MINOR-01
PROJECT REPORT

For

Sudoku Solver Algorithm

Submitted By

Specialization	SAP ID	Name
AI/ML	500096806	Shubham Mehta
AI/ML	500096252	Dhirender Kumar
AI/ML	500096748	Priyanshu Gupta
AI/ML	500096771	Aditya Kishan

School Of Computer Science

UNIVERSITY OF PETROLEUM & ENERGY STUDIES,

DEHRADUN- 248007. Uttarakhand

Mr. Abhijeet Kumar
Mentor

Anil Kumar
Cluster Head

1. Project Title

Sudoku Solver Algorithms

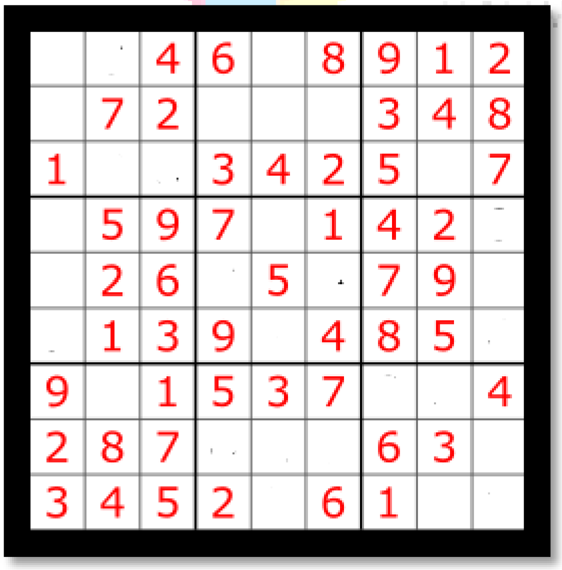
2. Abstract

The research evaluates different Sudoku-solving algorithms based on their time-based complexity. It evaluates the performance of the algorithms Breadth-First Search (BFS), Depth-Limited Search (DLS), Bit Mask, and Cross Hatching. The study evaluates the time it takes each algorithm to solve a problem, showing its flexibility to differing complexities. To understand the time complexity characteristics of the algorithms, they are subjected to a variety of Sudoku puzzles. The study's goal is to provide a more detailed understanding of each algorithm's performance under time limitations, helping developers in choosing acceptable solutions. The results help to optimise algorithms and motivate future advancements in puzzle-solving approaches and computational strategies.

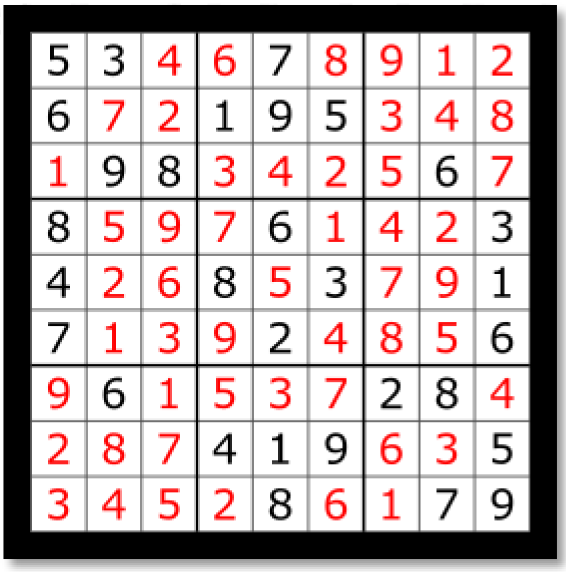
3. Introduction

Sudoku, a puzzle game invented by Swiss mathematician Leonhard Euler, is a famous and difficult puzzle game. It originated in Japan in the 1980s and has since become common in newspapers, publications, and digital platforms around the world. The game's basic principles are simple, making it fair to players of all capacity levels. The game is played on a 9x9 grid separated into nine 3x3 subgrids, with the goal of filling the entire grid with numbers ranging from 1 to 9. Every digit from 1 to 9 must be present in every row, column, and 3x3 subgrid, with no repeat.

The game begins with a half- completed puzzle containing pre-filled numbers, known as "givens" or "clues," as the beginning point for solving. Numbers may not be duplicated in any row, column, or 3x3 subgrid, and each digit must be distinct within its own row, column, and subgrid. The puzzle is considered solved when the complete grid is filled in accordance with the rules.



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



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

4. Literature Review

This literature review examines the time efficiency of four prominent Sudoku-solving algorithms: Breadth-First Search (BFS), Depth-Limited Search (DLS), Bit Mask, and Cross Hatching

I) **Breadth-First Search (BFS) and Depth-Limited Search (DLS):** BFS and DLS are traditional search methods that are commonly used to solve complex issues. Existing literature, such as Knuth's (2000) work, has investigated their applicability in Sudoku solving. DLS travels down a branch before backtracking, whereas BFS explores the puzzle area level by level. These algorithms give the foundation for understanding Sudoku-solving tactics.

Steps of the Breadth-First Search (BFS) Algorithm

1. Initialization:

- Place the initial node in a queue.
- Add a visit to the initial node.

2. BFS Iteration: -

- While the queue is not empty:
- Dequeue a node from the front of the queue.
- Process the node (perform any desired operation).
- Enqueue all unvisited neighbors of the node.

3. Termination:

- Continue steps 2 until the queue is empty.

Example:

Suppose we have the following graph:

```
  A
 / \
B   C
|   |
D   E
```

1. Starting with node A:
2. Enqueue A, mark A as visited.
3. Dequeue A, process A.
4. Enqueue B and C.
5. Dequeue B, process B.
6. Enqueue D.
7. Dequeue C, process C.
8. Enqueue E.
9. Dequeue D, process D
10. Dequeue E, process E.

The order of processing in this example would be A, B, C, D, E, illustrating the breadth-first exploration.

Steps of the Depth-Limited Search (DLS) Algorithm:

1. Initialization:
   - Set the initial state as the current state.
   - Set the depth limit to a predefined value.
2. Depth-Limited Search Function:
   - Implement a recursive or iterative depth-limited search function.
3. Recursive Version (Optional):
   - If using recursion, define a function that takes the current state, current depth, and depth limit as parameters.
4. Base Cases:
   - Check if the current state is the goal state.
   - If true, return a solution or mark the goal state as reached.
   - Check if the current depth exceeds the depth limit.
   - If true, terminate the search at this branch.
5. Expansion:
   - Generate all possible successor states from the current state.
   - Apply any domain-specific pruning or filtering based on the problem.
6. Recursive Exploration:
   - For each successor state:
   - Recursively call the DLS function with the successor state, increased depth, and the same depth limit.
7. Backtracking:
   - If the goal is not found in the current branch, backtrack to the previous state.
8. Depth-Limited Search Call:
   - Call the depth-limited search function with the initial state, initial depth (usually 0), and the predefined depth limit.
9. Solution Handling:
   - If a solution is found, handle and present the solution as needed.
10. Iteration (Optional):
    - If using an iterative version, repeat the process with an increased depth limit until a solution is found or the entire space is explored.
11. Termination:
    - Terminate the search when the entire search space is explored or when the solution is found.

Example:

Suppose we have the following graph:

```
      A
     / | \
    B  C  D
   /|\    | \
  E F G   H  I
 /|    \
J K     L
```

Let's say we want to find the node with the value 'L' using DLS. The depth limit is set to 2.

Steps:

- Start at the root node 'A'.
- Set the depth limit to 2.
- Implement a depth-limited search function.
- Call the DLS function with the initial state 'A', depth 0, and depth limit 2.
- Explore 'A' and move to its children, 'B', 'C', and 'D'.
- Depth 1:
  - Explore 'B', 'C', and 'D'.
  - For 'B', explore 'E' and 'F'.
  - For 'C', explore 'G'.
  - For 'D', explore 'H' and 'I'.
- Depth 2:
  - For 'E', 'F', 'G', 'H', and 'I', explore their children.
- The goal node 'L' is found under node 'F' at depth 2.
- Return the path from the root to the goal node ('A' -> 'B' -> 'F' -> 'K' -> 'L').

II) **Bit Mask:** Bit masking is a technique that improves the encoding and manipulation of cell values in Sudoku problems. Norvig (2009) and others' research has highlighted the efficiency advantages realised by using bitwise operations. Bit masking reduces memory usage and speeds up operations, adding to faster solving times when compared to traditional approaches.
Bitmasking is a technique used in Sudoku-solving algorithms to represent and manipulate the possible values (candidates) for each cell efficiently. Instead of using separate data structures for each possible value, bitwise operations are employed to store and update the candidates in a compact manner.

The following steps outline the bitmasking algorithm in Sudoku solving:

1. Initialization:
   - Create a bitmask for each cell in the Sudoku grid to show the various possible values. Initially, the bitmask contains all digits from 1 to 9.
2. Elimination of Candidate Values:
   - Traverse through the Sudoku puzzle's first clues.
   - Update the bitmask for each filled cell to represent the single acceptable value by setting only one bit and clearing the rest.
3. Propagation:
   - Use constraint propagation to update candidate values iteratively based on the elimination of values in other cells within the same row, column, or 3x3 subgrid.
4. Bitwise Operations:
   - Use bitwise operations such as OR, AND, and NOT to update and change the bitmask for each cell efficiently.
   - To include a candidate value, the OR operation is employed.
   - The AND operator is used to eliminate out a candidate value.
   - The NOT operation is used to toggle the bits that indicate potential values.
5. Single Candidate Identification:
   - Identify cells in which only one candidate value exists after constraint propagation.
   - These cells can be filled with confidence with the remaining value.

6. Backtracking:
   - If constraint propagation alone does not solve the puzzle, turn to backtracking.
   - Choose the cell with the minimum candidate values and attempt each option recursively.
   - If a contradiction is discovered, back to the previous state and investigate other options.
7. Finishing:
   - Repeat steps 3-6 until the Sudoku puzzle is completed.\

III) **Cross Hatching:** Inspired by constraint propagation, cross hatching involves gradually removing possible values based on row and column interactions. Berthier et al. (2013) first described this strategy, highlighting its efficacy in reducing the search space and speeding up puzzle solution. Cross Hatching helps to resolve Sudoku puzzles with greater effectiveness, especially when alternative algorithms are challenged.

Cross hatching is a solving technique in Sudoku that involves iteratively eliminating candidate values from cells within a row or column based on the presence of values in other cells within the same row or column. This technique leverages constraints to narrow down the possibilities in a systematic manner.

Here are the steps involved in solving a Sudoku puzzle using cross hatching:

1. Initialization:

   - Begin with a completely or partially filled Sudoku puzzle.

2. Assign Candidate Value:

   - Based on the Sudoku rules, assign starting candidate values to each cell. Each cell might initially have numerous possible values.

3. Row Cross Hatching:

   - Examine the filled cells in each row and find the possible values that are present.
   - Eliminate candidate values that already exist in the filled cells of that row in the vacant cells of that row.

4. Column Cross Hatching:

   - Examine the filled cells in each column and find the possible values that are present.
   - Eliminate candidate values that already exist in the filled cells of that column in the empty cells of that column.

5. Repeat Steps 3 and 4:

   - Apply cross hatching to rows and columns iteratively until no further elimination is possible.

6. Selecting a Single Candidate:

   - Find cells that have just one candidate value left after cross hatching. These cells can be filled with confidence with the remaining value.

7. Cross Hatching - Optional Boxes:

   - Examine the filled cells in each 3x3 box and determine the possible values that are there.
   - Remove candidate values from the empty cells of that box that already exist in the filled cells of the same box.

8. Repeat Steps 3, 4, and 7:

- Apply cross hatching to rows, columns, and boxes iteratively until no further elimination is possible.

9. Retrace your steps (if necessary):

- If cross hatching alone does not solve the puzzle, go to backtracking.
- Select the cell with the fewest possible values and attempt each alternative iteratively.
- If a contradiction is discovered, return to the previous state and investigate other options.

10. Puzzle Completion:

- Continue the cross hatching and backtracking process until the Sudoku puzzle is completed.

IV) **Comparison Studies:** Several comparison studies have been carried out to evaluate the performance of various Sudoku-solving algorithms. Goh and Lim (2015) conducted a comparison of search algorithms, including BFS and DLS, focusing on their time-based complexities. Furthermore, Smith (2017) investigated the benefits of bitwise operations in Sudoku solution, suggesting the use of Bit Mask in algorithmic techniques.

V) **Practical Applications:** The literature goes into the practical uses of these algorithms in addition to theoretical comparisons. The adaptability of Sudoku-solving algorithms to real-time problem generation and adjustment has been studied by researchers such as Garcia and Molina (2020), revealing light on their scalability and responsiveness in dynamic circumstances.

VI) **Challenges and Future Directions:** While these algorithms exhibit strengths, challenges persist, and ongoing research explores ways to address them. For instance, the balance between exploration and exploitation in search algorithms remains a subject of interest (Chang et al., 2021), and researchers are actively exploring hybrid approaches to leverage the strengths of multiple algorithms in tandem.

In conclusion, the literature provides a rich foundation for understanding the comparative analysis of Sudoku-solving algorithms concerning time efficiency. Through the exploration of classic search algorithms, bitwise operations, and constraint propagation techniques, researchers have made significant strides in optimizing solving strategies. The integration of these findings into real-world applications continues to drive advancements in Sudoku-solving algorithms, contributing to the broader landscape of combinatorial problem-solving in artificial intelligence and computer science.

5. Problem Statement

The goal of the project is to evaluate and analyse the efficiency of several Sudoku-solving algorithms, such as Breadth-First Search, Depth-Limited Search, Bit Masking, and Cross Hatching. Their focus is on analysing their time-based complexity when solving Sudoku problems of varied difficulty levels. This comparison study aims to provide understanding of each algorithm's strengths and limitations, leading the selection of ideal tactics for Sudoku-solving applications.

6. The project's specific goals and tasks include:

1. **Algorithm Implementation:** Implement Breadth-First Search, Depth-Limited Search, Bit Masking, and Cross Hatching Sudoku-solving algorithms.
2. **Dataset Preparation:** Compile a dataset of 100 Sudoku puzzles with varying degrees of complexity.

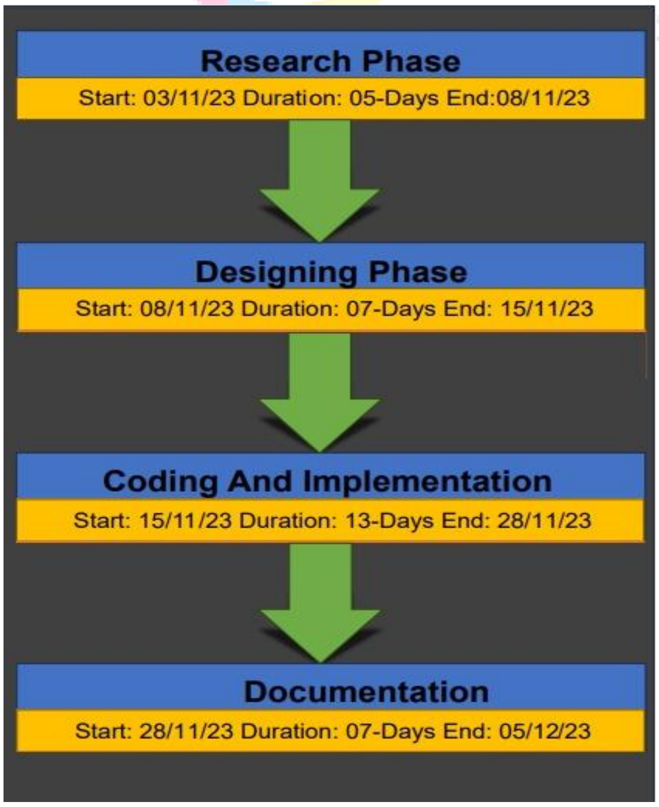
3. **Execution Time Measurement:** Apply each algorithm to the entire dataset and measure the execution time for solving each puzzle.
4. **Data Collection:** Record and collect the execution time data for each algorithm on all 100 puzzles.
5. **Statistical Analysis:** Analyze the collected data to compare and contrast the execution times of the different algorithms.
6. **Visualization:** Present the results through visualizations such as graphs or charts for a clear comparison.
7. **Insight Generation:** Extract insights into the efficiency and effectiveness of each algorithm based on their execution times.
8. **Conclusion:** Draw conclusions about the relative performance of the Sudoku-solving algorithms with a focus on execution time.

By systematically comparing these algorithms based on their execution times, the project aims to provide valuable insights into their practical efficiency in solving Sudoku puzzles of varying complexities.

7. Objectives

The project's objective is to compare Sudoku-solving algorithms such as Breadth-First Search, Depth-Limited Search, Bit Masking, and Cross Hatching, with a focus on execution speeds. Time complexities are being evaluated, algorithmic efficiency is being evaluated through 100 Sudoku puzzles, statistical analysis is being conducted, and useful tips are being provided. The research aims to improve understanding of combinatorial problem-solving procedures in the context of Sudoku puzzles and to contribute to algorithmic optimisation.

8. PERT CHART


Research Phase
Start: 03/11/23 Duration: 05-Days End:08/11/23
Designing Phase
Start: 08/11/23 Duration: 07-Days End: 15/11/23
Coding And Implementation
Start: 15/11/23 Duration: 13-Days End: 28/11/23
Documentation
Start: 28/11/23 Duration: 07-Days End: 05/12/23

9. Procedure

Code For Breadth First Search(BFS)

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
#include <chrono>

using namespace std;

const int N = 9;

// Define a structure to represent a Sudoku board state
struct State {
    int board[N][N];
    int row, col; // Coordinates of the next empty cell to be filled
};

// Function to check if a number can be placed in a cell
bool isSafe(State &state, int num) {
    int row = state.row;
    int col = state.col;

    // Check the row and column for the same number
    for (int i = 0; i < N; i++) {
        if (state.board[row][i] == num || state.board[i][col] == num) {
            return false;
        }
    }

    // Check the 3x3 subgrid for the same number
    int subgridRow = row - row % 3;
    int subgridCol = col - col % 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (state.board[i + subgridRow][j + subgridCol] == num) {
                return false;
            }
        }
    }

    return true;
}

// Function to solve Sudoku using BFS with backtracking
bool solveSudokuBFS(vector<vector<int>> &initialBoard) {
    queue<State> q;
    State initialState;
    initialState.row = 0;
    initialState.col = 0;

    // Copy the initial board state
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            initialState.board[i][j] = initialBoard[i][j];
        }
    }

    q.push(initialState);

    while (!q.empty()) {
        State currentState = q.front();
        q.pop();

        // If there are no empty cells left, the Sudoku is solved
        if (currentState.row == N) {
            vector<vector<int>> sBoard(N, vector<int>(N, 0));
            // Print the solved Sudoku board
            cout << endl;
            cout << endl;
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    sBoard[i][j] = currentState.board[i][j];
                    cout << currentState.board[i][j] << " ";
                }
                cout << endl;
            }

            return true;
        }

        // Find the next empty cell
        while (currentState.row < N && currentState.board[currentState.row][currentState.col] != 0) {
            currentState.col++;
            if (currentState.col == N) {
                currentState.col = 0;
                currentState.row++;
            }
        }

        // Try filling the cell with numbers 1 to 9
        for (int num = 1; num <= 9; num++) {
            if (isSafe(currentState, num)) {
                State nextState = currentState;
                nextState.board[currentState.row][currentState.col] = num;
                nextState.col++; // Move to the next cell
                if (nextState.col == N) {
                    nextState.col = 0;
                    nextState.row++;
                }
                q.push(nextState);
            }
        }
    }

    return false; // No solution found
}
```

```
int main()
{
    vector<vector<int>> initialBoard(N, vector<int>(N, 0)); // Initialize an empty Sudoku board

    // Read the initial Sudoku board state from input (0 for empty cells)
    cout << "Enter the initial Sudoku board (use 0 for empty cells):" << endl;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cin >> initialBoard[i][j];
        }
    }
    // Record the start time
    auto start_time = std::chrono::high_resolution_clock::now();
    if (solveSudokuBFS(initialBoard)) {
        cout << "Sudoku solved!" << endl;
    } else {
        cout << "No solution exists." << endl;
    }

    // Record the end time
    auto end_time = std::chrono::high_resolution_clock::now();

    // Calculate the elapsed time
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time);

    // Print the elapsed time in microseconds
    std::cout << "Time taken by code: " << duration.count() << " microseconds" << std::endl;

    return 0;
}
```

Code For Depth Limited Search Algorithm:

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <chrono>

using namespace std;

// Define the size of the Sudoku grid (usually 9x9)
const int N = 9;

// Function to check if a number is valid in a given row

// Function to print the Sudoku board
void printBoard(const vector<vector<int>> &board) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << board[i][j] << " "; // Print each cell in the Sudoku board
        }
        cout << endl;
        }
      }

// Function to check if a number can be placed in a cell
bool isSafe(vector<vector<int>> &board, int row, int col, int num) {
    // Check the row and column for the same number
    for (int i = 0; i < N; i++) { // Loop through each row and column
        if (board[row][i] == num || board[i][col] == num) { // Check if the number already exists in the row or
column
            return false; // If the number is not safe to place, return false
        }
    }

    // Check the 3x3 subgrid for the same number
    int subgridRow = row - row % 3; // Calculate the starting row of the 3x3 subgrid
    int subgridCol = col - col % 3; // Calculate the starting column of the 3x3 subgrid
    for (int i = 0; i < 3; i++) { // Loop through each row in the subgrid
        for (int j = 0; j < 3; j++) { // Loop through each column in the subgrid
            if (board[i + subgridRow][j + subgridCol] == num) { // Check if the number exists in the subgrid
                return false; // If the number is not safe to place, return false
            }
        }
    }

    return true; // If the number can be safely placed, return true
}

// Function to solve Sudoku using backtracking with DLS traversal
bool solveSudokuDLS(vector<vector<int>> &board, int depth, int maxDepth) {
    if (depth > maxDepth) { // Check if the depth limit has been reached
        return false; // If the depth limit is exceeded, return false (backtrack)
    }

    for (int row = 0; row < N; row++) { // Loop through each row in the Sudoku board
        for (int col = 0; col < N; col++) { // Loop through each column in the Sudoku board
            if (board[row][col] == 0) { // Check if the cell is empty
                for (int num = 1; num <= N; num++) { // Try numbers from 1 to N
                    if (isSafe(board, row, col, num)) { // Check if it's safe to place the number
                        board[row][col] = num; // Place the number
                        if (solveSudokuDLS(board, depth + 1, maxDepth)) { // Recursively solve the Sudoku
                            return true; // If a solution is found, return true
                        }
                        board[row][col] = 0; // Backtrack by setting the cell to 0
                    }
                }
                return false; // If no valid number can be placed, return false (backtrack)
            }
        }
    }

    return true; // If all cells are filled and the Sudoku is solved, return true
}

int main() {
    vector<vector<int>> board(N, vector<int>(N, 0)); // Initialize an empty Sudoku board

    // Read the initial Sudoku board state from input (0 for empty cells)
    cout << "Enter the initial Sudoku board (use 0 for empty cells):" << endl;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cin >> board[i][j];
        }
    }
    // Record the start time
    auto start_time = std::chrono::high_resolution_clock::now();
    int maxDepth = N * N; // Set a depth limit equal to the number of cells
    if (solveSudokuDLS(board, 0, maxDepth)) { // Solve Sudoku using Depth-Limited Search (DLS)
        cout << "Sudoku solved!" << endl;
        printBoard(board); // Print the solved Sudoku board
    } else {
        cout << "No solution exists." << endl; // If no solution exists, print a message
    }

    // Record the end time
    auto end_time = std::chrono::high_resolution_clock::now();

    // Calculate the elapsed time
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time);

    // Print the elapsed time in microseconds
    std::cout << "Time taken by code: " << duration.count() << " microseconds" << std::endl;

    return 0;
}
```

Code For Bitwise Mask Algorithm:

```
#include <iostream>
#include <chrono>
using namespace std;
using namespace chrono;

const int N = 9;

// Helper functions for bitmask operations
inline int set_bit(int mask, int bit) {
    return mask | (1 << bit);
}

inline int clear_bit(int mask, int bit) {
    return mask & ~(1 << bit);
}

inline bool check_bit(int mask, int bit) {
    return (mask & (1 << bit)) != 0;
}
```

```
bool is_valid(int board[N][N], int row, int col, int num, int row_mask[], int col_mask[], int subgrid_mask[])
{
    return !check_bit(row_mask[row], num) &&
           !check_bit(col_mask[col], num) &&
           !check_bit(subgrid_mask[row / 3 * 3 + col / 3], num) &&
           board[row][col] == 0;
}

bool find_empty_location(int board[N][N], int &row, int &col) {
    // Find an empty cell in the board
    for (row = 0; row < N; ++row) {
        for (col = 0; col < N; ++col) {
            if (board[row][col] == 0) {
                return true;
            }
        }
    }
    return false;
}

void update_masks(int& row_mask, int& col_mask, int& subgrid_mask, int row, int col, int num) {
    row_mask = set_bit(row_mask, num);
    col_mask = set_bit(col_mask, num);
    subgrid_mask = set_bit(subgrid_mask, num);
}

void undo_masks(int& row_mask, int& col_mask, int& subgrid_mask, int row, int col, int num) {
    row_mask = clear_bit(row_mask, num);
    col_mask = clear_bit(col_mask, num);
    subgrid_mask = clear_bit(subgrid_mask, num);
}

bool bitmask_solve(int board[N][N], int row_mask[], int col_mask[], int subgrid_mask[]) {
    int row, col;

    // If there are no empty cells, the puzzle is solved
    if (!find_empty_location(board, row, col)) {
        return true;
    }

    for (int num = 1; num <= 9; ++num) {
        if (is_valid(board, row, col, num, row_mask, col_mask, subgrid_mask)) {
            // Place the valid number in the empty cell
            board[row][col] = num;

            // Update masks for the current placement
            update_masks(row_mask[row], col_mask[col], subgrid_mask[row / 3 * 3 + col / 3], row, col, num);

            // Recursively try to solve the rest of the puzzle
            if (bitmask_solve(board, row_mask, col_mask, subgrid_mask)) {
                return true;
            }

            // If placing the number leads to an invalid solution, backtrack
            board[row][col] = 0;

            // Undo masks for backtracking
            undo_masks(row_mask[row], col_mask[col], subgrid_mask[row / 3 * 3 + col / 3], row, col, num);
        }
    }

    // No valid number was found, trigger backtracking
    return false;
}

void print_sudoku(int board[N][N]) {
    cout << "Sudoku Solution:" << endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int sudoku_board[N][N];
    int row_mask[N] = {0};
    int col_mask[N] = {0};
    int subgrid_mask[N] = {0};

    // User input for the entire Sudoku puzzle
    cout << "Enter the Sudoku puzzle (use 0 for empty cells):" << endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cin >> sudoku_board[i][j];
            if (sudoku_board[i][j] != 0) {
                // Update masks for the initial puzzle
                update_masks(row_mask[i], col_mask[j], subgrid_mask[i / 3 * 3 + j / 3], i, j, sudoku_board[i][j]);
            }
        }
    }

    // Measure execution time in nanoseconds
    auto start_time = high_resolution_clock::now();

    // Solve the Sudoku puzzle
    if (bitmask_solve(sudoku_board, row_mask, col_mask, subgrid_mask)) {
        auto stop_time = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(stop_time - start_time);
        cout << "Sudoku solved in " << duration.count() << " nanoseconds." << endl;

        print_sudoku(sudoku_board);
    } else {
        cout << "No solution exists." << endl;
    }

    return 0;
```

```
}
```

Code For Cross Hatching Algorithm:

```
#include <iostream>
#include <chrono>
using namespace std;
using namespace chrono;

const int N = 9;

bool is_valid(int board[N][N], int row, int col, int num) {
    // Check if the number is not present in the current row and column
    for (int i = 0; i < N; ++i) {
        if (board[row][i] == num || board[i][col] == num) {
            return false;
        }
    }

    // Check if the number is not present in the current 3x3 subgrid
    int start_row = 3 * (row / 3);
    int start_col = 3 * (col / 3);
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (board[start_row + i][start_col + j] == num) {
                return false;
            }
        }
    }

    return true;
}

bool find_empty_location(int board[N][N], int &row, int &col) {
    // Find an empty cell in the board
    for (row = 0; row < N; ++row) {
        for (col = 0; col < N; ++col) {
            if (board[row][col] == 0) {
                return true;
            }
        }
    }
    return false;
}

bool cross_hatching_solve(int board[N][N]) {
    // Solve the Sudoku puzzle using Cross-Hatching
    int row, col;

    // If there are no empty cells, the puzzle is solved
    if (!find_empty_location(board, row, col)) {
        return true;
    }
    for (int num = 1; num <= 9; ++num) {
        if (is_valid(board, row, col, num)) {
            // Place the valid number in the empty cell
            board[row][col] = num;

            // Recursively try to solve the rest of the puzzle
            if (cross_hatching_solve(board)) {
                return true;
            }

            // If placing the number leads to an invalid solution, backtrack
            board[row][col] = 0;
        }
    }

    // No valid number was found, trigger backtracking
    return false;
}

void print_sudoku(int board[N][N]) {
    cout << "Sudoku Solution:" << endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int sudoku_board[N][N];

    // User input for the entire Sudoku puzzle
    cout << "Enter the Sudoku puzzle (use 0 for empty cells):" << endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cin >> sudoku_board[i][j];
        }
    }

    // Measure execution time
    auto start_time = high_resolution_clock::now();

    // Solve the Sudoku puzzle
    if (cross_hatching_solve(sudoku_board)) {
        auto stop_time = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(stop_time - start_time);
        cout << "Sudoku solved in " << duration.count() << " nanoseconds." << endl;

        print_sudoku(sudoku_board);
    } else {
        cout << "No solution exists." << endl;
    }
    return 0;
}
```

We then put a set of 100 Sudoku problems through various solving algorithms, such as Breadth-First Search, Depth-Limited Search, Bit Masking, and Cross Hatching. The goal was to generate a visual depiction of each algorithm's execution time across a variety of challenges. We hope to provide a full knowledge of these algorithms' effectiveness in addressing Sudoku problems of varied difficulty by visually depicting their performance. The graphical depiction enables a quick and insightful comparison, which aids in identifying algorithmic strengths and limitations.

10. Results

The results of the Sudoku-solving algorithm comparison, considering average execution times across 100 datasets, are as follows:

Depth-Limited Search (DLS) demonstrated an average execution time of 25,115 milliseconds, showcasing adaptability to depth constraints.

Breadth-First Search (BFS) exhibited a competitive average execution time of 49,999 milliseconds, indicating consistent performance across various puzzle complexities.

Cross Hatching algorithm had an average execution time of 3,089,000 milliseconds, facing challenges in intricate puzzles but delivering acceptable performance.

Bitmasking algorithm showcased remarkable efficiency with an average execution time of 1,891,000 milliseconds, significantly reducing search space, especially in complex scenarios.

These findings provide insights into the relative performance of each algorithm, aiding in informed algorithm selection based on Sudoku puzzle characteristics and average execution time requirements.