

```

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
end .

```

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 is possible. Let us therefore define an algorithm trying to perform a next move. A first approach is shown in (3.24).

```

procedure try next move;
begin initialize selection of moves;
  repeat select next candidate from list of next moves;
    if acceptable then
      begin record move;
        if board not full then
          begin try next move;
            if not successful then erase previous recording
          end
        end
      end
    until (move was successful)  $\vee$  (no more candidates)
  end

```

(3.24)

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:

```

type index = 1 ..  $n$ ;
var  $h$ : 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:

```

 $h[x, y] = 0$ : field  $\langle x, y \rangle$  has not been visited
 $h[x, y] = i$ : field  $\langle x, y \rangle$  has been visited in the  $i$ th move
                ( $1 \leq i \leq n^2$ )

```

(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 " $i < n^2$." 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 $q1$ is introduced and used as the result parameter in the recursive call of this algorithm, then $q1$ may be substituted for "move was successful." Thereby we arrive at the formulation shown in (3.27).

```

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 ( $1 \leq u \leq n$ )  $\wedge$  ( $1 \leq v \leq n$ )  $\wedge$  ( $h[u,v]=0$ ) then
        begin  $h[u,v] := i$ ;
          if  $i < \text{sqr}(n)$  then
            begin try( $i+1, u, v, q1$ );
              if  $\neg q1$  then  $h[u,v] := 0$ 
            end else  $q1 := \text{true}$ 
          end
        until  $q1 \vee$  (no more candidates);
       $q := q1$ 
end
  
```

(3.27)

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 $\langle x, y \rangle$, there are eight potential candidates for coordinates $\langle u, v \rangle$ 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

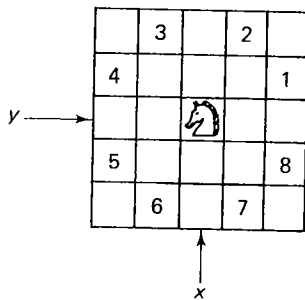


Fig. 3.8 The eight possible moves of a knight.

