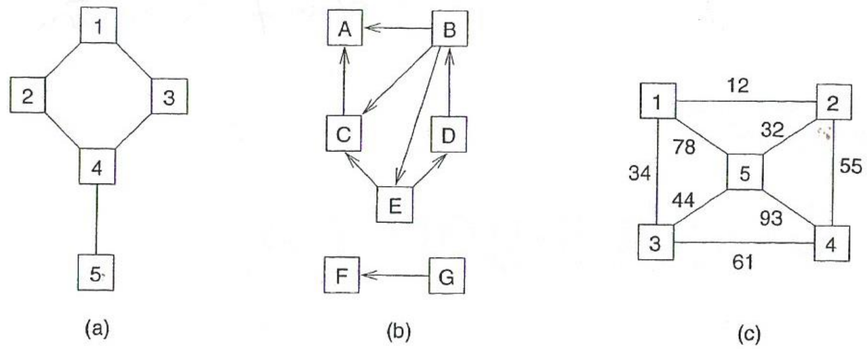

Figure 7.1 Examples of graphs. (a) Undirected graph; (b) directed graph; (c) weighted (undirected) graph.

middle is *directed* (all the arcs are pointing in one direction). Therefore, just like a tree, a directed graph is also a non-linear data structure since an item can have more than one successor. The difference with trees is that in a graph, a node can have zero or more predecessors whereas each node in a tree (except the root) has only one predecessor.

A *path* from v_1 to v_n is a sequence of nodes $v_1, v_2, \ldots, v_n$ such that every pair $v_{i-1}v_i$ constitutes an edge (or arc). A *simple path* is a path where all the v_i in the path are distinct. A *cycle* is like a simple path except that $v_1 = v_n$, that is, it starts and ends at the same node. A graph is said to be acyclic if it has no cycles. In the example graph in Figure 7.1(b), the sequence $\{D, B, C, A\}$ constitutes a simple path between D and A whereas the sequence $\{B, E, D, B\}$ forms a cycle.

If there is a path between each pair of the nodes, the graph is said to be *connected* as in Figures 7.1(a) and 7.1(c), otherwise, it is disconnected as in Figure 7.1(b). Therefore, a tree can be defined as a connected acyclic graph.

The set of nodes directly connected by an edge to a particular node are said to be *adjacent* to that node. In Figure 7.1(a), the nodes that are adjacent to 4 are $\{2, 3, 5\}$ whereas in Figure 7.1(b), the nodes that are adjacent to B are $\{A, C, E\}$. In a directed graph, the number of nodes adjacent to a given node is called the *out degree* of that node and the number of nodes to which a particular node is adjacent to is called the *in degree*. For example, node C in the graph in Figure 7.1(b) has an in degree of 2 and an out degree of 1. In an undirected graph, the in degree is equal to the out degree and often, it is referred to simply as the degree of the node.

Sometimes, it is useful to associate a *cost* (or *weight*) with every edge in the graph. A weighted graph is illustrated in Figure 7.1(c).

7.2 The graph ADT

We now discuss how to handle graphs in a functional language. Since there are many possible representations, a graph will be manipulated as an ADT (cf. Chapter 5). The description presented here is for a weighted graph but it could equally apply to an unweighted graph by ignoring weight information.

First, we need a suitable definition of a type `(Graph n w)` where:

- `n` is the type of the nodes: we assume it can be an index (class `Ix`). We are only making this assumption in order to use a node as an array index.
- `w` is the type of the weight: we assume that weights are numbers (class `Num`).

Given this type, the operations needed to manipulate the graph ADT are:

`mkGraph :: (Ix n,Num w) => Bool->(n,n)->[(n,n,w)]->(Graph n w)`
takes the lower and upper bound of the vertices set, a list of edges (each edge consists of the two end vertices and the weight) and returns a graph; the first boolean parameter indicates whether the graph is directed; if `False` then the given arcs are added in both directions.

`adjacent :: (Ix n,Num w) => (Graph n w) -> n -> [n]`
returns the nodes adjacent to a particular node.

`nodes :: (Ix n,Num w) => (Graph n w) -> [n]`
returns all the nodes in the graph.

`edgesD,edgesU :: (Ix n,Num w) => (Graph n w) -> [(n,n,w)]`
returns all the edges in a directed and an undirected graph respectively.

`edgeIn :: (Ix n,Num w) => (Graph n w) -> (n,n) -> Bool`
returns `True` only if the corresponding edge is in the graph.

`weight :: (Ix n,Num w) => n -> n -> (Graph n w) -> w`
returns the weight of the edge connecting two nodes.

The graph of Figure 7.1(c) can then be created with the following expression:

```
mkGraph False (1,5) [(1,2,12),(1,3,34),(1,5,78),(2,4,55),
                     (2,5,32),(3,4,61),(3,5,44),(4,5,93)]
```

The rest of this section describes possible implementations of the graph ADT.

7.2.1 Pointer representation

One way of representing a graph is to declare it just like a general tree using the following declaration:

```
data Graph n w = Vertex n [((Graph n w),w)]
```

In this representation, circular references are possible using local variables to reference nodes. For example, the graph of Figure 7.1(c) is represented as:

```
graphPTR = v1
    where
      v1 = Vertex 1 [(v2,12),(v3,34),(v5,78)]
      v2 = Vertex 2 [(v1,12),(v4,55),(v5,32)]
      v3 = Vertex 3 [(v1,34),(v4,61),(v5,44)]
      v4 = Vertex 4 [(v2,55),(v3,61),(v5,93)]
      v5 = Vertex 5 [(v1,78),(v2,32),(v3,44),(v4,93)]
```

This representation is very space efficient because traversing the graph does not involve the creation of intermediate data structures. One problem is that if the graph is not