**1.0 Exact Cover Problem**

*What is an exact cover problem?*

As Donald Knuth puts it: “Given a matrix of 0s and 1s, does it have a set of rows containing exactly one 1 in each column?” [1]. This general approach will be the first step toward solving the N-Queens problem. This matrix will be called the 1-0 matrix henceforth.

The 1-0 matrix has a particular structure in the context of this problem, namely: The columns correspond to the satisfaction of the problem’s constraints, while the rows correspond to all the possible queen placements. For each placement, there will be a 1 for each satisfied constraint, and a 0 otherwise. Details of the constraints are discussed below, but first we must define the board and coordinates clearly.

In standard chess we refer to horizontal rows of the board as ranks, and vertical columns as files. Although in chess it is standard to begin counting at 1, in this context we will begin at 0 (We are using python after all).

Here we can see the ranks and files of a size n=4 chessboard:

The Ranks and Files of a 4x4 Board


Figure 1: The Ranks and Files of a 4x4 Board

Now we must discuss how this matrix is formed for a board of size n.

**2.0 Forming the 1-0 Matrix**

Speaking technically, we will not solve the actual problem using a matrix or NumPy array, instead we will create a set of doubly linked circular lists, running vertically and horizontally. However, it seems the most direct way to create this linked list, is to first create a NumPy array, with the 1s and 0s correctly filled, and then to convert it into a doubly linked circular list.

The columns of the 1-0 matrix correspond to the various constraints in the problem, namely two primary and two secondary constraints:

**Primary**

1.) There must be one and only one queen per rank.

2.) There must be one and only one queen per file.

**Secondary**

3.) There can be at most one queen per diagonal.

4.) There can be at most one queen per reverse-diagonal.

The rows of this matrix correspond to each of the possible queen placements, for example a queen at (0,0) or at (2,3).

**2.1 Numbering the constraints**

It is worth noting here that we can disregard the four diagonals of length one, namely the two diagonals at points (0, n) and (n, 0) and the two reverse-diagonals at (0, 0) and (n, n). If a queen is occupying a diagonal of length one, it is simply not possible to have another queen occupy the same diagonal, therefore there is no need to ‘record’ whether or not such a constraint has been satisfied.

There are exactly *2n – 3* diagonals (and reverse-diagonals) on the board.

Consider the uppermost rank on the board and the first file (the left and top edges), each square has one diagonal originating from it, and traversing the board toward the bottom right. This would give us a total of 2n diagonals, n for each side. We must subtract two for the aforementioned diagonals of length one, which originate in the corners, and also subtract an additional one for the ‘middle’ diagonal, which will be counted twice. This logic shows there are 2n-3 diagonals (and similarly 2n-3 reverse-diagonals).

It is helpful to number these constraints, as these numeric labels will correspond to the constraints (the columns) of the 1-0 matrix.

The rank constraints will be numbered *0,1, …, n-1.*

The file constraints will be numbered *n, n+1, …, 2n-1.*

The diagonal constraints will be numbered *2n, 2n+1, …, 4n-4.*

The reverse-diagonal constraints will be numbered *4n-3, 4n-3, …, 6n-7.*

Below we can see a simple diagram illustrating these constraints for a n=4 chessboard.

The diagonals are coloured in blue, while the back diagonals are coloured in red.

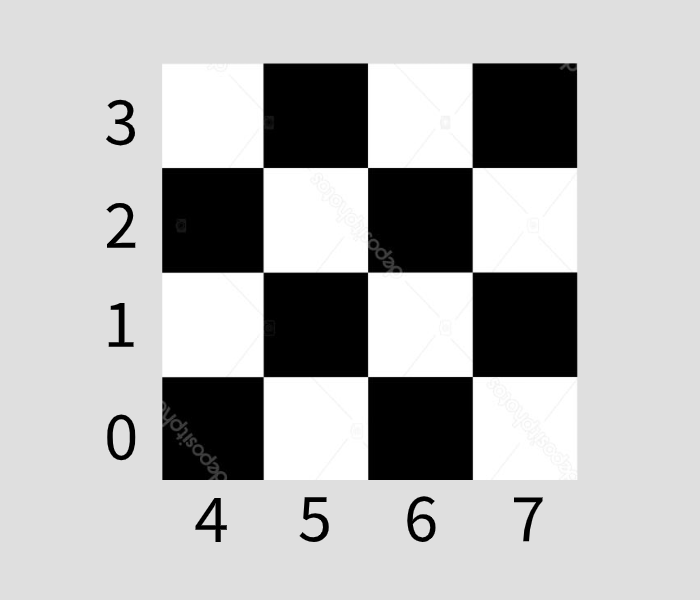


Figure 2: The Rank and File constraints

Shape

Description automatically generated

Figure 3: The Diagonal (Blue) and Reverse-Diagonal (Red) constraints

**2.2 A method for calculating the constraints**

Next, for a given space on the board, we must calculate which number row, column, diagonal and reverse-diagonal (if any) it occupies, so that we may construct the matrix.

Consider a queen placed at coordinate

The rank constraint is equal to x, while the file constraint is equal to y + n.

The diagonal and back diagonal constraints are more difficult to calculate, however this becomes easier when we realise that for each square lying on a given diagonal, the sum of the x and y coordinates is the same, and for each square lying on a reverse-diagonal, the difference *y-x* of the coordinates is the same.

Consider the diagonal of length three, originating at the point (0,2), which passes through the points (1,1) and (2,0). This is labelled #9 on the diagram above. For each of the points on this diagonal the sum of the x and y coordinates is 2.

Consider a general diagonal of unknown length L, say it originates at a square (0, k). It will pass through the square one space below and to the right, which is (1, k - 1).

This pattern continues, as the diagonal is traversed, we move one square down and one to the right, a square (i, j) becomes (i + 1, j – 1) and the sum of these coordinates remains the same.

Thus, we can define the diagonal constraint:

If D does not lie in this range, then the square must lie on one of the diagonals of length 1 and as such we disregard it.

For a reverse-diagonal originating from the point (i, j), the next point on the line is (i+1, j+1) as the diagonal ‘traverses’ the board toward the upper right. Now instead of adding the x and y coordinate we take the difference between them.

For example, consider the reverse-diagonal labelled #14 above, it originates from the point (1,0), and also includes the points (2,1) and (3,2). Here the difference, y-x, remains constant as -1 throughout the diagonal.

Thus, we can define the reverse-diagonal constraint, B:

For the example above, we have 5(4)-5-1=14, as we expect.

**2.3 Calculating the constraints**

Now we can calculate all the constraints which are satisfied by the various positions of queens:

We can see the python output for the case n=4 here, with the constraints roughly labelled following the outline specified before; ranks 0 through n-1, files 0 through n-1 then diagonals and reverse-diagonals. Along the rows we can see the various queen placements, first (0,0) through (0, n-1), then (1,0) through (1, n-1) and so on.

A screen shot of a computer

Description automatically generated with low confidence

Figure 4: Labelled console output of the 1-0 matrix

**3.0 Conversion to a doubly linked list**

Three separate classes are used for the formation of this list, first a *CircularList* class, next a *column* class and lastly a *node* class.

The node class simply contains four attributes: up, down, left and right. These point to the other nodes around this node and will be used in the DLX algorithm to cover and uncover rows.

The column class contains the same four attributes as the node class, as well as two additional attributes: name and size. Name is a string which will allow humans to interpret which constraint this column represents (Rank 0 or Diagonal 3 for example), size is an integer which refers to the number of nodes or 1s, that are below a particular column header, and is used to select optimal rows during DLX.

The *CircularList* class contains only one attribute, a column object with the name ‘Master’, the entire list object will be built around this, and during DLX it will serve as a starting pointer to traverse the list. This master node lives left of the first column header, and as such the up, down and size attributes are not used.

Diagram, schematic

Description automatically generated

Figure 5: Illustration of a doubly linked circular list from Donald Knuth's Dancing Links paper

A class specific function *convert\_one\_zero*, is used to convert the 1-0 matrix into a circular list object. Briefly in pseudocode:

Repeat for each column of 1-0 matrix:

Create column node

Name column node

Connect column node to the previous column node

Repeat for each row of 1-0 matrix:

Search the row

If row[i] is 1:

Create a node

If not first node in row:

Connect *node.left* to the previous node

Find the corresponding column header

Search until the bottom of this column/vertical list is found

Connect below to node

Repeat for each column:

Traverse column to the bottom

Update *column.size* accordingly

Connect ‘lowest’ *node.down* to the header

Connect *header.up* to the ‘lowest’ node

Connect furthest right column *header.right* to the master

Connect *master.left* to the furthest right column header

This function converts the 1-0 matrix into a doubly linked circular list.

**4.0 References**

**[1] Paper (Donald E. Knuth):**

“Dancing Links”, [available here.](https://arxiv.org/pdf/cs/0011047.pdf)