Solving General Lattice Puzzles

Gill Barequet[1] and Shahar Tal[2]

[1] Center for Graphics and Geometric Computing
Dept. of Computer Science
Technion—Israel Institute of Technology
Haifa 32000, Israel
barequet@cs.technion.ac.il

[2] Dept. of Computer Science
The Open University
Raanana, Israel
shahar_t@hotmail.com

Abstract. In this paper we describe general methods for solving puzzles on any structured lattice. We define the puzzle as a graph induced by the lattice, and apply a back-tracking method for iteratively find all solutions by identifying parts of the puzzle (or transformed versions of them) with subgraphs of the puzzle, such the entire puzzle graph is covered without overlaps by the graphs of the puzzle parts. Alternatively, we reduce the puzzle problem to a submatrix-selection problem, and solve the latter problem, using the "dancing-links" trick. Experimental results on various lattice puzzles are presented,

Keywords: Polyominoes, polycubes.

1 Introduction

Lattice puzzles intrigued the imagination of "problem solvers" along many generations. Probably the most popular lattice is the two-dimensional orthogonal lattice, in which puzzle parts are so-called "polyominoes" (edge-connected sets of squares), and the puzzle is a container which should be fully covered without overlaps by translated, rotated (and, possibly, also flipped) versions of the parts.

One very popular set of parts is the 12 "pentominoes" (5-square polyominoes), which is shown in Figure 1. We describe here a few examples of the many works on pentomino puzzles. Such games were already discussed in the mid 1950's by Golomb [Go54] and Gardner [Ga57]. Scott [Sc58] found all 65 essentially-different (up to symmetries) solutions of the 8×8 puzzle excluding the central 2×2 square. Covering the entire 8×8 square with the twelve pentominoes and one additional 2×2 square part, without insisting on the location of the latter part, is the classic Dudeney's puzzle [Du08]. The Haselgrove couple [HH60], as well as Fletcher [Fl65], computed the 2,339 essentially-different solutions to the 6×10 pentomino puzzle. Other pentomino puzzles were discussed by de Bruijn [Br71]. If pentominoes are not allowed to be flipped upside-down, then there exist 18 such 1-sided pentominoes. Golomb [Go65] provided one tiling of a 9×10 rectangle by all these pentominoes, while Meeus [Me73]

credited Leech for finding all the 46 tilings of a 3×30 rectangle by the set of 1-sided pentominoes. Haselgrove [Ha74] found one of the 212 essentially-different tilings of a 15×15 square by 45 copies of the "Y" polyomino. Golomb [Go65] provided many other polyomino puzzles in his seminal book "Polyominoes." See also [Kn00] for a listing of other well-studied puzzles.

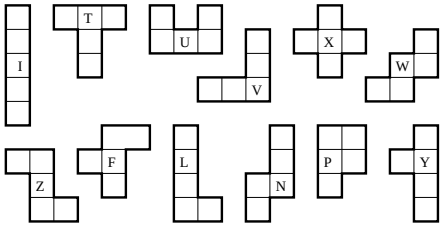


Fig. 1. Twelve essentially-different 2-dimensional pentominoes

It is well-known that finite-puzzle problems on an orthogonal lattice are NP-Complete. This is usually shown by a reduction from the Wang tiling problem, which is also known to be NP-Complete [Le78]. For completeness, we provide a short direct proof of this fact (by using a reduction from the *bin packing* problem). It is not surprising, then, that no better method than back-tracking is used for solving puzzles. Many works, e.g., [Sc58,GB65,Br71], suggested heuristics for speeding-up the process, usually by taking first steps with the least number of branches (possible next steps).

Knuth [Kn00] suggested the *dancing links* method, which allowed solving several problems, including orthogonal and triangular lattice puzzles, much more efficiently than before. The main idea was a combination of a link-handling "trick" that enables easy and efficient "unremove" operations of an object from a doubly-connected list (a key step in back-tracking), and an abstract representation of the problems as a matrix-cover problem: Given a 0/1-matrix, choose a subset of its rows such that each column of the shrunk matrix contains exactly one '1' entry. The reader is referred to the cited reference for more details of this extremely elegant method.

In this paper we present a back-tracking approach to solving a general lattice puzzle. We formulate the puzzle problem in general terms, so that it would fit various types of puzzles. We give a simple proof that a simple version of the problem, namely, a two-dimensional puzzle on an orthogonal lattice, is NP-Complete. Thus, we cannot hope (unless P=NP) for subexponential algorithms, and so, design an exponential back-tracking heuristics. We employ some tricks to expedite the algorithm. We fully implemented our algorithm and ran it on many puzzle problems which lie of several types of lattices.

This paper is organized as follows. In Section 2 we define the puzzle-solving problem, and prove in Section 3 that it is NP-Complete. In Section 4 we describe two algorithms to solve it, and present in Section 5 our experimental results. We end in Section 6 with some concluding remarks.

2 Statement of the Problem

Let $\mathcal{L}$ be a *lattice* generated by a repeated pattern. To avoid confusion, let us refer to the dual of $\mathcal{L}$, that is, to its adjacency graph, and use interchangeably the terms "cell" (of the lattice) and "node" (of the graph). A "pattern," in it simplest form, is a single node v and a set of labeled half-edges adjacent to it. (Each undirected edge e of the graph is considered as a pair of half-edges, each of which is outgoing from one endpoint of e.) The half-edges incident to v are labeled 0 through $d(v) - 1$, where $d(v)$ is the degree of v. In addition, there is a specification of how copies of v connect to each other to form $\mathcal{L}$. For a 1-node pattern, this specification is simply a pairing of the labels. In addition, there is a set of transformations defined on v. A transformation is a permutation on the set of labels, i.e., a 1-to-1 mapping between the set $\{0, \ldots, d(v) - 1\}$ and itself.

For example, the orthogonal two-dimensional lattice is generated by the pattern shown in Figure 2: The patern contains a single node, v, of degree 4. The four half-edges adjacent to v are labeled 'W'=0, 'N'=1, 'E'=2, and 'S'=3, which may be coupled extacly with 'E,' 'S,' 'W,' and 'N,' respectively. The pairing specification is, then, $\{(0, 2), (1, 3)\}$. Only one transformation, so-called *rotate left* (RL, in short) is defined on v. The mapping defined by RL on the half-edges incident to v is $\mathrm{RL}(i) = (i + 1) \bmod 4$. The composition of one to four RL transformations brings v to any possible orientation.

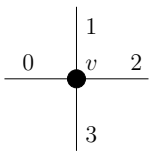


Fig. 2. Repeated pattern of $\mathbb{Z}^2$

Another example is shown in Figure 3: The repeated pattern of this 3D lattice (see Figure 6(c)) is a prism with a hexagonal cross-section. The degree of the single-node pattern v is 8. The eight half-edges, with labels $0, \ldots, 7$, are coupled by $\{(0, 1), (2, 5), (3, 6), (4, 7)\}$. Two transformations are defined on v:
(a) An "x-flip": $F(0|1|2|3|4|5|6|7) = (1|0|2|7|6|5|4|3)$; and
(b) A "yz-rotate": $R(0|1|2|3|4|5|6|7) = (0|1|3|4|5|6|7|2)$.
Compositions of F and R bring v (with repetitions) to all possible orientations in this lattice.

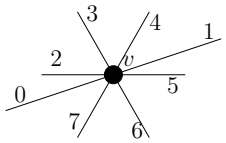


Fig. 3. The hexagonal-prism pattern

A lattice puzzle $\mathcal{P}$ is a finite subgraph of $\mathcal{L}$. Dangling half-edges (that is, half-edges that are *not* coupled with half-edges of other copies of the repeated pattern) are marked as the *boundary* of the puzzle. Puzzle parts are also finite subgraphs of $\mathcal{L}$, and they undergo a similar procedure. A solution to the puzzle is a *covering* of $\mathcal{P}$ by a collection of parts, such that:

1. All nodes of $\mathcal{P}$ are covered by nodes of the parts;
2. The label of dangling half-edges of parts match the labels of the respective half-edges of $\mathcal{P}$; except
3. Dangling half-edges of parts may extend out of the boundary of $\mathcal{P}$.

Some freedom can be given to the set of parts, so as to form variants of the puzzle problem. We may have a single copy of a given part, or an unlimited amount of copies. Different types of transformations may be defined for different parts of the puzzle.[3] While solving a puzzle, we may want to compute one solution (thus, determine whether or not the puzzle is solvable), or to find the entire set of solutions to the puzzle.

3 NP-Completeness

For completeness, we first show that even simple 2D puzzle problems on an orthogonal lattice are NP-Complete. We do this by using a trivial reduction from the bin-packing problem, which is known to be NP-Complete [GJ79].

Consider an instance of the bin-packing problem, in which one is given a set E of n elements, with the respective sizes $s_1, \ldots, s_n$, and one is asked whether or not E can be partitioned into at most k subsets, such that the total size of the elements in each subsets is at most m. Trivially, $\sum_{i=1}^{n} s_i \leq km$, otherwise the answer is immediatly "No."

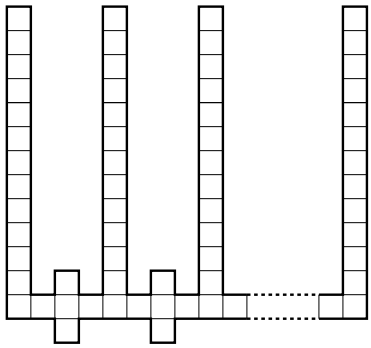


Fig. 4. The 2-dimensional puzzle problem is NP-Complete

Such an instance of bin packing can easily be represented as a puzzle, even if we require the puzzle to be connected and the parts to cover the entire puzzle. The parts of the puzzle are:

- n "sticks" with the same lengths of the elements of E, that is, rectangles of size $s_i \times 1$, for $1 \leq i \leq n$;
- $k-1$ copies of the "X" pentomino; and
- $km - \sum_{i=1}^{n} s_i$ (possibly zero) singleton squares.

The puzzle is shown in Figure 4. The size of the puzzle is $5(k-1) + km = k(m+5) - 5 = \Theta(km)$. The total number of parts is $n + k - 1 + km - \sum_{i=1}^{n} s_i \leq n + (k+1)m - 1 = \Theta(km+n)$, and their total size is identical to that of the puzzle. The original bin-packing problem requires $\Theta(n + \log(km))$ storage. Although it seems like the size of the puzzle and the time to build it are polynomial in n, k, and m, only $\Theta(n + \log(km))$ storage and reduction time are needed due to the

[3] For example, consider the *flip* transformation in a two-dimensional othogonal-lattice puzzle. Its analogue in three dimensions is the *mirror* transformation, which is mathematically well defined but hard to realize with physical puzzle pieces made of wooden cubes. In such a puzzle, whether or not to allow the mirror transformation is a matter of personal taste.

repetitive shape of the puzzle. The puzzle is stored efficiently, say, by a standard two-dimensional run-length encoding.

In time linear in the size of the puzzle, one can verify that a candidate solution is indeed a real one. It is also trivial to verify that a solution to the bin-packing problem corresponds to a solution to the puzzle, and vice versa. A key observation is that all the $k-1$ copies of the "X" pentomino must be located at the connections between the teeth of the comb—they were introduced in order to create a connected puzzle. The purpose of the singleton squares is to ensure that the puzzle is filled entirely. Needless to say, variants in which the puzzle is not connected, and/or it does not necessarily have to be completely filled, and/or different part orientation are allowed, are NP-Complete as well.

4 Algorithms

In this section we describe two puzzle-solving algorithm: Direct back-tracking and matrix cover. Since both methods are well-known, emphasis is put not on these paradigms but rather on the data structures and details of the implementation in the setting of *general* lattice puzzles.

4.1 Back-Tracking

Our first approach to solving the puzzle problem is by direct back-tracking.

Puzzle and Part Data Structures As mentioned above, the puzzle and parts are represented by graphs. Edges of the graphs are labeled with numbers: for each node v, the outgoing half-edges of v are labeled $0, \ldots, d(v)-1$, where $d(v)$ is the degree of v. Dangling half-edges of both puzzle and parts (that is, half-edges beyond their boundaries), are omitted.

One specific node of each part is marked as the *origin* of the part. For example, in the two-dimensional orthogonal lattice, one may fix the origin at the leftmost cell of the topmost row of the part. Note that the edge labels have to induce a total order on the neighbors of a cell, thereby, on all the cells of the puzzle or its parts. In a general lattice, we simply choose the lexicographically-smallest cell, induced by this order, as the origin of the part. Similarly, we mark the origin of the puzzle. For example, in the two-dimensional orthogonal lattice, a cell is always "smaller" than its bottom and left neighbors, and "greater" than its top and right neighbors. This implies the chosen origin in this lattice.

Part Orientations A special data structure stores the transformations allowed for each part of the puzzle. As mentioned above, a transformation is a 1-1 mapping between the set of possible labels and itself. In a preprocessing step, we compute all the possible orientations of each part of the puzzle. Specifically, we apply all possible transformations (and combinations of them) to the half-edges outgoing from the origin cell. Applying a single combination to the origin changes

the orientations of all the its neighbors (manifested by new edge labels), and the process continues recursively in a depth-first (or breadth-first) manner to all the cells of the part.

Naturally, the graph that represents a part may contain cycles, in which case "back-edges" are encountered in the course of the search. Note that the formal definition of a lattice does *not* ensure that back-edges are consistent edge-labeling-wise. Such inconsistency simply means that a specific transformation (the analogue of a rotation) is not allowed in the dealt-with lattice. However, in all lattices we experimented with, such a situation can never occur; therefore, we did never check for consistency of back-edges.

After all orientations of a part have been found, two steps should be taken: (a) Recomputing the origin cell of each oriented copy; (b) Removing duplicate copies. In fact, both steps can be performed simultaneously. Moreover, the removal of duplicates may be avoided if we precompute the symmetries of the original part. Nevertheless, all these operations are performed in a preprocessing step, before running the main puzzle-solving algorithm (which takes the main bulk of the running time), so any approach will practically do.

Essentially-Different Solutions In most cases we are not interested in finding multiple solutions that are inherently the same, up to some transformation defined for the specific lattice. Instead of computing all the solutions and look for repetitions (an operation which might render the algorithm infeasible if there are too many solutions), we applied a simple pruning method. Suppose that S is a solution to a puzzle. One only need to observe that if the entire puzzle has some symmetry realized by the transformation T, then $T(S)$ is also a solution to the puzzle. Thus, if we discard all copies of an arbitrary part, obtained by transformations that realize symmetries of the puzzle, we guarantee that only essentially-different solutions to the puzzle will be found. To maximize efficiency, we choose the part with the maximum number of copies. (Usually, this is the part with the least number of symmetries.)

Solving the Puzzle Our first method is a classical back-tracking algorithm. We attempt systematically to *cover* the puzzle graph with the part graphs. A part graph covers exactly a portion of the puzzle graph by an injective mapping of the nodes of the part to the nodes of the puzzle, subject to adjacency relations in the two graphs, while the labels of the (half-)edges of the part fully match those of the covered portion in the puzzle.

Initially, all parts are "free," and all nodes in the puzzle graph are "empty." We initialize a variable, called the *anchor*, to be the origin cell of the puzzle. Naturally, it should be covered by some cell c of some part p. Moreover, c must be the origin of p, otherwise, after positioning p in the puzzle, the origin of p will occupy a puzzle cell which is different from the origin of the puzzle, which is a contradiction to the lexicographic minimality of the puzzle origin. Thus, positioning a part in the puzzle (in one of its possible orientations) is achieved by identifying the anchor with the origin of the part and traversing the part

graph. A simultaneous and identical (edge-label-wise) traversal is performed in the puzzle graph. This yields precisely which portion of the puzzle is covered by the part. An attempt to position a part in the puzzle fails if either the traversal of the puzzle graph reaches an "occupied" cell, or it gets out of the graph (that it, the boundary of the puzzle is about to be crossed).

If the part-positioning is successful, we mark the part as "used," mark all the nodes of the puzzle graph that are matched to nodes of the positioned part as "occupied," and proceed to the next part-positioning step as follows. First, we set the anchor of the puzzle to be the new lexicographically-minimal "empty" node in the puzzle graph. This is achieved by scanning the puzzle lexicographically, starting from the previous anchor and looking for a free node. Then, we attempt to position a new "free" part (according to a predefined order) in the puzzle by identifying its origin with the new anchor and proceeding as above. The new anchor must be the origin of the next positioned part for the same reason as for the first positioned part.

This part-positiongs process continues until one of two things happen: Either (a) The puzzle graph is fully covered (this situation is identified by not being able to reset the anchor); or (b) The algorithm is stuck in a situation in which the puzzle graph is not fully covered, yet no new part can be positioned. In the former situation the algorithm declares a solution and terminates. In the latter situation the algorithm back-tracks: The last positioned part is removed from the puzzle, restoring its status to "free" and marking again the covered puzzle nodes as "empty." Then, if another orientation of the same part exists, the algorithm attempts to position it as above. Otherwise, the algorithm proceeds to the next available part (according to the predefined order) and attempts to position it in the same manner.

In case we like to find *all* the solutions to the puzzle, a minor step of the algorithm is modified: When the algorithm finds a solution, the latter is reported, but then refers to the last part-positioning step as a failure and then back-tracks. In such a situation, the algorithm terminates when no back-tracking options remain: This happens when all options for the *first* part-positioning are exhausted.

We conclude the description of the algorithm by referring to the dangling half-edges in part graphs. In principle, they should be matched too to half-edges in the puzzle graph, in order to ensure proper *neighborhoods* between parts positioned in the puzzle. However, this was redundant in all the lattices that we handled. Thus, we ignored all dangling half-edges in the part-positioning operations. A cover of the puzzle graph was actually only an exact cover of the *nodes* of the graph. Edges in the puzzle graph, that represent neighborhood relations between parts, were not covered. Exact match of labels was enforced only between *inter-part* edges and the corresponding edges in the puzzle graph.

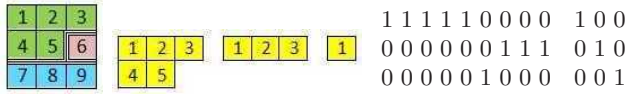
4.2 Matrix Cover

Our second approach to solving the puzzle problem is by a reduction to *matrix cover*, and solving the latter by using the "dancing links" technique [Kn00].

Reduction The puzzle problem is represented by a binary matrix in the following manner. We create an $M \times N$ matrix, where M is the total number of options to position parts (in all orientations) in the puzzle, and N is the number of cells of the graph. By "part-positioning" options we mean all possible mappings of the anchor of a part graph to a node in the puzzle graph, such that the part will partially cover the puzzle and will not exceed its boundary. An entry (i, j) in the matrix contains the value 1 if the jth node of the puzzle is covered by some node in the ith part-positioning option; otherwise it is 0.

Naturally, solving the puzzle amounts to choosing a subset of the lines, in which every column contains a single 1 with 0 in all other entries. The chosen lines represent the choice of parts, while the requirement for a single 1 per column guarantees that every cell of the puzzle will be covered by exactly one part.

We *may* want to ensure that every part will be used exactly once, among all its possible orientations and positions in the puzzle. This is easily achieved by adding more columns to the matrix, one for each part. In each such column, we put 1 in all the lines that correspond to the same part, and 0 elsewhere. This guarantees that exactly one part orientation and position will be chosen.



(a) Puzzle (b) Parts (c) Submatrix solution

Fig. 5. A puzzle solution represented by a submatix

For example, consider the simple 3X3 puzzle shown in Figure 5(a). The three puzzle parts are shown in Figure 5(b). The left part has eight different orientations (allowing flipping), in each of which it can be positioned in the puzzle in two ways, for a total of 16 options. The middle part has two different orientations, in each of which it can be positioned in the puzzle in three ways, for a total of 6 options. Finally, the right part has only one orientation, which can be positioned in the puzzle in nine ways. Thus, the matrix we need to cover is made of 31 lines (the total number of part-positioning options) and 12 columns (9 puzzle cells plus three original parts). One possible solution to the puzzle is represented by the 3-line submatrix shown in Figure 5(c): The 9 left columns show the exact covering of the puzzle, while the 3 right columns show that each part is used exactly one.

The matrix-cover algorithm is also back-tracking in nature, but its running time is reduced significantly as described below.

Dancing Links The algorithm is sped up tremendously by an elegant pointer-manipulation trick [Kn00]. An element x is removed from a doubly-connected linked list by executing the pair of operations `Next[Prev[`x`]]` := `Next[`x`]` and `Prev[Next[`x`]]` := `Prev[`x`]`. It is a good programming practice not to access the

storage of an element x after it is deleted, since it may already serve a different purpose. However, if it is guaranteed that the area allocated to x is not altered even after it is deleted, one can easily undo the deletion of x by making the links "dance": The pair of operations `Next[Prev[`x`]]` := x and `Prev[Next[`x`]]` := x readily return x to its original location in the linked list.

Efficient Branching A very efficient heuristic, which reduces the running time significantly, is ordering the branches of the algorithm according to their (anticipated) size. The speed-up of the algorithm naturally depends on the quality of the prediction of the sizes of branches. In our setting, branching occurs at the selection of an additional column. Recall that in a valid solution, exactly one of the lines comprising the submatrix contains the value 1 in this column. (This means that exactly one part covers the puzzle cell corresponding to this column.) Since no a priori information is known (or computed), a reasonable choice is to explore first columns with the *least* number of 1-entries. This is because we will have fewer parts among which to choose the one covering this cell of the puzzle.

Stranding Another simple pruning method is to consider the size of the connected component of empty nodes, that include the anchor, in the puzzle graph. If all free parts are larger than this component, then the algorithm nay backtrack without any further checking.

5 Experimental Results

The two algorithms were implemented in JAVA on a dual-core 2.2GHz PC with Sun's Virtual Machine 5+, under the Windows XP and Linux operating systems. The two algorithms allowed simple parallelism for using the dual-core CPU. The software consisted of 5,500 lines of code. In addition, we implemented a warm-restart mechanism which allowed to resume the course of the algorithm from occasional crashes of the system.

(a) 3D orthogonal (b) Packed spheres (c) Hexagonal prisms

Fig. 6. Lattice puzzles

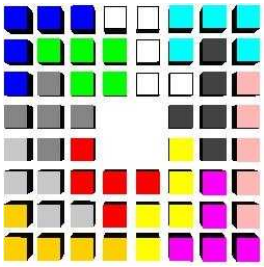
The first two lattice types which we experimented with were the two- and three-dimensional orthogonal lattices. The formal structure of the 2D lattice is described in the introduction of this paper. It is straightforward to generalize it to three dimensions. Figure 6(a) shows a sample puzzle in the 3D lattice. The third

lattice type is the "packed-spheres" lattice. Figure 6(b) shows a sample puzzle in this lattice. In this lattice, the repeated pattern is a node of degree 12. A complete characterization of the group of transformations in this lattice is provided in the full version of the paper. The fourth lattice type is the "hexagonal-prism" lattice, whose structure is also described in the introduction. Figure 6(c) shows a sample puzzle in this lattice.

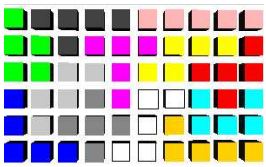
Lattice	Puzzle		Parts			Solutions		Method	Branches	Time (Sec.)	
	Name	Size	Orig.	Total	Reduced	Unique	Sym.			First	All
2D Ort.	$8 \times 8 - 2 \times 2$	60	12	56	56	65	*8	BT	2,757,203	0.2	8.0
								BT+ST	1,523,328	0.0	3.4
								DL	2,091,625	0.4	12.5
								DL+ST	2,375,555	0.3	9.5
								DL+SZ	117,437	0.1	1.5
								DL+SZ+ST	81,638	0.0	1.2
2D Ort.	10×6	60	12	56	56	2,339	*4	BT	10,595,621	0.1	20.0
								BT+ST	5,802,074	0.1	13.0
								DL	10,597,824	0.2	52.0
								DL+ST	9,235,621	0.1	40.0
								DL+SZ	1,728,193	0.0	13.0
								DL+SZ+ST	1,476,710	0.0	10.0
3D Ort.	$10 \times 2 \times 3$	60	12	168	68	12	*8	BT(+ST)	861,525	0.1	0.9
								DL(+ST)	1,317,971	0.1	5.5
								DL+SZ(+ST)	72,671	0.1	0.5
3D Ort.	$5 \times 4 \times 3$	60	12	168	144	3,940	*8	BT(+ST)	1,259,189,714	1.5	4,020.0
								DL(+ST)	1,969,102,501	2.7	6,180.0
								DL+SZ(+ST)	10,103,602	0.0	73.0
3D Ort.	$6 \times 5 \times 2$	60	12	168	93	264	*8	BT(+ST)	67,714,344	0.3	168.0
								DL(+ST)	107,596,954	12.0	485.0
								DL+SZ(+ST)	677,083	0.0	5.5
3D Ort.	Green Happy Cube	98	6	91	91	20	*24	BT(+ST)	1,215	0.1	0.6
								DL(+ST)	1,953	0.0	0.9
								DL+SZ(+ST)	159	0.0	0.6
3D Ort.	Orange Happy Cube	98	6	79	79	2	*24	BT(+ST)	846	0.1	0.3
								DL(+ST)	1,320	0.0	0.6
								DL+SZ(+ST)	81	0.0	0.6
3D Ort.	Strip	60	12	168	85	6	*4	BT	187,883	0.2	1.9
								BT+ST	99,272	0.1	0.7
								DL	195,684	0.3	2.6
								DL+ST	181,554	0.2	2.6
								DL+SZ	14,076	0.0	0.1
								DL+SZ+ST	13,706	0.0	0.1
3D Ort.	Stairs	55	??	186	186	640	*1	BT	2,088,970	0.1	10.0
								DL	3,143,814	3.7	23.0
3D Ort.	Big Y	60	12	186	157	14	*1	BT	161,584,456	6.0	444.0
								DL(+ST)	210,494,003	3.6	545.0
								DL+SH(+ST)	204,910	0.2	1.5
Spheres	Tetrahedron 4	20	6	137	50	6	*12	BT	1,228	0.4	0.4
								DL	2,261	0.5	0.5
								DL+SZ	44	0.5	0.5
Hex Prism	Hex prisms	45	11	92	89	2	*3	BT	2,268	0.0	0.1
								DL(+ST)	2,826	0.0	0.1
								DL+SZ(+ST)	93	0.0	0.1

Table 1. Statistics of puzzle solving

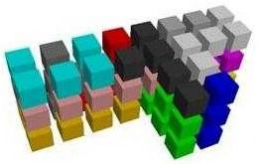
We experimented with many puzzles in these lattices, and report here (see Table 1) about solving only a few of them. The *size* of a puzzle is the number of cells it contains. For parts we provide three numbers: the number of original parts, the total number of different oriented parts, and the number of oriented parts that can fit into the puzzle. (Naturally, a long part cannot fit at all into a thin puzzle in the wrong orientation. This can be checked in a preprocessing step for general puzzle graphs too.) We provide two counts of solutions: essentially different and the total number of solutions. The method codes are the following: BT: back-tracking; ST: the stranding heuristics; DL: dancing links; and SH: size heuristics. Branches are decision points where the back-tracking algorithms place a part or the matrix-cover algorithm chooses a column. Finally, we report the running times which were needed in order to find either a single solution or all solutions to the puzzle. Figure 7 shows representative solutions to these puzzles.



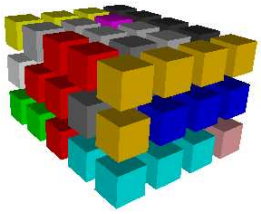
(a) 2D $8 \times 8 - 2 \times 2$



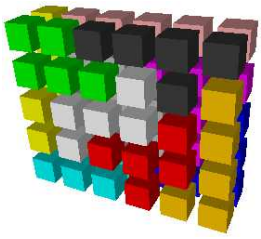
(b) 2D 10×6



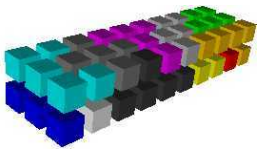
(c) 3D Big Y



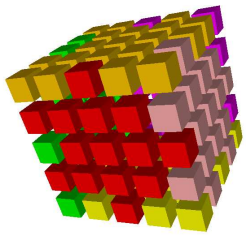
(d) 3D $5 \times 4 \times 3$



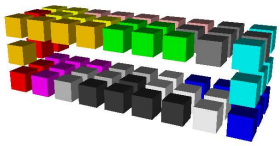
(e) 3D $6 \times 5 \times 2$



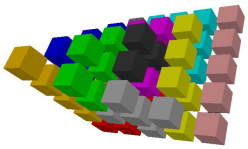
(f) 3D $10 \times 3 \times 2$



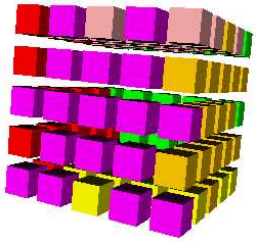
(g) 3D Green Happy Cube



(h) 3D Strip

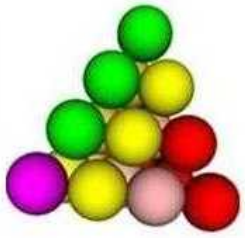


(i) 3D Stairs



(j) 3D Orange Happy Cube

(k) Hexagonal prisms



(l) Sphere Pyramid

Fig. 7. Solutions to various puzzles

As can be easily observed from the data in Table 1, the matrix-cover algorithm, coupled with the dancing-link "trick," is superior over the back-tracking algorithm. Both methods can be improved heuristically. The table shows clearly that the size-ordering heuristics contributes the largest reduction in running time. Without it or on top of it, the stranding heuristics can also make a modest improvement to the algorithm.

6 Conclusion

In the paper we presented two puzzle-solving algorithms, and identified one of them (the matrix cover) as the method of choice. The algorithms run on puzzles in any general lattice. We experimented comprehensively with many puzzles in four different types of lattices. Our future goals are to extend the algorithms to other lattices, to incorporate more restrictions and conditions on the structures of puzzles and/or parts, and to continue to improve the efficiency of our implementation of the algorithms.

References

[Br71] N.G. de Bruijn, Programmeren van de pentomino puzzle, *Euclides*, 47 (1971–1972), 90–104.

[Du08] H.E. Dudeney, 74.—The broken chessboard, in: *The Canterbury Puzzles*, 1908, 90–92.

[Fl65] J.G. Fletcher, A program to solve the pentomino problem by the recursive use of macros, *Comm. of the ACM*, 8 (1965), 621–623.

[Ga57] M. Gardner, Mathematical games: More about complex dominoes, plus the answers to last month;s puzzles, *Scientific American*, 197 (1957), 126–140.

[GJ79] M. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.

[Go54] S.W. Golomb, Checkerboards and polyominoes, *American Mathematical Monthly*, 61 (1954), 675–682.

[Go65] S.W. Golomb, *P*olyominoes, Scribners, New York, 1965; 2nd ed., Princeton Univ. Press, 1994.

[GB65] S.W. Golomb and L.D. Baumart, Backtrack programming, *J. of the ACM*, 12 (1965), 516–524.

[HH60] C.B. Haselgrove and J. Haselgrove, A computer program for pentominoes, *Eureka*, 23 (1960), 16–18.

[Ha74] J. Haselgrove, Packing a square with Y-pentominoes, *J. of Recreational Mathematics*, 7 (1974), 229.

[Kn00] D.E. Knuth, Dancing links, in: *Millennial Perspectives in Computer Science* (J. Davies, B. Roscoe, and J. Woodcock, eds.), 187–214, Palgrave Macmillan, England, 2000.

[Le78] H.R. Lewis, Complexity of solvable cases of the decision problem for the predicate calculus, *19th Ann. Symp. on Foundations of Computer Science*, Ann Arbor, MI, 35–47, 1978.

[Me73] J. Meeus, Some polyomino and polyamond problems, *J. of Recreational Mathematics*, 6 (1973), 215–220.

[Sc58] D.S. Scott, Programming a combinatorial puzzle, Technical Report 1, Dept. of Electrical Engineering, Princeton Univ., June 1958.