Acta Cybernetica, Vol. 12, No. 1, Szeged, 1995

Demonstration of a Problem-Solving Method*

Judit Nyéky-Gaizler † Márta Konczné-Nagy †
Ákos Fóthi † Éva Harangozó †

Abstract

A program for backtrack seeking is proved here by using deduction rules. The problem of whether a chessboard can be moved over by the knight stepping on every square once and only once, is studied, and is traced back to the theorem of backtrack seeking in two ways. A comparison is made between the programs obtained.

1 Introduction

The last forty years have seen a rapid development in programming. Initially the hardware developed more rapidly than the software technology. For a long time the effectiveness of the programs had been the most important factor in programming, *but the importance of the reliability* of the programs became underlined by the improving quality of hardware tools and by the demand for producing increasingly larger systems.

The first works of Floyd, Hoare, Dijkstra and others [2,1,11] on proving program correctness were published in the 70s; work in this field was continued by Gries, Mili, Jackson, Wirth, etc [7,8,9,10,13]. Parallel with the theoretical research the results were translated into practice.

To prove the correctness of existing programs is only one possibility. A better approach is: write correct programs. Several programming theorems are proven [3,8] etc. for solving classes of important problems. In the present paper a program for the general problem of backtrack seeking is proved by using deduction rules. The problem of whether a chessboard can be moved over by the knight stepping on every square once and only once, is studied, and is traced back to the theorem of backtrack seeking in two ways. A comparison is also made between the programs obtained.

The most important definitions and theorems that are necessary to understand the present paper are available in the literature [3,4,5,6].

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. 2045

†Dept. of General Computer Science, Eötvös Loránd University, Budapest, Hungary, 1088 Budapest, Múzeum krt. 6-8., E-mail: nyeky@ludens.elte.hu

2 Theorem of Backtrack Seeking

2.1 The problem

Let $\mathbf{U}_1, \dots, \mathbf{U}_n$ be finite sets. Denote by α_i the number of elements of $\mathbf{U}_i$:

$$|\mathbf{U}_i| = \alpha_i \qquad \forall i \in [1, n].$$

Let $\mathbf{U}$ denote

$$\mathbf{U} = \mathop{\times}_{i=1}^{n} \mathbf{U}_i.$$

Let $\rho : \mathbf{U} \to \mathbf{L}$ be a logical function having the following properties: there exists a $\rho_i (i \in [0, n])$ sequence of logical functions, for which:

1. $\rho_0 = TRUE$
2. $\rho_{i+1}(u) \Rightarrow \rho_i(u) \qquad \forall i \in [1, n-1]$
3. $\forall j \in [1, i] : u_j = v_j \Rightarrow \rho_i(u) = \rho_i(v)$
   this means that ρ_i depends only on the first i component of u.
4. $\rho_n = \rho$

The problem is to decide whether there exists a $u \in \mathbf{U}$, for which $\rho(u)$ is true. If yes, let $u \in \mathbf{U}$ with the property ρ being given.

2.2 The specification of the problem

Let

$$\boldsymbol{N} = \mathop{\times}_{i=1}^{n} \mathbf{V}_i, \qquad \mathbf{V}_i = [0, \alpha_i - 1] \subset \mathrm{N}_0 \qquad \forall i \in [1, n].$$

In this case: $|\boldsymbol{N}| = |\mathbf{U}|$.

$\mathbf{U}_i$ can be ordered from 0 to $(\alpha_i - 1)$, $\forall i \in [1, n]$.

Denote by $u_{i_j} \in \mathbf{U}_i$ the jth element of $\mathbf{U}_i$.

Let ϕ denote a function, which is a bijection between $\boldsymbol{N}$ and $\mathbf{U}$: $\phi : \boldsymbol{N} \to \mathbf{U}$, and if $\nu \in \boldsymbol{N}$, then: $\phi(\nu) = (u_{1_{\nu_1}}, \dots, u_{n_{\nu_n}})$.

We can consider the elements of $\boldsymbol{N}$ as numbers encoded in a mixed radix number system [12]. Therefore we have defined an ordering on $\boldsymbol{N}$, and can speak about the "follower" of an element.

Let us denote by $f(\nu)$ the value of $\nu \in \boldsymbol{N}$ in the decimal system, that is

$$f(\nu) = \sum_{i=1}^{n} (\nu_i * \prod_{j=i+1}^{n} \alpha_j).$$

If $\nu', \nu'' \in \boldsymbol{N}$, then we shall consider $\nu' < \nu''$ iff $f(\nu') < f(\nu'')$.

Denote by ϵ_0 the zero value of $\boldsymbol{N}$: $\epsilon_0 = (0, 0, \dots, 0)$ and by ϵ_n the unit value of $\boldsymbol{N}$: $\epsilon_n = (0, 0, \dots, 1)$. Moreover $\forall i \in [1, n-1], let \epsilon_i = (0, 0, \dots, 1, \dots, 0) \in \boldsymbol{N}$, such that:

$$f(\epsilon_i) = \prod_{j=i+1}^{n} \alpha_j.$$

With the help of these we can write the specification of the problem. Let the state space be $\boldsymbol{N} \times \mathbf{L}$, and its variables ν and l.

$$\begin{array}{ll} A: & \boldsymbol{N} \times \mathbf{L} \\ & \nu \quad l \end{array}$$

The precondition of the problem is
$Q : TRUE$;
the postcondition of the problem is
$R : l = (\exists\, \nu' \in \boldsymbol{N} : \rho(\phi(\nu'))) \wedge l \Rightarrow (\rho(\phi(\nu)))$.

2.3 Solution of the problem

Disregarding the special features of ρ, the original problem can be solved by the theorem of the third variation of linear seeking [3] in the interval $[1, |\, \boldsymbol{N} \,|\,]$, for the property ρ, and with the stopping-condition $f(\nu) \geq |\, \boldsymbol{N} \,| -1$.

We can increment the values in the mixed radix number system by the unit value ϵ_n .

Let ϵ_0 be the initial value of ν, to avoid the problems coming from the use of negative values in the mixed radix number system; ϵ_0 being the first $\nu \in \boldsymbol{N}$ to be interpreted.

The result is given by

```
program {1}:
    $l, v, \nu := \rho(\phi(\nu)), \mathbf{false}, \epsilon_0$
    while $\neg l \wedge \neg v$ loop
        $\nu := \nu \oplus \epsilon_n$
        $l := \rho(\phi(\nu))$
        $v := f(\nu) \geq |\, \boldsymbol{N} \,| -1$
    endloop
end
```

We can significantly increase the effectiveness of the algorithm by using the special features of ρ, namely, that if $\rho_i(\phi(\nu)) = \mathbf{true}$ and $\rho_{i+1}(\phi(\nu)) = \mathbf{false}$, then for every $\nu' \in \boldsymbol{N}$, which satisfies $\forall j \in [1, i+1] : \nu_j = \nu'_j$, then $\rho_{i+1}(\phi(\nu'))$ will also be false because of the third property of ρ.

So instead of $\nu \oplus \epsilon_n$ the next possible $\nu \in \boldsymbol{N}$ will be the value $\nu \oplus \epsilon_{i+1}$.

The counting algorithm will be more simple if we amplify ν with an overflow bit - denoted by c. If the overflow bit changes to 1, it means that we no longer have the possibility to change ν.

From these it follows that it is worth supplementing the assignment $l := \rho(\phi(\nu))$ by seeking the smallest index for which $\rho_i(\phi(\nu)) = \mathbf{false}$.

Using the rules of deduction [3] the following program can be achieved.

As the invariant of the loop let us use: $P : (\forall \nu' : (0 \leq f(\nu') < f(\nu) + c * \;|\, \boldsymbol{N} \,|: \neg\rho(\phi(\nu') \wedge l = \rho(\phi(\nu)) \wedge \neg l \Rightarrow (\rho_{m-1}(\phi(\nu)) \wedge \neg\rho_m(\phi(\nu))) \wedge (\forall i \in [m+1, n] : \nu_i = 0))$

If $\pi = \neg l \wedge (c = 0)$ is considered as the condition of the loop, then $P \wedge \neg\pi \Rightarrow R$ is really completed.

Let the terminator function be $t = |\, \boldsymbol{N} \,| - f(\nu) - c * \;|\, \boldsymbol{N} \,|$. Evidently $t > 0$, while $P \wedge \pi$ is true. The function t will be decreased by increasing ν.

To perform the condition $\mathbf{Q} \Rightarrow \mathbf{P}$ we need adequate initial values for the variables before starting the loop.
The **Backtrack** program will be:

```
program {2}:
    $\nu, c, m := \epsilon_0, 0, 1$
    SEEK($\nu, m, l$)                $\leftarrow \mathbf{Q}'$
    while $\neg l \wedge (c = 0)$ loop
      SUM($\nu, m, c$)
      SEEK($\nu, m, l$)
    endloop
end
```

To verify $\mathbf{Q}' \Rightarrow \mathbf{P}$ let us define the **SEEK** program. Let the specification of the **SEEK**(ν, m, l) program be:

$$A_{\mathrm{SEEK}} : \begin{array}{ccccc} \boldsymbol{N} & \times & \mathrm{N}_0 & \times & \mathbf{L} \\ \nu & & m & & l \end{array}$$

$$B_{\mathrm{SEEK}} : \begin{array}{ccc} \boldsymbol{N} & \times & \mathrm{N}_0 \\ \nu' & & m' \end{array}$$

$$Q_{\mathrm{SEEK}} : \nu = \nu' \wedge m = m' \wedge \rho_{m'-1}(\phi(\nu))$$

$$R_{\mathrm{SEEK}} : \nu = \nu' \wedge l = (\forall i \in [m', n] : \rho_i(\phi(\nu))) \wedge$$
$$\neg l \Rightarrow (\neg \rho_m(\phi(\nu)) \wedge \forall i \in [m', m) : \rho_i(\phi(\nu))) \wedge \rho_{m'-1}(\phi(\nu)))$$

This problem can also be solved by the theorem of the third variation of linear seeking [3] bearing in mind that the following two statements are equivalent: (1) every element of a set has a certain property, (2) there is no a single element in the set without this property.

Thus, the **SEEK** program will be

```
program {3}:
    $l, m :=$ true, $m - 1$
    while $l \wedge (m \neq n)$ loop
      $l := \rho_{m+1}(\phi(\nu))$
      $m := m + 1$
    endloop
end
```

Therefore in the main program there will be $\mathbf{Q}' = \mathbf{R}_{\mathbf{SEEK}} \wedge (\nu = \epsilon_0) \wedge (c = 0)$ and $\mathbf{Q}' \Rightarrow \mathbf{P}$ simply follows .

To prove the implication $\mathbf{P} \wedge \pi \Rightarrow wp(S_0, \mathbf{P})$ we need determine the **SUM** program as well. Let the specification of the **SUM**(ν, m, c) program be:

$$A_{\mathrm{SUM}} : \begin{array}{ccccc} \boldsymbol{N} & \times & \mathrm{N}_0 & \times & E \\ \nu & & m & & c \end{array}$$

$$B_{\mathrm{SUM}} : \begin{array}{ccc} \boldsymbol{N} & \times & \mathrm{N}_0 \\ \nu' & & m' \end{array}$$

$$Q_{\mathrm{SUM}} : \nu = \nu' \wedge m = m'$$

$$R_{\mathrm{SUM}} : (f(\nu) + c * \mid \boldsymbol{N} \mid = f(\nu') + \textstyle\prod_{i=m'+1}^{n} \alpha_i) \wedge m \in [0, m'] \wedge$$
$$(\forall i \in [m+1, m'] : \nu_i = 0) \wedge (c = 0 \Rightarrow \nu_m \neq 0).$$

As the invariant of the loop let us use

$$P_{\mathrm{SUM}}: (f(\nu) + c * \prod_{i=m+1}^{n} \alpha_i = f(\nu') + \prod_{i=m'+1}^{n} \alpha_i) \wedge m \in [0, m'] \wedge \\ (\forall i \in [m+1, m'] : \nu_i = 0) \wedge (c = 0 \Rightarrow \nu_m \neq 0).$$

Let us consider $\pi_{\mathrm{SUM}} = (m \neq 0) \wedge (c \neq 0)$ as the condition of the loop.

In this case $\mathbf{P}_{\mathrm{SUM}} \wedge \neg\pi_{\mathrm{SUM}} \Rightarrow \mathbf{R}_{\mathrm{SUM}}$.

Let the terminator function be: $t_{\mathrm{SUM}} = m + c$.

Evidently $t_{\mathrm{SUM}} > 0$ while $P_{\mathrm{SUM}} \wedge \pi_{\mathrm{SUM}}$ is true.

The function t_{SUM} will be decreased by increasing either m or c. Thus the program will be

```
program {4}:
   $c := 1$                                   $\leftarrow Q'_{\mathrm{SUM}}$
   while $(m \neq 0) \wedge (c \neq 0)$ loop
     if $\nu_m = \alpha_m - 1$ then $\nu_m := 0$
                                  $m := m - 1$
                          else $c := 0$
                                  $\nu_m := \nu_m + 1$
     endif
   endloop
end
```

In this case $Q'_{\mathrm{SUM}} : (\nu = \nu') \wedge (m = m') \wedge (c = 1)$. Therefore $Q'_{\mathrm{SUM}} \Rightarrow \mathbf{P}_{\mathrm{SUM}}$.

To verify $\mathbf{P}_{\mathrm{SUM}} \wedge \pi \Rightarrow \mathrm{wp}(\mathrm{S0}_{\mathrm{SUM}}, \mathbf{P}_{\mathrm{SUM}})$ we have to prove:

1. $\mathbf{P}_{\mathrm{SUM}} \wedge \pi \wedge (\nu_m = \alpha_m - 1) \Rightarrow \mathrm{wp}((\nu_m := 0; m := m - 1), \mathbf{P}_{\mathrm{SUM}})$
2. $\mathbf{P}_{\mathrm{SUM}} \wedge \pi \wedge (\nu_m \neq \alpha_m - 1) \Rightarrow \mathrm{wp}((c := 0; \nu_m := \nu_m + 1), \mathbf{P}_{\mathrm{SUM}})$

These are consequences of the definition of the function $f(\nu)$ using the weakest precondition of the assignment statement.

Having proved the SUM program for the verification of the main program we need $\mathbf{P} \wedge \pi \Rightarrow \mathrm{wp}((\mathrm{SUM}; \mathrm{SEEK}), \mathbf{P})$ and this follows from the above.

3 Solution of a demonstration problem

3.1 The problem

The 8×8 $(n \times n)$ chessboard is given. We have to decide whether it is possible for the knight to move over the whole chessboard stepping on each square once and only once. If it is possible, we should be able to give a "tour".

Two possible solutions of this problem will be given and compared below.

3.1.1 Specification of the first solution

Since we have to step on 64 (n^2) squares, we can use a vector of 64 (n^2) length for the storage of the knight's moves. The j^{th} component of the vector denotes the position of the j^{th} step. Let us number each square of the table line by line from 0 to 63 $(0-(n^2-1))$:

$$A: \underset{\nu}{\boldsymbol{N}} \times \underset{l}{L} \qquad \boldsymbol{N} = \overset{64}{\underset{i=1}{\times}} [0,63]$$

The precondition of the problem is
$Q: TRUE$;
the postcondition of the problem is
$R: l = (\exists\, \nu' \in \boldsymbol{N} : \rho(\nu')) \wedge l \Rightarrow (\rho(\nu))$.

Let us denote by $LIN_i = \nu_i/8$;and by $COL_i = \nu_i - 8 * LIN_i$, (in general: $LIN_i = \nu_i/n$; and by $COL_i = \nu_i - n * LIN_i$), where the fraction bar denotes the division between integers.

Let $\rho: \boldsymbol{N} \rightarrow \mathbf{L}$ be a logical function to be defined as follows: Let $\rho_i(i \in [0, n^2])$ be a sequence of logical functions satisfying

1. $\rho_0 = \rho_1 = TRUE$
2. $\rho_i(\nu) = \rho_{i-1}(\nu) \wedge \mu_i(\nu) \wedge \gamma_i(\nu) \qquad \forall i \in [2, 64]$

$$\mu_i(\nu) = \nu_i \text{ knight-move-distance from } \nu_{i-1} =$$
$$= (|LIN_i - LIN_{i-1}| = 2 \wedge |COL_i - COL_{i-1}| = 1) \vee$$
$$(|LIN_i - LIN_{i-1}| = 1 \wedge |COL_i - COL_{i-1}| = 2).$$
$$\gamma_i(\nu) = \nu_i \text{ different from the squares over}$$
$$= (\forall i : 1 \leq j < i : \nu_j \neq \nu_i)$$

In this case $\rho_i(\nu) \Rightarrow \rho_{i-1}(\nu)$, and obviously:

3.$\forall j \in [1, i] : \nu_j = \nu'_j \Rightarrow \rho_i(\nu) = \rho_i(\nu')$; that is, ρ_i depends on the first i component of $\boldsymbol{N}$ only.

4. $\rho_{64} = \rho$

3.1.2 The first solution of the problem

It can be seen that this specification is equivalent to the specification of the general Backtrack seeking algorithm, therefore the program for solving it can be used with the following way of correspondence:

$n = 64$

$\forall i \in [1, 64] : \mathbf{U}_i = \{0, 1, \ldots, 63\}, \quad \alpha_i = 64$

(We shall use a 64 based number system instead of the general mixed radix number system.)

ϕ is the identical mapping.

The program is as follows

```
program {5}:
  $\nu, c, m := \epsilon_0, 0, 1$
  SEEK($\nu, m, l$)
  while $\neg l \wedge (c = 0)$ loop
    SUM($\nu, m, c$)
    SEEK($\nu, m, l$)
  endloop
end
```

The **SEEK** program is given by

```
program {6}:
  $l, m :=$ true$, m - 1$
  while $l \wedge (m \neq 63)$ loop
    $l := \rho_{m+1}(\nu)$
    $m := m + 1$
  endloop
end
```

The **SUM** program will be

```
program{7}:
  $c := 1$
  while $(m \neq 0) \wedge (c \neq 0)$ loop
    if $\nu_m = 63$ then $\nu_m := 0$
                         $m := m - 1$
                   else $c := 0$
                         $\nu_m := \nu_m + 1$
    endif
  endloop
end
```

We now need the program for the assignment statement $l := \rho_{m+1}(\nu)$ only. As we have defined $\rho_{m+1}(\nu) = \rho_m(\nu) \wedge \mu_{m+1}(\nu) \wedge \gamma_{m+1}(\nu)$, consequently the precondition of this program is

$Q : ((l = \rho_m(\nu)) \wedge (\nu = \nu') \wedge l');$

and the postcondition is

$R : (l = \rho_{m+1}(\nu) \wedge (\nu = \nu'))$.

In the state space A this is equivalent to

$R : (l_1 = \mu_{m+1}(\nu) \wedge l_2 = \gamma_{m+1}(\nu) \wedge l = (l' \wedge l_1 \wedge l_2) \wedge (\nu = \nu'))$.

The program realizing this condition is the sequence of states below

```
program {8}:
  $l_1 := \mu_{m+1}(\nu)$
  $l_2 := \gamma_{m+1}(\nu)$
  $l := l_1 \wedge l_2$
end
```

The solution of the assignment $l := \mu_{m+1}(\nu)$ will be:

$l_1 = (| LIN_{m+1} - LIN_m |= 2 \wedge \; | COL_{m+1} - COL_m |= 1) \vee$
$(| LIN_{m+1} - LIN_m |= 1 \wedge \; | COL_{m+1} - COL_m |= 2)$.

The assignment $l_2 := \gamma_{m+1}(\nu)$ can be solved by the theorem of the third variation of linear seeking, with the considerations written under the **SEEK** program. The program $l_2 := \gamma_{m+1}(\nu)$ will be

```
program {9}:
    $l_2, i$ := true, 0
    while $l_2 \wedge (i \neq m)$ loop
        $l_2 := \nu_{m+1} \neq \nu_{i+1}$
        $i := i + 1$
    endloop
end
```

This completes the first solution.

3.1.3 Specification of the second solution

The main idea of the second solution is to take advantage of the fact that we cannot step anywhere from a certain square of the chessboard. We can choose only from the eight possible moves of the knight. All the moves are represented by a vector showing the relative movement of the knight by two components, the first for the horizontal (lines) direction, the second for the vertical (columns) direction.

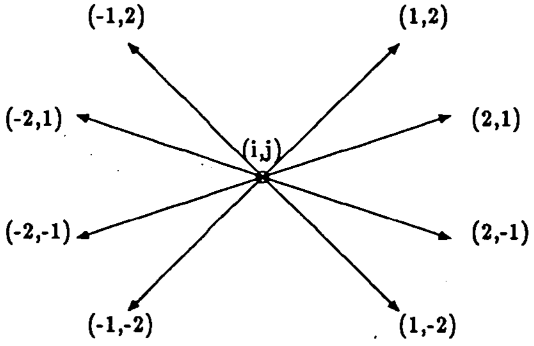

Figure 1: The knight moves

Let us consider these steps as the components of a constant vector called "knight-move-vector":

$$h = ((1,2),(2,1),(2,-1),(1,-2),(-1,-2),(-2,-1),(-2,1),(-1,2))$$

We use the **Backtrack** algorithm again with the following correspondence:

n = 64

$\mathbf{U}_i = \{(i,j) \mid 0 \leq i,j \leq 7\}$, give an arrangement with the enumeration of the elements:

$\mathbf{U}_i = \{(0,0),(0,1),(0,2),\dots,(7,0),(7,1),\dots,(7,7),\}$

If we represent the chess-board by a matrix, the elements of $\mathbf{U}_i$ will be the values of the possible start positions.

$\alpha_1 = 64$

Let H denote the eight-element set obtained with the help of the knight-move-vector h, we define the arrangement on H with the enumeration in h.

$\forall i \in [2,64]\ \mathbf{U}_i = H$ $\qquad$ $\mid \alpha_i \mid = 8$ $\qquad$ $\mathbf{U} = \overset{64}{\underset{i=1}{\times}} \mathbf{U}_i$

Using the sets $\mathbf{U}_i$ the actual knight move sequence on the chess-board can be given by the function:

pos: $\mathbf{U} \to \mathbf{V}$ $\quad \mathbf{V} = \overset{64}{\underset{i=1}{\times}} P \quad \mathbf{P} = (LIN, COL) \quad LIN, COL = \mathbf{N}_0$

$v_1 = pos(u)_1 = u_1$

$v_i = pos(u)_i = pos(u)_{i-1} \oplus u_i \qquad \forall i \in [2, 64]$

$\oplus$ denotes the addition component by component.

The correspondence between $\boldsymbol{N}$ and $\mathbf{U}$ is given by

$\boldsymbol{N} = [0, 63] \times (\overset{63}{\underset{i=1}{\times}} [0, 7])$

$\phi : \boldsymbol{N} \to U$

$u_1 = \phi(\nu)_1 = (\nu_1/8, \nu_1 - \nu_1/8 * 8)$ (the fraction bar denotes the division between integers)

$u_i = \phi(\nu)_i = h_{\nu_i} \qquad \forall i \in [2, 64]$.

The specification with the values above is

$$A: \begin{array}{cc} \boldsymbol{N} & \times \mathbf{L} \\ \nu & l \end{array}$$

$Q : TRUE$

$R : l = (\exists\ \nu' \in \boldsymbol{N} : \rho(\phi(\nu'))) \land l \Rightarrow (\rho(\phi(\nu)))$.

Let $\rho : \boldsymbol{N} \to \mathbf{L}$ be a logical function to be defined as follows: let $\rho_i(i \in [1, 64])$ be a sequence of logical functions satisfying

1. $\rho_1 = TRUE$
2. $\rho_i(\phi(\nu)) = \rho_{i-1}(\phi(\nu)) \land \mu_i(\phi(\nu)) \land \gamma_i(\phi(\nu)) \qquad \forall i \in [2..64]$

where $\mu_i(\phi(\nu))$ = the i^{th} move does not move off the chess-board
$= pos(\phi(\nu))_i \in [0, 7] \times [0, 7]$

and $\gamma_i(\phi(\nu)) = pos(\phi(\nu))_i$ different from the squares over
$= (\forall j : 1 \leq j < i : pos(\phi(\nu))_j \neq pos(\phi(\nu))_i)$

In this case $\rho_i(\phi(\nu)) \Rightarrow \rho_{i-1}(\phi(\nu))$, and obviously:

3. $\forall j \in [1, i] : \nu_j = \nu'_j \Rightarrow \rho_i(\phi(\nu)) = \rho_i(\phi(\nu'))$; that is, ρ_i depends on the first i component of N only.
4. $\rho_{64} = \rho$.

3.1.4 The second solution of the problem

It can be observed, that the function pos given by a recursive formula in the program working out the assignment $l := \rho_{m+1}(\phi(\nu))$ can be substituted by a variable, and thus its evaluation will be significantly simpler. Let us amplify the state space with a component of type $\mathbf{V}$, denoted its variable by$v$.

The first m element of v shows then the sequence of actual positions of the knight.

The program obtained is

```
program{10}:
  ν, c, m := ε₀, 0, 1
  v₁ := (0, 0)                  ← startposition
  SEEK(ν, m, l)
  while ¬l ∧ (c = 0) loop
    SUM(ν, m, c)
    SEEK($\nu, m, l$)
  endloop
end
```

Since the **SEEK** program differs from that written in the first solution in realizing the assignment $l := \rho_{m+1}(\phi(\nu))$ only, here we give just the difference.

The **SUM** program is changed in comparison with the first solution:

```
program{11}:
  $c := 1$
  while $(m \neq 0) \wedge (c \neq 0)$ loop
    if $(m = 1 \wedge \nu_m = 63) \vee (m \neq 1 \wedge \nu_m = 7)$ then $\nu_m := 0$
                                                      $m := m - 1$
                                                 else $c := 0$
                                                      $\nu_m := \nu_m + 1$
    endif
  endloop
end
```

Let us give the program solving $l := \rho_{m+1}(\nu)$!

Since $\rho_{m+1}(\phi(n)) = \rho_m(\phi(\nu)) \wedge \mu_{m+1}(\phi(\nu)) \wedge \gamma_{m+1}(\phi(\nu))$; thus, the precondition of the program is

$Q : ((l' = \rho_m(\phi(\nu'))) \wedge (\nu = \nu') \wedge l')$;

the postcondition is

$R : (l = \rho_{m+1}(\phi(\nu)) \wedge (\nu = \nu'))$.

This is equivalent in the state space A with

$R : (l_1 = \mu_{m+1}(\phi(\nu)) \wedge l_2 = \gamma_{m+1}(\phi(\nu)) \wedge l = (l_1 \wedge l_2) \wedge (\nu = \nu'))$.

The program realizing this condition is the sequence of statements below.

```
program {12}:
  $v_{m+1} := v_m \oplus h_{\nu_{m+1}}$
  $l_1 := \mu_{m+1}(v_{m+1})$
  $l_2 := \gamma_{m+1}(v)$
  $l := l_1 \wedge l_2$
end
```

The solution of the assignment $l_1 := \mu_{m+1}(v_{m+1})$ will be

$$l_1 := (0 \leq (v_{m+1})_1 \leq 7) \wedge (0 \leq (v_{m+1})_2 \leq 7)$$

The assignment $l_2 := \gamma_{m+1}(v)$ can be solved by the theorem of the third variation of linear seeking, with the considerations written under the **SEEK**program.

Thus the program $l_2 := \gamma_{m+1}(v)$ is

```
program {13}:
  $l_2, i :=$ true, 0
  while $l_2 \wedge (i \neq m)$ loop
    $l_2 := v_{m+1} \neq v_{i+1}$
    $i := i + 1$
    endloop
end
```

This program is essentially the same as the corresponding one in the first solution. Thus the second solution is completed.

3.2 Comparison of the two solutions

If the two solutions are compared from the viewpoint of execution time, the second one is found to be essentially faster. The reason for this lies in the number of potential attempts at the first solution

$$|\boldsymbol{N}| = \prod_{i=1}^{n^2} n^2.$$

The corresponding value at the second solution is smaller by an order of magnitude:

$$|\boldsymbol{N}| = \prod_{i=1}^{n^2} n.$$

The above example indicates that one should never automatically trace the problems back to the various programming theorems, since the innovational way of thinking of the expert programmer is essential.

References

[1] Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Strukturált programozás (Structured programming), Műszaki Könyvkiadó, Budapest, 1978.

[2] Dijkstra, E.W.: A Discipline of Programming, Englewood Cliffs, 1976, Prentice-Hall Series in Automatic Computation.

[3] Fóthi Á.: Introduction into Programming (Bevezetés a programozáshoz), in Hungarian, manuscript, ELTE TTK, Budapest, 1983.

[4] Fóthi Á.: A Mathematical Approach to Programming, *Annales Uni. Sci. Budapest. de R. Eötvös Nom. Sectio Computatorica,* Tom. IX. (1988), 105-114.

[5] Fóthi Á., Horváth Z.: The Weakest Precondition and the Theorem of the Specification, in Proceedings of the Second Symposium on Programming Languages and Software Tools, Pirkkala, Finland, August 21-23, 1991, Eds.: Kai Koskimies and Kari-Jouko Räihä, Uni. of Tampere, Dep. of Comp. Sci. Report A-1991-5.

[6] Horváth Z.: Parallel asynchronous computation of the values of an associative function, *Acta Cybernetica.* 12 (1995) 83-94.

[7] Gries, D.: The Science of Programming, Springer Verlag, Berlin, 1981.

[8] Jackson, M.A.: Principles of Programming Design, Academic Press, New York, 1975.

[9] Mili, A.: A Relational Approach to the Design Deterministic Programs, *Acta Informatica,* 20 (1983) 315-328.

[10] Mili, A., Desharnais, J., Gagné, J.R.: Formal Models of Stepwise Refinement of Programs, *ACM Computing Surveys,* 18 (1986) 231-276.

[11] Mills, H.D.: The New Math of Computer Programming, *Comm. of the ACM* 18 (1975) 43-48.

[12] Szabo, M., S. Nicholas, Tanaka, I. Richard: Residue Arithmetic and its Applications to Computer Technology, McGraw-Hill Book Company, New Y. - San Francisco - Toronto, 1967.

[13] Wirth, N.: Systematic programming. An Introduction., Prentice-Hall Inc. 1973.

Received April, 1994