Knuth's Algorithm X

Zitong Xiao

xiao.zit@northeastern.edu

Emmanuel Botwe

botwe.e@northeastern.edu

Mandadi Ayush Reddy

reddy.man@northeastern.edu

Salman Awaise

awaise.s@northeastern.edu

Rohan Tanwar

tanwar.ro@northeastern.edu

December 10, 2024

Abstract

Knuth's Algorithm X is a powerful backtracking algorithm designed for solving exact cover problems. One of its most efficient implementations uses the Dancing Links (DLX) data structure, which allows dynamic manipulation of a sparse binary matrix. This paper explores the application of algorithm X to solving Hyper Sudoku puzzles, a challenging variant of the classic Sudoku game. The theoretical basis, history and practical implementation of the algorithm are introduced in detail. In addition, we provide insights into the visualization, challenges and efficiency of the solution.

1 Introduction

Knuth's algorithm X is a general and elegant algorithm for solving precise coverage problems. Exact coverage problems occur in many computing scenarios, from tiling problems to constraint satisfaction tasks. This algorithm, combined with innovative dance link technology, has attracted attention for its efficiency in solving large and complex problems.[1]

This paper focuses on the application of algorithm X to the superseries unique puzzle. Hyper Sudoku is a variant of Sudoku that incorporates additional constraints that add complexity, making the problem ideal for testing algorithm X. In general, the exact cover algorithm is applied to the decryption of Sudoku problems, while the DLX data structure is used to realize the Exact Cover, and the toroidal doubly linked list is the focus of this paper.

1.1 History of Algorithm X

Algorithm X was introduced by Donald E. Knuth as a method to solve the exact cover problem, a generalization of problems that require selecting subsets to cover a universal set without overlap. In his seminal paper, Knuth also proposed the Dancing Links data structure, which allows efficient manipulation of the problem's constraints during the backtracking process.[4] Over the years, Algorithm X has been applied to a wide variety of problems, including tiling puzzles, Sudoku, and combinatorial design, demonstrating its versatility and efficiency.

2 Sudoku Game

Sudoku is a popular logic-based puzzle that requires filling a 9×9 grid with digits from 1 to 9. The puzzle enforces specific constraints: each row, each column, and each of the nine 3×3 sub-grids must contain every digit exactly once. [3]

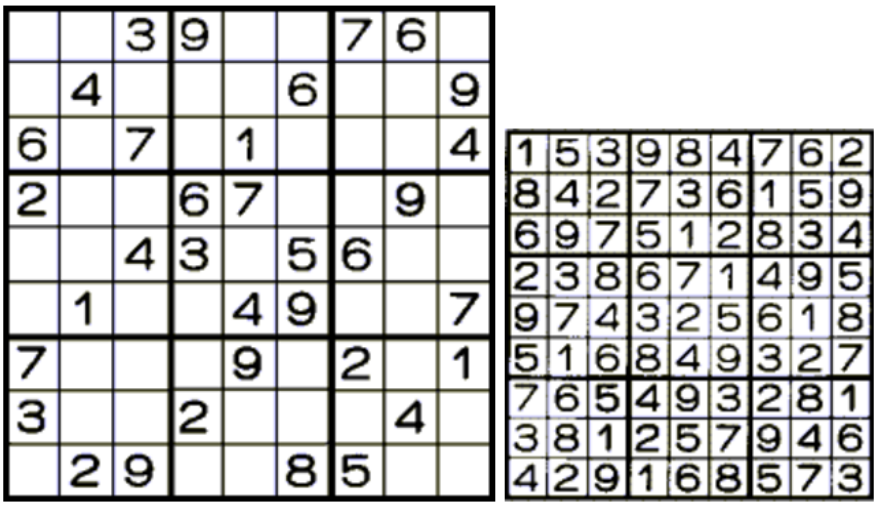


Figure 1: An example of a Sudoku game. Source: Sudoku Paper

2.1 Hyper Sudoku

Hyper Sudoku differs from normal Sudoku by including four additional overlapping regions within the standard 9x9 grid. These regions, along with rows, columns, and 3x3 blocks, must contain unique numbers from 1 to 9. This added constraint increases the puzzle's complexity and difficulty compared to normal Sudoku. [3]

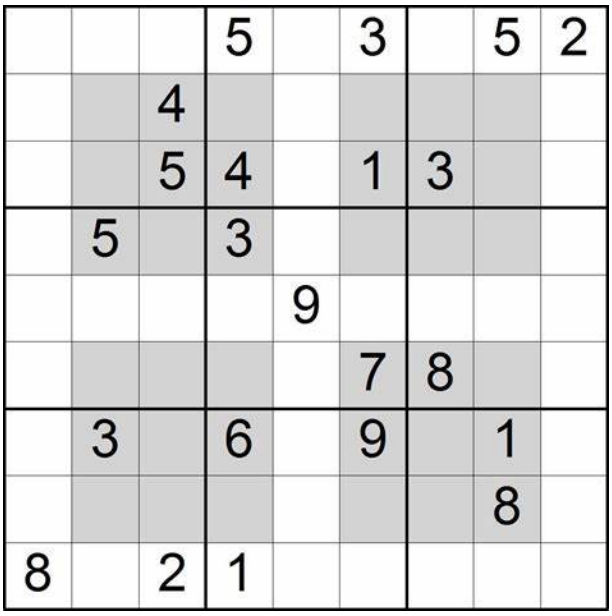


Figure 2: An example of Hyper Sudoku. Source: Sudoku Paper

3 Exact Cover

Exact coverage is a mathematical problem that involves selecting subsets from a set so that each element in a given range of elements is covered only once, with no overlap. It is very effective in solving problems like Sudoku compared to conventional brute force retrieval or DFS algorithms. Give a simple example to understand the principle:

Problem Setup

We are given the universal set $X = \{1, 2, 3, 4, 5, 6, 7\}$ and the following collection of subsets:

$$S_1 = \{3, 5, 6\},$$
$$S_2 = \{1, 4, 7\},$$
$$S_3 = \{2, 3, 6\},$$
$$S_4 = \{1, 4\},$$
$$S_5 = \{2, 7\},$$
$$S_6 = \{4, 5, 7\}.$$

The goal is to select a subset of these rows such that:

- Every element in X is covered exactly once.
- No element in X is covered more than once.

Matrix Representation

The exact cover describes problems in which a matrix of 0's and 1's. chu2006sudoku So the above example can be converted to the following matrix:

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

where:

- Rows correspond to subsets $S_1, S_2, \ldots, S_6$.
- Columns correspond to elements of $X = \{1, 2, 3, 4, 5, 6, 7\}$.
- A 1 in the matrix indicates that a subset contains the corresponding element.

Solution

From the resulting binary matrix, the goal is to select rows so that each column of the matrix is covered exactly once. (i.e., each column is covered by only one "1")

Step 1: Select $S_1 = \{3, 5, 6\}$ - row 1

- **Row 1 covers columns 3, 5, and 6.**

Step 2: Remove Covered Rows and Columns
Remove:

- **Columns 3, 5, and 6** from the matrix (since they are now covered).

- All rows that overlap with these columns (rows 3 and 6).

The updated matrix becomes:

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

At this time the matrix is formed by:

row 2

row 4

row 5

Step 3: Select $S_2 = \{1, 4, 7\}$ - row 2

- **Row 2 covers column 1, 4 and 7 **

* At this time, the disadvantages of simply using the exact cover are revealed. Since this algorithm continues to dynamically delete rows and columns, and may need to intervene in backtracking operation in the future, the representation of each position is very vague, and it is difficult to accurately and uniformly represent the position in the matrix. Therefore, DLX was subsequently introduced to achieve the exact cover, making the index more intuitive and convenient.

Step 4: Remove Covered Rows and Columns

Remove:

- **Columns 1, 4, and 7** from the matrix (since they are now covered).
- All rows that overlap with these columns (rows 4 and 5).

At this time, matrix has been deleted, and for the row 2 we selected, there is still a ”0” in column 2 that is not filled by ”1”, so the selection of row 2 is incorrect. This is what we need to backtrack to step2 and choose aother row.

Step 5: Backtrack to step 2 and select $S_4 = \{1, 4\}$ - row 4 instead

- **Row 4 covers column 1 and 4 **

Step 6: Remove Covered Rows and Columns
Remove:

- **Columns 1 and 4** from the matrix (since they are now covered).
- All rows that overlap with these columns (rows 2 and 4).

The updated matrix becomes:

$$\begin{bmatrix} 1 & 1 \end{bmatrix}$$

At this time the matrix is formed by:
row 5

Since all columns in the final matrix are covered by "1", when row 5 is selected, all columns can be covered by "1", and the goal is achieved.

Step 7: Select $S_6 = \{2, 7\}$ - row 5
The exact cover solution is:

$$S_1 = \{3, 5, 6\},$$
$$S_4 = \{1, 4\},$$
$$S_5 = \{2, 7\}$$

4 The Dancing Links

Dancing Links is an efficient algorithmic technique used to solve the Exact Cover problem, particularly as part of Donald Knuth's Algorithm X. It represents the binary matrix as a doubly linked list, where rows and columns correspond to nodes connected via horizontal and vertical

pointers, respectively. As stated earlier in this paper, this structure enables fast traversal and manipulation during the backtracking process.

The term *Dancing Links* arises from the dynamic way pointers in the data structure are updated during the algorithm's execution, creating a "dance-like" movement as elements are removed and reinserted. This approach allows for efficient undoing of operations, which is essential in backtracking algorithms. The primary advantage of Dancing Links lies in its ability to maintain the structural integrity of the matrix while dynamically removing and restoring rows and columns during recursive exploration. [4]

4.1 The Dancing Links Process and Features

Process:

1. **Matrix Representation:** The binary matrix is converted into a circular, doubly linked list, where each node represents a `1` and columns have header nodes for easy traversal.
2. **Column Selection:** Choose a column, often the one with the fewest `1`s, to minimize branching.
3. **Row Removal:** Remove all rows containing a `1` in the selected column and temporarily adjust the pointers.
4. **Backtracking:** If no solution is found, restore removed rows and columns by undoing the pointer changes.

Features:

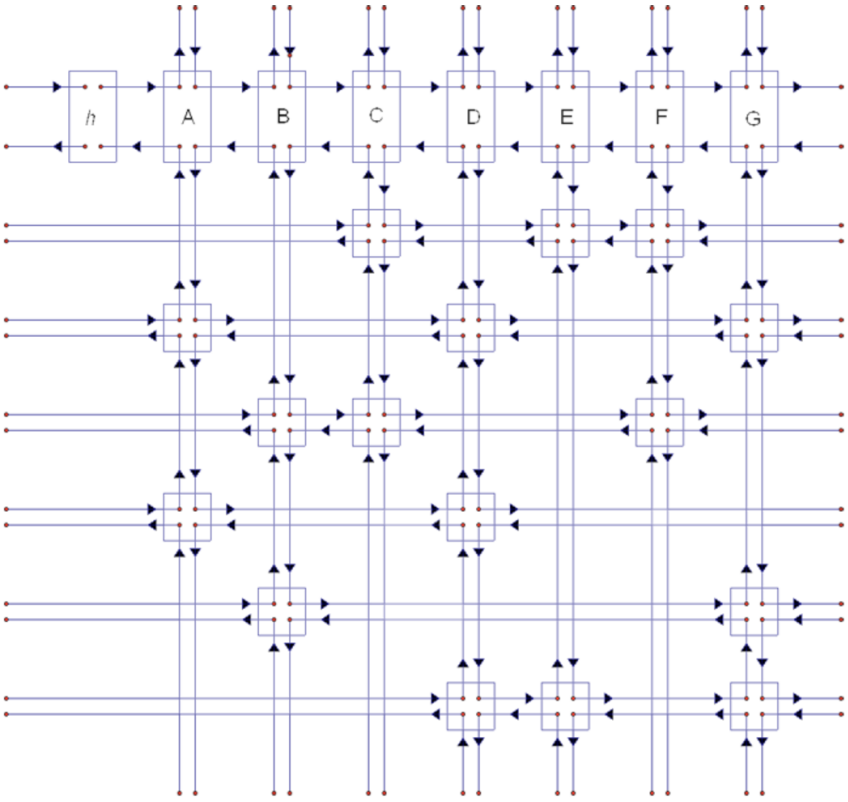


Figure 3: An example of a doubly linked list of dancing links. Source: Sudoku Paper.

- **Bidirectional Circular Linked List:** Each node connects to its neighbors via four pointers (up, down, left, right), enabling efficient traversal and updates.
- **Header Node:** The head node (`h`) serves as the entry point, linking to column headers (`A, B, C, ...`), which organize the matrix structure.
- **Dynamic Removal and Restoration:** Nodes can be dynamically removed or restored by adjusting pointers, making backtracking efficient without reconstructing the structure.

4.2 Psudocode and Algorithm expansion.

The Dancing Links Algorithm can be implement based the Psudocode like this:

Given the ColumnNode h, the searching algorithm is then simplified to:

```
if( h.getRight() == h ) {
    printSolution();
    return;
}
else {
    ColumnNode column = chooseNextColumn();
    cover(column);
    for( Node row = column.getDown() ; rowNode != column ; rowNode = rowNode.getDown() ) {
        solutions.add( rowNode );
        for( Node rightNode = row.getRight() ; otherNode != row ; rightNode = rightNode.getRight() )
            cover( rightNode );
        Search( k+1);
        solutions.remove( rowNode );
        column = rowNode.getColumn();
        for( Node leftNode = rowNode.getLeft() ; leftNode != row ; leftNode = leftNode.getLeft() )
            uncover( leftNode );
    }
    uncover( column );
}
```

Figure 4: Dancing Links Algorithm Psudocode.

Expansion:

- **Solution Found:** If `h.getRight() == h`, then all columns have been covered, and a valid solution to the Exact Cover problem has been found.
- **No Solution or Backtracking:** If `h.getRight() != h` after attempting to remove all elements, it indicates that no solution exists for the current path. The algorithm must backtrack (unstack the recursive calls) or output ”no solution” if all possibilities are exhausted.

4.3 Problem Review

Take an example fro the exact cover section, now demonstrate the What is the expressability of using DLX to handle the exact cover problems.

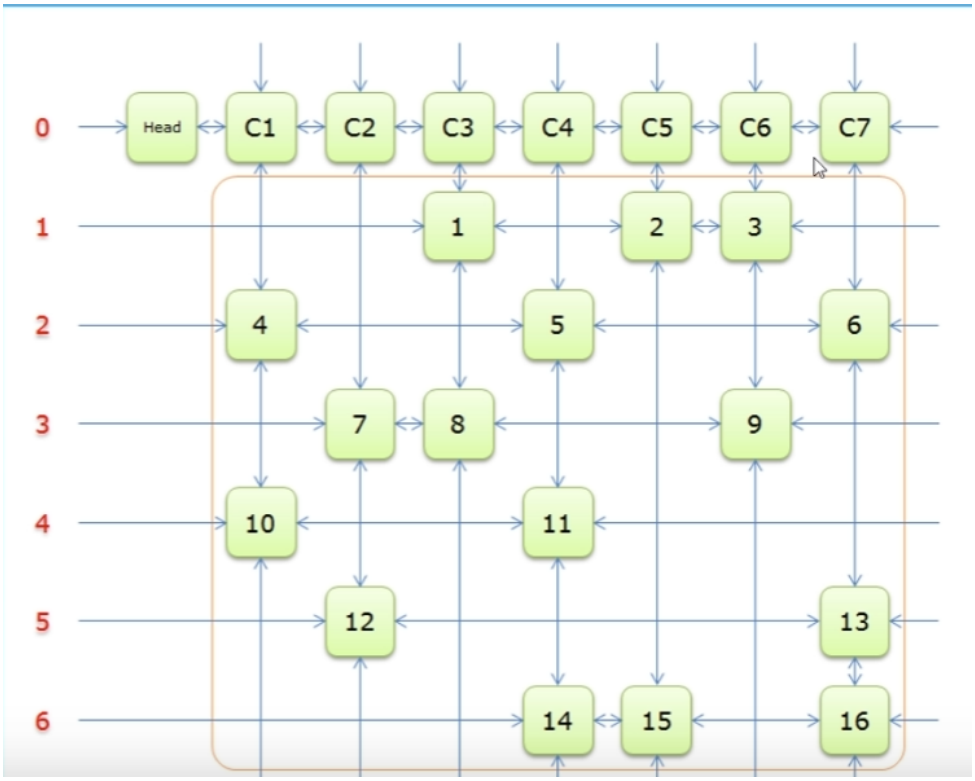
Call Back:

The binary matrix:

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

Build the doubly linked list based on it:

In this figure:



- Head is the head node of the doubly linked list.
- C1, C2... represent the head node of each column.
- Start with `h.getRight() == C1`

DXL will recursively select rows until it finds the solution, the answer to which is already known in the previous article, that is, selecting row1, row4, and row5 can cover all columns. We will now demonstrate the case where the selection is wrong and backtrack is required.

Step 1: `h.getRight() == C1` - C1 column

- **C1 contain ”1” for row 2 and row 4, we pick row 2 to start covering.**
- **Row 2 covers columns 1, 4, and 7.**

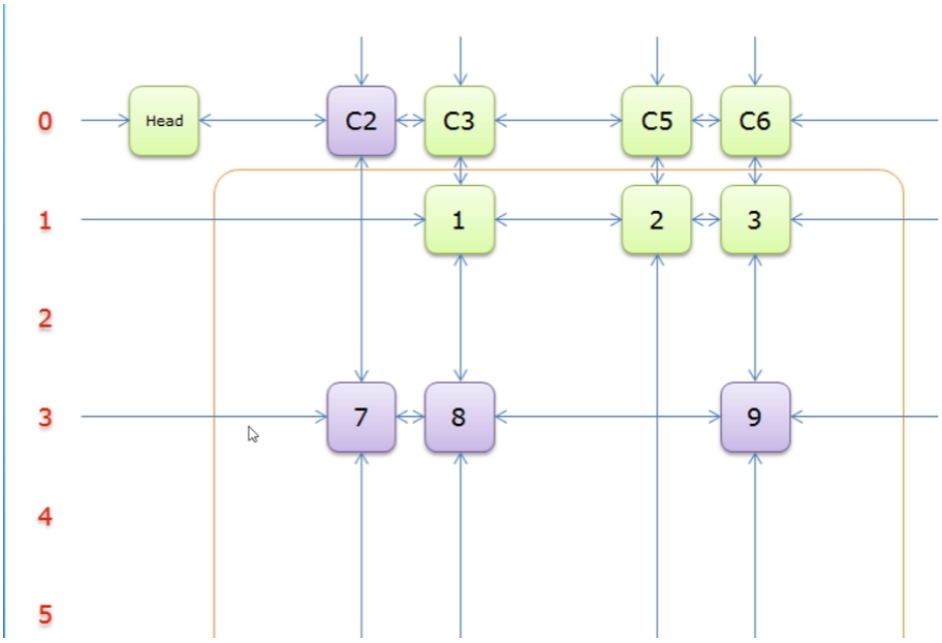
Answer stack: row 2

Step 2: Remove Covered Rows and Columns

Remove:

- **Columns 1, 4, and 7** from the matrix (since they are now covered).
- All rows that overlap with these columns (rows 4, 5 and 6).

The doubly linked list becomes:



Step 3: `h.getRight() == C2` - C2 column

- **C2 contain ”1” for row 3 only, we pick row 3 to start covering.**
- **Row 3 covers columns 2, 3, and 6.**

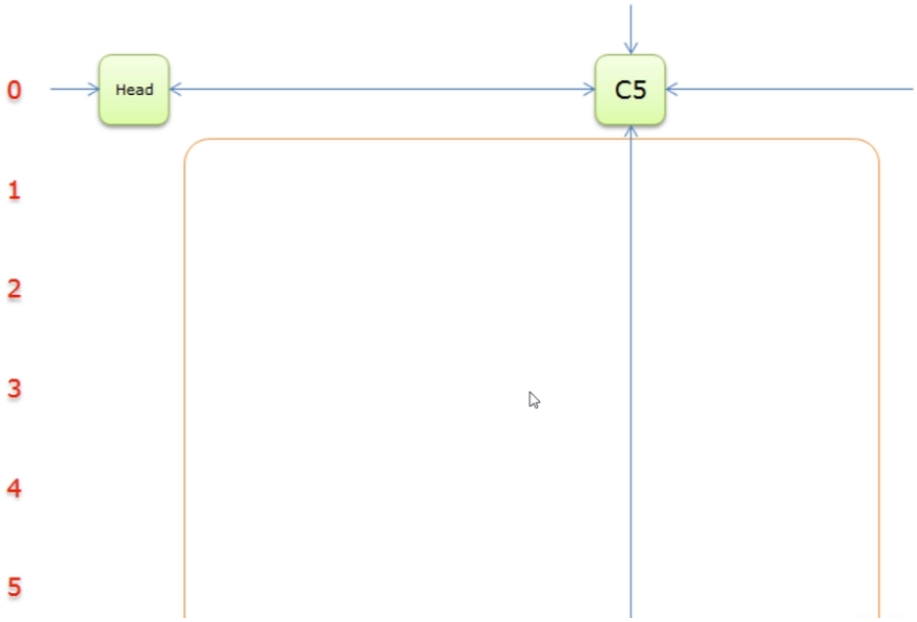
Answer stack: row 2, row 3

Step 4: Remove Covered Rows and Columns

Remove:

- **Columns 2, 3, and 6** from the matrix (since they are now covered).
- All rows that overlap with these columns (rows 3 and 1).

The doubly linked list becomes:



Obviously, the matrix has been subtracted at this point.
However, `h.getRight() == C5` instead of `h`, which means that row choice is wrong. At this point, the backtracking mechanism triggers and you should go back to the previous step and re-select the row. However, notice that the resulting C2 from step 3 corresponds to row 3 only, so DLX will go back to step 1 and select again. This mechanism keeps recursing until it reaches a solution.

For Step 2, if row 4 is selected, then row 5 is selected, and finally row 1 is selected, then the solution is reached.

5 Apply Algorithm X on Hyper Sudoku

In general, apply Algorithm X on Hyper Sudoku is divided into three parts:

- Convert Hyper Sudoku into linked list.
- Apply DLX to find the solution to the problem recursively.
- Convert linked list back to Hyper Sudoku.

The hardest part is to build the columns and rows of the linked list according to the constraints and number decision of each grid.

5.1 Hyper Sudoku to Linked List - encoding and decoding

5.1.1 Transforming Constraints into Columns of the linked list

1. **Each cell can only contain one number:**
    - A total of 81 cells, mapped to 81 columns.
    - Column 1 represents cell (1,1) containing a number.
    - Column 2 represents cell (1,2) containing a number.
    - ...
    - Column 81 represents cell (9,9) containing a number.
2. **Each row can only contain each number once:**
    - A total of 9 rows, with 9 numbers each, mapped to 81 columns.
    - Column 82 represents row 1 containing number 1.
    - Column 83 represents row 1 containing number 2.
    - ...
    - Column 162 represents row 9 containing number 9.
3. **Each column can only contain each number once:**

- A total of 9 columns, with 9 numbers each, mapped to 81 columns.
- Column 163 represents column 1 containing number 1.
- Column 164 represents column 1 containing number 2.
- ...
- Column 243 represents column 9 containing number 9.

4. **Each sub-grid can only contain each number once:**

   - A total of 9 sub-grids, with 9 numbers each, mapped to 81 columns.
   - Column 244 represents sub-grid 1 containing number 1.
   - Column 245 represents sub-grid 1 containing number 2.
   - ...
   - Column 324 represents sub-grid 9 containing number 9.

5. **Each highlighted region can only contain each number once(Hyper Sudoku Only):**

   - A total of 4 highlighted regions, with 9 numbers each, mapped to 36 columns.
   - Column 325 represents highlighted region 1 containing number 1.
   - Column 326 represents highlighted region 1 containing number 2.
   - ...
   - Column 360 represents highlighted region 4 containing number 9.

5.1.2 Transforming Decisions into Rows of the linked list

1. **For a cell filled with a number:**

   - Example: Filling cell (1,3) with number 8.
   - This corresponds to the following constraints:
     - Cell (1,3) contains a number.
     - Row 1 contains number 8.
     - Column 3 contains number 8.
     - Sub-grid 1 contains number 8.

- If (1,3) is in a highlighted region, the highlighted region contains number 8.

- In the linked list, this corresponds to 4 or 5 columns being 1, requiring 1 row to represent this decision.

2. **For an empty cell:**

   - Enumerate all possible fillings for this cell:
     - For (1,1), filling number 1 corresponds to 4 or 5 columns being 1.
     - For (1,1), filling number 2 corresponds to 4 or 5 columns being 1.
     - ...
     - For (1,1), filling number 9 corresponds to 4 or 5 columns being 1.
   - In the worst case, all 81 cells are empty, and each cell has 9 possible fillings, resulting in $81 \times 9 = 729$ rows.

5.1.3 Dancing Links

1. **Relation Between Rows and Columns:**

   - Each row corresponds to 4 or 5 constraints, matching a row in the linked list.
   - Each row adds 4 or 5 nodes, resulting in $729 \times 4 = 2916$ nodes.

2. **Covering and Uncovering Columns:**

   - For covering a column:
     - Select a column and remove all rows that intersect with it.
   - For uncovering a column:
     - During backtracking, restore the removed rows and columns.

3. **Dynamic Management of Rows and Columns:**

   - Utilize the doubly-linked list in the dancing links structure for efficient row and column management.
   - Ensure efficient deletion and restoration during traversal.

Summary: A 729-row by 360-column linked list is dynamically managed using the dancing links structure to efficiently solve the exact cover problem for Hyper Sudoku.

6 Encoding and Decoding

Based on the constrains and number selection,
Given:

- A Hyper Sudoku grid S with cells indexed by (i, j), where:
  - i is the row index ($0 \leq i \leq 8$),
  - j is the column index ($0 \leq j \leq 8$).
- A linked list representation L of rows and columns (r, c) used to solve S via the Dancing Links algorithm.

6.1 Transforming Sudoku Grid to Linked List

6.1.1 Position (i, j) in Sudoku to Row r in Linked List

For any cell (i, j) in the Sudoku grid and number $k \in [1, 9]$, the row r in the linked list is given by:

$$r = i \cdot 9 \cdot 9 + j \cdot 9 + (k - 1)$$

where:

- i: Row index ($0 \leq i \leq 8$),
- j: Column index ($0 \leq j \leq 8$),
- k: Number to be placed in the cell ($1 \leq k \leq 9$).

6.1.2 Row r in Linked List to Columns c_1, c_2, c_3, c_4, c_5

Each row r satisfies multiple constraints, mapped to the following columns in the linked list:

1. **Cell Constraint (81 columns):**

$$c_1 = i \cdot 9 + j$$

Ensures that each cell in S contains only one number.

2. **Row Constraint (81 columns):**

$$c_2 = 81 + i \cdot 9 + (k-1)$$

Ensures that each number k appears exactly once in row i.

3. **Column Constraint (81 columns):**

$$c_3 = 162 + j \cdot 9 + (k-1)$$

Ensures that each number k appears exactly once in column j.

4. **Sub-grid Constraint (81 columns):**

$$c_4 = 243 + \left\lfloor \frac{i}{3} \right\rfloor \cdot 3 + \left\lfloor \frac{j}{3} \right\rfloor \cdot 9 + (k-1)$$

Ensures that each number k appears exactly once in its 3×3 sub-grid.

5. **Highlighted Region Constraint (36 columns):** If (i, j) is in a highlighted region h, then:

$$c_5 = 324 + h \cdot 9 + (k-1)$$

Ensures that each number k appears exactly once in its highlighted region.

6.2 Transforming Linked List Solution to Sudoku Grid

After the Dancing Links algorithm finds a solution (a set of selected rows $\mathcal{R}$), decode each $r \in \mathcal{R}$ back into the Sudoku grid position (i, j) and value k as follows:

1. **Row i:**

$$i = \left\lfloor \frac{r}{81} \right\rfloor$$

2. **Column j:**

$$j = \left\lfloor \frac{r \mod 81}{9} \right\rfloor$$

3. **Value k:**

$$k = (r \mod 9) + 1$$

Finally, assign k to position (i, j) in the Sudoku grid:

$$S[i][j] = k$$

6.3 Example: Row and Column Mapping for a Hyper Sudoku

Assume there is a Hyper Sudoku S with $S(0, 0) = 5$, and $S(7, 9) = 0$ (Empty Cell)

6.3.1 Decision: Place 5 in Cell $(0, 0)$

For the decision to place 5 in cell $(0, 0)$:

- The row index r in the binary matrix is calculated as:

$$r = i \cdot 9 \cdot 9 + j \cdot 9 + (k - 1)$$

  Substituting $i = 0, j = 0, k = 5$:

$$r = 0 \cdot 81 + 0 \cdot 9 + (5 - 1) = 4$$

  Thus, this decision is represented by **Row 4**.

- The columns in the binary matrix where this row has 1s correspond to the constraints

satisfied by this decision:

$$\begin{aligned}
\text{Cell constraint: } c_1 &= i \cdot 9 + j = 0 \cdot 9 + 0 = 0, \\
\text{Row constraint: } c_2 &= 81 + i \cdot 9 + (k-1) = 81 + 0 \cdot 9 + (5-1) = 85, \\
\text{Column constraint: } c_3 &= 162 + j \cdot 9 + (k-1) = 162 + 0 \cdot 9 + (5-1) = 166, \\
\text{Sub-grid constraint: } c_4 &= 243 + \left\lfloor \frac{i}{3} \right\rfloor \cdot 3 + \left\lfloor \frac{j}{3} \right\rfloor \cdot 9 + (k-1) \\
&= 243 + 0 \cdot 3 + 0 \cdot 9 + (5-1) = 247, \\
\text{Highlighted region constraint: } c_5 &= 324 + h \cdot 9 + (k-1) = 324 + 0 \cdot 9 + (5-1) = 328.
\end{aligned}$$

6.3.2 Binary Matrix Row Representation

In the binary matrix, **Row 4** is represented as:

$$[1, 0, 0, \ldots, 1, \ldots, 1, \ldots, 1, \ldots, 1, \ldots, 0]$$

Where:

- The 1s appear at column indices $\{0, 85, 166, 247, 328\}$,
- All other entries in the row are 0.

6.3.3 Linked List Representation

In the linked list representation:

- Each 1 in the binary matrix row corresponds to a node in the linked list.

- The nodes for this row are:

Node 1: Column 0,

Node 2: Column 85,

Node 3: Column 166,

Node 4: Column 247,

Node 5: Column 328.

- These nodes are doubly linked to their neighbors, allowing for efficient traversal during the Dancing Links algorithm.

6.3.4 Decision: Place 0 in Cell $(7, 9)$

For the empty cell at position $(7, 9)$, all possible values $k \in [1, 9]$ must be considered. Here, we show the transformation for $k = 4$.

Row Index in Binary Matrix The row index r in the binary matrix is given by:

$$r = i \cdot 9 \cdot 9 + j \cdot 9 + (k - 1)$$

Substituting $i = 7, j = 9, k = 4$:

$$r = 7 \cdot 81 + 9 \cdot 9 + (4 - 1) = 567 + 81 + 3 = 651$$

Column Indices in Binary Matrix The columns where this row has 1s are determined by the following constraints:

1. **Cell Constraint:**

$$c_1 = i \cdot 9 + j = 7 \cdot 9 + 9 = 81$$

2. **Row Constraint:**

$$c_2 = 81 + i \cdot 9 + (k-1) = 81 + 7 \cdot 9 + (4-1) = 147$$

3. **Column Constraint:**

$$c_3 = 162 + j \cdot 9 + (k-1) = 162 + 9 \cdot 9 + (4-1) = 246$$

4. **Sub-grid Constraint:**

$$c_4 = 243 + \left\lfloor \frac{i}{3} \right\rfloor \cdot 3 + \left\lfloor \frac{j}{3} \right\rfloor \cdot 9 + (k-1)$$

Substituting $i = 7, j = 9, k = 4$:

$$c_4 = 243 + 2 \cdot 3 + 3 \cdot 9 + 3 = 279$$

5. **Highlighted Region Constraint:** If (i, j) belongs to highlighted region $h = 3$:

$$c_5 = 324 + h \cdot 9 + (k-1) = 354$$

Binary Matrix Row Representation The binary matrix row for $r = 651$ has 1s in the columns:

$$\{c_1, c_2, c_3, c_4, c_5\} = \{81, 147, 246, 279, 354\}$$

All other columns in the row are 0.

Linked List Representation In the linked list:

- Each 1 in the binary matrix corresponds to a node.

- The nodes for this row are:

Node 1: Column 81,

Node 2: Column 147,

Node 3: Column 246,

Node 4: Column 279,

Node 5: Column 354.

6.4 Example: Number and Sudoku Grid Mapping for a Linked List Solution

Assume the following linked list solution rows:

$$\mathcal{R} = \{4, 651\}$$

Each row r corresponds to a specific decision in the Sudoku grid, represented by the row (i), column (j), and number (k) placed in that cell.

6.4.1 Mapping Row 4

The first row in the solution is $r = 4$. Using the formulas:

1. **Row Index i:**
$$i = \left\lfloor \frac{r}{81} \right\rfloor = \left\lfloor \frac{4}{81} \right\rfloor = 0$$

2. **Column Index j:**
$$j = \left\lfloor \frac{r \mod 81}{9} \right\rfloor = \left\lfloor \frac{4 \mod 81}{9} \right\rfloor = \left\lfloor \frac{4}{9} \right\rfloor = 0$$

3. **Number k:**
$$k = (r \mod 9) + 1 = (4 \mod 9) + 1 = 4 + 1 = 5$$

Thus, row $r = 4$ represents the decision "place 5 in cell $(0, 0)$".

6.4.2 Mapping Row 651

The second row in the solution is $r = 651$. Using the same formulas:

1. **Row Index i:**
$$i = \left\lfloor \frac{r}{81} \right\rfloor = \left\lfloor \frac{651}{81} \right\rfloor = 7$$

2. **Column Index j:**
$$j = \left\lfloor \frac{r \mod 81}{9} \right\rfloor = \left\lfloor \frac{651 \mod 81}{9} \right\rfloor = \left\lfloor \frac{9}{9} \right\rfloor = 1$$

3. **Number k:**
$$k = (r \mod 9) + 1 = (651 \mod 9) + 1 = 3 + 1 = 4$$

Thus, row $r = 651$ represents the decision "place 4 in cell $(7, 9)$".

For the Hyper Sukodu problem, as described above, the complete Hyper Sukodu solution can be obtained by decoding the set of rows obtained from the linked list processed by the algorithm.

7 Project Tech

7.1 Language and Library

Python is an excellent choice for implementing the Dancing Links (DLX) algorithm to solve Hyper Sudoku, So we use python as the programming language.

Advantage of using Python are:

1. Readability and Simplicity

Python's clean and high-level syntax allows for easy implementation of complex algorithms like DLX. Its readability:

- Reduces development time.
- Makes the code easier to debug and maintain.
- Facilitates collaboration among team members.

2. Rich Data Structures

Python provides powerful and flexible data structures, such as:

- **Lists:** Useful for representing rows and columns dynamically.
- **Dictionaries:** Efficient for mapping indices to constraints or decisions.
- **Sets:** Ideal for managing covered and uncovered constraints during the DLX process.

These structures simplify the implementation of the sparse matrix required by DLX.

3. Dynamic Memory Management

Python handles memory allocation and garbage collection automatically, eliminating the need for manual pointer management. This is particularly useful for the dynamic cover and uncover operations in the DLX algorithm compare to programming language like C or C++.

4. Visualization Tools

Python integrates well with libraries for visualization:

- **Matplotlib/Seaborn:** For visualizing the binary matrix or algorithm progress.
- **Graphviz/NetworkX:** For graphically displaying the linked list structure or solution tree.

7.1.1 Visualization

The current implementation provides console-based visualization by printing the solved Sudoku grid in a readable format. Each cell is displayed with its corresponding number, and empty cells (if any) are represented by dots (.). Future enhancements can include graphical interfaces or web-based visualizations to demonstrate the algorithm's execution step-by-step.

7.2 Project Running Method

7.2.1 Prerequisites

Python 3.6 or higher is required to run the solver.

7.2.2 Steps

1. **Clone the Repository:**

```
git clone https://github.com/botchway44/CS5100-hypersudoku-solver
cd CS5100-hypersudoku-solver
```

2. **(Optional) Create a Virtual Environment:**

```
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install Dependencies:** Matplotlib Library needs to be installed for the code to run Command for Python 3 users -

```
pip3 install matplotlib
```

7.2.3 Usage

Running the Solver

Execute the script via the command line, providing the path to your Hyper Sudoku puzzle file as an argument.

```
python3 HyperSudoku.py grids/grid_1.txt
```

7.2.4 Example

Given an input file named `grid_1.txt` located in the project directory, you can solve it by running the following command in the terminal:

```
python3 sudoku.py grid_1.txt
```

7.3 Input Format

The Hyper Sudoku puzzle is provided as a text file, formatted as follows:

- **Grid Representation:** The The Hyper Sudoku puzzle is represented as a 9x9 grid. puzzle is represented as a 9×9 grid.
- **File Format:** Inputs are provided via a text file where each line corresponds to a row in the Hyper Sudoku grid.
- **Cell Delimiter:** Cells within a row are separated by spaces.
- **Empty Cells:** Represented by the number `0`.

7.3.1 Example Input

An example input file might look like this:

```
0 0 0 0 0 0 3 0 0
3 9 0 0 0 0 0 2 0
8 0 7 0 0 0 9 6 0
1 1 0 0 4 3 0 6 0
0 6 0 0 0 0 0 0 0
0 7 5 1 0 0 0 0 9
```

```
0 8 0 0 4 0 0 0 0
0 4 0 0 0 0 0 2 0
0 0 5 0 0 0 0 0 2
```

7.3.2 Providing Inputs

- **File Input:** Users must create a text file (`.txt`) following the format described above.
- **Command-Line Argument:** The file path is provided as a command-line argument when running the program.

Example File Name: `grid_1.txt`

7.4 Dependencies

This project utilizes Python's standard libraries:

- `argparse`: For parsing command-line arguments.
- `pathlib`: For handling file paths.
- `time`: For timing the visualization.

No external libraries are required, ensuring ease of installation and compatibility. Except:

- `Matplotlib`: For visualization.

8 Reflection

8.1 What Went Well

Several aspects of the project went well, contributing to its overall progress and success:

- There were ample resources available to help the team understand the Dancing Links algorithm and how it could be implemented to solve a Sudoku puzzle. These resources ranged from research papers to tutorials and practical guides.
- Team members demonstrated strong ownership of their assigned tasks, ensuring that all aspects of the project were addressed collaboratively and efficiently.

8.2 What Didn't Go Well

While the project was ultimately successful, the team faced a few challenges:

- Implementing the Dancing Links algorithm to solve a Hyper Sudoku puzzle proved to be more challenging than anticipated. The added complexity of the Hyper Sudoku constraints required additional debugging and development effort.
- Initially, the team encountered communication challenges due to the inability of all members to meet in person on campus. This forced the team to rely solely on Microsoft Teams for discussions and collaboration, which sometimes slowed down decision-making and coordination.

8.3 What We'd Do Differently

Reflecting on the project, there are several improvements that could be made for future endeavors:

- **Better Visualization:** Developing a visual representation of the Dancing Links algorithm and its state as the Hyper Sudoku puzzle is being solved. This would make the algorithm easier to understand and debug.
- **Interactive Visualization:** Building an interactive tool where users can pause or stop the algorithm's execution and modify inputs dynamically through the visualization interface. This would make the tool more versatile and user-friendly.

References

[1] Jonathan Chu. *A Sudoku Solver in Java Implementing Knuth's Dancing Links Algorithm*. 2006. URL: https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html#dlxSudoku.

[2] Jonathan Chu. *Sudoku: Dancing Links Algorithm Presentation*. 2006. URL: https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/presentationboard.pdf.

[3] Sylvain Hu. *AI for Solving Puzzles*. BSc Computer Science with Artificial Intelligence Final Report, University of Leeds. 2021. URL: https://info.bb-ai.net/student_projects/project_reports/Silvain-Hu-Sudoku-FinalReport.pdf.

[4] Donald E. Knuth. *Dancing Links*. 2001. URL: https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/0011047.pdf.