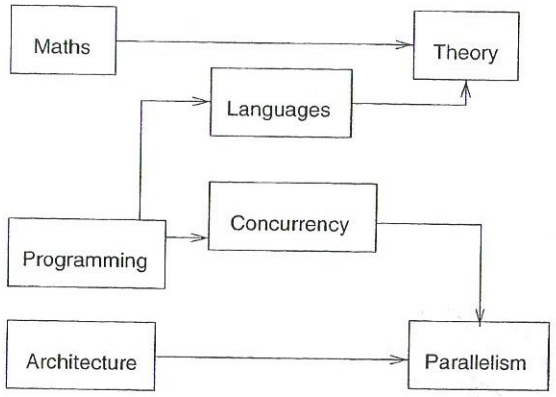

Figure 7.3 Representing a course prerequisite structure as a graph.

If the graph is not connected, the depth-first search and breadth-first search functions presented here only produce the list of all graph nodes reachable from the starting node. Section 7.6 describes depth-first search for non-connected graphs.

7.4 Topological sort

We now consider a graph problem with numerous applications. Given a directed acyclic graph, a *topological sort* is an ordering of vertices such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering. As the example in Figure 7.3 shows, we may want to represent the modules in a course as a graph, in which an arc from module i to module j indicates that i is a prerequisite module for the module j.

A topological sort of a graph corresponds to an appropriate sequence in the choice of modules that conforms to the prerequisite structure. An example is the sequence *Maths, Programming, Languages, Concurrency, Architecture, Parallelism, Theory*. Note that the sequence is not unique: the modules *Concurrency* and *Architecture* could have been swapped in the sequence.

A simple program for the topological sort will be developed based on the depth-first traversal function `depthFirstSearch` function presented in Section 7.3.1. First, a closer look at the body of the `dfs` function reveals how these argument lists are modified from one recursive call to the next:

```
Candidate nodes:  (c:cs)   →  (adjacent g c)++cs
   Visited nodes:  vis     →  vis++[c]
```

In the topological sort program, these lists must be updated as follows:

```
Candidate nodes:  (c:cs)   →  cs
   Visited nodes:  vis     →  c:(tsort' (adjacent g c) vis)
```


This means that when a node is inserted in the solution, it precedes the topological sort (called recursively) of all its adjacent nodes. Also, the initial list of candidate nodes must consist of all nodes with no incoming edges (that is, nodes with an in degree equal to zero). The complete program is:

```
inDegree g n  = length [t | v<-nodes g, t<-adjacent g v, (n==t)]

topologicalSort g = tsort [n | n<-nodes g , (inDegree g n == 0)]
                          []
  where
    tsort [] r    = r
    tsort (c:cs) vis
     | elem c vis = tsort cs vis
     | otherwise  = tsort cs (c:(tsort (adjacent g c) vis))
```

When applied to the example in Figure 7.2 created by the following expression:

```
g = mkGraph True (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
                        (5,4,0),(6,2,0), (6,5,0)]
```

it generates the following reduction sequence:

```
topologicalSort g
  ⇒ tsort' [1] []
  ⇒ tsort' [] (1:(tsort' [2,3,4] []))
  ⇒ (1:(tsort' [2,3,4] []))
  ⇒ 1:(tsort' [2,3,4] [])
  ⇒ 1:(tsort' [3,4] (2:(tsort' [] [])))
  ⇒ 1:(tsort' [3,4] [2])
  ⇒ 1:(tsort' [4] (3:(tsort' [6] [2])))
  ⇒ 1:(tsort' [4] (3:(tsort' [] (6:(tsort' [2,5] [2])))))
  ⇒ 1:(tsort' [4] (3:(6:(tsort' [2,5] [2]))))
  ⇒ 1:(tsort' [4] (3:(6:(tsort' [5] [2]))))
  ⇒ 1:(tsort' [4] (3:(6:(tsort' [] (5:(tsort' [4] [2]))))))
  ⇒ 1:(tsort' [4] (3:(6:(5:(tsort' [4] [2])))))
  ⇒ 1:(tsort' [4] (3:(6:(5:(tsort' []
                                   (4:(tsort' [] [2])))))))
  ⇒ 1:(tsort' [4] (3:(6:(5:(4:(tsort' [] [2]))))))
  ⇒ 1:(tsort' [4] [3,6,5,4,2])
  ⇒ 1:(tsort' [] [3,6,5,4,2])
  ⇒ [1,3,6,5,4,2]
```

A topological sort sequence corresponds to a depth-first search but not every depth-first search is a topological sort. For example, the depth-first sequence `[1,2,3,6,5,4]` is not a topological sort because the value 2 appears before 6.

The course graph of Figure 7.3 can be created by the following definitions: