INTRODUCTION

Sudoku is a logic-based number-placement puzzle. It Originated from Japan in the late 20th century. The game is played on a 9x9 grid, divided into smaller 3x3 grids. The goal is to fill each row, column, and 3x3 grid with the numbers 1 through 9, without repeating any numbers within the row, column, or grid. Sudoku starts with some cells already filled in, and player needs to fill the remaining cells. Player can fill the cells in any order. Once all the cells are filled without violating sudoku constrains, sudoku is said to be solved. Some Sudoku puzzle may have multiple valid solutions. It depends on how its initial values are filled in.

While the standard Sudoku is played on a 9x9 grid, there are larger and more challenging versions of the game. ex:16x16 Sudoku, also known as “Hexadoku”. The rules are the same as the standard 9x9 Sudoku, but instead of using the numbers 1-9, the 16x16 grid uses the numbers 1-16. This means each row, column, and 4x4 region must contain each number and letter exactly once. There are many algorithms used to solve the sudoku. This implementation uses Dancing Links implementation of Algorithm X to efficiently solve Sudoku puzzles. And also, it uses backtracking to search solution space. This report presents optimized solution to Sudoku puzzles. It offers the details of the method, the problems we encountered, and suggests ways to make it even better.

BACKGROUND AND CONSTRAINTS

This Sudoku solver solves the sudoku by converting the sudoku problem into an extract cover problem. Then it solves that exact cover problem using the Algorithm X. This dancing links with Algorithm X is developed by Donald Knuth. Exact cover problem involves selecting a combination of sets such that each element appears in exactly one selected set, while no set overlaps with others in the chosen subset.

Sudoku problem can be converted to exact cover problem by representing the constrains of sudoku in a sparse matrix. Sudoku have four constraints. A position constraint: Only 1 number can occupy a cell, A row constraint: Only 1 instance of a number can be in the row, A column constraint: Only 1 instance of a number can be in a column, A region constraint: Only 1 instance of a number can be in a region.

When considering 9x9 sudoku, it has 81 cells. So, it has 81 position constraints. It has 9 rows and every row must have 9 values in it. So, it has 81 row constraints. Like this, it has 9 columns. So, it has 81 column constraints. It has 9 boxes and every box must have 9 values in it. So, it has 81 region constraints. So total 81+81+81+81 = 324 constraints. It is equivalent to SIZE x 4. In this 9x9, there are 9 rows, 9 columns, and 9 values in a Sudoku puzzle. So, it has 9*9*9 = 729 cell position with value combinations. Constraints are represented by columns of sparse matrix and cell position with value combinations are represented by rows of sparse matrix. Dancing links employs a doubly linked list data structure to represent the constraints. And also, this dancing list is a toroidal linked list.

IMPLEMENTATION

As per the project guidelines, program is developed to accept command line arguments. Program will use the command line argument as file name and read that file. It will push the read number into a 2D vector called sudoku grid which is a global vector without a predefined size. While reading the file program updates the integer variable SIZE to calculate the sudoku grid size. After reading the sudoku file, program initializes the values needed for upcoming calculations. They are row size of sparse matrix, column size of sparse matrix, sqrt of size and size squared.

Then it creates a dynamically allocated sparse matrix to represent constrains of sudoku to convert it as an exact cover problem. Then it calls the create sparse matrix and that function iterates through the sparse matrix and sets the corresponding position of matrix as 1 to indicate the cell constrains, row constrains, column constrains and box constrains.

Then it calls buildTorodiallinkedlist function to build toroidal linked list which will used to apply Algorithm X. program uses a struct Node to represent nodes in the linked list. That node has left, right, top, down node pointer attributes to navigate neighbor nodes. It has head node pointer attribute to find the corresponding head of the node. And also, it has size attribute to represent the number of uncovered nodes in the column. This size attribute is only used and updated by head nodes. Also, node struct has three element rowID array to represent nodes position. It indicates the value, row of sudoku grid, col of sudoku grid.

Initially buildTorodiallinkedlist creates a node pointer for master head node. Master node will be used to start the algorithm X and end the search of solution. And assigns if self to all of its neighbors and head. And also assigns size as -1 to indicate it does not hold any nodes at any time. Then it creates the nodes for all of the columns on the right side of the master node. Size of the nodes are set to zero. Because still these head column nodes are not holding any nodes. At the end every node is doubly linked with their rigtht, left, top, down nodes. Here top-down nodes of the respective nodes are itself. Then it iterates every row, calculates the rowID of the nodes and iterate each colums. When iterating every colums, it creates nodes for the positions where sparse matrix has 1. It assigns the calculated row id to the node. At the last all nodes in linked lists are connect with their neighbors and forms torodial linked list.

After creating toroidal linked list, it calls InitializeSudokuInTorodialLinkedlist function to represent the initial state of sudoku in the Torodiallinkedlist. This function iterates the filled positions of sudoku, finds the corresponding row in the Torodiallinkedlist, and cover all the colums of the nodes in that row. Then it calls the search function by passing an integer zero. It will used as index to store the selected nodes in the solution grid. This function uses two utility function. Thay are cover column, uncover column. Cover column updates the pointers of neighbor header nodes to bypass the covered column. Also cover column function iterates nodes in every row of that column and find the other nodes in the rows and vertically unlink that node and reduces the no of nodes size in the header node.

Covering a column efficiently removes the entire column from further consideration. Uncover function is the inverse of the cover function. In cover function, vertical links are only unlinked. Still, they have horizontal links. Using that horizontal links, it does the undo operation. It is used for backtracking.

At the starting of the function, it checks whether master head has any nodes in the right side. if master node's right-side node is master node, it has no right-side nodes. So, sudoku is solved and function starts to map the solution to the sudoku grid. If master node has right nodes, it selects the right node of master column, iterates all columns in the right hand of that column and find the column which has minimum no of nodes. Then it covers that column. Then it iterates every node under that column. For each node it put the node into the solution array and cover other nodes which are in the row of that node. Then it makes a recursive call to the search function with incremented index for solution grid. When a call ends and it does not solve, it again sets the element of corresponding solution index as null and uncover the previously covered columns.

At last, MapSolutionToGrid function, iterates solution array and finds the corresponding index in the sudoku grid and put the value in that position. And also, program writes the solved array to output_filename.txt file. If solution is not found, it writes No solution to the file.

OPTIMIZATIONS

These are the optimizations used in the implementation of the sudoku solver.

DLX's internal optimization techniques:

DLX algorithm uses Constraint Propagation in conjunction with backtracking. It performs constraint propagation by covering columns and updating the linked list structure at each step. This process helps eliminate possibilities that violate the rules of Sudoku, narrowing down the search space. Backtracking avoids unnecessary computations and explores only the needed paths in the solution space.

Minimum Remaining Values (MRV) Heuristic:

The algorithm uses the Minimum Remaining Values (MRV) heuristic to prioritize columns with the fewest remaining nodes needs to be explored in a particular column. This heuristic minimizes the search space at each step of the backtracking process. It results quicker convergence to a valid solution.

Dynamic Memory Allocation for 2D Array:

The code dynamically allocates a 2D array (bool** matrix). It gives faster access time because arrays are stored in the contiguous memory.

Direct Access Using References:

The code uses references to access Boolean values in the 2D sparse matrix array. By using references, program achieves direct access to the elements. It gives fast access times and results a reduced solver time.

Early Termination:

This algorithm can find multiple solutions for a sudoku, if sudoku has multiple solutions. But implemented algorithm stops the searching of solution when a solution is found. It reduces the solver time by eliminating unwanted searching.

CHALLENGES FACED

1. Algorithmic Complexity: The Dancing Links algorithm is a complex algorithm, so implementation required a deeper understanding of that algorithm and linked list data structure.
2. Parallelization Efficiency: Parallelization is applied to search function to increase the speed of the solver. But it does not give any difference in solver time. Then it is removed.
3. Dynamic Size of puzzles: Fixed sudoku size solver gives a lower solver time than dynamically size allocated solver. It is due to the memory allocation overhead, cache inefficiency due the dynamic allocation.

LIMITATIONS

1. Memory Usage: The algorithm creates a toroidal doubly linked list to represent the constraints, which can lead to high memory consumption for large puzzles. This may limit the size of puzzles that can be effectively solved within available memory.

FUTURE IMPROVEMENTS

1. Optimizing Data Structures: Explore more efficient data structures for representing the Sudoku puzzle constraints.
2. Performance optimization: Investigating more advanced heuristics to achieve faster solving times.
3. Memory Optimization: Investigate ways to optimize memory usage, especially for large puzzles, by dynamically allocating and deallocating memory as needed during the search.
4. Parallel Backtracking: Implementing parallel backtracking techniques to explore different branches of the search space concurrently, improving the algorithm's efficiency.
5. Multiple Solution Implementation: DLX algorithm has the capability to find all solution. Investigating ways to implement this with a reduced solver time.

This solver solves 9*9 puzzle in 1-3 milli seconds and 16*16 puzzle in 29 - 33 milliseconds.