```
nodes g        = range (l,u) where ((l,_),(u,_)) = bounds g

edgeIn g (x,y)= (g!(x,y)) /= Nothing

weight x y g   = w where (Just w) = g!(x,y)

edgesD g       = [(v1,v2,unwrap(g!(v1,v2)))
                      | v1 <-nodes g, v2 <- nodes g,
                        edgeIn g (v1,v2)]
    where unwrap (Just w) = w

edgesU g       = [(v1,v2,unwrap(g!(v1,v2)))
                    | v1 <-nodes g, v2 <- range (v1,u),
                      edgeIn g (v1,v2)]
    where (_,(u,_)) = bounds g
          unwrap (Just w) = w
```

`mkGraph` first creates an array filled with `Nothing` values which are then replaced by `Just v` according to the given list of arcs. Note that the `edgesU` function works by scanning half of the adjacency matrix only. It uses the function `range` which produces the list of indices that exist between a pair of bounds. The main inefficiency of this representation is having to build the matrix with `Nothing` constructors then updating it with weight values. We could avoid this problem by assuming that the list of edges supplied to the `mkGraph` function contains all possible combinations and that a special value represents a non-existent edge.

As with the previous implementation, using an adjacency matrix is only suitable if there are no (or few) changes to the structure of the graph. The main advantage over it is that access to the weight of an edge is constant. However, computing the adjacency list of any node now takes $O(|V|)$ steps.

7.2.4 Comparison between the representations

The pointer representation is very space efficient but all its operations are complicated and time inefficient. When comparing between the adjacency list and the adjacency matrix representations, the efficiencies of their graph-manipulating functions depend on the parameters $|V|$ and $|E|$ of the graph, or on the *degree of sparsity*. Informally, when there is a large number of edges, the graph is said to be *dense*, and when there are few connections, it is *sparse*. In some textbooks, a graph is said to be sparse when $|E| < |V| \log |V|$. In general, the matrix representation is better with dense graphs and the adjacency list representation with sparse graphs.

7.3 Depth-first and breadth-first search

We can identify two main strategies for traversing a graph:

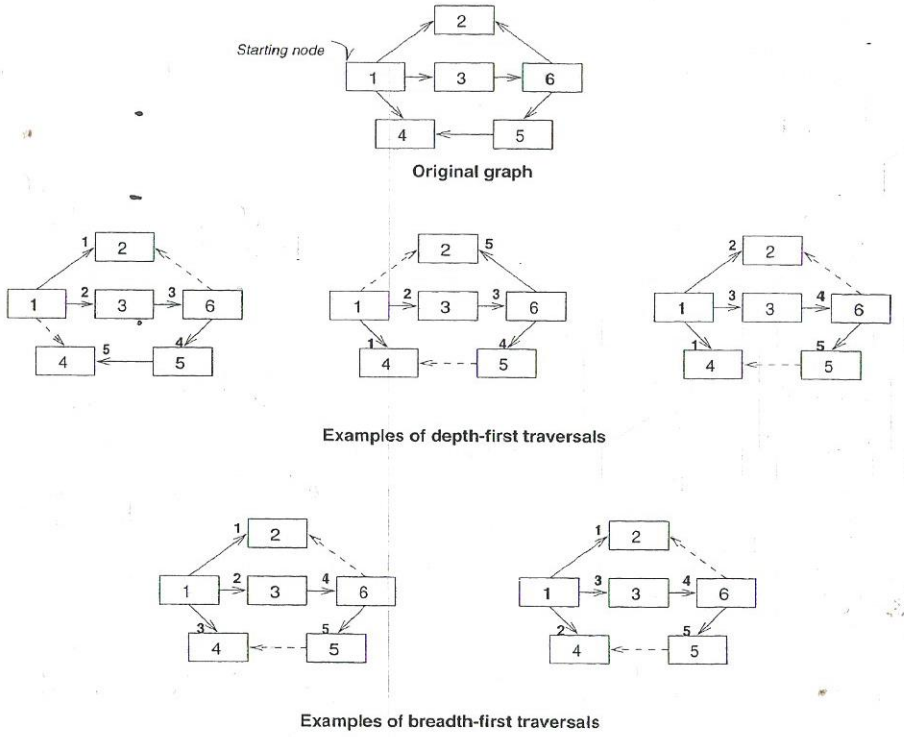

Figure 7.2 Depth-first and breadth-first traversals of a graph.

- **Depth-first search** after visiting a node, all its (non-visited) adjacent nodes are recursively visited using a depth-first search. This is a generalization of the preorder traversal of the tree.
- **Breadth-first search** after visiting a node, all its (non-visited) adjacent nodes are visited. Then, the algorithm is applied to each of these adjacent nodes separately.

For example, Figure 7.2 shows how a graph can be traversed in different ways using a depth-first or a breadth-first strategy. This example shows a directed graph but the strategy is the same for undirected graphs if we consider an edge to be equivalent to two arcs, one in each direction.

These graph traversal strategies constitute a framework which can be used to design other graph algorithms. The *topological sort* algorithm, considered later in this chapter, is an example of graph traversal that uses a depth-first strategy.

The remaining part of this section examines different ways of implementing these graph traversals. In Section 2.6.5, we have described several strategies for traversing the tree data structure. A graph traversal is similar to a tree traversal but in the presence of cycles we need to remember which nodes have been visited so that they will not be visited again.