connected, each component has to be defined separately. Another problem is that since pointers that denote circularly defined values cannot be manipulated directly or compared, constructing or traversing the graph cannot be achieved in a simple manner. Because of these drawbacks, we will not elaborate further on this representation but in some cases this representation might prove useful.

7.2.2 Adjacency list representation

Another representation is the *adjacency list* representation. A linear structure holds every node in the graph together with its adjacent nodes. For example, if we decide to use a list to hold the nodes, a graph is defined as:

```
type Graph n w = [(n,[(n,w)])]
```

In this representation, access to the adjacency list of a node would be in $O(|V|)$. A better representation uses an array to allow a constant access to any adjacency list:

```
type Graph n w = Array n [(n,w)]
```

In this case, the graph of Figure 7.1(c) can be directly defined as:

```
graphAL = array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                       (2,[(1,12),(4,55),(5,32)]),
                       (3,[(1,34),(4,61),(5,44)]),
                       (4,[(2,55),(3,61),(5,93)]),
                       (5,[(1,78),(2,32),(3,44),(4,93)])]
```

or created with the following call:[1]

```
graphAL' = 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)]
```

Unlike with a list, modifications to the shape of the array (needed when adding or removing vertices) cannot be easily achieved. However, this is sufficient for our purpose since the algorithms described later consider a graph as a static entity that is not modified.

We now show an adjacency list implementation of the graph ADT. We can see that for undirected graphs, listing the edges without duplicates is achieved by ordering vertices in an edge in increasing value order.

```
mkGraph dir bnds es =
    accumArray (\xs x -> x:xs) [] bnds
               ([(x1,(x2,w)) | (x1,x2,w) <- es] ++
                if dir then []
                else [(x2,(x1,w))|(x1,x2,w)<-es,x1/=x2])
```

[1] graphAL and graphAL' are equivalent graphs but the ordering of the arcs might be different depending on the implementation of mkGraph.

```
adjacent g v   = map fst (g!v)

nodes g        = indices g

edgeIn g (x,y)= elem y (adjacent g x)

weight x y g   = head [c | (a,c)<-g!x , (a==y)]

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

edgesU g       = [(v1,v2,w) | v1<- nodes g ,(v2,w) <-g!v1 ,v1<v2]
```

MkGraph starts from an empty array and then creates for each vertex the list of destination nodes and weights by adding a tuple in front of its list. If the graph is undirected then the reverse arcs are added, except for arcs that would start and end at the same vertex. The main characteristic of this representation is that traversing adjacent nodes as well as computing the weight of an edge takes a time proportional to the length of the adjacency list. In the worst case, the length is equal to $|V| - 1$ when the node is connected to all other nodes.

7.2.3 Adjacency matrix representation

The next representation uses a two-dimensional square array of values of dimension $|V| \times |V|$ where both coordinates i and j are nodes and where the entry (i, j) is equal to the weight of the corresponding edge (or arc) between nodes i and j:

```
type Graph a b = Array (a,a) b
```

The main problem with using this representation is that we need a value that corresponds to non-existent edges. For this, we use the predefined Maybe algebraic type with two constructors, Nothing and Just b. It is then possible to detect whether there is an edge between two nodes by testing for the value Nothing. Given this type, the adjacency matrix implementation of the graph ADT now becomes:

```
type Graph n w = Array (n,n) (Maybe w)

mkGraph dir bnds@(l,u) es
    = emptyArray // ([((x1,x2),Just w) |(x1,x2,w)<-es] ++
                     if dir then []
                     else [((x2,x1),Just w) |(x1,x2,w)<-es,x1/=x2])
      where emptyArray
              = array ((l,l),(u,u)) [((x1,x2),Nothing) |
                                      x1 <- range bnds,
                                      x2 <- range bnds]

adjacent g v1 = [ v2 | v2 <-nodes g,(g!(v1,v2))/= Nothing]
```