Nat.Lab. Unclassified Report NL-UR 2000/828

Date of issue: 12/01

The Puzzle Processor Project

Towards an Implementation

Erik van der Tol and Tom Verhoeff

Authors' address data: E. B. van der Tol; erik.van.der.tol@philips.com
T. Verhoeff; T.Verhoeff@tue.nl

Unclassified Report: NL-UR 2000/828

Title: The Puzzle Processor Project
Towards an Implementation

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

Part of project: Puzzle Processor Project

Customer: Not applicable

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 a certain kind 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, such as scheduling.

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 shown how a **general-purpose backtrack program** for solving packing puzzles can be transformed systematically into a **puzzle-specific program** involving just a few computational primitives. This transformation can even be automated.

Next we have specified and designed a **puzzle processor** to execute these computational primititves efficiently. It has only five instructions acting on four registers. A packing puzzle can now be compiled into a dedicated program for this processor. When executed, the program determines solutions to the puzzle.

The programs are puzzle-specific, and the processor is domain-specific. This can be exploited to arrive at very efficient puzzle solvers. We have compared three simple implementations.

The **high branching density** of typical programs for the puzzle processor, together with **low branching predictability**, pose a challenge for efficient pipelining, which we have not attempted to tackle in this report.

Future research will also look into the possibility of operating many puzzle processors in parallel to improve performance further.

Contents

1 Introduction **1**

2 Puzzle descriptions **2**

2.1 Cells and pieces 2

2.2 Generalization to aspects 3

2.3 Abstract puzzles 5

3 Solving abstract puzzles **7**

4 Transforming the basic procedure **8**

4.1 Introducing an extra parameter for the set of free aspects 8

4.2 Converting parameters into global variables 9

4.3 Refining the choice of free aspect by introducing a parameter 10

4.4 Eliminating a parameter by instantiation for all relevant values 11

4.5 Simplifying the iteration by partitioning its domain 11

4.6 Unrolling the for-loops 13

4.7 Eliminating a global variable 13

4.8 Expanding the embeddings 14

4.9 Exploiting overlap among embeddings 14

4.10 Representing a set by a boolean array 15

4.11 Example 15

5 Refinement toward hardware **18**

5.1 Instruction set 19

5.2 Encoding instructions 20

5.3 Encoding sets of aspects 21

5.4 VLSI Programming 22

6 Tangram Designs **23**

6.1 Straightforward implementation 23

6.2 Procedures and precomputed values 24

6.3 Prefetching 24

7 Conclusion **25**

References **26**

A Program in C for Simple Puzzle 27

B Puzzle-Processor ‘Assembly Listing’ for Simple Puzzle 29

C Puzzle-Processor ‘Machine Code’ for Simple Puzzle 31

D Puzzle-Processor Interpeter in C 31

E Straigthforward design with minimal parallelism 33

F Design with procedures and precomputed values 34

G Design with prefetching 35

Distribution

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* [7], shown in Figure 1. The pentomino puzzle consists of a box of 6 by 10 unit

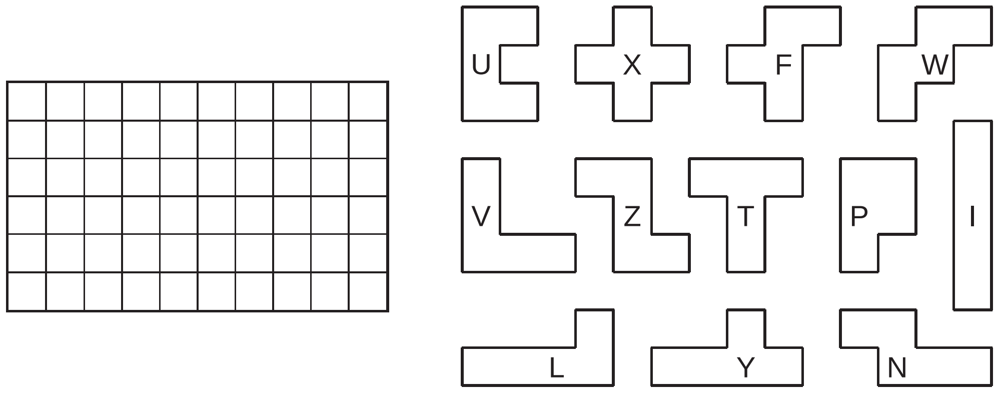

Figure 1: *The* 6×10 *pentomino puzzle: box (left) and 12 pieces (right)*

squares (cells), in which the 12 pentominoes have to be placed. Each pentomino consists of a unique combination of 5 unit squares. There are exactly 12 such combinations. The pentominoes may be freely translated, rotated, and reflected when placed in the box. Thus, there are many ways to place each piece in the box. Note that the unit squares of the pieces are indistinguishable. For example, piece I can be placed in the box in 56 ways: $6 * 6$ horizontally and $2 * 10$ vertically. Figure 2 shows one of the 9356 solutions[1] for the 6×10 pentomino puzzle.[2]

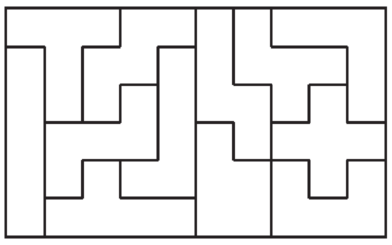

Figure 2: *An elegant solution for the* 6×10 *pentomino puzzle*

Algorithms for solving this kind of puzzles are usually based on backtracking [8]. Instead of a general-purpose backtrack program that takes a puzzle description as input, we 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 occurring in the puzzle-specific backtrack programs. To solve a puzzle, we generate a dedicated program from the puzzle's description in terms of puzzle-processor instructions and then execute it on the puzzle processor.

[1]2339 modulo rotation and reflection.

[2]A solution to the 5×12 pentomino puzzle can be found on the 4th floor of building WAY at the Philips Research Lab on a tapestry called “Maartens pentomino” by Maarten Vliegenthart.

2 Puzzle descriptions

Let us look at a concrete example of a very simple puzzle. Figure 3 shows a 2×3 rectangular box and three pieces, named A, B, and C, to be fit into the box. For ease of reference, the cells in the box have been labeled from 0 to 5. We use this puzzle to illustrate our ideas.

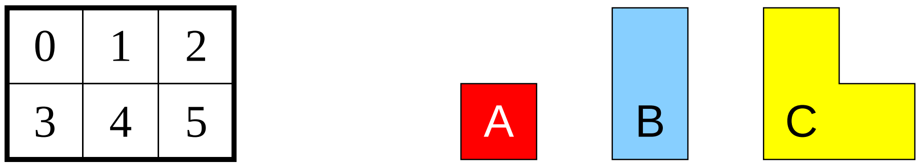

Figure 3: *A simple puzzle:* 2×3 *box (left) and 3 pieces (right)*

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. Several approaches are possible, most of which distinghuish between the role of the cells in the box and the role of the pieces (see e.g. [3]). This is discussed further in the next section. We take a more general approach, which we present in §2.2.

2.1 Cells and pieces

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

1. *Concentrate on the cells*. Every cell has to be covered to obtain a solution. Consider the cells in some order, for instance in 'reading order'. Separately investigate each possible way to cover the 'next' empty cell by an unused piece. Note that a piece may be put in the box in various orientations. Each such covering results in a partial solution, leaving a similar puzzle with a smaller box[3] and fewer pieces. When all cells have been covered, a solution has been obtained. For example, the top-left cell of the simple example puzzle (cell 0 in 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*. Every piece has to be used to obtain a solution. Consider the pieces in some order, for instance in alphabetic order. Separately investigate each possible way to place the 'next' unused piece in an 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 'cells' strategy, the order in which the cells are attempted affects the running time of the program, while in the 'pieces' strategy, the order of the pieces is crucial. Tonneijk has compared several backtrack strategies for solving puzzles in [14]. Tonneijk observes that, in general, the 'cells' strategy is faster than the 'pieces' strategy, because the former gives rise to 'better-related' subproblems, which all behave decently under the same cell order.

[3]That is, a box with less empty space.

2.2 Generalization to aspects

Until now we have regarded the cells 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 cells have been covered and which pieces have been used. In our kind of puzzles, a cell 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 cell and for each piece.

We now take a more formal approach to describing puzzles. Let Γ be the set of cells and Π be the set of pieces. For the simple puzzle of Figure 3, we have

$$\begin{aligned} \Gamma &= \{0, 1, 2, 3, 4, 5\} \\ \Pi &= \{\mathsf{A}, \mathsf{B}, \mathsf{C}\} \end{aligned}$$

The placement of a single piece in the box is completely described by indicating which cells γ have been covered and which piece π has been used. Thus, a **piece placement** can be formalized as a pair (γ, π), with $\gamma \subseteq \Gamma$ and $\pi \in \Pi$. A **solution** to the puzzle consists of a set of such placements with the property that every cell is covered exactly once and every piece has been used exactly once.

It is, however, unnecessary to distinguish the role of the cells and the role of the pieces. By suitable renaming we can ensure $\Gamma \cap \Pi = \emptyset$. Define set A by

$$A = \Gamma \cup \Pi \tag{1}$$

It unifies the notions of a cell and a piece into a single generalized notion, which we call an **aspect** of the puzzle. A piece placement now corresponds to a set e of aspects, that is, $e \subseteq A$. We call such a set of aspects an **embedding**. In terms of the piece placement modeled as a pair (γ, π) we have the correspondence:

$$\begin{aligned} e &= \gamma \cup \{\pi\} && (2) \\ \gamma &= e \cap \Gamma && (3) \\ \{\pi\} &= e \cap \Pi && (4) \end{aligned}$$

A solution is a set of embeddings that partitions the set of aspects.

The shape of the pieces determines which subsets of the aspects can be covered. This set of embeddings is obtained by considering all possible placements of the pieces in the box, taking into account translations, rotations, and reflections. The simple puzzle from Figure 3 has $6 + 3 = 9$ aspects. Figure 4 depicts all its 21 embeddings, 6 for piece A, 7 for B, and 8 for C.

The 6×10 pentomino puzzle has $60 + 12 = 72$ aspects and, as it turns out, a total of 2056 embeddings. Table 1 lists the number of embeddings for each piece.

The set of all embeddings need not be determined explicitly. It can be constructed on the fly by translating, rotating, and reflecting the basic shapes of the pieces during backtracking. Often, the rotations and reflections of the piece shapes are precomputed and the translations are done on the fly (as in [3]). Even doing only the translations on the fly incurs a time penalty. When there is enough memory, all embeddings can be precomputed to speed up the backtracking. This is a matter of time-memory trade-off.

The generalization to aspects has several advantages:

1. It *simplifies the data structure* needed to store a puzzle.

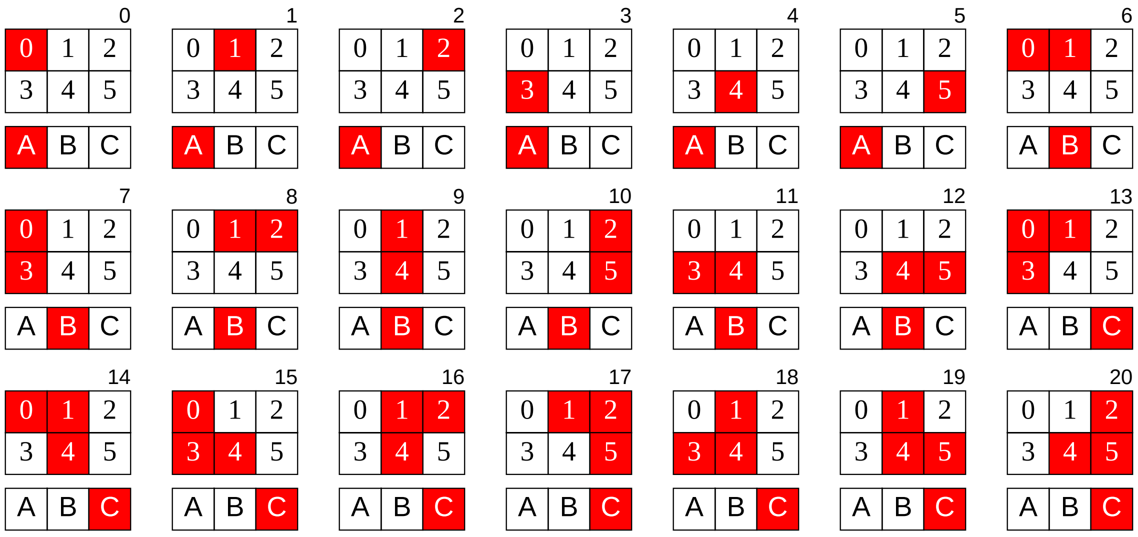

Figure 4: *The 21 embeddings for the simple puzzle in Fig. 3*

Piece	Embeddings	Piece	Embeddings
F	256	U	152
I	56	V	128
L	248	W	128
N	248	X	32
P	304	Y	248
T	128	Z	128

Table 1: *Number of embeddings for each piece in the* 6×10 *pentomino puzzle*

2. It allows *mixed backtrack strategies* based on a combined order of cells and pieces.

3. It helps in solving *variations on puzzels* with additional constraints on the allowed embeddings. As an example, consider the 6×10 pentomino puzzle with the additional constraint that all pieces touch the boundary.

4. It can be used to avoid finding *equivalent solutions* more than once, by forbidding appropriate embeddings. For example, observe that the box of the simple puzzle in Figure 3 has four symmetries generated by vertical and horizontal reflection, whereas piece C has only one of these symmetries (viz. the identity). Therefore, solutions come in groups of four. By forbidding three of the four rotations of piece C, only one solution in each group of four is allowed.

The proposed generalization still has some limitations. One can imagine puzzles where cells 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). It is straightforward to adapt our treatment for such more general puzzles by using *bags* instead of sets (also see §4.10). It is also possible to imagine puzzles where not all cells have to be covered or not all pieces have to be used in a solution. We do not deal with these more general situations in this report, but the framework can be adapted appropriately.

2.3 Abstract puzzles

We now formally define the notion of an abstract puzzle, which avoids the distinction between cells to cover and pieces to use. We also formally define what a solution and a partial solution of an abstract puzzle are. Finally, we give some useful properties for solving abstract puzzles.

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

$$E \subseteq \mathcal{P}(A) \ \wedge \ E \neq \emptyset \ \wedge \ \emptyset \notin E \tag{5}$$

A member of A is called an **aspect** of the puzzle, and a member of E is called an **embedding**. An embedding is a nonempty set of aspects. Condition (5) helps to exclude certain pathological cases, in particular it implies $A \neq \emptyset$.

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

$$\begin{array}{llllll}
(& \{0,1,2,3,4,5,\mathsf{A},\mathsf{B},\mathsf{C}\} & & & & \\
, & \{ \ \{0,\mathsf{A}\} & , & \{1,\mathsf{A}\} & , & \{2,\mathsf{A}\} \\
 & , \ \{3,\mathsf{A}\} & , & \{4,\mathsf{A}\} & , & \{5,\mathsf{A}\} \\
 & , \ \{0,1,\mathsf{B}\} & , & \{0,3,\mathsf{B}\} & , & \{1,2,\mathsf{B}\} \\
 & , \ \{1,4,\mathsf{B}\} & , & \{2,5,\mathsf{B}\} & , & \{3,4,\mathsf{B}\} \\
 & , \ \{4,5,\mathsf{B}\} & , & \{0,1,3,\mathsf{C}\} & , & \{0,1,4,\mathsf{C}\} \\
 & , \ \{0,3,4,\mathsf{C}\} & , & \{1,2,4,\mathsf{C}\} & , & \{1,2,5,\mathsf{C}\} \\
 & , \ \{1,3,4,\mathsf{C}\} & , & \{1,4,5,\mathsf{C}\} & , & \{2,4,5,\mathsf{C}\} \\
 & \} & & & & \\
) & & & & &
\end{array} \tag{6}$$

with 9 aspects and 21 embeddings (6 involve piece A, 7 involve B, and 8 involve C, also see Figure 4). The aspects are *dummies*, in the sense that systematic renaming of the aspects yields an *isomorphic* puzzle. If we restrict piece C to one of its four rotations, e.g. to embeddings 18 and 20 in Figure 4, then the resulting abstract puzzle has only 15 embeddings:

$$\begin{array}{llllll}
(& \{0,1,2,3,4,5,\mathsf{A},\mathsf{B},\mathsf{C}\} & & & & \\
, & \{ \ \{0,\mathsf{A}\} & , & \{1,\mathsf{A}\} & , & \{2,\mathsf{A}\} \\
 & , \ \{3,\mathsf{A}\} & , & \{4,\mathsf{A}\} & , & \{5,\mathsf{A}\} \\
 & , \ \{0,1,\mathsf{B}\} & , & \{0,3,\mathsf{B}\} & , & \{1,2,\mathsf{B}\} \\
 & , \ \{1,4,\mathsf{B}\} & , & \{2,5,\mathsf{B}\} & , & \{3,4,\mathsf{B}\} \\
 & , \ \{4,5,\mathsf{B}\} & , & \{1,3,4,\mathsf{C}\} & , & \{2,4,5,\mathsf{C}\} \\
 & \} & & & & \\
) & & & & &
\end{array} \tag{7}$$

For set s of embeddings ($s \subseteq E$), we define the set $\bigcup s$ of **aspects covered by** s by

$$\bigcup s \quad = \quad (\bigcup e : e \in s : e) \tag{8}$$

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

$$s \subseteq E \tag{9}$$

$$(\forall e, e' : e \in s \ \wedge \ e' \in s \ \wedge \ e \neq e' : e \cap e' = \emptyset) \tag{10}$$

$$\bigcup s = A \tag{11}$$

Condition (10) expresses that embeddings in a solution are pairwise disjoint (do not overlap), and condition (11) that every aspect is covered. For example,

$$\{\{1, \mathsf{A}\}, \{2, 5, \mathsf{B}\}, \{0, 3, 4, \mathsf{C}\}\} \tag{12}$$

is a solution of the abstract puzzle (6) corresponding to Figures 3 and 4. It is depicted in Figure 5. Set (12) is *not* a solution of the abstract puzzle (7), which is a restricted version of (6).

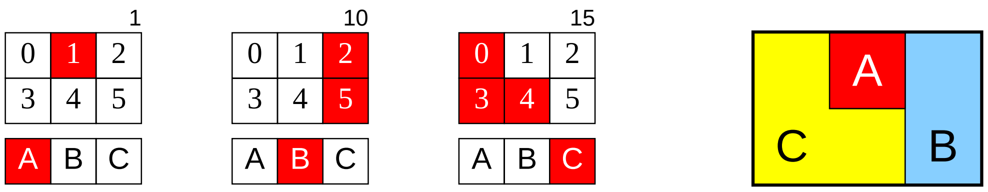

Figure 5: *Solution of simple puzzle*

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

$$S(A, E) \quad = \quad \{\, s \mid s \subseteq E \ \wedge \ s \text{ partitions } A \,\} \tag{13}$$

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:

$$p \subseteq E \tag{14}$$

$$(\forall\, e, e' : e \in p \ \wedge \ e' \in p \ \wedge \ e \neq e' : e \cap e' = \emptyset) \tag{15}$$

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

$$PS(A, E) \quad = \quad \{\, p \mid p \subseteq E \ \wedge \ p \text{ is free of overlap} \,\} \tag{16}$$

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

$$PS(A, E) \supseteq S(A, E) \tag{17}$$

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

$$\emptyset \in PS(A, E) \tag{18}$$

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

$$p \in PS(A, E) \ \wedge \ e \cap \bigcup p = \emptyset \quad \equiv \quad e \notin p \ \wedge \ p \cup \{\, e \,\} \in PS(A, E) \tag{19}$$

Note that this property relies on $\emptyset \notin E$ (cf. (5)), which implies $e \neq \emptyset$. A partial solution that covers *all* of A is a solution (useful to **terminate** the search for a solution):

$$p \in PS(A, E) \ \wedge \ \bigcup p = A \quad \equiv \quad p \in S(A, E) \tag{20}$$

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

$$\{ \mathit{true} \} \quad \mathit{Solve} \quad \{ (\forall s : s \in S(A, E) : \mathit{Solution}(s) \text{ called once}) \} \tag{21}$$

where *Solution* is a procedure that processes a solution, for example, printing it or just counting it. Note that deciding whether an abstract puzzle has a solution is an NP-complete problem [6]. Furthermore, puzzles exist whose number of solutions is superexponential in the size of the puzzle. Consider for instance the abstract puzzle (A_n, E_n) with

$$\begin{aligned} A_n &= \{ a \in \mathbf{N} \mid 0 \le a < 2n \} \\ E_n &= \{ \{ \gamma, \pi \} \mid 0 \le \gamma < n \le \pi < 2n \} \end{aligned}$$

which has $2n$ aspects, n^2 embeddings, and $n!$ solutions.

Specification (21) can be generalized by introducing parameter p, being a partial solution, and requiring that $\mathit{Solve}(p)$ processes once every solution that extends p:

$$\{ p \in \mathit{PS}(A, E) \} \quad \mathit{Solve}\ (\ p\ ) \quad \{ (\forall s : s \in S_p(A, E) : \mathit{Solution}(s) \text{ called once}) \} \tag{22}$$

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

$$S_p(A, E) \quad = \quad \{ s \mid s \in S(A, E) \ \wedge\ p \subseteq s \} \tag{23}$$

Taking $p := \emptyset$ (cf. (18)) yields the original specification (21), because $\emptyset \subseteq s$ holds vacuously:

$$S_p(A, E) \quad = \quad S(A, E) \quad \text{if } p = \emptyset \tag{24}$$

Thus, specification (22) indeed generalizes specification (21), and $\mathit{Solve}(\emptyset)$ satisfies the latter.

If $\bigcup p = A$, then p is actually a solution, and it is the only one that extends p (cf. (20)):

$$S_p(A, E) \quad = \quad \{ p \} \quad \text{if } \bigcup p = A \tag{25}$$

If $\bigcup p \ne A$, then all possible ways of extending p to cover a particular aspect $a \in A - \bigcup p$ can be considered (cf. (19)):

$$\begin{aligned} S_p(A, E) \quad = \quad & \left(\bigcup e : e \in E \ \wedge\ a \in e \ \wedge\ e \cap \bigcup p = \emptyset : S_{p \cup \{ e \}}(A, E) \right) \\ & \text{if } a \in A - \bigcup p \end{aligned} \tag{26}$$

Here is a recursive implementation for *Solve* based on properties (24) through (26):

```
proc Solve ( $p$: $\mathcal{P}(E)$ )
  { pre:  $p \in PS(A, E)$
    post: $(\forall s : s \in S_p(A, E) : Solution(s)$ called once)
    vf:   $\#(A - \bigcup p)$
  }
  |[ if $\bigcup p = A \rightarrow$ { cf. (25), $p \in S(A, E)$ } $Solution(p)$
    [] $\bigcup p \neq A \rightarrow$ { cf. (26), $A - \bigcup p \neq \emptyset$ }
      |[ var $a$: $A$ ; $e$: $E$
       ; $a :\in A - \bigcup p$ { $a$ must be covered in each solution, try all possibilities }
       ; for $e \in E$ with $a \in e \ \wedge\ e \cap \bigcup p = \emptyset$ do
            { $p \cup \{e\} \in PS(A, E)$ }
            Solve ( $p \cup \{e\}$ )
         od
      ]|
    fi
  ]|
```

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

Solve ($\emptyset$)

Note that there still is a large degree of freedom (nondeterminism) in the choice of aspect a to be covered next ($a :\in A - \bigcup p$). We reduce that freedom in §4.3. This can be done because the order in which partial solutions are extended does not affect the final result (though it does affect the order in which solutions are processed, and hence also the time it takes to find the first solution).

4 Transforming the basic procedure

Procedure *Solve* 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 *Solve* accesses these data structures to solve the puzzle. We do not take this approach. Instead, we 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 Introducing an extra parameter for the set of free aspects

Observe that expression $A - \bigcup p$ occurs a number of times in procedure *Solve*. It yields the set of remaining aspects to be covered. To avoid recomputation, we eliminate this expression, by introducing an extra parameter q with value $A - \bigcup p$:

```
proc Solve1 ( $p$: $\mathcal{P}(E)$; $q$: $\mathcal{P}(A)$ )
  { pre:  $p \in PS(A, E) \ \wedge\ q = A - \bigcup p$
    post: $(\forall s : s \in S_p(A, E) : Solution(s)$ called once)
```

```
        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 each solution, try all ways }
         ; for $e \in E$ with $a \in e \ \wedge \ e \subseteq q$ do { $p \cup \{ e \} \in PS(A, E)$ }
               $Solve1$ ( $p \cup \{ e \}$, $q - e$ )
           od
        ]|
      fi
    ]|
```

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

$Solve1$ ($\emptyset$, A)

4.2 Converting parameters into global variables

As the next step in this sequence of transformations, we get rid of the explicit parameter passing for each call of procedure *Solve1*. This is done by converting parameters p and q into global variables. The pre- and postcondition of the new procedure *Solve2* are strengthened to ensure that the values of p and q are invariant. Before *Solve2* 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 $Solve2$ { glob: $p, q$ }
    { pre:  true
      post: $(\forall s : s \in S_p(A, E) : Solution(s)$ called once$)$
            $p = \tilde{p} \ \wedge \ q = \tilde{q}$ ($p, q$ unchanged)
      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 each solution, try all ways }
         ; for $e \in E$ with $a \in e \ \wedge \ e \subseteq q$ do { $p \cup \{ e \} \in PS(A, E)$ }
               $p, q := p \cup \{ e \}, q - e$
             ; $Solve2$ { acts on $p, q$ }
             ; $p, q := p - \{ e \}, q \cup e$
           od
        ]|
      fi
    ]|
```

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

$$p, q := \emptyset, A \text{ ; } \mathit{Solve2}$$

4.3 Refining the choice of free aspect by introducing a parameter

Various ways to refine $a :\in q$, which chooses the next free aspect to be covered, are considered in [14]. We restrict ourselves here to a simple choice, even if that is not always optimal for performance. The choice is based on a total order $<$ for A. This order is fixed in advance (statically), i.e., it does not vary during the operation of the program (dynamically). The total order $<$ induces a successor operator *succ* and a minimum operator min on A. The choice $a :\in q$ is now refined by picking the $<$-least uncovered aspect, i.e. $a := \min q$.

Note that there still is freedom in choosing the total order. How to make a good choice for the order does not concern us in this report. See [1] for some considerations and heuristics to choose such an order.

It is not so easy to speed up the calculation of $\min q$ by maintaining $a = \min q$ for a fresh variable a. 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 the somewhat weaker precondition $a \le \min q$. If $a \in q$ then $q \ne \emptyset \ \wedge \ a = \min q$. Otherwise, if $a \notin q$, then $a < \min q$ and, hence, $\mathit{succ}(a) \le \min q$. Furthermore, if $a > \max A$ then $q = \emptyset$. Let $A^{+} = A \cup \{\infty\}$ where $\infty = \mathit{succ}(\max A)$, that is, we add an imaginary 'infinite' aspect as sentinel. 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 $\mathit{Solve3}$ ( $a$: $A^{+}$ ) { glob: $p, q$ }
  { pre:  $a \le \min q$
    post: $(\forall s : s \in S_p(A,E) : \mathit{Solution}(s)$ called once$)$
          $p = \tilde{p} \ \wedge \ q = \tilde{q}$
    vf:   $\#q$
  }
  |[ if $a = \infty$ $\to$ { $q = \emptyset$, hence $p \in S(A,E)$ } $\mathit{Solution}(p)$
     [] $a \ne \infty \ \wedge \ a \notin q$ $\to$ { $\mathit{succ}(a) \le \min q$ } $\mathit{Solve3}$ ( $\mathit{succ}(a)$ )
     [] $a \in q$ $\to$ { $a = \min q$ must be covered in each solution, try all ways }
       |[ var $e$: $E$
       ; for $e \in E$ with $a \in e \ \wedge \ e \subseteq q$ do { $p \cup \{e\} \in PS(A,E)$ }
             $p, q := p \cup \{e\}, q - e$
          ;  $\mathit{Solve3}$ ( $\mathit{succ}(a)$ )
          ;  $p, q := p - \{e\}, q \cup e$
         od
       ]|
     fi
  ]|
```

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

$$p, q := \emptyset, A \text{ ; } \mathit{Solve3}\ (\ \min A\ )$$

Note that in case $a \in q$, it may be possible, depending on e, to increase the a-parameter of the recursive call to *Solve3* even more. In fact, the recursive call $\mathit{Solve3}(\mathit{succ}(a))$ can in general be optimized to

$$\mathit{Solve3}\ (\ \min\{\, x \in A \mid a < x \ \wedge\ x \notin e \,\}\ )$$

but we ignore that in this report.

4.4 Eliminating a parameter by instantiation for all relevant values

We eliminate parameter a by instantiating $\mathit{Solve3}(a)$ for each $a \in A^+$. The resulting procedures are named $\mathit{Solve4}_a$.

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

proc $\mathit{Solve4}_a$ { glob: $p, q$ } foreach $a \in A$ { hence $a \neq \infty$ }
  { pre:  $a \le \min q$
    post: $(\forall s : s \in S_p(A,E) : \mathit{Solution}(s)$ called once)
          $p = \tilde{p} \ \wedge\ q = \tilde{q}$
    vf:   $\#q$
  }
  |[ if $a \notin q \rightarrow$ { $\mathit{succ}(a) \le \min q$ } $\mathit{Solve4}_{\mathit{succ}(a)}$
     [] $a \in q \rightarrow$ { $a = \min q$, $a$ must be covered in each solution, try all ways }
       |[ var $e$: $E$
       ; for $e \in E$ with $a \in e \ \wedge\ e \subseteq q$ do { $p \cup \{e\} \in PS(A,E)$ }
             $p, q := p \cup \{e\}, q - e$
           ; $\mathit{Solve4}_{\mathit{succ}(a)}$
           ; $p, q := p - \{e\}, q \cup e$
         od
       ]|
     fi
  ]|

proc $\mathit{Solve4}_\infty$ { glob: $p, q$ }
  { pre:  $\infty \le \min q$, hence $q = \emptyset$ and $p \in S(A,E)$
    post: $(\forall s : s \in S_p(A,E) : \mathit{Solution}(s)$ called once)
          $p = \tilde{p} \ \wedge\ q = \tilde{q}$
  }
  |[ $\mathit{Solution}(p)$ ]|
```

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

$$p, q := \emptyset, A\ ;\ \mathit{Solve4}_{\min A}$$

4.5 Simplifying the iteration by partitioning its domain

We have now obtained a much larger program, because for each aspect a, a separate procedure $\mathit{Solve4}_a$ has been introduced. The advantage is that the for-loops in the procedures $\mathit{Solve4}_a$ with

$a \in A$ involve disjoint subsets of E, because

$$a = \min q \ \wedge \ a \in e \ \wedge \ e \subseteq q \quad \Rightarrow \quad a = \min e \tag{27}$$

Hence, every $e \in E$ selected in the for-loop of $Solve4_a$ satisfies $\min e = a$. The domain of the for-loop in $Solve4_a$ can, thus, be restricted to

$$E_a \quad = \quad \{\, e \mid e \in E \ \wedge \ \min e = a \,\} \tag{28}$$

The sets E_a are pairwise disjoint, because

$$\begin{array}{ll} & e \in E_a \cap E_{a'} \\ \equiv & \qquad \{ \text{ definition of } E_a \ \} \\ & e \in E \ \wedge \ \min e = a \ \wedge \ \min e = a' \\ \Rightarrow & \qquad \{ \text{ property of } = \} \\ & a = a' \end{array}$$

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** $\{\, e \in E_a$, hence $a \in e \,\}$
```

The resulting procedures are named $Solve5_a$.

For example, in abstract puzzle (6), the order

$$0, 1, 2, 3, 4, 5, \mathsf{A}, \mathsf{B}, \mathsf{C} \tag{29}$$

induces the following partition of E:

a	**members of** E_a
0	$\{0,\mathsf{A}\}, \{0,1,\mathsf{B}\}, \{0,3,\mathsf{B}\}, \{0,1,3,\mathsf{C}\}, \{0,1,4,C\}, \{0,3,4,\mathsf{C}\}$
1	$\{1,\mathsf{A}\}, \{1,2,\mathsf{B}\}, \{1,4,\mathsf{B}\}, \{1,2,4,\mathsf{C}\}, \{1,2,5,\mathsf{C}\}, \{1,3,4,\mathsf{C}\}, \{1,4,5,\mathsf{C}\}$
2	$\{2,\mathsf{A}\}, \{2,5,\mathsf{B}\}, \{2,4,5,\mathsf{C}\}$
3	$\{3,\mathsf{A}\}, \{3,4,\mathsf{B}\}$
4	$\{4,\mathsf{A}\}, \{4,5,\mathsf{B}\}$
5	$\{5,\mathsf{A}\}$

(30)

Note that E_{A}, E_{B}, and E_{C} are empty. Alternatively, the order

$$0, 3, 1, 4, 2, 5, \mathsf{A}, \mathsf{B}, \mathsf{C} \tag{31}$$

induces this partition of E:

a	**members of** E_a
0	$\{0,\mathsf{A}\}, \{0,1,\mathsf{B}\}, \{0,3,\mathsf{B}\}, \{0,1,3,\mathsf{C}\}, \{0,1,4,C\}, \{0,3,4,\mathsf{C}\}$
3	$\{3,\mathsf{A}\}, \{3,4,\mathsf{B}\}, \{1,3,4,\mathsf{C}\}$
1	$\{1,\mathsf{A}\}, \{1,2,\mathsf{B}\}, \{1,4,\mathsf{B}\}, \{1,2,4,\mathsf{C}\}, \{1,2,5,\mathsf{C}\}, \{1,4,5,\mathsf{C}\}$
4	$\{4,\mathsf{A}\}, \{4,5,\mathsf{B}\}, \{2,4,5,\mathsf{C}\}$
2	$\{2,\mathsf{A}\}, \{2,5,\mathsf{B}\}$
5	$\{5,\mathsf{A}\}$

(32)

Again, E_{A}, E_{B}, and E_{C} are empty.

4.6 Unrolling the for-loops

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

$$E_a \quad = \quad \{ e_0, e_1, \ldots, e_{n-1} \} \tag{33}$$

replace

```
**for** $e \in E_a$ **with** $e \subseteq q$ **do**
      $p, q := p \cup \{ e \}, q - e$
   ;  $Solve5_{succ(a)}$
   ;  $p, q := p - \{ e \}, q \cup e$
**od**
```

by

```
    **if** $e_0 \subseteq q$ **then**
          $p, q := p \cup \{ e_0 \}, q - e_0$
       ;  $Solve6_{succ(a)}$
       ;  $p, q := p - \{ e_0 \}, q \cup e_0$
    **fi**
    ⋮
 ;  **if** $e_{n-1} \subseteq q$ **then**
          $p, q := p \cup \{ e_{n-1} \}, q - e_{n-1}$
       ;  $Solve6_{succ(a)}$
       ;  $p, q := p - \{ e_{n-1} \}, q \cup e_{n-1}$
    **fi**
```

to obtain $Solve6_a$. Note that embeddings $e_0, \ldots, e_{n-1}$ actually depend on a. The order in which the embeddings occur in the unrolled loop can be chosen freely.

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

4.7 Eliminating a global variable

Each embedding $e \in E$ now occurs in a unique if-statement in the program. Global variable p is no longer needed, since its value can be reconstructed from the stacked return addresses of the calls to $Solve6_a$. Thus, the if-statement involving $e \in E$ in the unrolled loops can be reduced to

```
**if** $e \subseteq q$ **then**
   $q := q - e$  $\{ p := p \cup \{ e \} \}$
 ; $Solve7_{succ(a)}$
 ; $q := q \cup e$  $\{ p := p - \{ e \} \}$
**fi**
```

obtaining $Solve7_a$. Note that p is still needed as a ghost variable in the annotation and that $Solve7_{succ(\max A)}$ needs to process the solution encoded on the stack. Also note that q is invariant over the body of the if-statement.

4.8 Expanding the embeddings

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

$$e \quad = \quad \{\, a_0, a_1, \ldots, a_{k-1} \,\} \tag{34}$$

where all a_i are distinct. Note that the aspects $a_0, \ldots, a_{k-1}$ actually depend on the choice of $e \in E$. The guard $e \subseteq q$ can now be replaced by

$$a_0 \in q \;\wedge\; \ldots \;\wedge\; a_{k-1} \in q \tag{35}$$

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

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

and

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

Because all a_i are distinct, 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
    $q := q - \{\, a_1 \,\}$
  ; ...
    ;  if $a_{k-1} \in q$ then
         $q := q - \{\, a_{k-1} \,\}$
       ; $Solve8_{succ(a)}$
       ; $q := q \cup \{\, a_{k-1} \,\}$
       fi
    ...
  ; $q := q \cup \{\, a_1 \,\}$
  fi
; $q := q \cup \{\, a_0 \,\}$
fi
```

to obtain procedures $Solve8_a$. The order of the aspects in each embedding can still be chosen freely.

4.9 Exploiting overlap among embeddings

Consider two embeddings e_0 and e_1 occurring in the same for-loop, and assume that these have aspect a in common: $a \in e_0 \cap e_1$. The order of the statements after unrolling the loop (§4.6) and expanding the embeddings (§4.8) can be chosen such that the following structure emerges:

```
   if $a \in q$ then $S_0$ fi
;  if $a \in q$ then $S_1$ fi
```

Because statements S_0 and S_1 leave q invariant, the common guard can be distributed outward:

```
**if** $a \in q$ **then**
   $S_0$
   $\{ a \in q \}$
 ; $S_1$
**fi**
```

This distribution reduces the code size and improves the execution speed. The technique can be applied recursively to common-guarded if-statements that have become adjacent in $S_0; S_1$. Exploiting the freedom to order statements after unrolling the loop and expanding the embeddings, typically yields a 50% improvement according to [1]. In §4.11 we illustrate this with an example.

4.10 Representing a set by a boolean array

By representing q as an array of booleans, the code can be further refined to

```
**var** $q$: **array** [ $A$ ] **of** *Boolean* ;
...
**if** $q[a_0]$ **then**
   $q[a_0] := false$
 ; ...
 ; $q[a_0] := true$
**fi**
```

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

```
   **for** $a \in A$ **do**
         $q[a] := true$
   **od**
 ; $Solve9_{\min A}$
```

For a more general class of puzzles (see end of §2.2), involving bags instead of sets, an array of natural numbers would be used to implement q.

4.11 Example

Applying all transformations of the preceding sections yields a simple program, consisting of procedures $Solve_a$ that operate on one global array q of $\#A$ booleans. The structure of the puzzle is captured in the the program's control flow, not in its data.

There are only two kinds of freedom in these transformations:

1. The order in which the program attempts to cover the aspects, i.e., the chosen total order $<$ for A.

2. How the overlap in embeddings is exploited.

The first choice —ordering the aspects— does not affect the overall size of the program, but it may have a large impact on the execution time. This order of aspects determines the partitioning of embeddings into the $Solve_a$ procedures and, hence, the size of the (induced) search tree. This is a difficult *global* optimization issue. A heuristic approach, called the *footprint method*, is presented in [1].

The second choice —exploiting overlap— is carried out within each procedure separately. It affects the code size and, to a smaller extent, the execution time. This is a *local* optimization isssue. A reasonably good greedy approach is presented in [1].

We illustrate the method by considering a program for a *restricted version* of the simple puzzle of Figure 3 (see Eqn. (7)), where piece C is restricted to one of its four rotations, viz. embeddings 18 and 20 in Figure 4. The order in which we attempt to cover the 9 aspects of the puzzle is

$$0, 3, 1, 4, 2, 5, \mathsf{A}, \mathsf{B}, \mathsf{C}. \tag{36}$$

When aspects 0 through 5 have been covered, it is also known that aspects A, B, and C have been covered. Consequently, the procedures $Solve_a$ corresponding to A, B, and C can be omitted and the program consists of the following seven $Solve_a$ procedures:

Procedure	**Deals with**
`Solve_0`	Embeddings 0, 6, 7
`Solve_3`	Embeddings 3, 11, 18
`Solve_1`	Embeddings 1, 8, 9
`Solve_4`	Embeddings 4, 12, 20
`Solve_2`	Embeddings 2, 10
`Solve_5`	Embedding 5
`Solve_ZZ`	A solution ($a = \infty$)

Within these procedures, overlap among embeddings has been exploited. For example, the precondition of procedure $Solve_1$ is that aspects 0 and 3 have already been covered. If aspect 1 is still free, it might be covered by each of the embeddings 1, 8, and 9. These have aspect 1 in common (of course), and embeddings 8 and 9 have aspect B in common. This is reflected by the following body of $Solve_1$ (expressed in the C programming language):

```
if ( q[1] ) { q[1] = 0;
  if ( q[A] ) { q[A] = 0;   /* embedding 1 placed */
    Solve_4 ( );
    q[A] = 1; }
  if ( q[B] ) { q[B] = 0;
    if ( q[2] ) { q[2] = 0; /* embedding 8 placed */
      Solve_4 ( );
      q[2] = 1; }
    if ( q[4] ) { q[4] = 0; /* embedding 9 placed */
      Solve_4 ( );          /* could be optimized to Solve_2 ( ); */
      q[4] = 1; }
    q[B] = 1; }
  q[1] = 1; }
else Solve_4 ( );
```

When introducing appropriate abbreviations `IF` (*If Free*), `SOLVE`, and `MF` (*Make Free*) to reduce clutter, this can be written concisely as:

```
#define IF(a)           if ( q[a] ) { q[a] = 0;
#define SOLVE(e,a)      /* embedding e placed */ Solve_##a ( );
#define MF(a)           q[a] = 1; }

IF ( 1 )
  IF ( A ) /* embedding 1 placed */
    SOLVE ( 1, 4 )
    MF ( A )
  IF ( B )
    IF ( 2 ) /* embedding 8 placed */
      SOLVE ( 8, 4 )
      MF ( 2 )
    IF ( 4 ) /* embedding 9 placed */
      SOLVE ( 9, 4 )
      MF ( 4 )
    MF ( B )
  MF ( 1 )
else Solve_4 ( );
```

A complete program is shown in Appendix A. This program produces the following output (also see Fig. 6):

```
Solving simple puzzle (modulo symmetry)
Solution    1:  0 18 10
Solution    2:  6  3 20
Solution    3:  7  1 20
Number of solutions = 3
```

In view of the four-fold symmetry of the box, there are 12 solutions altogether.

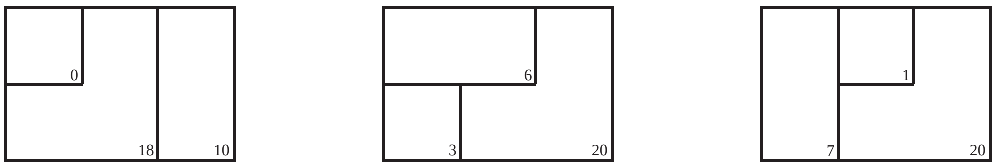

Figure 6: *The three solutions (modulo symmetry) for the simple puzzle of Fig. 3*

One can also consider the program that does not exploit overlap, and the programs based on a different order, viz. A, B, C (programs not shown). For each of these four programs, Table 2 shows how often each of the instructions IF, MF, and SOLVE occurs in the program text. Obviously, the counts for IF and MF are equal, and the count for SOLVE equals the number of embeddings in the puzzle (ignoring the 'startup' call). Furthermore, when not exploiting overlap, the order does not matter for the number of IF instructions, because each aspect of each embedding is tested separately ($41 = 6 * 2 + 7 * 3 + 2 * 4$).

For each of the four programs, Table 3 shows how many if statements were executed (IF), how many times each if condition occurred, and how many embeddings were placed (calls to SOLVE). Obviously, the number of MF executions equals the number of true IF-conditions. Concerning the SOLVE count (number of embeddings placed), observe that this is at least 9, because there are 3 solutions, each involving 3 embeddings without any common embeddings. Thus, the SOLVE count of 10 in the first two columns, indicates that only one embedding was placed in vain (without

	order 0, 3, 1, 4, 2, 5, . . .		**order A, B, C,. . .**	
Instruction	overlap exploited	overlap not exploited	overlap exploited	overlap not exploited
`IF`	28	41	24	41
`MF`	28	41	24	41
`SOLVE`	15	15	15	15

Table 2: *Instruction counts (static) for four program versions*

	order 0, 3, 1, 4, 2, 5, . . .		**order A, B, C,. . .**	
Instruction	overlap exploited	overlap not exploited	overlap exploited	overlap not exploited
`IF`	47	79	154	275
`IF` true	27	47	107	208
`IF` false	20	32	47	67
`MF`	27	47	107	208
`SOLVE`	10	10	37	37

Table 3: *Execution counts (dynamic) for four program versions*

leading to a solution), viz. embedding 11 when 0 had already been placed (also see branch 0, 11 in Fig. 7).

Note that, when exploiting overlap, the static counts for the order A, B, C are somewhat smaller than the counts for the order 0, 3, 1, 4, 2, 5, but that their dynamic counts are considerably larger. See Figures 7 and 8 for the (dynamic) search trees induced by the two orders we have considered. Note the very different shapes and sizes. In these figures, the tree nodes show q. The bold-framed aspect is selected to be covered in all possible ways. Each possibility corresponds to a branch, which is labeled by the number of the embedding. The light-shaded aspects represent the embedding placed on the incoming branch. The dark-shaded aspects were already covered earlier. A cross marks a conflict which prohibits placement. A smiley indicates a solution.

5 Refinement toward hardware

We now have a program that works on one global boolean array variable q, and incorporates the puzzle's set of embeddings as constants. The entire program for solving the puzzle can be decomposed into just five simple operations:

1. **if** $q[a]$ **then** $q[a]$:= *false* ; . . . **fi** , abbreviated `IF`(a)
2. **call** *Solve*$_a$, abbreviated `SOLVE`$(\ldots, a)$
3. $q[a]$:= *true* , abbreviated `MF`(a)
4. **return**
5. *Solution*

Instead of solving puzzles by a general-purpose processor with a large instruction set, we aim at using a special-purpose **puzzle processor** with a minimal instruction set. We hope that this is more

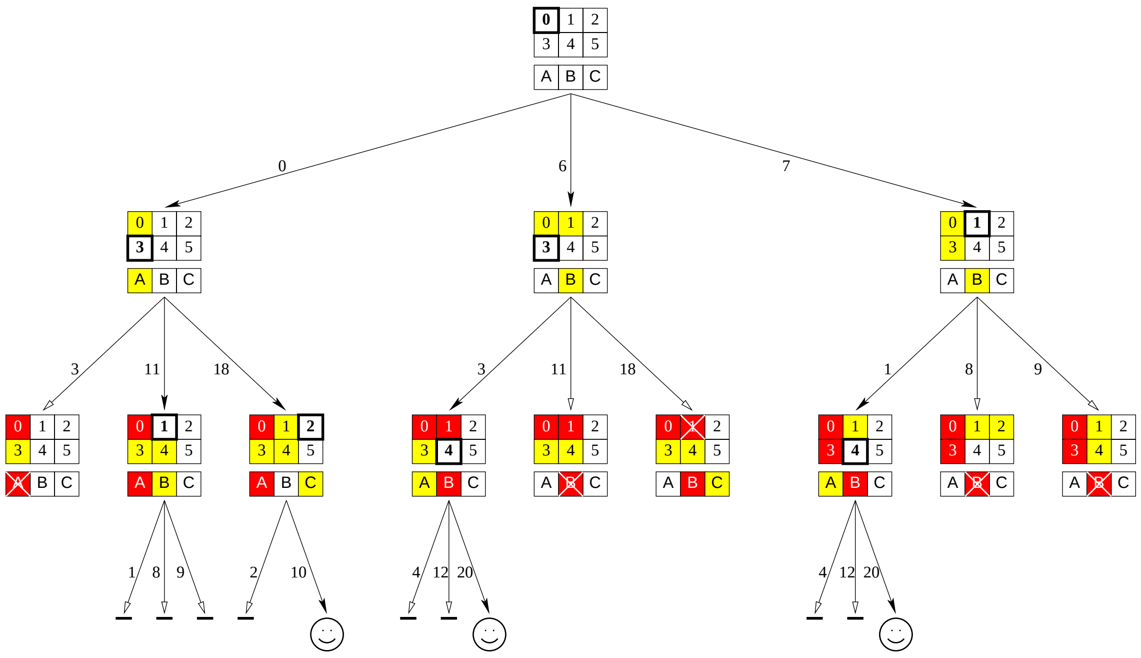

Figure 7: *Search tree for restricted simple puzzle, induced by the order* 0, 3, 1, 4, 2, 5, . . .

effective, because such dedicated processors can be much smaller and faster. Moreover, many such processors can operate in parallel, each processor exploring a separate part of the search space.

5.1 Instruction set

The envisioned puzzle processor has a q-register, a program counter *pc*, a small stack, a stack pointer *sp*, and possibly some resources for processing solutions (such as a counter). The instruction set is shown in Table 4.

Instruction	Operands and operation
`IF`	*aspect* to test-and-cover, *relative address* to jump to if aspect not free
`SOLVE`	*relative address* of routine to call, push current address on stack
`MF`	*aspect* to make free
`RETURN`	pop address from stack and make current
`SOLUTION`	process solution encoded on stack

Table 4: *The five instructions with their operands and operation*

In Appendix B, we have listed an 'assembly' version of the program for the simple puzzle from Appendix A. It clearly shows that only five instructions are needed to express the entire program. Execution starts with the first instruction listed, an empty stack, and the q-register initialized to all-1, and ends when `RETURN` is executed on an empty stack.

Appendix C shows a 'dump' of the puzzle-processor machine-code translation of the programs in the preceding appendices. To improve human readability,

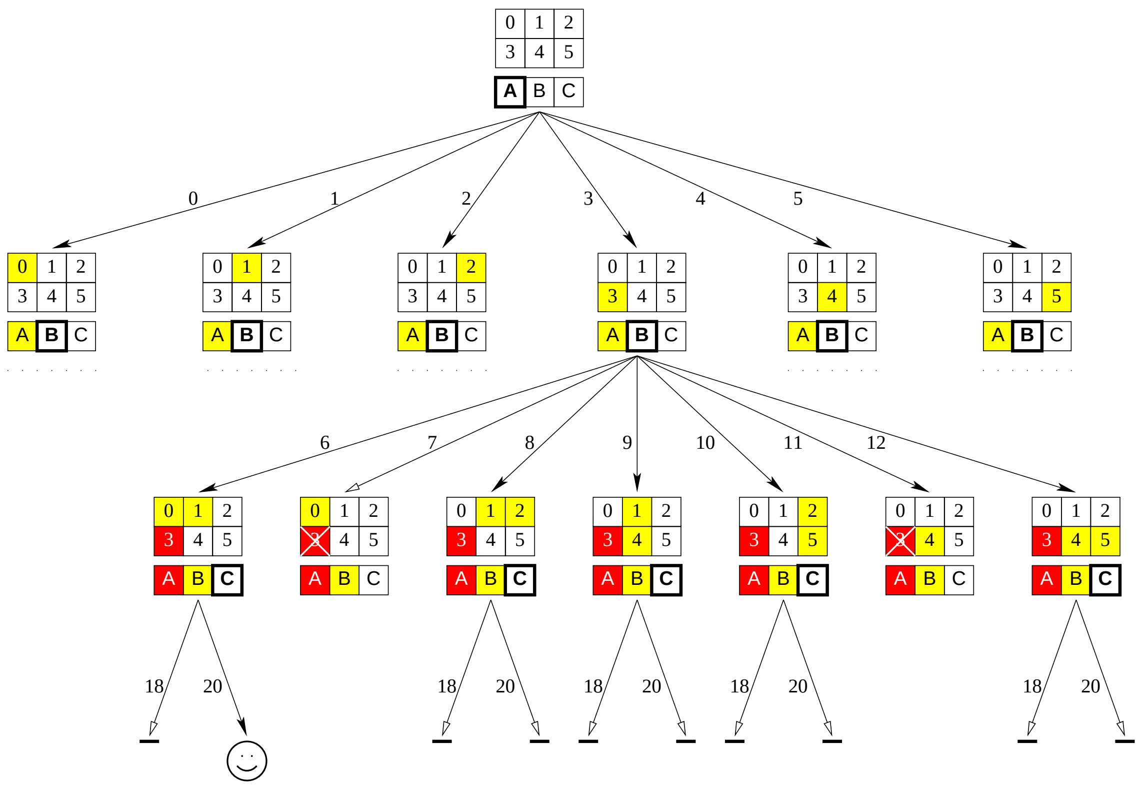

Figure 8: *Part of the search tree induced by the order A, B, C, . . .*

- opcodes are represented by a single character and operands by decimal integers,
- instructions are indented corresponding to the abstract program,
- the listing is printed in multiple columns, and
- each instruction is numbered (you can view this as the address).

In Appendix D we have listed an interpreter —written in C— for the puzzle-processor instruction set. It can be thought of as defining the semantics of the instruction set. This interpreter reads in a machine-code file in the format of appendix C, that is, a text file with one instruction per line encoded as a single-character opcode followed by its operands in decimal. The program is loaded into memory and executed. This version of the interpreter only counts solutions; it does not print them.

5.2 Encoding instructions

Let us consider some hardware-related issues, such as how to encode the instructions listed in Table 4.

Operation code: 3 bits At least three bits are needed for encoding the five operations. We propose the following systematic encoding (opcodes):

`IF`	111
`CALL`	101
`MF`	110
`RETURN`	001
`SOLUTION`	000

The first bit indicates whether or not the operation has any operands. The second bit indicates whether there is an aspect as operand. The third bit indicates whether there is an address as operand, except when there are no operands, then the third bit distinguishes between `RETURN` and `SOLUTION`. Another encoding may be chosen if that would simplify the decoding hardware.

Aspect operand: 7 bits The Pentomino puzzle has 72 aspects (60 cells and 12 pieces). Thus, its aspects can be encoded in seven bits. Several other challenging puzzles (25 Y-Pentomino, 25 N-Pentomino, Hollow Pyramid [15]) have no more than 128 aspects.

Address operand: 14 bits The range of addresses needed for a program can be quite large. For instance, the program of Appendix A for the simple puzzle of Figure 3 —when translated into puzzle-processor code— consists already of 79 instructions. It is listed in Appendix C. For the Pentomino puzzle, the number of instructions exceeds $2^{14} = 16,384$. However, when relative addresses are used, the Pentomino puzzle and others can easily be coded with 14-bit addresses.

opcode	*aspect*	*relative address*
3 bit	7 bit	14 bit

Table 5: *Instruction layout*

This instruction encoding for the puzzle-processor requires $3 + 7 + 14 = 24$ bits, which looks like a fair number. Also see Table 5. There are many other possibilities for designing the instruction set. We have just chosen one and used it for further evaluation.

5.3 Encoding sets of aspects

A subset v of the set of aspects A can be encoded in several ways. One way is a bit vector (boolean array, characteristic function), using one bit per aspect. Assuming $\#A = 2^n$, this requires 2^n bits. Another way is to enumerate the aspects in subset v. Assuming $\#v = k$, this requires $k * n$ bits, plus possibly some overhead for 'gluing' the aspects together. The enumeration is more memory efficient when $k * n < 2^n$. In case of $n = 7$, as we have chosen, this means that subsets with fewer than $2^7/7 \approx 18$ aspects are more efficiently enumerated, whereas bigger subsets are best represented as bit vectors.

The embeddings are subsets of A that are usually rather sparse, that is with small $\#v = k$. For the simple puzzle (Fig. 3), k ranges from 2 to 4, and for the Pentomino puzzle $k = 6$. Thus, embeddings are indeed best represented by enumeration.

Global variable q is also a subset of A (viz. of uncovered aspects). In contrast to embeddings, the size of q varies from full (initially) to empty (solution). Thus, q is indeed best represented by a bit vector.

5.4 VLSI Programming

In the next section, we present several hardware designs for a puzzle processor using the instruction set of the preceding subsection. These designs are derived from the interpreter of Appendix D.

The hardware designs are expressed in the *Tangram* formalism [2]. *Tangram* is a VLSI-programming language developed at the Philips Research Laboratories in Eindhoven. *Tangram* programs are expressed in a language based on Hoare's *CSP* [9] and Dijkstra's *Guarded Command Language* [4]. The main *Tangram* tools [10] are a compiler, an analyzer, a simulator, and a viewer for the output of the simulator. The compiler translates a *Tangram* program via a so-called handshake circuit into an asynchronous VLSI circuit. See [5] for the application of Tangram to processor design.

The designs were first run on the simple test program in Table 6. This program executes each type of instruction at least once, including both directions of the conditional branch instruction (see Table 7).

```
0: I 1 5      ; should NOT take branch
1:   I 1 1    ; should take branch to 3
2:     R      ; should NOT be executed
3:   M 1      ; should execute
4: I 1 1      ; should NOT take branch
5:   C 1      ; should call 7
6: R          ; should terminate program
7: S          ; should increment nos
8: R          ; should return to 6
```

Table 6: Listing of simple test program

Values	Time Slot →							
Register ↓	0	1	2	3	4	5	6	7
pc	0	1	3	4	5	7	8	6
sp	1	=	=	=	=	2	=	1
nos	0	=	=	=	=	=	1	=
oc	7	7	6	7	5	0	1	1
asp	1	1	1	1	0	0	0	0
adr	5	1	0	1	1	0	0	0
Instruction	I	I	M	I	C	S	R	R

Table 7: Execution trace for test program, showing register values (= indicates no access)

The designs presented in the next section were also benchmarked on some small puzzles to compare various characteristics of the designs. Our primary aim has been to optimize for speed, whereas area and power consumption only receive secondary attention. For the benchmarks we have used two small puzzles:

3×4 **domino puzzle** A rectangular box of 3×4 cells has to be filled with 6 dominoes. A domino covers 2 adjacent cells. See Figure 9. Since there is only one kind of piece, it need not be included as an aspect of the puzzle. Therefore, this puzzle has $3 * 4 = 12$ aspects and

17 embeddings ($3 * 3$ horizontal, $2 * 4$ vertical). The puzzle has 11 solutions (5 modulo rotation and reflection; these are easy to find by hand).

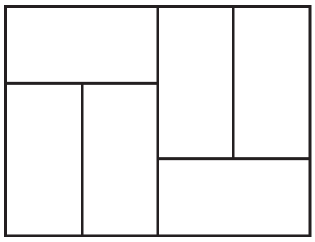

Figure 9: *A solution for the* 3×4 *domino puzzle*

dodecahedron-domino puzzle A dodecahedron (platonic solid with 12 regular pentagons as faces) has to be covered with 6 dominoes. A domino covers two adjacent faces. Since there is only one kind of piece, there are 12 aspects. Each embedding corresponds to an edge. Hence, there are 30 embeddings. The puzzle has 125 solutions (only 5 modulo rotation and reflection; these are quite hard to find by hand).

For the Pentomino puzzle (with 6×10 box), only one simulation was performed, because of the large amount of simulation time needed for this puzzle.

6 Tangram Designs

The designs that we present in this section are all based on a common exo-architecture (everything visible across the external processor interface, including the instruction set down to the bit level). They not only execute exactly the same instruction set, but they also involve the same initialization and finalization phases:

At startup, the processor reads initial values from channel *StatesFile* for variables q, *pc*, *nos* (number of solutions), *sp*, and the stack contents itself. This provides the means to start the program in an arbitrary state, which can be useful both for testing manufactured processor cores and for exploring part of a puzzle's search tree.

Upon termination, the processor writes the value of *nos* to channel *STACKout*.

Solutions are only counted, but they (that is, the contents of the stack) could easily be output along channel *STACKout*, as the name already suggests.

The program, consisting of a sequence of puzzle-processor instructions, is assumed to be loaded in a ROM named *InstrROM*. In simulations, the ROM is initialized from a file called `in.rom`.

The stack and program memory for the puzzle processor are chosen on-chip, because they are not so big and this allows fastest operation.

6.1 Straightforward implementation

Appendix E shows a straightforward Tangram implementation of the puzzle processor. Some trivial parallelism has been introduced: the updates to the various pieces of state are done in parallel (q, *pc*, *sp*, and *stack*).

The size of this design is 1335 gate equivalents (excluding ROM).

6.2 Procedures and precomputed values

Appendix F shows an implementation where various possible new values of the program counter *pc* and stack pointer *sp* are precomputed during the instruction fetch. For that purpose, fresh variables *pc1*, *pc2*, *sp1*, *sp2*, and *sp3* have been introduced. The idea is

- that these values can be computed while fetching the next instruction, without time penalty, and
- that these values can be used to update the state during execution in less time (e.g. fewer sequential steps, i.e. semicolons).

Only some of the precomputed values will actually be used. Thus, there is a nonzero power cost. The goal is to reduce area by sharing more hardware. Since new variables add overhead, one needs to verify afterwards what the net gain is.

Another refinement is that computation of the new address when jumping ($pc := pc + adr$) is carried out in the shared procedure *pcadr*.

Compared to the previous design,

- the size is reduced by 7%, to 1235 gate equivalents (excluding ROM),
- the speed is improved by over 20%, and
- power consumption goes up by more than 25%.

6.3 Prefetching

Appendix G shows an implementation with a simple form of prefetching. During the execution phase, also the instruction at address $pc + 1$ is fetched. Sometimes, but not always (roughly 50% of the time), this is indeed the next instruction to be executed. Speed is improved if prefetching during execution incurs no overhead and the normal fetch phase is not slowed down too much. Note that the normal fetch phase now also needs to check whether or not to use the prefetched instruction, and if so, copy it locally.

Compared to the previous design, the prefetching design

- is 33% larger (viz. 1644 gate equivalents),
- is 35% slower, and
- consumes 12% more energy.

Note that prefetching incurs an overhead, both in area and in speed. Whether there is a net gain in speed depends on how often during execution the prefetched instruction can actually be used. Note that the programs for solving puzzles derived in Section 4 have a high density of branch instructions (static characteristic). Furthermore, the conditions in the branch instructions are not very skewed, but rather evenly divided between true and false (dynamic characteristic). Thus, little gain was to be expected. Still we do not completely understand why the prefetching design is so much slower.

7 Conclusion

We have presented a systematic transformation from general-purpose backtrack programs for solving packing puzzles to puzzle-specific programs for a special-purpose puzzle processor. This puzzle-processor has five simple instructions operating on a single bit-vector register, a solution counter, a program counter, a small stack, and a stack pointer. Three Tangram designs for such a puzzle processor with 24-bit instructions have been compared. Our main aim has been to improve the speed of the computations.

The transformation has been explained in small steps, such that each step can be formally verified. The transformation steps resemble those often encountered in the optimization of embedded software. The final transformed program can be automatically generated from a description of the puzzle.

There are many ways to define an instruction set for a puzzle processor. We have pursued only one particular choice as a preliminary investigation.

We have compared three simple Tangram implementations of the puzzle processor. The high branching density of typical programs for the puzzle processor, together with low branching predictability, pose a challenge for efficient pipelining, which we have not attempted to tackle in this report.

Future research will also look into the possibility of operating many puzzle processors in parallel to improve performance further.

Acknowledgments

We would like to acknowledge Marc Peters and Ad Peeters for their help with the Tangram designs and for reviewing a preliminary version of this report.

References

[1] Bálint, Z., *Puzzle Processor Project*, Master's Thesis, Department of Computer Science, Eötvös Lórand University, Budapest, 2000

[2] van Berkel, Kees, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, Cambridge Univ. Press, 1993

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

[4] Dijkstra, Edsger W., *A Discipline of Programming*, Addison-Wesley, 1976

[5] van Gageldonk, Hans, *An Asynchronous Low-Power 80C51 Microcontroller*, Dissertation, Eindhoven University of Technology, Department of Computing Science, 1998

[6] Garey, Michael and Johnson, David, *Computers and Intractability*, Freeman, 1979

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

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

[9] Hoare, C. A. R, *Communicating Sequental Processes*, Prentice Hall, 1985

[10] Kessels, Joep and Peeters, Ad, "The Tangram Framework: Asynchronous Circuits for Low Power", In *Proc. of Asia and South Pacific Design Automation Conference*, pp. 255–260, Feb. 2001

[11] Kreher, Donald L. and Stinson, Douglas R., *Combinatorial Algorithms: Generation, Enumeration, and Search*, CRC Press, 1999

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

[13] Skiena, Steven S., *The Algortihm Design Manual*, Springer, 1998

[14] 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

[15] Verhoeff, K. and Verhoeff, T., Hollow Pyramid Puzzle, *CFF: Cubism For Fun, Newsletter of NKC, the Dutch Cubists Club*, Nr. 31, pp. 15–16, 1991

A Program in C for Simple Puzzle

```
/* Solve simple puzzle using order 0,3,1,4,2,5,... and exploiting overlap */

#include <stdio.h>

int q[9];   /* bag of free aspects */
int ns = 0; /* number of solutions */
int sp = 0; /* stack pointer */
int st[3];  /* stack */

#define A 6
#define B 7
#define C 8
#define IF(a) if ( q[a] ) { q[a] = 0;
#define SOLVE(e,a) st[sp++]=e; Solve_##a ( ); --sp;
#define MF(a) q[a] = 1; }

int Solve_0 ( void ) /* cover all, starting at aspect 0 */
  /* not ‘void Solve_i ( void )’ to avoid need for forward declarations */
{
  IF ( 0 )
    IF ( A ) /* embedding 0 placed */
      SOLVE ( 0, 3 )
      MF ( A )
    IF ( B )
      IF ( 1 ) /* embedding 6 placed */
        SOLVE ( 6, 3 )
        MF ( 1 )
      IF ( 3 ) /* embedding 7 placed */
        SOLVE ( 7, 3 )
        MF ( 3 )
      MF ( B )
    MF ( 0 )
  else Solve_3 ( );
}

int Solve_3 ( void ) /* assuming 0 covered, cover rest starting at 3 */
{
  IF ( 3 )
    IF ( A ) /* embedding 3 placed */
      SOLVE ( 3, 1 )
      MF ( A )
    IF ( 4 )
      IF ( B ) /* embedding 11 placed */
        SOLVE ( 11, 1 )
        MF ( B )
      IF ( 1 )
        IF ( C ) /* embedding 18 placed */
          SOLVE ( 18, 1 )
          MF ( C )
        MF ( 1 )
      MF ( 4 )
    MF ( 3 )
  else Solve_1 ( );
}

int Solve_1 ( void ) /* assuming 0,3 covered, cover rest starting at 1 */
{
  IF ( 1 )
     IF ( A ) /* embedding 1 placed */
      SOLVE ( 1, 4 )
      MF ( A )
    IF ( B )
      IF ( 2 ) /* embedding 8 placed */
        SOLVE ( 8, 4 )
        MF ( 2 )
      IF ( 4 ) /* embedding 9 placed */
        SOLVE ( 9, 4 )
        MF ( 4 )
      MF ( B )
    MF ( 1 )
  else Solve_4 ( );
}

int Solve_4 ( void ) /* assuming 0,3,1 covered, cover rest starting at 4 */
{
  IF ( 4 )
    IF ( A ) /* embedding 4 placed */
      SOLVE ( 4, 2 )
      MF ( A )
    IF ( 5 )
      IF ( B ) /* embedding 12 placed */
        SOLVE ( 12, 2 )
        MF ( B )
      IF ( 2 )
        IF ( C ) /* embedding 20 placed */
          SOLVE ( 20, 2 )
          MF ( C )
        MF ( 2 )
      MF ( 5 )
    MF ( 4 )
  else Solve_2 ( );
}

int Solve_2 ( void ) /* assuming 0,3,1,4 covered, cover rest starting at 2 */
{
  IF ( 2 )
    IF ( A ) /* embedding 2 placed */
      SOLVE ( 2, 5 )
      MF ( A )
    IF ( B )
      IF ( 5 ) /* embedding 10 placed */
        SOLVE ( 10, 5 )
        MF ( 5 )
      MF ( B )
    MF ( 2 )
  else Solve_5 ( );
}

int Solve_5 ( void ) /* assuming 0,3,1,4,2 covered, cover rest starting at 5 */
{
  IF ( 5 )
    IF ( A ) /* embedding 5 placed */
      SOLVE ( 5, ZZ )
      MF ( A )
    MF ( 5 )
  else Solve_ZZ ( );
```

```
}

int Solve_ZZ ( void )
{
  int i;
  printf ( "Solution %4d:", ++ns );
  for ( i=0; i!=3; ++i ) { printf ( " %2d", st[i] ); }
  printf ( "\n" ); fflush ( stdout );
}

int main ( void )
{
  int i;
  printf ( "Solving simple puzzle (modulo symmetry)\n" );
  for ( i=0; i!=9; ++i ) { q[i] = 1; } /* all aspects are free once */
  Solve_0 ( );
  printf ( "Number of solutions = %d\n", ns );
}
```

B Puzzle-Processor ‘Assembly Listing’ for Simple Puzzle

```
Solve_0: /* cover all, starting at aspect 0 */
  IF ( 0 )
    IF ( A ) /* embedding 0 placed */
      CALL Solve_3
      MF ( A )
    IF ( B )
      IF ( 1 ) /* embedding 6 placed */
        CALL Solve_3
        MF ( 1 )
      IF ( 3 ) /* embedding 7 placed */
        CALL Solve_3
        MF ( 3 )
      MF ( B )
    MF ( 0 )
    RETURN
/* else fall through ... */

Solve_3: /* assuming 0 covered, cover rest starting at 3 */
  IF ( 3 )
    IF ( A ) /* embedding 3 placed */
      CALL Solve_1
      MF ( A )
    IF ( 4 )
      IF ( B ) /* embedding 11 placed */
        CALL Solve_1
        MF ( B )
      IF ( 1 )
        IF ( C ) /* embedding 18 placed */
          CALL Solve_1
          MF ( C )
        MF ( 1 )
      MF ( 4 )
    MF ( 3 )
    RETURN
/* else fall through ... */
```

```
Solve_1: /* assuming 0,3 covered, cover rest starting at 1 */
  IF ( 1 )
     IF ( A ) /* embedding 1 placed */
      CALL Solve_4
      MF ( A )
    IF ( B )
      IF ( 2 ) /* embedding 8 placed */
        CALL Solve_4
        MF ( 2 )
      IF ( 4 ) /* embedding 9 placed */
        CALL Solve_4
        MF ( 4 )
      MF ( B )
    MF ( 1 )
    RETURN
/* else fall through ... */

Solve_4: /* assuming 0,3,1 covered, cover rest starting at 4 */
  IF ( 4 )
    IF ( A ) /* embedding 4 placed */
      CALL Solve_2
      MF ( A )
    IF ( 5 )
      IF ( B ) /* embedding 12 placed */
        CALL Solve_2
        MF ( B )
      IF ( 2 )
        IF ( C ) /* embedding 20 placed */
          CALL Solve_2
          MF ( C )
        MF ( 2 )
      MF ( 5 )
    MF ( 4 )
    RETURN
/* else fall through ... */

Solve_2: /* assuming 0,3,1,4 covered, cover rest starting at 2 */
  IF ( 2 )
    IF ( A ) /* embedding 2 placed */
      CALL Solve_5
      MF ( A )
    IF ( B )
      IF ( 5 ) /* embedding 10 placed */
        CALL Solve_5
        MF ( 5 )
      MF ( B )
    MF ( 2 )
    RETURN
/* else fall through ... */

Solve_5: /* assuming 0,3,1,4,2 covered, cover rest starting at 5 */
  IF ( 5 )
    IF ( A ) /* embedding 5 placed */
      CALL Solve_ZZ
      MF ( A )
    MF ( 5 )
    RETURN
/* else fall through ... */
```

```
Solve_ZZ:
  SOLUTION
  RETURN
```

C Puzzle-Processor ‘Machine Code’ for Simple Puzzle

```
 1:  I 0 13       21:        C 9    41:        M 4    61:  I 2 10
 2:    I 6 2      22:        M 7    42:      M 7      62:    I 6 2
 3:      C 11     23:      I 1 4    43:    M 1        63:      C 8
 4:      M 6      24:        I 8 2  44:    R          64:      M 6
 5:    I 7 7      25:          C 5  45:  I 4 15       65:    I 7 5
 6:      I 1 2    26:          M 8  46:    I 6 2      66:      I 5 2
 7:        C 7    27:        M 1    47:      C 13     67:        C 4
 8:        M 1    28:      M 4      48:      M 6      68:        M 5
 9:      I 3 2    29:    M 3        49:    I 5 9      69:      M 7
10:        C 4    30:    R          50:      I 7 2    70:    M 2
11:        M 3    31:  I 1 13       51:        C 9    71:    R
12:      M 7      32:    I 6 2      52:        M 7    72:  I 5 6
13:    M 0        33:      C 11     53:      I 2 4    73:    I 6 2
14:    R          34:      M 6      54:        I 8 2  74:      C 3
15:  I 3 15       35:    I 7 7      55:          C 5  75:      M 6
16:    I 6 2      36:      I 2 2    56:          M 8  76:    M 5
17:      C 13     37:        C 7    57:        M 2    77:    R
18:      M 6      38:        M 2    58:      M 5      78:  S
19:    I 4 9      39:      I 4 2    59:    M 4        79:  R
20:      I 7 2    40:        C 4    60:    R
```

D Puzzle-Processor Interpeter in C

```
#include <stdio.h>

#define MAXASPECT 128
#define MAXADDRESS 16384
#define MAXSP 100

typedef char OpCode;   /* I, M, C, R, S */
typedef int  Aspect;   /* 0 .. MAXASPECT-1, index for q[ ] */
typedef int  Address;  /* 0 .. MAXADDRESS-1, index for mem[ ] */
typedef struct {
    OpCode  oc;  /* operation code */
    Aspect  asp; /* aspect operand */
    Address adr; /* address operand */
    } Instruction;
typedef int  StackAdr; /* 0 .. MAXSP-1, index for stack[ ] */

Instruction mem [ MAXADDRESS ]; /* the instruction memory */
Address     pc;                 /* the program counter */
Address     stack [ MAXSP ];    /* the stack */
StackAdr    sp;                 /* the stack pointer */
int         q [ MAXASPECT ];    /* the free-aspects register */
int         ns;                 /* the number of solutions found */

void load ( void )  /* load puzzle-processor code from stdin into memory */
{
  Instruction inst; /* to capture instruction read from stdin */
  Aspect      a;    /* to initialize q */

  pc = 0;  /* start loading at address 0 */

  while ( scanf ( " %c", & inst . oc ) != EOF ) { /* read opcode from stdin */
    switch ( inst . oc ) { /* collect operands from stdin */
      case ’I’: scanf ( " %d %d", & inst . asp, & inst . adr ); break;
      case ’M’: scanf ( " %d", & inst . asp ); break;
      case ’C’: scanf ( " %d", & inst . adr ); break;
      case ’R’: break;
      case ’S’: break;
      default : printf ( "Illegal instruction ’%c’ @ %d\n", inst . oc, pc );
    }
    mem [ pc ++ ] = inst; /* store instruction and increment pc */
  }

  printf ( "# instructions loaded = %d\n", pc );
  for ( a = 0; a != MAXASPECT; ++ a ) q [ a ] = 1;
}

void run ( void )  /* execute puzzle-processor code in memory */
{
  Instruction inst; /* current instruction */

  pc = 0; sp = 0; ns = 0;

  while ( sp >= 0 ) {
    inst = mem [ ++ pc ];  /* fetch instruction and increment pc */
    switch ( inst . oc ) { /* decode and execute instruction */
      case ’I’: /* IF */
        if ( q [ inst . asp ] ) -- q [ inst . asp ];
        else pc += inst . adr;
        break;
      case ’M’: /* MF */
        ++ q [ inst . asp ];
        break;
      case ’C’: /* CALL */
        stack [ sp ++ ] = pc;
        pc += inst . adr;
        break;
      case ’R’: /* RETURN */
        pc = stack [ -- sp ];
        break;
      case ’S’: /* SOLUTION */
        ++ ns;
        break;
    }
  }

  printf ( "# solutions = %d\n", ns );
}

int main ( int argc, char **argv )
{
  load ( );
  run ( );
  exit ( 0 );
}
```

E Straigthforward design with minimal parallelism

```
  opcode      = type [0..7]
& aspect      = type [0..127]
& stackrange  = type [0..127]
& address     = type [0..16383]
& int         = type [0..65535]
& instruction = type <<opcode,aspect,address>>

& PP : main proc ( StatesFile? chan int & STACKout! chan int ).

begin /* variables */
    q           : ram array aspect of bool
  & done        : var bool := false
  & oc          : var opcode
  & asp         : var aspect
  & adr         : var address
  & pc          : var address ff
  & i           : var aspect ff := 0
  & nos, k      : var int ff
  & sp          : var stackrange ff
  & j           : var stackrange ff := 1
  & stack       : ram array stackrange of address
  & InstrROM    : rom array [0..32767] of instruction := file("in.rom")

| /* initialize processor */
  for 128 do
      StatesFile?k; q[i]:=(k=1); i:=i+1
  od
; StatesFile?k ; pc:=k fit address
; StatesFile?nos
; StatesFile?k ; sp:=k fit stackrange
; do -(j=sp) then
      StatesFile?k ; stack[j]:=k fit address ; j:=j+1
  od
  /* main loop */
; do -done then
     <<oc,asp,adr>> := InstrROM[pc] /* instruction fetch */
   ; case oc     /* decode and execute */
     is 7 then   {IfFree}
            if q[asp] then q[asp]:=false || pc:=pc+1
            else pc:=pc+adr
            fi
     or 6 then   {MakeFree}
            q[asp]:=true || pc:=pc+1
     or 5 then   {Call}
            stack[sp]:=pc+1
          ; sp:=sp+1 || pc:=pc+adr
     or 1 then   {Return}
            sp:=sp-1
          ; pc:=stack[sp] || done:=(sp=0)
     or 0 then   {Solution}
            nos:=nos+1 || pc:=pc+1
     si
  od
  /* output result */
; STACKout!nos
end
```

F Design with procedures and precomputed values

```
  opcode      = type [0..7]
& aspect      = type [0..127]
& stackrange  = type [0..127]
& address     = type [0..16383]
& int         = type [0..65535]
& instruction = type <<opcode,aspect,address>>

& PP : main proc ( StatesFile? chan int & STACKout! chan int ).

begin
  /* variables */
    q               : ram array aspect of bool
  & done            : var bool := false
  & oc              : var opcode
  & asp             : var aspect
  & adr             : var address
  & pc,pc1,pc2      : var address
  & i               : var aspect ff := 0
  & nos,k           : var int ff
  & sp,sp1,sp2,sp3  : var stackrange
  & j               : var stackrange ff := 1
  & stack           : ram array stackrange of address
  & InstrROM        : rom array [0..32767] of instruction := file("in.rom")

  & pcadr: proc(). pc:=pc2+adr

| /* initialize processor */
  for 128 do StatesFile?k ; q[i]:=(k=1) ; i:=i+1 od
; StatesFile?k ; pc:=k fit address
; StatesFile?nos
; StatesFile?k ; sp:=k fit stackrange
; do -(j=sp) then StatesFile?k ; stack[j]:=k fit address ; j:=j+1 od
  /* main loop */
; do -done then
     <<oc,asp,adr>>:=InstrROM[pc] ||
       pc1:=pc+1 || pc2:=pc || sp1:=sp+1 || sp2:=sp-1 || sp3:=sp
   ; case oc
     is 7 then   {IfFree}
            if q[asp] then q[asp]:=false || pc:=pc1
            else pcadr()
            fi
     or 6 then   {MakeFree}
            q[asp]:=true || pc:=pc1
     or 5 then   {RecurseBegin}
            stack[sp3]:=pc1 || sp:=sp1 || pcadr()
     or 1 then   {RecurseEnd}
            sp:=sp2 || pc:=stack[sp2] || done:=(sp2=0)
     or 0 then   {Print}
            nos:=nos+1 || pc:=pc1
     si
  od
  /* output result */
; STACKout!nos
end
```

G Design with prefetching

```
  opcode      = type [0..7]
& aspect      = type [0..127]
& stackrange  = type [0..127]
& address     = type [0..16383]
& int         = type [0..65535]
& instruction = type <<opcode,aspect,address>>

& PP : main proc ( StatesFile? chan int & STACKout! chan int ).

begin
  /* variables */
    q               : ram array aspect of bool
  & done,ok         : var bool := false
  & oc,oc1          : var opcode
  & asp,asp1        : var aspect
  & adr,adr1        : var address
  & pc,pc1,pc2      : var address
  & i               : var aspect ff := 0
  & nos,k           : var int ff
  & sp,sp1,sp2,sp3  : var stackrange
  & j               : var stackrange ff := 1
  & stack           : ram array stackrange of address
  & InstrROM        : rom array [0..32767] of instruction := file("in.rom")

  & pcadr: proc(). pc:=pc2+adr
  & romread: func( p?var address ). InstrROM[p]

| /* initialize processor */
  for 128 do StatesFile?k ; q[i]:=(k=1) ; i:=i+1 od
; StatesFile?k ; pc:=k fit address
; StatesFile?nos
; StatesFile?k ; sp:=k fit stackrange
; do -(j=sp) then StatesFile?k ; stack[j]:=k fit address ; j:=j+1 od
  /* main loop */
  do -done then
     <<oc,asp,adr>> := if ok then <<oc1,asp1,adr1>> else romread ( pc ) fi
       || pc1:=pc+1 || pc2:=pc || sp1:=sp+1 || sp2:=sp-1 || sp3:=sp
  ;  case oc
     is 7 then   {IfFree}
            if q[asp] then q[asp]:=false || pc:=pc1 || ok:=true
            else pcadr() || ok:=false
            fi
     or 6 then   {MakeFree}
            q[asp]:=true || pc:=pc1 || ok:=true
     or 5 then   {RecurseBegin}
            stack[sp3]:=pc1 || sp:=sp1 || pcadr() || ok:=false
     or 1 then   {RecurseEnd}
            sp:=sp2 || pc:=stack[sp2] || done:=(sp2=0) || ok:=false
     or 0 then   {Print}
            nos:=nos+1 || pc:=pc1 || ok:=true
     si
       || <<oc1,asp1,adr1>> := romread ( pc1 )  /* prefetch */
  od
  /* output result */
; STACKout!nos
end
```

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:	P.E. Wierenga	WB-57
	E.P.C. van Utteren	WDC-01
Department Head:	G.F.G. Depovere	WO-01
	E. Dijkstra	WDC-01

Abstract

–	–	–

Full report

Lex Augusteijn	Nat.Lab.	WDC-01
Dennis Alders	Nat.Lab.	WDC-11
Jan van Amstel	Nat.Lab.	WDC-01
Kees van Berkel	Nat.Lab.	WDC-31
Martijn van Balen	Nat.Lab.	WO-01
Ralph Braspenning	Nat.Lab.	WO-01
Richard Doornbos	Nat.Lab.	WY-21
Jos van Eijndhoven	Nat.Lab.	WDC-01
Hans van Gageldonk	Nat.Lab.	WDC-01
Gerben Hekstra	Nat.Lab.	WO-01
Christian Hentschel	Nat.Lab.	WO-01
Paul Hoogendijk	Nat.Lab.	WDC-01
Egbert Jaspers	Nat.Lab.	WO-01
Roos Joordens	Nat.Lab.	WY-01
Piërre van der Laar	Nat.Lab.	WDC-21
Joep Kessels	Nat.Lab.	WDC-31
Willem Mallon	Nat.Lab.	WDC-01
Ad Peeters	Nat.Lab.	WDC-31
Clara Otero-Perez	Nat.Lab.	WDC-31
Marc Peters	Nat.Lab.	WY-03
Evert-Jan Pol	Nat.Lab.	WAY-21
Bram Riemens	Nat.Lab.	WO-01

Martijn Rutten	Nat.Lab.	WDC-31
David Simons	Nat.Lab.	WDC-11
Erik van der Tol	Nat.Lab.	WO-01
Ramses van der Toorn	Nat.Lab.	WAY-41
Joachim Trescher	Nat.Lab.	WDC-01
René v.d. Vleuten	Nat.Lab.	WO-01
Bernard van Vlimmeren	Nat.Lab.	WDC-21
Rik Willems	Nat.Lab.	WDC-11
Pieter van der Wolf	Nat.Lab.	WDC-31
Clemens Wüst	Nat.Lab.	WY-21
Frits Schalij	ED&T	WAY-3
Rik van de Wiel	ED&T	WAY-31
Rob Zijlstra	CIP	WAH-1
Pieter van Dam	ASA Lab.	SFJ-5
Eric Körber	ASA Lab.	SFJ-4
Erik Mallens	ASA Lab.	SFJ-5
Peter Lambooij	TASS	HCZ-1
Marc Willekens	TASS	HCZ-1
Abdelhamid Ualit	PMS	QR