Examining Donald Knuth's Dancing Links Technique by Efficiently Solving Exact Cover Problems

Final Year Project

Oisín Hodgins, Seán Monahan

Supervised by Professor Götz Pfeiffer

April 2021

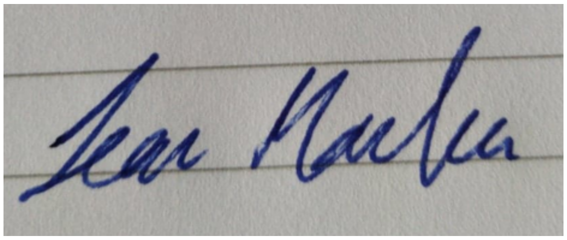
School of Mathematics, Statistics and Applied Mathematics
National University of Ireland, Galway

NUI Galway
OÉ Gaillimh

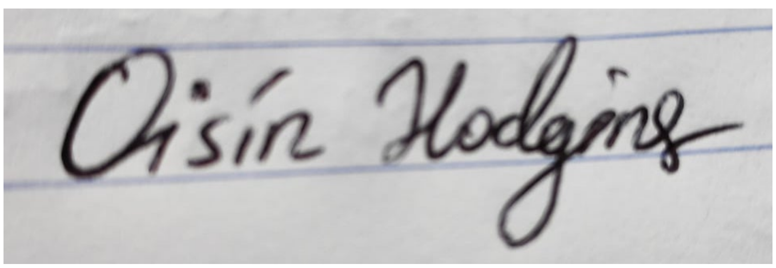
Contents

1 Declaration page

We hereby certify that this material, which we now submit for assessment on the programme of study leading to the award of degree, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within our document.



Signed: Sean Monahan.
Student ID: 17372341
Date: 21/04/2021



Signed: Oisín Hodgins.
ID: 17480216,
Date: 21/04/2021

2 Introduction

Donald Knuth's *Dancing Links* is a technique that cleverly exploits a data structure called doubly linked lists to vastly improve solving times of the exact cover problem. In this paper we will illustrate all aspects of the technique, detailing the exact cover problem that Dancing Links facilitates, its application to puzzles such as the N Queens problem and Latin Square, its advantages over classic backtracking techniques, and how it can be implemented. Our primary source of information will be Knuth's paper [9], where he first introduced the Dancing Links technique. Supplementary to this we have Knuth's more recent publication "The Art of Computer Science Vol. 4B: Fascicle 5" [10, 7.2.2.1] which provides additional insight on the topic.

3 History

In this section we will give some brief context to the three pieces of literature that are central to the dancing links technique.

Table 1 Timeline

1972	Dijkstra publishes *Structured Programming* [7].
1979	Hitotumatu and Noshita publish a short paper entitled *A technique for implementing backtrack algorithms and its application* [8].
2000	Knuth publishes his paper *Dancing Links* [9].

Dijkstra provides a basis for the classical approach which utilises boolean arrays to solve the exact cover problem in [7, p.72–82]. His algorithm is used as the foundation for Hitotumatu and Noshita's *Algorithm H*, published seven years later in [8]. Their paper was overlooked at the time, and is credited by Knuth as the first known implementation of the Dancing Links technique. Over twenty years later, Knuth popularises the technique used in *Algorithm H*, dubbing it Dancing Links. In his paper, this technique is used in conjunction with *Algorithm X*, which is also based on Dijkstra's approach.

4 Exact Cover Problem

To give the full context of what the Dancing Links technique achieves, we first need to examine the type of problem that it handles, the exact cover problem. From [2] we have the following definition: given a set X, and a collection of its subsets S, the definition of an **exact cover** is the sub-collection S^* of S in which each element of X appears in exactly one subset of S^*. In an **exact cover problem** we are concerned with determining whether an exact cover exists. One method of representing an exact cover problem is by using a matrix.

$$M = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

In this example, the elements of set X in are the columns of matrix M, and the collection of subsets S are the rows of matrix M. Our exact cover S^* can be expressed as matrix M^*.

$$M^* = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

The rows of M^* are our set S^*, where M^* corresponds to rows 1, 4 and 5 of the matrix M. One important aspect of this matrix is that fact that each column has exactly one 1. This fulfills our definition of an exact cover given above, where we needed each element of X to appear in exactly one subset of S^*. Exact cover problems can be applied to puzzles including Sudoku and the N Queens problem. We'll first consider the Latin Square puzzle to help illustrate how the exact cover problem is applied.

4.1 Latin Square Problem

	0	1	2
0	A	B	C
1	B	C	A
2	C	A	B

A Latin Square is a puzzle similar to Sudoku that was accredited to Swiss mathematician Leonhard Euler, where an NXN grid must be filled with N distinct shapes, such that no two identical shapes share the same column or row. Here we include notation that denotes where each symbol A, B, C lie, i.e. in above 3X3 example, A lies at $(0,0)$ and $(1,2)$ and $(2,1)$. As an example we will construct a simple $2X2$ Latin Square, posing the puzzle as an Exact Cover Problem. The grid below illustrates the $2X2$ grid including coordinates before any symbols are placed.

	0	1
0		
1		

The first step in formulating our exact cover problem is to break down the **constraints** of the 2 X 2 puzzle, where C_i refers to the i^{th} constraint.

- First, we consider the fact that each square in the grid needs a **symbol**:
  - C_1 : (0,0) must have a symbol
  - C_2 : (0,1) must have a symbol
  - C_3 : (1,0) must have a symbol
  - C_4 : (1,1) must have a symbol
- Next we consider the **row constraint**:
  - C_5 : Symbol A must appear in row 0
  - C_6 : Symbol B must appear in row 0
  - C_7 : Symbol A must appear in row 1
  - C_8 : Symbol B must appear in row 1
- Finally we consider the **column constraint**:
  - C_9 : Symbol A must appear in column 0
  - C_{10} : Symbol B must appear in column 0
  - C_{11} : Symbol A must appear in column 1
  - C_{12} : Symbol B must appear in column 1

These are our 12 constraints for the puzzle. Now we can construct a checklist, where C_i are the constraints, and S@(x,y) refers to the symbol at coordinates (x,y)

Table 2 Constraints table for the 2 X 2 Latin Square problem

	Constraints											
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}
$A@(0,0)$	✓	X	X	X	✓	X	X	X	✓	X	X	X
$A@(0,1)$	X	✓	X	X	X	✓	X	X	✓	X	X	X
$A@(1,0)$	X	X	✓	X	✓	X	X	X	X	✓	X	X
$A@(1,1)$	X	X	X	✓	X	✓	X	X	X	✓	X	X
$B@(0,0)$	✓	X	X	X	X	X	✓	X	X	X	✓	X
$B@(0,1)$	X	✓	X	X	X	X	X	✓	X	X	✓	X
$B@(1,0)$	X	X	✓	X	X	X	✓	X	X	X	X	✓
$B@(1,1)$	X	X	X	✓	X	X	X	✓	X	X	X	✓

Using Table 1, we can construct an exact cover problem matrix by simply changing ✓ to ones and X to zeros while retaining the $8X12$ shape of the table, which we will denote as M for this example:

$$
M = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

Using M we note that, in the context of puzzles, the constraints are our elements of set X, and the collection of all possible moves is our collection S. Now that we have our problem posed in a suitable format, let's venture into our goal, as Donald Knuth posed in [9, p.3]:

> "Given a matrix of 0s and 1s, does it have a set of rows containing exactly one 1 in each column?"

4.2 Solution to the Latin Square

Deducing a solution through trial and error is quite trivial in this case since there are so few options. One might deduce the following solution:

	0	1
0	A	B
1	B	A

This gives us one possible solution with the following choices; A@(0,0), B@(0,1), B@(1,0), and A@(1,1). We can eliminate the other options in the table:

Table 3 Constraints table for the 2 X 2 Latin Square problem

	Constraints											
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}
$A@(0,0)$	✓	X	X	X	✓	X	X	X	✓	X	X	X
$A@(1,1)$	X	X	X	✓	X	✓	X	X	X	✓	X	X
$B@(0,1)$	X	✓	X	X	X	X	X	✓	X	X	✓	X
$B@(1,0)$	X	X	✓	X	X	X	✓	X	X	X	X	✓

In Table 2 we see the solution to the problem, where every constraint is filled exactly once. We can denote in matrix form as:

$$M^* = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

We see that the rows of M^* correspond to an exact cover S^*, as we see exactly one 1 in every column. This was a very simple example of an exact cover problem where a trial and error approach gives us the answer quickly, so we'll introduce a more complex problem that will be the main use-case of our python implementation.

4.3 N Queens Problem

Here, we will discuss the specifics of the N Queens problem, a classic chessboard puzzle that is commonly used by programmers to test the capabilities of a backtracking algorithm. We chose to focus our attention on this particular problem as it was also the focus of Dijkstra's [7] and consequently of Hitotumatu and Noshita's implementations in [8]. The N Queens puzzle requires N Queens to be placed on a N by N chessboard such that no two Queens oppose each other. This means that Queens cannot share the same column, row, diagonal, and reverse diagonal, as seen in the figure 1 which was created using lichess. [3].

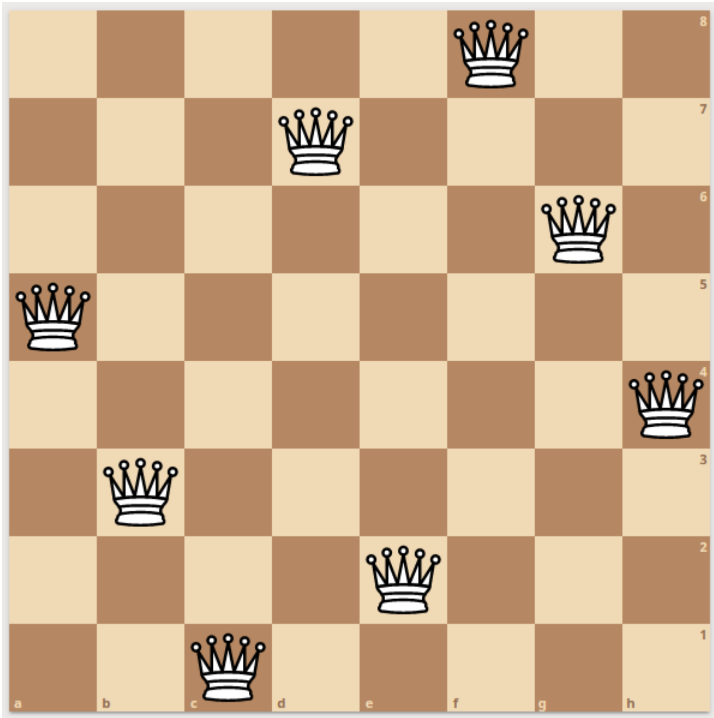


Figure 1: An 8 Queens Solution

The N Queens variant of the exact cover problem differs from the Latin Square as its rules can be broken down into both primary and secondary constraints. Our primary constraints necessitate that each row has a Queen, and each that column has a Queen. These primary constraints also fulfill the requirement that Queens cannot share a row or column. Our secondary constraints are that there can be at most one Queen in each diagonal and reverse diagonal on the board. These are considered secondary constraints since **we do not require each diagonal and reverse diagonal to have a Queen**. The existence of secondary constraints is why the N Queens problem is considered to be a *generalized* exact cover problem.
We have the following sets of constraints (for $N \geq 2$):

- Exactly one Queen must appear in each row. We have N rows, so this set accounts for a total of N constraints.
- Exactly one Queen must appear in each column. We have N columns, so this set accounts for another N constraints.
- At most one Queen may appear in each diagonal. We have $2N-3$ non-trivial diagonals, so this set accounts for a total of $2N-3$ constraints.
- At most one Queen may appear in each reverse diagonal. We, again, have $2N-3$ non-trivial diagonals, so this set accounts for another of $2N-3$ constraints.

Thus by simple addition we have $N + N + (2N - 3) + (2N - 3) = 6(N - 1)$ constraints. We are free to place a Queen in any square on the board, which yields N^2 possible moves. When considering this in a matrix representation, our exact cover has N^2 moves (rows), and $6 * (N - 1)$ constraints (columns).

For large N one can imagine that it would be nearly impossible to find all solutions by hand. To motivate a computational approach to the problem, we'll look at its NP-Complexity.

4.4 NP-Complexity

Polynomial time is a metric used to classify the complexity of a decision problem. From [4], an algorithm is defined to be of polynomial time if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm i.e., $T(n) = O(n^c)$ for some positive constant c. In other words, polynomial time is a measure of the efficiency of the algorithm, which measures the growth of the number of operations relative to the size n of the problem. **NP**, an abbreviation of "nondeterministic, polynomial time", refers to a problem that is solvable by a nondeterministic Turing Machine in polynomial time. Essentially, what we would need is a computer program which can output enough random "guesses" will be able to solve the problem within a realistic amount of time. An **NP-Complete** problem is one that upholds the definition of an NP problem, but can also be used for other problems with similar solvability. In this project we will use algorithm X using the Dancing Links technique developed by Donald Knuth to fulfill the requirements of an NP-Complete problem.

5 Algorithm X

Algorithm X a **nondeterministic**, **recursive**, **depth-first backtracking** algorithm used to find all solutions to the exact cover problem defined by any given matrix M of 0s and 1s. Before we go into the algorithm itself, let's review the definitions and importance of these properties to gleam some understanding of the strengths of algorithm X.

Nondeterministic algorithms, given a particular input, will not always produce the same output. This is essential for an exact cover problem as the algorithm isn't restricted to finding only one solution. This is illustrated in the figure 2, where we see that nondeterminstic algorithms provide a branching path to that solution, whereas a deterministic algorithm is only capable of finding at most one solution.

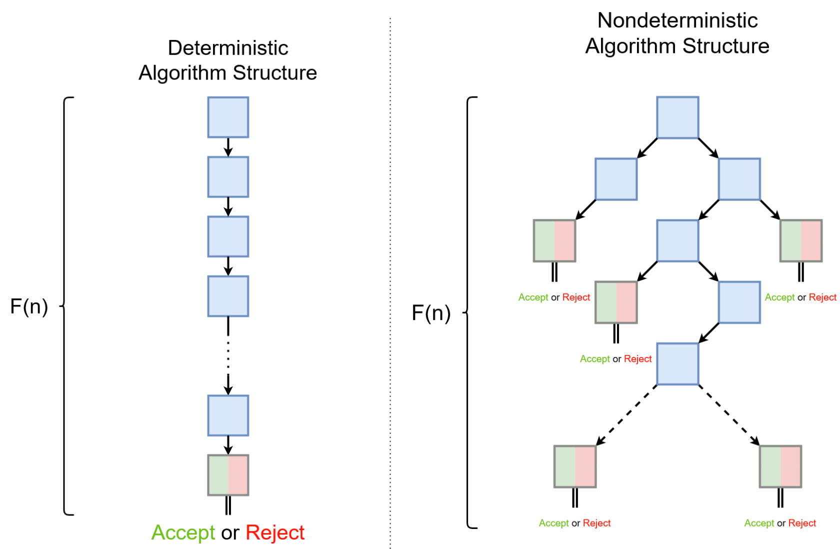


Figure 2: Deterministic vs. Nondeterministic Algorithms

Recursive algorithms will rerun subroutines and functions until a specific condition is met, allowing efficiency by reusing a block of code until a solution is found. This allows for a simpler and more succinct algorithm in contrast to an iterative approach. It also leads to the nondeterministic branching structure seen in figure 2. When incrementally building a solution, **backtracking** allows the algorithm to revisit unseen branches once a solution or invalid path is reached. In figure 2, the algorithm backtracks is when a node labeled *"Accept or Reject"* is reached. **Depth-first backtracking** means that the algorithm will not completely abandon the path when backtracking, and will instead backtrack to the most recently visited node with multiple paths.

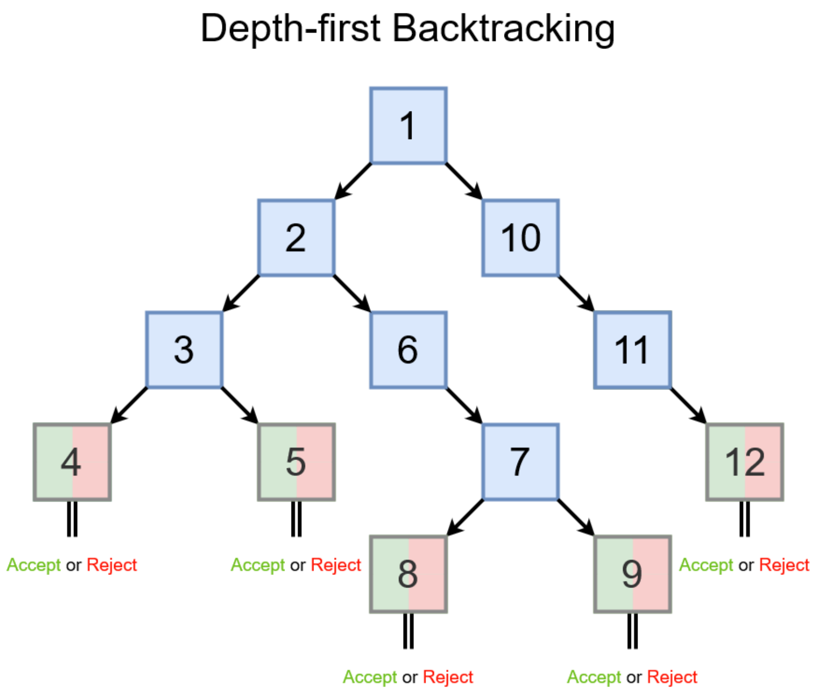


Figure 3: Depth-first backtracking

In figure 3, we see a demonstration of depth-first backtracking, where the numbering of the nodes refers to the order in which they are visited. In the context of the DLX, the algorithm follows the its first path until the node 4 is reached. This is either a solution, or an erroneous path that gets discarded, then the algorithm backtracks to the most recent node with an open path, which is node 3, and continues on from there to node 5. Under this method, the algorithm fully explores a branch before visiting a new one.

5.1 Algorithm X Pseudo-code

We will base our pseudo-code on Knuths' in [9, p.4], with some variations.

- Let M be the given zero-one matrix.
- Let U be the set of unsatisfied columns. These are columns that have more than exactly one 1 in its rows.
- Let r_i be the i^{th} row of M.
- Let c_j be the j^{th} column of M.

Using this notation we can define the algorithm as follows:

Algorithm 1 Algorithm X

if U is empty **then**: the problem is solved **return** M
end if
Otherwise choose a column c from set U
Choose a row r of c that has value 1 (nondeterministically)
for each j such that $M[r, j] = 1$ **do**:
    remove column j from set U
    for all i such that $M[i, j] = 1$ **do**:
        delete row i from matrix M
    end for
end for
Repeat this algorithm recursively on the reduced matrix M.

The algorithm works by methodically taking each unsatisfied constraint, randomly choosing one position that can fulfill its constraint and removing other positions that have conflict with that constraint. Then we continue this process until a solution is found. This implementation is slightly different to Knuth's which does not include U, which was added in for illustrative purposes seen in the next section. This should not pose an issue as the algorithm does not have a strict format, as said in [9, p.3], "Algorithm X is simply a statement of the obvious trial-and-error approach".

5.2 Algorithm X with a 2 X 2 Latin Square (by hand)

In this section we will use algorithm X to find one exact cover. Returning to our $2X2$ Latin Square example, we recall the exact cover problem we found:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

1. Following our algorithm, we note that U is not empty as every column is unsatisfied, so we can choose any column in M. Let's choose the third column and highlight it.

$$M^* = \begin{bmatrix} 1 & 0 & \mathbf{0} & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & \mathbf{0} & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbf{0} & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & \mathbf{0} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & \mathbf{0} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & \mathbf{0} & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

2. We see that there are two 1s in this matrix we pick one at random and denote its row in blue. This row is now part of our solution set.

$$M^* = \begin{bmatrix} 1 & 0 & \mathbf{0} & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & \mathbf{0} & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbf{0} & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & \mathbf{0} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & \mathbf{0} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & \mathbf{0} & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Since we want each constraint to be satisfied exactly once, we discard all other rows that fill the constraints of our solution row, which in this case is rows 3, 5 and 8.

$$M^* = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

4. Now we begin the algorithm again and pick an unsatisfied constraint, in this case we'll choose column number 2.

$$M^* = \begin{bmatrix} 1 & \boxed{\mathbf{0}} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & \boxed{\mathbf{1}} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & \boxed{\mathbf{0}} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \boxed{\mathbf{1}} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 \end{bmatrix}$$

5. We choose row number 2 at random, adding it to our partial solution by denoting its row in blue

$$M^* = \begin{bmatrix} 1 & \boxed{\mathbf{0}} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 \\ 0 & \boxed{\mathbf{0}} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \boxed{\mathbf{1}} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 \end{bmatrix}$$

6. We once again delete rows that fulfill constraints of our solution row, in this case we delete rows 1, 3 and 4

$$M^* = \begin{bmatrix} \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 \\ \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}0 & \color{blue}1 \end{bmatrix}$$

7. This partial solution is incorrect. This can be seen in the first, fourth, fifth, eighth, tenth, and eleventh columns that lack 1s. One of the random choices that we made has led us down the wrong path, so we'll need to backtrack to the step number 4, and denote our erroneous path in red.

$$M^* = \begin{bmatrix} 1 & \mathbf{0} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & \mathbf{0} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

8. Now we choose the only other option in our chosen column, row 4, by denoting it in blue.

$$M^* = \begin{bmatrix} 1 & \mathbf{0} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & \mathbf{0} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

9. We again eliminate any conflicting constraints

$$M^* = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

10. And finally we are left with a full solution. Every column has exactly one 1, thus each constraint is fulfilled exactly once. This is our exact cover. To check what our solution looks like, let's recall our table from earlier.

    We find that these rows correspond to the following moves:

    - **A @ (0,0)**
    - **A @ (1,1)**

- **B @ (0,1)**
- **B @ (1,0)**

that leads us to the following Latin Square

	0	1
0	A	B
1	B	A

which is indeed a valid solution.

5.3 Potential for Improvement with algorithm X

We have now seen algorithm X correctly finding a solution to an example exact cover problem, in fact should we continue working through problems by hand we will see that algorithm X finds *all* solutions to a given exact cover problem. This is because the algorithm will iterate over each possible row it can choose in a given column, making one recursive call for each.

However this is a computationally demanding task: each recursive call results in a new subroutine, and that generation of subroutines will generate even more subroutines, naturally resulting in a search tree forming (such as the one seen in figure 2). For any practical real-world problems these search trees have a great many nodes, and an inefficient implementation of algorithm X will not suffice.

In practice it is computationally too expensive to operate using '0-1' matrices, as these matrices will be both large and relatively sparse. In it's simplest implementation algorithm X would require such a matrix to be cloned for each subroutine, with only minor differences between each (reduced with respect to different rows).

In [9, p.4-5], Knuth outlines these issues with a naive implementation, and further states that any improvement will come from an efficient method of narrowing the search as well as storing the state information (the data which represents the exact cover problem).

Using matrices or stacks will not hold up well as we encounter practical problems, however there are other potential solutions out there. One such optimisation is Dijkstra's program for the N Queens problem, as outlined in [7], which uses three global Boolean arrays to store the state information.

Throughout the following sections we will discuss the far more efficient optimisation found by Hitotumatu and Noshita in [8], and popularised by Knuth in [9].

6 A Brief Recap: Linked Lists

Before we can discuss the dancing links technique as a solution to some of the problems associated with algorithm X, we must first recall: *what are linked lists*?

6.1 Singly Linked Lists

Linked lists are one of the fundamental data structures used in programming, and they have been around since the Information Processing Language(IPL) in the 1950's.
These lists are often compared to arrays, the key difference being how their elements are stored in physical memory. Arrays store all of their elements sequentially in memory, allowing for fast search times but require a full copy elsewhere in memory if any new elements are appended, while each element in a linked list points to the next thus allowing for them to be stored non-sequentially, however they have longer search times as a result of this.
A comparison between arrays and linked lists is not the focus of this project, instead we will simply focus our attention on linked lists as the dancing links technique only applies to them (as opposed to arrays).

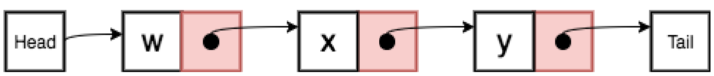


Figure 4: An illustration of a singly linked list

In figure 4 we can see an example of a typical singly linked list. Here each node of the list has several fields, in this case two, the former stores the important data we are concerned with, while the latter points to the memory address of the next node in the list. At the beginning of the list we can see the *head node*, while at the end we can see the *tail node*. These *sentinel nodes* facilitate traversal of the list, the *head node* is a starting point for operations on the list while the *tail node* tells us when we have reached the end of the list.

6.2 Circular Linked Lists

A circular linked list refers to a list where the last node points back to the first one, as opposed to some *tail node*.

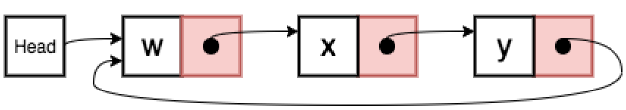


Figure 5: An illustration of a circular singly linked list

As we can see in figure 5 this singly linked list is circular. The final node in the list, which is storing the value y, points back to the first node, which stores the value w.

6.3 Doubly Linked Lists

We have mentioned that each node can have a number of distinct fields in a linked list, and these fields can either hold data or pointers. We refer to lists where the nodes have multiple pointers as *multiply linked lists*, a name which we adjust for the number of links.

For example we now consider a *doubly linked list*, where each node has two pointers, often refereed to as '*prev*' and '*next*' however in for the purposes of this project we will refer to them as: '*left*' and '*right*'.

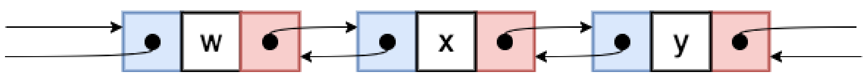


Figure 6: An illustration of a circular doubly linked list

We can see an example of such a list in figure 6, which is also a circular list. Here the *left* pointer fields have been highlighted in *blue* and the *right* pointer fields in *red*. We can see that the first node, storing the value w, points to the last node on the left, and similarly the last node points to the first on the right (these links extending to the edge of the figure are intended to *wrap around* to the far side).

7 Dancing Links

Dancing links is a technique that was named and popularised by Donald Knuth in [9] which cleverly utilizes doubly linked lists to make a simple "covering" function. To explain the core concept of Dancing Links, we'll first introduce the following operations. Suppose we have a doubly linked list with that has an node x. We define $L[x]$ and $R[x]$ to be a link pointing to the predecessor and successor of x, respectively.

$$L[R[x]] \leftarrow L[x], \qquad R[L[x]] \leftarrow R[x]$$

We see above that in the left operation the predecessor of the successor of x, $L[R[x]]$, is assigned to the $L[x]$, thus removing x from the list. Similarly in the right equation, the successor of the predecessor of x, $R[L[x]]$, is assigned to the $R[x]$, thus removing x from the list. An illustration of the current state of the list after these operations is seen in figure 7.

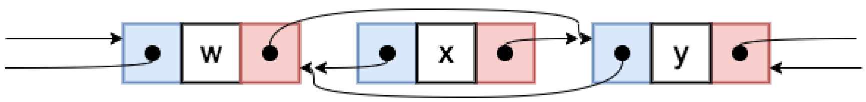


Figure 7: Doubly linked list with node x removed

Note that node x retains its information on nodes w and y. This is a simple operation that was well known before Knuth's paper, essentially we "cut out the middleman" to remove an unwanted node x. We call this operation **covering**. But what if we want to reintroduce x?

$$L[R[x]] \leftarrow x, \qquad R[L[x]] \leftarrow x$$

In the left operation, the predecessor of the successor of x, $L[R[x]]$, is assigned to x and similarly in the right operation, the successor of the predecessor of x, $R[L[x]]$, is assigned to the x. We simply undo the first set of operations, to "reintroduce the middleman" to recover x, so to speak. We call this operation **uncovering**. The list is then restored to its original structure seen in figure 6. It seems intuitive given the first set of operations, but Knuth claims that these subtle operations were overlooked by many computer scientists at the time. He credits Hirosi Hitotumatu and Kohei Noshita with the idea in their paper [8], they originally used these operations in an implementation of the N Queens problem, which *cut solving time almost in half* without adding a significant amount of complexity. Our question now is; how does this translate in to a more efficient algorithm, and how do we apply doubly linked lists to the exact cover problem?

8 Four Way Linked Lists

As discussed in section 7 the **dancing links** operation allows for the easy covering and uncovering of nodes in a *doubly linked list*, therefore in order to use this operation we must store the state information of the exact cover problem in a linked list structure.

Hitotumatu and Noshita [8], used a combination of doubly linked lists and Boolean arrays to store the state information of the **N Queens** problem in their original implementation. This technique was then further expanded by Knuth [9, p.5-8], who instead used a structure which he named the '**four way linked list**'.

In section 5.3 we discussed the inefficiencies associated with algorithm X and we will now see how Knuth's use of four way linked lists counteracts these. This approach allows for the highly efficient storage of state information, which results in each step of the search (algorithm X) being relatively simple and quick, including any necessary backtracking operations.

8.1 Structure of a Four Way Linked List

A four way linked list is made up of two sets of interlocking circular doubly linked lists, one aligned horizontally and one vertically, along with a single 'master' header which serves as a starting point for all operations on the four way linked lists.

We can see an illustration of one of these list objects in figure 8, created using *diagrams.net* [1]:

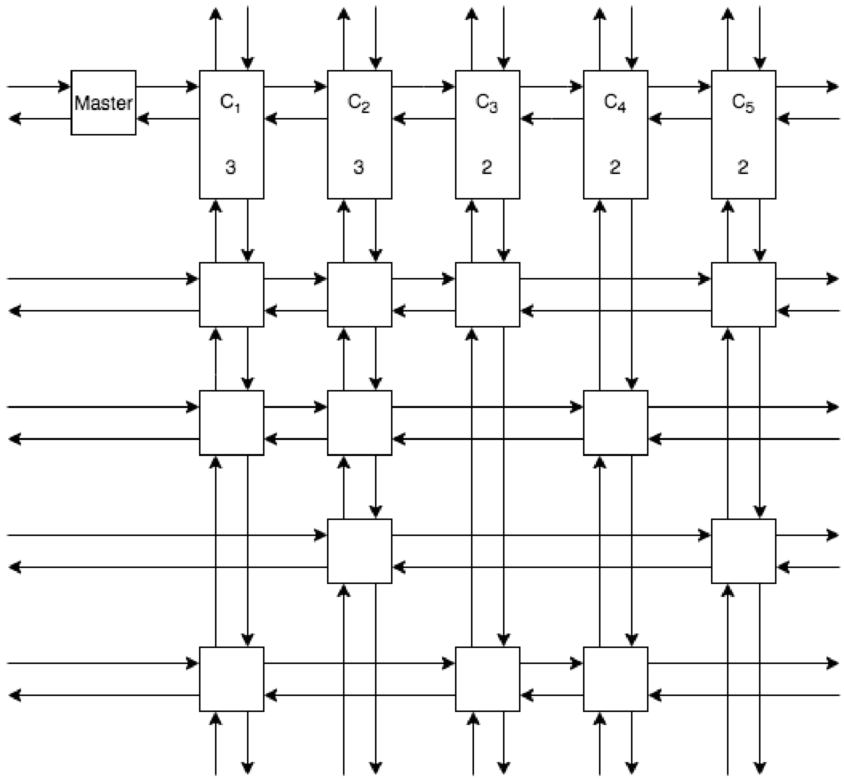


Figure 8: Illustration of four way linked list

Of note here is the upper-most row, which is made up of the column/list headers (including the 'master' header) labelled C_i, these special nodes serve as 'markers' for each column and facilitate operations on the list.
The rows and columns are made up of nodes, with each node's links to its neighbours being represented by the arrows in the figure. Here any links which extend toward the edge of the figure are in fact *wrapping around* the object to connect again on the far side. This is what is meant by a circular list, and is topologically equivalent to a torus.
The data structure as a whole is analogous to a binary matrix, and we can think of its rows and columns as a one-to-one mapping onto such a matrix of 1's and 0's (think: a matrix representing an exact cover problem), where each 1 in the matrix corresponds to a node in the four way linked list.

8.2 The Structure of Nodes

The nodes we can see in figure 8 are similar to the nodes of a typical doubly linked lists, in that they are made up of a number of fields containing pointers or some important data.
In this case each node consists of five fields, all of which are pointer fields. Four of these point to

the node's immediate neighbours: **up**, **down**, **left** and **right**, while the fifth points to the **column header** that this node belongs to (these column links are omitted from all figures throughout the report for clarity and conciseness).
There is no need to store any data in these nodes, therefore no need to specify a field for such a task. Considering that the overall list object is analogous to a binary matrix, we implicitly record the value: '1' through the existence of the node, and similarly record a value: '0' through the lack of a node being present, for example the bottom row in figure 8 corresponds to the row vector $(1, 0, 1, 1, 0)$.

8.2.1 Nodes: Implementation in Python

The default Python list structure that we all know and love, is in fact a dynamic array. The advantages and disadvantages between linked lists and dynamic arrays are not central to this topic, and as such will not be discussed here, the only point we need to take away from this is that *we will have to implement linked lists ourselves.*
This is because we cannot access and change the links of a dynamic array in the same way we could a linked list, it is not a suitable data structure for the problem.

You may ask; " *Why not use a third party library or existing implementation of linked lists?*, this was an option however we decided it would not be in the spirit of the project or serve any educational benefit onto ourselves or the reader (we did however use other modules for a small amount of matrix manipulation and plotting, this is further discussed in sections 2 and 10 respectively).
Instead our approach to this problem was the implementation of a number of classes with the end goal of building a four way linked list, namely a class each for nodes, column headers and the four way linked list itself. In this section we will outline the node class, we can see the code used to declare it here:

```python
class Node:
    def __init__(self, left=None, right=None, up=None, down=None, column=None):
        self.left = left
        self.right = right
        self.up = up
        self.down = down
        self.column = column
```

Figure 9: Declaration of the node class

Here we can see the attributes each instance of the node class will have, and that their values are initially set to a default value of 'None'. We will then specify these attributes later in the program, for each now instance of the node class we create.

A brief description of each attribute is as follows:

Node Class Attributes

Name	**Data Type**	**Description**
Left	Node/Column object	Points to the object left of this node
Right	Node/Column object	Points to the object right of this node
Up	Node/Column object	Points to the object above this node
Down	Node/Column object	Points to the object below this node
Column	Column object	Points to this node's column header

The data type here is not as integral to python code as it would be in say, *C++*, however we still note that each of these pointers stores the memory address of the object it is pointing to. It is also worth noting here that there are no methods for this class, instead all methods and actions performed on them will be implemented in the *four way linked list* class.

The aforementioned 'Column' attribute points to the column/list header which marks the vertical list this node belongs to. From this point henceforth we will simply refer to these as column headers, and we will now discuss them in detail.

8.3 The Structure of Column Headers

The column headers can be considered special cases of regular nodes, as they are functionally similar in that they have the same pointer fields as nodes, however they also have three data fields unlike regular nodes.
The first two of these, namely the **name** and **size** fields, are inherited from Knuth's implementation [9]. The **name** field is a simple string used to identify which constraint a column corresponds to, and during the printing of solutions helps us humans understand the output. The **size** field is an integer value representing the number of nodes in the column header's column, and is considered optional by Knuth however we chose to include it, as it reduces the branching factor of the algorithm considerably. These fields can both be seen in figure 8, where each column header has a visible name field C_i, and size field just below that.
The third field was added in our implementation: '**Primary**', which is a Boolean value used to

distinguish between primary and secondary columns/constraints. Here a secondary constraint can be satisfied *at most once* but can remain unsatisfied for a full solution. We decided to create this attribute in order to more easily implement the motivating example central to our program: The **N Queens problem**, this problem is discussed in more detail in section 4.3.

8.3.1 Column Headers: Implementation in Python

Here we can see the declaration of the column class in Python:

```python
class Column:
    def __init__(self, left=None, right=None, up=None, down=None, size=0, name=None, primary=True):
        self.left = left
        self.right = right
        self.up = up
        self.down = down
        self.size = size
        self.name = name
        self.primary = primary
```

Figure 10: Declaration of the column class

The up, down, left and right attributes behave identically to those of the 'Node' class, and we have already discussed the size, name and primary attributes. Once again these attributes have an default value of 'None' (or 0 for the size attribute as it is an integer value), and will need to be updated after initialisation. The primary attribute has a default value of true, and all headers will keep this default value except in the N Queens problem, where diagonal constraints will have this attribute set to false. It's worth noting here that another solution to the problem of secondary constraints (that can be satisfied at **most** once) is to append one row to the four way linked list for each of these constraints, which has only one node and belongs to the column corresponding to that constraint. In the event a full solution is found, except that the secondary constraint(s) in question have not yet been satisfied, then these 'singleton' rows can be included in the solution.

We decided to use an attribute based approach as outlined earlier instead, because we also required a method to distinguish the 'master' header from the other column headers such that it will never be chosen during program run-time. Knuth suggests setting the master header's size to a infinite or exceedingly large value, and indeed it would only take a single search of the initial row of column headers to determine to largest column size, n, and then set $master.size \leftarrow n{+}1$. However we felt this attribute based approach provided a *cleaner* solution, regardless of any memory/efficiency concerns.

8.4 Four Way Linked List: Implementation in Python

Now that we have declared classes for both nodes and column headers, we can bring them together to declare the final class: **'FourWayLinkedList'**.

```python
class FourWayLinkedList:
    def __init__(self, main_file_name="main_output.txt", log_file_name="log.txt", master_node=Column(name="Master",primary=False)):
        self.main_file = main_file_name
        self.log_file = log_file_name
        self.master_node = master_node
        self.solution_list = []
        self.total_solutions = 0
        self.header_list = []
```

Figure 11: Declaration of the four way linked list class

Each instance of this class will consist of the master node, header list, two file tracking variables and two solution tracking variables. These solution and file tracking variables, namely '*solution_list*', '*total_solutions*', '*main_file*' and '*log_file*' are discussed in detail in section 9.4.6 and for sake of brevity will not be discussed here.

The aforementioned **master header** serves as a *sentinel node* or starting node, meaning we store it's address in local memory as a variable or in this case as an attribute of the class itself, and begin any operations on the list with it.
As can be seen in figure 8 the master header is situated to the left of the four way linked list, and is connected on the left and right the other column headers.
During initialisation we give it a suitable name, '*Master*', and set its primary attribute to be '*False*'. By setting the primary attribute in this way, we ensure the master node will never be chosen by the DLX algorithm as a constraint to cover (Knuth suggests instead setting the size of the master node to be infinite/exceedingly large, however as we have already declared the primary attribute in the 'column' class definition and this attribute is used to exclude non-primary columns during the DLX constraint selection process, it is convenient to use here also).

The header list attribute is used to store the original ordering of the header columns, before they are covered and uncovered during the program. This list can then be used later on to ensure the outputted solutions are ordered correctly, this will be discussed in detail in section 9.4.6.

8.4.1 Initialising: Conversion of a 0-1 Matrix

Now that we have discussed the framework used to implement these *four way linked lists* we can consider the action of initialising an instance of this class.

During the implementation our primary goal was to use the *N Queens problem* as a motivating example of *Algorithm DLX*, which will be discussed soon in section 9. However, as a secondary goal, we also wanted to allow a user of this program to see *algorithm DLX* in action on their own favourite matrix (an imported matrix, from a .csv file or similar) and as such we would need a method to convert '1-0' matrices into four way linked lists. This would allow a user to solve any formulated exact cover problem, such as the our original example the Latin Square puzzle.
As a result of this secondary goal, we decided the best approach would be to create the *N Queens*

'0-1' matrix explicitly as a matrix (using NumPy [5]) and then to create a single method which converts '0-1' matrices into *four way linked lists*.
We defined a method of the *FourWayLinkedList* class for this task, namely the *convert exact cover* method. It can be briefly summarised as follows:

1. Initialise a list of *column headers*, one for each column in the matrix
2. Iterate over each row of the matrix, creating a node for each 1 encountered
3. For each new node, set its *left*, *right*, *up*, *down* and *column* links accordingly

The difficulty here is setting these links in an efficient manner, and the pseudocode in Algorithm 2 briefly describes the exact method used (note: 'Master' is the master header, and 'p' is used to store the previous node or column object).

Algorithm 2 convert_exact_cover

```
Set p ← Master
for each column in matrix do:
    Create Column c
    Set R[p] ← c
    Set L[c] ← p
    Set R[c] ← Master
    Set L[Master] ← c
    Set U[c] ← c
    Set D[c] ← c
    Set p ← c
end for
for each row in matrix do:
    for each j in row do:
        if row[j] is 1 then:
            Create node n
            Set L[n] ← p and R[p] ← n
            c ← R[Master]
            Do j times: c ← R[c]
            Set C[n] ← c
            Set S[C[n]] ← S[C[n]] + 1
            While c ≠ C[n] do: c ← D[c]
            D[c] ← n
            U[n] ← c
            U[C[n]] ← n
            D[n] ← C[n]
        end if
    end for
end for
```

The core principal in Algorithm 2 is to begin with a fully connected list, and insert the new nodes into that list accordingly. We begin with the column headers, and work down through the

matrix row by row.
Each column header is connected to itself above and below initially, and as new nodes are created, we insert them at the bottom of these columns. Each node in a row is connected to the previous node on the left, and for this reason we must keep track of the previous node's pointer, namely p in the method.

We can see the code used for this implementation in figure 12.

```python
def convert_exact_cover(self, matrix, log):
        self.begin_file_writing(log)
        self.file_write_one_zero(matrix)
        dims = np.shape(matrix)
        x, y = dims
        previous_header = self.master_node
        for i in range(y):
            new = Column(left=previous_header, right=self.master_node, name="Constraint {0}".format(i))
            new.up, new.down = new, new
            previous_header.right = new
            self.master_node.left = new
            previous_header = new
        for i in range(x):
            current_row = matrix[i]
            prev_node = None
            for j in range(len(current_row)):
                current_node = Node()
                current_node.left, current_node.right = current_node, current_node
                if current_row[j] == 1:
                    if prev_node is not None:
                        current_node.right, prev_node.right.left = prev_node.right, current_node
                        current_node.left, prev_node.right = prev_node, current_node
                    current_node.column = self.find_column_by_index(j + 1)
                    current_node.column.size = current_node.column.size + 1
                    current_above = current_node.column
                    while current_above.down != current_node.column:
                        current_above = current_above.down
                    current_above.down, current_node.column.up = current_node, current_node
                    current_node.up, current_node.down = current_above, current_node.column
                    prev_node = current_node
        return self.master_node
```

Figure 12: **Convert exact cover** method definition

9 Algorithm DLX

In this section we will discuss *algorithm DLX* in detail along with our implementation of it in Python.
Algorithm DLX is the name given to the implementation of *algorithm X* using the *dancing links* technique (this name is given by Knuth in [9]). In the preceding sections we have discussed both *algorithm X* and the need for improvement with it. We have also outlined the *dancing links* technique and the data structure used to implement it: *four way linked list*.
Now we are in position to discuss and implement algorithm DLX, first let us examine the pseudocode provided by Knuth in [9].

9.1 Pseudocode

This methodology provided by Knuth in *algorithm 3* served as the basis of our own implementation, and the general structure of our program and Knuths are closely related.

Algorithm 3 Algorithm DLX

1: If $R[master] = master$, print the current solution and return.
2: Otherwise choose a column object c.
3: Cover column c (see cover column pseudocode).
4: **for** each $r \leftarrow D[c], D[D[c]]$, ..., while $r \neq c$, **do**
5:     set $O_k \leftarrow r$;
6:     **for** each $j \leftarrow R[r], R[R[r]]$, ..., while $j \neq r$, **do**
7:         cover column j;
8:     **end for**
9:     search(k + 1);
10:     set $r \leftarrow O_k$ and $c \leftarrow C[r]$;
11:     **for** each $j \leftarrow L[r], L[L[r]]$, ..., while $j \neq r$, **do**
12:         uncover column j.
13:     **end for**
14: **end for**
15: Uncover column c and return.

In algorithm 3 there is a lot of information to unpack, and we will outline our approach to each aspect of the code in due course, however first let us jump to lines 3 and 7 for the *cover column* method and lines 13 and 15 for the *uncover column* method.
These two actions are the most important aspects of *algorithm DLX* as a whole, and serve to further illustrate the dancing links technique. Before we discuss the algorithm as a whole, let us examine these two methods in detail.

9.2 The Cover Column Method

During the algorithm, when we choose some row to add to the partial solution it will satisfy some number of constraints (i.e. the row has a number of nodes which lie in certain columns), and we must ensure that no more rows are chosen that also satisfy these constraints (as each can be satisfied *at most once*).

In order to complete this task, we cover any columns (constraints) that are satisfied in this manner. Should a column be covered, this means there are no rows currently accessible (i.e. non-covered rows) that can be chosen to satisfy the covered column.

9.2.1 Pseudocode

Let us examine Knuth's explanation of the method in [9].

Algorithm 4 Covering a Column c

Set $L[R[c]] \leftarrow L[c]$ and $R[L[c]] \leftarrow R[c]$.
for each $i = D[c], D[D[c]]$, ..., while $i \neq c$, **do**
    for each j $\leftarrow R[i], R[R[i]]$, ..., while $j \neq i$, **do**
        Set $U[D[j]] \leftarrow U[j], D[U[j]] \leftarrow D[j]$,
        and set $S[C[j]] \leftarrow S[[j]] - 1$.
    end for
end for

This method iterates through the rows that lie in the column being covered, altering the links above and below each node to remove them from their respective columns. You may recognise the operation of removing a node from section 7.

9.2.2 A Motivating Illustration

Here we will consider the same example four way linked list as seen in section 8, and we will be covering the C_3 column.

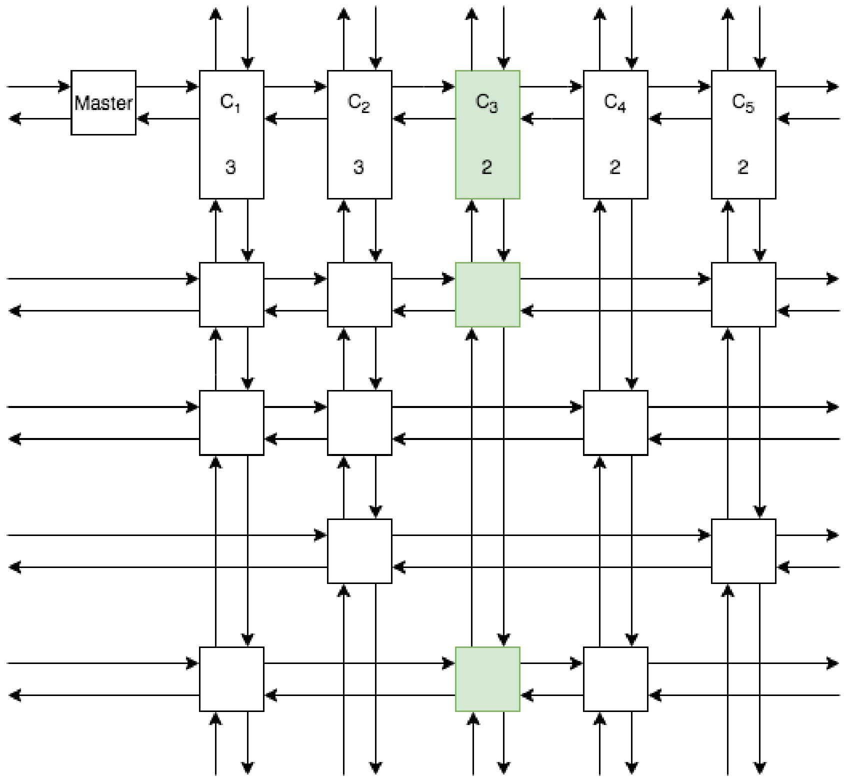


Figure 13: Illustration of a four way linked list, with column C_3 highlighted

In figure 13 we can see the *four way linked list* object in its original/default state, with the C_3 column has been highlighted (in green). Now let us begin the *cover column* method.

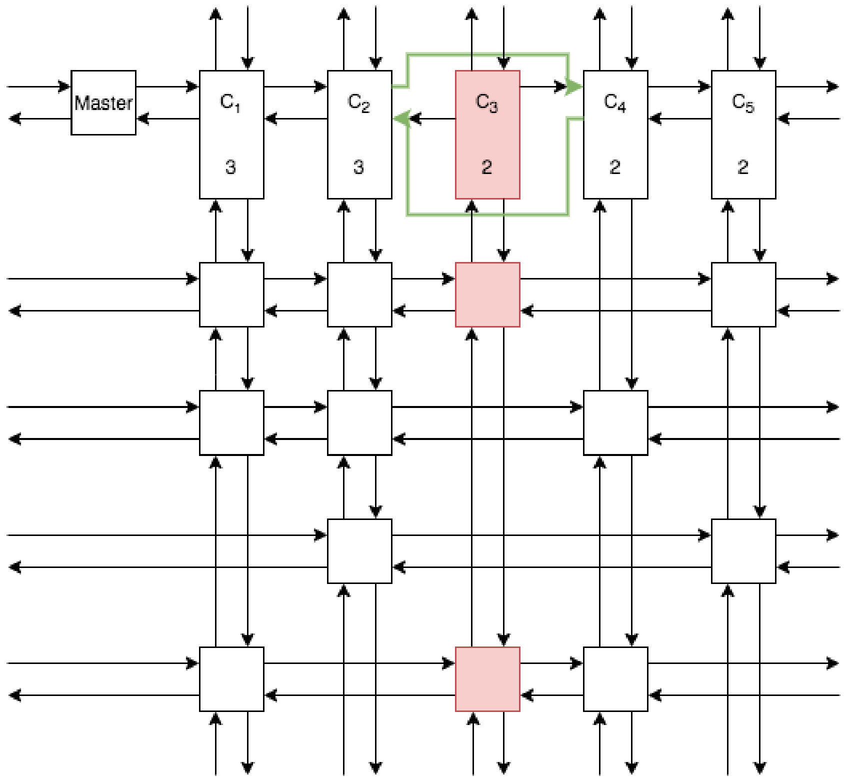


Figure 14: Cover column step 1: C_3 column header covered

In figure 14 we can see the effect of line 1 in the *cover column* method. Note that the changes to the structure have been highlighted in green.
This line uses *exactly* the operation of removing a node from a doubly linked list:

- R[L[C$_3$]] $\leftarrow$ R[C$_3$]
- L[R[C$_3$]] $\leftarrow$ L[C$_3$]

in order to *cover* the C_3 column header, removing it from the left to right list of column headers.

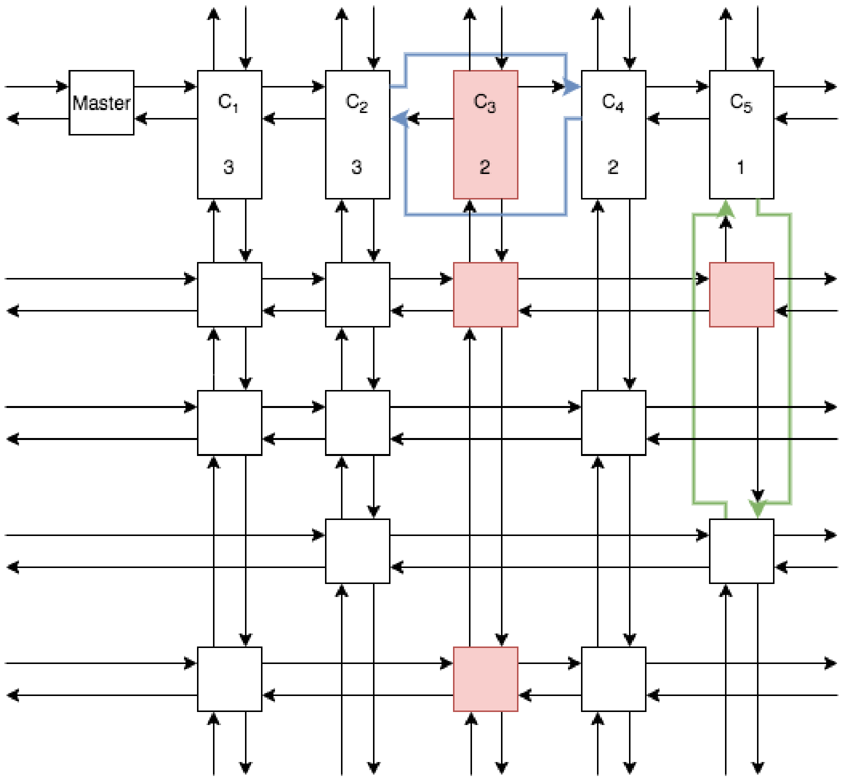


Figure 15: Cover column step 2: First node of first row covered

In figure 15 we follow the method by iterating down through the rows of the C_3 column once, and then traversing right across that row once.
The same action of removing a node is repeated here, except now we are *covering* a node from its neighbours above and below. In addition to this we alter the size value of the column header this node belongs to, in this case the C_5 column header has its size changed from 2 to 1.

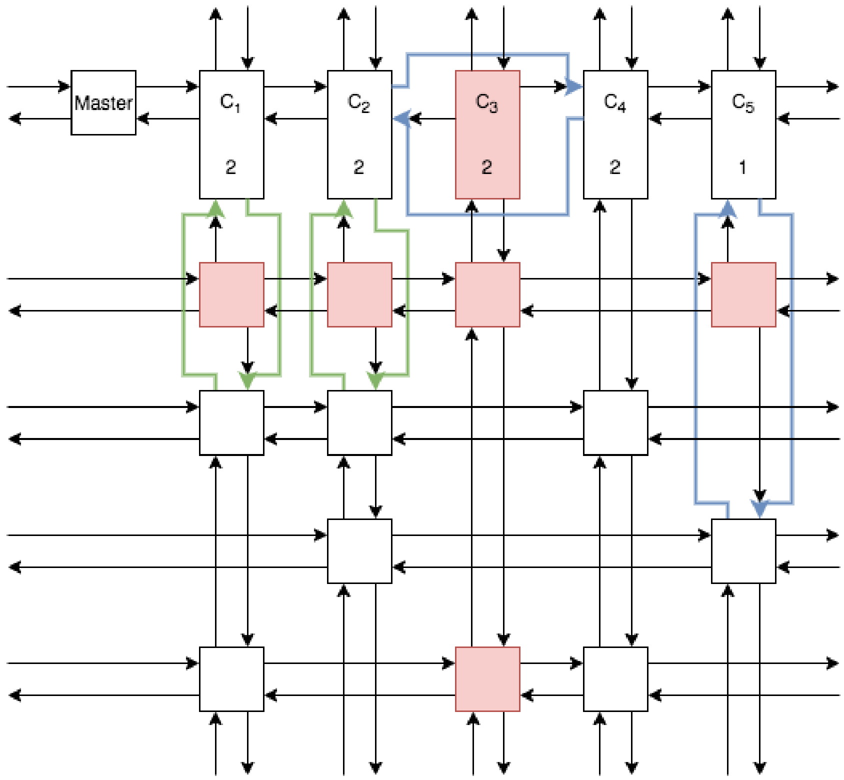


Figure 16: Cover column step 3: Remainder of first row covered

In figure 16 we continue traversing right across the row until we reach the node we began at, altering the links above and below each node we step across and adjusting the column header's size values accordingly.
Now that this row is fully covered, we return to iterating down through the column once more.

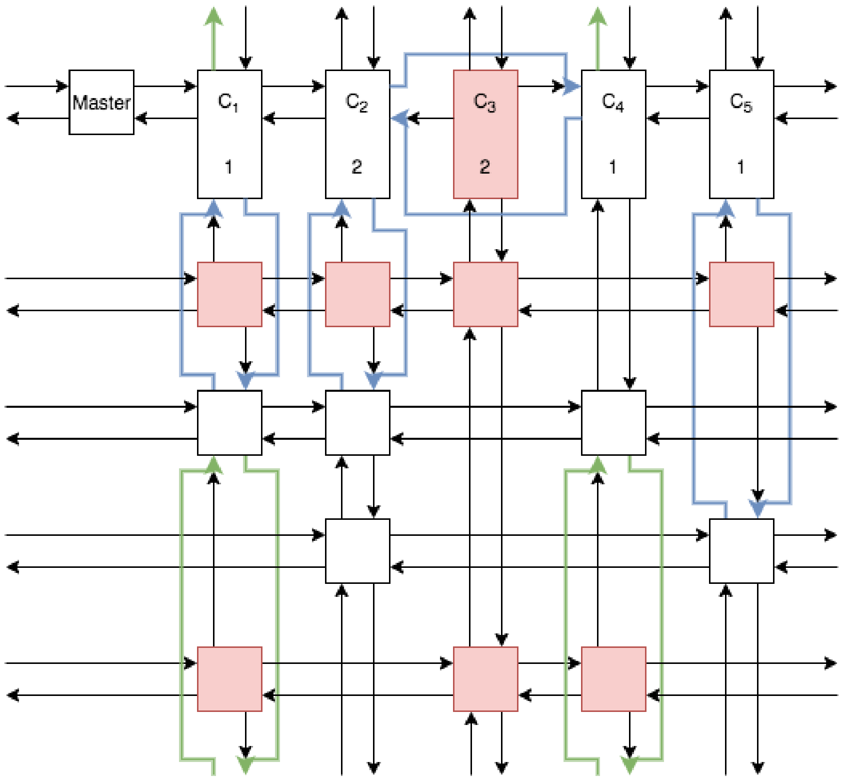


Figure 17: Cover column step 4: Second row covered

In figure 17 we have iterated down to the next (and final) row of the C_3 column, and traversed across it to the right in exactly the same manner as we have just seen for the first row.
We alter the links above and below each node, such that the are removed from their respective columns, while adjusting the size of each header accordingly.
Note how we do not alter the up and down links of the nodes inside the C_3 column itself, there is no need to as every other node is covered in these rows and we will need these links preserved in order to uncover the column later.

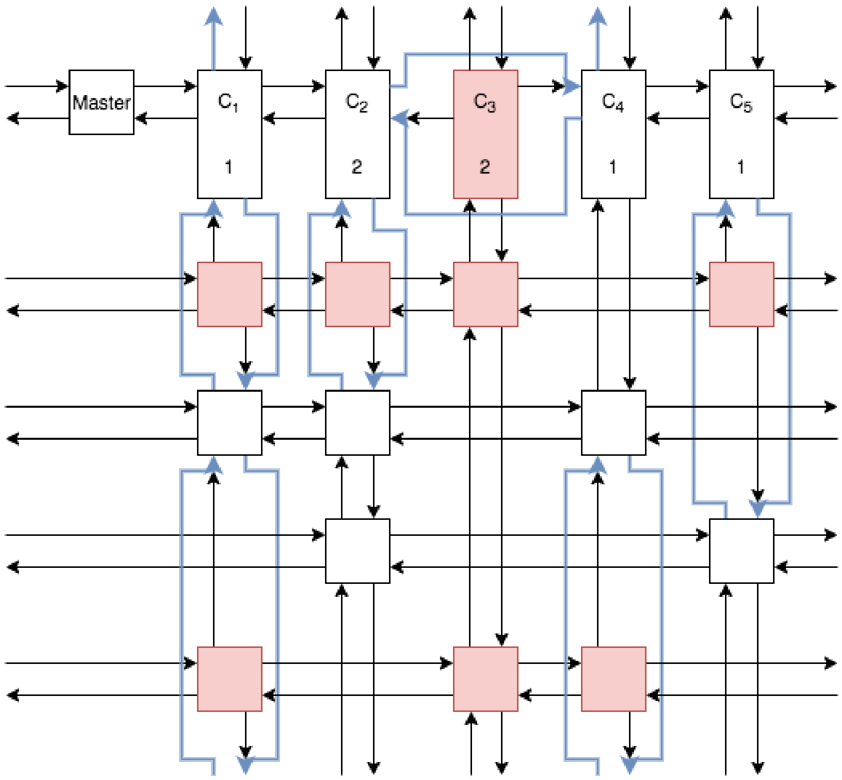


Figure 18: Four way linked list with column C_3 covered

Figure 18 shows the now fully covered C_3 column, note how all of the links that have been altered are highlighted in *blue*, and all the nodes/headers that have been covered are highlighted in *red*.

Now should we search the *four way linked list* for a new row to add to the partial solution, we are only able to choose rows that do not satisfy the C_3 column, exactly the effect we desired from covering the C_3 column in the first place.

9.2.3 Cover Column Implementation

The implementation of this method in Python closely resembles the pseudocode provided by Knuth which we discussed in earlier in section 9.2.1. We can see the exact code used in figure 19 where we define the *cover column* method of the *four way linked list* class. Here we simply pass the column header object to the method and follow the logic specified by Knuth.
We access the *left*, *right*, *up*, *down* and *column* fields of each node in Python using the *dot* operator, and use while loops (and iterators) to traverse down through the column and right through the

rows. These iterators are named in the following format throughout the entire program: *current (direction)*, so in this case we have *current down* and *current right*.

```python
def cover_column(self, column):
        column.left.right = column.right
        column.right.left = column.left
        current_down = column.down
        while current_down != column:
            current_right = current_down.right
            while current_right != current_down:
                current_right.down.up = current_right.up
                current_right.up.down = current_right.down
                current_right.column.size = current_right.column.size - 1
                current_right = current_right.right
            current_down = current_down.down
        return None
```

Figure 19: **Cover column** method definition in Python

9.3 The Uncover Column Method

As we have seen in the previous section, using the *cover column* method we can effectively remove a column and all of the rows residing in it from the *four way linked list*, however this is only half of the puzzle.

The main feature of this dancing links technique is the reversal of the *cover column* method, this is analogous to the simple example in section 7 where we removed a node from a doubly linked list and then used the dancing links technique to restore it. Now we would like to *cover* a column in the *four way linked list*, and then use the dancing links technique to restore it.

In practice, when we find that our partial solution is not a full solution, and none of the rows we can choose from will change the fact, then we need to *backtrack*. As discussed in section 5.3 we would like to do this as efficiently as possible, and now we are finally ready to showcase the efficient solution.
The *uncover column* method efficiently reverse the action of the *cover column* method, and thus backtracks without the need for any duplication of state information or creation of a new data structure.

9.3.1 Pseudocode

Let us examine Knuth's explanation of this method in [9].

Algorithm 5 Uncovering a Column c

1: **for** each $i = U[c], U[U[c]]$, ..., while $i \neq c$, **do**
2: **for** each j $\leftarrow L[i], L[L[i]]$, ..., while $j \neq i$, **do**
3: set $S[C[j]] \leftarrow S[[j]] + 1$,
4: and set $U[D[j]] \leftarrow j, D[U[j]] \leftarrow j$.
5: **end for**
6: **end for**
7: $L[R[c]] \leftarrow c$ and $R[L[c]] \leftarrow c$.

In section 9.2.1 we discussed the pseudocode for the *cover column* method, there we used the simple operation of removing a node from a doubly linked list as the main operation, as well as several loops to traverse all the necessary rows in the correct order. As we can see in algorithm 5 (lines 4 and 7) the main operation of this *uncover column* method is the dancing links operation (as seen in section 7).
In order for this method to be a perfect inverse we must uncover each node in exactly the opposite order that we covered them. In the *cover column* method we traversed *down* through the column and *right* through each row, now we iterate *up* through the column and *left* through each row.

We are claiming that this method is an exact inverse of the previous method we examined, the *cover column* method, and now we will examine another motivating illustration to hopefully convince the reader this is in fact true.

9.3.2 A Motivating Illustration

Let us now consider the example *four way linked list* that we used to illustrate the *cover column* method. We will soon see that the *uncover column* action completely restores the *list* to its original state.

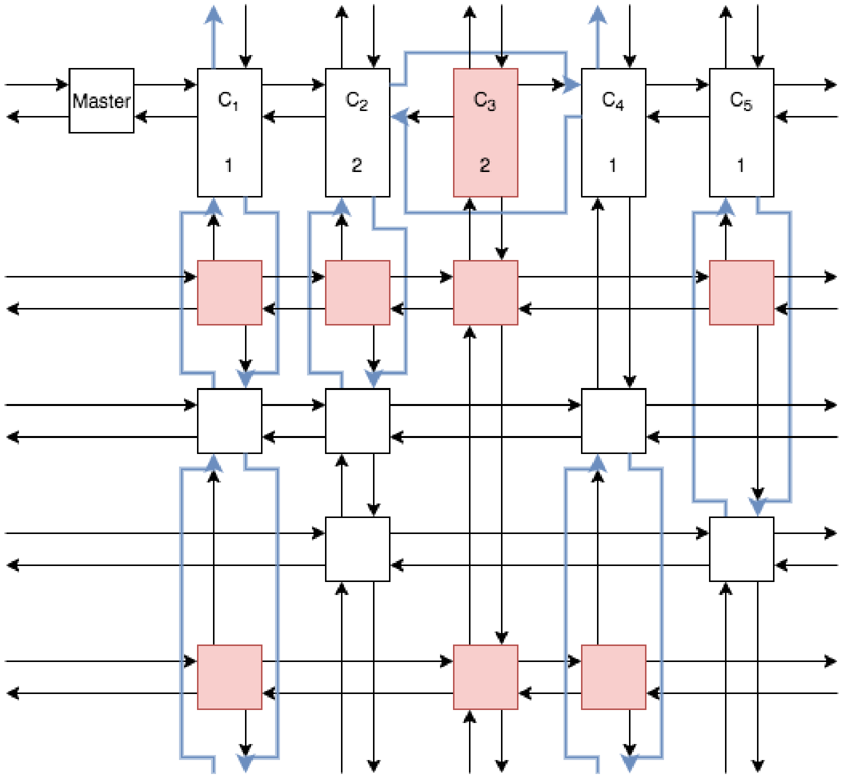


Figure 20: Four way linked list with column C_3 covered

Figure 20 shows the *four way linked list* prior to any *uncovering* action.

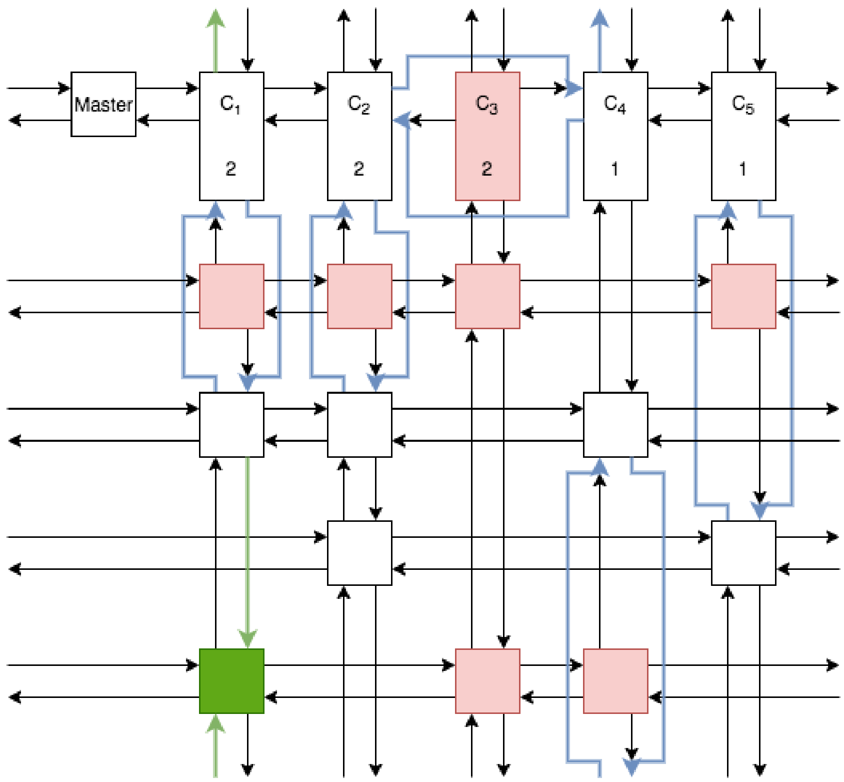


Figure 21: Uncover column step 1: Leftmost node of lowest column uncovered

Figure 21 illustrates the first iteration in the *uncover column* method, here we have traverse the C_3 column up one step, wrapping back around to the bottom, and traverse the row one step to the left.
The first node we uncover here is precisely the last node that was covered previously, and that is no coincidence.
The order in which the operations are performed is exactly the opposite of the *cover column* method, this ensures that the state of the four way linked list is recovered perfectly.

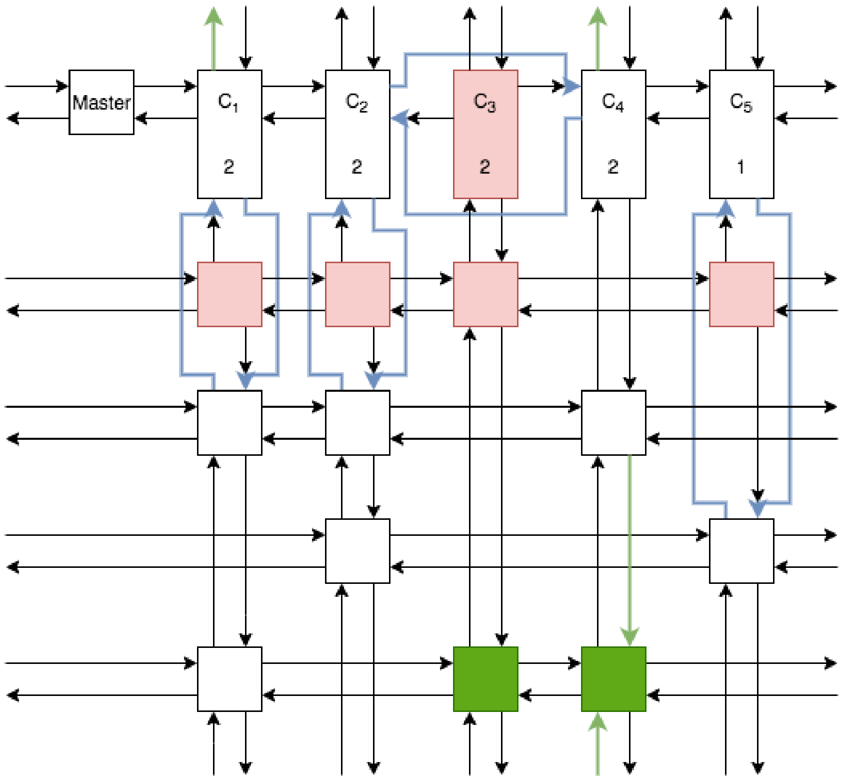


Figure 22: Uncover column step 2: Remaining nodes in row uncovered

In figure 22 we continue uncovering the nodes in this row, iterating to the left to wrap back around the figure, by restoring the links above and below each node using the dancing links operation:

- U[D[x]] ← x,
- D[U[x]] ← x.

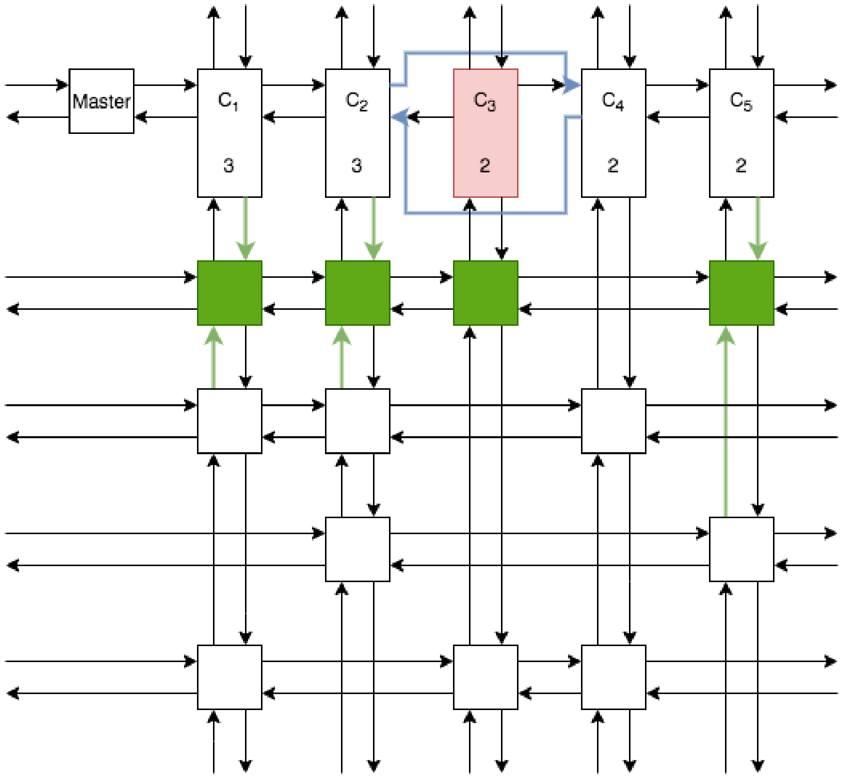


Figure 23: Uncover column step 3: Second row uncovered

Figure 23 illustrates the method as it traverses further up the column, and leftwards throughout the next row, restoring the links above and below each node as it goes. Note also that the method increases the size of the respective column headers each time a node is uncovered, this ensures the sizes always reflect the number of nodes in the column.

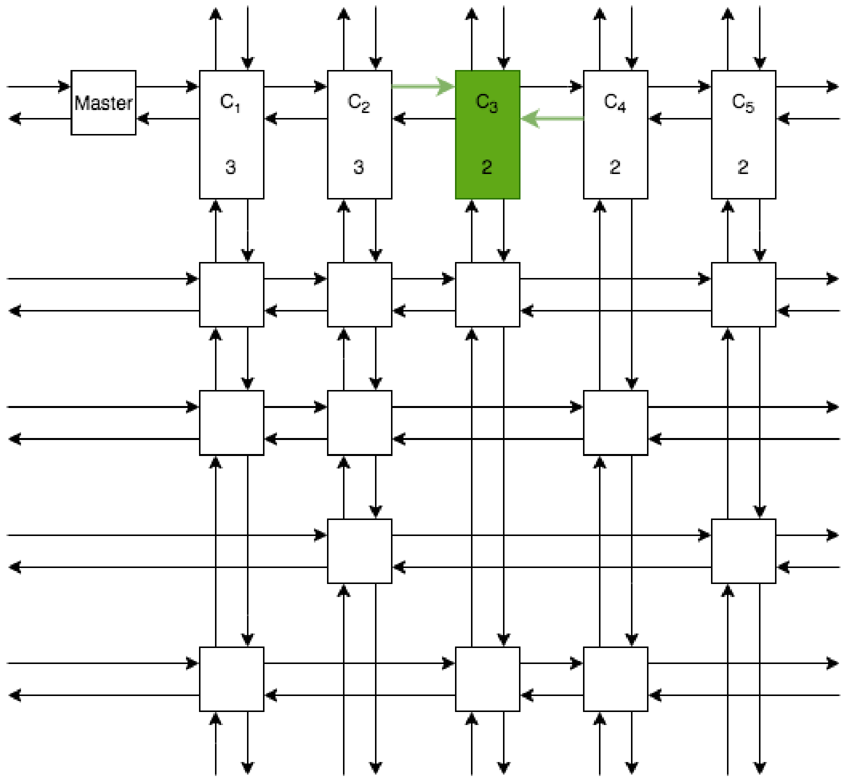


Figure 24: Four way linked list with column C_3 covered

Finally in figure 24 we exit the for loops and execute the final operation: restoring the column header to the *four way linked list*. This final action of the *uncover column* method is precisely the first action of the *cover column* method.

As we have now seen, the *cover column* and *uncover column* methods are inverses of each other, and we have completely undone the actions of the former using the latter.

The changing of the links which has happened in these diagrams, highlighted in blue and green, is the namesake for the algorithm: *Dancing Links*. This refers to the dance-like motion they undergo throughout the algorithm, as various columns are covered and uncovered in quick succession.

During algorithm DLX we cover columns as we add new rows to the partial solution, and then we uncover these columns after. The efficiency of these two methods together is the essential aspect of algorithm DLX, and now that we have discussed it in detail we can continue examine the algorithm as a whole, to see how partial solutions are stored (and outputted should they be correct) as well as determining which columns to cover and which rows to add to the partial solution. First however let us briefly discuss the implementation of this method in python.

9.3.3 Uncover Column Implementation

Similar to the *cover column* method definition, this definition closely resembles the pseudocode provided by Knuth in [9].
We can see how the two methods are very similar in nature, the only differences being the direction of travel (through the *four way linked list*), the order of operations and the links themselves.

Uncover column is a method of the *four way linked list* class, and takes the same input values as the *cover column* method, namely the column header of the column to be uncovered.
We can see the definition itself in figure 25.

```python
def uncover_column(self, column):
        current_up = column.up
        while current_up != column:
            current_left = current_up.left
            while current_left != current_up:
                current_left.column.size = current_left.column.size + 1
                current_left.down.up = current_left  # Restore link below
                current_left.up.down = current_left  # Restore link above
                current_left = current_left.left  # Step left
            current_up = current_up.up  # Step up
        # Add column to the header chain
        column.left.right = column
        column.right.left = column
        return None
```

Figure 25: **Uncover column** method definition in Python

9.4 Algorithm DLX Implementation

Now that the framework is in place to create instances of the *four way linked list* class, as well as convert a matrix into a *four way linked list* and update it by covering and uncovering columns, we can move on to the full implementation of algorithm DLX.
In this section we will discuss the details of our implementation and highlight any changes when compared to the pseudocode provided by Knuth in [9].
Our implementation can be seen in figure 26, and throughout the following sections we will explain the details of our program and any major design decisions we made while implementing it.

```python
def dlx(self, k, log=True):
        if not self.master_node.right.primary:
            self.file_write_solution(True)
            if log:
                self.file_write_solution(False)
            return None
        else:
            if self.dead_constraint(log):
                return None
            current_column = self.find_best_column()
            current_node = current_column.down
            self.cover_column(current_column)
            while current_node != current_column:
                self.set_solution_k(current_node, k)
                if log:
                    self.file_write_log_row(current_node, k, backtrack=False)
                current_right = current_node.right
                while current_right != current_node:
                    self.cover_column(current_right.column)
                    current_right = current_right.right
                self.dlx(k+1, log)
                current_node = self.solution_list[k]
                current_column = current_node.column
                current_left = current_node.left
                while current_left != current_node:
                    self.uncover_column(current_left.column)
                    current_left = current_left.left
                current_node = current_node.down
            self.uncover_column(current_column)
        return None
```

Figure 26: **Algorithm DLX** definition in Python

9.4.1 Determining Full Solutions

In both Knuth's pseudocode (algorithm 3) and our implementation in Python (figure 26) the first line of the method checks to see if a full solution has been found.

This begs the question: *How do we store solutions in the first place*? Partial solutions are stored in current memory and updated during the program, however we do not need to examine the current partial solution stored in memory to determine if it is a full solution, we can simply check the columns of the *four way linked list.*

In Knuth's guide, he specifies this check should be: 'If $R[master] = master$', in other words if

the next uncovered column to the right of the master header itself, then all columns are covered. This of course means all of the constraints are currently satisfied by the choice of rows, as columns are analogous to the constraints of the exact cover problem (see section 8.3 for more details).
Our implementation of this check differs slightly, as we chose to focus our program on the N Queens problem and thus need to account for secondary constraints (recall, a secondary constraint can be satisfied *at most once*, and can remain unsatisfied for a full solution), we did this by simply checking if the column to the right of the master header has a primary value of *true*: 'If not self.master_node.right.primary'.
This works because of the particular way in which we order the constraints when formulating the N Queens problem, all of the primary constraints first followed by the secondary ones moving left to right. This also works in the case of a regular exact cover problem where all columns are primary because we initialise the primary attribute of the master header to be false. If the master header is pointing to itself on the right, then the header to the right (itself) has a primary value of *false* and thus the condition is met successfully.

In the case that a full solution is found we call the *file write solution* method and return, this method is discussed in section 9.4.6 however all we need to know now is that the check we implemented is functioning correctly and control only passes further on into the algorithm (i.e. we don't return) if the current partial solution is not a full one.

9.4.2 Determining if Backtracking is Necessary

After we have checked for a full solution, if none is found we proceed to check if there are any *dead constraints*, in other words if it is still possible to find any full solutions by continuing forward in the current state.

Here we define a *dead constraint* to be a column with: primary = *true* and size = 0, if we are faced with such a situation algorithm DLX will choose this *dead* column as it has the smallest size available. If a *dead constraint* is present this means we have a currently unsatisfied column with no possible rows to satisfy it, thus no choice of column and row will result in a full solution.

In the case that a *dead constraint* is present Knuth outlines that the algorithm should iterate over the available rows (which is 0 iterations as the column has no uncovered rows) and then return control back to the last recursive call.
Note here that Knuth does not mention this explicitly in the pseudocode (algorithm 3) as the method itself should perform correctly without the need for any *dead constraint* check.
This approach however did not suffice for us, as we had several unusual errors occurring in such a situation and for sake of clarity and ease (with the notable sacrifice of speed) we implemented a simple helper method to check if any columns are *dead*. This can be seen in figure 27, where we simply iterate across the header list, traversing to the right, checking to see if any columns meet the aforementioned condition and if so, we log this and return *True*. If at least one column is *dead* then there is no need check any others, so there is no need to continue checking after one has been found. If no such columns are found we can return *False* and continue with algorithm DLX as normal. The *log* argument here is explained in section 9.4.6 as it concerns the file management

associated with our implementation.

```
def dead_constraint(self, log):
        current_header = self.master_node.right
        while current_header != self.master_node:
            if current_header.primary and current_header.size <= 0:
                if log:
                    self.file_write_log_row(current_header, backtrack=True)
                return True
            current_header = current_header.right
        return False
```

Figure 27: **Dead constraint** method definition in Python

9.4.3 Choosing Columns

Knuth simply states that we should choose a column to cover in the pseudocode (provided in algorithm 3), however he further elaborates in [9, p.6] that we should choose from the available columns the one with the smallest size.
In fact this is the main reason we would like to track the size field of each column header during the program, as choosing column headers with smaller sizes greatly reduces the number of nodes in the search tree of the algorithm and therefore the number of operations required.

We chose to include this optional size attribute as it provides a relatively simple optimisation, and we would like to track the size of each column to determine if any are *dead* regardless.
We have discussed how the size field was implemented in section 8.3.1 and how it is maintained during algorithm DLX in sections 9.2 and 9.3.1. Searching for the column with the smallest size is as simple as iterating through the list of column headers and storing the current best choice, however we must also check that the column has a primary value of *true* as we must allow for the N Queens problem specific requirements. The implementation of this can be seen in figure 28.

```python
def find_best_column(self):
        current_header = self.master_node.right
        best_header = current_header
        while current_header != self.master_node:
            if current_header.size < best_header.size and current_header.primary:
                best_header = current_header
            current_header = current_header.right
        return best_header
```

Figure 28: **Find best column** method definition

9.4.4 Recursive Calls and Storing Partial Solutions

Now that we have chosen the best available column, let us proceed through Knuth's pseudocode (algorithm 3, line 3). We will try to find full solutions using each row in this column one by one, however our first step is to cover the chosen column (as we know each choice of row will involve this column, we can cover it now).
After covering the column we then iterate down through its rows, during each iteration we add that row to the partial solution list at depth k. We then iterate across the row, for each node we encounter we cover that node's column as we have now satisfied that constraint/column.
After all necessary columns are covered we make a recursive call to algorithm DLX, with the depth variable increased by 1.
We will now discuss the depth variable k, it is initially set to 0 and each time we make a recursive call we increase its value by 1. This depth variable is used to index the partial solutions stored in the solution list (i.e. during the very first call of algorithm DLX, each row from the chosen column will be stored in the 0^{th} index of the partial solution list).

After storing the chosen row in the partial solution list at the k^{th} index, we then iterate through that row and cover each of the columns that the row has a node in . We must cover these columns as we have now satisfied each of them by choosing this row and storing it as part of the partial solution.
Following that covering we then make a recursive call to algorithm DLX, with the depth variable k increased by 1.

In our implementation we use a native python list for the *solution list* variable, which is an attribute of the *four way linked list* class, and is initialised to be empty. The solution list stores the rows that are currently in the partial solution, and will be used to output those rows if they turn out to be a full solution. However, there is no need to store an entire row in this list, instead we store a single node and during the outputting of solutions we also output that node's neighbours to the left and right (in fact we iterate around the row to either the left or right and output each node).
As algorithm DLX is a method of the *four way linked list* class, we can simply access the solution

list in the method without needing to pass it in as a variable.
The depth variable k however is passed in for each iteration and increased by 1 for each recursive call which we can see in figure 26.

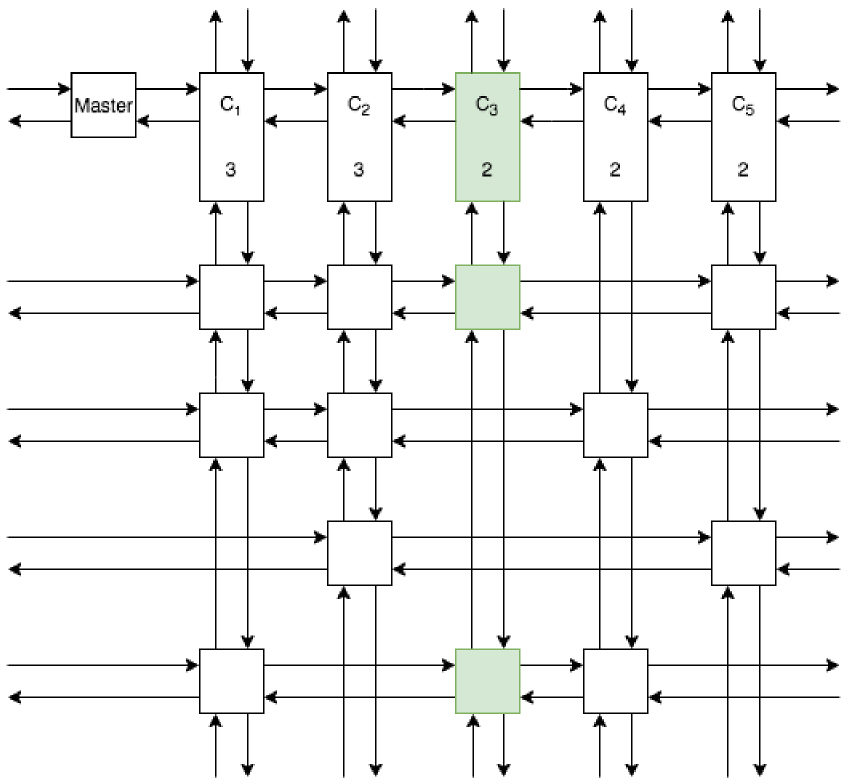


Figure 29: Illustration of a four way linked list, with column C_3 highlighted

Consider the *four way linked list* from previous sections, seen again here in figure 29. In this example we would like to store the first node of the C_3 column as a partial solution (ignoring the fact we need to cover the C_3 column first), it will be in the first index: $k = 0$. As we progress forward new nodes will be added to solution list at an index corresponding to the k variable of the DLX function call, until a full solution is reached and we can output the entire solution list. If we need to backtrack, for example we are at $k = 1$ and find that no solution is possible, (i.e. a *dead column* is found) we therefore backtrack to $k = 0$ and simply overwrite the previous node stored in the 0^{th} index of the solution list.

The native list structure in Python returns an error if we assign a variable to an index which

is out of range, for example if we have a list $x = [8, 16]$ which has the indices $x[0] = 8$ and $x[1] = 16$, we cannot try to set: $x[2] = 64$, as this index is currently out of range of the list. Instead we must use the append method to add a new value and expand the length of the list: $x.append(64)$. For this reason we use a try and except statement in figure 30. We first try to set the k^{th} index equal to the node we would like to store and if this returns an error we instead append it on to the end of the list.
This will always work out to be the desired k^{th} index, as our list begins with a length of zero, and the k variable increases by exactly one for each recursive call. As we will always be storing at least one node for each k value (we never choose columns with size $= 0$ due to the *dead constraint* method), we are never more than one index out of range.

```python
def set_solution_k(self, new_node, k):
        try:
            self.solution_list[k] = new_node
        except:
            self.solution_list.append(new_node)
        return None
```

Figure 30: **Set solution k** method definition in Python

Storing solutions in this manner is an interesting feature of Knuth's algorithm DLX, as we need only specify a single list for partial solutions throughout the program. Each time the algorithm embarks upon a new branch, backtracking from a previous one, it will overwrite the nodes stored in the solution list, and when the time comes to print a full solution we can be confident that all of the correct nodes are stored in the current version of the list.

9.4.5 Uncovering the Chosen Column

Once the recursive call is made, the program will fully evaluate that call before bringing control back to our first iteration of algorithm DLX. This depth first nature of the algorithm means that for a chosen column with several rows, all possible solutions containing the first row (at that depth, and with respect to the current partial solution of course) will be checked before control passes back and we can check for the second row.
Once we reach that second row, we will add it to the partial solution at the current depth and proceed as normal by covering each column that the row satisfies. However we must first uncover all the columns associated with the previously chosen row from the last iteration, as that row is no

longer part of the partial solution (here we have tried some row at depth k, and following that are trying a new row at the same depth k, thus replacing the first) we must undo the action of choosing it.

In essence here we have *backtracked* after choosing the first row, and need to undo the actions of covering each of that row's columns. Thankfully we have just the method for that, and in lines 10-15 of algorithm 3 we use the *uncover column* method to undo the previous actions. We retrieve the chosen row from the solution list, which we know was stored at the k^{th} index corresponding the to the k value for this iteration, and iterate leftwards across the chosen row uncovering each of the node's columns.
Note here that we do not uncover the originally chosen column as we terminate the loop before making a full circle, this is intentional as we do not want to uncover that column until we have attempted a solution for all rows in it.

Our implementation follows this guideline closely, as we can see in figure 26. We retrieve the originally chosen row from the solution list, and using that we retrieve the column header itself.
We proceed to iterate leftward across the row, uncovering each node's column as we go. Following that we continue iterating down through each row in the originally chosen column, however once we have exhausted the rows in that column, we exit the loop and finally uncover the originally chosen column, then we return.
This action will happen every time the algorithm is called, ensuring that we return the exact same *four way linked list* that we were passed.

9.4.6 Bookkeeping

In this section we will explain the methods used to create and update the output files throughout the execution of the program, including the outputting of full solutions.

There are two distinct output files used in the program: the *main* file and the *log* file. The former of these is intended for humans and contains all of the solutions found by algorithm DLX, these are labelled and a short introduction is also written. The latter of these, the *log* file, is only intended for use with the N Queens visuals (discussed in section 10), and contains a log of every action performed during the program. This includes every row that is added to the partial solution list and every time a *dead constraint* is found resulting in a backtrack.
It should be noted here that the *log* file is entirely optional, and the user can specify to not have it maintained throughout the program during start-up.

The *main* file is typically named *main_output.txt*, however if the user is solving the N Queens problem it is instead named *N_queen_output.txt* with the specified value of N. The *log* file follows a similar rule, it is named *log.txt* by default but will be named *N_queen_log.txt* should the user be solving the N Queens problem.
The filenames of both the *main* and *log* files are stored as attributes of the *four way linked list* class, this allows for methods of the class to open either file without the need for the filename to be passed as an argument.

There are a number of methods for specific file management, all of which I will briefly discuss here, but first we will consider the action of outputting a full solution should one be found.

The *file write solution* method outputs the current solution list to either of the files (specified by a Boolean argument) as well as updating the total solutions counter, and it's defintion can be seen in figure 31.

```python
def file_write_solution(self, main_file=True):
        # Only update the counter for the main output file
        if main_file:
            self.total_solutions = self.total_solutions + 1
            file = open(self.main_file, "a")
        else:
            file = open(self.log_file, "a")
        if self.total_solutions == 1:
            file.write("\n\nSolutions:\n\n")
        file.write("Solution {0}\n".format(self.total_solutions))
        for i in range(len(self.solution_list)):
            furthest_left = self.find_furthest_left(self.solution_list[i])
            file.write(furthest_left.column.name)
            file.write(", ")
            file.write(furthest_left.right.column.name)
            file.write("\n")
        file.write("\n")
        file.close()
        return None
```

Figure 31: **File write solution** method definition in Python

During the writing of solutions we only output two nodes from each row, the furthest left node and its neighbour to the right. In order to find the furthest left node in a given row we use the *find furthest left* method which can be seen in figure 32. This method simply iterates around the row in question, and compares the index of each node's column header. The comparison of these indices is done using the *find original index by name* method, which uses the *header list* attribute of the *four way linked list* to evaluate the header's original indices.

```python
def find_furthest_left(self, current_node):
        dummy_node, best_node = current_node.left, current_node
        best_index = self.find_original_index_by_name(current_node.column.name)
        while dummy_node != current_node:
            dummy_index = self.find_original_index_by_name(dummy_node.column.name)
            if dummy_index < best_index:
                best_node, best_index = dummy_node, dummy_index
            dummy_node = dummy_node.left
        return best_node
```

Figure 32: **Find furthest left** method definition in Python

If efficiency and speed were greater concerns during this implementation we could remove these aesthetic file writing methods, and simply output the column header names associated with a row in any order and then arrange them in another program.

Once a *four way linked list* converts a *0-1 matrix* the initial file functions are called. The main file is initialised by the *main file initial* method which can be seen in figure 33.

```python
def main_file_initial(self):
        # Use the write method here to overwrite any existing file contents
        solution_file = open(self.main_file, "w")
        solution_file.write("Algorithm DLX\n\n")
        solution_file.write("This algorithm finds all solutions to an exact cover problem.\n")
        solution_file.write("An implementation of Donald Knuth's algorithm X, using dancing links, is used.\n")
        solution_file.write("For more information visit: https://github.com/Hedge-Hodge/Dancing_Links_N_Queens\n")
        solution_file.write("This file contains these solutions.\n")
        solution_file.close()
        return None
```

Figure 33: **Main file initial** method definition in Python

When the program is started the user is prompted: *Would you like an extensive log of all steps taken*? If they respond affirmatively the log file will be initialised by the *log file initial* method, seen in figure 34.

```python
def log_file_initial(self):
        log_file = open(self.log_file, "w")
        log_file.write("DLX Log\n\n")
        log_file.write("See '{0}' for the proper algorithm output and aesthetic solution list.\n".format(self.main_file))
        log_file.write("This file contains a detailed record of each iteration of the recursive DLX algorithm.\n")
        log_file.write("Each time a new row is chosen, this will be logged here.\n")
        log_file.write("Each time the algorithm finds itself in a 'dead end' (i.e. when some of the remaining "
                       "constraints have size=0) this will be recorded also.\n\n\n")
        log_file.write("Begin log:\n")
        log_file.close()
        return None
```

Figure 34: **Log file initial** method definition in Python

In both of the initial file methods we use the corresponding filename attribute of the *four way linked list* to open the file in *write mode*, which will overwrite any previous contents stored there. We also write a short introduction for the user in each, then close the file and return.

We decided it would be helpful to the user if the *exact cover problem* was represented in some way in the main output file and as such we print the *0-1 matrix* to the file output file, using the *file write one zero* method (figure 35).
Here we use the *savetxt* function from NumPy [5] to write the *0-1 matrix* to the file, the $fmt = \%i$ argument specifies that the matrix should be printed as integer values as opposed to floats.

```python
def file_write_one_zero(self, matrix):
        solution_file = open(self.main_file, "a")  # Open the file in append mode
        solution_file.write("\nThe following matrix represents the exact cover problem in question:\n")
        np.savetxt(solution_file, matrix, delimiter=' ', fmt='%i')  # fmt argument writes integers instead of float
        solution_file.close()
        return None
```

Figure 35: **File write one zero** method definition in Python

Should the user be solving the N Queens problem a short piece is written to the main output file by the *file write n queen* method, which can be seen in figure 36. This method simply opens the file in append mode and outputs some predetermined text, as well as the specified N value, before closing the file.

```python
def file_write_n_queen(self, N):
        solution_file = open(self.main_file, "a")  # Open the file in append mode
        solution_file.write("Here we have the classic N-Queens exact cover problem, with:\n")
        solution_file.write("N = {0}\n".format(N))
        solution_file.write("The columns of the matrix above correspond to the row, column and diagonal constraints.\n")
        solution_file.write("While the rows correspond to possible queen placements.\n")
        solution_file.close()
        return None
```

Figure 36: **File write n queen** method definition in Python

The final file method to discuss is the *file write log row* method, which is responsible for updating the log file each time a new row is added to the partial solution, or the algorithm finds a dead constraint and consequently backtracks. We can see its definition in figure 37, here we specify which kind of operation will take place (either a new row added or a backtrack needed) using the Boolean argument *backtrack*. Should a backtrack be required we will pass the dead constraint as the node argument and output it accordingly, if however we are logging a new row that has been added to the partial solution we instead pass a new node from that row in the node argument and output all of its neighbours (using the find furthest left method in figure 32) to the left and right as well as the current depth k.

```python
def file_write_log_row(self, node, k=0, backtrack=False):
        log_file = open(self.log_file, "a")
        if backtrack:
            log_file.write("BACKTRACK necessary,\t")
            log_file.write(node.name)
            log_file.write("\tis a dead constraint.\n")
        else:
            current_node = self.find_furthest_left(node)
            log_file.write("k={0}\n".format(k))
            log_file.write(current_node.column.name)
            dummy_node = current_node.right
            while dummy_node != current_node:
                log_file.write("\t")
                log_file.write(dummy_node.column.name)
                dummy_node = dummy_node.right
            log_file.write("\n")
        log_file.close()
        return None
```

Figure 37: **File write log row** method definition in Python

These methods all come together to create the output files for the user, while they are not the most efficient methods they do clarify exactly what is happening in the program which is their purpose.
If efficiency were a greater concern we would approach these bookkeeping methods with heightened minimalism.

10 N Queens Visuals, Variants and Further Research

In this section we will look a visual representation of the N Queens solver in action, review a variant of the N Queens puzzle that requires a chessboard shaped as a torus and finally explore some additional topics that could be researched further.

10.1 Visuals

Making a visualization of the N Queens problem was an interesting problem that we wanted to tackle once the main solver was up and running. We researched plots and visual tools implemented into the python language that would produce a figure similar to an NXN board. We found what is called a "pcolor" from the matplotlib python in [6] that is usually used as a heat-map.

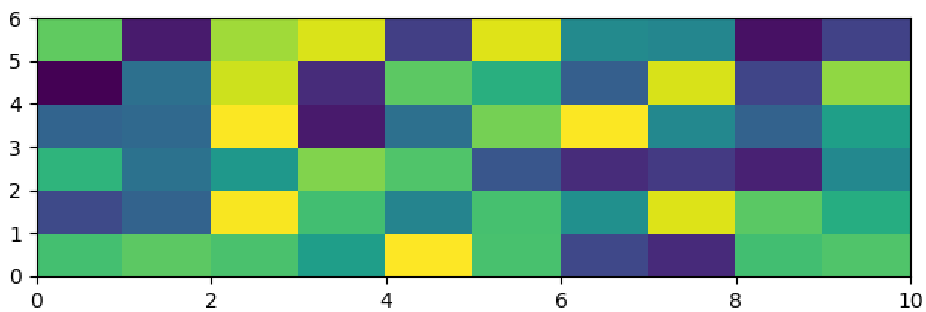


Figure 38: Demonstration of pcolor figure from official documentation in [6]

The plot works by converting an N X N matrix of zeros and twos into an N X N grid, as from the below matrix and figure 39. We see that blank spaces are denoted as 0 in the matrix which corresponds to white in the grid, while Queen placements correspond to 2 in the matrix and Green in the grid. We decided to use 0's and 2's for this plotting matrix instead of 0's and 1's to avoid confusion with the zero-one matrix. By logging the rank and file of moves taken by the algorithm, we succeeded in making a simple N X N board that would show the algorithm's progress in action.

$$\begin{pmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$

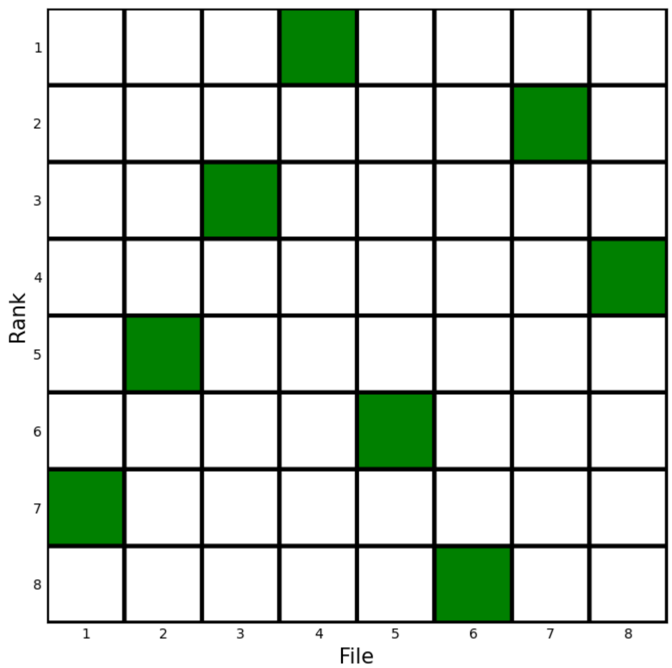


Figure 39: pcolor plot of an 8 X 8 solution

A demonstration of the visuals can be seen by clicking the above image, which links to an MP4 video hosted on github. We decided to keep this implementation separate from the main implementation as it slows the implementation down significantly.

10.2 Toroidal Chessboard for N Queens

One last aspect of the N Queens problem that we wanted to cover was a variant that was posed to us by our supervisor. This variation requires a circularly linked chessboard where the board's surface is shaped like a discrete torus. We can prove that there **could** be solutions to this problem for $N \geq 2$, and give some justification of a computational method to find those solutions using the argument below.

For the N Queens puzzle where $N \geq 2$, we have $2N-1$ diagonals on the board. This includes the trivial diagonals which need to be considered for this problem, and aren't necessary for the classical N Queens problem as in subsection 4.3. For the standard non-circular chessboard, N diagonals are opposed and $N-1$ diagonals are unopposed. An example of this is presented in figure 40, where we denote an example of a diagonal as a blue line and an example of a reverse diagonal as a red line.

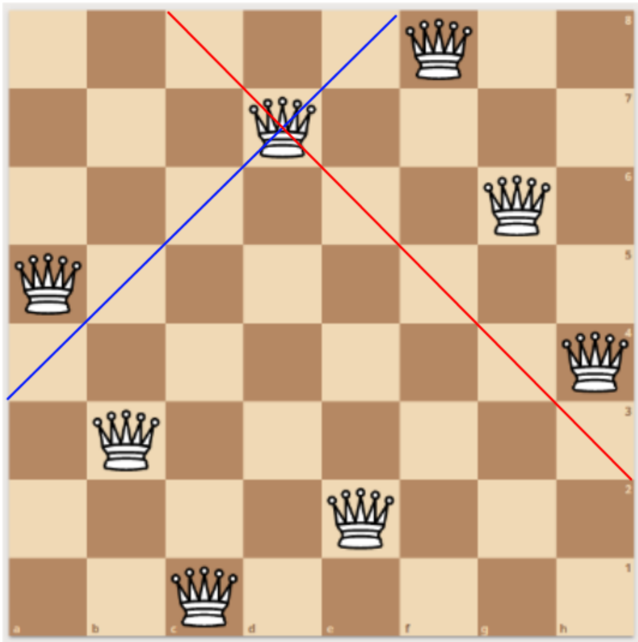


Figure 40: Standard 8 by 8 chessboard with a diagonal and a reverse diagonal marked

Figure 40 above illustrates that a standard 8 by 8 board has $2(8) - 1 = 15$ diagonals where 8 are opposed and $8 - 1 = 7$ are unopposed.

We will use figure 41 below as our representation of the toroidal shape. Here, the dark chessboards are permutations of our original 8 by 8 board, and represent the circular nature of the torus which allows us to illustrate all opposed diagonals. We see that the diagonal marked in blue opposes both diagonal 5 and diagonal 13. In fact, **every diagonal opposes two diagonals on the toroidal board, except for the main diagonal which opposes only one.**

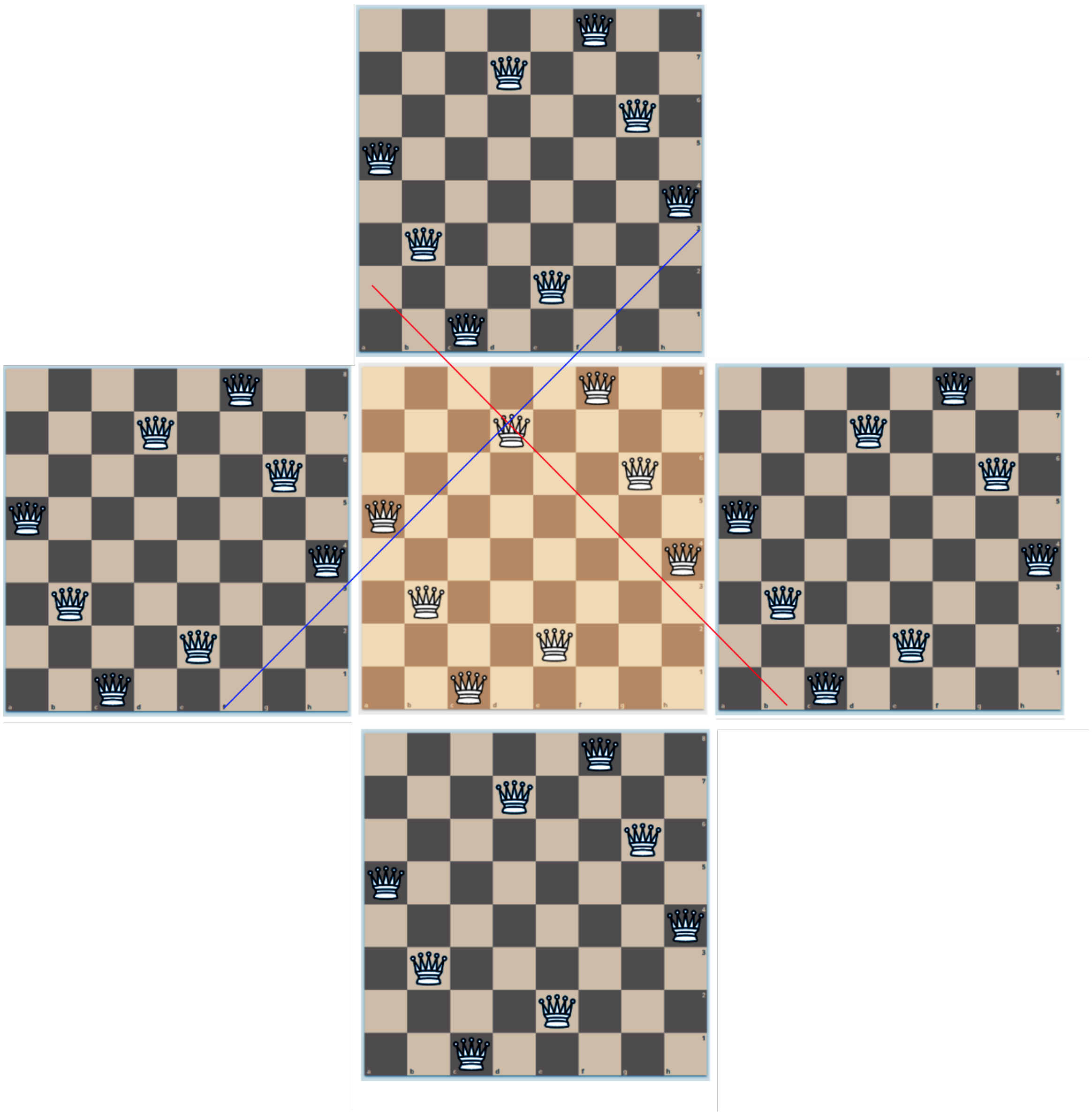


Figure 41: 8 by 8 Torus Queens problem with permuted board

We see that each diagonal is linked to another diagonal, and we can pair off every diagonal except the main diagonal, as seen in the figure 42.

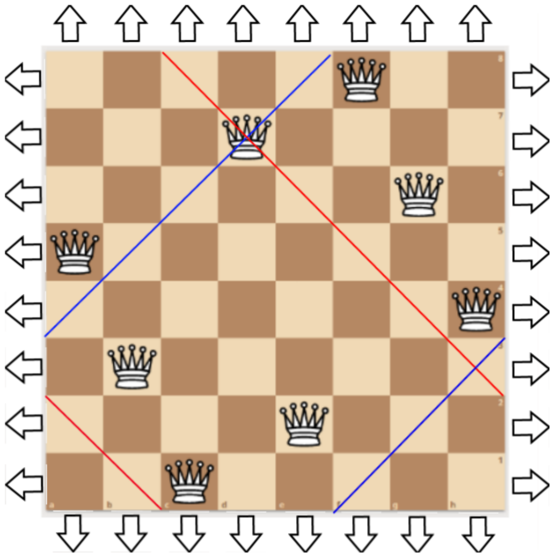


Figure 42: Coupled columns become opposed due to the toroidal shape of the board

Thus this problem may have solutions for $N \geq 2$, where we would require all $2N - 1$ diagonals to be opposed by exactly one Queen. Given more time to work on the project, we could have implemented another version of our program specifically for this variant, where the diagonal constraint was changed from secondary to primary, and the diagonals that oppose each other on the torus would be grouped together as one diagonal.

Through researching this variant of the problem we found that Hungarian mathematician Pólya, G. has a paper [11, p.237–247] in which he asserts that the toroidal variant has solutions in all N that is indivisible by both 2 and 3.

10.3 Further Research

Given more time we would like to put the knowledge and experience we have gained into use by implementing algorithm DLX in C++, providing a faster option for any practical use.

The N Queens problem has a set of fundamental solutions for each given N that can be used to generate the full set using symmetry operations. Further investigation on these fundamental solutions might give us superior solving times for all N.

Applying dancing links to other exact cover problems, such as Sudoku and 3-D pentimino tiling in Knuth' [10, 7.2.2.1], would be another great exercise. As it stands, our implementation can *solve* these other problems only if the user formulates the appropriate 0-1 matrix themselves and passes it using a correctly formatted file. Given more time we would set up the the automatic formulation of these other problems, similar to how we provide the composition of the N Queen's 0-1 matrix given an input N.

11 Collaboration Outline

This project was undertaken by two students, Seán Monahan and Oisín Hodgins. We collaborated on all aspects of the project, where Seán was primarily responsible for sections 4, 5, 7, 10, and Oisín was primarily responsible for sections 6, 8, 9.

12 Conclusion

In conclusion we found the implementation of algorithm DLX to be an educational and insightful experience and now are better prepared to implement or analyse other algorithms in the future.

There is a narrative here that can be easily overshadowed by the technical details associated with the implementation, the main characters of this are Dijkstra, Hitotumatu, Knuth and Noshita.

Hitotumatu and Noshita published their optimisation of Dijkstra's program for the N Queens problem in 1979 [8], a short two page description of what would come to be called the *dancing links* technique.
The operation itself was quite the discovery, and relatively simple too, but this paper went *under the radar* so to speak for quite some time before Knuth popularised it in [9].

The lesson to be learned here is that sometimes the most efficient and elegant solution is so simple, it can be easily overlooked.
It begs the question as to what other revolutionary developments sit published now, but will go completely unrealised until they are popularised many years in the future.

References

[1] diagrams.net graphing software. documentation available here.

[2] Exact cover (wikipedia). available here.

[3] lichess (chessboard editor). available here.

[4] Np-completeness (wikipedia). available here.

[5] Numpy python library. documentation available here.

[6] pcolor from matplotlib. documentation available here.

[7] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., GBR, 1972.

[8] Kohei Noshita Hirosi Hitotumatu. *A Technique for Implementing Backtrack Algorithms and its Application*, 1979.

[9] Donald Knuth. *Dancing Links*, 2000. available here.

[10] Donald Knuth. *The Art of Computer Programming, VOL 4B: Fascicle 5*. Addison-Wesley Professional, 2019.

[11] G. Polya. *Collected papers Vol. IV*. The MIT Press, 1984. *Uber die 'doppelt-periodischen' Losungen des n-Damen-Problems.*

13 Appendices

A main.py

```python
# Imports
import numpy as np
import time

def user_interface():
    print("Welcome to DLX\n",
          "This algorithm solves an exact cover problem of your choice.\n")
    choice = 'Z'
    while choice != 'A' and choice != 'B':
        print("Please input the letter of your choice: 'A' or 'B'\n",
          "A.) Solve the N-Queens problem\n",
          "B.) Import your favourite matrix to solve (you will need the path/filename of this matrix)")
        choice = input("Answer: ").upper()
        if choice == 'A':
            print("You have chosen option A: \t Solve the N Queens problem.\n Please now specify N.")
            n = int(input("N = "))
            log_decision = 'Z'
            while log_decision != 'Y' and log_decision != 'N':
                print("Would you like an extensive log of all steps taken? (Will increase execution time of DLX)\n",
                      "Please answer: (Y/N)")
                log_decision = input().upper()
            if log_decision:
                begin_dlx_n_queen(n, True)
            else:
                begin_dlx_n_queen(n, False)
            return None
        elif choice == 'B':
            print("You have chosen option B:\t Solve your favourite matrix.")
            delimiter_type = 'Z'
            while delimiter_type != 'A' and delimiter_type != 'B':
                print("Please now specify if the values in your file are separated by:\nA.) commas \nB.) whitespace")
                delimiter_type = input("Answer: ").upper()
            print("Please now input the filename with path, including the extension, that holds the your favourite matrix.",
                  "\nFor example: user/project/dlx/input.csv")
            print("If the file is in the same directory as I am, no need to input the path.")
            filename = input("Filename (and path): ")
            try:
                if delimiter_type == 'A':
                    input_matrix = py.loadtxt(filename, dtype = int, delimiter=',')
                    print(input_matrix)
                elif delimiter_type == 'B':
                    input_matrix = py.loadtxt(filename, dtype=int)
                    print(input_matrix)
                log_decision = 'Z'
                while log_decision != 'Y' and log_decision != 'N':
                    print("Would you like an extensive log of all steps taken? (Will increase execution time of DLX)\n",
                          "Please answer: (Y/N)")
                    log_decision = input().upper()
                if log_decision:
```

```
                    begin_dlx_user_input_matrix(input_matrix, True)
                else:
                    begin_dlx_user_input_matrix(input_matrix, False)
                return None
            except:
                print("Error: Could not open file, please try again")
                return None
            return None

# Specific NQueens function: Creates an empty matrix of size n^2 by 2(3n-3),
# to hold all possible placements and constraints.
# See report for details about these dimensions.
# Arguments: n = size of board
# Return: one_zero = Empty matrix
def create_one_zero_matrix(n):
    one_zero = np.zeros(((n**2), (2*(3*n-3))), dtype=int)
    return one_zero


# Specific NQueens function: Populates the one-zero matrix according to all possible queen placements
# Arguments: one_zero_matrix = empty 1-0 matrix created by 'create_one_zero_matrix', n = size of board
# Return: one_zero_matrix = matrix defining the exact cover problem
def populate_one_zero_matrix(one_zero_matrix, n):
    counter = 0
    # Iterate over the row indices, from (0,j) to (n,j)
    for i in range(n):
        x = i
        # Iterate over the column indices, from (i,0) to (i,n)
        for k in range(n):
            current_row = []    # Define an emtpy row, with length equal to the number of constraints
            for p in range(2 * (3 * n - 3)):
                current_row.append(0)
            y = k
            # Compute the diagonal and backward diagonal constraints
            diag_constraint = (2*n - 1) + x + y
            back_diag_constraint = 5*n - 5 - x + y
            # Now populate the current row with 1's wherever constraints are satisfied
            current_row[x] = 1
            current_row[y + n] = 1
            # Check to see if this is a significant diagonal
            if (2 * n - 1) < diag_constraint < (4*n - 3):
                current_row[diag_constraint] = 1
            # Check to see if this is a significant backward diagonal
            if (4*n - 3) <= back_diag_constraint < (6*n - 6):
                current_row[back_diag_constraint] = 1
            one_zero_matrix[counter] = current_row
            counter = counter + 1
    return one_zero_matrix

# Class declaration for the regular nodes.
# All attributes initialised to 'None' by default
class Node:
    def __init__(self, left=None, right=None, up=None, down=None, column=None):
        self.left = left        # Points to the (node/column header) to the left of this object
        self.right = right      # Points to the (node/column header) to the right of this object
        self.up = up            # Points to the (node/column header) above this object
        self.down = down        # Points to the (node/column header) below this object
        self.column = column    # Points to the column header of the column this object belongs to
```

```


# Class declaration for the column headers
# All attributes initialised to 'None' or '0' by default, except primary attribute set to 'True'
class Column:
    def __init__(self, left=None, right=None, up=None, down=None, size=0, name=None, primary=True):
        self.left = left        # Points to the (node/column header) to the left of this object
        self.right = right      # Points to the (node/column header) to the right of this object
        self.up = up            # Points to the (node/column header) above this object
        self.down = down        # Points to the (node/column header) below this object
        self.size = size        # Refers to the number of nodes in this object's column (below)
        self.name = name        # Cosmetic attribute for outputting solutions
        self.primary = primary  # If set to 'False' object cannot be chosen by DLX and can be unsatisfied for solutions


# Class declaration for the overall list object.
# The master header is specified initially and its attributes are defined accordingly.
# The solution_list/total_solutions variables are initialised.
# The main and log file names are stored as attributes of this object.
class FourWayLinkedList:
    def __init__(self, main_file_name="main_output.txt", log_file_name="log.txt", master_node=Column(name="Master",primary=False)):
        self.main_file = main_file_name     # Store name of MAIN file for access later
        self.log_file = log_file_name       # Store name of LOG file for access later
        self.master_node = master_node      # Create a column header to be the master node
        self.solution_list = []             # Used to store the nodes in the solution
        self.total_solutions = 0            # Used to count the number of solutions, for labelling their output later
        self.header_list = []               # Stores the original order of header names, for outputting solutions

    # Helper function: Finds a named column's index
    # Starts at the master node
    # Arguments: name = The name of the column to search for
    # Return: The index of the column
    def find_column_index_by_name(self, name):
        current_node = self.master_node
        index = 0
        while current_node.name != name:
            current_node = current_node.right  # Step right
            index = index + 1  # Increase index
        return index

    # Helper function: Finds a particular column header, given its index
    # Arguments: index = The index of the column to be found
    # Return: column object
    def find_column_by_index(self, index):
        current_column = self.master_node
        # Take number of steps right equal to the index
        for i in range(index):
            current_column = current_column.right  # Step right
        return current_column

    # Specific N-Queens function: Transform the column headers to resemble the N-Queens problem.
    # Changes the names of the column headers according to n.
    # Sets the column.primary attributes to false for the diagonal and back diagonal constraints.
    # Calls the NQueen specific file write function, to record these changes for the user.
    # Arguments: n = number of ranks/files of the board
```

```
    # Return: None
    def transform_n_queen(self, n):
        current_column = self.master_node
        # Iterate over all headers, using a for loop to easily track the index
        for i in range(2 * (3 * n - 3)):
            current_column = current_column.right  # Step right
            if i < n:   # Ranks
                current_column.name = "Rank {0}".format(i + 1)
            elif i < 2*n:   # Files
                current_column.name = "File {0}".format(int((i % n) + 1))
            elif i < (4*n - 3): # Diagonals
                current_column.name = "Diagonal {0}".format(int((i % (2*n)) + 1))
                current_column.primary = False
            else:   # Back Diagonals
                current_column.name = "Back Diagonal {0}".format(int((i % (4*n - 3)) + 1))
                current_column.primary = False
        self.file_write_n_queen(n)  # Write the according introduction to the main output file
        self.create_original_header_list()  # Needs to be called again here as the column header's names have
    changed
        return None

    # Helper function: Updates the solution list by adding a new node
    # The list index is changed to the new node, or the new node is appended if the index is out of range
    # Arguments: new_node = the new node to be added to the solution
    # k = the depth of the DLX algorithm/ the index of the list to be updated
    # Return: None
    def set_solution_k(self, new_node, k):
        try:
            self.solution_list[k] = new_node  # Try to update index k
        except:
            self.solution_list.append(new_node)  # If out of range, append instead
        # Algorithm DLX will not 'skip' a k, ie. the index will never be more than one step out of range
        return None

    # Debug function: Prints all of the four way linked list object's column headers, along with their size,
    # to the standard console output.
    # Useful for monitoring the list throughout DLX
    # Arguments: None
    # Return: None
    def print(self):
        current_header = self.master_node.right
        while current_header != self.master_node:
            print(current_header.name, current_header.size)
            current_header = current_header.right
        return None

    # Debug function: Prints the solution list to the standard console output
    # Useful for monitoring the progress throughout DLX
    # Arguments: None
    # Return: None
    def print_solution(self):
        print("Solution")
        for i in range(len(self.solution_list)):
            print(self.solution_list[i].column.name, self.solution_list[i].right.column.name)
        print("End Solution")
        return None

    # File function: Calls the appropriate functions to initialise the output ad log files.
```

```
    # Arguments: log = boolean value specifying if the user desires an extensive log
    # Return: None
    def begin_file_writing(self, log):
        self.main_file_initial()
        if log:
            self.log_file_initial()
        return None

    # File function: Begins the main output file, also writing a small introduction
    # Arguments: None
    # Return: None
    def main_file_initial(self):
        # Use the write method here to overwrite any existing file contents
        solution_file = open(self.main_file, "w")  # Open the file in write mode
        solution_file.write("Algorithm DLX\n\n")
        solution_file.write("This algorithm finds all solutions to an exact cover problem.\n")
        solution_file.write("An implementation of Donald Knuth's algorithm X, using dancing links, is used.\n")
        solution_file.write("For more information visit: https://github.com/Hedge-Hodge/Dancing_Links_N_Queens\n")
        solution_file.write("This file contains these solutions.\n")
        solution_file.close()  # Good practice to close files when finished
        return None

    # File function: Begins the log file, also writing a small introduction
    # Arguments: None
    # Return: None
    def log_file_initial(self):
        log_file = open(self.log_file, "w")
        log_file.write("DLX Log\n\n")
        log_file.write("See '{0}' for the proper algorithm output and aesthetic solution list.\n".format(self.main_file))
        log_file.write("This file contains a detailed record of each iteration of the recursive DLX algorithm.\n")
        log_file.write("Each time a new row is chosen, this will be logged here.\n")
        log_file.write("Each time the algorithm finds itself in a 'dead end' (i.e. when some of the remaining "
                       "constraints have size=0) this will be recorded also.\n\n\n")
        log_file.write("Begin log:\n")
        log_file.close()
        return None

    # N-Queens/File function: Add a brief NQueens specific description to the main output file
    # Arguments: N = the number of ranks/files of the chessboard
    # Return: None
    def file_write_n_queen(self, N):
        solution_file = open(self.main_file, "a")  # Open the file in append mode
        solution_file.write("Here we have the classic N-Queens exact cover problem, with:\n")
        solution_file.write("N = {0}\n".format(N))
        solution_file.write("The columns of the matrix above correspond to the row, column and diagonal constraints.\n")
        solution_file.write("While the rows correspond to possible queen placements.\n")
        solution_file.close()
        return None

    # File function: Used to write the 1-0 matrix to the main output file, used in general and with NQueens
    # Arguments: matrix = The numpy n dimension array holding the 1-0 matrix
    # Return: None
    def file_write_one_zero(self, matrix):
        solution_file = open(self.main_file, "a")  # Open the file in append mode
        solution_file.write("\nThe following matrix represents the exact cover problem in question:\n")
        np.savetxt(solution_file, matrix, delimiter=' ', fmt='%i')  # Here the fmt argument writes only integers
```

```
        solution_file.close()
        return None

    # File function: Write a single iteration of DLX to the log
    # This write will include the depth of the algorithm as well as the row it has chosen to try.
    # This will also record a BACKTRACK in the log, depending on the backtrack argument
    # Arguments: node = the node/row to be recorded
    # k = the depth of the algorithm at this time, default = 0
    # backtrack[boolean] = whether or not to record a backtrack in the log, default = False
    # Return: None
    def file_write_log_row(self, node, k=0, backtrack=False):
        log_file = open(self.log_file, "a")
        # If the algorithm is in a 'dead-end' record a backtrack in the log
        if backtrack:
            log_file.write("BACKTRACK necessary,\t")
            log_file.write(node.name)
            log_file.write("\tis a dead constraint.\n")
        # Otherwise write this row into the solution
        else:
            current_node = self.find_furthest_left(node)
            #current_node = node
            # Write the current depth of the algorithm
            log_file.write("k={0}\n".format(k))
            # Simple placeholder
            #log_file.write("Rank:\t")
            log_file.write(current_node.column.name)
            dummy_node = current_node.right
            # Iterate across the row, writing the name of the node's column header each time
            while dummy_node != current_node:
                log_file.write("\t")
                log_file.write(dummy_node.column.name)
                dummy_node = dummy_node.right
            log_file.write("\n")
        log_file.close()
        return None

    # File function: Used to write a single solution to either output file
    # The filename is passed as an argument here allowing this to be used in both the main output and log files
    # Some bad practice here, with if statements. Open to suggestions.
    # Arguments: main_file = boolean value, true for the main file, false for the log file
    # Return: None
    def file_write_solution(self, main_file=True):
        # Only update the counter for the main output file, otherwise we would update twice for each solution
        if main_file:
            self.total_solutions = self.total_solutions + 1
            file = open(self.main_file, "a")
        else:
            file = open(self.log_file, "a")
        # If this is the first time writing a solution, include this header
        if self.total_solutions == 1:
            file.write("\n\nSolutions:\n\n")
        # This sub-header provides the solution number, equal to the total number of solutions at the time of writing
        file.write("Solution {0}\n".format(self.total_solutions))
        # Write the entire solution list to the file
        for i in range(len(self.solution_list)):
            furthest_left = self.find_furthest_left(self.solution_list[i])
            # Write the node in the solution, as well as the node to the right of it
```

```python
            file.write(furthest_left.column.name)
            file.write(", ")
            file.write(furthest_left.right.column.name)
            file.write("\n")
        file.write("\n")
        file.close()
        return None

    # Helper function: Finds a column header's original index, regardless of any current covered columns.
    # This function facilitates the find_furthest_left function, to ensure rows of the solution always start with the
    # furthest left node.
    # Arguments: name = the name of the column to be searched for
    # Return: index = the column header's original index.
    # Note this can also return None, if no match was found for the inputted name.
    def find_original_index_by_name(self, name):
        index = 0
        for i in range(len(self.header_list)):
            if name == self.header_list[i]:
                return index
            index = index + 1
        print("Warning: No matching name found in the original column header list.")
        return None

    # Helper function: Initialises the header list, used to ensure the order in which nodes in the solution are written
    # to file, is correct. This MUST be called each time the column header's names change.
    # Arguments: None
    # Return: None
    def create_original_header_list(self):
        current_header = self.master_node.right
        while current_header != self.master_node:
            self.header_list.append(current_header.name)
            current_header = current_header.right
        return None

    # Helper function: Finds the furthest left node in a row. Used to ensure the order in which nodes in the solution
    # are written to file are correct.
    # Arguments: current_node = a node in the row to be searched
    # Return: best_node = the furthest left node
    def find_furthest_left(self, current_node):
        dummy_node, best_node = current_node.left, current_node
        #best_node = current_node
        best_index = self.find_original_index_by_name(current_node.column.name)
        while dummy_node != current_node:
            dummy_index = self.find_original_index_by_name(dummy_node.column.name)
            if dummy_index < best_index:
                best_node, best_index = dummy_node, dummy_index
                #best_index = dummy_index
            dummy_node = dummy_node.left
        return best_node


    # Core function: This very important function converts a exact cover matrix into a general list object
    # No checks are performed to see if the problem is well defined
    # The column headers are given default names in the format: "constraint {i}" from 0, number of constraints
    # This function has three sections:
```

```
    #   1.) Create the column headers
    #   2.) Create the rows
    #   3.) Join any loose ends on the left/right of rows, and top/bottom of columns
    # Arguments: matrix = the 1-0 matrix to be converted
    # Return: master_node = the master node of the list object, so its pointer can be stored elsewhere outside
    # these methods
    def convert_exact_cover(self, matrix, log):
        self.begin_file_writing(log)
        self.file_write_one_zero(matrix)  # First record the matrix in the main output file
        dims = np.shape(matrix)  # Find the dimensions of the matrix
        x, y = dims  # Number of rows, number of columns
        # Create the column headers
        previous_header = self.master_node
        for i in range(y):
            new = Column(left=previous_header, right=self.master_node, name="Constraint {0}".format(i))  # Initialise new column header
            new.up, new.down = new, new
            previous_header.right = new
            self.master_node.left = new
            previous_header = new   # Update pointer for next iteration
        # Create each row
        for i in range(x):
            current_row = matrix[i]     # Extract corresponding row from 1-0 Matrix
            # Iterate over the extracted row
            prev_node = None
            for j in range(len(current_row)):
                current_node = Node()
                current_node.left, current_node.right = current_node, current_node
                if current_row[j] == 1:         # If significant, as we only record 1s from the 1-0 matrix
                    if prev_node is not None:
                        current_node.right, prev_node.right.left = prev_node.right, current_node
                        current_node.left, prev_node.right = prev_node, current_node
                    current_node.column = self.find_column_by_index(j + 1)          # Define column header for new node
                    current_node.column.size = current_node.column.size + 1
                    current_above = current_node.column
                    while current_above.down != current_node.column:            # Find 'lowest' node in the column
                        current_above = current_above.down
                    current_above.down, current_node.column.up = current_node, current_node
                    current_node.up, current_node.down = current_above, current_node.column
                    prev_node = current_node
        return self.master_node

    # DLX helper function: Cover a column of the list object
    # Implemented as directly as possible from Dancing Links paper by Knuth.
    # Alters the links around a column header, and around the nodes in each row of a column such that they are removed
    # from the list.
    # Arguments: column = the column header of the column to be covered
    # Return: None
    def cover_column(self, column):
        # Remove column header from the header chain
        column.left.right = column.right  # Alter link to the left
        column.right.left = column.left  # Alter link to the right
        # Iterate down through the column
        current_down = column.down
        while current_down != column:
            # Iterate right across the row
```

```
            current_right = current_down.right
            while current_right != current_down:
                current_right.down.up = current_right.up  # Alter link below
                current_right.up.down = current_right.down  # Alter link above
                current_right.column.size = current_right.column.size - 1  # Alter column size
                current_right = current_right.right  # Step right
            current_down = current_down.down  # Step down
        return None

    # DLX helper function: Uncover a column of the list object
    # Implemented as directly as possible from Dancing Links paper by Knuth.
    # Alters the links around a covered column header, and around the nodes in each row of a column such that they are restored
    # to the list.
    # This is the inverse of the cover_column method, notice that all operations are reversed and done in the opposite order
    # Arguments: column = the column header of the column to be uncovered
    # Return: None
    def uncover_column(self, column):
        current_up = column.up
        while current_up != column:
            current_left = current_up.left
            while current_left != current_up:
                current_left.column.size = current_left.column.size + 1
                current_left.down.up = current_left  # Restore link below
                current_left.up.down = current_left  # Restore link above
                current_left = current_left.left  # Step left
            current_up = current_up.up  # Step up
        # Add column to the header chain
        column.left.right = column
        column.right.left = column
        return None

    # DLX helper function: Find the column with the smallest size to cover
    # By choosing the column with the smallest size, we limit the branching factor of the algorithm and increase it's
    # speed.
    # Arguments: None
    # Return: None
    def find_best_column(self):
        current_header = self.master_node.right
        best_header = current_header
        while current_header != self.master_node:
            # We only want to choose primary headers, there should only be non-primary headers in the NQueen application
            if current_header.size < best_header.size and current_header.primary:
                best_header = current_header
            current_header = current_header.right
        return best_header

    # DLX helper function: Check to see if the current list object has an columns of size=0, if it does the constraint
    # is dead. In this case a backtrack is necessary.
    # Arguments: None
    # Return: None
    def dead_constraint(self, log):
        current_header = self.master_node.right
        while current_header != self.master_node:
```

```
            # We do not care is non-primary constraints are dead
            if current_header.primary and current_header.size <= 0:
                if log:
                    self.file_write_log_row(current_header, backtrack=True)  # Log this backtrack
                return True
            current_header = current_header.right
        return False

    # Main DLX function: This is where we lose ourselves to dance(Daft Punk).
    # This calls many of the methods above.
    # This is a recursive function, see report for more details
    # Arguments: k = depth of the algorithm
    # Return: None
    def dlx(self, k, log=True):
        #print("Starting algorithm DLX. k=", k)  # DEBUG
        # self.print()  # DEBUG
        # self.print_solution()  # DEBUG
        # If the only constraints remaining are non-primary ones, we have found a solution!
        if not self.master_node.right.primary:
            #print("O frabjous day! Callooh! Callay!")  # DEBUG
            #self.print_solution()  # DEBUG
            # Write this solution to both output files
            self.file_write_solution(True)
            if log:
                self.file_write_solution(False)
            return None
        else:
            # Check to see if there are any dead constraints
            if self.dead_constraint(log):
                #print("Dead constraint, need to backtrack")  # DEBUG
                # Return as the problem is not well defined anymore
                return None
            # Choose the column with the smallest size, by calling the find_best_constraint function
            current_column = self.find_best_column()
            # Best column now found
            #print("Best column found, ", current_column.name)  # DEBUG
            # Branch now for each row in this column
            current_node = current_column.down  # Start below the column header, iterate down from here
            self.cover_column(current_column)  # First cover this column
            while current_node != current_column:
                self.set_solution_k(current_node, k)  # Add this to the solution list, will be overwritten if not a
                # solution.
                # Record this step of the algorithm in the log
                if log:
                    self.file_write_log_row(current_node, k, backtrack=False)
                # Iterate across this row
                current_right = current_node.right
                while current_right != current_node:
                    # Cover the column this node belongs to
                    self.cover_column(current_right.column)
                    current_right = current_right.right  # step right
                # print("Recursive call")  # DEBUG
                # Call dlx again, with depth += 1
                self.dlx(k+1, log)
                current_node = self.solution_list[k]  # Retrieve the current node from the solution list
                current_column = current_node.column  # Find its column
                # Iterate left to uncover
```

```
                current_left = current_node.left
                while current_left != current_node:
                    # Uncover column of current node
                    self.uncover_column(current_left.column)
                    current_left = current_left.left  # step left
                current_node = current_node.down  # Step down
            self.uncover_column(current_column)  # Uncover the column originally chosen as best
        return None


def begin_dlx_n_queen(n, log):
    start_time = time.time()
    print("Solving N Queens problem, for N = ", n)
    print("Creating 1-0 Matrix...")
    one_zero_matrix = create_one_zero_matrix(n)
    populate_one_zero_matrix(one_zero_matrix, n)
    print("Done.")
    print("Solving now for:\n", one_zero_matrix)
    overall_list = FourWayLinkedList("{0}_queen_output.txt".format(n),"{0}_queen_log.txt".format(n))
    overall_list.convert_exact_cover(one_zero_matrix, log)
    overall_list.transform_n_queen(n)
    master_node = overall_list.master_node
    #test_circular_list(master_node)
    overall_list.dlx(0, log)
    print("Algorithm DLX finished, execution time:")
    if overall_list.total_solutions == 0:
        print("It appears no solutions were found for your matrix, the problem may not be well defined")
    print("--- %s seconds ---" % (time.time() - start_time))
    print("Output can now be seen in '{0}_queen_output.txt'".format(n))
    return None


def begin_dlx_user_input_matrix(user_input_matrix, log):
    start_time = time.time()
    print("Now solving your favourite matrix:\n", user_input_matrix)
    overall_list = FourWayLinkedList()
    overall_list.convert_exact_cover(user_input_matrix, log)
    master_node = overall_list.master_node
    #test_circular_list(master_node)
    overall_list.dlx(0, log)
    print("Algorithm DLX finished, execution time:")
    if overall_list.total_solutions == 0:
        print("It appears no solutions were found for your matrix, the problem may not be well defined")
    print("--- %s seconds ---" % (time.time() - start_time))
    print("Output can now be seen in 'main_output.txt'")
    return None


def test_circular_list(master_node):
    test_right = master_node.right
    while test_right != master_node:
        test_below = test_right.down
        print(test_right.name)
        while test_below != test_right:
            counter = 0
            row_right = test_below.right
            while row_right != test_below:
                counter = counter + 1
```

```
                row_right = row_right.right
            test_below = test_below.down
        test_right = test_right.right
    return None


user_interface()
```