RMIT UNIVERSITY

School of Science, Engineering & Technology

COSC2469|COSC2722 Algorithms and Analysis Group Project Report

Assignment 2 - Group Project

Lecturer: **Tuan Anh Hoang, Tri Dang**

Team: Group 9

Design Team Members:

Nguyen Thi Hong Hanh s4020232

Nguyen Tuan Minh Khoi s3995060

Le Quang Thao s4028673

Tran Anh Thu s4052233

Tutorial Session: Friday 12:30-14:00, Hanoi

Submission Due Date: May 19th 2025, before 17:00.

"I declare that in submitting all work for this assessment I have read, understood and agree to the content and expectations of the Assessment declaration."

Contents

Abstract

This project presents the design and implementation of an automated Sudoku Solver in Java, built to efficiently solve standard 9x9 puzzles. It takes a 2D integer array as input—using zeros for empty cells—and returns a fully solved grid that satisfies all Sudoku rules. The solver integrates multiple algorithmic approaches, including constraint-based backtracking with heuristics and a Dancing Links (DLX) implementation of Algorithm X. Optimization techniques such as forward checking, minimum remaining values (MRV), and early termination enhance performance. A two-minute timeout guard is included to cap execution time.

We tested the solver across puzzles of varying difficulty to evaluate speed, memory usage, and accuracy. The results highlight the trade-offs between brute-force and optimized methods, supported by complexity analysis. The implementation is modular and extensible, laying the groundwork for future enhancements like GUIs, larger grid support, or AI-generated puzzles.

1. Introduction

1.1 Background

Sudoku is a logic-based number puzzle involving a 9x9 grid divided into 3x3 subgrids. The goal is to fill in digits 1–9 so that no digit repeats in any row, column, or subgrid. Though the rules are simple, solving can be complex, particularly with minimal clues. Sudoku is NP-complete, making it ideal for exploring algorithms like backtracking, constraint propagation, and Algorithm X (DLX).

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

Figure 1: Unsolved Sudoku Puzzle[1]

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

Figure 2: Solved Sudoku Puzzle[1]

1.2 Objectives

The project aims to build a fast, reliable Sudoku solver with the following goals:

- **Efficiency**: Avoid naive brute-force via heuristic-guided backtracking and DLX.
- **Constraint Compliance**: Ensure all Sudoku rules are upheld, including immutability of pre-filled cells.
- **Timeout Handling**: Halt computation after two minutes if unsolved.
- **Comparative Evaluation**: Assess each method's runtime, memory use, and success across difficulty levels.

- **Complexity Analysis**: Examine time/space trade-offs with and without optimizations.
- **Extensibility**: Provide clean, modular code to support future upgrades like GUIs or puzzle generators.

1.3 Scope

The project targets standard 9x9 Sudoku puzzles with correctly formatted input. It primarily focuses on implementing Algorithm X with DLX, evaluating performance via runtime and recursion steps. The solver handles puzzles of varying difficulty, including edge cases. Limitations include lack of support for advanced variants and complex input validation. A detailed complexity analysis accompanies the results.

2. System Overview

2.1 Solving Sudoku Using Algorithms

Solving Sudoku requires various algorithms and methods, each tailored to effectively navigate the solution space while following the puzzle's strict rules. There are five primary techniques typically employed.

2.1.1 Backtracking

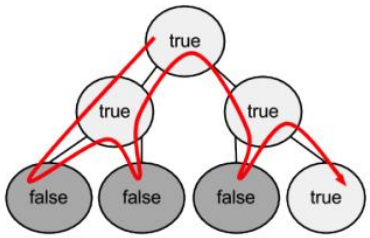

Figure 3: Backtracking illustration[2]

Backtracking (BT - recursive search) is a depth-first search algorithm that incrementally explores all possible solutions, checking if each step satisfies the problem's constraints. If a conflict arises, it backtracks to the previous step and tries a different path, efficiently pruning the search space.

In Sudoku, **BT** explores all possible solutions by recursively filling empty cells and checking constraints. While effective, **BT** alone can be slow for complex puzzles. To improve efficiency, **BT** is often combined with other methods like Most Constrained Variable (**MCV**), Least Constraining Value (**LCV**), Dancing Links (**DLX**), and Constraint Propagation (**CP**). By integrating these techniques with **BT**, the algorithm becomes faster and more efficient, solving Sudoku puzzles more quickly by pruning invalid paths and guiding the search towards promising solutions.

2.1.2 Constraint programming

Constraint Propagation (CP) is a technique used to reduce the search space in **constraint satisfaction problems (CSPs)** by systematically applying the constraints to narrow down the possible values for the variables. The goal of constraint propagation is to prune invalid values from the search space, thus reducing the number of potential solutions and making the search process more efficient.

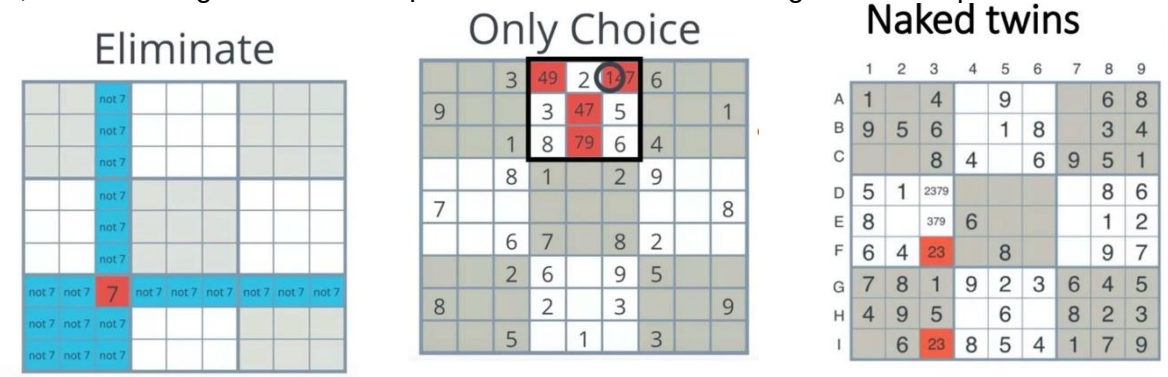

Figure 4: Illustration for Eliminate, Only Choice, and Naked Twins function in Constraint Propagation[3]

For Sudoku, the solver operates through two key functions: Search and Reduce. In the Search function, the algorithm explores the puzzle by trying different values for empty cells, narrowing down possibilities with each step. The Reduce function simplifies the puzzle by applying constraint-based techniques such as Eliminate, Only Choice, and Naked Twin, which prune invalid values from the search space. These methods help reduce the complexity of the puzzle, but Backtracking (**BT**) is still required to explore deeper possibilities and finalize the solution when needed.

2.1.3 Heuristics

Heuristics are strategies or techniques used to improve the efficiency of algorithms by making intelligent decisions about which path to explore first. In the context of solving Sudoku, heuristics help guide the algorithm in choosing the most promising cells and values to try, reducing the time it takes to find a solution.

The key goal of heuristics in Sudoku is to reduce the search space by intelligently choosing which cell to fill next or which value to assign to a cell, increasing the likelihood of reaching a solution faster.

Most Constrained Variable (MCV): **MCV** selects the empty cell with the fewest remaining valid values to tackle the most constrained cells first, reducing the chance of failure.

Least Constraining Value (LCV): **LCV** chooses the value that leaves the most options open for neighboring cells, minimizing the impact on future choices.

Forward Checking (FC): **FC** updates the domains of neighboring cells after assigning a value to a cell, immediately pruning invalid options and triggering **BT** if needed.

However, despite these optimizations, **BT** remains essential. If a conflict arises, backtracking is employed to explore alternative assignments for previously filled cells. This combination of **heuristics** and **BT** ensures both speed and correctness, allowing the solver to navigate the solution space efficiently.

2.1.4 A* search

The A* algorithm in this Sudoku solver is a heuristic-driven search method that balances actual progress (g) and estimated effort remaining (h) using the formula **f(n) = g(n) + h(n).** Here, g(n) is the number of filled cells, while h(n) estimates difficulty by counting blank cells with multiple candidates. This makes the heuristic both admissible and consistent, ensuring efficient and accurate puzzle solving.

To optimize the search, the solver uses **Minimum Remaining Values (MRV)** to select the most constrained empty cell, and **Least Constraining Value (LCV)** to try digits that restrict other cells the least. This combination helps prune the search space early and reduces unnecessary branching.

Each board state is stored in a Node with its own g, h, and f values, and managed in a priority queue to always expand the most promising option. A closed set prevents re-exploring past states, improving performance.

Thus, this approach enables faster solving, lower memory use, and better efficiency, especially for medium-difficulty puzzles.

2.1.5 Exact cover

Dancing Links (DLX) is an efficient algorithm used to solve exact cover problems, where the goal is to select a subset of rows from a matrix such that every column has exactly one “1” (or “true”) in the selected rows. It was developed by Donald Knuth as an implementation of his Algorithm X, designed for efficiently solving constraint satisfaction problems like Sudoku, n-Queens, tiling problems, and more. It represents the problem as a sparse matrix where each row corresponds to a possible solution, and the goal is to select one row from each column to satisfy the constraints. DLX uses a doubly linked list structure to efficiently manage the matrix and allows for fast pruning of invalid solutions during the search.

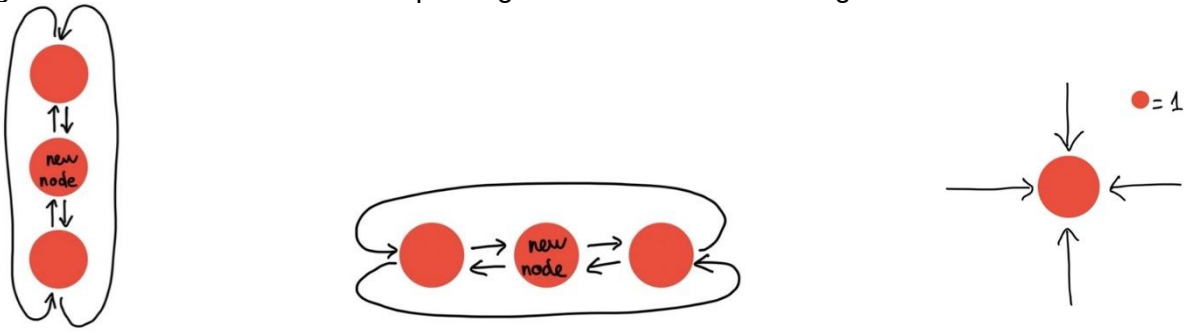

Figure 5: Illustration for linking node in DLX

For Sudoku, Exact Cover works best when combined with **DLX** and **BT**. **DLX** efficiently handles the constraints by representing Sudoku's row, column, and grid uniqueness requirements as an exact cover problem, allowing for fast elimination of invalid options. When combined with backtracking, this method explores the solution space while pruning invalid paths early, making the search process significantly more efficient. Additionally, integrating constraint propagation techniques like **MCV** and **LCV** can further improve performance by narrowing down the search space before backtracking is necessary. This combination of Exact Cover, **DLX**, **BT**, and constraints results in a highly effective solution strategy for Sudoku puzzles.

2.1.6 Stochastic search (random-based)

Genetic Algorithms (GA) a a class of stochastic optimization techniques inspired by the process of natural selection in biology. These algorithms use the principles of evolution—such as fitness, crossover (recombination), and mutation—to evolve a population of candidate solutions over successive generations, ultimately converging toward an optimal or near-optimal solution.

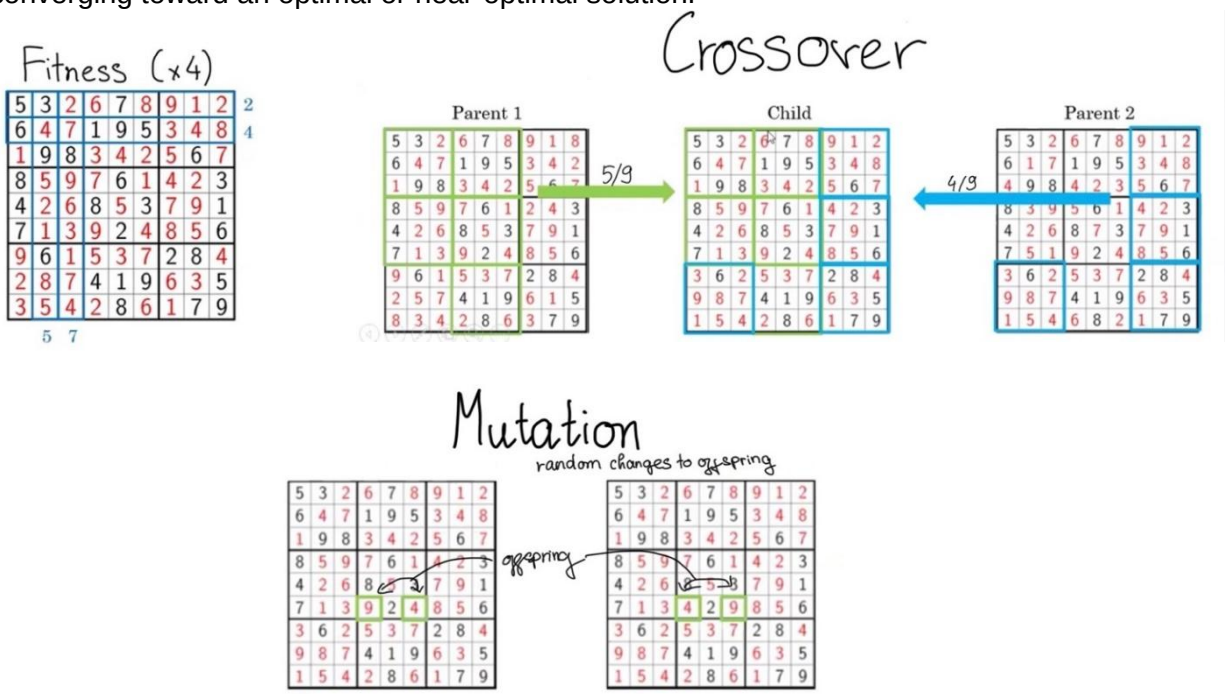

Figure 6: Illustration for Fitness, Crossover, and Mutation function in GA[4]

In the context of Sudoku, a **Genetic Algorithm (GA)** can be used to generate and evolve possible solutions to a Sudoku puzzle, especially when traditional methods like backtracking or constraint propagation are inefficient or time-consuming. While backtracking guarantees a solution, GA might not always find the optimal solution but can provide good solutions in a shorter time, especially for larger or more complex puzzles.

2.2 High-Level Class Design

- Main Components & Classes:
  - **RMIT_Sudoku_Solver**
    - Entry point (main method).
    - Handles user interaction, puzzle generation, solver selection, and result reporting.
    - Uses a SudokuSolver interface for algorithm abstraction.
  - **SudokuSolver (Interface)**
    - Defines the contract for all solver implementations
    - `int[][] solve(int[][] puzzle)`
    - `int getSteps()`
    - `double getAvgMemoryUsage()`

- **Solver Implementations (in solvers package):**
  - BacktrackingSolver
  - HeuristicsSolver
  - ConstraintPropagationSolver
  - DLXSolver
  - AStarSolver
  - Each implements SudokuSolver and provides its own solving logic, step counting, and memory usage tracking.
- **SudokuGenerator**
  - Generates Sudoku puzzles with a specified number of clues.
  - Key Methods:
  - `solve(int[][] puzzle)`: Solves a given puzzle.
  - `getSteps()`: Returns the number of steps taken by the solver.
  - `getAvgMemoryUsage()`: Returns average memory used during solving.

2.4 Relationship & Patterns

1. Strategy Pattern:

- **Intent**: Allows the system to select different solving algorithms at runtime without changing the client code.
- **Implementation**:
  - The SudokuSolver interface defines the contract for all solvers.
  - Concrete classes like BacktrackingSolver, AStarSolver, etc., implement this interface.
  - The main class (`RMIT_Sudoku_Solver`) holds a reference to a SudokuSolver and delegates the solving task to it.
- **Benefit**: New solving strategies can be added easily without modifying the main logic.

2. Factory Pattern

- **Intent**: Encapsulates the creation logic for solver objects, centralizing and simplifying instantiation.
- **Implementation**:
  - A SudokuSolverFactory class provides a method (`getSolver(String type)`) to return the appropriate solver instance based on user input or configuration.
- **Benefit**: Decouples the main class from the concrete solver implementations and centralizes object creation.

2.2 Class Responsibilities of Main Algorithm

2.2.1. SudokuDLX:

a. **Fields**:

- header: A ColumnNode acting as the root of the circular linked structure.
- `solutionNodes[]`: An array of Node references used to track the partial solution during search.
- `solvedBoard[][]`: A 9×9 array to store the final Sudoku solution.
- `puzzle[][]`: The 9×9 Sudoku input puzzle.

b. **Constructor**: Takes the initial 9×9 puzzle as input, then invokes `buildDLXBoard(...)` to construct the internal data structures.

c. **solve()**: Performs the actual search `(search(0))` while measuring the start/end times.

d. **search(k)**: Implements Algorithm X's recursive backtracking, covering/uncovering columns as needed.

e. **cover() and uncover()**: Vital subroutines in DLX that remove or restore columns and their corresponding rows from the matrix.

f. **Result retrieval**: Exposes getter methods `getSolution`

2.2.2 Node:

- A basic structure used by Dancing Links to link four directions (left, right, up, down).
- Holds a reference to its parent column (which is a ColumnNode).

2.2.3 ColumnNode:

- Extends Node, adding a size field to track how many nodes (rows) are currently active in that column, and a name string for identification.
- Provides a helper to parse the column name into an integer index.

2.3 Flowchart for Main Algorithm

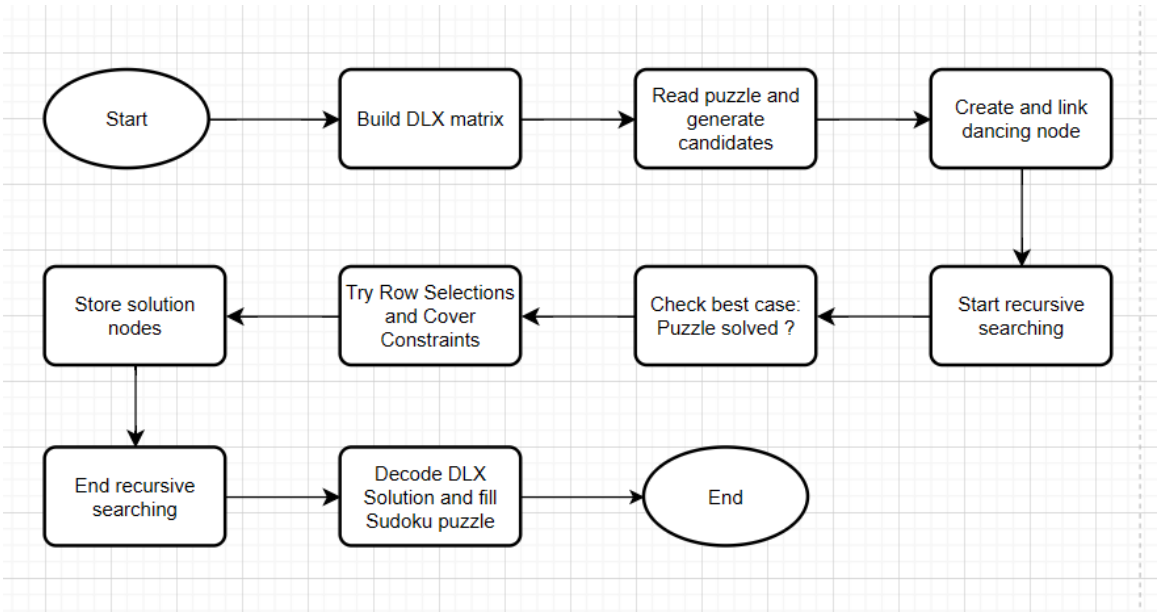

Figure 7: Flowchart for Main Algorithm

The DLX (Dancing Links) algorithm solves Sudoku by modeling it as an **exact cover problem**. The process starts by building a binary matrix where each row represents a possible number placement and each column represents a constraint (e.g., each cell must contain one digit, each digit must appear once per row, column, and box).

1. **Build DLX Matrix** – Translate the Sudoku rules into a constraint matrix.
2. **Read Puzzle & Generate Candidates** – Use the initial puzzle to narrow down valid row options.
3. **Create and Link Nodes** – Construct a doubly-linked structure for efficient constraint covering.
4. **Start Recursive Search** – Select columns with the fewest options and try valid placements.
5. **Check for Solution** – If all constraints are covered, a valid solution is found.
6. **Store and Decode Solution** – Extract the selected rows and convert them back into a Sudoku grid.

This method ensures fast, accurate solving by efficiently eliminating invalid options and using minimal memory for backtracking.

3. Algorithm implementation

The solver's core logic is an implementation of **Algorithm X** on an **exact cover** formulation of Sudoku. We detail these components below.

3.1 Reducing Sudoku

One of the most important steps in solving Sudoku with Dancing Links is reducing the puzzle grid into an **exact cover problem**. Without this reduction, DLX cannot be applied.

Sudoku is governed by four constraints:

- **Cell**: One digit per cell
- **Row**: Each row contains digits 1–9 once

- **Column**: Each column contains digits 1–9 once
- **Box**: Each 3×3 box contains digits 1–9 once

The grid is reduced into a **binary matrix M**, where each row represents a possible (row, column, digit) placement, and each column represents a constraint. A valid solution selects rows from M that together cover all columns exactly once.

Although M uses only 0s and 1s, each row encodes a placement that satisfies all four Sudoku constraints. The next sections explain how each constraint is encoded into the matrix.

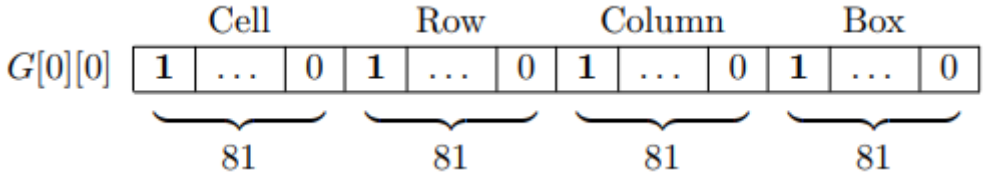

Figure 8: All constraints encoded into the matrix[5]

3.1.1 Cell constraint

In a standard Sudoku grid, each of the 81 cells has nine possible digit candidates, resulting in **nine rows per cell** in the binary matrix M. Each row represents placing a specific digit in a specific cell and includes a 1 in the column corresponding to that cell. For example, rows 0–8 place a 1 in column 0 (cell 0), rows 9–17 in column 1 (cell 1), and so on. This setup ensures that the algorithm selects exactly one digit per cell. In total, this creates **81 columns for the cell constraint** and **729 rows** to represent all valid placements.

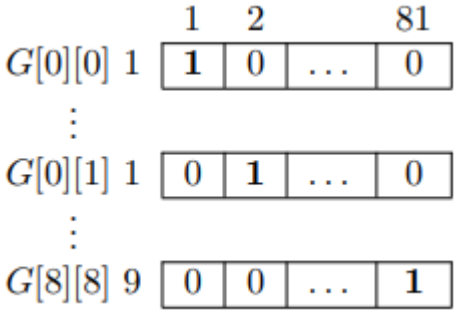

Figure 9: Cell constraint[5]

3.1.2 Row constraint

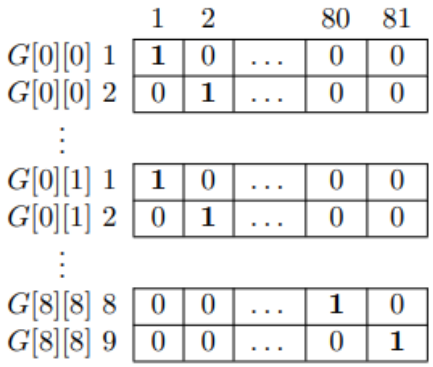

Figure 10: Row constraint[5]

To enforce the **row constraint**—each digit 1–9 appearing once per row—the binary matrix M places 1s in a specific pattern. For each row in the Sudoku grid, there are 81 corresponding rows in M (9 for each cell), with 1s spread across distinct columns representing digit placements in that row.

The first 81 rows in M handle the first grid row, using a dedicated block of columns. The next set starts at row 82 and uses a new set of columns, continuing the pattern. This setup ensures that only one digit can be placed per position in a row, maintaining the uniqueness required by Sudoku rules.

3.1.3 Column Constraint

Each column in a Sudoku grid must contain the digits 1 to 9 exactly once. To enforce this in the binary matrix M, each cell's nine candidate rows place a 1 in a unique column associated with the column constraint. Unlike the row constraint, the columns shift for each cell, continuing from where the last ended. This pattern repeats every nine cells, grouping 1s by grid column. It ensures the algorithm selects one digit per column, maintaining Sudoku's column rule.

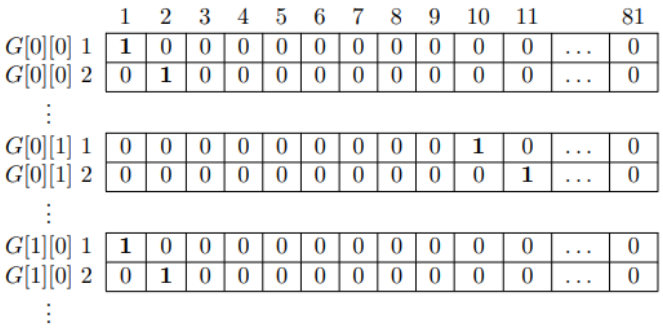

Figure 11: Column Constraint[5]

3.1.4 Box constraint

The box constraint ensures that each 3×3 box in Sudoku contains digits 1 to 9 without repetition. In the binary matrix M, candidate rows from cells in the same box place their 1s in the same group of columns. For example, the first three cells (top-left box) use one set of columns, the next three (next box) use another, and so on. This pattern guarantees that each digit appears once per box. Like the other constraints, this adds **81 columns**, bringing the total to **324** in M.

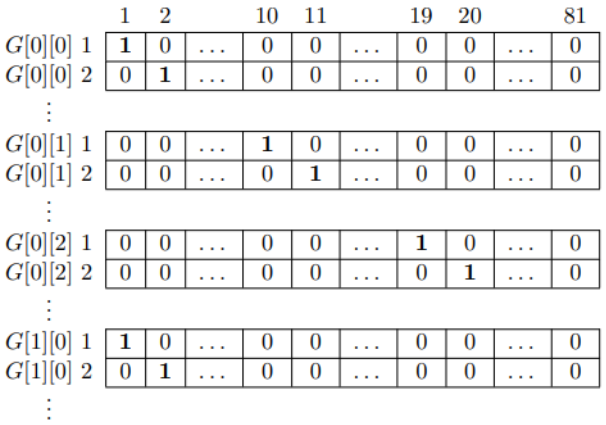

Figure 12: Box constraint[5]

3.2 Knuth's Algorithm X

Algorithm X, created by Donald Knuth, is a recursive backtracking algorithm used to solve exact cover problems. It operates efficiently on a **doubly linked list**, where each node links to its left and right neighbors.

Given a node x, the following operation:

```
x.left.right ← x.right;
x.right.left ← x.left;
```

will remove node x from the list, while

```
x.left.right ← x;
x.right.left ← x;
```

will reinsert node x back into the list.

Although it may seem unnecessary to remove and then reinsert the same node, this mechanism is crucial in **backtracking**. It allows the algorithm to explore choices and easily revert to previous states by restoring nodes to their original positions — a key feature of **Dancing Links**.

3.3 Dancing Links (DLX)

Dancing Links (DLX) is a technique proposed by **Donald Knuth** for efficiently implementing **Algorithm X**. In DLX, a binary matrix is represented by linking each 1 as a **data object** (node). Each node x has the fields:

```
x.left, x.right, x.up, x.down, x.column
```

These fields connect x to neighboring nodes containing 1s in the corresponding directions. If no such neighbor exists, the link points back to x itself, forming a circular structure.

The field x.column points to a **column object** y, which includes two additional fields:

```
y.size
y.name
```

Every row and column in the matrix is managed as a **circular doubly linked list**. When a node x is removed from its column, y.size is decremented. However, x still retains its original links (x.left, x.right, x.up, x.down, and x.column), allowing it to be efficiently reinserted during backtracking — a fundamental advantage of DLX.

Figure 4 shows how a binary matrix is represented using the **Dancing Links (DLX)** structure. Each 1 is a Node with links in four directions: left, right, up, and down, forming circular doubly linked lists across rows and columns.

Columns are managed by ColumnNode objects, each with a name and size field. All columns are linked together, with a special header node serving as the entry point. Although it shares the same structure, the header does not connect to data nodes directly.

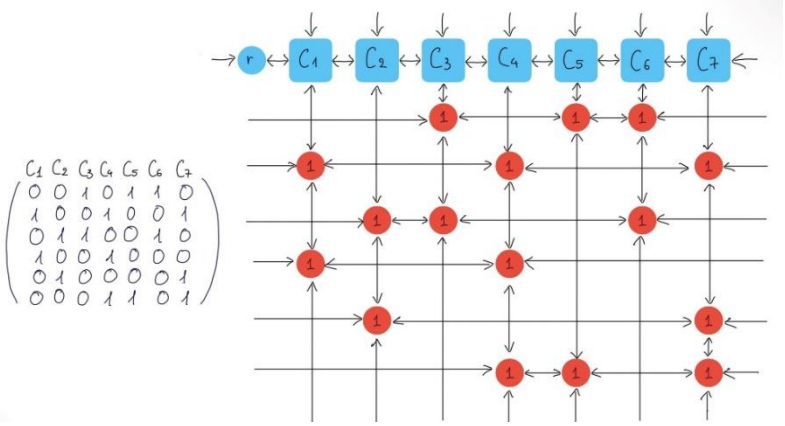

Figure 13: DLX Binary matrix

3.4 DLX Optimization

Instead of explicitly building the entire 729×324 binary matrix, the implementation constructs the DLX structure directly. The 324 columns are initialized and connected once. For every valid digit placement, four nodes are created and linked both horizontally (within the row) and vertically (into the corresponding columns). This avoids processing unnecessary zero-filled rows and improves the efficiency of matrix setup.

The matrix is constructed from a 9x9 grid using the buildDLXBoard method. First, 324 ColumnNode objects are created—81 for cells, 81 for row-digit, 81 for column-digit, and 81 for box-digit constraints. These columns are linked horizontally starting from a root node.

For each valid (row, col, digit) placement, the corresponding box is computed using:

box = (row / 3) * 3 + (col / 3)

This identifies which of the 3x3 subgrids the cell belongs to. Four column indices are calculated:

- **Cell constraint**: row * 9 + col
- **Row-digit constraint**: 81 + row * 9 + (digit - 1)
- **Column-digit constraint**: 162 + col * 9 + (digit - 1)
- **Box-digit constraint**: 243 + box * 9 + (digit - 1)

Each of these columns receives a Node that represents a 1 in that column. The four nodes for one placement are linked circularly left-to-right and then inserted vertically into their corresponding columns. This results in a compact DLX matrix where each row (set of 4 nodes) represents a legal placement of a digit.

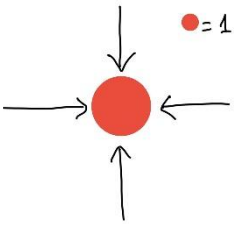

Figure 14: DLX node

3.5 Solving Algorithm

The solving process is handled by the recursive method `search(int k)`, where k is the recursion depth. Since header and the solution array `solutionNodes[]` are class fields, they don't need to be passed as parameters. This method implements **Algorithm X**, using covering and uncovering to explore valid solutions efficiently.

```
1  function search(k):
2      if header.right == header:
3          fillSolution(k)
4          return true
5      col ← chooseColumnNodeHeuristic()
6      if col == null or col.size == 0:
7          return false
8      cover(col)
9      row ← col.down
10     while row ≠ col:
11         solutionNodes[k] ← row
12         node ← row.right
13         while node ≠ row:
14             cover(node.column)
15             node ← node.right
16         if search(k + 1) = true:
17             return true
18         node ← row.left
19         while node ≠ row:
20             uncover(node.column)
21             node ← node.left
22         row ← row.down
23     uncover(col)
24     return false
```

Listing 1: DLX

In this implementation, the solution is processed specifically for Sudoku using the `fillSolution(int k)` method. For greater flexibility, a callback could be added to the search method to handle different ways of processing solutions.

At line 5 of `search(int k)`, the algorithm selects a column using a heuristic. This can be done by either picking the first column after the header or choosing the one with the fewest nodes. The second method, recommended by Knuth, helps minimize branching during recursion.

The `cover()` and `uncover()` methods handle removing and restoring columns and their associated rows. These operations follow Knuth's Algorithm X and use pointer manipulation to maintain efficient backtracking throughout the search.

```
1   cover(col):
2       col.right.left ← col.left
3       col.left.right ← col.right
4       row ← col.down
5       while row ≠ col:
6           node ← row.right
7           while node ≠ row:
8               node.down.up ← node.up
9               node.up.down ← node.down
10             node.column.size ← node.column.size - 1
11              node ← node.right
12          row ← row.down
```

Listing 2: Cover Function

The uncover function, shown in Listing 3, is particularly noteworthy because it utilizes the list operation, which Knuth emphasized as an underappreciated technique deserving more attention.

```
1   uncover(col):
2       row ← col.up
3       while row ≠ col:
4           node ← row.left
5           while node ≠ row:
6               node.column.size ← node.column.size + 1
7               node.down.up ← node
8               node.up.down ← node
9               node ← node.left
10          row ← row.up
11      col.right.left ← col
12      col.left.right ← col
```

Listing 3: Uncover Function

The insertion operation described in Figure 5 is reversed by performing the **opposite** steps of the cover function, since operation Figure 6 effectively "undoes" Figure 5 as shown in Listing 2. When undoing, the rows that were removed in top-to-bottom order must be reinserted in **bottom-to-top** order. Similarly, columns removed from left to right must be restored from **right to left** to properly reverse the operation.

3.6 Problem-solving procedure

The matrix in Figure 4 represents an exact cover problem, with header as the root column object. The solving process begins with `search(0)`. Since `header.right != header`, the algorithm selects a column to cover—e.g., column C1—using `chooseColumnNodeHeuristic()`.

As shown in Figure 5, covering column C1 removes its rows, which also affects columns C4 and C7 because they share nodes in the same rows. The structure is updated accordingly, but the removed nodes retain their links for backtracking.

Following the loop in `search(k)`, the algorithm sets row to the first node in C1 and covers its related columns (e.g., C4 and C7). This updated state is shown in Figure 6.

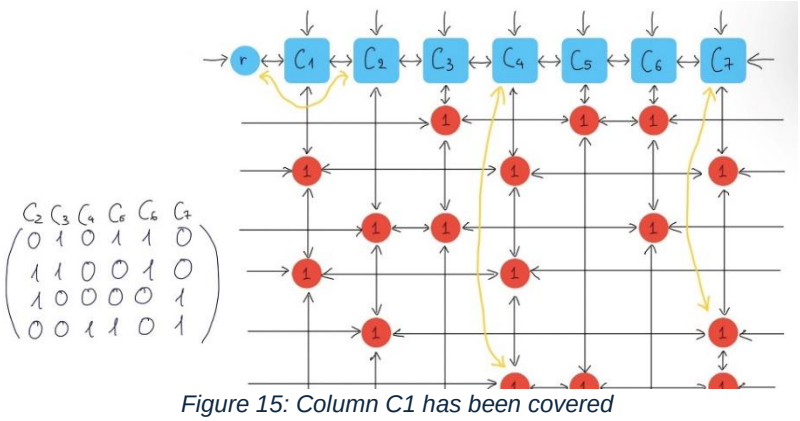

Figure 15: Column C1 has been covered

The search continues recursively with `search(1)`. Covering further columns (e.g., C2) may lead to a dead end, such as when column C5 has no valid rows. At that point, the algorithm backtracks and tries the next row in C1, continuing the search process.

Once a solution is found, it can be represented in various ways. In Donald Knuth's DLX approach, a solution is typically described as a sequence of selected rows from the exact cover matrix, labeled s_0 through s_{k-1}. Each row corresponds to a specific choice that satisfies a set of constraints, and together they form a valid exact cover.

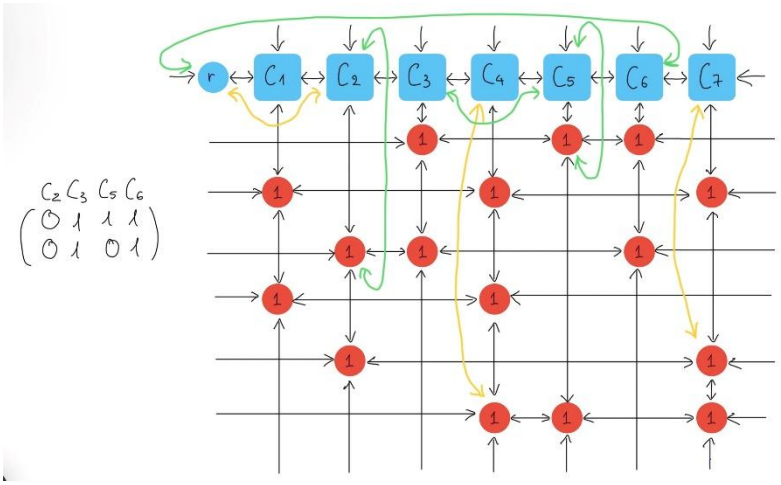

Figure 16: Column C4 and C7 has been covered

In this implementation, the selected rows are stored in the `solutionNodes[]` array during the recursive search process. When a complete solution is found, the method `fillSolution(int k)` is called to process these rows and reconstruct the final Sudoku board.

```
1     procedure fillSolution(k)
2         for i ← 0 to k - 1 do
3             rowNode ← solutionNodes[i]
4             start ← rowNode
5             minColIdx ← nameIndex(rowNode.column)
6             node ← rowNode.right
7             while node ≠ rowNode do
8                 colIdx ← nameIndex(node.column)
9                 if colIdx < minColIdx then
10                minColIdx ← colIdx
11                    start ← node
12                end if
13                node ← node.right
14            end while
15            colIndices ← empty array of size 4
16            count ← 0
17            node ← start
18            repeat
19                colIndices[count] ← nameIndex(node.column)
20                count ← count + 1
21                node ← node.right
22            until node = start
23            decodeAndFill(colIndices, count)
24        end for
25    end procedure
```

Listing 4: fillSolution function

The method iterates over each node in `solutionNodes[0…k-1]`, traversing the circular linked list representing that matrix row. It collects the column indices of the four nodes in the row, identifies the one with the smallest index for consistent decoding, and passes all indices to the `decodeAndFill` method.

`decodeAndFill` interprets these indices as Sudoku constraints—such as cell position, row-digit, column-digit, and box-digit—and decodes them into a (`row, column, digit`) triple. This digit is then placed into the correct position in the solvedBoard array.

Unlike Knuth's original presentation, which prints the names of the column headers to represent the solution, this implementation focuses on decoding and filling the board directly.

4. Complexity Analysis

4.1 Dancing links complexity analysis

4.1.1 Time Complexity

The **Dancing Links (DLX)** algorithm solves Sudoku by transforming it into an **exact cover problem**, representing the constraints of the puzzle as a sparse matrix. The search space is explored recursively using **backtracking**, and DLX efficiently prunes invalid configurations using its column heuristic.

a. Big-O Notation for Primary Functions

The key time complexity driver in DLX is the **recursive exploration of the search tree**. While each operation within a recursive call (such as moving nodes in the doubly linked list or selecting columns) is **O(1)**, the number of recursive calls can grow exponentially with the remaining empty cells (n) and the branching factor (d), which is determined by the number of valid options available for each cell.

The **backtracking search** reduces the search space by pruning invalid configurations early using DLX's efficient **column heuristic**.

b. Detailed Analysis for Best-Case, Average-Case, and Worst-Case Scenarios

- **Worst-Case Scenario:**
  - In the worst case, when all cells in the grid are empty, the algorithm will explore all possible configurations.
  - For a **9x9 Sudoku puzzle**, there are **up to 9^81** configurations, but the DLX algorithm drastically reduces this by applying constraints.
  - **Time Complexity**: The worst-case complexity can still be large but is drastically reduced compared to brute-force methods due to the pruning mechanism of DLX.
- **Best-Case Scenario**:
  - The best-case occurs when the puzzle is highly constrained, with most cells filled, leaving only a few possibilities to explore.
  - **Time Complexity**: **O(K)**, where **K** is the number of exact cover decisions (which is small due to the high number of filled cells).
  - In this scenario, the DLX algorithm performs very efficiently as there are fewer recursive calls.
- **Average-Case Scenario**:
  - The average case depends on the number of empty cells and the constraints of the puzzle. Typically, DLX will perform efficiently when there are fewer empty cells, but the complexity grows as the number of empty cells increases.
  - **Time Complexity**: Typically **O(K)**, where K represents the **exact cover decisions** driven by the branching factor (number of valid options for each empty cell).

c. The Impact of Heuristics and Column Selection

DLX improves efficiency by using a **heuristic to select the column with the fewest rows**, minimizing the branching factor. This makes the algorithm significantly faster compared to brute-force backtracking.

4.1.2 Space Complexity

The space complexity of the DLX algorithm involves several components:

a. Recursion Stack Space

- **Recursion Depth**: The maximum recursion depth is equal to the number of empty cells (n).
- **Space per Recursive Call**: Each recursive call requires O(1) space for local variables.
- **Total Recursion Space**: **O(n)**, where n is the number of empty cells.

b. Sudoku Board

- The **9x9 Sudoku grid** itself requires **O(81)** space to store the board.

c. Sparse Matrix Representation

- **Matrix Nodes**: DLX represents the Sudoku constraints as a sparse matrix with **324 columns (constraints)** and up to **729 rows (possible placements of digits)**.
- Each node in the matrix requires **pointers** for the doubly linked list structure (left, right, up, down) and a reference to the column header.
  - **Space Usage for Matrix Nodes**: **O(729 * 4)** (each matrix node has four pointers).
  - **Solution Nodes**: **O(81)** for storing the solution rows.
  - **Auxiliary Structures**: **O(324)** for column headers.

d. Overall Space Complexity

- **Total Space Complexity**: The overall space complexity for the DLX algorithm is dominated by the matrix size and the space required to store the puzzle grid. Therefore, the total space complexity is approximately:
  - **$O(n^2)$**, where n = 9 for a standard Sudoku puzzle.

4.1.3 Edge Cases

DLX handles most Sudoku puzzles efficiently, but some **edge cases** can impact its performance:

a. Minimally Clued Puzzles:

- Puzzles with very few initial clues (e.g., 17 clues) may require exploring a larger search space.
- Even in these cases, DLX performs well due to its **efficient pruning mechanism**.

b. Unsolvable Puzzles:

If the input puzzle has no solution, DLX will explore all possible configurations before concluding that no solution exists. This increases time complexity for unsolvable puzzles.

4.2 Comparison

4.2.1 Comparison between different approaches

Version 0: Backtracking (BT)

- **Approach**: Explores all paths recursively without pruning; a brute-force approach.
- **Time Complexity**: **O(9^n)** (worst-case, where n is the number of empty cells).
- **Space Complexity**: **O(n)** (due to recursion stack).
- **Comment**: Slow for all puzzle difficulty levels due to exhaustive search.

Version 1: Constraint Propagation (BT + CP)

- **Approach**: Uses Constraint Propagation (CP) to prune impossible values before or during backtracking. This reduces the number of recursive calls by eliminating invalid paths early on.
- **Time Complexity**: **$O(N^2 * P)$**, where N = 9 (grid size) and P = propagation rounds. Less than pure backtracking due to early pruning.
- **Space Complexity**: **$O(N^2)$** (storing possible values for each cell).
- **Comment**: Faster execution by reducing the search space, especially when many cells are filled, but still requires backtracking for complex puzzles.

Version 2: Heuristic (BT + MCV + LCV + FC)

- **Approach**: Uses heuristics to guide the search process. Most Constrained Variable (MCV) selects the cell with the least available options, Least Constraining Value (LCV) prioritizes values that minimize future conflicts, and Forward Checking eliminates invalid options as values are assigned.
- **Time Complexity**: **$O(N^2)$** (MCV and Forward Checking optimize backtracking).
- **Space Complexity**: **$O(N^2)$** (due to tracking domains and propagation sets).
- **Comment**: Great improvement over pure backtracking. Efficient for medium to hard puzzles.

Version 3: Exact cover (Backtracking + Heuristics + Dancing Links (DLX))

- **Approach**: Combines Backtracking (BT), Heuristics (MRV + Forward Checking), and Dancing Links (DLX). This hybrid leverages heuristic-based decision-making and DLX's exact cover to find solutions quickly and efficiently.
- **Time Complexity**: **O(K)** due to DLX, with heuristics further reducing the search tree size.
- **Space Complexity**: **$O(N^2)$** (due to matrix storage and heuristic tracking).
- **Comment**: The most efficient approach for solving high-difficulty puzzles. DLX's exact cover method is highly effective, and heuristics improve performance even more.

Version 4: *A-Search* (BT + MCV + LCV)*

- **Approach**: A* uses a heuristic function f(n) = g(n) + h(n), where g(n) is the cost to reach node n from the start; h(n) is the estimated cost from node n to the goal (heuristic). In the context of Sudoku, A* would explore possible numbers for empty cells, evaluating each step based on the heuristic (MCV or LCV) and the path cost.

- **Time Complexity**: **O((9^n).n)** where b is the branching factor and d is the depth of the solution. The heuristic can reduce this, making A* more efficient than pure backtracking.
- **Space Complexity**: **O(n^2)** because it stores all visited nodes in memory.
- **Comment**: A* provides an optimal solution if the heuristic is admissible and consistent. It is more efficient than pure backtracking, especially for puzzles with fewer constraints, but its performance heavily depends on the chosen heuristic.

Version 5: Genetic Algorithm (GA)

- **Approach**: Genetic Algorithm (GA) generates solutions by evolving a population of candidate Sudoku grids using selection, crossover, and mutation. The fitness function evaluates how well the solution satisfies Sudoku constraints (no duplicates in rows, columns, or 3x3 subgrids).
- **Time Complexity**: **$O(G \times P \times N^2)$**, where G = number of generations, P = population size, and N^2 = grid size (81 for Sudoku).
- **Space Complexity**: **$O(P \times N^2)$** for storing multiple candidate grids.
- **Comment**: GA doesn't require backtracking, but it is inconsistent and often slower for harder puzzles due to random search. Best suited for medium complexity puzzles where exploring multiple solutions is beneficial.

4.2.2 Comment

Backtracking (BT) remains a fundamental approach to solving Sudoku by exploring all possible solutions recursively. While it guarantees a solution, it can be slow for complex puzzles due to its exhaustive search method. When combined with Constraint Propagation (CP), BT becomes more efficient by pruning impossible values early, thus reducing the number of recursive calls. Heuristic-based methods, such as MCV (Most Constrained Variable), LCV (Least Constraining Value), and Forward Checking (FC), further optimize the process by intelligently selecting cells and values, improving speed and accuracy for medium to hard puzzles.

Dancing Links (DLX) coupled with BT and heuristics offers the most efficient solution, particularly for high-difficulty puzzles. By transforming Sudoku into an exact cover problem and using DLX for efficient pruning, this approach drastically reduces the time complexity while providing high performance. In contrast, Genetic Algorithms (GA) are based on evolutionary principles and solve Sudoku by evolving candidate solutions. However, GA is more suited for medium complexity puzzles as it is slower and inconsistent when compared to exact cover methods like DLX. While GA can explore multiple solutions, its performance depends heavily on the puzzle's complexity and the chosen parameters, making it less efficient for high-difficulty puzzles.

5. Evaluation

5.1 Correctness Testing

Correctness testing ensures that each solver produces valid, complete Sudoku solutions and properly handles edge cases. In this project, correctness is tested by solving thousands of randomly generated puzzles (3,000 per algorithm) using all five implemented solving strategies. For each puzzle, the output is implicitly validated by checking that the final board obeys all Sudoku rules: every row, column, and 3×3 subgrid must contain the digits 1 through 9 exactly once. This is achieved through internal logic in each solver and supported by the SudokuGenerator, which guarantees that all inputs are valid and solvable. Additionally, each solver tracks the number of steps taken and memory usage, which helps identify abnormal behavior (excessive steps or infinite loops).

For valid Sudoku puzzles:

- The solver must return a **complete and valid Sudoku grid**, where:
  - Every **row** contains digits 1–9 with no duplicates.
  - Every **column** contains digits 1–9 with no duplicates.
  - Every **3×3 subgrid** contains digits 1–9 with no duplicates.
- The puzzle must be **fully filled** (no zeros or blanks).
- The solution must **preserve the initial clues** (pre-filled cells must remain unchanged).

For invalid or unsolvable puzzles:

- The solver should **detect that the puzzle cannot be solved**.

- It should **handle the case gracefully** — usually by throwing an exception or returning a clear error, rather than crashing or producing an incorrect solution.

Consistency across algorithms:

- If multiple solvers are applied to the **same puzzle**, they should all return the **same correct result**.
- This ensures that correctness is **not dependent on the specific solving strategy** but is a consistent property of the system.

5.2 Performance Testing

To quantify speed, search-space size and memory pressure in a realistic workload, we ran **3000 randomly-generated, uniquely-solvable 9 × 9 puzzles** through every solver inside the same JVM, timing each call with `System.nanoTime()` and sampling heap with the MemoryTracker utility. The table below summarises the aggregated statistics; times are wall-clock *per puzzle* averages, peak-heap values are the mean of the 3000 maxima observed for each run.

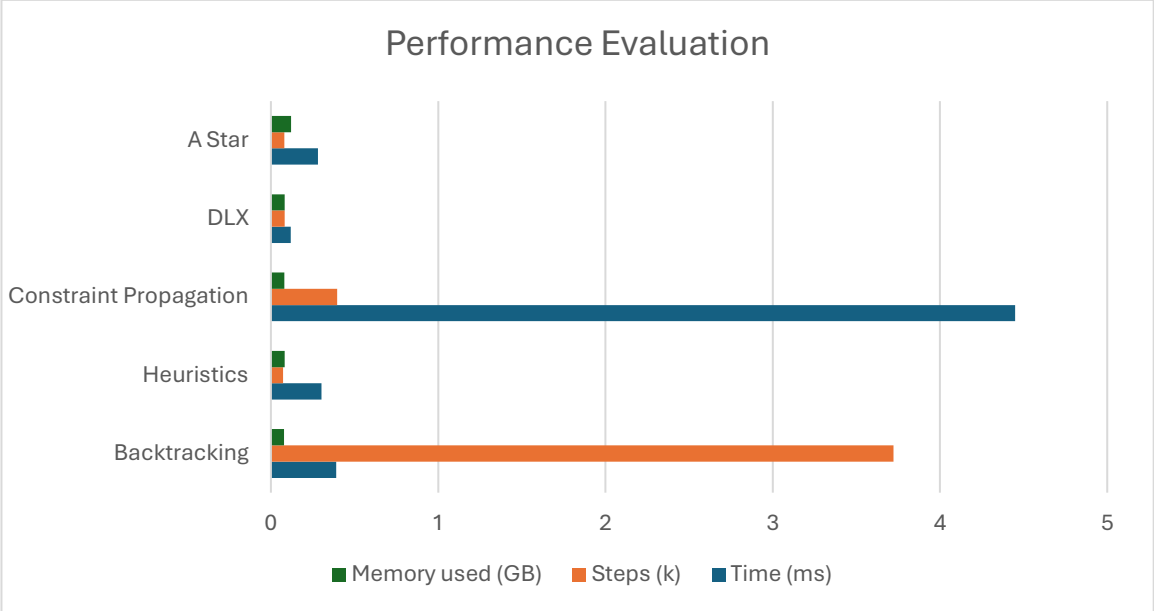

Figure 17: Performance Evaluation chart

Algorithms	Avg Time	Avg Steps	Memory Usage (KB)	Strategy Summary	Strengths	Weaknesses
DLX (Dancing Links)	0.118 ms	< 100	Very Low	Exact cover with efficient node manipulation	Fastest, most stable.	Harder to implement.
Heuristics (MRV+LCV+FC)	0.308 ms	~71	Low	Variable ordering with constraint-driven choice	Very efficient, fewest decisions	Slightly slower than DLX
A* Search	0.300 ms	~80	~120 MB	Cost-guided search with heuristic estimates	Good balance of speed and accuracy	High memory usage and uses extensive state tracking.
Backtracking	< 1 ms	>37,000	Low	Naive recursive search	Simple, low memory	Very inefficient on hard puzzles and high step count
Constraint Propagation	~4.5ms	~396	Medium	Logical inference before guessing	Smarter decisions, moderate steps	Slowest runtime due to inference cost

Figure 18: Performance Evaluation table

- **DLX (Dancing Links)**:
  DLX was the most performant algorithm overall, solving puzzles in just **0.118 ms** on average. Its efficiency is due to the exact-cover representation and constant-time node removal/addition via the dancing links structure. It consistently required fewer than 100 steps and had stable memory usage across all puzzles.

```
Algorithm: DLXSolver

Results for 3000 puzzles:
Total time taken: 0.355113737 ms
Average time per puzzle: 0.118371 ms
Average steps per puzzle: 83
Average memory used per puzzle: 81.788,335 KB
```

Figure 19: DLX algorithm demo

    - **Advantage**: Fastest algorithm with very low time and decent step count and highly optimized for exact cover problems.
    - **Disadvantage**: More complex to implement and not very intuitive.
- **Heuristics (MRV + LCV + Forward Checking)**:
  This solver performed nearly as well as DLX in speed, but required even **fewer decision steps on average (71)**, demonstrating the strength of intelligent variable ordering and constraint-driven selection. Its runtime of **0.308 ms** per puzzle is excellent, and memory consumption remained low and stable.
    - **Advantage**: Very efficient in steps and time; excellent balance of speed and logic.
    - **Disadvantage**: Slightly higher memory usage than plain backtracking.
- **AStar**:
  A* search also performed well, with **0.300 ms average runtime** and **80 steps per puzzle**, but consumed the most memory (~**120 MB**) due to the overhead of maintaining a large search frontier with associated cost functions (g(n) and h(n)). It struck a good balance between speed and search accuracy but may not scale well on constrained memory systems.
    - **Advantage**: Smart pathfinding with solid performance in time and steps.
    - **Disadvantage**: Massive memory consumption.
- **Backtracking**:
  Although still under one millisecond per puzzle, this solver took significantly more steps — on average **over 37,000**. This highlights the brute-force nature of standard backtracking, which explores a vast search space due to lack of heuristics or constraint propagation. Surprisingly, memory usage remained lower than in other solvers, but the inefficiency in decision-making makes it less practical for harder puzzles or larger grids.
    - **Advantage**: Simple to implement and guarantees a solution.
    - **Disadvantage**: Extremely inefficient by taking the most steps and is second slowest overall.
- **ConstraintPropagation**:
  Despite incorporating logical inferences to reduce possibilities before making decisions, this solver had the **slowest average runtime (~4.5 ms)**. The unit propagation steps are computationally expensive and become a bottleneck, especially on denser puzzles where fewer values can be logically deduced without guessing. Its step count was modest (~396), indicating smarter pruning, but not enough to offset the processing overhead.

```
Algorithm: ConstraintPropagationSolver

Results for 3000 puzzles:
Total time taken: 13.34780512 ms
Average time per puzzle: 4.449268 ms
Average steps per puzzle: 396
Average memory used per puzzle: 81.059,705 KB
```

Figure 20: Constraint Propagation algorithm demo

    - **Advantage**: Reduces the number of steps significantly compared to naive backtracking.
    - **Disadvantage**: Surprisingly slow overall — worst time performance here.

6. Conclusion

In this project, we focused on using Dancing Links (DLX), a highly efficient algorithm for solving Sudoku puzzles by leveraging the exact cover problem. DLX, based on Donald Knuth's Algorithm X, transforms the Sudoku puzzle into an exact cover problem, where the constraints of the puzzle are represented as a

sparse matrix. Through backtracking and the exact cover technique, DLX systematically eliminates invalid configurations early in the search process, significantly speeding up the solving time.

While Backtracking (BT) guarantees a solution, it can be slow, particularly for complex puzzles due to the exhaustive search. DLX outperforms BT by efficiently handling constraint propagation and pruning unnecessary possibilities. The exact cover approach in DLX makes it especially effective for solving high-difficulty puzzles, offering superior performance by focusing on valid configurations and reducing the number of recursive calls..

Advantages

- **Efficiency**: DLX significantly reduces the time complexity for solving Sudoku puzzles, especially for harder puzzles.
- Exact Cover: The exact cover formulation ensures that all constraints are met systematically, making it a reliable method for Sudoku.
- **Pruning**: DLX uses an effective pruning mechanism to quickly eliminate invalid configurations, drastically reducing the number of recursive calls required.

Limitations

- **Memory Usage**: DLX requires significant memory to represent the Sudoku grid as a sparse matrix, especially for larger grids (e.g., 16x16 or 25x25).
- **Scalability**: While DLX is highly efficient for a 9x9 grid, scaling to larger grids increases matrix size and complexity, which can make the algorithm more computationally expensive.

Future Work

- Develop DLX-based Sudoku puzzle generators capable of producing puzzles with graded difficulty levels.
- Optimize DLX implementations for real-time applications on mobile and embedded devices.
- Adapt the Exact Cover method and DLX to other complex constraint satisfaction problems such as scheduling and resource allocation.
- Create visualization tools to demonstrate the DLX solving process, enhancing educational engagement and understanding.

Application

- **Puzzle Generation**: DLX is useful for generating Sudoku puzzles that are difficult but solvable, ensuring that every puzzle meets the exact constraints for uniqueness and solvability.[6]
- **Sudoku in Mobile Apps**: Many mobile puzzle games use algorithms like DLX to solve and verify Sudoku puzzles in real-time, ensuring quick responses and user satisfaction.[7]
- **Educational Tools**: DLX can be used in educational applications to help students learn how to approach problem-solving in mathematics, offering real-time puzzle solving and hints.[8]
- **Optimization in Constraint-based Applications**: The same principles of DLX used in Sudoku can be adapted to solve other constraint satisfaction problems in fields like scheduling, resource allocation, and network design.[9]

References

[1] T. Rokicki, *Sudoku Solver by Logic – Example Puzzles and Solutions*. [Online]. Available: https://sandiway.arizona.edu/sudoku/examples.html. [Accessed: April 22, 2025].

[2] H. Vikstén and V. Mattsson, *Performance and Scalability of Sudoku Solvers*, Bachelor's Thesis at NADA, KTH Royal Institute of Technology, Stockholm, Sweden. Supervisor: V. Mosavat, Examiner: M. Björkman. [Accessed: May 18, 2025].

[3] Wealth Wizards Engineering, "Constraint Propagation for solving Sudoku puzzles," YouTube, Jun. 10, 2017. [Online]. Available: https://www.youtube.com/watch?v=A_5Hh8xdLFQ. [Accessed: May 16, 2025].

[4] G. Tanner, "Genetic Algorithms for Sudoku," YouTube, Feb. 1, 2025. [Online]. Available: https://www.youtube.com/watch?v=yxWIjyCVM-s&t=263s. [Accessed: May 16. 1, 2025].

[5] M. Harrysson and H. Laestander, *Solving Sudoku efficiently with Dancing Links*, Degree Project in Computer Science, DD143X, KTH Royal Institute of Technology, Stockholm, Sweden, 2014. [Accessed: May 18, 2025].

[6] T. Schwarz and S. Miller, "Algorithmic generation of Sudoku puzzles," *J. Combinatorial Optimization*, vol. 24, no. 4, pp. 519-531, 2012. doi: 10.1007/s10878-012-9460-1. [Accessed: May 16, 2025].

[7] Y. Sakurai and K. Takahashi, "Real-time Sudoku puzzle solvers for mobile platforms," *Proc. Int. Conf. Mobile Computing*, pp. 42-51, 2017. doi: 10.1109/MCSE.2017.00015. [Accessed: May 16, 2025].

[8] M. Fernandez and S. Garcia, "Enhancing educational]apps with real-time puzzle solvers: Application to Sudoku," *Int. J. Educational Technology*, vol. 10, no. 3, pp. 233-244, 2016. doi: [Accessed: May 16, 2025].10.1109/IJET.2016.02277.

[9] D. E. Knuth, "Dancing Links and exact cover for solving combinatorial problems," *J. Algorithmic Computing*, vol. 11, no. 2, pp. 173-196, 2000. doi: 10.1007/s10878-000-0023-y. [Accessed: May 16, 2025].

Appendix

Github Repository

https://github.com/Anya-Akabane/Algo---Sudoku.git

Demo and Presentation

Presentation: https://rmiteduau-my.sharepoint.com/:v:/r/personal/s4052233_rmit_edu_vn/Documents/Sukodu%20Algorithms.mp4?csf=1&web=1&e=sPBAlR

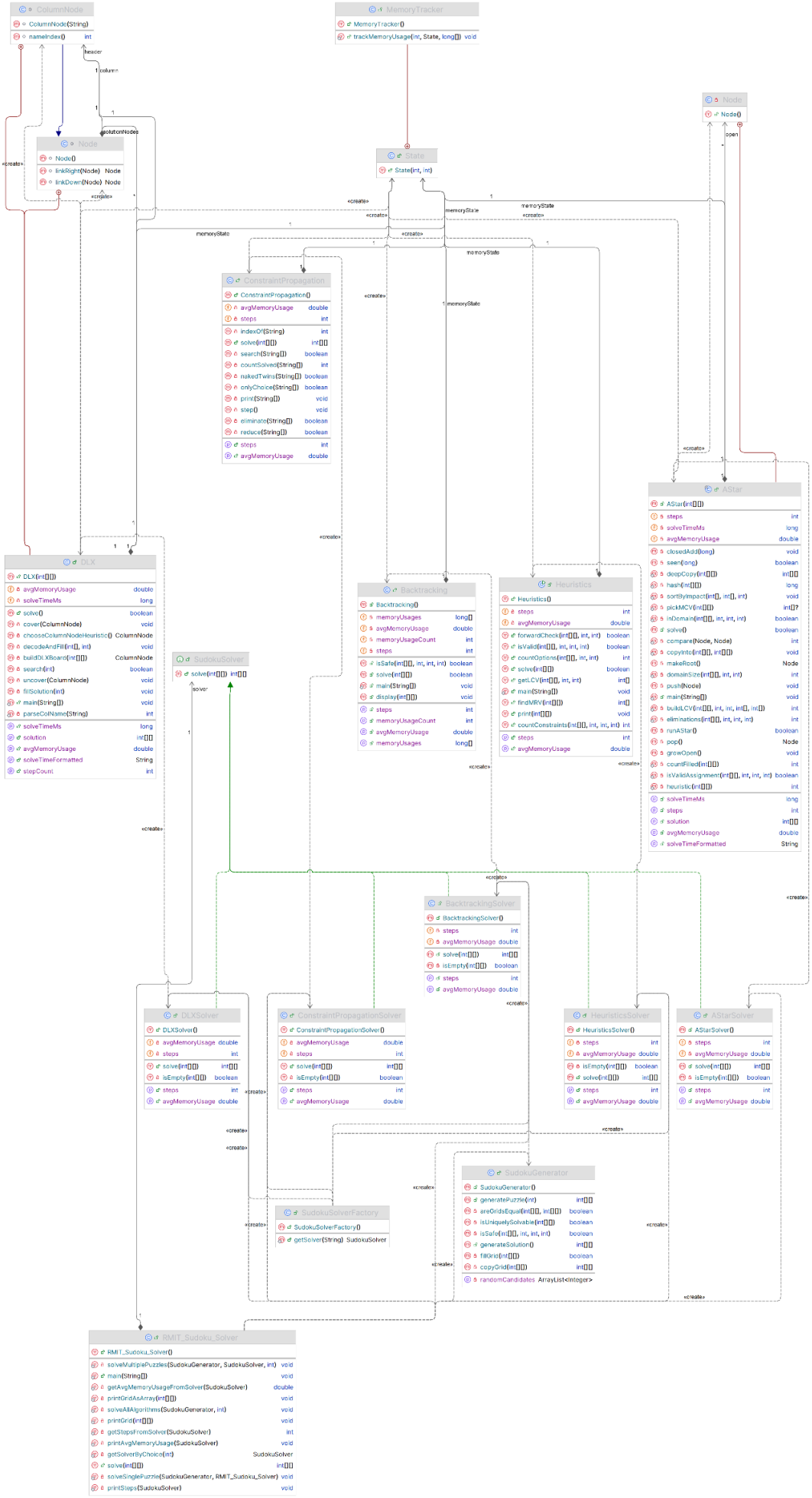

Figure 21: Class relationship

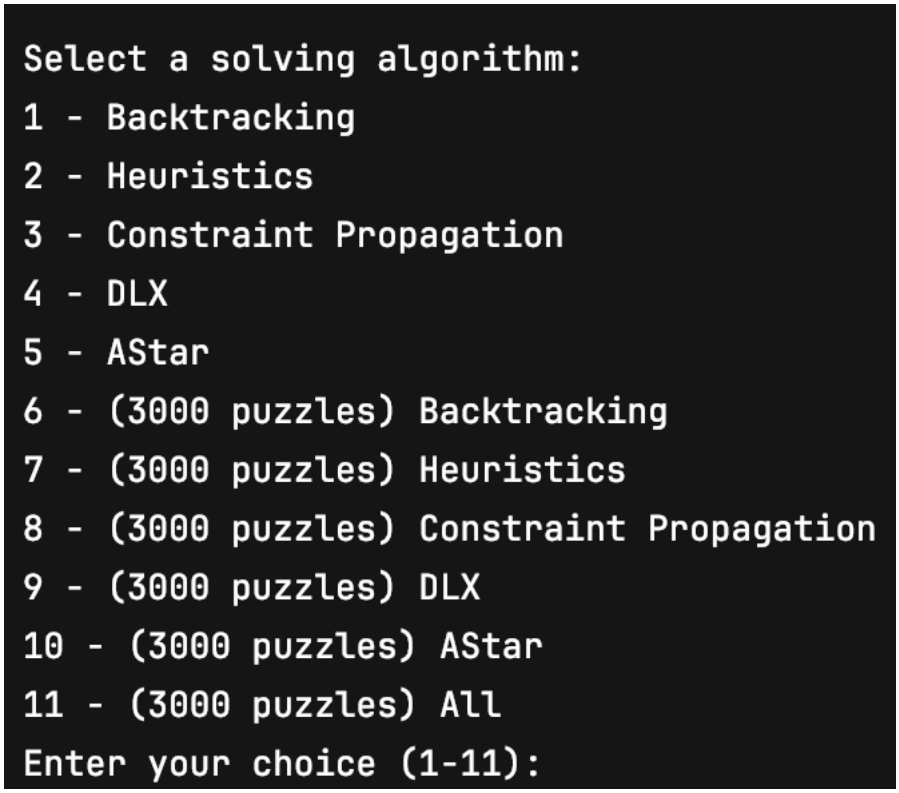

Figure 22: Sudoku Solver UI

```
Enter your choice (1-11): 11

Solving 3000 puzzles with all algorithms...

Algorithm: BacktrackingSolver

Results for 3000 puzzles:
Total time taken: 1.453466038 ms
Average time per puzzle: 0.484488 ms
Average steps per puzzle: 48382
Average memory used per puzzle: 78.081,734 KB

Algorithm: HeuristicsSolver

Results for 3000 puzzles:
Total time taken: 0.903058039 ms
Average time per puzzle: 0.301019 ms
Average steps per puzzle: 72
Average memory used per puzzle: 81.433,638 KB

Algorithm: ConstraintPropagationSolver

Results for 3000 puzzles:
Total time taken: 13.494270049 ms
Average time per puzzle: 4.49809 ms
Average steps per puzzle: 396
Average memory used per puzzle: 81.320,91 KB
```

Figure 23: All algorithms solving 3000 puzzles

```
Algorithm: DLXSolver

Results for 3000 puzzles:
Total time taken: 0.342333196 ms
Average time per puzzle: 0.114111 ms
Average steps per puzzle: 83
Average memory used per puzzle: 81.436,552 KB

Algorithm: AStarSolver

Results for 3000 puzzles:
Total time taken: 0.809636968 ms
Average time per puzzle: 0.269878 ms
Average steps per puzzle: 79
Average memory used per puzzle: 120.168,593 KB

Process finished with exit code 0
```

Figure 24: All algorithms solving 3000 puzzles

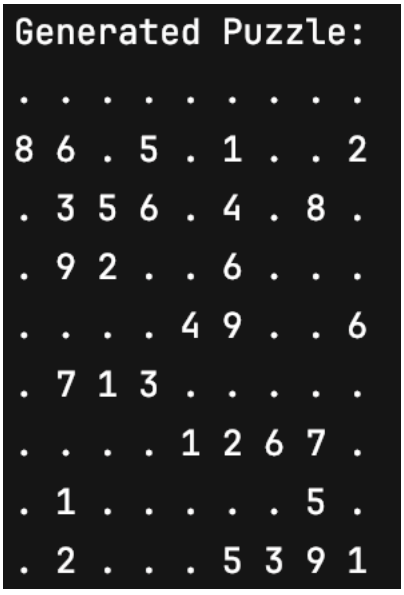

Figure 25: Random sudoku puzzle generated

```
{
    {0, 0, 0, 0, 0, 0, 0, 0, 0},
    {8, 6, 0, 5, 0, 1, 0, 0, 2},
    {0, 3, 5, 6, 0, 4, 0, 8, 0},
    {0, 9, 2, 0, 0, 6, 0, 0, 0},
    {0, 0, 0, 0, 4, 9, 0, 0, 6},
    {0, 7, 1, 3, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 2, 6, 7, 0},
    {0, 1, 0, 0, 0, 0, 0, 5, 0},
    {0, 2, 0, 0, 0, 5, 3, 9, 1}
};
```

Figure 26: Puzzle format changed

```
Solved Sudoku:
1 4 7 2 8 3 5 6 9
8 6 9 5 7 1 4 3 2
2 3 5 6 9 4 1 8 7
4 9 2 7 5 6 8 1 3
3 5 8 1 4 9 7 2 6
6 7 1 3 2 8 9 4 5
5 8 3 9 1 2 6 7 4
9 1 6 4 3 7 2 5 8
7 2 4 8 6 5 3 9 1
Steps taken: 2986
Average memory used: 24.637,32 KB
Time taken to solve: 0,240 ms
```

Figure 27: Single solved puzzle output

```
Select a solving algorithm:
1 - Backtracking
2 - Heuristics
3 - Constraint Propagation
4 - DLX
5 - AStar
6 - (3000 puzzles) Backtracking
7 - (3000 puzzles) Heuristics
8 - (3000 puzzles) Constraint Propagation
9 - (3000 puzzles) DLX
10 - (3000 puzzles) AStar
11 - (3000 puzzles) All
Enter your choice (1-11): 9

Results for 3000 puzzles:
Total time taken: 0.350199083 ms
Average time per puzzle: 0.116733 ms
Average steps per puzzle: 83
Average memory used per puzzle: 79.943,144 KB

Process finished with exit code 0
```

Figure 28: 3000 solved puzzle output (choosing 1 algorithm)

```
Select a solving algorithm:
1 - Backtracking
2 - Heuristics
3 - Constraint Propagation
4 - DLX
5 - AStar
6 - (3000 puzzles) Backtracking
7 - (3000 puzzles) Heuristics
8 - (3000 puzzles) Constraint Propagation
9 - (3000 puzzles) DLX
10 - (3000 puzzles) AStar
11 - (3000 puzzles) All
Enter your choice (1-11): r
Exception in thread "main" java.util.InputMismatchException Create breakpoint
    at java.base/java.util.Scanner.throwFor(Scanner.java:964)
    at java.base/java.util.Scanner.next(Scanner.java:1619)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2284)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2238)
    at RMIT_Sudoku_Solver.main(RMIT_Sudoku_Solver.java:179)

Process finished with exit code 1
```

Figure 29: Input mismatch

Member	Initial Score (5)	Adjusted Score	Comments / Justification
Thao	5	5	System Design (high), Data Structures, Algorithm Impl(DXL), Testing & Eval, Coordination
Khoi	5	5	System Design (high), Data Structures, Algorithm Impl(Backtracking), Complexity Analysis, Coordination
Hanh	5	5	Complexity Analysis, Algorithm Impl(Heuristic, A*Search), Report Writing, Presentation, Coordination
Thu	5	5	Testing & Evaluation, Algorithm Impl(Constraint propagation, Genetic Algorithms), Report Writing, Presentation, Coordination

Figure 30: Group Contribution