27Feb2023

**Exact Cover, Algorithm X, Dancing Links**

[ExactCover](#ExactCover)  
[AlgorithmX](#AlgorithmX)  
[DancingLinks](#DancingLinks)

Exact Cover

1. Exact Cover is the name given to a situation involving a set and it’s subsets, whereby a list of subsets is required whose union without overlapping subset elements (ie. empty intersections - disjointed) exactly describe (cover) the master set.

This problem is best illustrated by the following simple example (taken from <https://en.wikipedia.org/wiki/Exact_cover> ):  
  
Let S = {*N*, *O*, *P*, *E*} be a collection of subsets of a set *X* = {1, 2, 3, 4} such that:

* *N* = { },
* *O* = {1, 3},
* *P* = {1, 2, 3}, and
* *E* = {2, 4}.

The subcollection {*O*, *E*} is an exact cover of *X*, since the subsets *O* = {1, 3} and *E* = {2, 4} are disjoint and their union is *X* = {1, 2, 3, 4}.   
  
Another, more detailed example is:  
  
Let S = {*A*, *B*, *C*, *D*, *E*, *F*} be a collection of subsets of a set *X* = {1, 2, 3, 4, 5, 6, 7} such that:

* *A* = {1, 4, 7};
* *B* = {1, 4};
* *C* = {4, 5, 7};
* *D* = {3, 5, 6};
* *E* = {2, 3, 6, 7}; and
* *F* = {2, 7}.

Then the subcollection S\* = {*B*, *D*, *F*} is an exact cover, since each element in *X* is contained in exactly one of the subsets:

* *B* = {1, 4};
* *D* = {3, 5, 6}; or
* *F* = {2, 7}.

The relation "contains" in the [example](https://en.wikipedia.org/wiki/Exact_cover#Detailed_example) above can be represented by a 6×7 incidence matrix:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| ***A*** | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| ***B*** | **1** | 0 | 0 | **1** | 0 | 0 | 0 |
| ***C*** | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| ***D*** | 0 | 0 | **1** | 0 | **1** | **1** | 0 |
| ***E*** | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| ***F*** | 0 | **1** | 0 | 0 | 0 | 0 | **1** |

Again, the subcollection S\* = {*B*, *D*, *F*} is an exact cover, since each element is contained in exactly one selected subset, i.e., each column contains a 1 in exactly one selected row, as the highlighting makes clear.

1. **Pentomino, Sudoku, N Queens** examples:

(See the article for a description of these puzzles and how to go about defining the constraints)

Algorithm ‘X’

Taken from: <https://garethrees.org/2007/06/10/zendoku-generation/#figure-2>

“Algorithm X” is a name given by D.Knuth to an algorithm for solving the “Exact Cover” problem. See <https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X> for an example of how to use this to solve the 2nd problem in “Exact Cover” above. This algorithm is defined as follows:

# If the matrix *A* has no columns, the current partial solution is a valid solution; terminate successfully.

1. Otherwise choose a column *c* ([deterministically](https://en.wikipedia.org/wiki/Deterministic_algorithm)).
2. Choose a row *r* such that *Ar*, *c* = 1 ([nondeterministically](https://en.wikipedia.org/wiki/Nondeterministic_algorithm" \o "Nondeterministic algorithm)).
3. Include row *r* in the partial solution.
4. For each column *j* such that *Ar*, *j* = 1,   
    for each row *i* such that *Ai*, *j* = 1,   
    delete row *i* from matrix *A*.  
    delete column *j* from matrix *A*.
5. Repeat this algorithm recursively on the reduced matrix *A*.

# The partial solution is a valid solution when each column in the solution is constrained; ie the total number of constrained columns in the solution equals the total number of constraint columns in the incidence matrix (the constraints table).

The following is a more realistic illustrated example (but not covering columns) taken from  
https://garethrees.org/2007/06/10/zendoku-generation/#figure-2

**4.1 Example: solving a 2×2 Latin square**

I’ll illustrate the constraint satisfaction approach with a very (very!) small problem: the solution of 2×2 Latin squares. A Latin square is like a sudoku except that there are no blocks, so in this case we have to put the numbers 1 and 2 into the cells of a 2×2 grid so that there is a 1 and a 2 in each row and each column (see [figure 1](https://garethrees.org/2007/06/10/zendoku-generation/#figure-1)).

|  |
| --- |
|  |

You should find it straightforward to check by hand that there are exactly two solutions!

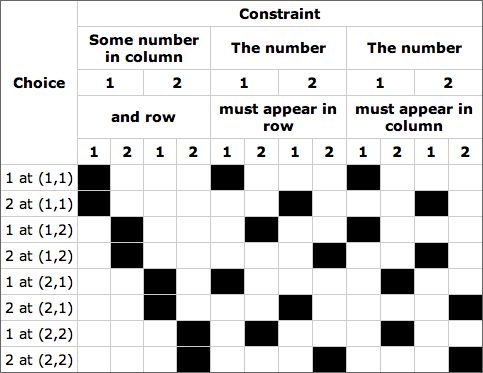
We begin by building a matrix whose columns are the constraints and whose rows are possible choices of placement. In the case of 2×2 Latin squares, there are 12 constraints (3 types of constraint, 2×2 = 4 constraints of each type).

* Constraints 1–4: We must place some number in each cell.
* Constraints 5–8: We must place each number somewhere in each row.
* Constraints 9–12: We must place each number somewhere in each column.

There are 8 choices of placement: either number may be placed in each cell. Expressed as a binary matrix, the combinations of constraints satisfied by each choice is shown in [figure 2](https://garethrees.org/2007/06/10/zendoku-generation/#figure-2).

Note that each of the 3 constraint columns may have their constraint definitions set differently. For example, Constraint column 1 may become   
 **“Some number in row  
 1, 2   
 and column   
 1, 2, 1, 2”**

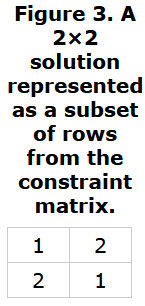
Or, Constraint column 2 may become  
 **“The row  
 1, 2  
 must have number  
 1, 2, 1, 2”**

**Figure 2. Constraint matrix for a 2×2 Latin square.**  


The size of this table should make it clear why I’ve chosen such a small example: the smallest sudoku, 4×4, has 64 constraints and 64 ways of placing a number, while a 9×9 sudoku has 324 constraints and 729 ways of placing a number. You can see the entire matrix for the latter [illustrated in glorious ASCII](http://www.stolaf.edu/people/hansonr/sudoku/exactcovermatrix.htm) by Robert Hanson.

So how does this constraint matrix help? Well, every solution to the Latin square corresponds to a subset of rows from this matrix such that each constraint is covered exactly once—that is, each column has a single black square in the chosen subset of rows. Computer scientists will recognize this as an instance of the well-known NP-hard [EXACT COVER problem](https://en.wikipedia.org/wiki/Exact_cover).

[Figure 3](https://garethrees.org/2007/06/10/zendoku-generation/#figure-3) shows the rows corresponding to a solution to the 2×2 Latin square. Note that each column contains a single black square.





**4.2 “Algorithm X”**

We can find solution sets, if any exist, by a straightforward application of depth-first search with backtracking. In particular, we can incrementally build up a set of rows making up a solution as follows:

1. Pick an unsatisfied constraint: that is, a column with no black cell in any of the rows in the solution set (at initialization choose column 1). If there are no unsatisfied constraints remaining, the solution set is complete: if all we wanted was any solution, we’re done; if we want all solutions, then make note of the solution we’ve got and then backtrack to the previous time we chose a row and take the next choice instead.
2. Pick a row that satisfies that constraint: that is, one with a black cell in the chosen column. If there is no such row, then we’ve reached a dead end and we must backtrack to the previous time we chose a row and take the next choice instead.
3. Add that row to the solution set.
4. Delete all rows that satisfy any of the constraints satisfied by the chosen row: that is, all rows that have a black cell in the same column as a black cell in the chosen row.
5. Return to step 1.

Donald Knuth [comments](http://lanl.arxiv.org/pdf/cs/0011047) on this algorithm, “I will call [it] algorithm X for lack of a better name […] Algorithm X is simply a statement of the obvious trial-and-error approach. (Indeed, I can’t think of any other reasonable way to do the job, in general.)”

The choice of constraint at step 1 doesn’t affect the correctness of the algorithm, but makes a great deal of difference to its running speed. The obvious heuristic is to always pick the constraint satisfied by the smallest number of rows, as this reduces the branching factor of the search. Knuth calls this the “*S* heuristic”; see [section 4.4](https://garethrees.org/2007/06/10/zendoku-generation/#section-4.4) for how well it performs in sudoku.

The choice of row at step 2 doesn’t affect the correctness of the algorithm either, so long as the backtracking makes sure to visit all satisfying rows, but if there are multiple solutions the choice does affect the order in which the solutions are generated.

[Figure 4](https://garethrees.org/2007/06/10/zendoku-generation/#figure-4) shows this algorithm applied to our example. The algorithm makes a bad decision at step 6 and has to backtrack.

|  |
| --- |
| **Figure 4. Solving the 2×2 Latin square using Algorithm X.** |
| 1. Pick an unsatisfied constraint. 2. Pick a row satisfying that constraint. |
| 3. Add the row to the solution set. 4. Delete all rows that satisfy any of the constraints satisfied by the chosen row. |
| 5. Pick an unsatisfied constraint. 6. Pick a row satisfying that constraint. |
| 7. Add the row to the solution set. 8. Delete all rows that satisfy any of the constraints satisfied by the chosen row. |
| 9. Pick an unsatisfied constraint. 10. No rows! Backtrack to last choice. |
| The last choice was at step 6, the selection of a row that satisfies the fourth constraint, so we go back and pick the next row satisfying the constraint. |
| 11. Pick next row satisfying the constraint. |
| 12. Add the row to the solution set. 13. Delete all rows that satisfy any of the constraints satisfied by the chosen row. |
| And so on. The remaining two rows will be added to the solution set one at a time. It doesn’t matter which choices are made now, the algorithm can make no more mistakes. |

Dancing Links method

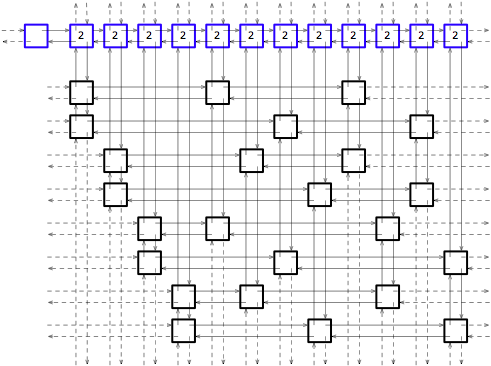
Taken from: <https://garethrees.org/2007/06/10/zendoku-generation/#figure-2>

**4.3 “Dancing Links”**

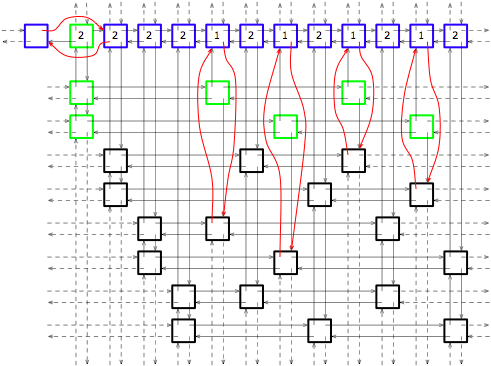
Donald Knuth describes an efficient technique for implementing Algorithm X, which he calls “[Dancing Links](http://lanl.arxiv.org/pdf/cs/0011047)”. This starts with the observation that in typical instances of the EXACT COVER problem, the number of constraints satisfied by each row is quite small. This is particularly true in 9×9 sudoku, where the full constraint matrix has 324 columns and 729 rows—236,196 cells in all—but only 2,916 of those cells are occupied.

We need to be able to search efficiently along columns (in steps 2 and 4) and along rows (in step 4), so we’ll make linked lists for each row and column. We need to be able to efficiently remove cells from their column (in step 4) so we’ll use doubly-linked lists. This means that each occupied cell has four pointers, going to the occupied cells to the left, right, above and below. We also need to be able to pick the best column to search (at step 1) so we’ll have additional data structures representing columns, with a count of the number of occupied cells in each.

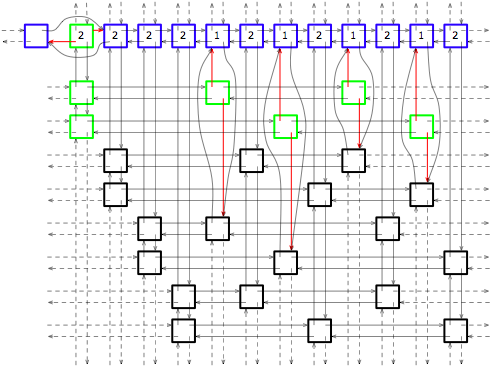
[Figure 5](https://garethrees.org/2007/06/10/zendoku-generation/#figure-5) shows the initial setup of the matrix ready for searching. The blue cells are the column headers, each storing the number of cells in that column. The extra cell at the top left is the root of the whole data structure; it’s there so that we can find the column headers of the constraints that we haven’t satisfied yet. The dashed lines indicate links that wrap around the diagram.

**Figure 5. Dancing Links matrix for the 2×2 Latin square.** 

The basic operation is to remove a column from the matrix, together with all the rows that intersect that column. This operation can be used to implement step 1 and step 4. Knuth calls this “covering a column”. [Figure 6](https://garethrees.org/2007/06/10/zendoku-generation/#figure-6) shows the matrix after covering the leftmost column. The green cells have been removed from the matrix; the red arrows show the links that have been altered.

**Figure 6. “Covering” a column.** 

The key observation that makes Dancing Links efficient is that as long as you store a pointer to the column header, the “covering” operation is easy to reverse. That’s because the cells that have been removed from matrix (coloured green in [figure 6](https://garethrees.org/2007/06/10/zendoku-generation/#figure-6)) still have pointers to their neighbours; they can be used to reverse the operation; Knuth calls this “uncovering a column”. In [figure 7](https://garethrees.org/2007/06/10/zendoku-generation/#figure-7) these pointers are marked in red.

**Figure 7. The pointers that can be used to “uncover” the column.** 

We need to take some care when backtracking to uncover the columns in precisely the reverse order to that in which we covered them. For the fiddly details, see Knuth’s “[Dancing Links](http://lanl.arxiv.org/pdf/cs/0011047)

Sudoku:

For a general discussion on the application of the above to generating and solving Sudoku puzzles, see <https://garethrees.org/2007/06/10/zendoku-generation/#figure-2>