Donald Knuth's Dancing Links

Oisín Hodgins, Seán Monahan
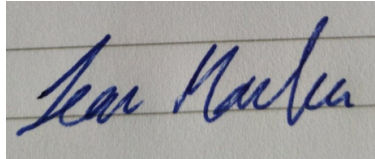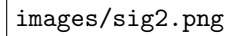
NUIG Final Year Project (Mathematics)

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.

images/sig2.png

Signed: Sean Monahan, Oisín Hodgins

2   Introduction

Donald Knuth is seen as the founder of many aspects of modern computer science, often considered the "father of the analysis of algorithms". Even the typeset that this project is written in, LaTeX, is built using the TeX system developed by Knuth. Our motivation for this project is to gain some insight into how the field of computer science developed in to what it is today, with credit to Knuth's vast list of contributions. In this report, we will be looking at his *Dancing Links* technique used in conjunction with *Algorithm X* from Knuth's paper in [7]. It's a fascinating case where a simple operation, which was overlooked by most programmers at the time, improved the solving time of the exact cover problem immensely. In this project we seek to uncover all aspects of the Dancing Links technique, as well as how it can be implemented using python.

2.1   History

One of our primary motivations in undertaking this project, was the historical significance of algorithm DLX, and highly influential creator Donald Knuth. Informally referred to as the "father of analysis of algorithms" and formally awarded the ACM Turing Award in 1974 (among many others), Donald Knuth is a fundamental figure in the field of computer science. His published works include "*The art of computer programming*", a multi-volume comprehensive monograph, detailing many aspects of computer programming but particularly algorithms and an analysis of them. It is through one of these volumes, "*The Art of Computer Programming. 4, Fascicle 5: Mathematical Preliminaries Redux, Backtracking, Dancing Links*", that we initially discovered algorithm DLX, after it was kindly recommended by our supervisor.

3   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. Given a set X, and a collection of its subsets S, the definition of an **exact cover**, from [2], 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.

3.1 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 ith 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 1 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 [7]:

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

3.2 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 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
$A@(1,1)$	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:

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

change this part 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.

3.3   N-Queens Problem

In this section 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 [5] and consequently of Hitotumatu and Noshita's implementations in [6]. 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.
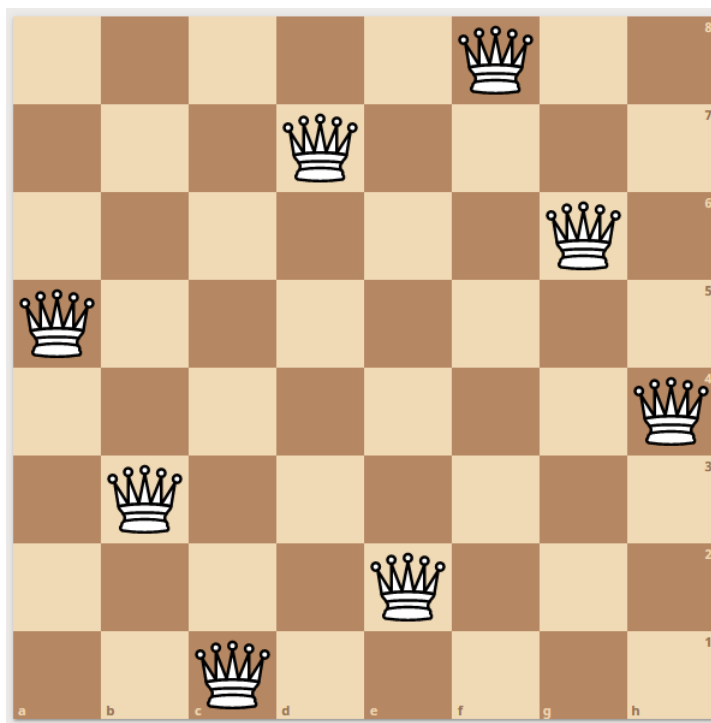
Figure 1: An 8 Queens Solution

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 to have a Queen**. This is why the N-Queens problem is considered to be a generalized exact cover problem. When considering this in a matrix representation, our exact cover has N^2 moves (rows), and $6 * (N - 1)$ constraints (columns), so for large N one can imagine that this 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.

3.4 NP-Complexity

Polynomial time is a metric used to classify the complexity of a decision problem. From [3], 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.

4   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.
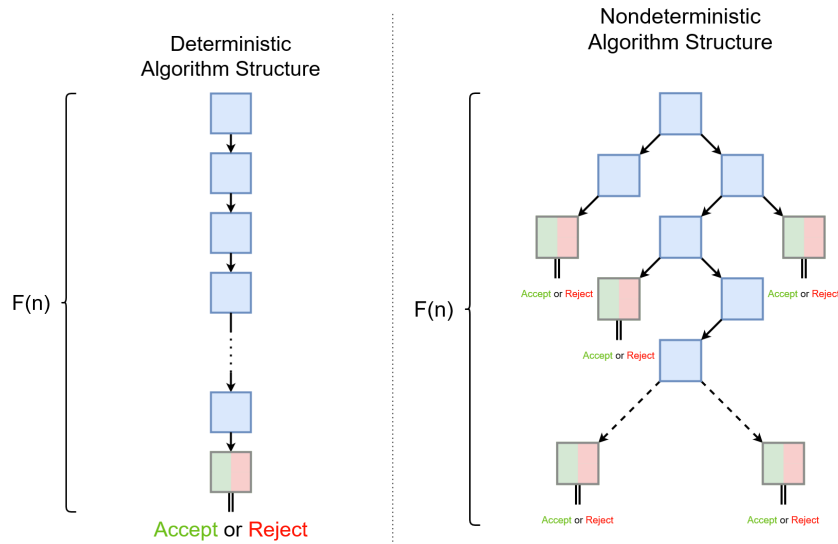
Figure 2: Deterministic vs. Nondeterministic Algorithms

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. This leads to the nondeterministic branching structure seen in Figure 2. When incrementally building a solution, **backtracking** allows the algorithm to revisit unsearched 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.
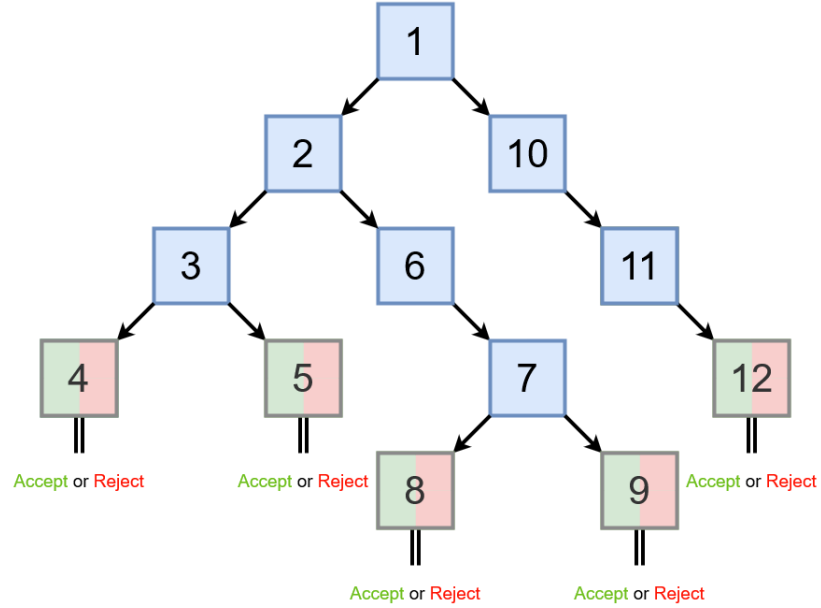
Depth-first Backtracking

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.

4.1   Algorithm X Pseudo-code

We will base our pseudo-code on Knuths' in [7] 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 ith row of M.

- Let c_j be the jth column of M.

13

Using this notation we can define the algorithm as follows:

Algorithm 1 Algorithm X

1: **if** U is empty **then**: the problem is solved **return** M
2: **end if**
3: Otherwise choose a column c from set U
4: Choose a row r of c that has value 1 (nondeterministically)
5: **for** each j such that $M[r, j] = 1$ **do**:
6:     remove column j from set U
7:     **for all** i such that $M[i, j] = 1$ **do**:
8:         delete row i from matrix M
9:     **end for**
10: **end for**
11: 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 all removing other positions that have conflict with that constraint. Then we continue this process until a solution is found. This is implementation is slightly different to Knuth's which does not include U, this 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 [7], "Algorithm X is simply a statement of the obvious trial-and-error approach".

4.2   Algorithm X with a $2X2$ Latin Square (by hand)

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.

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

15

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.

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

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.

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

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

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

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

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

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. 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.

4.3   Problems with Algorithm X

With the classic Algorithm X approach, we use standard matrices to represent the exact cover problem. For complex problems such as the 8 Queens Problem, this results in sparse matrices that are non-circular. Searching through massive matrices causes longer solving times due to the increased memory usage. Our goal now is to establish a method to link each element in the matrix such that we can search efficiently.

5 Dancing Links

Dancing links is a technique that was named and popularised by Donald Knuth in [7] which cleverly utilizes doubly linked lists to make a simple "covering" function. A doubly linked list is a data structure where each node is linked to both the previous node and the following node in the structure. A visualization of this is seen in Figure 4.
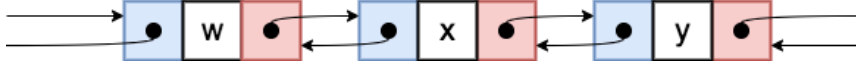

Figure 4: Doubly-linked list structure

To explain the core concept of Dancing Links, we'll first introduce the following functions. Suppose we have a doubly linked list with that has an element 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 5.
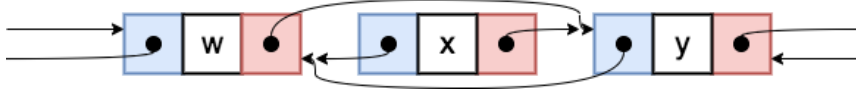

Figure 5: Doubly-linked list with node x removed

Note that element x retains its information on elements w and y. This is a simple backtracking operation that was well known before Knuth's paper, essentially we "cut out the middleman" to remove an unwanted element 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 stucture seen in Figure 4. 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 the Hirosi Hitotumatu and Kohei Noshita with the idea in their paper in [6], who 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.

6 Four Way Linked Lists

As discussed in section 5 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 [6], 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 [7], who instead used a structure which he named the '**four way linked list**'.

In section 4.3 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.

6.1 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 start point for all operations on the four way linked lists.

We can see an illustration of one of these list objects below, created using *Drawio* [1]:
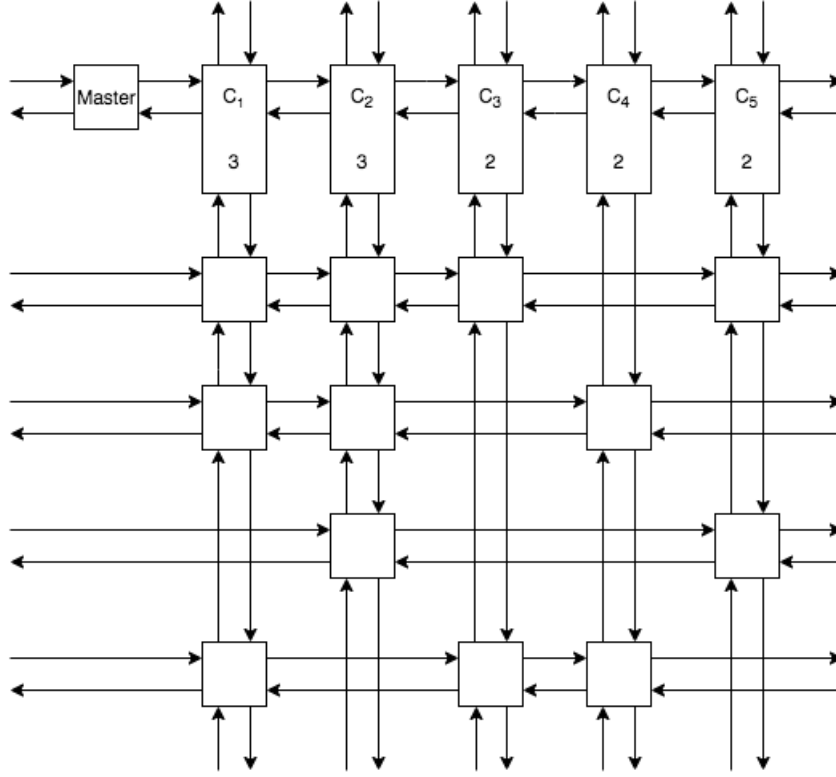
Figure 6: Illustration of four way linked list

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, which in fact results in the proper 3-d object having a distinct *torus* shape, which is not relevant to the discussion at hand but none-the-less is somewhat interesting.

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.

6.2   Node

The nodes we can see in figure 6 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 6 corresponds to the row vector $(1, 0, 1, 1, 0)$.

6.2.1   Node: Implementation in Python

The first difficulty we encountered during the implementation was the absence of linked lists 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: **We will have to implement linked lists ourselves.**

You may ask; "*Why not use a third party library or existing implementation?*, 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.

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:

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

Figure 7: 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.

6.3   Column Header

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 [7]. 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 6, 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**. The use of this attribute will be discussed in more detail in section *N Queens section number*.

6.3.1   Column Header: Implementation in Python

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

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

Figure 8: 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.

6.4 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, master_node=Column()):
        self.master_node = master_node
        master_node.name = "Master"
        master_node.primary = False
        self.solution_list = []
        self.total_solutions = 0
        self.file_write_initial()
        self.file_write_initial_log()
        self.header_list = []
```

Figure 9: 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 **bookkeeping** 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 6 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 **bookkeeping**.

6.4.1   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 **DLX section**. 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.
In light of

Now that the framework is in place to create four way linked lists, it is time to define a method of the class 'FourWayLinkedList' which will convert standard NumPy matrices into an instance of this class. This will allow the user to import their own favourite matrix into the algorithm (in a space or comma separated text file) or simply to use the default N Queens example for an arbitrary N value.

This conversion takes place in the 'convert_exact_cover' function, and can be briefly described as:

1. Create a list of headers, with one header 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 and down links accordingly

The difficulty here is setting these links in an efficient manner, and the pseudocode below briefly describes the exact method used (here master is the master header, and the p pointer represents the previous node or column):

The core principal here 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 code above.

Algorithm 2 convert_exact_cover

1: Set $p \leftarrow Master$
2: **for** each column in matrix **do**:
3:     Create Column c
4:     Set $R[p] \leftarrow c$
5:     Set $L[c] \leftarrow p$
6:     Set $R[c] \leftarrow Master$
7:     Set $L[Master] \leftarrow c$
8:     Set $U[c] \leftarrow c$
9:     Set $D[c] \leftarrow c$
10:     Set $p \leftarrow c$
11: **end for**
12: **for** each row in matrix **do**:
13:     **for** each j in row **do**:
14:         **if** row$[j]$ is 1 **then**:
15:             Create node n
16:             Set $L[n] \leftarrow p$ and $R[p] \leftarrow n$
17:             $c \leftarrow R[Master]$
18:             **Do j times**: $c \leftarrow R[c]$
19:             Set $C[n] \leftarrow c$
20:             Set $S[C[n]] \leftarrow S[C[n]] + 1$
21:             **While** $c \neq C[n]$ **do**: $c \leftarrow D[c]$
22:             $D[c] \leftarrow n$
23:             $U[n] \leftarrow c$
24:             $U[C[n]] \leftarrow n$
25:             $D[n] \leftarrow C[n]$
26:         **end if**
27:     **end for**
28: **end for**

7 Algorithm DLX

In this section we will discuss the low level details concerning the implementation of algorithm DLX, as outlined by Donald Knuth in the Dancing Links paper [7].

7.1 The Dance in Action

Before we can begin implementing algorithm DLX, we must first implement two essential methods, namely 'cover_column' and 'uncover_column'. The first of these methods removes a chosen column from the four way linked list, along with all of the rows in that column, while the second method entirely reverses the first, by returning the column, along with its rows, into the list.

We closely followed Donald Knuth's pseudocode[7] during the implementation of these methods, both of which will be declared as methods of the 'FourWayLinkedList'.

7.1.1 The cover_column Method

First we consider the 'cover_column' method, described by the following pseudocode[7]:

Algorithm 3 Covering a Column c

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

This were implemented as directly as possible in Python, we can see the code used here:

```
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_node = column.down
    while current_node != column:
        # Iterate right across the row
        current_right = current_node.right
        while current_right != current_node:
            # Need to specify a temporary pointer to the node below & below
            current_below = current_right.down
            current_above = current_right.up
            current_below.up = current_above  # Alter link below
            current_above.down = current_below  # Alter link above
            current_right.column.size = current_right.column.size - 1  # Alter column size
            current_right = current_right.right  # Step right
        current_node = current_node.down  # Step down
    return None
```

Figure 10: Definition of the cover_column method

This method is applied to the chosen constraint during algorithm DLX, which it covers in such a manner that this constraint will not be chosen again by the algorithm, and any rows that also satisfy this constraint are removed (until the column is subsequently uncovered). This is achieved by temporarily 'removing' this column's header from the overall list object, along with all of the rows in this column. First consider the list object as discussed earlier:
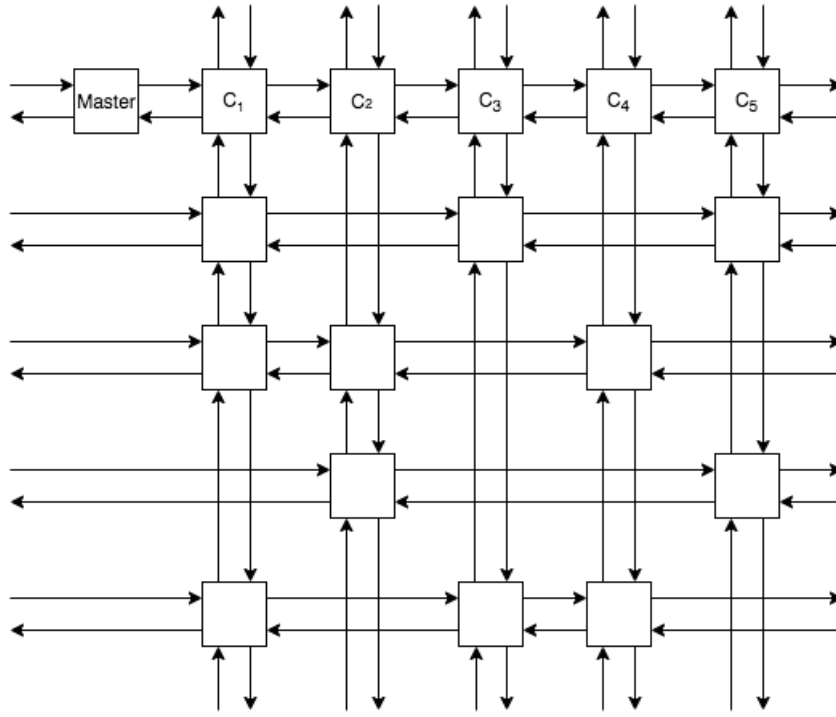
Figure 11: Illustration of a Four Way Linked List before any action

Suppose we would like to call the above method to cover the third constraint here, C_3, which is highlighted now in red.
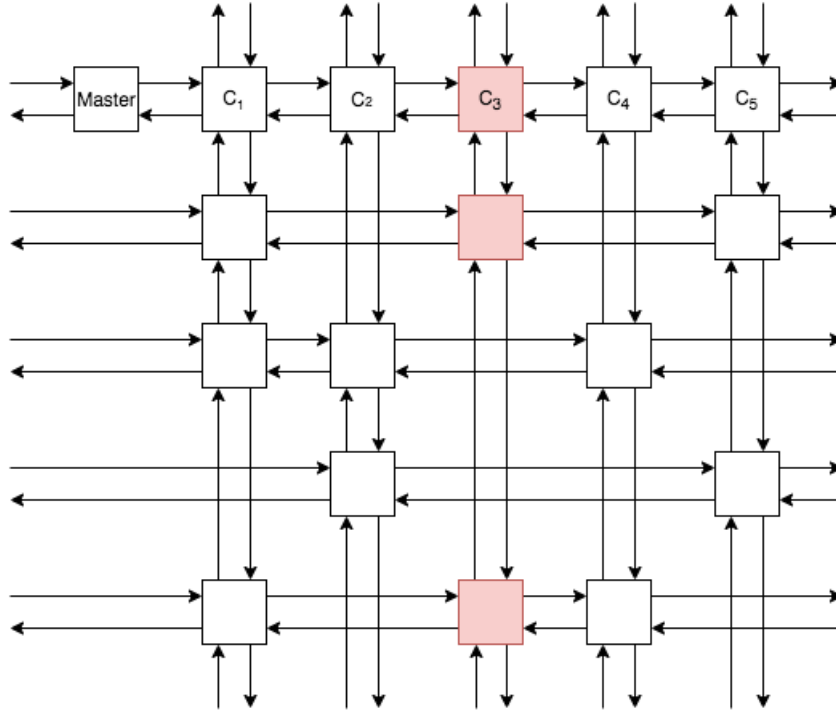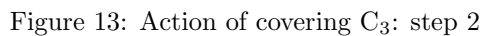
Figure 12: Action of covering C_3: step 1

The first step would be to remove the C_3 column header from the header list (the first row as seen in this diagram). As outlined in the pseudocode, we must alter the links to the left and right of C_3 such that they 'bypass' C_3 in the list:

Figure 13: Action of covering C_3: step 2

The links highlighted in blue have been altered by the operations:

- $R[L[C_3]] \leftarrow R[C_3]$

- $L[R[C_3]] \leftarrow L[C_3]$

Once algorithm DLX is called again, it will iterate through the header list in an effort to choose the best column to cover, this search operation will pass from C_1 to C_2 to C_4 etc skipping the now covered C_3 column. However, we must also remove all of the rows with a node in C_3 as well, as this constraint is satisfied once already, and our goal is to satisfy each constraint exactly once. Focusing our attention on the first row of the C_3 (highlighted in red) we can see the action of covering it.
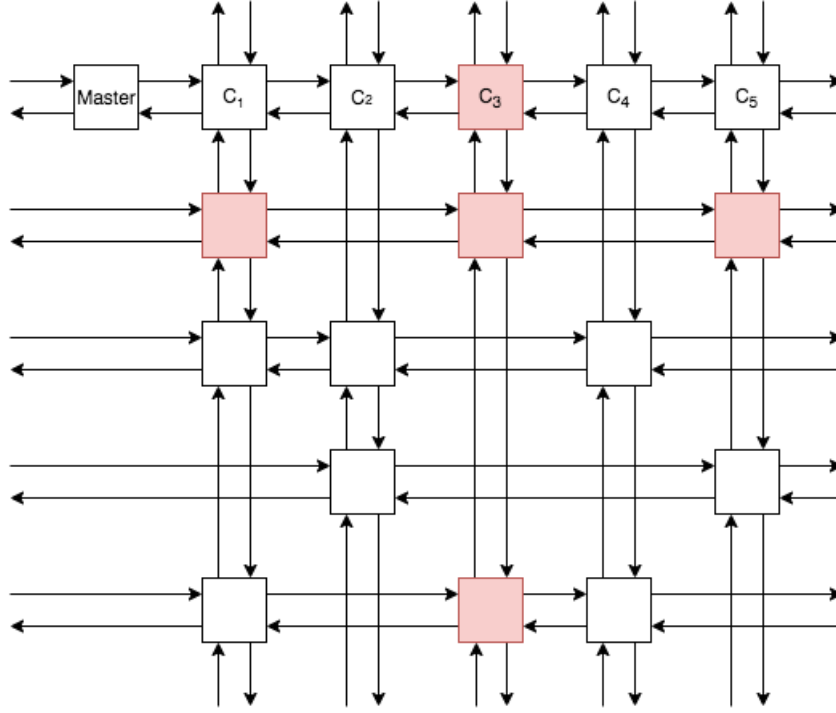
Figure 14: Action of covering C_3: step 3

The nodes in this row which are not in the C_3 column, namely the first and last nodes which are in columns C_1 and C_5 respectively, must be removed from these columns. Here we can see this action:
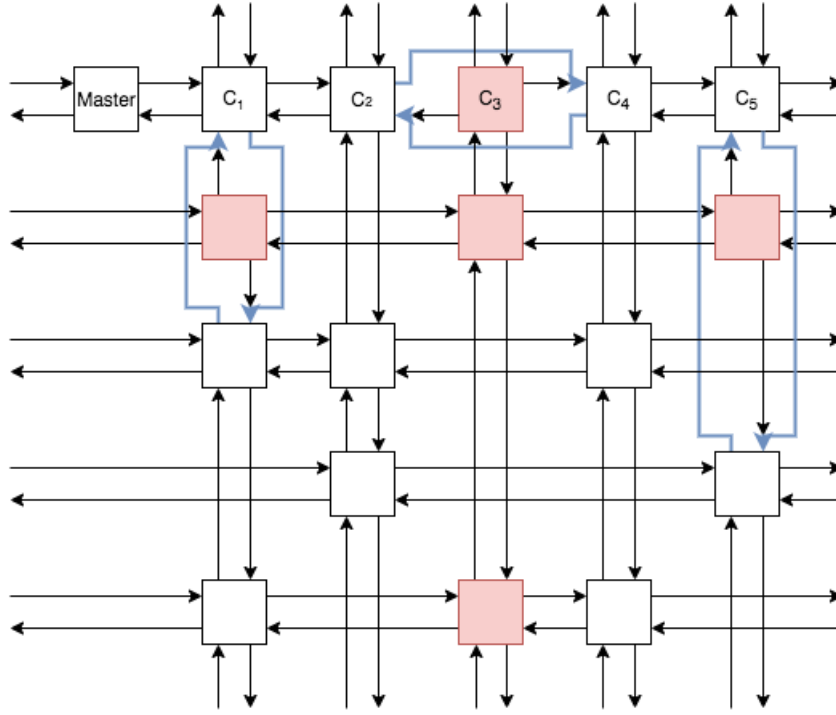
Figure 15: Action of covering C_3: step 4

The nodes above and below these highlighted nodes have had their links redirected, such that they bypass the highlighted nodes entirely. Lastly the method must also cover the other rows in column C_3, of which there is only one. We can see this action below:
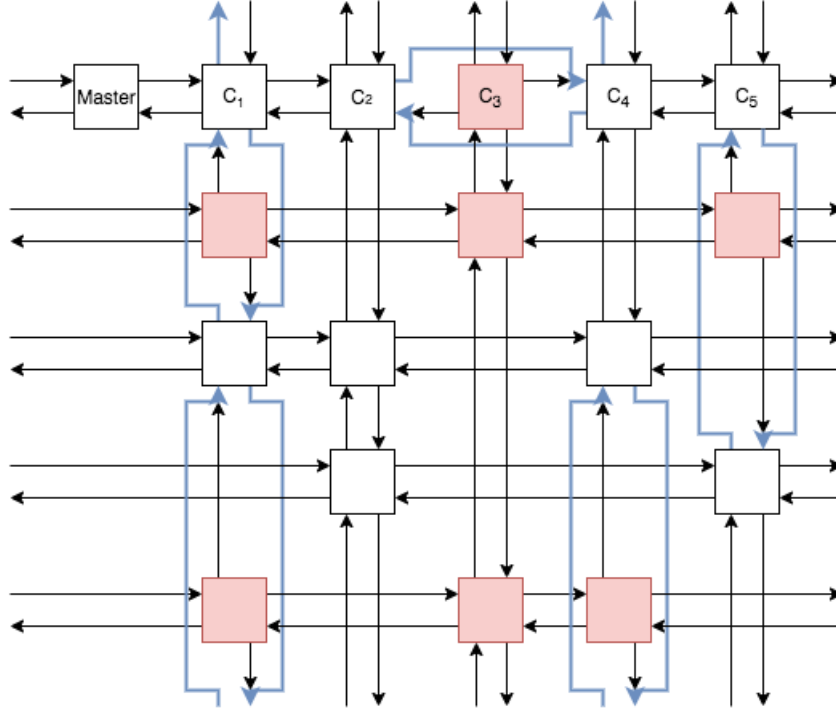
Figure 16: Action of covering C_3: step 5

Now the 'cover_column' method is finished, and algorithm DLX can proceed. Observe how the up and down links of the highlighted (red) nodes have not been altered at all. Although no uncovered nodes are currently pointing to them, they still point to the nodes they were initially adjacent to and as such they 'remember' which position they occupied in the four way linked list. For example the furthest left node in the second row still points to the column header C_1 and to the node below it, the furthest left node in the third row. The innovation behind DLX is using these links to revert the overall list object back to before the 'cover_column' method was called. We can see how this takes place by examining the 'uncover_column' method.

7.1.2   The uncover_column Method

This method, much like the previous one, is very similar to the pseudocode which it is based on:

This were implemented as directly as possible in Python, which we can see here:

Algorithm 4 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 $R[L[c]] \leftarrow c.$

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

Figure 17: Definition of the uncover_column method

This method works to undo the 'cover_column' method, by performing the opposite of each action precisely in the reverse order. As we observed before, the covered nodes still retain their original outgoing pointers and as such we can navigate to these neighbour nodes and restore the links accordingly:

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

To see this method in action let us consider the same example as before, however now we are uncovering the column C_3.
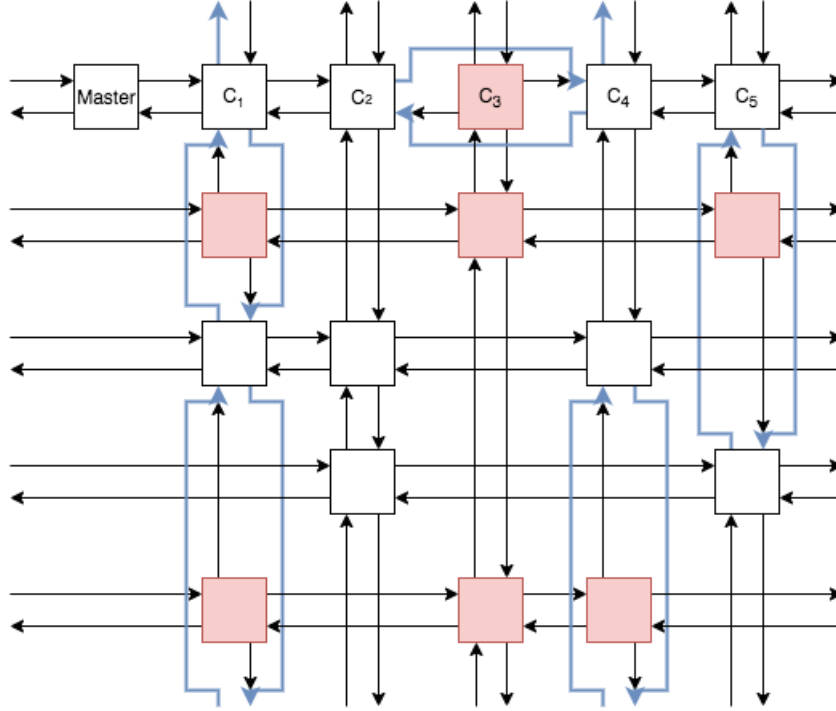
Figure 18: Four Way Linked List with C_3 covered

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. The last action of the 'cover_column' method was to cover the leftmost node of the furthest down row in C_3, the target column. Indeed that method began at the uppermost row, working from right to left through each row, and down through the column. To ensure no information is lost, this method begins at the furthest down row in the column, and works from left to right.
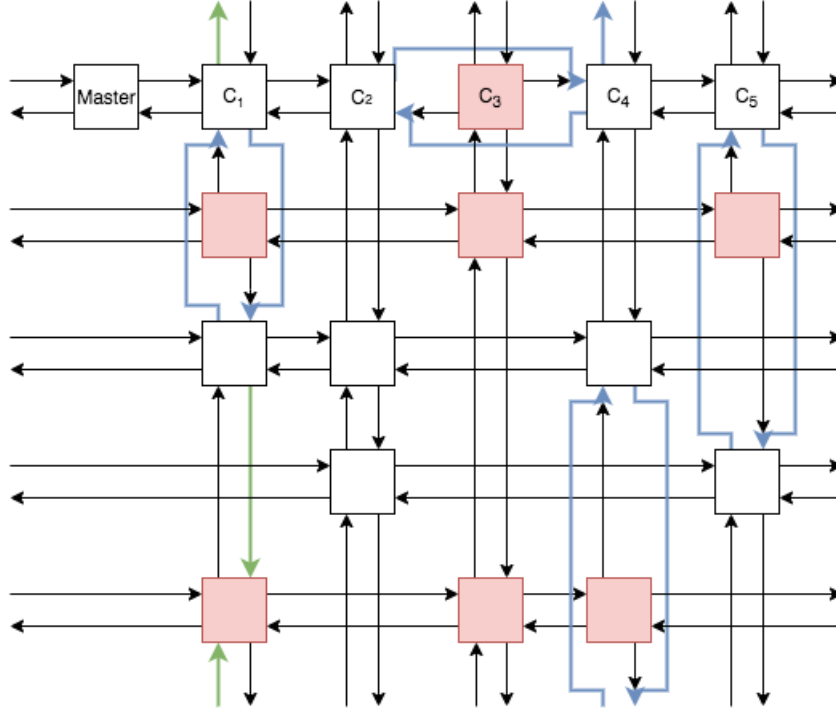
Figure 19: Action of uncovering C$_3$: step 1

As we can see the method has traversed the C$_3$ column upwards one step, wrapping around to the furthest down row from our perspective, and traversed this row one step to the left. This node is then uncovered by the operation:

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

Or in this context:

$$U[D[x]] \leftarrow x, \ D[U[x]] \leftarrow x$$

The method continues with that row by restoring the links of the other nodes, traversing leftwards and wrapping around as follows:
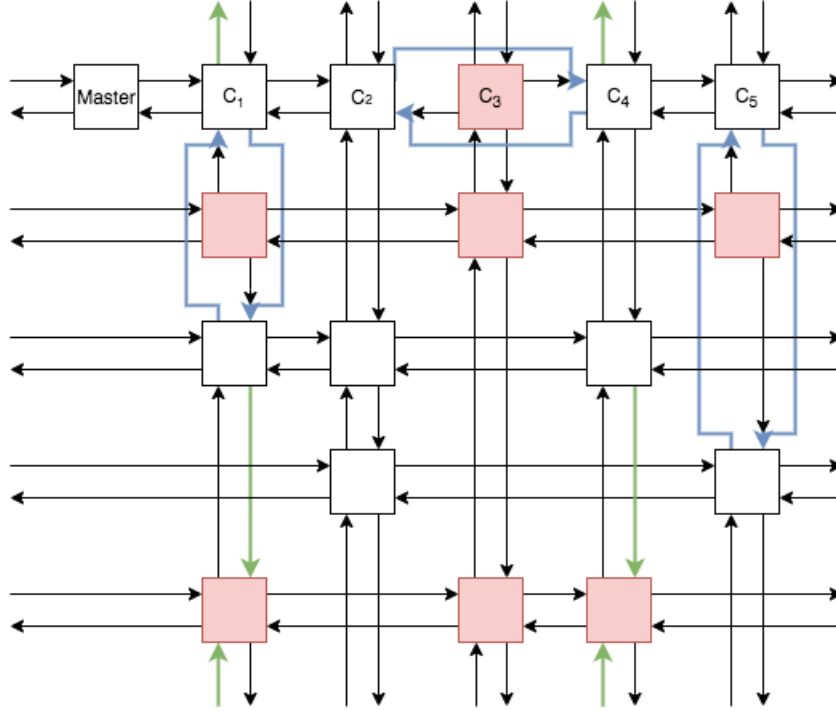
Figure 20: Action of uncovering C$_3$: step 2

After the row is fully restored, and the size attributes of C$_1$ and C$_4$ have been increased accordingly, the method continues traversing the column upwards and restores that row as well
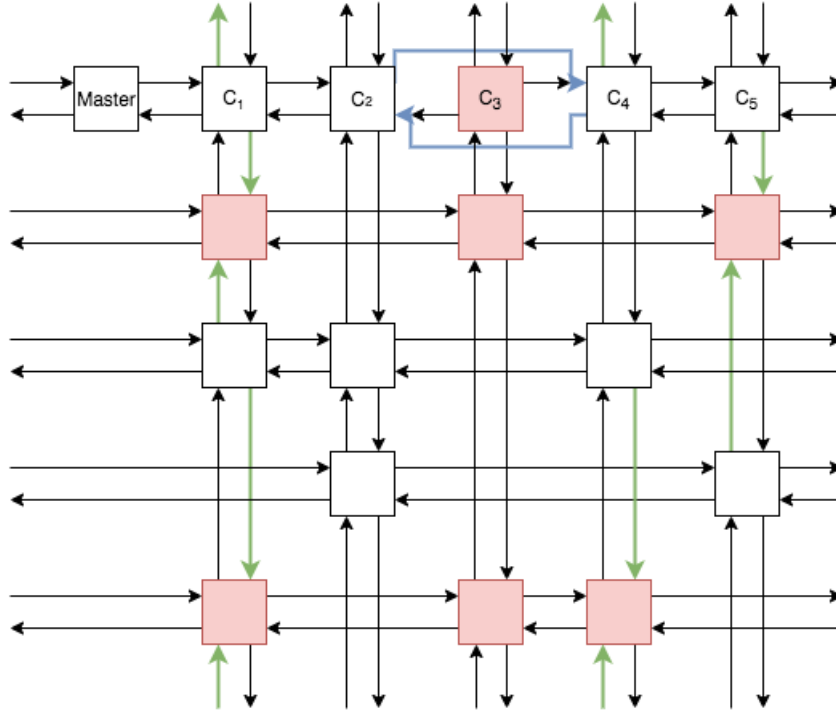
Figure 21: Action of uncovering C_3: step 3

The final operation of the method is to return the C_3 column header to the list:
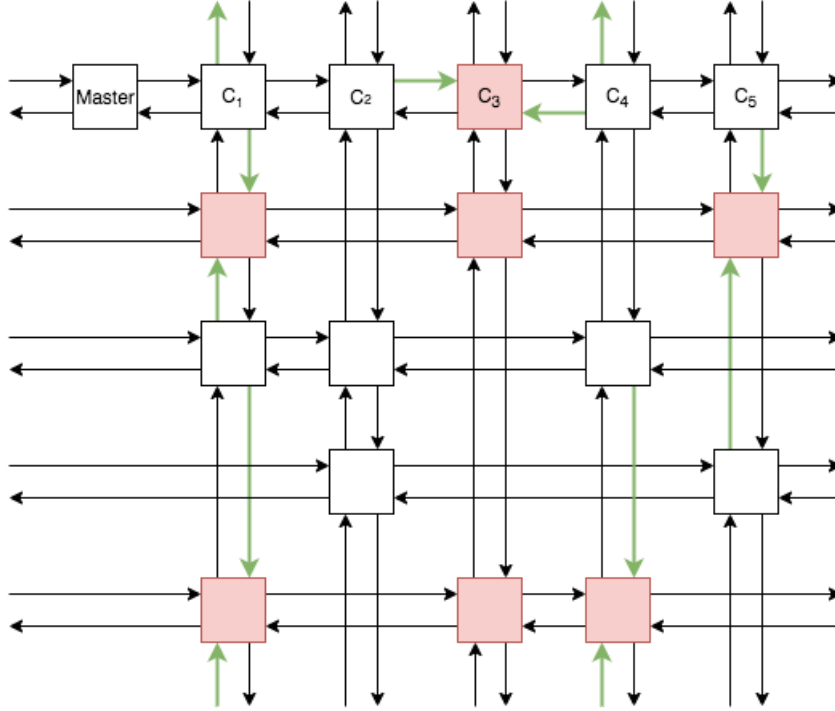
Figure 22: Action of uncovering C_3: step 4

Now the list is fully restored to its original state, before the action of the 'cover_column' method. 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.

7.2 Algorithm DLX

Need to decide on terminology: dead constraint or list stability? Then use consistently. Is there a need to directly reference the line number of the python code? like [Appendix lines 76-79].

Now that the framework is in place to create the four way linked list, as well as convert a matrix into one and update the list object by covering and uncovering columns, we can move on to the full implementation of algorithm DLX. First we must examine the pseudocode provided by Donald Knuth[7]:

This served as a guide for the implementation, which we followed as closely as possible. Choosing columns, as well as covering and uncovering them were abstracted into functions of their own, as well as functions which check if any

44

Algorithm 5 Algorithm DLX

1: If $R[h] = h$, 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.

---

constraints/columns are 'dead' and ones to write solutions and steps to the output files. We can see the implementation in full here *[Appendix]*:

```python
def dlx(self, k, log=True):
    if not self.master_node.right.primary:
        self.print_solution()
        self.file_write_solution("main_output.txt")
        if log:
            self.file_write_solution("log_output.txt")
        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 23: Definition of Algorithm DLX

The following sections will explain, in detail, the actions of determining if a full solution has been found, as well as choosing columns and evaluating the stability of the list.

7.2.1 Determining if a Full Solution has been Found

Wordy sub-heading, needs to be changed?

Due to specifics of the problem space, and of the Python programming language this implementation differs more from the outlined pseudocode than the

aforementioned 'cover_column' and 'uncover_column' methods. The first section of the algorithm performs a check to determine if a full solution has been found, while this may appear different to the check performed by Donald Knuth: 'If $R[h] = h$', it functions almost identically. Due to the nature of the N-Queens problem, which has both primary and secondary constraints, a full solution will likely have several uncovered secondary columns so this '$R[h] = h$' check will not suffice. This issue led to the inclusion of the primary attribute for column objects, as the columns are ordered in such a way that all of the primary columns lie to the left of the secondary ones, we can simply check if the column header to the right of the master header has a primary value of '*False*', if so we return the full solution. In the case that all columns are in fact covered, the primary attribute of the master header is set to '*False*' (and if the master header is the only remaining header, it will be connected to itself on the left and right), so that our check will return true if '$R[h] = h$' is true also. This situation will occur when the algorithm is solving an exact cover problem other than N-Queens, as in that case all the columns will have the primary attribute set to: '*True*'.

After we have checked for a full solution, if none is found we proceed to check if there are any 'dead' constraints (and therefore if a backtrack operation is necessary).

7.2.2   Determining if a Backtrack Operation is Necessary

Too informal tone??

To prevent any nasty errors occurring when no suitable columns can be chosen (i.e. when a primary column has a size of zero), we have defined a method which checks the 'stability' of the list. Here stability is defined as follows: Can a solution be found from the current state of the list? This condition is false if any primary columns have a size $= 0$, to interpret this as humans we can think: If such and such a column has a size of zero but remains uncovered, then that constraint has not yet been satisfied and no further choices of rows can satisfy it! This method can be seen here *[Appendix]*:

```python
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 24: Definition of 'dead_constraint' method

Here 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 is explained in detail in section **8.4.6**.

7.2.3 Choosing the Best Column

Donald Knuth states in his paper[7] that there is no need to track the size of columns unless we would like to limit the branching factor of the algorithm. During our implementation we chose to include the size attribute, as it considerably reduces the execution time of the algorithm. *Citation needed?* As such we choose a column based of the following criteria:

- The column must be a primary one (the primary attribute must be set to '*True*'),

- It must have the lowest size of all available columns.

The implementation is as follows *[Appendix]*:

```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 25: Definition of 'find_best_column' method

Here we simply iterate across the header list, checking the condition for each column and storing the current best header in the 'best_header' variable, which is return after the header list has been fully traversed.

7.2.4 Branching and Storing Partial Solutions

Messy subsection? Needs to be split into two separate sections?
Now that we have chosen the best possible column, following Donald Knuth's explanation, we cover this column and iterate downward through it. For each row in this column we add that row to the partial solution, using the 'set_solution_k' method, and iterate through that row covering each node's column, then we make a recursive call to algorithm DLX.

For each node in the selected row, we must cover that node's column as the chosen row satisfies those constraints too, we cannot satisfy and constraint more than once. This directly follows the pseudocode given by Donald Knuth.

The recursive call happens once for each row in the originally chosen column, by this I mean we branch the algorithm for each row and attempt to find full solutions of the now reduced four way linked list. For example in the diagrams seen earlier where we covered column C_3, the algorithm would branch two times making one new call for each row in the column.
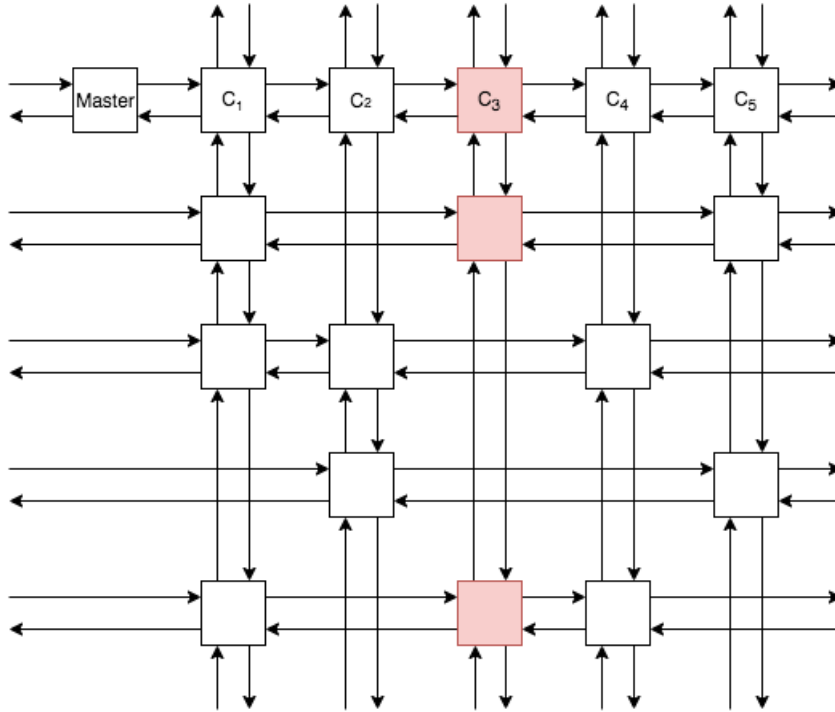

Figure 26: Four Way Linked List with C_3 highlighted

These recursive calls are kept track of by the depth variable k, which is initially zero. In this example when the algorithm covers C_3 the depth is: $k = 0$. For each row we make a recursive call with $k = 1$, and use this k variable to index the partial solutions. To see how this works we will first consider the function 'set_solution_k' [$Appendix$]:

```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 27: Definition of the 'set_solution_k' method

The solution list is an attribute of the four way linked list, and it is initially empty. It 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 our example we would like to store the first node of the C_3 column as a partial solution, 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 $k = 3$ and find that no solution is possible therefore backtrack to $k = 2$, when we reach $k = 3$ again we will simply overwrite the previous node stored in the kth index of the solution list.

Due to pythons native list structure, which 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 the above code, first try to set the kth index equal to the node we would like to store, 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).

7.2.5   Restoring the Four Way Linked List

Here we arrive at arguably the most important step of algorithm DLX, where the dance itself happens. We have already defined the 'uncover_column' method, and now it is time to bring it into action.

It is difficult to visualise the order in which the compiler will execute the instructions given in DLX, however we do not need to understand this concept fully. After we have added a node from our chosen row to the partial solution list, and made a recursive call to DLX, a number of instructions will be executed.

50

What is important right now, is that we must continue with the other rows in the originally chosen column, C_3, and as such we must undo the previous action of covering the first row.

To do this, we recover the chosen node from the first row by accessing the kth index of the solution list, then we iterate leftward through this row uncovering each node's column (it is important we iterate leftward here, as we iterated rightward when covering the row and we must ensure a proper inverse operation as discussed in section 8.3.2)

Now the list object is restored (aside from the covered C_3 column which we are still iterating through) and we can move on to the next row. Once all of the rows are exhausted, we exit the while loop and finally uncover the chosen column, allowing future executions of the DLX algorithm to proceed.

8 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 [4] that is usually used as a heat-map.
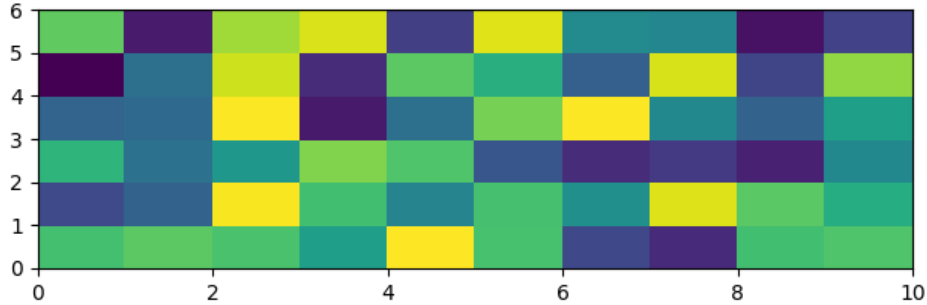

Figure 28: Demonstration of pcolor figure from official documentation in [4]

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 29. 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.

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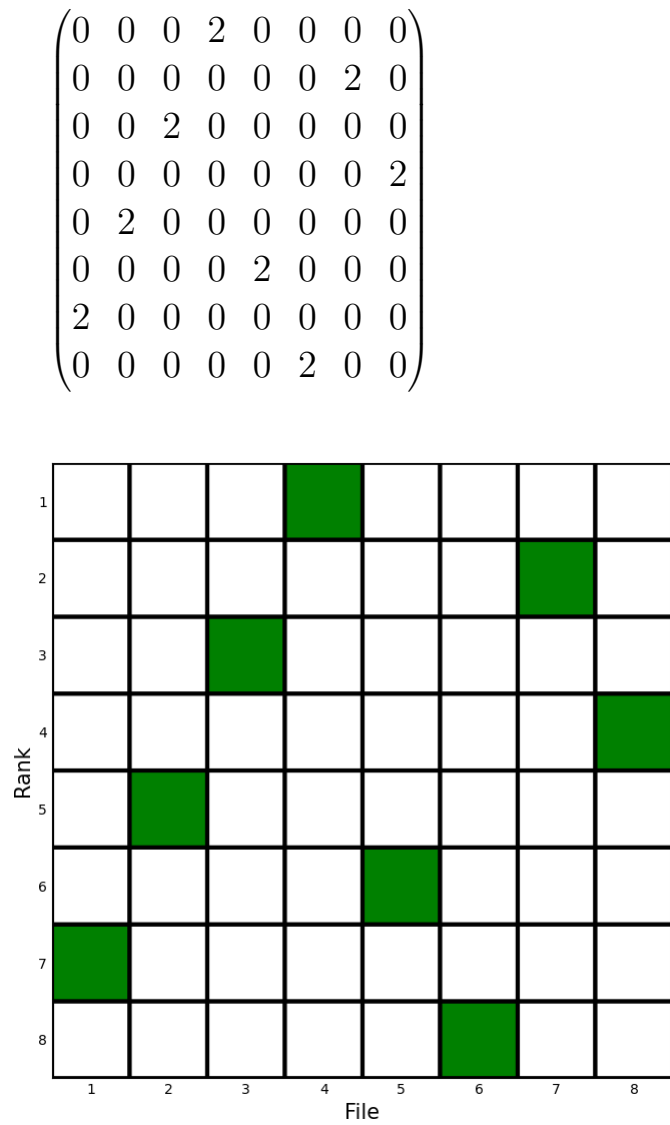

Figure 29: pcolor plot of an 8 X 8 solution

A demonstration of the visuals can be seen by clicking the above image, which links to an MP4 video hosted on github.

8.0.1 Bookkeeping

In this section we will explain the methods used to create and update the output file throughout the execution of the program. There are two distinct output files created by the program: 'main_output.txt' and 'log_output.txt'. The former records each distinct solution found by the algorithm, while the latter also records each step of algorithm DLX including all necessary backtracking actions. The primary output file for users is 'main_output.txt' which is readable to us humans, while the 'log_output.txt' file is more extensive and used as raw data for the visual implementation. It should be noted here that the extensive log file: 'log_output.txt' is entirely optional, and the user can specify to not have it maintained throughout the program during start-up.

There are a number of methods for specific file management, all of which I will briefly discuss here. When the four way linked list object is initialised it calls one initial method for each of the output files: 'file_write_initial' and 'file_write_initial_log'. These methods simply create their respective files, or override them if they already exist, and write a brief introduction for the user.

Should the user be solving the N Queens problem, the method 'file_write_n_queen' is called, which simply writes a specific description of the problem into the 'main_output.txt' file, also including the value of N chosen.

Regardless of the problem being solved, the method 'file_write_exact_cover' will be called which writes the exact cover matrix into the 'main_output.txt' file.

Each time a full solution is found, this is recorded in both output files by the 'file_write_solution' method, which also tracks the 'total_solutions' attribute of the four way linked list class. This method prints the solution number, followed by all of the rows in that solution, which are extracted from the 'solution_list' attribute and ordered correctly by the 'find_furthest_left' method.

For the extensive log file, if the user has specified to create it, each time a new row is chosen during algorithm DLX it is logged using the 'file_write_log_row' method, which also uses the 'find_furthest_left' method to order the output correctly. This file writing method has a Boolean argument: 'backtrack', when a dead constraint is found this method is called with that argument set to true, and records the backtracking action as opposed to the row choosing action.

These methods come together to create the output files for the user, if you are looking for a more in-depth explanation than this, please examine the methods themselves in *[Appendix]* where the code can be seen and is described with comments.

9   Collaboration Outline

This project was undertaken by two students, Seán Monahan and Oisín Hodgins. Main implementation of the algorithm was Oisín's main focus while Seán handled the visuals and theory of the exact cover problem.

10   Conclusion

References

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

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

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

[4] pcolor documentation from matplotlib. available here.

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

[6] Kohei Noshita Hirosi Hitotumatu. *Information Processing Letters 8.* 1979.

[7] Donald Knuth. *Millennial Perspectives in Computer Science.* Palgrave Macmillan, 2000.

11 Appendices

This page not for final script

To do list:

visuals code

Format of manuscript: (From Blackboard)
•Cover page
•Declaration page
•Introduction
•Main body (structured into sections as per topic requires).
•Collaboration Outline
•Conclusion
•Bibliography
•Appendices (if applicable)
This page just for reference