# Graphs

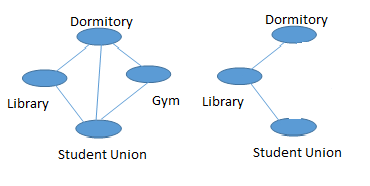
Graphs are an important mathematical concept with significant applications not only in computer science, but also in many other fields. You can view a graph as a mathematical construct, a data structure, or an abstract data type. This chapter provides an introduction to graphs that allows you to view a graph in any of these three ways. It also presents the major operations and applications of graphs that are relevant to the computer scientist.

## Terminology

You are undoubtedly familiar with graphs: Line graphs, bar graphs, and pie charts are in common use. The simple line graph in Figure XX-1 is an example of the type of graph that this chapter considers: a set of points that are joined by lines. Clearly, graphs provide a way to illustrate data. However, graphs also represent the relationships among data items, and it is this feature of graphs that is important here.

A **graph** G consists of two sets: a set V of vertices, or nodes, and a set E of edges that connect the vertices. For example, the campus map in Figure 20-2a is a graph whose vertices represent buildings and whose edges represent the sidewalks between the buildings. This definition of a graph is more general than the definition of a line graph. In fact, a line graph, with its points and lines, is a special case of the general definition of a graph.

Figure 1A graph and one of its subsets



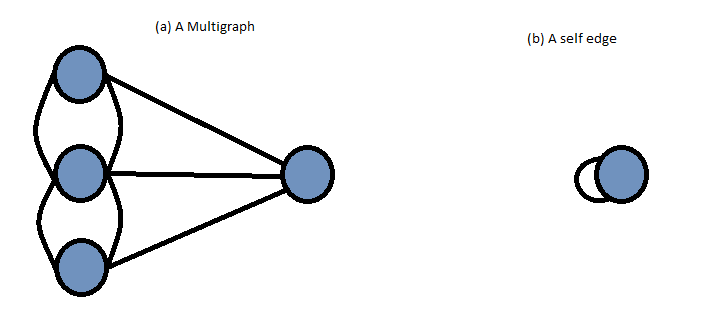
A **subgraph** consists of a subset of a graph’s vertices and a subset of its edges. Figure 20-2b shows a subgraph of the graph in Figure 20-2a. Two vertices of a graph are **adjacent** if they are joined by an edge. In Figure 20-2b, the Library and the Student Union are adjacent. A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex. For example, there is a path in Figure 20-2a that begins at the Dormitory, leads first to the Library, then to the Student Union, and finally back to the Library. Although a path may pass through the same vertex more than once, as the path just described does, a **simple path** may not. The path Dormitory-Library-Student Union is a simple path. A **cycle** is a path that begins and ends at the same vertex; a simple cycle is a cycle that does not pass through other vertices more than once. A graph is **connected** if each pair of distinct vertices has a path between them. That is, in a connected graph, you can get from any vertex to any other vertex by following a path. A **disconnected** graph is one that has one or more vertices that cannot be reached.

In a **complete** graph, each pair of distinct vertices has an edge between them. That is, a complete graph has as many edges as possible. Clearly, a complete graph is a connected graph. That is not necessarily true the other way though.

A graph with n vertices has at most n \* (n -1) / 2 edges; a complete graph will have the maximal number of edges. A graph that has almost the maximum number of edges is said to be **dense**. Conversely, a **sparse** graph has relatively few, or O(n), edges.

Because a graph has a set of edges, a graph cannot have duplicate edges between vertices. However, a multigraph, as illustrated in Figure 20-4a, does allow multiple edges. Thus a multigraph is not a graph. A graph’s edges cannot begin and end at the same vertex, Figure 2 shows such an edge, which is called a **self edge** or **loop**.

Figure 2 Graph-Like Structures that are not Graphs



You can label the edges of a graph. When these labels represent numeric values, the graph is called a **weighted** graph.

All the previous graphs are examples of **undirected** graphs, because the edges do not indicate a direction. That is, you can travel in either direction along the edges between the vertices of an undirected graph. In contrast, each edge in a **directed** graph, or **digraph**, has a direction and is called a **directed edge**. Although each distinct pair of vertices in an undirected graph has only one edge between them, a directed graph can have two edges between a pair of vertices, one in each direction.

## Graphs as ADTs

You can treat a graph as an abstract data types. Addition and removal operations are somewhat different for graphs than for other ADTs that you have studied, in that they apply to either vertices or edges. In an ADT, you can choose weather a vertices have values or not. However, the following ADT graph operations do assume that the graph’s vertices contain values.

Note: ADT Graph Operations

* Test whether a graph is empty
* Get the number of vertices in a graph
* Get the number of edges in a graph
* See whether an edge exists between two given vertices
* Add a new vertex to a graph whose vertices have distinct values that differ from the new vertex’s value
* Add a new edge between two given vertices in a graph
* Remove a particular vertex from a graph and any edges between the vertex and other vertices.
* Remove the edge between two given vertices in a graph
* Retrieve from a graph the vertex that contains a given value

Several variations of this ADT are possible. For example, if the graph is directed, you can replace occurrences of “edges” with “directed edges”. You can also add traversal operations to the ADT. Grapt-traversal algorithms are discussed in a later section.

The following listing contains an interface that specifies in more detail the ADT operations for an undirected graph.

/\*\* An interface for the ADT undirected, connected graph.

@file GraphInterface.h \*/

#ifndef GRAPH\_INTERFACE

#define GRAPH\_INTERFACE

Template <class LabelType>

Class GraphInterface

{

virtual int getNumVertices() const = 0;

virtual int getNumEdges() const = 0;

virtual bool add(LabelType start, LabelType end, int edgeWeight) = 0;

virtual bool remove(LabelType start, LabelType end) = 0;

virtual int getEdgeWeight(LabelType start, LabelType end) = 0;

virtual void depthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;

virtual void breathFirstTraversal(LableType start, void visit(LabelType&)) = 0;

virtual ~GraphInterface() {}

}; // end GraphInterface

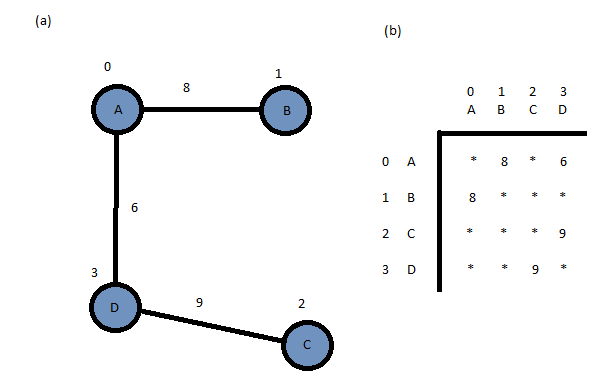
#endif

### Implementing Graphs

The two most common implementations of a graph are the adjacency matrix and the adjacency list. An **adjacency matrix** for a graph with n vertices numbered 0, 1, …, n-1 is an n by n array matrix such that matrix[i][j] is 1 (true) if there is an edge from vertex I to vertex j, and 0 (false) otherwise.

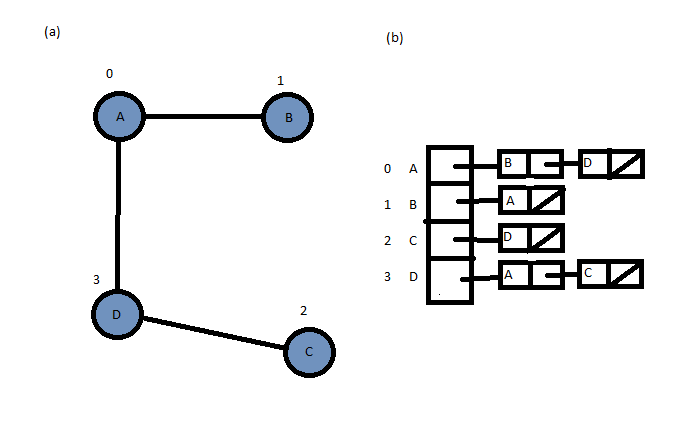
Our definition of an adjacency matrix does not mention the value. If any value is required in a vertex, you will also have an array of vertices that contains a secondary list. For a weighted graph, see example 3.

Figure 3 A Weighted undirected graph



An **adjacency list** for a graph with n vertices numbered 0, 1, … n-1 consists of n lined lists. The ith linked list has a node for vertex j if and only if the graph contains an edge from vertex i to vertex j. Figure 4 shows an example.

Figure 4 An Adjacency List



### Graph Traversals

A **graph traversal** will not stop until it has visited each node in the graph that it can reach that has not already been visited. Unlike a tree, the graph traversal will not visit all the vertices unless they are connected to your starting point.

If a graph is not connected, a graph traversal that begins at vertex v will visit only a subset of the graph’s vertices. This subset is called a **connected component** containing v. You can find all of the connected components of a graph by repeatedly starting a traversal at an unvisited vertex.

If the graph contains a cycle, a graph traversal algorithm can loop indefinitely. To prevent such a misfortune, the algorithm must mark each vertex during a visit and must never visit a vertex more than once.

There are two basic graph traversal algorithms. These apply to both directed and undirected graphs. They are **depth-first search** (DFS) and **breath-first search** (BFS) algorithms.

### Depth-First Search

From a given vertex v, the depth-first search (DFS) strategy of graph traversal proceeds along a path from v as deeply into the graph as possible before backing up. That is, after visiting a vertex, a DFS visits, if possible, an unvisited adjacent vertex.

dfs(v: Vertex)

{

Mark v as visited

For (each unvisited vertex u adjacent to v)

dfs(u)

}

The depth-first search algorithm does not completely specify the order in which it should visit the vertices adjacent to v. One possibility is to visit the vertices adjacent to v in sorted order. This possibility is natural either when an adjacency matrix represents the graph or when the nodes in each linked list f an adjacency list are linked in sorted order.

An interactive version of the DFS algorithm is also possible by using a stack:

dfs(v: Vertex)

{

s = a new empty stack

s.push(v)

Mark v as visited

while (! s.isEmpty())

{

if (no unvisited vertices are adjacent to the vertex)

s.pop()

else

{

Select an unvisited vertex u adjacent to the vertex v

s.push(u)

mark u as visited

}

}

}

### Breadth-First Search

After visiting a given vertex v, the breadth-first search (BFS) strategy of graph traversal visits every vertex adjacent to v that it can before visiting any other vertex. After marking and visiting v, the BFS traversal algorithm marks and then visits each of the vertices u, w and x. Since no other vertices are adjacent to v, the BFS algorithm visits, if possible, all unvisited vertices adjacent to u. Thus, the traversal visits q and t.

A BFS traversal will not embark from any of the vertices adjacent to v until it has visited all possible vertices adjacent to v. Whereas a DFS is a last visited, first explored strategy, a BFS is a first visited, first explored strategy. It is not surprising, then, that a breath-first search uses a queue. An interactive version of this algorithm follows.

bfs(v: Vertex)

{

q = a new empty queue

q.enqueue(v)

Mark v as visited

While (! q.isEmpty())

{

q.dequeue(w)

for (each unvisited vertex u adjacent to w)

{

Mark u as visited

q.enqueue(u)

}

}

}

## Applications of Graphs

There are many useful applications of graphs. This section surveys some of these common applications.

### Topological Sorting