a reference to the list of adjacent node indices.

Another implementation might require that each node keep a list of its adjacent nodes.

Graph Traversals

In a perfect world we would have full knowledge of the graph's contents, letting us optimize indices for the vertices and edges for efficient look-ups. But there are numerous open-ended problems in computer science where indexing is impractical or even impossible. Graph traversals let us tackle these problems. The traversals let us efficiently search large spaces where objects are defined by their relationships to other objects rather than properties of the objects themselves.

Depth-First Search

Start at vertex a, visit its neighbour b, then b's neighbour c and keep going until reach 'a dead end' then iterate back and visit nodes reachable from second last visited vertex and keep applying the same principle.

```
// Search in the subgraph for a node matching 'criteria'. Do not re-examine
// nodes listed in 'visited' which have already been tested.
GraphNode depth_first_search(GraphNode node, Predicate criteria, VisitedSet visited) {
// Check that we haven't already visited this part of the graph
if (visited.contains(node)) {
 return null;
}
visited.insert(node);
// Test to see if this node satisfies the criteria
if (criteria.apply(node.value)) {
 return node;
}
// Search adjacent nodes for a match
for (adjacent in node.adjacentnodes()) {
 GraphNode ret = depth_first_search(adjacent, criteria, visited);
 if (ret != null) {
  return ret;
 }
}
// Give up - not in this part of the graph
return null;
}
```

Breadth-First Search

Breadth first search visits the nodes neighbours and then the unvisited neighbours of the neighbours, etc. If it starts on vertex a it will go to all vertices that have an edge from a. If some points are not reachable it will have to start another BFS from a new vertex.

Data Structures
Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs