Nat.Lab. Unclassified Report ???/00

Date of issue: 06/00

The Puzzle Processor Project

Towards an Implementation

Erik van der Tol and Tom Verhoeff

Authors' address data: E. B. v.d. Tol; erik.van.der.tol@philips.com
T. Verhoeff; T.Verhoeff@TUE.NL

Unclassified Report: ???/00

Title: The Puzzle Processor Project
Towards an Implementation

Author(s): Erik van der Tol and Tom Verhoeff

Part of project: Puzzle Processor Project

Customer: TUE

Keywords: Packing Puzzles; Set Partitioning; Backtracking; Processor Design; VLSI Programming

Abstract: The Puzzle Processor Project seeks to develop a special-purpose processor for efficiently solving certains kinds of puzzles. The puzzles are packing problems where a collection of pieces and a box are given with the goal to fit the pieces into the box. Packing problems appear both in recreational and in more serious settings.

First, we reformulate these packing problems in terms of set partitioning. Next, we derive an instruction set for the puzzle processor by transforming a backtrack program for set partitioning. Finally, we present and analyze a design for the puzzle processor expressed in Tangram, a VLSI programming language developed at Philips Research Laboratories.

Conclusions: We have specified and designed a puzzle processor. It has five instructions acting on four registers. A packing puzzle can be compiled into a dedicated program for this processor. When executed, the program determines solutions to the packing problem.

Contents

1 Introduction

We are interested in solving puzzles consisting of a collection of pieces that have to be placed in a box. Such puzzles are also known as packing problems. A well-known example is the 6×10 pentomino puzzle [2], shown in Figure 1. The pentomino puzzle consists

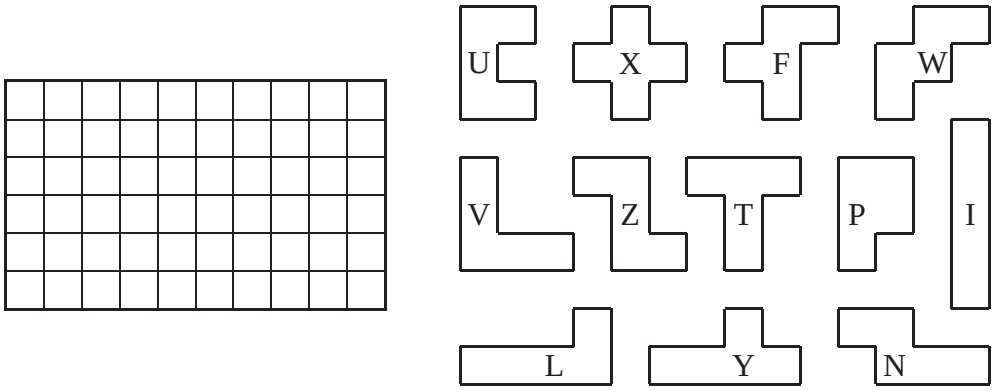

Figure 1: The 6×10 pentomino puzzle: box (left) and 12 pieces (right)

of a box of 6 by 10 unit squares (fields), in which the 12 pentominoes have to be placed. Each pentomino consists of a unique combination of 5 unit squares. The pentominoes may be translated, rotated, and reflected when placed in the box. Thus, there are many ways to place each piece in the box. Figure 2 shows one of the 9356 solutions[1] for the 6×10 pentomino puzzle.

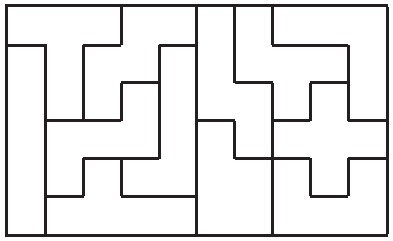

Figure 2: An elegant solution for the 6×10 pentomino puzzle

Algorithms for solving this kind of puzzles are usually based on backtracking [3]. Instead of a general-purpose backtrack program that takes a puzzle description as input, we will develop puzzle-specific backtrack programs. The resulting programs involve just a few data structures and operations, which serve as the basis for the specification of a special-purpose puzzle processor. The puzzle processor is optimized for dealing with the data structures and operations appearing in the puzzle-specific backtrack programs. To solve a puzzle, we generate a dedicated program for the puzzle processor and then execute it on that processor.

[1] 2339 modulo rotation and reflection.

2 Problem description

Let us look at a concrete example of a very simple puzzle. Figure 3 shows a 2×3 rectangular box and three pieces to be fit into the box. We will use this puzzle to illustrate our ideas.

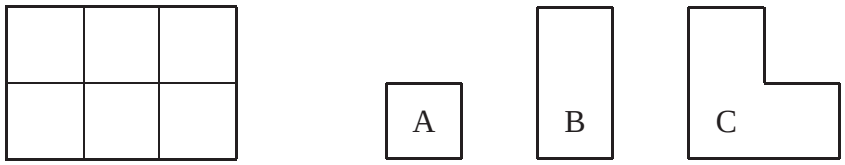

Figure 3: A simple puzzle: 2×3 box (left) and 3 pieces (right)

No doubt you have already found the twelve solutions of the puzzle in Figure 3. How did you do it? We want to develop a computer program that determines all solutions for such puzzles. There are several approaches possible, most of which distinghuish between the role of the fields in the box and the role of the pieces (see e.g. [1]). This will be discussed further in the next section. We will, however, take a more general approach, which we present in Section 2.2.

2.1 Fields and pieces

A systematic approach is required for determining *every* solution of a puzzle just once. When treating the fields in the box and the pieces as clearly distinct entitities, one can consider two backtrack strategies:

1. *Concentrate on the fields*, by covering them in some order. The 'next' empty field, for instance in 'reading order', is covered in all possible ways by one of the pieces. Note that a piece may be put in the box in various orientations. Each covering results in a partial solution, leaving a similar puzzle with a smaller box and fewer pieces. When all fields have been covered, a solution has been obtained. For example, the top-left field of the simple example puzzle (Fig. 3) can initially be covered in six ways (once by piece A, twice by B, and in three ways by C).

2. *Concentrate on the pieces*, by using them in some order. The 'next' unused piece, for instance in alphabetic order, is placed in all possible ways in the empty part of the box. Each such placement results in a partial solution, leaving a similar puzzle with a smaller box and fewer pieces. When all pieces have been used, a solution has been obtained. For example, piece B of the simple example puzzle (Fig. 3) can initially be used in seven ways (three vertical and four horizontal).

In the 'fields' strategy, the order in which the fields are attempted affects the running time of the program, while in the 'pieces' strategy, the order of the pieces is crucial. Tonneijk [5] has compared several backtrack strategies for solving puzzles in [5]. In general, the 'fields' strategy turns out to be faster than the 'pieces' strategy.

2.2 Generalization to aspects

Until now we have regarded the fields in the box and the pieces as two distinct entities. There is, however, a clear resemblance between them. A partial solution of a puzzle is captured by recording which fields have been covered and which pieces have been used. In our kind of puzzles, a field can be covered at most once, and a piece may be used also at most once. This can easily be tracked by a boolean variable for each field and for each piece.

Of course, one can imagine puzzles where fields and/or pieces may be covered or used more than once. In this more general case, the usage of both commodities can be tracked by a counter (natural number). We will not deal with this more general situation, but it is straightforward to adapt our treatment for such more general puzzles by using bags instead of sets.

...

2.3 Abstract puzzles

An **abstract puzzle** is a pair (A, E) of sets such that

$$E \subseteq \mathcal{P}(A) \ \wedge \ E \neq \emptyset \ \wedge \ \emptyset \notin E \tag{1}$$

A member of A is called an **aspect** of the puzzle, and a members of E is called an **embedding**. An embedding is a nonempty set of aspects. Observe that (1) implies $A \neq \emptyset$.

The puzzle from Figure 3 is expressed as an abstract puzzle by

$$\begin{array}{lllll}
(& \{0,1,2,3,4,5,A,B,C\} & & & \\
, & \{ \ \{0,A\} & , \{1,A\} & , \{2,A\} & \\
 & , \{3,A\} & , \{4,A\} & , \{5,A\} & \\
 & , \{0,1,B\} & , \{0,3,B\} & , \{1,2,B\} & \\
 & , \{1,4,B\} & , \{2,5,B\} & , \{3,4,B\} & \\
 & , \{4,5,B\} & , \{0,1,3,C\} & , \{0,1,4,C\} & \\
 & , \{0,3,4,C\} & , \{1,2,4,C\} & , \{1,2,5,C\} & \\
 & , \{1,3,4,C\} & , \{1,4,5,C\} & , \{2,4,5,C\} & \\
 & \} & & & \\
)
\end{array} \tag{2}$$

with 9 aspects and 21 embeddings (6 involve piece A, 7 involve B, and 8 involve C). The aspects are dummies, in the sense that systematic renaming of the aspects yields an isomorphic puzzle.

For set s of embeddings ($s \subseteq E$), we define

$$\bigcup s \ = \ (\bigcup e : e \in s : e) \tag{3}$$

Thus, $\bigcup s \subseteq A$ is the set of aspects covered by s.

A **solution** for puzzle (A, E) is a subset s of E that partitions A:

1. $s \subseteq E$
2. $(\forall e, e' : e \in s \ \wedge \ e' \in s \ \wedge \ e \neq e' : e \cap e' = \emptyset)$
3. $\bigcup s = A$

Condition 2 expresses that embeddings in a solution are pairwise disjoint (do not overlap), and condition 3 that every aspect is covered. For example,

$$\{\{0, A\}, \{3, 4, B\}, \{1, 2, 5, C\}\} \tag{4}$$

is a solution of the abstract puzzle 2 corresponding to Figure 3.

Let $S(A, E)$ be the set of all solutions of (A, E), that is,

$$S(A, E) \ = \ \{s \mid s \subseteq E \ \wedge \ s \text{ partitions } A\} \tag{5}$$

The notion of a solution can be generalized to that of a partial solution by dropping the third condition that A is *completely* covered. A **partial solution** for puzzle (A, E) is a subset p of E that partitions a *subset* of A:

1. $p \subseteq E$
2. $(\forall e, e' : e \in p \ \wedge \ e' \in p \ \wedge \ e \neq e' : e \cap e' = \emptyset)$

Let $PS(A, E)$ be the set of all partial solutions of (A, E), that is,

$$PS(A, E) \ = \ \{p \mid p \subseteq E \ \wedge \ p \text{ is overlapfree}\} \tag{6}$$

The set of partial solutions for puzzle (A, E) has some useful properties: First of all, PS indeed generalizes S:

$$PS(A, E) \supseteq S(A, E) \tag{7}$$

The empty set is a partial solution (useful to initialize the search for a solution):

$$\emptyset \in PS(A, E) \tag{8}$$

A partial solution p can be extended by an embedding $e \in E$ that does not overlap $\bigcup p$ (useful as a step towards a solution):

$$p \in PS(A, E) \ \wedge \ e \cap \bigcup p = \emptyset \ \equiv \ e \notin p \ \wedge \ p \cup \{e\} \in PS(A, E) \tag{9}$$

A partial solution that covers *all* of A is a solution (useful to terminate the search for a solution):

$$p \in PS(A, E) \ \wedge \ \bigcup p = A \ \equiv \ p \in S(A, E) \tag{10}$$

3 Solving abstract puzzles

Let (A, E) be an abstract puzzle. We are interested in procedure *Solve* that processes each solution once. It should satisfy

$$\{\ true\ \}\quad Solve\quad \{\ (\forall\, s : s \in S(A, E) : Solution(s) \text{ called once})\ \} \tag{11}$$

where *Solution* is a procedure that processes a solution, for example, printing them or just counting them.

The specification can be generalized by introducing parameter p, being a partial solution, and requiring that *Solve1*(p) processes once every solution that extends p:

$\{\ p \in PS(A, E)\ \}$
Solve1 (p)
$\{\ (\forall\, s : s \in S_p(A, E) : Solution(s) \text{ called once})\ \}$

where $S_p(A, E)$ denotes the set of solutions that extend partial solution p:

$$S_p(A, E) \quad = \quad \{\, s \mid s \in S(A, E) \ \wedge\ p \subseteq s \,\} \tag{12}$$

Taking $p := \emptyset$ yields the original specification (cf. (8)), because $\emptyset \subseteq s$ holds vacuously:

$$S_p(A, E) \quad = \quad S(A, E) \quad \text{if } p = \emptyset \tag{13}$$

If $\bigcup p = A$ then p is actually a solution (cf. (10)):

$$S_p(A, E) \quad = \quad \{\, p \,\} \quad \text{if } \bigcup p = A \tag{14}$$

If $\bigcup p \neq A$, then all possible ways of extending p to cover a particular aspect $a \in A - \bigcup p$ can be considered (cf. (9)):

$$S_p(A, E) \quad = \quad (\bigcup e : e \in E \ \wedge\ a \in e \ \wedge\ e \cap \bigcup p = \emptyset : S_{p \cup \{ e \}}(A, E)) \quad \text{if } a \in A - \bigcup p \tag{15}$$

Here is a recursive implementation for *Solve1* based on these properties:

proc *Solve1* (p: $\mathcal{P}(E)$)
  { **pre**: $p \in PS(A, E)$
    post: $(\forall\, s : s \in S_p(A, E) : Solution(s)$ called once)
    vf: $\#(A - \bigcup p)$
  }
  |[**if** $\bigcup p = A \rightarrow$ { $p \in S(A, E)$ } *Solution*(p)
    [] $\bigcup p \neq A \rightarrow$ { $A - \bigcup p \neq \emptyset$ }
      |[**var** a: A ; e: E
      ; $a :\in A - \bigcup p$ { a must be covered in every solution }
      ; **for** $e \in E$ **with** $a \in e \ \wedge\ e \cap \bigcup p = \emptyset$ **do**
          { $p \cup \{ e \} \in PS(A, E)$ }


```
            Solve1 ( p ∪ { e } )
        od
      ]|
    fi
  ]|
```

Each solution of puzzle (A, E) will be processed exactly once by calling

Solve1 ($\emptyset$)

Note that there is still a large degree of freedom (nondeterminism) in the choice of aspect a to be covered next ($a :\in A - \bigcup p$). We will reduce that freedom later.

4 Transforming the basic procedure

Procedure *Solve1* can be used in a general-purpose puzzle-solving program. The input to such a program is a puzzle description, which is somehow stored in data structures for A and E. Procedure *Solve1* accesses these data structures to solve the puzzle. We shall not take this approach. Instead, we will transform the basic procedure, in a number of steps, to eliminate the data structures and to incorporate them into the topology of the program. What results is a puzzle-specific program using only a few puzzle-independent data structures and operations.

4.1 Eliminating $A - \bigcup p$

As a first step, we eliminate the expression $A - \bigcup p$, which captures the remaining aspects to be covered, by introducing an extra parameter q with the same value:

```
proc Solve2 ( p: P(E); q: P(A) )
  { pre:  p ∈ PS(A, E)  ∧  q = A − ⋃ p
    post: (∀ s : s ∈ S_p(A, E) : Solution(s) called once)
    vf:   #q
  }
  |[ if q = ∅ → { p ∈ S(A, E) } Solution(p)
    [] q ≠ ∅ →
      |[ var a: A ; e: E
       ; a :∈ q { a must be covered in every solution }
       ; for e ∈ E with a ∈ e  ∧  e ⊆ q do { p ∪ { e } ∈ PS(A, E) }
            Solve2 ( p ∪ { e }, q − e )
         od
      ]|
    fi
  ]|
```


Each solution of puzzle (A, E) will be processed exactly once by calling

Solve2 ($\emptyset$, A)

4.2 Converting parameters p, q to global variables

As the next step in this transformation, we get rid of the explicit parameter passing for each call of procedure *Solve2*. This is done by converting parameters p and q into global variables. The pre- and postcondition of the new procedure *Solve3* are strengthened to ensure that the values of p and q are invariant. Before *Solve3* is recursively called, the values of p and q are adjusted, and directly after the recursive call returns, these changes are undone. In the annotation of a procedure, we write $\tilde{v}$ for the initial value of v when the procedure is invoked.

var p: $\mathcal{P}(E)$; q: $\mathcal{P}(A)$; { **inv**: $p \in PS(A, E) \ \wedge \ q = A - \bigcup p$ }

proc *Solve3* { **glob**: p, q }
  { **pre**: true
    post: ($\forall s : s \in S_p(A, E) :$ *Solution*(s) called once)
          $p = \tilde{p} \ \wedge \ q = \tilde{q}$
    vf: $\#q$
  }
  |[**if** $q = \emptyset \rightarrow$ { $p \in S(A, E)$ } *Solution*(p)
    [] $q \neq \emptyset \rightarrow$
      |[**var** a: A ; e: E
      ; $a :\in q$ { a must be covered in every solution }
      ; **for** $e \in E$ **with** $a \in e \ \wedge \ e \subseteq q$ **do** { $p \cup \{e\} \in PS(A, E)$ }
            $p, q := p \cup \{e\}, q - e$
        ; *Solve3*
        ; $p, q := p - \{e\}, q \cup e$
        od
      ]|
    fi
  ]|

Each solution of puzzle (A, E) will be processed exactly once by

$p, q := \emptyset, A$; *Solve3*

4.3 Refining the choice $a :\in q$

There are various ways to refine $a :\in q$ for choosing the next free aspect to be covered [**?**]. We will restrict ourselves here to a simple choice, even if that is not always optimal for performance. The choice will be based on a total order $<$ for A, namely $a := \min q$. Note

that there is still the freedom to choose the total order. How to make a good choice for the order does not concern us at this moment.

It is not so easy to speed up the calculation of $\min q$ by maintaining $m = \min q$ for a fresh variable m. In particular, the operation $q := q - e$ complicates this. On the other hand, we do know $\min(q - e) > \min q$. Therefore, we introduce a parameter a with precondition $a \leq \min q$. If $a \in q$ then $q \neq \emptyset \ \wedge \ a = \min q$. Otherwise, if $a \notin q$, then $a < \min q$ and, hence, $succ(a) \leq \min q$. Furthermore, if $a > \max A$ then $q = \emptyset$. Let $A^+ = A \cup \{ succ(\max A) \}$. We now have constructed:

var p: $\mathcal{P}(E)$; q: $\mathcal{P}(A)$; { **inv**: $p \in PS(A, E) \ \wedge \ q = A - \bigcup p$ }

proc *Solve4* (a: A^+) { **glob**: p, q }
  { **pre**: $a \leq \min q$
    post: $(\forall s : s \in S_p(A, E) : Solution(s)$ called once)
      $p = \tilde{p} \ \wedge \ q = \tilde{q}$
    vf: $\#q$
  }
  |[**if** $a > \max A \rightarrow$ { $q = \emptyset$, hence $p \in S(A, E)$ } *Solution*(p)
    [] $a \leq \max A \ \wedge \ a \notin q \rightarrow$ { $succ(a) \leq \min q$ } *Solve4* ($succ(a)$)
    [] $a \in q \rightarrow$ { $a = \min q$, a must be covered in every solution }
      |[**var** e: E
       ; **for** $e \in E$ **with** $a \in e \ \wedge \ e \subseteq q$ **do** { $p \cup \{ e \} \in PS(A, E)$ }
           $p, q := p \cup \{ e \}, q - e$
         ; *Solve4* ($succ(a)$)
         ; $p, q := p - \{ e \}, q \cup e$
        od
      ]|
    fi
  ]|

Each solution of puzzle (A, E) will be processed exactly once by

$p, q := \emptyset, A$; *Solve4* ($\min A$)

Note that in case $a \in q$, it may be possible, depending on e, to increase the a-parameter of the recursive call to *Solve4* even more. We will ignore this for now.

4.4 Instantiating *Solve4*(a) for each $a \in A^+$

We eliminate parameter a by instantiating *Solve4*(a) for each $a \in A^+$. The resulting procedures are named $Solve5_a$.

var p: $\mathcal{P}(E)$; q: $\mathcal{P}(A)$; { **inv**: $p \in PS(A, E) \ \wedge \ q = A - \bigcup p$ }

proc $Solve5_a$ { **glob**: p, q } **foreach** $a \in A$
  { **pre**: $a \leq \min q$
    post: $(\forall\, s : s \in S_p(A, E) : Solution(s)$ called once)
      $p = \tilde{p} \ \wedge \ q = \tilde{q}$
    vf: $\#q$
  }
  |[**if** $a \notin q \rightarrow$ { $succ(a) \leq \min q$ } $Solve5_{succ(a)}$
    [] $a \in q \rightarrow$ { $a = \min q$, a must be covered in every solution }
      |[**var** e: E
      ; **for** $e \in E$ **with** $a \in e \ \wedge \ e \subseteq q$ **do** { $p \cup \{e\} \in PS(A, E)$ }
          $p, q := p \cup \{e\}, q - e$
        ; $Solve5_{succ(a)}$
        ; $p, q := p - \{e\}, q \cup e$
        od
      ]|
    fi
  ]|

proc $Solve5_{succ(\max A)}$ { **glob**: p, q }
  { **pre**: $succ(\max A) \leq \min q$, hence $q = \emptyset$ and $p \in S(A, E)$
    post: $(\forall\, s : s \in S_p(A, E) : Solution(s)$ called once)
      $p = \tilde{p} \ \wedge \ q = \tilde{q}$
  }
  |[$Solution(p)$]|

Each solution of puzzle (A, E) will be processed exactly once by

$$p, q := \emptyset, A \,;\, Solve5_{\min A}$$

4.5 Partitioning E into E_a with $a \in A$

We have now obtained a much larger program, because for each aspect a, a separate procedure $Solve5_a$ has been introduced. The advantage is that within each procedure $Solve5_a$ with $a \in A$, disjoint subsets of E play a role in the for-loops. Here is why. Observe that

$$a = \min q \ \wedge \ a \in e \ \wedge \ e \subseteq q \ \ \Rightarrow \ \ a = \min e \tag{16}$$

Hence, every $e \in E$ selected in the for-loop of $Solve5_a$ satisfies $\min e = a$. The domain of the for-loop in $Solve5_a$ can, thus, be restricted to

$$E_a \ \ = \ \ \{e \mid e \in E \ \wedge \ \min e = a\} \tag{17}$$

The sets E_a are pairwise disjoint, because

$$e \in E_a \cap E_{a'}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ definition of } E_a \ \} \\
 & e \in E \ \wedge \ \min e = a \ \wedge \ \min e = a' \\
\Rightarrow & \{ \text{ Leibniz' Rule applied to function min } \} \\
 & a = a'
\end{array}
$$

Consequently, the data structure storing E can be distributed over the procedure instances. Simply replace

for $e \in E$ **with** $a \in e \ \wedge \ e \subseteq q$ **do**

by

for $e \in E_a$ **with** $e \subseteq q$ **do**

observing that $e \in E_a$ implies $a \in e$. The resulting procedures are named $Solve6_a$.

4.6 Unrolling the for-loops over E_a

The data structure for storing E_a can be incorporated into the topology of the program by completely unrolling the for-loops. Assuming

$$E_a \ = \ \{ e_0, e_1, \ldots, e_{n-1} \} \tag{18}$$

replace

for $e \in E_a$ **with** $e \subseteq q$ **do**
  $p, q := p \cup \{ e \}, q - e$
; $Solve6_{succ(a)}$
; $p, q := p - \{ e \}, q \cup e$
od

by

if $e_0 \subseteq q$ **then**
  $p, q := p \cup \{ e_0 \}, q - e_0$
; $Solve7_{succ(a)}$
; $p, q := p - \{ e_0 \}, q \cup e_0$
fi
$\vdots$
; **if** $e_{n-1} \subseteq q$ **then**
  $p, q := p \cup \{ e_{n-1} \}, q - e_{n-1}$
; $Solve7_{succ(a)}$
; $p, q := p - \{ e_{n-1} \}, q \cup e_{n-1}$
fi

to obtain $Solve7_a$. Note that embeddings $e_0, \ldots, e_{n-1}$ actually depend on a.

The program has grown further, but its data structures have been reduced in size. The length of the program is on the order of the number of embeddings, that is, $\mathcal{O}(\#E)$.

4.7 Eliminating p

Each embedding $e \in E$ now occurs in a unique if-statement in the program. The global variable p is no longer needed, since its value can be reconstructed from the stacked return addresses of the calls to $Solve7_a$. Thus, the if-statement involving $e \in E$ in the unrolled loops can be reduced to

if $e \subseteq q$ **then**
  $q := q - e$
; $Solve8_{succ(a)}$
; $q := q \cup e$
fi

obtaining $Solve8_a$. Note that p is still needed as a ghost variable in the annotation and that $Solve8_{succ(\max A)}$ needs to process the solution encoded on the stack.

4.8 Expanding the embeddings $e \in E$

To simplify the operations further, we expand each embedding e into its elements, say

$$e \;=\; \{ a_0, a_1, \ldots, a_{k-1} \} \tag{19}$$

Note that the aspects $a_0, \ldots, a_{k-1}$ actually depend on e. The guard $e \subseteq q$ can now be replaced by

$$a_0 \in q \;\wedge\; \ldots \;\wedge\; a_{k-1} \in q \tag{20}$$

and the assignments $q := q - e$ and $q := q \cup e$ by

$$q := q - \{ a_0 \} ; \ldots ; q := q - \{ a_{k-1} \}$$

and

$$q := q \cup \{ a_0 \} ; \ldots ; q := q \cup \{ a_{k-1} \}$$

The resulting code for the if-statement involving embedding e can be reordered as

if $a_0 \in q$ **then**
  $q := q - \{ a_0 \}$
; **if** $a_1 \in q$ **then**
    $q := q - \{ a_1 \}$
  ; $\ldots$
    ; **if** $a_{k-1} \in q$ **then**
      $q := q - \{ a_{k-1} \}$
      ; $Solve9_{succ(a)}$
      ; $q := q \cup \{ a_{k-1} \}$


```
        fi
    ...
  ; q := q ∪ { a_1 }
  fi
; q := q ∪ { a_0 }
fi
```

to obtain procedures $Solve9_a$. By representing q as a boolean array, the code can be further refined to

```
if q[a_0] then
  q[a_0] := false
; if q[a_1] then
    q[a_1] := false
  ; ...
    ; if q[a_{k-1}] then
        q[a_{k-1}] := false
      ; Solve9_{succ(a)}
      ; q[a_{k-1}] := true
      fi
    ...
  ; q[a_1] := true
  fi
; q[a_0] := true
fi
```

Each solution of puzzle (A, E) will be processed exactly once by

```
  for a ∈ A do
      q[a] := true
  od
; Solve9_{min A}
```

4.9 Exploiting overlap among embeddings

References

[1] de Bruijn, N. G., *Programmeren van de Pentomino Puzzle*, Euclides, **47**:90–104, 1971

[2] Golomb, Solomon W., *Polyominoes*, Charles Scribner's Sons, 1965 (Revised Edition 1994)

[3] Golomb, Solomon W. and Baumert, Leonard D., *Backtrack Programming*, Journal of the ACM, **12**(4):516–524, Oct. 1965

[4] Martin, George E., *Polyominoes: A Guide to Puzzles and Problems in Tiling*, Mathematical Association of America, 1991

[5] Tonneijk, B. L. A., *Het Puzzel-Processor Project: Vergelijking van Puzzel-Algoritmen*, Master's Thesis, Eindhoven University of Technology, Fac. of Math. and CS, Aug. 1994

A …

Author(s) Erik van der Tol and Tom Verhoeff

Title The Puzzle Processor Project
Towards an Implementation

Distribution

Nat.Lab./PI	WB-5	
PRL	Redhill, UK	
PRB	Briarcliff Manor, USA	
LEP	Limeil–Brévannes, France	
PFL	Aachen, BRD	
CIP	WAH	
Director:	Dr. G.J. Koel	WB56
Department Head:	Dr.U.K.P. Biermann	WZ07

Abstract

D.B. Jansen	Nat.Lab.	WAY-31
R.C.A.M. Jansen	Nat.Lab.	WAY-31

Full report

G.F.M. Jansen	Nat.Lab.	WY-01
A.P.J. Jansen	Nat.Lab.	WY-01
J.G. Jansen	Nat.Lab.	WY-01
A.K. Jansen	Nat.Lab.	WY-01
A. Jansen	Nat.Lab.	WY-01
P. Jansen	Nat.Lab.	WY-01
F. Jansen	Nat.Lab.	WAY-31
M. Jansen	Nat.Lab.	WY-01
P. Jansen	Nat.Lab.	WY-01
D. Jansen	Nat.Lab.	WY-01
J. Jansen	Nat.Lab.	WY-01
A. Jansen	Nat.Lab.	WY-01
L.R.M. Jansen	Nat.Lab.	WY-01

