Learn to solve pentominoes

Find out how the theory behind a kid's toy can illustrate some deep computer science

Pentominoes, a simple educational game, can reveal some surprisingly deep computer science. Although the first backtracking solution to the puzzle was implemented in 1958, Donald Knuth devised an entirely new way of looking at the problem in 2000. This method is now called the Dancing Links algorithm (or, more usually, DLX).

A pentomino set consists of 12 different pieces, each representing one of the ways of joining five squares along their edges to form a unique shape. Figure 1 shows the complete set, together with each piece's usual name: a letter of the alphabet that most closely resembles it (F, I, L, N, P, T, U, V, W, X, Y, and Z). Since there are 12 pieces, each five squares in size, you could investigate solutions to fitting them all in a 3 x 20 rectangle, a 4 x 15 rectangle, a 5 x 12 rectangle or a 6 x 10 rectangle, as well as other more fanciful shapes.

Back in my early teens I was given a plastic set of pentominoes arranged in a 6 x 10 box, presumably for my birthday or Christmas. I seem to remember spending an inordinate amount of time over several months trying to find all the ways of arranging the pieces in the box. There are 2,339 different ways, not counting rotations or reflections. I even used a notebook with squared paper to record the permutations I discovered. (Although I still have that pentomino set, the notebook has been lost in the mists of time.) Needless to say, I did not find all 2,339 methods.

Exploring the methods

Figure 2 overleaf shows a solution for the 6 x 10 box (this is the solution I carefully kept written down with the box, since not knowing how to put the pieces back would have been embarrassing). If you look carefully, you'll see that you can immediately generate another by swapping over the T and F.

After playing around with the pieces for a while, the natural question to ask (well, natural if you're a programmer) is this: how can I write a program to generate all the solutions to a particular puzzle? Let's look at what might be needed for a 3 x 20 rectangular board (there are two solutions to this particular puzzle, not counting rotations and reflections).

A first idea might be the following algorithm: take the pieces in alphabetical order, F to Z. Place F somewhere on the empty board and mark off the squares it covers. Place the I on the board, making sure that it doesn't cover any of the already covered squares, and mark off the squares it covers. Place the L likewise. At some point you'll find that you won't be able to place another piece, so you should then backtrack and move one of the already placed pieces to another position (perhaps by rotating or flipping it over) and then try again. Continue this process, with the realisation that you might have to backtrack several times at any one check, and eventually you'll discover all permutations.

The big problem with this solution is that it will take some time. There are 144 different places and orientations for the F in a 3 x 20 board, 48 for the I, 136 for the L, and so on. Many of these will be completely illegal (for ▸

Origins of the link idea

The 'unlink a node/relink the same node' idea was first used by Hitotumatu and Noshita to speed up Dijkstra's algorithm for solving the N queens chess problem. This problem was first proposed in 1848 as the eight-queen problem: put eight queens on a chessboard such that no queen can capture any other (no two queens can be in the same row, column or diagonal). Dijkstra published a solution using then-unfamiliar structured programming methodology and Hitotumatu and Noshita made it twice as fast by using the unlink/relink trick. ■

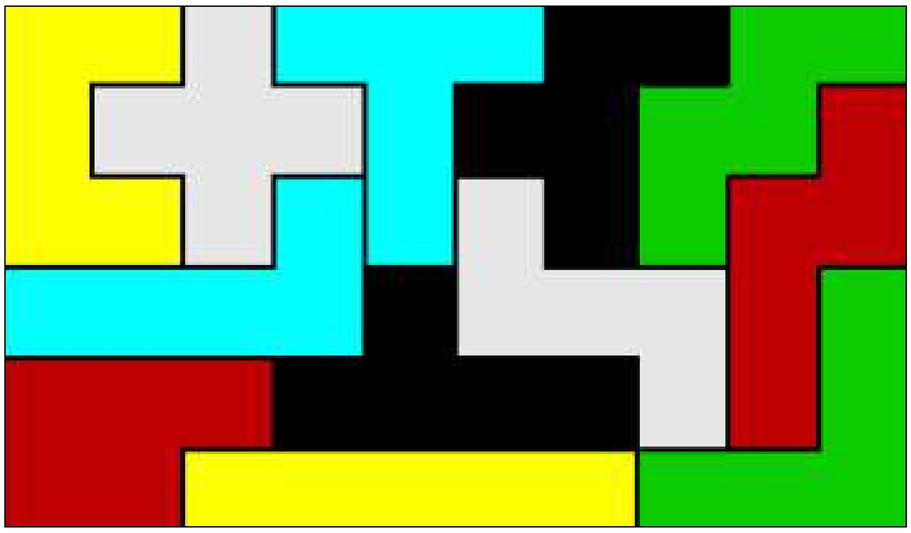
example, the F cannot appear right at the edge of the board, since in doing so you will always leave at least one empty square). So, in all you'll be trying out 144 × 48 × 136 × ... different permutations. Not very fast at all.

A better implementation might be to switch it around a little. Consider the square at the upper left of the board. Select a piece that covers it and then mark off the remaining four squares also covered by this piece. Find the next uncovered square (for this type of board, because it's so narrow, it'll be better to search down the narrow axis rather than the long axis). Try to find a piece that can be placed legally there. If there isn't one, backtrack again and change the piece you've already placed. Sometimes you might have to backtrack several steps.

Because you're more likely to identify impossible solutions more quickly (for example, there are only two possible ways to place the F to cover the top left square and both are shown to be illegal when considering what to place in the next square downwards), you'll find that this algorithm is much more efficient at identifying valid solutions.

If you try and implement either of the above solutions, though, you'll probably find yourself calculating the positions (and which squares are covered by each of the individual pieces) over and over again. It would be better to calculate them once and then put the values into a table that you can refer to as you try out various placements.

Before we discuss the best data structure for that table, let's step back for a moment and consider what's known as the Exact Cover Problem. Imagine the following scenario. You're given a matrix of ones and zeros; that is, where the cells contain either 1 or 0. The first part of Figure 3 shows such a matrix. The problem you have to solve is to work out which rows of the matrix form a set that contains exactly one 1 in each column. There may be no such set of rows, there may be exactly one or there may be more than one. A solution to this problem is known as an 'exact cover': a set of rows that exactly covers the columns. Take a moment to work out the exact cover for Figure 3.


▲ **Figure 2: One of several possible solutions to the 6 x 10 pentomino puzzle.**

Solving a matrix

Despite the ease with which a visual inspection will provide the answer with this small matrix (rows 2, 3, and 5), this problem gets much harder (it's 'NP-complete' in computer science terms) the more rows and columns there are.

The algorithm for solving it is recursive. Let's look at Figure 3. You're given a matrix. Choose a column (say the first). Choose a row such that the cell at the intersection of the column and row is a 1 (let's go for row 4). Add that row to our solution. Now since that row will have a 1 in column 1, we must remove all the other rows that have a 1 in column 1 (that would be row 5). Not only that, but we must also remove all the rows that have a 1 in the same column as this chosen row. At row 4 we hit the jackpot: we can get rid of row 5 (again), row 2 and row 6. We remove row 4 (since it's in our possible solution) and the three columns that row 4 covered (there's no need to recheck them because we've already covered them). Our matrix now looks like the second part of Figure 3.

We do the same thing with this reduced matrix. We choose row 3 as our row since it intersects with column 2 in a 1. Unfortunately, we then have to delete row 1 as well since there's a clash in column 6, meaning that columns 5 and 7 never get covered and we failed to get an exact cover.

So we backtrack. Maybe choosing row 3 was the mistake. However, there's no way to cover column 2 in that particular reduced matrix, and so we must backtrack to our choice of row 4, made right at the beginning. Since that choice led to no exact cover, we see if we can make another choice of row to cover column 1. This time, it's row 5.

Selecting row 5 means we get rid of row 4 (again) and 1, as well as columns 1, 3 and 7 (since row 5 covers them). We're then left with the reduced matrix shown in part 3 of Figure 3.

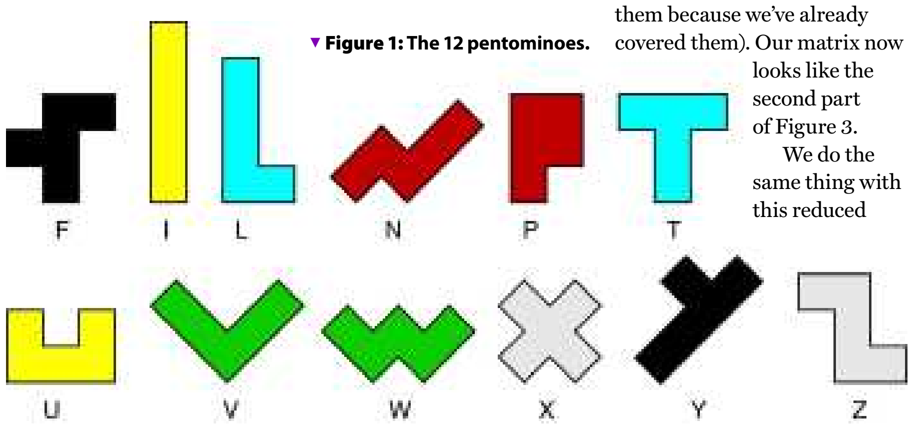
Here we make the choice of row 3, which removes row 6 as well as columns 2 and 6. And this, as I'm sure you can see, leaves us with row 2 covering the remaining two columns. (To be pedantic, we then select row 2, which removes the final two columns as well.) Hence our exact cover solution is rows 2, 3 and 5.

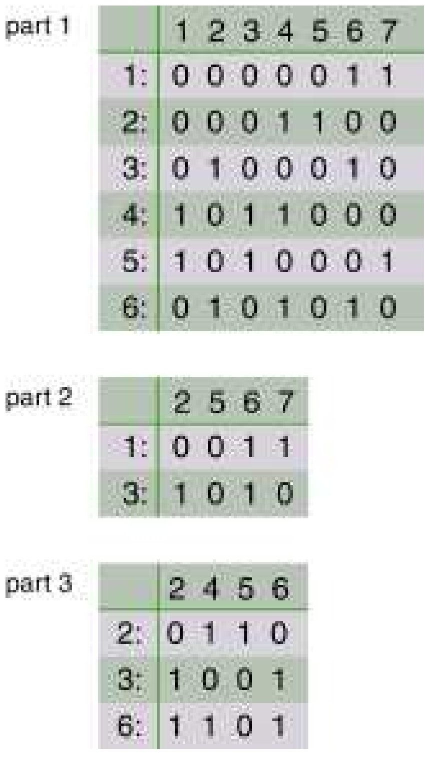
That's all fine and dandy, but what does this exact cover problem have to do with pentominoes? In particular, how does it help with defining the table of where a piece can be placed and which squares it covers?

The answer is to set up the puzzle as an exact cover problem. Let's define the columns first: the initial 12 columns will be the pentomino pieces, one column for each piece. The next 60 columns will be one per square from the board: counting from top left across the top row (1 to 20), then the second row (21 to 40), then the final row (41 to 60).

Now we can define each row as being a single piece placed on the board in a particular position. Since it's a single piece, we'll have a 1 in only one of the first 12 columns, according to which piece it is. Now we put a 1 in each column of the next 60 that corresponds to the squares covered by the piece in that position. For example, if we take F in its 'normal' orientation and place it abutting the left side,

▼ **Figure 1: The 12 pentominoes.**





part 1

	1	2	3	4	5	6	7
1:	0	0	0	0	0	1	1
2:	0	0	0	1	1	0	0
3:	0	1	0	0	0	1	0
4:	1	0	1	1	0	0	0
5:	1	0	1	0	0	0	1
6:	0	1	0	1	0	1	0

part 2

	2	5	6	7
1:	0	0	1	1
3:	1	0	1	0

part 3

	2	4	5	6
2:	0	1	1	0
3:	1	0	0	1
6:	1	1	0	1

▲ **Figure 3: An exact cover problem in part 1; partial steps to a solution in parts 2 and 3.**

the row will have a 1 under the F column, and a 1 under columns for squares 2, 3, 21, 22, 42. All the other 66 columns for that row will be 0. (In fact, every row we add to the matrix will contain six 1s and 66 0s.)

How many rows will there be? To find out, we sum all the different ways we can place each piece. For example, for F there are 18 positions for each of its orientations and eight orientations, making 144 distinct positions. I did the maths and came up with the following number of positions for each piece: F: 144, I: 48, L: 136, N: 68, P: 220, T: 72, U: 110, V: 72, W: 72, X: 18, Y: 136, Z: 72. In other words, the puzzle matrix has 1,168 rows. That is one huge matrix, but note that it's also very sparse: only 1 in 12 cells is a 1.

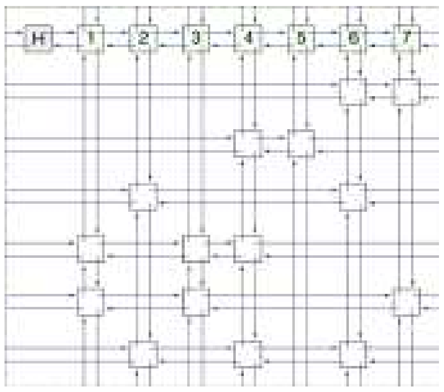
Now we have to find the exact covers for this 72 x 1,168 matrix. Because we haven't done anything to avoid solutions that would be rotations or reflections of other solutions, we'll find four times as many solutions as expected.

The fun part is how to solve the exact cover problem efficiently. It's a non-polynomial problem (probably exponential), so for a much larger matrix it'll get very hard very quickly. However, for this sparse, mid-sized matrix it can be solved pretty quickly.

Dancing links

The algorithm to use is the one I mentioned above (Donald Knuth calls it Algorithm X). However, keeping track of all the rows and columns you're removing or adding back – the housekeeping, if you like – is quite complicated. To aid in this housekeeping issue, Knuth came up with a refinement that he called Algorithm DLX, standing for something like 'Dancing Links implementation of Algorithm X'. You can get the paper from **www-cs-faculty.stanford.edu/~knuth/preprints.html** – it's paper P159.

Consider a doubly linked list where each node has a 'Next' and a 'Prev' reference to the next and previous nodes in the list. Then, in order to delete a given node X, all we have to do is set 'X.Next.Prev' to **X.Prev** and 'X.Prev.Next' to **X.Next**. X will no longer be in the linked list. However, providing that we don't alter X in any way, we can just as easily add it back into the list: set both 'X.Next.Prev' and 'X.Prev.Next' to **X**. So although the linked list has no more references to X, X itself still has 'knowledge' of where it was in the list.



▲ **Figure 4: A linked list tapestry for the exact cover problem demonstrated in Figure 3.**

That's all very clever, but so what? Well, consider our 72 x 1,168 sparse matrix. How would we represent that in memory? We could use a standard matrix of boolean values, which would take 84,096 cells, but such a structure is too rigid. Better to use a linked list of cells containing a 1 along each row. Knuth went even further: not only are the '1' cells joined in a linked list along each row, but they are equivalently joined up and down in a linked list as well. The matrix becomes a tapestry of linked lists, with each '1' cell having a link to its left, right, up and down neighbouring '1' cells. '0' cells are not stored at all.

Figure 4 shows the exact cover problem matrix that we saw in Figure 3 as a linked list matrix. Each column has a header cell and the header cells have their own single header cell. The lists are circular lists: the links off to the left join at the right (and vice versa) and the links off the bottom join at the top (and vice versa).

Now that we have this woven data structure of linked lists, we can make use of the remove/add-back-again idea that I just introduced. To cover a column we unlink it from the column header's linked list using our code. We can easily unlink the rows it contains by the same method. If we need to backtrack, we can add the rows back, in reverse order, and finally add the column back in. The housekeeping of Algorithm X is done through this property of deleted nodes 'knowing' where they came from. We don't need to have any special additional data structure to hold the removed rows and columns: the nodes themselves keep track of all that automatically.

In essence, Algorithm DLX finds an exact cover if, during a recursive call, the header of header's node points to itself. It fails to find one if there are still column headers but they have no rows; we shall then need to backtrack. **PCP**

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk

Spotlight on… Solving Sudoku

The DLX algorithm can also be used to solve Sudoku puzzles. The rows in a Dancing Links matrix for Sudoku are the exhaustive list of all possible digits in all possible cells. Since there are 81 cells, each containing one of nine possible values, we'll have 729 rows. Name each row in the format '[D, R, F]' where D is the digit, R is the rank (the row in the Sudoku grid), and F the file (the grid column).

View the columns as being constraints. There are four kinds of constraints to how we can fill in a Sudoku grid.

1. Each cell must contain exactly one digit.
2. Each digit must appear in each rank exactly once.
3. Each digit must appear in each file exactly once.
4. Each digit must appear in each box exactly once.

Each constraint will be represented by columns. Looking at item 2, we have nine possible digits in nine possible ranks – 81 possible pairs, and therefore 81 columns. In each column we put a 1 in each cell that intersects the digit/rank pair for that column. So, for example, if we're filling in the column for digit 2, rank 5, we'd put a 1 in the rows named [2,5,1], [2,5,2], and so on. The same argument goes for the other constraints as well: each 81 columns, 324 columns in all.

Now the matrix is set up, we can remove the rows and columns that correspond to the Sudoku puzzle we're trying to solve and find the remaining exact cover. ■

Improving performance

We can make a simple change to DLX to improve its performance. Instead of choosing a column at random to cover, we should choose them in some kind of order. The best method is to choose the column with the fewest cells. In any backtracking type algorithm, choosing a subproblem that has fewer branches means that backtracking will occur earlier rather than later. Since backtracking is not 'forward motion' in finding a solution, we should minimise it as much as possible. In fact, by using this tweak in Sudoku DLX, most easy/medium puzzles are solved without backtracking at all. ■