# 6

# Exploring Graphs

## 6.1 INTRODUCTION

A great many problems can be formulated in terms of graphs. We have seen, for instance, the shortest route problem and the problem of the minimal spanning tree. To solve such problems, we often need to look at all the nodes, or all the edges, of a graph. Sometimes, the structure of the problem is such that we need only visit some of the nodes or edges. Up to now, the algorithms we have seen have implicitly imposed an order on these visits: it was a case of visiting the nearest node, the shortest edge, and so on. In this chapter we introduce some general techniques that can be used when no particular order of visits is required.

We shall use the word "graph" in two different ways. A graph may be a data structure in the memory of a computer. In this case, the nodes are represented by a certain number of bytes, and the edges are represented by pointers. The operations to be carried out are quite concrete: to "mark a node" means to change a bit in memory, to "find a neighbouring node" means to follow a pointer, and so on.

At other times, the graph exists only implicitly. For instance, we often use abstract graphs to represent games: each node corresponds to a particular position of the pieces on the board, and the fact that an edge exists between two nodes means that it is possible to get from the first to the second of these positions by making a single legal move. When we explore such a graph, it does not really exist in the memory of the machine. Most of the time, all we have is a representation of the current position (that is, of the node we are in the process of visiting) and possibly representations of a few other positions. In this case to "mark a node" means to take any appropriate measures that enable us to recognize a position we have already seen, or to avoid arriving at the same position twice; to "find a neighbouring node" means to change the current position by making a single legal move; and so on.

However, whether the graph is a data structure or merely an abstraction, the techniques used to traverse it are essentially the same. In this chapter we therefore do not distinguish the two cases.

## 6.2 TRAVERSING TREES

We shall not spend long on detailed descriptions of how to explore a tree. We simply remind the reader that in the case of binary trees three techniques are often used. If at each node of the tree we visit first the node itself, then all the nodes in the left-hand subtree, and finally, all the nodes in the right-hand subtree, we are traversing the tree in *preorder*; if we visit first the left-hand subtree, then the node itself, and finally, the right-hand subtree, we are traversing the tree in *inorder*; and if we visit first the left-hand subtree, then the right-hand subtree, and lastly, the node itself, then we are visiting the tree in *postorder*. Preorder and postorder generalize in the obvious way to nonbinary trees.

These three techniques explore the tree from left to right. Three corresponding techniques explore the tree from right to left. It is obvious how to implement any of these techniques using recursion.

**Lemma 6.2.1.** For each of the six techniques mentioned, the time $T(n)$ needed to explore a binary tree containing $n$ nodes is in $\Theta(n)$.

*Proof.* Suppose that visiting a node takes a time in $\Theta(1)$, that is, the time required is bounded above by some constant $c$. Without loss of generality, we may suppose that $c \geq T(0)$.

Suppose further that we are to explore a tree containing $n$ nodes, $n > 0$, of which one node is the root, $g$ nodes are situated in the left-hand subtree, and $n-g-1$ nodes are in the right-hand subtree. Then

$$T(n) \leq \max_{0 \leq g \leq n-1} (T(g) + T(n-g-1) + c) \quad n > 0 .$$

We prove by mathematical induction that $T(n) \leq dn + c$ where $d$ is a constant such that $d \geq 2c$. By the choice of $c$ the hypothesis is true for $n = 0$. Now suppose that it is true for all $n$, $0 \leq n < m$, for some $m > 0$. Then

$$\begin{aligned} T(m) &\leq \max_{0 \leq g \leq m-1} (T(g) + T(m-g-1) + c) \\ &\leq \max_{0 \leq g \leq m-1} (dg + c + d(m-g-1) + c + c) \\ &\leq dm + 3c - d \leq dm + c \end{aligned}$$

so the hypothesis is also true for $n = m$. This proves that $T(n) \leq dn + c$ for every $n \geq 0$, and hence $T(n)$ is in $O(n)$.

On the other hand, it is clear that $T(n)$ is in $\Omega(n)$ since each of the $n$ nodes is visited. Therefore $T(n)$ is in $\Theta(n)$. □

**Problem 6.2.1.** Prove that for any of the techniques mentioned, a recursive implementation takes memory space in $\Omega(n)$ in the worst case. □

*** Problem 6.2.2.** Show how the preceding exploration techniques can be implemented so as to take only a time in $\Theta(n)$ and space in $\Theta(1)$, even when the nodes do not contain a pointer to their parents (otherwise the problem becomes trivial). □

**Problem 6.2.3.** Show how to generalize the concepts of preorder and postorder to arbitrary (nonbinary) trees. Assume the trees are represented as in Figure 1.9.5. Prove that both these techniques still run in a time in the order of the number of nodes in the tree to be traversed. □

## 6.3 DEPTH-FIRST SEARCH : UNDIRECTED GRAPHS

Let $G = <N, A>$ be an undirected graph all of whose nodes we wish to visit. Suppose that it is somehow possible to mark a node to indicate that it has already been visited. Initially, no nodes are marked.

To carry out a *depth-first* traversal of the graph, choose any node $v \in N$ as the starting point. Mark this node to show that it has been visited. Next, if there is a node adjacent to $v$ that has not yet been visited, choose this node as a new starting point and call the depth-first search procedure recursively. On return from the recursive call, if there is another node adjacent to $v$ that has not been visited, choose this node as the next starting point, call the procedure recursively once again, and so on. When all the nodes adjacent to $v$ have been marked, the search starting at $v$ is finished.

If there remain any nodes of $G$ that have not been visited, choose any one of them as a new starting point, and call the procedure yet again. Continue in this way until all the nodes of $G$ have been marked. Here is the recursive algorithm.

```
procedure search(G)
  for each $v \in N$ do $mark[v] \leftarrow$ not-visited
  for each $v \in N$ do
    if $mark[v] \neq$ visited then dfs(v)

procedure dfs(v : node)
  { node v has not been visited }
  $mark[v] \leftarrow$ visited
  for each node w adjacent to v do
    if $mark[w] \neq$ visited then dfs(w)
```

The algorithm is called depth-first search since it tries to initiate as many recursive calls as possible before it ever returns from a call. The recursivity is only stopped when exploration of the graph is blocked and can go no further. At this point the recursion "unwinds" so that alternative possibilities at higher levels can be explored.

**Example 6.3.1.** If we suppose that the neighbours of a given node are examined in numerical order, and that node 1 is the first starting point, a depth-first search of the graph in Figure 6.3.1 progresses as follows:

| | | |
|---|---|---|
| 1. | *dfs* (1) | initial call |
| 2. | *dfs* (2) | recursive call |
| 3. | *dfs* (3) | recursive call |
| 4. | *dfs* (6) | recursive call |
| 5. | *dfs* (5) | recursive call ; progress is blocked |
| 6. | *dfs* (4) | a neighbour of node 1 has not been visited |
| 7. | *dfs* (7) | recursive call |
| 8. | *dfs* (8) | recursive call ; progress is blocked |
| 9. | there are no more nodes to visit. | |

□

**Problem 6.3.1.** Show how a depth-first search progresses through the graph in Figure 6.3.1 if the neighbours of a given node are examined in numerical order but the initial starting point is node 6. □

How much time is needed to explore a graph with $n$ nodes and $a$ edges? Since each node is visited exactly once, there are $n$ calls of the procedure *dfs*. When we visit a node, we look at the mark on each of its neighbouring nodes. If the graph is represented in such a way as to make the lists of adjacent nodes directly accessible (type *lisgraph* of Section 1.9.2), this work is proportional to $a$ in total. The algorithm therefore takes a time in $O(n)$ for the procedure calls and a time in $O(a)$ to inspect the marks. The execution time is thus in $O(\max(a, n))$.

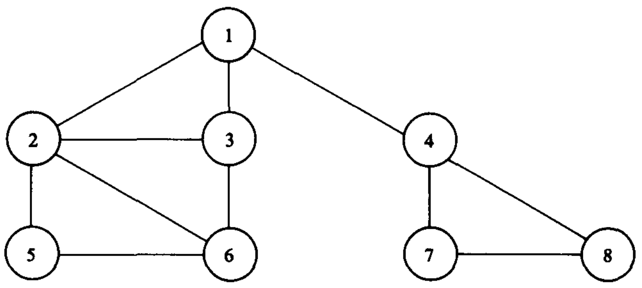

**Figure 6.3.1.** An undirected graph.

**Problem 6.3.2.** What happens if the graph is represented by an adjacency matrix (type *adjgraph* of Section 1.9.2) rather than by lists of adjacent nodes? □

**Problem 6.3.3.** Show how depth-first search can be used to find the connected components of an undirected graph. □

A depth-first traversal of a connected graph associates a spanning tree to the graph. The edges of the tree correspond to the edges used to traverse the graph; they are directed from the first node visited to the second. Edges that are not used in the traversal of the graph have no corresponding edge in the tree. The initial starting point of the exploration becomes the root of the tree.

**Example 6.3.2.** (Continuation of Example 6.3.1) The edges used in the depth-first search of Example 6.3.1 are $\{1,2\}$, $\{2,3\}$, $\{3,6\}$, $\{6,5\}$, $\{1,4\}$, $\{4,7\}$ and $\{7,8\}$. The corresponding directed edges $(1,2)$, $(2,3)$, and so on, form a spanning tree for the graph in Figure 6.3.1. The root of the tree is node 1. See Figure 6.3.2. □

If the graph being explored is not connected, a depth-first search associates to it not merely a single tree, but rather a forest of trees, one for each connected component of the graph.

A depth-first search also provides a way to number the nodes of the graph being visited: the first node visited (the root of the tree) is numbered 1, the second is numbered 2, and so on. In other words, the nodes of the associated tree are numbered in preorder. To implement this numbering, we need only add the following two statements at the beginning of the procedure *dfs* :

```
pnum ← pnum + 1
prenum[v] ← pnum
```

where *pnum* is a global variable initialized to zero.

**Example 6.3.3.** (Continuation of Example 6.3.1) The depth-first search illustrated by Example 6.3.1 numbers the nodes as follows:

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| *prenum* | 1 | 2 | 3 | 6 | 5 | 4 | 7 | 8 |

. □

Of course, the tree and the numbering generated by a depth-first search in a graph are not unique, but depend on the chosen starting point and on the order in which neighbours are visited.

**Problem 6.3.4.** Exhibit the tree and the numbering generated by the search of Problem 6.3.1. □

### 6.3.1 Articulation Points

A node $v$ of a connected graph is an *articulation point* if the subgraph obtained by deleting $v$ and all the edges incident on $v$ is no longer connected. For example, node 1 is an articulation point of the graph in Figure 6.3.1; if we delete it, there remain two connected components $\{2,3,5,6\}$ and $\{4,7,8\}$. A graph $G$ is *biconnected* (or *unarticulated*) if it is connected and has no articulation points. It is *bicoherent* (or *isthmus-free*, or *2-edge-connected*) if each articulation point is joined by at least two edges to each component of the remaining sub-graph. These ideas are important in practice: if the graph $G$ represents, say, a telecommunications network, then the fact that it is biconnected assures us that the rest of the network can continue to function even if the equipment in one of the nodes fails; if $G$ is bicoherent, we can be sure that all the nodes will be able to communicate with one another even if one transmission line stops working.

The following algorithm finds the articulation points of a connected graph $G$.

**a.** Carry out a depth-first search in $G$, starting from any node. Let $T$ be the tree generated by the depth-first search, and for each node $v$ of the graph, let *prenum* $[v]$ be the number assigned by the search.

**b.** Traverse the tree $T$ in postorder. For each node $v$ visited, calculate *lowest* $[v]$ as the minimum of

  **i.** *prenum* $[v]$

  **ii.** *prenum* $[w]$ for each node $w$ such that there exists an edge $\{v,w\}$ in $G$ that has no corresponding edge in $T$

  **iii.** *lowest* $[x]$ for every child $x$ of $v$ in $T$.

**c.** Articulation points are now determined as follows:

  **i.** The root of $T$ is an articulation point of $G$ if and only if it has more than one child.

  **ii.** A node $v$ other than the root of $T$ is an articulation point of $G$ if and only if $v$ has a child $x$ such that *lowest* $[x] \geq$ *prenum* $[v]$.

**Example 6.3.4.** (Continuation of Examples 6.3.1, 6.3.2, and 6.3.3) The search described in Example 6.3.1 generates the tree illustrated in Figure 6.3.2. The edges of $G$ that have no corresponding edge in $T$ are represented by broken lines. The value of *prenum* $[v]$ appears to the left of each node $v$, and the value of *lowest* $[v]$ to the right. The values of *lowest* are calculated in postorder, that is, for nodes 5, 6, 3, 2, 8, 7, 4, and 1 successively. The articulation points of $G$ are nodes 1 (by rule c(i)) and 4 (by rule c(ii)). □

**Problem 6.3.5.** Verify that the same articulation points are found if we start the search at node 6. □

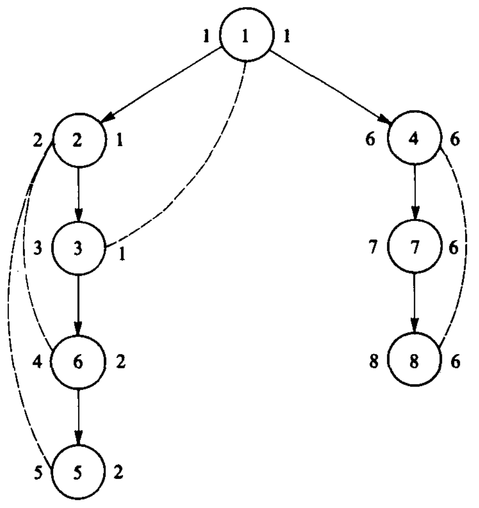

Figure 6.3.2. A depth-first search tree; *prenum* on the left and *lowest* on the right.

**Problem 6.3.6.** Prove that an edge of $G$ that has no corresponding edge in $T$ (a broken line in Figure 6.3.2) necessarily joins some node $v$ to one of its ancestors in $T$. □

Informally, we can define *lowest*$[v]$ by

*lowest*$[v]$ = min{ *prenum*$[w]$ | you can get to $w$ from $v$ by following down as many solid lines as you like and then going up at most one broken line } .

If $x$ is a child of $v$ and if *lowest*$[x]$ < *prenum*$[v]$, there thus exists a chain of edges that joins $x$ to the other nodes of the graph even if $v$ is deleted. On the other hand, there is no chain joining $x$ to the parent of $v$ if $v$ is not the root and if *lowest*$[x] \geq$ *prenum*$[v]$.

**Problem 6.3.7.** Complete the proof that the algorithm is correct. □

**Problem 6.3.8.** Show how to carry out the operations of steps (a) and (b) in parallel and write the corresponding algorithm. □

* **Problem 6.3.9.** Write an algorithm that decides whether or not a given connected graph is bicoherent. □

* **Problem 6.3.10.** Write an efficient algorithm that, given an undirected graph that is connected but not biconnected, finds a set of edges that could be added to make the graph biconnected. Your algorithm should find the smallest possible set of edges. Analyse the efficiency of your algorithm. □

**Problem 6.3.11.** Prove or give a counterexample:

**i.** If a graph is biconnected, then it is bicoherent.
**ii.** If a graph is bicoherent, then it is biconnected. □

**Problem 6.3.12.** Prove that a node $v$ in a connected graph is an articulation point if and only if there exist two nodes $a$ and $b$ different from $v$ such that every path joining $a$ and $b$ passes through $v$. □

**Problem 6.3.13.** Prove that for every pair of distinct nodes $v$ and $w$ in a biconnected graph, there exist at least two chains of edges joining $v$ and $w$ that have no nodes in common (except the starting and ending nodes). □

## 6.4 DEPTH-FIRST SEARCH : DIRECTED GRAPHS

The algorithm is essentially the same as the one for undirected graphs, the difference being in the interpretation of the word "adjacent". In a directed graph, node $w$ is adjacent to node $v$ if the directed edge $(v,w)$ exists. If $(v,w)$ exists and $(w,v)$ does not, then $w$ is adjacent to $v$ but $v$ is not adjacent to $w$. With this change of interpretation the procedures *dfs* and *search* from Section 6.3 apply equally well in the case of a directed graph.

The algorithm behaves quite differently, however. Consider a depth-first search of the directed graph in Figure 6.4.1. If the neighbours of a given node are examined in numerical order, the algorithm progresses as follows:

```
1.  dfs(1)          initial call
2.    dfs(2)        recursive call
3.      dfs(3)      recursive call; progress is blocked
4.    dfs(4)        a neighbour of node 1 has not been visited
5.      dfs(8)      recursive call
6.        dfs(7)    recursive call; progress is blocked
7.  dfs(5)          new starting point
8.    dfs(6)        recursive call; progress is blocked
9.  there are no more nodes to visit.
```

**Problem 6.4.1.** Illustrate the progress of the algorithm if the neighbours of a given node are examined in decreasing numerical order, and if the starting point is node 1. □

An argument identical with the one in Section 6.3 shows that the time taken by this algorithm is also in $O(\max(a,n))$. In this case, however, the edges used to visit all the nodes of a directed graph $G = <N, A>$ may form a forest of several trees even

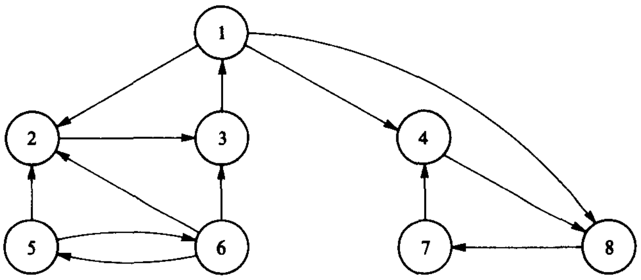

**Figure 6.4.1.** A directed graph.

if $G$ is connected. This happens in our example : the edges used, namely (1, 2), (2, 3), (1, 4), (4, 8), (8, 7), and (5, 6), form the forest shown by the solid lines in Figure 6.4.2. (The numbers to the left of each node are explained in Section 6.4.2.)

Let $F$ be the set of edges in the forest. In the case of an undirected graph the edges of the graph with no corresponding edge in the forest necessarily join some node to one of its ancestors (Problem 6.3.6). In the case of a directed graph three kinds of edges can appear in $A \setminus F$ (these edges are shown by the broken lines in Figure 6.4.2).

**i.** Those like (3, 1) or (7, 4) that lead from a node to one of its ancestors ;
**ii.** those like (1, 8) that lead from a node to one of its descendants ; and
**iii.** those like (5, 2) or (6, 3) that join one node to another that is neither its ancestor nor its descendant. Edges of this type are necessarily directed from right to left.

**Problem 6.4.2.** Prove that if $(v, w)$ is an edge of the graph that has no corresponding edge in the forest, and if $v$ is neither an ancestor nor a descendant of $w$ in the forest, then *prenum* $[v]$ > *prenum* $[w]$, where the values of *prenum* are attributed as in Section 6.3. □

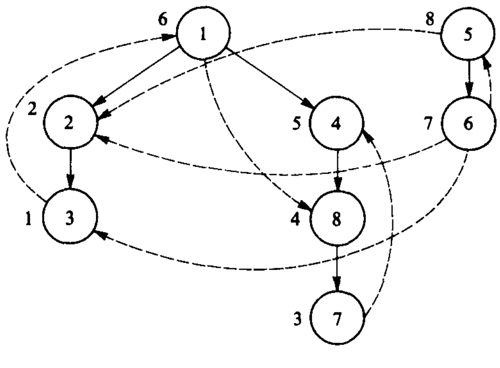

**Figure 6.4.2.** A depth-first search forest.

## 6.4.1 Acyclic Graphs: Topological Sorting

Directed acyclic graphs can be used to represent a number of interesting relations. This class of structures includes trees, but is less general than the class of all directed graphs. For example, a directed acyclic graph can be used to represent the structure of an arithmetic expression that includes repeated subexpressions: thus Figure 6.4.3 represents the structure of the expression

$$(a+b)(c+d) + (a+b)(c-d) .$$

Such graphs also offer a natural representation for partial orderings (such as the relation "smaller than" defined on the integers and the set-inclusion relation). Figure 6.4.4 illustrates part of another partial ordering defined on the integers. (What is the partial ordering in question?) Finally, directed acyclic graphs are often used to represent the different stages of a complex project: the nodes are different states of the project, from the initial state to final completion, and the edges correspond to activities that have to be completed to pass from one state to another. Figure 6.4.5 gives an example of this type of diagram.

Depth-first search can be used to detect whether a given directed graph is acyclic.

**Problem 6.4.3.** Let $F$ be the forest generated by a depth-first search on a directed graph $G = \langle N, A \rangle$. Prove that $G$ is acyclic if and only if $A \setminus F$ includes no edge of type (i) (that is, from a node of $G$ to one of its ancestors in the forest). □

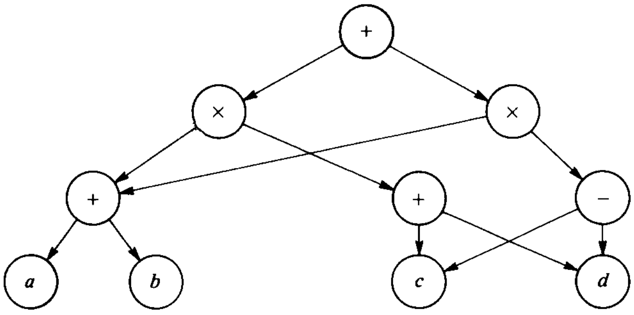

**Figure 6.4.3** A directed acyclic graph.

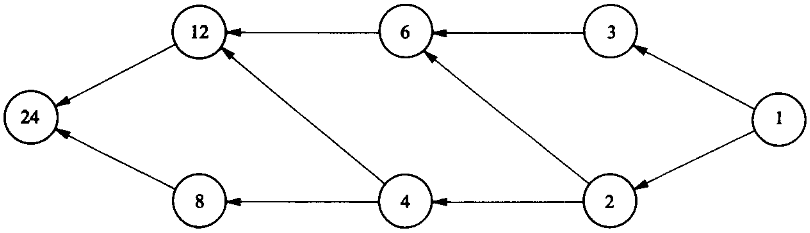

**Figure 6.4.4.** Another directed acyclic graph.

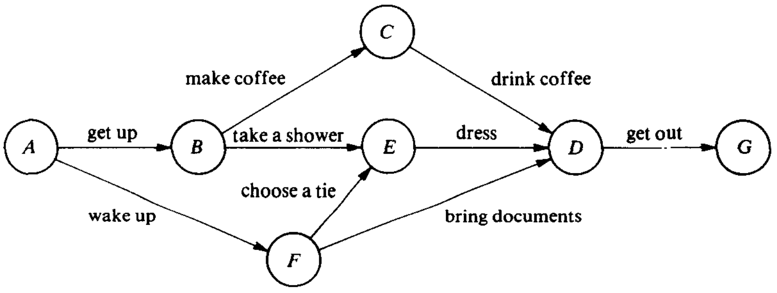

**Figure 6.4.5.** Yet another directed acyclic graph.

A *topological sort* of the nodes of a directed acyclic graph is the operation of arranging the nodes in order in such a way that if there exists an edge $(i, j)$, then $i$ precedes $j$ in the list. For example, for the graph of Figure 6.4.4, the natural order 1, 2, 3, 4, 6, 8, 12, 24 is adequate; but the order 1, 3, 2, 6, 4, 12, 8, 24 is also acceptable, as are several others. For graphs as in Figure 6.4.5, a topological ordering of the states can be used to obtain a feasible schedule for the activities involved in the project; in our example, the order $A$, $B$, $F$, $C$, $E$, $D$, $G$ will serve.

The necessary modification to the procedure *dfs* to make it into a topological sort is immediate : if we add a supplementary line

```
write v
```

at the end of the procedure, the numbers of the nodes will be printed in *reverse* topological order.

**Problem 6.4.4.** Prove this. □

**Problem 6.4.5.** For the graph of Figure 6.4.4, what is the topological order obtained if the neighbours of a node are visited in numerical order and if the depth-first search begins at node 1 ? □

### 6.4.2 Strongly Connected Components

A directed graph is *strongly connected* if there exists a path from $u$ to $v$ and also a path from $v$ to $u$ for every distinct pair of nodes $u$ and $v$. If a directed graph is not strongly connected, we are interested in the largest sets of nodes such that the corresponding subgraphs are strongly connected. Each of these subgraphs is called a *strongly connected component* of the original graph. In the graph of Figure 6.4.1, for instance, nodes $\{1,2,3\}$ and the corresponding edges form a strongly connected component. Another component corresponds to the nodes $\{4,7,8\}$. Despite the fact that there exist edges $(1,4)$ and $(1,8)$, it is not possible to merge these two strongly connected components into a single component because there exists no path from node 4 to node 1.

To detect the strongly connected components of a directed graph, we must first modify the procedure *dfs*. In Section 6.3 we number each node at the instant when exploration of the node begins. Here, we number each node at the moment when exploration of the node has been completed. In other words, the nodes of the tree produced are numbered in postorder. To do this, we need only add at the end of procedure *dfs* the following two statements:

```
$nump \leftarrow nump + 1$
$postnum[v] \leftarrow nump$ ,
```

where *nump* is a global variable initialized to zero. Figure 6.4.2 shows to the left of each node the number thus assigned.

The following algorithm finds the strongly connected components of a directed graph $G$.

**i.** Carry out a depth-first search of the graph starting from an arbitrary node. For each node $v$ of the graph let $postnum[v]$ be the number assigned during the search.

**ii.** Construct a new graph $G'$ : $G'$ is the same as $G$ except that the direction of every edge is reversed.

**iii.** Carry out a depth-first search in $G'$. Begin this search at the node $w$ that has the highest value of *postnum*. (If $G$ contains $n$ nodes, it follows that $postnum[w] = n$.) If the search starting at $w$ does not reach all the nodes, choose as the second starting point the node that has the highest value of *postnum* among all the unvisited nodes; and so on.

**iv.** To each tree in the resulting forest there corresponds one strongly connected component of $G$.

**Example 6.4.1.** On the graph of Figure 6.4.1, the first depth-first search assigns the values of *postnum* shown to the left of each node in Figure 6.4.2. The graph $G'$ is illustrated in Figure 6.4.6, with the values of *postnum* shown to the left of each node. We carry out a depth-first search starting from node 5, since $postnum[5] = 8$; the search reaches nodes 5 and 6. For our second starting point, we choose node 1, with $postnum[1] = 6$; this time the search reaches nodes 1, 3, and 2. For the third starting point we take node 4, with $postnum[4] = 5$; this time the remaining nodes 4, 7, and 8 are all reached. The corresponding forest is illustrated in Figure 6.4.7. The strongly connected components of the original graph (Fig. 6.4.1) are the subgraphs corresponding to the sets of nodes $\{5,6\}$, $\{1,3,2\}$ and $\{4,7,8\}$. □

***Problem 6.4.6.** Prove that if two nodes $u$ and $v$ are in the same strongly connected component of $G$, then they are in the same tree when we carry out the depth-first search of $G'$. □

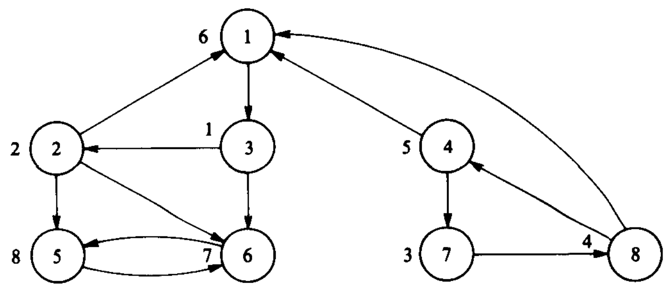

**Figure 6.4.6.** Reversing the arrows in the graph of Figure 6.4.1.

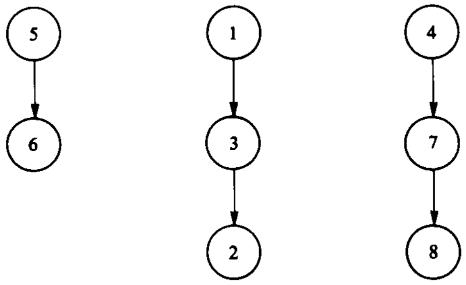

**Figure 6.4.7.** The forest of strongly connected components.

It is harder to prove the result the other way. Let $v$ be a node that is in the tree whose root is $r$ when we carry out the search of $G'$, and suppose $v \neq r$. This implies that there exists a path from $r$ to $v$ in $G'$ ; thus there exists a path from $v$ to $r$ in $G$. When carrying out the search of $G'$, we always choose as a new starting point (that is, as the root of a new tree) that node not yet visited with the highest value of *postnum*. Since we chose $r$ rather than $v$ to be the root of the tree in question, we have *postnum* $[r]$ > *postnum* $[v]$.

When we carried out the search in $G$, three possibilities seem to be open a priori:

- $r$ was an ancestor of $v$ ;
- $r$ was a descendant of $v$ ; or
- $r$ was neither an ancestor nor a descendant of $v$.

The second possibility is ruled out by the fact that *postnum* $[r]$ > *postnum* $[v]$. In the third case it would be necessary for the same reason that $r$ be to the right of $v$. However, there exists at least one path from $v$ to $r$ in $G$. Since in a depth-first search the edges not used by the search never go from left to right (Problem 6.4.2), any such path must go up the tree from $v$ to a common ancestor ($x$, say) of $v$ and $r$, and then go down the tree to $r$. But this is quite impossible. We should have *postnum* $[x]$ > *postnum* $[r]$ since $x$ is an ancestor of $r$. Next, since there exists a path from $v$ to $x$ in $G$, there would exist a path from $x$ to $v$ in $G'$. Before choosing $r$ as the root of a tree in the search of $G'$, we would have already visited $x$ (otherwise $x$ rather than $r$ would be chosen as the root of the new tree) and therefore also $v$. This contradicts the hypothesis that $v$ is in the tree whose root is $r$ when we carry out the search of $G$. Only the first possibility remains: $r$ was an ancestor of $v$ when we searched $G$. This implies that there exists a path from $r$ to $v$ in $G$.

We have thus proved that if node $v$ is in the tree whose root is $r$ when we carry out the search of $G'$, then there exist in $G$ both a path from $v$ to $r$ and a path from $r$ to $v$. If two nodes $u$ and $v$ are in the same tree when we search $G'$, they are therefore both in the same strongly connected component of $G$ since there exist paths from $u$ to $v$ and from $v$ to $u$ in $G$ via node $r$.

With the result of Problem 6.4.6, this completes the proof that the algorithm works correctly.

**Problem 6.4.7.** Estimate the time and space needed by this algorithm. □

## 6.5 BREADTH-FIRST SEARCH

When a depth-first search arrives at some node $v$, it next tries to visit some neighbour of $v$, then a neighbour of this neighbour, and so on. When a breadth-first search arrives at some node $v$, on the other hand, it first visits all the neighbours of $v$, and not until this has been done does it go on to look at nodes farther away. Unlike depth-first search, breadth-first search is not naturally recursive. To underline the similarities and the differences between the two methods, we begin by giving a nonrecursive formulation of the depth-first search algorithm. Let *stack* be a data type allowing two operations, *push* and *pop*. The type is intended to represent a list of elements that are to be handled in the order "last come, first served". The function *top* denotes the element at the top of the stack. Here is the modified depth-first search algorithm.

```
procedure dfs'(v : node)
  P ← empty-stack
  mark[v] ← visited
  push v on P
  while P is not empty do
    while there exists a node w adjacent to top(P)
      such that mark[w] ≠ visited do
        mark[w] ← visited
        push w on P  { w is now top(P) }
    pop top(P)
```

For the breadth-first search algorithm, by contrast, we need a type *queue* that allows two operations *enqueue* and *dequeue*. This type represents a list of elements that are to be handled in the order "first come, first served". The function *first* denotes the element at the front of the queue. Here now is the breadth-first search algorithm.

```
procedure bfs(v : node)
  Q ← empty-queue
  mark[v] ← visited
  enqueue v into Q
  while Q is not empty do
    u ← first(Q)
    dequeue u from Q
    for each node w adjacent to u do
      if mark[w] ≠ visited then mark[w] ← visited
                                enqueue w into Q
```

In both cases we need a main program to start the search.

```
procedure search(G)
  for each v∈N do mark[v] ← not-visited
  for each v∈N do
    if mark[v] ≠ visited then {dfs' or bfs}(v)
```

**Example 6.5.1.** On the graph of Figure 6.3.1, if the neighbours of a node are visited in numerical order, and if node 1 is used as the starting point, breadth-first search proceeds as follows.

| | Node Visited | $Q$ |
|---|---|---|
| 1. | 1 | 2,3,4 |
| 2. | 2 | 3,4,5,6 |
| 3. | 3 | 4,5,6 |
| 4. | 4 | 5,6,7,8 |
| 5. | 5 | 6,7,8 |
| 6. | 6 | 7,8 |
| 7. | 7 | 8 |
| 8. | 8 | — |

□

As for depth-first search, we can associate a tree with the breadth-first search. Figure 6.5.1 shows the tree generated by the search in Example 6.5.1. The edges of the graph that have no corresponding edge in the tree are represented by broken lines.

**Problem 6.5.1.** After a breadth-first search in an undirected graph $G = <N, A>$, let $F$ be the set of edges that have a corresponding edge in the forest of trees that is generated. Show that the edges $\{u, v\} \in A \setminus F$ are such that $u$ and $v$ are in the same tree, but that neither $u$ nor $v$ is an ancestor of the other. □

It is easy to show that the time required by a breadth-first search is in the same order as that required by a depth-first search, namely $O(\max(a, n))$. If the appropriate

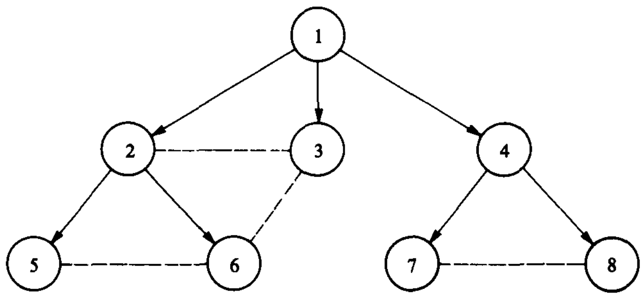

**Figure 6.5.1.** A breadth-first search tree.

interpretation of the word "neighbouring" is used, the breadth-first search algorithm can be applied without modification to either directed or undirected graphs.

**Problem 6.5.2.** Show how a breadth-first search progresses through the graph of Figure 6.4.1, assuming that the neighbours of a node are always visited in numerical order, and that necessary starting points are also chosen in numerical order. □

**Problem 6.5.3** (Continuation of Problem 6.5.2) Sketch the corresponding forest and the remaining edges of the graph. How many kinds of "broken" edges are possible? (see Section 6.4) □

Breadth-first search is most often used to carry out a partial exploration of certain infinite graphs or to find the shortest path from one point to another.

## 6.6 IMPLICIT GRAPHS AND TREES

As mentioned at the outset of this chapter, various problems can be thought of in terms of abstract graphs. For instance, we can use the nodes of a graph to represent configurations in a game of chess and edges to represent legal moves (see Section 6.6.2). Often the original problem translates to searching for a specific node, path or pattern in the associated graph. If the graph contains a large number of nodes, it may be wasteful or infeasible to build it explicitly in computer memory before applying one of the search techniques we have encountered so far.

An *implicit* graph is one for which we have available a description of its nodes and edges. Relevant portions of the graph can thus be built as the search progresses. Therefore computing time is saved whenever the search succeeds before the entire graph has been constructed. The economy in memory space is even more dramatic when nodes that have already been searched can be discarded, making room for subsequent nodes to be explored.

Backtracking is a basic search technique on implicit graphs. One powerful application is in playing games of strategy by techniques known as minimax and alpha-beta pruning. Some optimization problems can be handled by the more sophisticated branch-and-bound technique. We now discuss these notions.

### 6.6.1 Backtracking

Backtracking algorithms use a special technique to explore implicit directed graphs. These graphs are usually trees, or at least they contain no cycles. A backtracking algorithm carries out a systematic search, looking for solutions to some problem. At least one application of this technique dates back to antiquity: it allows one to find the way through a labyrinth without danger of going round and round in circles. To illustrate the general principle, we shall, however, use a different example. Consider the classic problem of placing eight queens on a chess-board in such a way that none of them threatens any of the others. (A queen threatens the squares in the same row, in the same column, or on the same diagonals.)

**Problem 6.6.1.** Solve this problem without using a computer. □

The first obvious way to solve this problem consists of trying systematically all the ways of placing eight queens on a chess-board, checking each time to see whether a solution has been obtained. This approach is of no practical use, since the number of positions we have to check would be $\binom{64}{8} = 4{,}426{,}165{,}368$. The first improvement we might try consists of never putting more than one queen in any given row. This reduces the computer representation of the chess-board to a simple vector of eight elements, each giving the position of the queen in the corresponding row. For instance, the vector $(3,1,6,2,8,6,4,7)$ does not represent a solution since the queens in the third and the sixth rows are in the same column, and also two pairs of queens lie on the same diagonal. Using this representation, we can write the algorithm very simply using eight nested loops.

```
program Queens 1
  for $i_1 \leftarrow 1$ to 8 do
    for $i_2 \leftarrow 1$ to 8 do
      .
      .
      .
        for $i_8 \leftarrow 1$ to 8 do
          try $\leftarrow (i_1, i_2, \ldots, i_8)$
          if solution(try) then write try
                                stop
  write "there is no solution"
```

This time, the number of cases to be considered is reduced to $8^8 = 16{,}777{,}216$, although in fact the algorithm finds a solution and stops after considering only 1,299,852 cases.

**Problem 6.6.2.** If you have not yet solved the previous problem, the information just given should be of considerable help ! □

Once we have realized that the chess-board can be represented by a vector, which prevents us from ever trying to put two queens in the same row, it is natural to be equally systematic in our use of the columns. Hence we now represent the board by a vector of eight *different* numbers between 1 and 8, that is, by a permutation of the first eight integers. The algorithm becomes

```
program Queens2
  try ← initial-permutation
  while try ≠ final-permutation and not solution(try) do
    try ← next-permutation
  if solution(try) then write try
                else write "there is no solution" .
```

There are several natural ways to generate systematically all the permutations of the first $n$ integers. For instance, we might put each value in turn in the leading position and generate recursively, for each of these leading values, all the permutations of the $n-1$ remaining elements.

```
procedure perm(i)
  if i = n then use(T)  { T is a new permutation }
           else for j ← i to n do exchange T[i] and T[j]
                                  perm(i+1)
                                  exchange T[i] and T[j]
```

Here $T[1..n]$ is a global array initialized to $[1,2,\ldots,n]$ and the initial call is *perm*(1).

**Problem 6.6.3.** If *use*($T$) consists simply of printing the array $T$ on a new line, show the result of calling *perm*(1) when $n = 4$. □

**Problem 6.6.4.** Assuming that *use*($T$) takes constant time, how much time is needed, as a function of $n$, to execute the call *perm*(1)? Now rework the problem assuming that *use*($T$) takes a time in $\Theta(n)$. □

This approach reduces the number of possible cases to $8! = 40{,}320$. If the preceding algorithm is used to generate the permutations, only 2,830 cases are in fact considered before the algorithm finds a solution. Although it is more complicated to generate permutations rather than all the possible vectors of eight integers between 1 and 8, it is, on the other hand, easier in this case to verify whether a given position is a solution. Since we already know that two queens can neither be in the same row nor in the same column, it suffices to verify that they are not in the same diagonal.

Starting from a crude method that tried to put the queens absolutely anywhere on the chess-board, we progressed first to a method that never puts two queens in the same row, and then to a better method still where the only positions considered are those where we know that two queens can neither be in the same row nor in the same column. However, all these algorithms share an important defect: they never test a position to see if it is a solution until all the queens have been placed on the board. For instance, even the best of these algorithms makes 720 useless attempts to put the last six queens on the board when it has started by putting the first two on the main diagonal, where of course they threaten one another!

Backtracking allows us to do better than this. As a first step, let us reformulate the eight queens problem as a tree searching problem. We say that a vector $V[1..k]$ of integers between 1 and 8 is *k-promising*, for $0 \le k \le 8$, if none of the $k$ queens placed in positions $(1, V[1])$, $(2, V[2])$, $\ldots$, $(k, V[k])$ threatens any of the others. Mathematically, a vector $V$ is $k$-promising if, for every $i \ne j$ between 1 and $k$, we have $V[i] - V[j] \notin \{i-j, 0, j-i\}$. For $k \le 1$, any vector $V$ is $k$-promising. Solutions to the eight queens problem correspond to vectors that are 8-promising.

Let $N$ be the set of $k$-promising vectors, $0 \le k \le 8$. Let $G = <N, A>$ be the directed graph such that $(U, V) \in A$ if and only if there exists an integer $k$, $0 \le k < 8$, such that $U$ is $k$-promising, $V$ is $(k+1)$-promising, and $U[i] = V[i]$ for every $i \in [1..k]$. This graph is a tree. Its root is the empty vector ($k = 0$). Its leaves are either solutions ($k = 8$) or else they are dead ends ($k < 8$) such as $[1, 4, 2, 5, 8]$ where it is impossible to place a queen in the next row without her threatening at least one of the queens already on the board. The solutions to the eight queens problem can be obtained by exploring this tree. We do not generate the tree explicitly so as to explore it thereafter, however: rather, nodes are generated and abandoned during the course of the exploration. Depth-first search is the obvious method to use, particularly if we only require one solution.

This technique has two advantages over the previous algorithm that systematically tried each permutation. First, the number of nodes in the tree is less than $8! = 40{,}320$. Although it is not easy to calculate this number theoretically, it is straightforward to count the nodes using a computer: $\#N = 2057$. In fact, it suffices to explore 114 nodes to obtain a first solution. Secondly, in order to decide whether a vector is $k$-promising, knowing that it is an extension of a $(k-1)$-promising vector, we only need to check the last queen to be added. This check can be speeded up if we associate with each promising node the sets of columns, of positive diagonals (at 45 degrees), and of negative diagonals (at 135 degrees) controlled by the queens already placed. On the other hand, to decide if some given permutation represents a solution, it seems at first sight that we have to check each of the 28 pairs of queens on the board.

To print all the solutions to the eight queens problem, call *Queens*$(0, \varnothing, \varnothing, \varnothing)$, where *try*$[1..8]$ is a global array.

```
procedure Queens ($k, col, diag45, diag135$)
  { $try[1..k]$ is $k$-promising,
    $col = \{try[i] \mid 1 \le i \le k\}$,
    $diag45 = \{try[i] - i + 1 \mid 1 \le i \le k\}$, and
    $diag135 = \{try[i] + i - 1 \mid 1 \le i \le k\}$ }
  if $k = 8$
  then { an 8-promising vector is a solution }
       write $try$
  else { explore $(k+1)$-promising extensions of $try$ }
       for $j \leftarrow 1$ to 8 do
         if $j \notin col$ and $j - k \notin diag45$ and $j + k \notin diag135$
         then $try[k+1] \leftarrow j$
              { $try[1..k+1]$ is $(k+1)$-promising }
              Queens ($k+1, col \cup \{j\}, diag45 \cup \{j-k\}, diag135 \cup \{j+k\}$)
```

It is clear that the problem generalizes to an arbitrary number of queens: how can we place $n$ queens on an $n \times n$ "chess-board" in such a way that none of them threatens any of the others?

**Problem 6.6.5.** Show that the problem for $n$ queens may have no solution. Find a more interesting case than $n = 2$. □

As we might expect, the advantage to be gained by using the backtracking algorithm instead of an exhaustive approach becomes more pronounced as $n$ increases. For example, for $n = 12$ there are 479,001,600 possible permutations to be considered, and the first solution to be found (using the generator given previously) corresponds to the 4,546,044th position examined; on the other hand, the tree explored by the backtracking algorithm contains only 856,189 nodes, and a solution is obtained when the 262nd node is visited.

**** Problem 6.6.6.** Analyse mathematically, as a function of the number $n$ of queens, the number of nodes in the tree of $k$-promising vectors. How does this number compare to $n!$ ? □

Backtracking algorithms can also be used even when the solutions sought do not necessarily all have the same length. Here is the general scheme.

```
procedure backtrack ($v[1..k]$)
  { $v$ is a $k$-promising vector }
  if $v$ is a solution then write $v$
  { otherwise } for each $(k+1)$-promising vector $w$
                    such that $w[1..k] = v[1..k]$ do backtrack($w[1..k+1]$)
```

The **otherwise** should be present if and only if it is impossible to have two different solutions such that one is a prefix of the other.

**Problem 6.6.7.** *Instant Insanity* is a puzzle consisting of four coloured cubes. Each of the 24 faces is coloured blue, red, green, or white. The four cubes are to be placed side by side in such a way that each colour appears on one of the four top faces, on one of the four bottom faces, on one of the front faces, and on one of the rear faces. Show how to solve this problem using backtracking. □

The $n$-queens problem was solved using depth-first search in the corresponding tree. Some problems that can be formulated in terms of exploring an implicit graph have the property that they correspond to an infinite graph. In this case, it may be necessary to use breadth-first search to avoid the interminable exploration of some fruitless infinite branch. Breadth-first search is also appropriate if we have to find a solution starting from some initial position and making as few changes as possible. (This last constraint does not apply to the eight queens problem where each solution involves exactly the same number of pieces.) The two following problems illustrate these ideas.

**Problem 6.6.8.** Give an algorithm capable of transforming some initial integer $n$ into a given final integer $m$ by the application of the smallest possible number of transformations $f(i) = 3i$ and $g(i) = \lfloor i/2 \rfloor$. For instance, 15 can be transformed into 4 using four function applications: $4 = gfgg(15)$. What does your algorithm do if it is impossible to transform $n$ into $m$ in this way? □

* **Problem 6.6.9.** Give an algorithm that determines the shortest possible series of manipulations needed to change one configuration of Rubik's Cube into another. If the required change is impossible, your algorithm should say so rather than calculating forever. □

**Problem 6.6.10.** Give other applications of backtracking. □

### 6.6.2 Graphs and Games: An Introduction

Most games of strategy can be represented in the form of directed graphs. A node of the graph corresponds to a particular position in the game, and an edge corresponds to a legal move between two positions. The graph is infinite if there is no a priori limit on the number of positions possible in the game. For simplicity, we assume that the game is played by two players, each of whom moves in turn, that the game is symmetric (the rules are the same for both players), and that chance plays no part in the outcome (the game is deterministic). The ideas we present can easily be adapted to more general contexts. We further suppose that no instance of the game can last forever and that no position in the game offers an infinite number of legal moves to the player whose turn it is. In particular, some positions in the game offer no legal moves, and hence some nodes in the graph have no successors: these are the *terminal positions*.

To determine a winning strategy for a game of this kind, we need only attach to each node of the graph a label chosen from the set *win, lose, draw.* The label corresponds to the situation of a player about to move in the corresponding position, assuming that neither player will make an error. The labels are assigned systematically in the following way.

i. The labels assigned to terminal positions depend on the game in question. For most games, if you find yourself in a terminal position, then there is no legal move you can make, and you have lost; but this is not necessarily the case (think of stalemate in chess).

ii. A nonterminal position is a winning position if *at least one* of its successors is a losing position.

iii. A nonterminal position is a losing position if *all* of its successors are winning positions.

iv. Any remaining positions lead to a draw.

**Problem 6.6.11.** Grasp intuitively how these rules arise. Can a player who finds himself in a winning position lose if his opponent makes an "error"? □

**Problem 6.6.12.** In the case of an acyclic finite graph (corresponding to a game that cannot continue for an indefinite number of moves), find a relationship between this method of labelling the nodes and topological sorting (Section 6.4.1). □

We illustrate these ideas with the help of a variant of Nim (also known as the Marienbad game). Initially, at least two matches are placed on the table between two players. The first player removes as many matches as he likes, except that he must take at least one and he must leave at least one. Thereafter, each player in turn must remove at least one match and at most twice the number of matches his opponent just took. The player who removes the last match wins. There are no draws.

**Example 6.6.1.** There are seven matches on the table initially. If I take two of them, my opponent may take one, two, three, or four. If he takes more than one, I can remove all the matches that are left and win. If he takes only one match, leaving four matches on the table, I can in turn remove a single match, and he cannot prevent me from winning on my next turn. On the other hand, if at the outset I choose to remove a single match, or to remove more than two, then you may verify that my opponent has a winning strategy.

The player who has the first move in a game with seven matches is therefore certain to win provided that he does not make an error. On the other hand, you may verify that a player who has the first move in a game with eight matches cannot win unless his opponent makes an error. □

***Problem 6.6.13.*** For $n \geq 2$, give a necessary and sufficient condition on $n$ to ensure that the player who has the first move in a game involving $n$ matches have a winning strategy. Your characterization of $n$ should be as simple as possible. Prove your answer. □

A position in this game is not specified merely by the number of matches that remain on the table. It is also necessary to know the upper limit on the number of matches that it is permissible to remove on the next move. The nodes of the graph corresponding to this game are therefore pairs $<i,j>$. In general, $<i,j>$, $1 \leq j \leq i$, indicates that $i$ matches remain on the table and that any number of them between 1 and $j$ may be removed in the next move. The edges leaving node $<i,j>$ go to the $j$ nodes $<i-k, \min(2k, i-k)>$, $1 \leq k \leq j$. The node corresponding to the initial position in a game with $n$ matches, $n \geq 2$, is $<n, n-1>$. All the nodes whose second component is zero correspond to terminal positions, but only $<0,0>$ is interesting: the nodes $<i,0>$ for $i > 0$ are inaccessible. Similarly, nodes $<i,j>$ with $j$ odd and $j < i-1$ cannot be reached starting from any initial position.

Figure 6.6.1 shows part of the graph corresponding to this game. The square nodes represent losing positions and the round nodes are winning positions. The heavy edges correspond to winning moves: in a winning position, choose one of the heavy edges in order to win. There are no heavy edges leaving a losing position, corresponding to the fact that such positions offer no winning move.

We observe that a player who has the first move in a game with two, three, or five matches has no winning strategy, whereas he does have such a strategy in the game with four matches.

**Problem 6.6.14.** Add nodes $<8,7>$, $<7,6>$, $<6,5>$ and their descendants to the graph of Figure 6.6.1. □

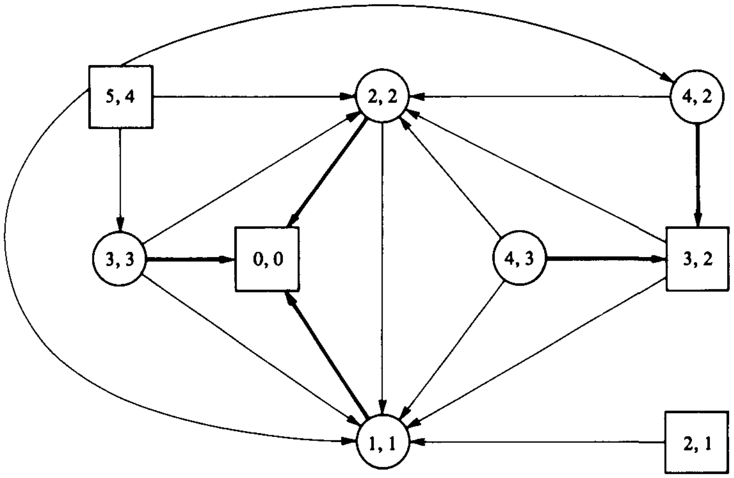

**Figure 6.6.1.** Part of a game graph.

**Problem 6.6.15.** Can a winning position have more than one losing position among its successors? In other words, are there positions in which several different winning moves are available? Can this happen in the case of a winning initial position $<n, n-1>$? □

The obvious algorithm to determine whether a position is winning is the following.

```
function rec(i, j)
  { returns true if and only if the node <i, j> is winning;
    we assume that $0 \le j \le i$ }
  for k ← 1 to j do
    if not rec(i−k, min(2k, i−k))
    then return true
  return false
```

**Problem 6.6.16.** Modify this algorithm so that it returns an integer $k$ such that $k = 0$ if the position is a losing position and $1 \le k \le j$ if it is a winning move to take away $k$ matches. □

This algorithm suffers from the same defect as the algorithm *fib*1 in Section 1.7.5: it calculates the same value over and over. For instance, *rec*(5,4) returns *false* having called successively *rec*(4,2), *rec*(3,3), *rec*(2,2) and *rec*(1,1), but *rec*(3,3) also calls *rec*(2,2) and *rec*(1,1).

**Problem 6.6.17.** Find two ways to remove this inefficiency. (If you want to work on this problem, do not read the following paragraphs yet!) □

The first approach consists of using dynamic programming to create a Boolean array $G$ such that $G[i,j] = true$ if and only if $<i,j>$ is a winning position. As usual with dynamic programming, we proceed in a bottom-up fashion, calculating all the values of $G[l,k]$ for $1 \le k \le l < i$, as well as all the values of $G[i,k]$ for $1 \le k < j$, before calculating $G[i,j]$.

```
procedure dyn(n)
  { for each $1 \le j \le i \le n$, G[i, j] is set to true
    if and only if configuration <i, j> is winning }
  G[0,0] ← false
  for i ← 1 to n do
    for j ← 1 to i do
      k ← 1
      while k < j and G[i−k, min(2k, i−k)] do k ← k + 1
      G[i, j] ← not G[i−k, min(2k, i−k)]
```

**Problem 6.6.18.** The preceding algorithm only uses $G[0,0]$ and the values of $G[l,k]$, $1 \le k \le l < i$, to calculate $G[i,j]$. Show how to improve its efficiency by also using the values of $G[i,k]$ for $1 \le k < j$. □

In this context dynamic programming leads us to calculate wastefully some entries of the array $G$ that are never needed. For instance, we know that $<15,14>$ is a winning position as soon as we discover that its second successor $<13,4>$ is a losing position. It is no longer of any interest to know whether the next successor $<12,6>$ is a winning or a losing position. In fact, only 27 nodes are really useful when we calculate $G[15,14]$, although the dynamic programming algorithm looks at 121 of them. About half this work can be avoided if we do not calculate $G[i,j]$ whenever $j$ is odd and $j < i-1$, since these nodes are never of interest, but there is no "bottom-up" reason for not calculating $G[12,6]$.

The recursive algorithm given previously is inefficient because it recalculates the same value several times. Because of its top-down nature, however, it never calculates an unnecessary value. A solution that combines the advantages of both the algorithms consists of using a memory function (Section 5.7). This involves remembering which nodes have already been visited during the recursive computation using a global Boolean array $init[0..n, 0..n]$, initialized to *false*, where $n$ is an upper bound on the number of matches to be used.

```
function nim(i, j)
  if init[i, j] then return G[i, j]
  init[i, j] ← true
  for k ← 1 to j do
    if not nim(i−k, min(2k, i−k)) then G[i, j] ← true
                                       return true
  G[i, j] ← false
  return false
```

At first sight, there is no particular reason to favour this approach over dynamic programming, because in any case we have to take the time to initialize the whole array $init[0..n, 0..n]$. Using the technique suggested in Problem 5.7.2 allows us, however, to avoid this initialization and to obtain a worthwhile gain in efficiency.

The game we have considered up to now is so simple that it can be solved without really using the associated graph. Here, without explanation, is an algorithm for determining a winning strategy that is more efficient than any of those given previously. In an initial position with $n$ matches, first call *precond*$(n)$. Thereafter a call on *whatnow*$(i, j)$, $1 \le j \le i$, determines in a time in $\Theta(1)$ the move to make in a situation where $i$ matches remain on the table and the next player has the right to take at most $j$ of them. The array $T[0..n]$ is global. The initial call of *precond*$(n)$ is an application of the preconditioning technique to be discussed in the next chapter.

```
procedure precond(n)
  T[0] ← ∞
  for i ← 1 to n do
    k ← 1
    while T[i−k] ≤ 2k do k ← k + 1
    T[i] ← k

function whatnow(i, j)
  if j < T[i] then { prolong the agony! }
                   return 1
  return T[i]
```

*** Problem 6.6.19.** Prove that this algorithm works correctly and that *precond*$(n)$ takes a time in $\Theta(n)$. □

Consider now a more complex game, namely chess. At first sight, the graph associated with this game contains cycles, since if two positions $u$ and $v$ of the pieces differ only by the legal move of a rook, say, the king not being in check, then we can move equally well from $u$ to $v$ and from $v$ to $u$. However, this problem disappears on closer examination. Remember first that in the game we just looked at, a position is defined not merely by the number of matches on the table, but also by an invisible item of information giving the number of matches that can be removed on the next move. Similarly, a position in chess is not defined simply by the positions of the pieces on the board. We also need to know whose turn it is to move, which rooks and kings have moved since the beginning of the game (to know if it is legal to castle), and whether some pawn has just been moved two squares forward (to know whether a capture *en passant* is possible). Furthermore, the International Chess Federation has rules that prevent games dragging on forever: for example, a game is declared to be a draw after 50 moves in which no irreversible action (movement of a pawn, or a capture) took place. Thus we must include in our notion of position the number of moves made since the last irreversible action. Thanks to such rules, there are no cycles in the graph corresponding to chess. (For simplicity we ignore exceptions to the 50-move rule, as well as the older rule that makes a game a draw if the pieces return three times to exactly the same positions on the board.)

Adapting the general rules given at the beginning of this section, we can therefore label each node as being a winning position for White, a winning position for Black, or a draw. Once constructed, this graph allows us to play a perfect game of chess, that is, to win whenever it is possible and to lose only when it is inevitable. Unfortunately (or perhaps fortunately for the game of chess), the graph contains so many nodes that it is quite out of the question to explore it completely, even with the fastest existing computers.

*** Problem 6.6.20.** Estimate the number of ways in which the pieces can be placed on a chess-board. For simplicity ignore the fact that certain positions are impossible, that is, they can never be obtained from the initial position by a legal series of moves (but take into account the fact that each bishop moves only on either white or black squares, and that both kings must be on the board). Ignore also the possibility of having promoted pawns. □

Since a complete search of the graph associated with the game of chess is out of the question, it is not practical to use a dynamic programming approach. In this situation the recursive approach comes into its own. Although it does not allow us to be certain of winning, it underlies an important heuristic called *minimax*. This technique finds a move that may reasonably be expected to be among the best moves possible while exploring only a part of the graph starting from some given position. Exploration of the graph is usually stopped before the leaves are reached, using one of several possible criteria, and the positions thus reached are evaluated heuristically. Then we make the move that seems to cause our opponent the most trouble. This is in a sense merely a systematic version of the method used by some human players that consists of looking ahead a small number of moves. Here we give only an outline of the technique.

**The minimax principle.** The first step is to define a static evaluation function *eval* that attributes some value to each possible position. Ideally, we want the value of *eval*$(u)$ to increase as the position $u$ becomes more favourable to White. It is customary to give values not too far from zero to positions where neither side has a marked advantage, and large negative values to positions that favour Black. This evaluation function must take account of many factors: the number and the type of pieces remaining on both sides, control of the centre, freedom of movement, and so on. A compromise must be made between the accuracy of this function and the time needed to calculate it. When applied to a terminal position, the evaluation function should return $+\infty$ if Black has been mated, $-\infty$ if White has been mated, and 0 if the game is a draw. For example, an evaluation function that takes good account of the static aspects of the position but that is too simplistic to be of real use might be the following: for nonterminal configurations, count 1 point for each white pawn, $3^1/_4$ points for each white bishop or knight, 5 points for each white rook, and 10 points for each white queen; subtract a similar number of points for each black piece.

If the static evaluation function were perfect, it would be easy to determine the best move to make. Suppose it is White's turn to move from position $u$. The best move would be to go to the position $v$ that maximizes *eval*$(v)$ among all the successors $w$ of $u$.

```
val ← −∞
for each configuration w that is a successor of u do
  if eval(w) ≥ val then val ← eval(w)
                        v ← w
```

It is clear that this simplistic approach would not be very successful using the evaluation function suggested earlier, since it would not hesitate to sacrifice a queen in order to take a pawn!

If the evaluation function is not perfect, a better strategy for White is to assume that Black will reply with the move that minimizes the function *eval*, since the smaller the value taken by this function, the better the position is supposed to be for him. (Ideally, he would like a large negative value.) We are now looking half a move ahead.

```
val ← −∞
for each configuration w that is a successor of u do
  if w has no successor
  then valw ← eval(w)
  else valw ← min{eval(x) | x is a successor of w}
  if valw ≥ val then val ← valw
                     v ← w
```

There is now no question of giving away a queen to take a pawn, which of course may be exactly the wrong rule to apply if it prevents White from finding the winning move: maybe if he looked further ahead the gambit would turn out to be profitable. On the other hand, we are sure to avoid moves that would allow Black to mate immediately (provided we *can* avoid this).

To add more dynamic aspects to the static evaluation provided by *eval*, it is preferable to look several moves ahead. To look $n$ half-moves ahead from position $u$, White should move to the position $v$ given by

```
val ← −∞
for each configuration w that is a successor of u do
  if Black(w, n) ≥ val then val ← Black(w, n)
                            v ← w
```

where the functions *Black* and *White* are the following:

```
function Black(w, n)
  if n = 0 or w has no successor
  then return eval(w)
  else return min{White(x, n−1) | x is a successor of w}

function White(x, n)
  if n = 0 or x has no successor
  then return eval(x)
  else return max{Black(w, n−1) | w is a successor of x} .
```

We see why the technique is called minimax: Black tries to minimize the advantage he allows to White, and White, on the other hand, tries to maximize the advantage he obtains from each move.

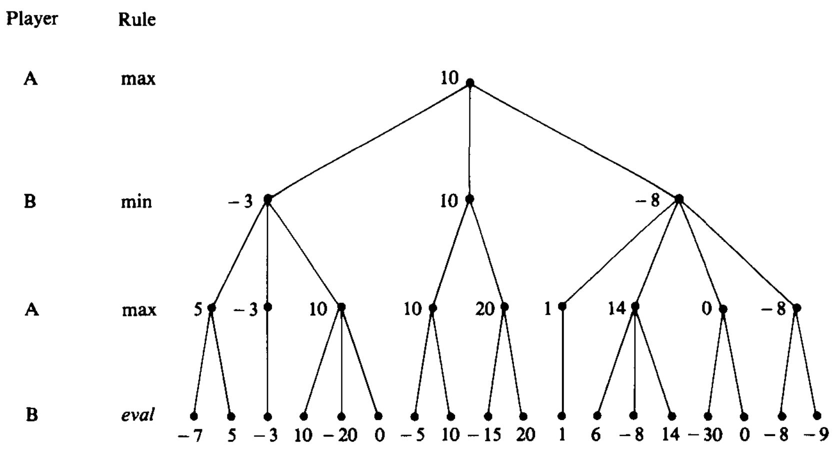

**Figure 6.6.2.** The minimax principle.

**Problem 6.6.21.** Let $u$ correspond to the initial position of the pieces. What can you say about $White(u, 12800)$, besides the fact that it would take far too long to calculate in practice? Justify your answer. □

**Example 6.6.2.** Figure 6.6.2 shows part of the graph corresponding to some game. If the values attached to the leaves are obtained by applying the function *eval* to the corresponding positions, the values for the other nodes can be calculated using the minimax rule. In the example we suppose that player A is trying to maximize the evaluation function and that player B is trying to minimize it.

If A plays so as to maximize his advantage, he will choose the second of the three possible moves. This assures him of a value of at least 10. □

**Alpha-beta pruning.** The basic minimax technique can be improved in a number of ways. For example, it may be worthwhile to explore the most promising moves in greater depth. Similarly, the exploration of certain branches can be abandoned early if the information we have about them is already sufficient to show that they cannot possibly influence the values of nodes farther up the tree. This second type of improvement is generally known as *alpha-beta pruning*. We give just one simple example of the technique.

**Example 6.6.3.** Look back at Figure 6.6.2. Let $<i,j>$ represent the $j$th node in the $i$th row of the tree. We want to calculate the value of the root $<1,1>$ starting from the values calculated by the function *eval* for the leaves $<4,j>$, $1 \le j \le 18$. To do this, we carry out a bounded depth-first search in the tree, visiting the successors of a given node from left to right.

If we want to abandon the exploration of certain branches because it is no longer useful, we have to transmit immediately to the higher levels of the tree any information obtained by evaluating a leaf. Thus as soon as the first leaf $<4,1>$ is evaluated, we know that $<4,1>$ has value $-7$ and that $<3,1>$ (a node that maximizes *eval*) has value at least $-7$. After evaluation of the second leaf $<4,2>$, we know that $<4,2>$ has value 5, $<3,1>$ has value 5, and $<2,1>$ (a node that minimizes *eval*) has value at most 5.

Continuing in this way, we arrive after evaluation of the leaf $<4,4>$ at the situation illustrated in Figure 6.6.3. Since node $<3,3>$ has value at least 10, whereas node $<2,1>$ has value at most $-3$, the exact value of node $<3,3>$ cannot have any influence on the value of node $<2,1>$. It is therefore unnecessary to evaluate the other descendants of node $<3,3>$; we say that the corresponding branches of the tree have been *pruned.*

Similarly, after evaluation of the leaf $<4,11>$, we are in the situation shown in Figure 6.6.4. Node $<2,3>$ has value at most 1. Since we already know that the value of $<1,1>$ is at least 10, there is no need to evaluate the other children of node $<2,3>$.

To establish that the value of the root $<1,1>$ is 10, we visit only 19 of the 31 nodes in the tree. □

**** Problem 6.6.22.** Write a program capable of playing brilliantly your favourite game of strategy. □

**** Problem 6.6.23.** Write a program that can beat the world backgammon champion. (This has already been done!) □

**** Problem 6.6.24.** What modifications should be made to the principles set out in this section to take account of those games of strategy in which chance plays a certain part? What about games with more than two players ? □

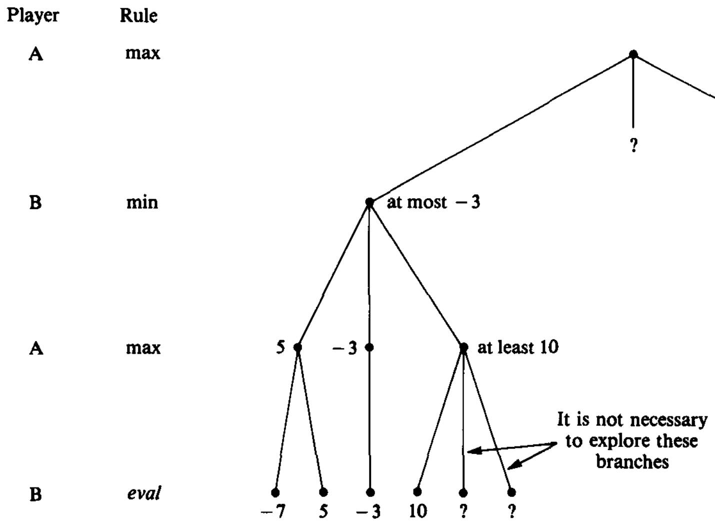

**Figure 6.6.3.** Alpha-beta pruning.

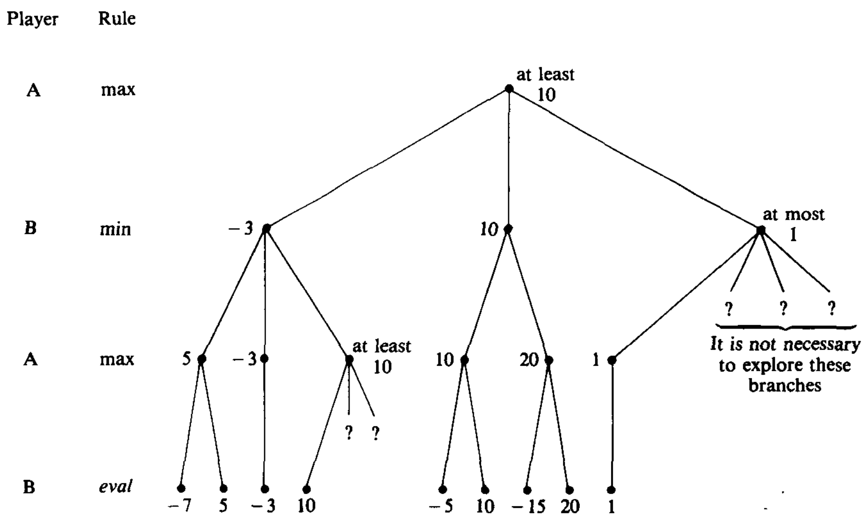

**Figure 6.6.4.** More alpha-beta pruning.

### 6.6.3 Branch-and-Bound

Like backtracking, branch-and-bound is a technique for exploring an implicit directed graph. Again, this graph is usually acyclic or even a tree. This time, we are looking for the optimal solution to some problem. At each node we calculate a bound on the possible value of any solutions that might happen to be farther on in the graph. If the bound shows that any such solution must necessarily be worse than the best solution we have found so far, then we do not need to go on exploring this part of the graph.

In the simplest version, calculation of these bounds is combined with a breadth-first or a depth-first search, and serves only, as we have just explained, to prune certain branches of a tree or to close certain paths in a graph. More often, however, the calculated bound is used not only to close off certain paths, but also to choose which of the open paths looks the most promising, so that it can be explored first.

In general terms we may say that a depth-first search finishes exploring nodes in inverse order of their creation, using a stack to hold those nodes that have been generated but not yet explored fully; a breadth-first search finishes exploring nodes in the order of their creation, using this time a queue to hold those that have been generated but not yet explored (see Section 6.5). Branch-and-bound uses auxiliary computations to decide at each instant which node should be explored next, and a priority list to hold those nodes that have been generated but not yet explored.

An example illustrates the technique.

**Example 6.6.4.** We return to the travelling salesperson problem (see Sections 3.4.2 and 5.6).

Let $G$ be the complete graph on five points with the following distance matrix:

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}.$$

We are looking for the shortest tour starting from node 1 that passes exactly once through each other node before finally returning to node 1.

The nodes in the implicit graph correspond to partially specified paths. For instance, node $(1,4,3)$ corresponds to two complete tours: $(1,4,3,2,5,1)$ and $(1,4,3,5,2,1)$. The successors of a given node correspond to paths in which one additional node has been specified. At each node we calculate a lower bound on the length of the corresponding complete tours.

To calculate this bound, suppose that half the distance between two points $i$ and $j$ is counted at the moment we leave $i$, and the other half when we arrive at $j$. For instance, leaving node 1 costs us at least 2, namely the lowest of the values 14/2, 4/2, 10/2, and 20/2. Similarly, visiting node 2 costs us at least 6 (at least 5/2 when we arrive and at least 7/2 when we leave). Returning to node 1 costs at least 2, the minimum of 14/2, 4/2, 11/2, and 18/2. To obtain a bound on the length of a path, it suffices to add elements of this kind. For instance, a complete tour must include a departure from node 1, a visit to each of the nodes 2, 3, 4, and 5 (not necessarily in this order) and a return to 1. Its length is therefore at least

$$2+6+4+3+3+2=20\ .$$

Notice that this calculation does not imply the existence of a solution that costs only 20.

In Figure 6.6.5 the root of the tree specifies that the starting point for our tour is node 1. Obviously, this arbitrary choice of a starting point does not alter the length of the shortest tour. We have just calculated the lower bound shown for this node. (This bound on the root of the implicit tree serves no purpose in the algorithm; it was computed here for the sake of illustration.) Our search begins by generating (as though for a breadth-first search) the four possible successors of the root, namely, nodes $(1,2)$, $(1,3)$, $(1,4)$, and $(1,5)$. The bound for node $(1,2)$, for example, is calculated as follows. A tour that begins with $(1,2)$ must include

- The trip $1-2$: 14 (formally, leaving 1 for 2 and arriving at 2 from 1: $7+7$)
- A departure from 2 toward 3, 4, or 5: minimum 7/2
- A visit to 3 that neither comes from 1 nor leaves for 2: minimum 11/2
- A similar visit to 4: minimum 3
- A similar visit to 5: minimum 3
- A return to 1 from 3, 4, or 5: minimum 2 .

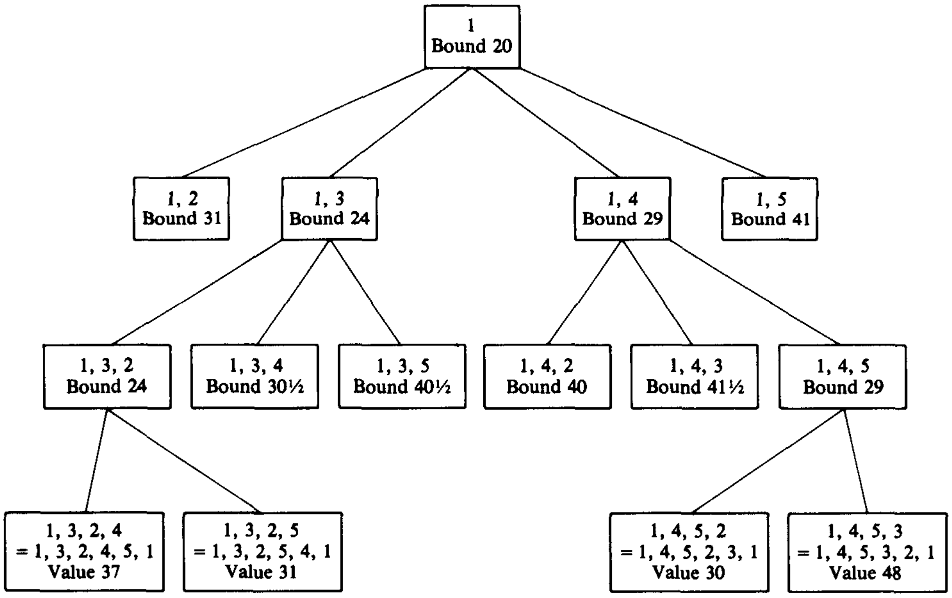

**Figure 6.6.5.** Branch-and-bound.

The length of such a tour is therefore at least 31. The other bounds are calculated similarly.

Next, the most promising node seems to be $(1,3)$, whose bound is 24. The three children $(1,3,2)$, $(1,3,4)$, and $(1,3,5)$ of this node are therefore generated. To give just one example, we calculate the bound for node $(1,3,2)$ as follows:

- The trip $1-3-2$: 9
- A departure from 2 toward 4 or 5: minimum $7/2$
- A visit to 4 that comes from neither 1 nor 3 and that leaves for neither 2 nor 3: minimum 3
- A similar visit to 5: minimum 3
- A return to 1 from 4 or 5: minimum $11/2$,

which gives a total length of at least 24.

The most promising node is now $(1,3,2)$. Its two children $(1,3,2,4)$ and $(1,3,2,5)$ are generated. This time, as node $(1,3,2,4)$, for instance, corresponds to exactly one complete tour $(1,3,2,4,5,1)$, we do not need to calculate a lower bound since we may calculate immediately its length 37.

We find that the length of the tour $(1,3,2,5,4,1)$ is 31. If we are only concerned to find one optimal solution, we do not need to continue exploration of the nodes $(1,2)$, (1,5) and (1,3,5), which cannot possibly lead to a better solution. Even exploration of the node (1,3,4) is pointless. (Why?) There remains only node (1,4) to explore. The only child to offer interesting possibilities is (1,4,5). After looking at the two complete tours (1,4,5,2,3,1) and (1,4,5,3,2,1), we find that the tour (1,4,5,2,3,1) of length 30 is optimal. This example illustrates the fact that although at one point (1,3) was the most promising node, the optimal solution does not come from there.

To obtain our answer, we have looked at merely 15 of the 41 nodes that are present in a complete tree of the type illustrated in Figure 6.6.5. □

**Problem 6.6.25.** Solve the same problem using the method of Section 5.6. □

* **Problem 6.6.26.** Implement this algorithm on a computer and test it on our example. □

**Problem 6.6.27.** Show how to solve the same problem using a backtracking algorithm that calculates a bound as shown earlier to decide whether or not a partially defined path is promising. □

The need to keep a list of nodes that have been generated but not yet completely explored, situated in all the levels of the tree and preferably sorted in order of the corresponding bounds, makes branch-and-bound quite hard to program. The heap is an ideal data structure for holding this list. Unlike depth-first search and its related techniques, no elegant recursive formulation of branch-and-bound is available to the programmer. Nevertheless, the technique is sufficiently powerful that it is often used in practical applications.

It is next to impossible to give any idea of how well the technique will perform on a given problem using a given bound. There is always a compromise to be made concerning the quality of the bound to be calculated: with a better bound we look at less nodes, but on the other hand, we shall most likely spend more time at each one calculating the corresponding bound. In the worst case it may turn out that even an excellent bound does not allow us to cut any branches off the tree, and all the extra work we have done is wasted. In practice, however, for problems of the size encountered in applications, it almost always pays to invest the necessary time in calculating the best possible bound (within reason). For instance, one finds applications such as integer programming handled by branch-and-bound, the bound at each node being obtained by solving a related problem in linear programming with continuous variables.

## 6.7 SUPPLEMENTARY PROBLEMS

**Problem 6.7.1.** Write algorithms to determine whether a given undirected graph is in fact a tree (i) using a depth-first search; (ii) using a breadth-first search. How much time do your algorithms take? □

**Problem 6.7.2.** Write an algorithm to determine whether a given directed graph is in fact a rooted tree, and if so, to find the root. How much time does your algorithm take? □

* **Problem 6.7.3.** A node $p$ of a directed graph $G = <N, A>$ is called a *sink* if for every node $v \in N$, $v \neq p$, the edge $(v,p)$ exists, whereas the edge $(p,v)$ does not exist. Write an algorithm that can detect the presence of a sink in $G$ in a time in $O(n)$. Your algorithm should accept the graph represented by its adjacency matrix (type *adjgraph* of Section 1.9.2). Notice that a running time in $O(n)$ for this problem is remarkable given that the instance takes a space in $\Omega(n^2)$ merely to write down. □

* **Problem 6.7.4. Euler's problem.** An *Euler path* in a finite undirected graph is a path such that every edge appears in it exactly once. Write an algorithm that determines whether or not a given graph has an Euler path, and prints the path if so. How much time does your algorithm take? □

* **Problem 6.7.5.** Repeat Problem 6.7.4 for a directed graph. □

**Problem 6.7.6.** The value 1 is available. To construct other values, you have available the two operations ×2 (multiplication by 2) and /3 (division by 3, any resulting fraction being dropped). Operations are executed from left to right. For instance

$$10 = 1 \times 2 \times 2 \times 2 \times 2 / 3 \times 2 .$$

We want to express 13 in this way. Show how the problem can be expressed in terms of exploring a graph and find a minimum-length solution. □

* **Problem 6.7.7.** Show how the problem of carrying out a syntactic analysis of a programming language can be solved in top-down fashion using a backtracking algorithm. (This approach is used in a number of compilers.) □

**Problem 6.7.8.** A Boolean array $M[1..n, 1..n]$ represents a square maze. In general, starting from a given point, it is permissible to go to adjacent points in the same row or in the same column. If $M[i,j]$ is *true*, then you may pass through point $(i,j)$; if $M[i,j]$ is *false*, then you may not pass through point $(i,j)$. Figure 6.7.1 gives an example.

**i.** Give a backtracking algorithm that finds a path, if one exists, from $(1,1)$ to $(n,n)$. Without being completely formal (for instance, you may use statements such as "**for** each point $v$ that is a neighbour of $x$ **do** $\cdots$"), your algorithm must be clear and precise.

**ii.** Without giving all the details of the algorithm, indicate how to solve this problem by branch-and-bound. □

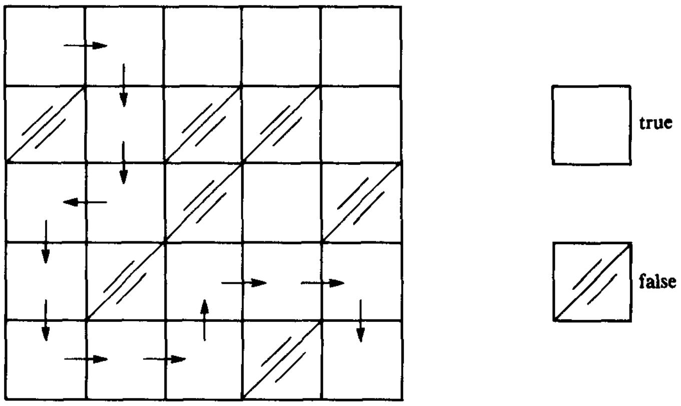

Figure 6.7.1. A maze.

## 6.8 REFERENCES AND FURTHER READING

There exist a number of books concerning graph algorithms or combinatorial problems that are often posed in terms of graphs. We mention in chronological order Christofides (1975), Lawler (1976), Reingold, Nievergelt, and Deo (1977), Gondran and Minoux (1979), Even (1980), Papadimitriou and Steiglitz (1982), and Tarjan (1983). The mathematical notion of a graph is treated at length in Berge (1958, 1970).

A solution of problem 6.2.2 is given in Robson (1973).

Several applications of depth-first search are taken from Tarjan (1972) and Hopcroft and Tarjan (1973). Problem 6.3.10 is solved in Rosenthal and Goldner (1977). A linear time algorithm for testing the planarity of a graph is given in Hopcroft and Tarjan (1974). Other algorithms based on depth-first search appear in Aho, Hopcroft, and Ullman (1974, 1983).

Backtracking is described in Golomb and Baumert (1965) and techniques for analysing its efficiency are given in Knuth (1975a). Some algorithms for playing chess appear in Good (1968). The book by Nilsson (1971) is a gold mine of ideas concerning graphs and games, the minimax technique, and alpha-beta pruning. The latter is analysed in Knuth (1975b). A lively account of the first time a computer program beat the world backgammon champion (Problem 6.6.23) is given in Deyong (1977). For a more technical description of this feat, consult Berliner (1980). The branch-and-bound technique is explained in Lawler and Wood (1966). The use of this technique to solve the travelling salesperson problem is described in Bellmore and Nemhauser (1968).