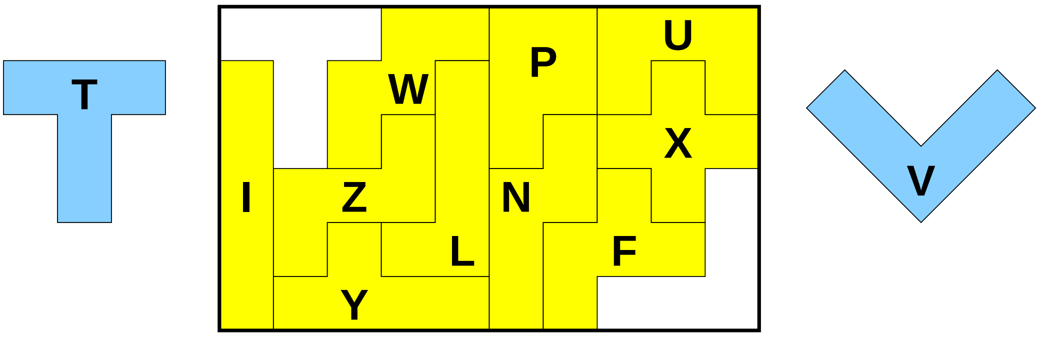
Efficient Solvers for Packing Puzzles

Tom Verhoeff



Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica

Topics

- **Packing Puzzles**:

  Concrete → Abstract

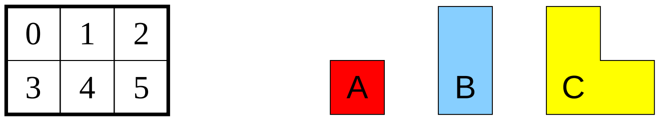
- **Solvers for Packing Puzzles**:

  Data in backtrack program → Program for puzzle processor

- **Results**

Simple Packing Puzzle

Pack the set of **pieces** in the **box**, without overlapping



- **Box** (left): 2×3 unit squares

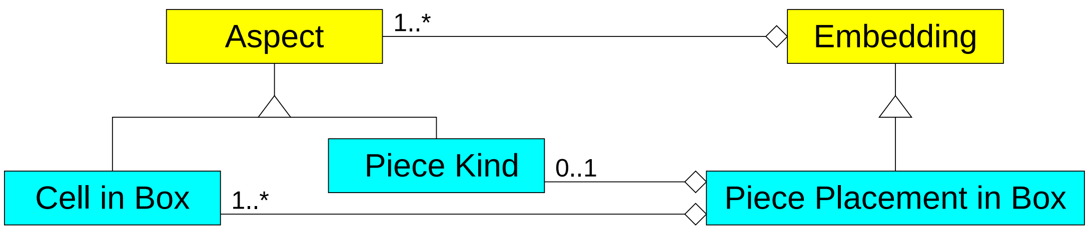
- **Pieces** (right): A, B, C

Aspects and Embeddings

- **Aspect**: Cell in box, or piece kind

- **Embedding**: Placement of piece in box

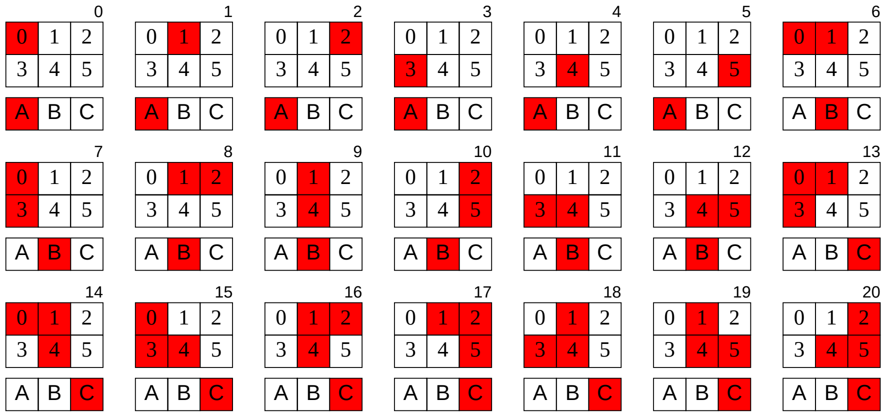
  Can be encoded by *set of aspects*



Aspects and Embeddings for Simple Puzzle

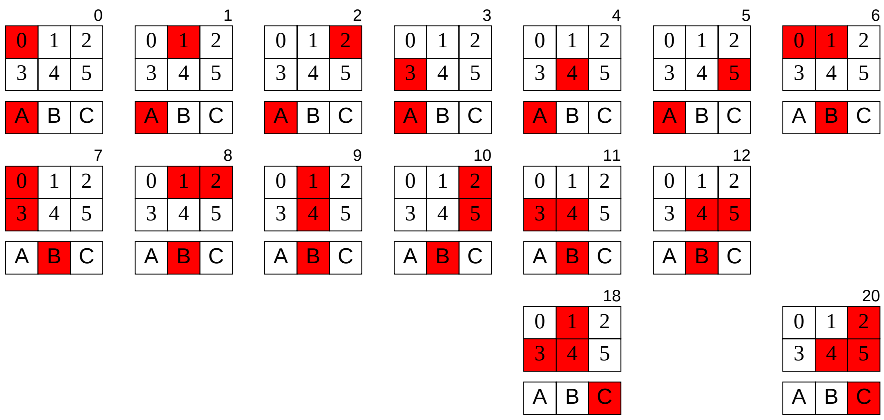
Aspects: $\{0, 1, 2, 3, 4, 5, A, B, C\}$

Embeddings:



Eliminating Symmetries by Restricting Embeddings

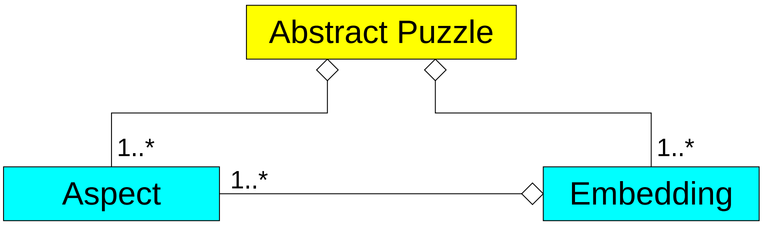
Restrict piece C to one orientation:



Abstract Puzzle

Abstract Puzzle: Pair (A, E) with

- A: Set of **aspects**
- E: Set of **embeddings** with $E \subseteq \mathcal{P}(A) \ \wedge \ E \neq \emptyset \ \wedge \ \emptyset \notin E$



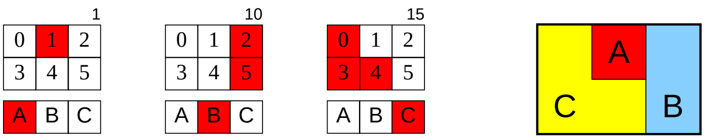
Captures **topology**, ignores **(geo)metric** information

Abstract Puzzle Solution

Solution: Subset s of E that **partitions** A:

Each aspect in A is covered **exactly once** by an embedding in s

Embeddings $\{1, 10, 15\}$ solve the Simple Puzzle:



Difficulty of Solving Abstract Puzzles

Deciding whether an abstract puzzle has a solution is an

NP-complete problem

Number of solutions can be **exponential** in size of puzzle

Solving Abstract Puzzles

Goal: Find **all** solutions

Given an abstract puzzle (A, E),

design procedure $Solve$ that

calls $Solution(s)$ **once** for **every** solution s

Method: **Exhaustive search**

Recursive Solver: Generalized Specification

Partial solution: Subset p of E such that

Each aspect in A is covered **at most once** by an embedding in p

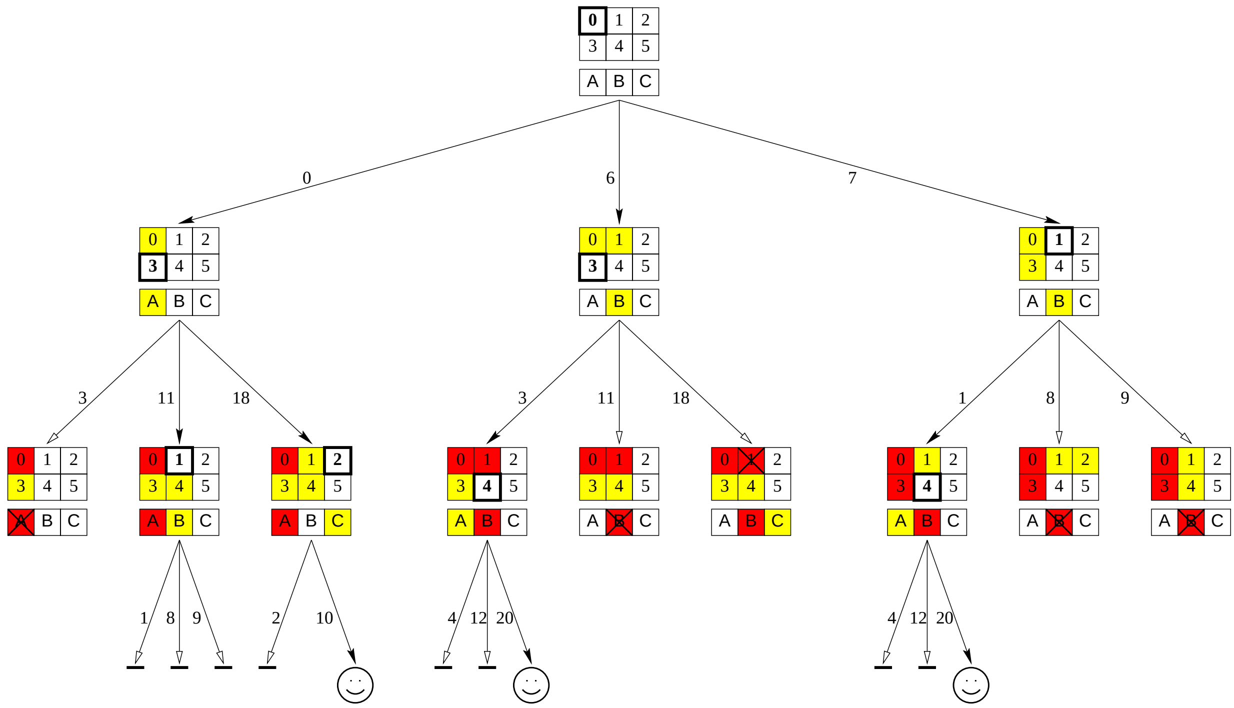
proc *Solve* (p: subset of E)
  { **pre**: p is partial solution
    post: $Solution(s)$ called once for each solution s extending p
  }

Use as: *Solve* ($\emptyset$)

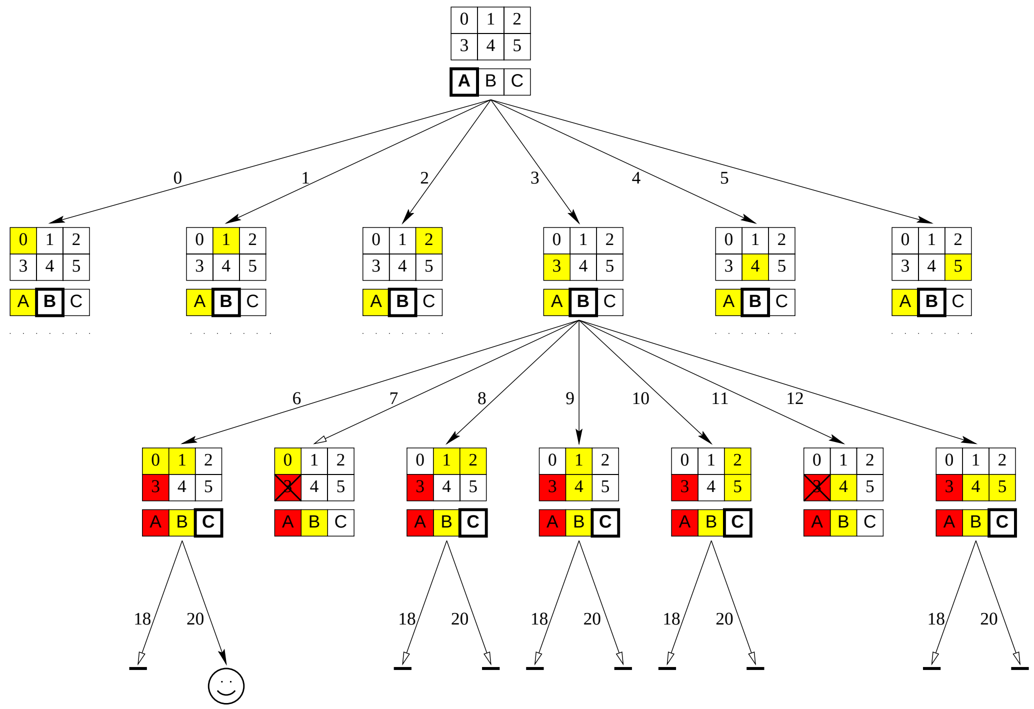
Recursive Solver: Implementation

```
proc Solve ( p: subset of E )
  { vf: number of uncovered aspects: #(A − ⋃p) }
  |[ if ⋃p = A → { p is solution } Solution(p)
     [] ⋃p ≠ A → { there are uncovered aspects }
        |[ var a: A; e: E;
           a :∈ A − ⋃p { a is not covered }
           { cover a in all possible ways }
         ; for e ∈ E with a ∈ e ∧ e ∩ ⋃p = ∅ do
               { p ∪ {e} is partial solution }
               Solve ( p ∪ {e} )
           od
        ]|
     fi
  ]|
```

Search Tree for Order 0, 3, 1, 4, 2, 5, . . .



Part of Search Tree for Order A, B, C, . . .



Choosing Next Aspect to Be Covererd

Refine assignment $a :\in \bigcup p - A$, picking uncovered aspect

- **Static choice** (computed at compile-time):

  Go through aspects in fixed order

  How to choose order?

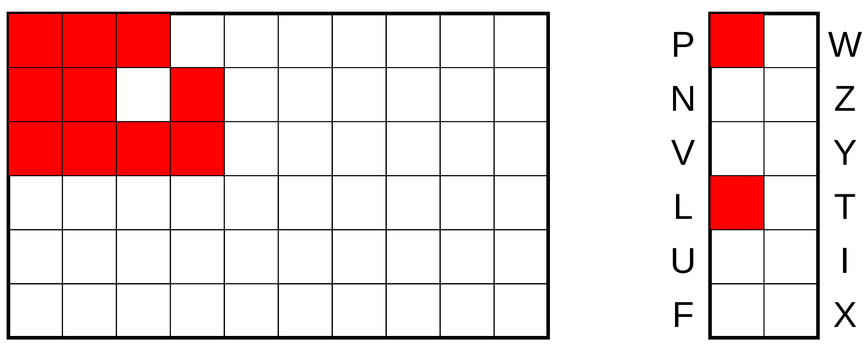
  Least-Growing Footprint

- **Dynamic choice** (computed at run-time):

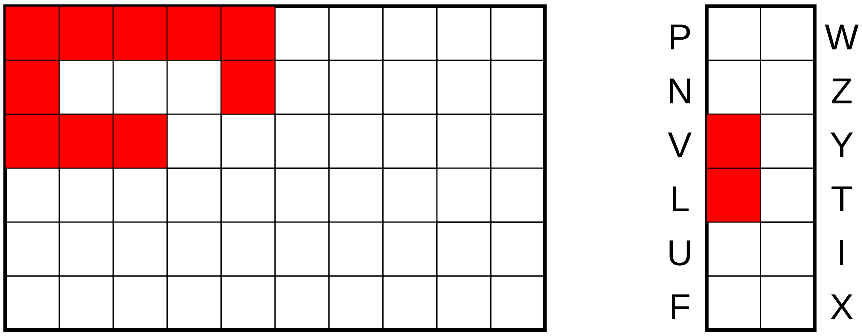
  Zero-fit Cut-Off (ZCO)

  Least-Fit First (LFF)

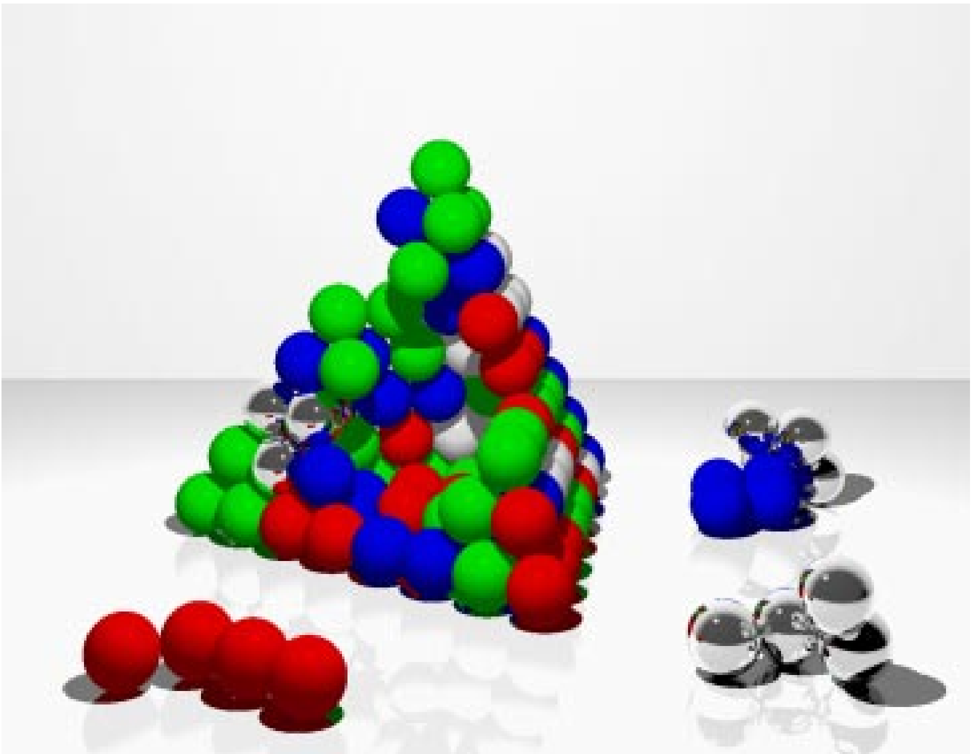
Zero-Fit Cut-Off in 6×10 Pentominoes



Least-Fit First in 6×10 Pentominoes



Hollow Pyramid Puzzle: Koos Verhoeff (198?)



All Fit Counts for Empty 6×10 Pentominoes

37	78	103	111	112					
78	166	219	234	235					
102	218	282	297	298					

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

Aspect-71 Fit Counts for Empty 6×10 Pentominoes

'Footprint' for aspect 71 (piece X):

	1	1	1	1	1	1	1	1	
1	3	4	4	4	4	4	4	3	1
1	4	5	5	5	5	5	5	4	1
1	4	5	5	5	5	5	5	4	1
1	3	4	4	4	4	4	4	3	1
	1	1	1	1	1	1	1	1	

P			W
N			Z
V			Y
L			T
U			I
F		32	X

Fanout: 32 **Total area**: 57

Aspect-0 Fit Counts for Empty 6×10 Pentominoes

‘Footprint’ for aspect 0 (upper lefthand corner cell):

37	23	13	5	1					
23	22	14	4						
13	14	6							
5	4								
1									

P	6	2	W
N	4	4	Z
V	3	4	Y
L	6	2	T
U	4	2	I
F	2		X

Fanout: 37 **Total area**: 26

Recursive Solver (Transforms 1, 2): Specification

Eliminate expression $A - \bigcup p$: introduce parameter $q = A - \bigcup p$

Eliminate parameters p, q: replace by global variables p, q

var p: subset of E; q: subset of A;
{ **inv**: p is partial solution, $q = A - \bigcup p$ (uncovered aspects) }

proc *Solve2* { **glob**: p, q }
{ **pre**: *true*
post: $\mathit{Solution}(s)$ called once for each solution s extending p
$p = \tilde{p} \wedge q = \tilde{q}$ (p, q unchanged)
}

Use as: $p, q := \emptyset, A$; *Solve2*

Recursive Solver (Transforms 1, 2): Implementation

proc $\mathit{Solve2}$ { **glob**: p, q }
  |[**if** $q = \emptyset$ → { p is solution } $\mathit{Solution}(p)$
    [] $q \neq \emptyset$ → { there are uncovered aspects }
      |[**var** a: A; e: E;
        $a :\in q$ { a is not covered }
        { cover a in all possible ways }
       ; **for** $e \in E$ **with** $a \in e \wedge e \subseteq q$ **do**
            { $p \cup \{ e \}$ is partial solution }
            $p, q := p \cup \{ e \}, q - e$
         ; $\mathit{Solve2}$ { acts on p, q }
         ; $p, q := p - \{ e \}, q \cup e$
        od
      ]|
    fi
  ]|

Recursive Solver (Transform 3): Specification

Refine choice $a :\in q$, by introducing

total order $\leq$ on A, and parameter a with $a \leq \min q$

var p: subset of E; q: subset of A;
  { **inv**: p is partial solution, $q = A - \bigcup p$ (uncovered aspects) }

proc $\mathit{Solve3}$ (a: $A \cup \{ \infty \}$) { **glob**: p, q }
  { **pre**: $a \leq \min q$
   post: $\mathit{Solution}(s)$ called once for each solution s extending p
        $p = \tilde{p} \wedge q = \tilde{q}$ (p, q unchanged)
  }

Use as: $p, q := \emptyset, A$; $\mathit{Solve2}$ ($\min A$)

Recursive Solver (Transform 3): Implementation

```
proc Solve3 ( a: A ∪ { ∞ } ) { glob: p, q }
  |[ if a = ∞ → { q = ∅, p is solution } Solution(p)
     [] a ≠ ∞ ∧ a ∉ q → { succ(a) ≤ min q } Solve3(succ(a))
     [] a ∈ q → { a = min q, a is not covered }
        |[ var e: E;
           { cover a in all possible ways }
           for e ∈ E with a ∈ e ∧ e ⊆ q do
                 { p ∪ { e } is partial solution }
                 p, q := p ∪ { e }, q − e
              ;  Solve3 ( succ(a) )
              ;  p, q := p − { e }, q ∪ e
           od
        ]|
     fi
  ]|
```


Further Transformations

4. Instantiate $\mathit{Solve3}(a)$ for each $a \in A \cup \{ \infty \}$

5. Partition E into $E_a = \{ e \mid \min e = a \}$ with $a \in A$

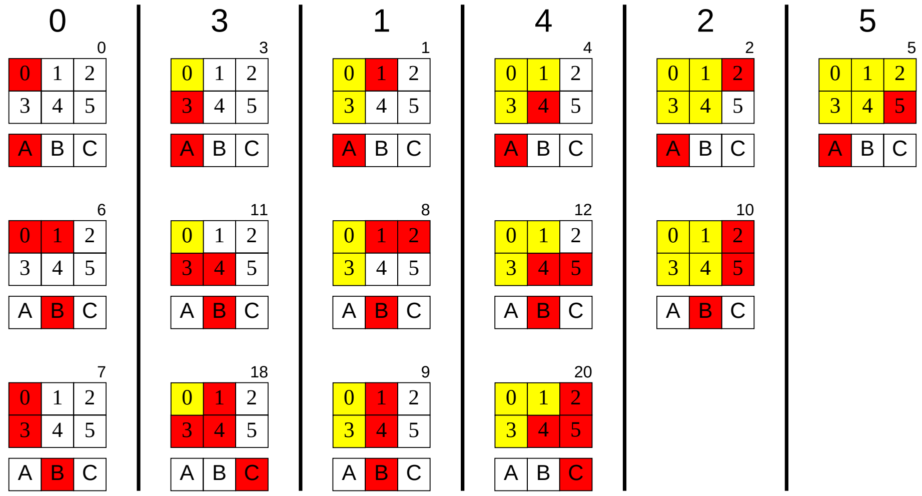
6. Unroll the for-loops over E_a

7. Eliminate p

8. Expand each embedding $e \in E$ into its aspects

9. Exploit overlap among embeddings

Partitioning Example for Simple Puzzle



Transformed Program for Simple Puzzle: `Solve_1`

```
if ( q[1] ) { q[1] = 0;
  if ( q[A] ) { q[A] = 0;    /* embedding 1 = {1,A} placed */
    Solve_4 ( );
    q[A] = 1; }
  if ( q[B] ) { q[B] = 0;
    if ( q[2] ) { q[2] = 0; /* embedding 8 = {1,2,B} placed */
      Solve_4 ( );
      q[2] = 1; }
    if ( q[4] ) { q[4] = 0; /* embedding 9 = {1,4,B} placed */
      Solve_4 ( );          /* can be optimized to Solve_2 */
      q[4] = 1; }
    q[B] = 1; }
  q[1] = 1;
  return; }
Solve_4 ( );                /* optimizable to "fall through" */
```

Instruction Set of Puzzle Processor

Instruction	Operands and Operation
`IF`	*aspect* to test-and-cover: `if ( q[a] ) { q[a] = 0;`
	rel. address to jump to if aspect not free: `... }`
`SOLVE`	*rel. address* of routine to call
	push current address on stack
`MF`	*aspect* to make free: `q[a] = 1;`
`RETURN`	pop address from stack and make current
`SOLUTION`	process solution encoded on stack

opcode	*aspect*	*rel. address*
3 bit	7 bit	14 bit

Puzzle-Processor Program for Simple Puzzle

```
Solve_1: IF   1, Solve_4
         IF   A, L1_0       ; embedding 1 = {1,A} placed
         CALL Solve_4
         MF   A
L1_0:    IF   B, L1_1
         IF   2, L1_2       ; embedding 8 = {1,2,B} placed
         CALL Solve_4
         MF   2
L1_2:    IF   4, L1_3       ; embedding 9 = {1,4,B} placed
         CALL Solve_2
         MF   4
L1_3:    MF   B
L1_1:    MF   1
         RETURN
Solve_4: ...
```

Packing Puzzles in Benchmark

- **Pentominoes** (12 pieces of **5 squares/cubes** each)

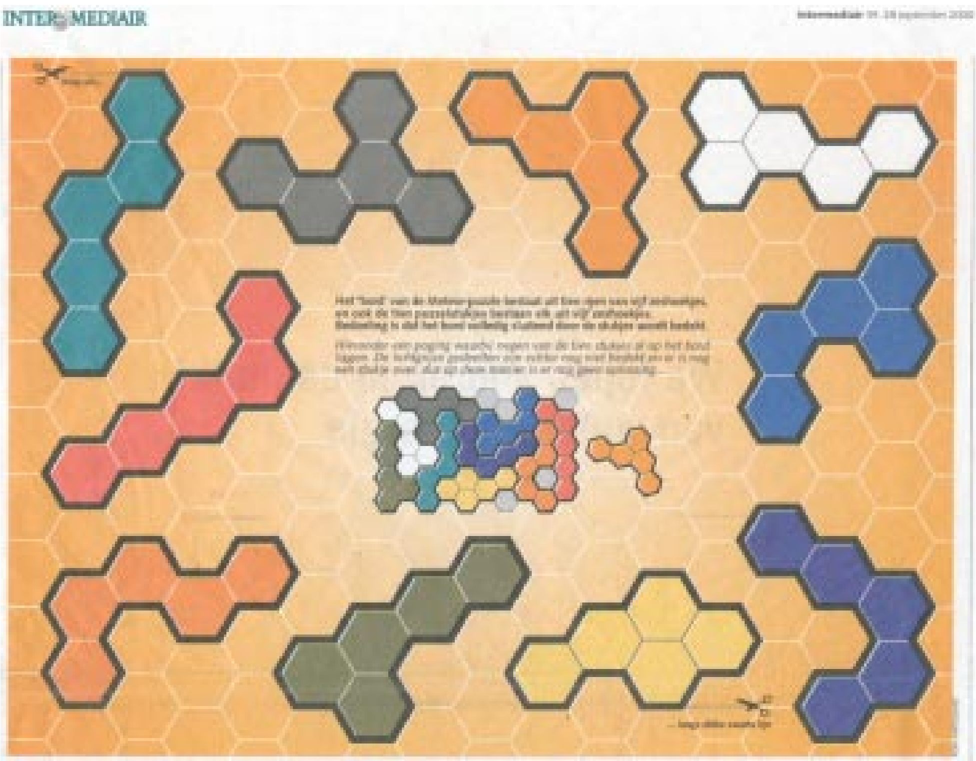
  2D box: 6×10; 3D box: $3 \times 4 \times 5$

  Variants: 25 Y in $5 \times 5 \times 5$, 25 N in $5 \times 5 \times 5$

- **Hollow Pyramid** (25 pieces of **4 spheres** each)

- **Meteor** (10 pieces of **5 hexagons** each)

Meteor Puzzle: Christopher Monckton (1999)



Benchmark: Aspects and Embeddings

Puzzle	Aspects	Embeddings		Total Size	
		Full	Restr.	Full	Restr.
Simple Puzzle	9	21	15	57	41
Meteor	60	2596	2447	15576	14682
6×10 Pentominoes	72	2056	1828	12336	10968
$3 \times 4 \times 5$ Pentominoes	72	2440	2062	14640	12372
25-Y Pentominoes	125	960	891	4800	4455
25-N Pentominoes	125	960	891	4800	4455
Hollow Pyramid	125	5632	4850	28160	24250

Total size $= (\sum e : e \in E : \#e)$

Benchmark: Execution Times

Puzzle	Execution		
	Full	Restr.	LGF
Simple Puzzle	–	–	–
Meteor	0.8	0.5	0.5
6×10 Pentominoes	7.9	1.8	1.7
$3 \times 4 \times 5$ Pentominoes	1220	401	500
25-Y Pentominoes		7426	105
25-N Pentominoes		2701	102
Hollow Pyramid	?	?	?

Times in seconds on Pentium III, 533EB

LGF = Least-Growing Footprint (others: Short-Side First)

Concluding Remarks

- Simple transformations yield efficient (but longer) program
- Programs involve only five operations on one boolean array
- Special-purpose puzzle processor feasible for faster execution
- Transformations are typical in compilers for embedded software
- Formal methods are indispensible
- Plenty of opportunity for further research . . .

Acknowledgments

- Koos Verhoeff ('58): genes and memes
- Jan de Ruiter ('93): first solution for hollow pyramid puzzle
- Bart Tonneijk ('94): comparison of ZCO, LFF
- Erik van der Tol ('99): VLSI design for puzzle processor
- Zoltán Bálint ('00): implementation of footprint method