begin startplot;
  i := 0; h := h0 div 4; x0 := 2*h; y0 := 3*h;
repeat i := i+1; x0 := x0−h;
  h := h div 2; y0 := y0+h;
  x := x0; y := y0; setplot;
  A(i); x := x+h; y := y−h; plot;
  B(i); x := x−h; y := y−h; plot;
  C(i); x := x−h; y := y+h; plot;
  D(i); x := x+h; y := y+h; plot;
until i = n;
endplot

Program 3.2 (Continued)

depth of recursion cannot become greater than n. In contrast to this recursive formulation, equivalent programs that avoid the explicit use of recursion are extremely cumbersome and obscure. The reader is urged to convince himself of this claim by trying to understand the programs shown in [3-3].

3.4. BACKTRACKING ALGORITHMS

A particularly intriguing programming endeavor is the subject of "general problem solving." The task is to determine algorithms for finding solutions to specific problems not by following a fixed rule of computation, but by trial and error. The common pattern is to decompose the trial-and-error process into partial tasks. Often these tasks are most naturally expressed in recursive terms and consist of the exploration of a finite number of subtasks. We may generally view the entire process as a trial or search process that gradually builds up and scans (prunes) a tree of subtasks. In many problems this search tree grows very rapidly, usually exponentially, depending on a given parameter. The search effort increases accordingly. Frequently, the search tree can be pruned by the use of heuristics only, thereby reducing computation to tolerable bounds.

It is not our aim to discuss general heuristic rules in this text. Rather, the general principle of breaking up such problem-solving tasks into subtasks and the application of recursion is to be the subject of this chapter. We start out by demonstrating the underlying technique by using an example, namely, the well-known knight's tour.

Given is a $n \times n$ board with $n^2$ fields. A knight—being allowed to move according to the rules of chess—is placed on the field with initial coordinates $x_0, y_0$. The problem is to find a covering of the entire board, if there exists one, i.e., to compute a tour of $n^2 − 1$ moves such that every field of the board is visited exactly once.
The obvious way to reduce the problem of covering \( n^2 \) fields is to consider the problem of either performing a next move or finding out that none possible. Let us therefore define an algorithm trying to perform a next move. A first approach is shown in (3.24).

\[
\text{procedure try next move;}
\begin{align*}
\text{begin initialize selection of moves; O} \\
\text{repeat select next candidate from list of next moves; O} \\
\quad \text{if acceptable then} \\
\qquad \text{begin record move;} \\
\qquad \quad \text{if board not full then} \\
\qquad \qquad \text{begin try next move;} \\
\qquad \qquad \quad \text{if not successful then erase previous recording} \\
\quad \end{align*}
\]
\[\text{until (move was successful) \lor \text{ (no more candidates)} \quad \text{end}}
\]

If we wish to be more precise in describing this algorithm, we are forced to make some decisions on data representation. An obvious step is to represent the board by a matrix, say \( h \). Let us also introduce a type to denote index values:

\[
\text{type index = } 1 \ldots n; \\
\text{var } h: \text{array [index, index] of integer}
\]

(3.25)

The decision to represent each field of the board by an integer instead of a Boolean value denoting occupation is because we wish to keep track of the history of successive board occupations. The following convention is an obvious choice:

\[
\begin{align*}
\text{if } h[x, y] = 0: & \quad \text{field } \langle x, y \rangle \text{ has not been visited} \\
\text{if } h[x, y] = i: & \quad \text{field } \langle x, y \rangle \text{ has been visited in the } i\text{th move} \\
& \quad \text{for } (1 \leq i \leq n^2)
\end{align*}
\]

(3.26)

The next decision concerns the choice of appropriate parameters. They are to determine the starting conditions for the next move and also to report on its success. The former task is adequately solved by specifying the coordinates \( x, y \) from which the move is to be made and by specifying the number \( i \) of the move (for recording purposes). The latter task requires a Boolean result parameter: \( q = \text{true} \) denotes success; \( q = \text{false} \) failure.

Which statements can now be refined on the basis of these decisions? Certainly "board not full" can be expressed as "\( i \leq n^2 \)." Moreover, if we introduce two local variables \( u \) and \( v \) to stand for the coordinates of possible move destinations determined according to the jump pattern of knights, then the predicate "acceptable" can be expressed as the logical combination of the conditions that the new field lies on the board, i.e., \( 1 \leq u \leq n \) and
1 \leq v \leq n$, and that it has not been visited previously, i.e., $h[u, v] = 0$. The operation of recording the legal move is expressed by the assignment $h[u, v] := i$, and the cancellation of this recording as $h[u, v] := 0$. If a local variable $q_1$ is introduced and used as the result parameter in the recursive call of this algorithm, then $q_1$ may be substituted for "move was successful." Thereby we arrive at the formulation shown in (3.27).

\begin{verbatim}
procedure try (i: integer; x,y: index; var q: boolean);
  var u,v: integer; q1: boolean;
  begin initialize selection for moves;
    repeat let u,v be the coordinates of the next move defined
      by the rules of chess;
        if (1 \leq u \leq n) \land (1 \leq v \leq n) \land (h[u,v] = 0) then
          begin h[u,v] := i;
            if i < sqr(n) then
              begin try(i+1,u,v,q1);
                if \neg q1 then h[u,v] := 0
              end else q1 := true
            end
        until q1 \lor \text{(no more candidates)};
        q := q1
    end

Just one more refinement step will lead us to a program expressed fully
in terms of our basic programming notation. We should note that so far
the program was developed completely independently of the laws governing
the jumps of the knight. This delaying of considerations of particularities
of the problem was quite deliberate. But now is the time to take them into
account.

Given a starting coordinate pair $(x, y)$, there are eight potential candidates
for coordinates $(u, v)$ of the destination. They are numbered 1 to
8 in Fig. 3.8.

A simple method of obtaining $u, v$ from $x, y$ is by addition of the coordinate
differences stored in either an array of difference pairs or in two arrays

\begin{center}
\begin{tabular}{|c|c|}
\hline
3 & 2 \\
4 & 1 \\
5 & 8 \\
6 & 7 \\
\hline
\end{tabular}
\end{center}

\textbf{Fig. 3.8} The eight possible moves of a knight.
program knightstour (output);
const n = 5; nsq = 25;
type index = 1 .. n;
var i,j: index;
q: boolean;
s: set of index;
a,b: array [1..8] of integer;
h: array [index, index] of integer;

procedure try (i: integer; x,y: index; var q: boolean);
var k,u,v: integer; q1: boolean;
begin k := 0;
repeat k := k+1; q1 := false;
u := x + a[k]; v := y + b[k];
if (u in s) \ (v in s) then
if h[u,v] = 0 then
begin h[u,v] := i;
if i < nsq then
begin try (i+1,u,v,q1);
if not q1 then h[u,v] := 0
end else q1 := true
end
until q1 \ (k=8);
q := q1
end try;

begin s := [1,2,3,4,5];
a[1] := 2; b[1] := 1;
a[8] := 2; b[8] := -1;
for i := 1 to n do
for j := 1 to n do h[i,j] := 0;
h[1,1] := 1; try(2,1,1,q);
if q then
for i := 1 to n do
begin for j := 1 to n do write(h[i,j]:5);
writeln
end
else writeln(' NO SOLUTION ')
end .

Program 3.3 Knight's Tour.
of single differences. Let these arrays be denoted by \( a \) and \( b \), appropriately initialized. Then an index \( k \) may be used to number the “next candidate.” The details are shown in Program 3.3. The recursive procedure is initiated by a call with the coordinates \( x_0, y_0 \) of that field as parameters, from which the tour is to start. This field must be given the value 1; all others are to be marked free.

\[
H[x_0, y_0] := 1;
\text{try (2, } x_0, y_0, q)\
\]

One further detail must not be overlooked: A variable \( h[u,v] \) does exist only if both \( u \) and \( v \) lie within the array bounds 1 . . . \( n \). Consequently, the expression in (3.27), substituted for “acceptable” in (3.24), is valid only if its first two constituent terms are true. A proper reformulation is shown in Program 3.3 in which, furthermore, the double relation \( 1 \leq u \leq n \) is replaced by the expression \( u \in [1, 2, \ldots, n] \), which for sufficiently small \( n \) is possibly more efficient (see Sect. 1.10.3). Table 3.1 indicates solutions obtained with initial positions \( \langle 1, 1 \rangle, \langle 3, 3 \rangle \) for \( n = 5 \) and \( \langle 1, 1 \rangle \) for \( n = 6 \).

It is possible to replace the result parameter \( q \) and the local variable \( q' \) by a global variable, thereby simplifying the program somewhat.

<table>
<thead>
<tr>
<th>1</th>
<th>6</th>
<th>15</th>
<th>10</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>9</td>
<td>20</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>7</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>24</td>
<td>17</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>18</td>
<td>3</td>
<td>12</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23</th>
<th>10</th>
<th>15</th>
<th>4</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5</td>
<td>24</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>1</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>20</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>21</td>
<td>12</td>
<td>7</td>
<td>2</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>16</th>
<th>7</th>
<th>26</th>
<th>11</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>25</td>
<td>12</td>
<td>15</td>
<td>6</td>
<td>27</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>33</td>
<td>8</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>35</td>
<td>24</td>
<td>21</td>
<td>28</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>18</td>
<td>3</td>
<td>30</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>36</td>
<td>31</td>
<td>22</td>
<td>19</td>
<td>4</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 3.1 Three Knights' Tours.
What abstractions can now be made from this example? Which patterns does it exhibit that is typical for this kind of “problem-solving” algorithm? What does it teach us? The characteristic feature is that steps toward total solution are attempted and recorded that may later be taken back erased in the recordings when it is discovered that the step does not lead to the total solution, that the step leads into a “dead-end street.” This action is called backtracking. The general pattern (3.28) is derived from (3.24), assuming that the number of potential candidates in each step is finite.

procedure try;
  begin initialize selection of candidates;
    repeat select next;
      if acceptable then
        begin record it;
          if solution incomplete then
            begin try next step;
              if not successful then cancel recording
            end
        end
      until successful ∨ no more candidates
  end

Actual programs may, of course, assume various derivative forms. A frequently encountered pattern uses an explicit level parameter indicating the depth of recursion and allowing for a simple termination condition.

If, moreover, at each step the number of candidates to be investigated is fixed, say $m$, then the derived schema (3.29) applies; it is to be invoked the statement “try(1).

procedure try(i: integer);
  var k: integer;
  begin k := 0;
    repeat k := k + 1; select k-th candidate;
      if acceptable then
        begin record it;
          if $i < n$ then
            begin try$(i+1)$;
              if not successful then cancel recording
            end
        end
    until successful ∨ (k = m)
  end
The remainder of this chapter is devoted to the treatment of three more examples. They display various incarnations of the abstract schema (3.29) and are included as further illustrations of the appropriate use of recursion.

3.5. THE EIGHT QUEENS PROBLEM

The problem of the eight queens is a well-known example of the use of trial-and-error methods and of backtracking algorithms. It was investigated by C. F. Gauss in 1850, but he did not completely solve it. This should not surprise anyone. After all, the characteristic property of these problems is that they defy analytic solution. Instead, they require large amounts of exacting labor, patience, and accuracy. Such algorithms have therefore gained relevance almost exclusively through the automatic computer, which possesses these properties to a much higher degree than people, and even geniuses, do.

The eight queens problem is stated as follows (see also [3-4]): Eight queens are to be placed on a chess board in such a way that no queen checks against any other queen.

Using the schema of Eq. (3.29) as a template, we readily obtain the following crude version of a solution:

\[
\text{procedure } \text{try}(i: \text{ integer}); \\
\text{begin} \\
\text{initialize selection of positions for } i\text{-th queen}; \\
\text{repeat make next selection;}
\quad \text{if } \text{safe} \text{ then} \\
\quad \text{begin setqueen;}
\quad \quad \text{if } i < 8 \text{ then} \\
\quad \quad \quad \begin{array}{c}
\text{begin } \text{try}(i+1) ; \\
\text{if } \text{not successful } \text{then remove queen}
\end{array} \\
\quad \text{end}
\end{array} \\
\text{until } \text{successful } \lor \text{ no more positions}
\text{end}
\]

In order to proceed, it is necessary to make some commitments concerning the data representation. Since we know from the rules of chess that a queen checks all other figures lying in either the same column, row, or diagonals on the board, we infer that each column may contain one and only one queen, and that the choice of a position for the \(i\)th queen may be restricted to the \(i\)th column. The parameter \(i\) therefore becomes the column index, and the selection process for positions then ranges over the eight possible values for a row index \(j\).

There remains the question of representing the eight queens on the board. An obvious choice would again be a square matrix to represent the board,