Sudoku Solver By Backtracking

Yasmina Mahdy 900221083

Janna Osama 900225228

Ousswa Chouchane 900225937

Youssef Badawy900214672

CSCE2211 - Section 1

Department of Computer Science and Engineering, AUC

Abstract

Our project aims to demonstrate how the backtracking algorithm can be used to solve Sudoku puzzles. In this paper we first explain the rules of sudoku, followed by detailing our implementation of the backtracking algorithm. Moreover, since our project also includes a “play mode” that simulates a real sudoku puzzle application, we also explain how we were able to implement such features in our program. Next, we provide specimens of our graphical output as well as some experimentational results, and finally we provide an analysis of our algorithms and their complexities, and compare them to Rule-Based and Boltzmann Machines algorithms for solving sudoku puzzles.

keywords: Backtracking algorithm, User-friendly interface, Sudoku, Puzzle solving, Unique board generation, Rule-based algorithm, Exponential complexity, Constraint techniques, QT framework, and Constraint techniques.

Introduction

Sudoku, an extremely popular and beloved number-placement game, is a fascinating example of how an old game can be the subject of interesting algorithms in the domain of computer science. Originating in Japan, the term "Sudoku" harmonizes "su" (number) and "doku" (single), capturing the nature of a game played within a 9x9 grid and divided into nine 3x3 subgrids. This paper dives into one of the algorithms commonly used to solve sudoku puzzles, namely the backtracking algorithm. We also provide our own interpretation of what a sudoku application should be like by adding a feature that challenges the user by presenting them with incomplete boards for them to solve.Moreover, we discuss the limitations of the backtracking algorithm, and how other algorithms that can be used to solve sudoku puzzles compare with it (Sudoku Rules for Complete Beginners | Play Free Sudoku, a Popular Online Puzzle Game, n.d.).

Problem Definition

Sudoku, at its core, challenges players to fill a 9x9 grid with numbers 1 through 9 in such a way that each row, column, and 3x3 subgrid contains unique digits (Sudoku Rules for Complete Beginners | Play Free Sudoku, a Popular Online Puzzle Game, n.d.). This research project goes a step further, aiming to create a program using the QT framework that provides a user-friendly interface for Sudoku enthusiasts. The program boasts two distinctive modes: one where users input a puzzle for the system to solve, and the other where the program generates unique puzzles for users to solve.

In tackling the rules of Sudoku, we scrutinize the intricacies of creating a robust program that not only understands the logic behind solving Sudoku puzzles but also engages users effectively. Leveraging the QT framework, known for its user interface capabilities, we aim to offer an interactive experience. The user interface will facilitate puzzle input, making it accessible for users to submit puzzles for the program to solve, while also providing an engaging platform for users to solve generated puzzles. By exploring this avenue, the project seeks to bridge the gap between Sudoku as a recreational activity and a captivating programming challenge. The duality of the program's modes broadens its utility, catering to both puzzle creators and avid solvers, thereby enriching the overall Sudoku experience.

Methodology

SUDOKU SOLVER

The primary method utilized to solve the sudoku board is backtracking. In simple terms, it works by attempting to place numbers from 1 to 9 in empty cells of the puzzle while ensuring that the placement follows Sudoku rules.

This is accomplished through the following functions:

- isValid Function: Checks whether placing a number in a specific cell is valid based on Sudoku rules (no repeated numbers in rows, columns, or 3x3 subgrids).
- findEmptyPosition Function: Finds the next empty position (cell with value 0) on the board.

- solveBoard Function: Uses backtracking to solve the Sudoku puzzle, where it finds an empty cell, attempts to place a number in it, and recursively solves the board until a solution is found or the board is deemed unsolvable.
- SudokuBoard Class: This class contains methods to initialize the board, solve the Sudoku puzzle, and print the solved board.

Now, let us go through how the solving program works step by step:

The program is initialized by creating a 9x9 matrix (sudoku object) filled with some numbers provided by the user. Then comes the role of the backtracking algorithm. The solveBoard function works recursively by finding an empty cell, trying the numbers from 1 to 9, and seeing if they are valid in that position. If a solution cannot be found with the current configuration, it backtracks and tries a different number. The solving continues until the board is solved or proven unsolvable. In the first case, the solved board is printed. Else, a message indicating that it cannot be solved is.

GENERATING UNIQUE BOARDS FOR SOLVING

The next task of our program is to generate unique solving boards for the user to solve, as well as return the solution to that board for the program to check the user's input. The main function that performs this task is *SolvingBoard*.

Solvingboard first calls a variation of the *solveBoard* function that first initializes four random positions on the board with four values from 1 to 9, then solves it. The reason this is necessary is because it adds restrictions on the function, which prevent it from going through its loops every time uninhibited and consequently producing the same board every time, which would have been the case had the board been empty from the

beginning. Next, the function has to create a solving board for the user whose unique solution is the board created in the previous step. In order to do that, a list of all 81 positions on the board is initialized, where each position is modeled as a structure that has a row and a column members. The list is then shuffled and the function loops over it and tests whether removing the value in each position would result in more than one solution for the board, in which case it skips this position and attempts to remove the next. This is so that the solution we have created in the beginning is the only valid solution for the board, and so can be easily used to check the user's input. The resulting solving board is then finally ready to be displayed to the user (Doc Brown).

Specification of Algorithms to be used (Maji, 2014)

FOR THE BACKTRACKING ALGORITHM:

- **Constraints**

  The algorithm ensures that numbers placed in cells adhere to the Sudoku rules, which include:

  - No repeated numbers in rows
  - No repeated numbers in columns.
  - No repeated numbers in 3x3 subgrids.

- **Recursive backtracking**

  Backtracking is mainly defined as systematically trying different options, evaluating their validity, and returning to the previous steps if no solution is suitable in the last step. Thus, when a partial solution is evaluated as invalid or

incomplete, the algorithm backtracks to the previous decision point and tries an alternative path.[1]

FOR THE UNIQUE SOLVING BOARD GENERATION:

- **Constraints:**
    - Ensure that a different board is generated each time the function is called.
    - Ensure that the solving board has only one possible solution.

Data Specifications

For the solve mode case:

The program takes values from the user from a displayed 9x9 board, places them in a 9x9 matrix, and outputs the solution to the same board or a message that the board is invalid if the board cannot be solved.

For the play mode case:

The program displays some values to the user on the 9x9 board and receives values from the user as input. The program then changes the color of the user's input to blue if the answer is correct and red if incorrect and displays a message when the whole board is solved along with the time taken to solve the board. The program also has the option to display 3 cells to help the user or display the final board.

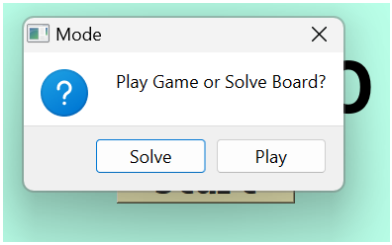
Graphical Output and Experimental Results

Graphical Output:

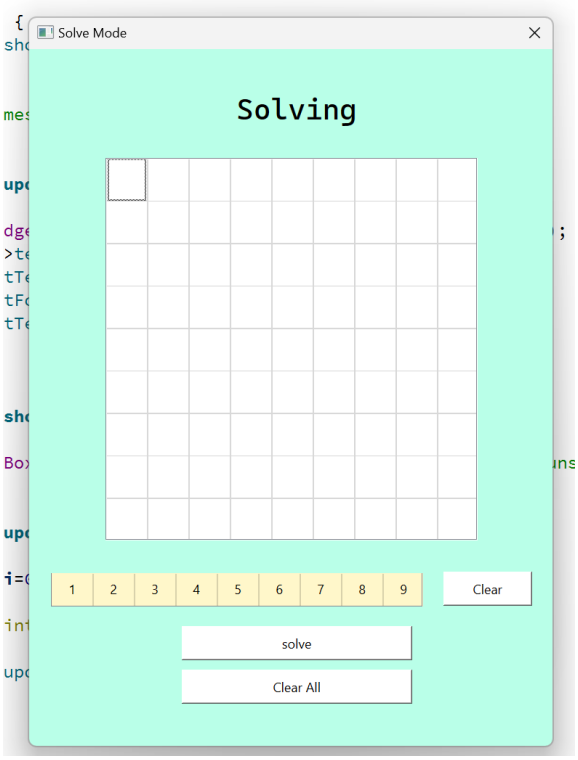
- Starting screen:

[1] How backtracking works in our sudoku game can be perfectly illustrated by the following video. https://youtu.be/lK4N8E6uNr4?si=pQ7K3jWNY3nhd0Mh

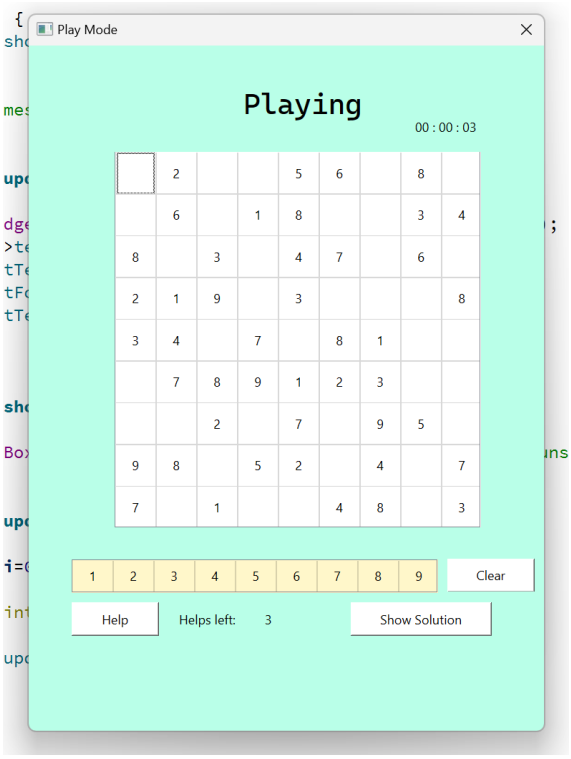

Start
Sudoko
Start


Mode
Play Game or Solve Board?
Solve
Play

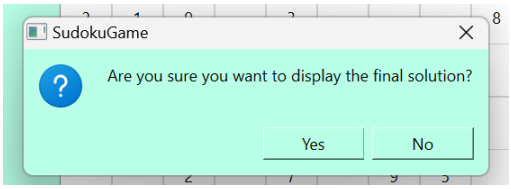
- Solve mode::

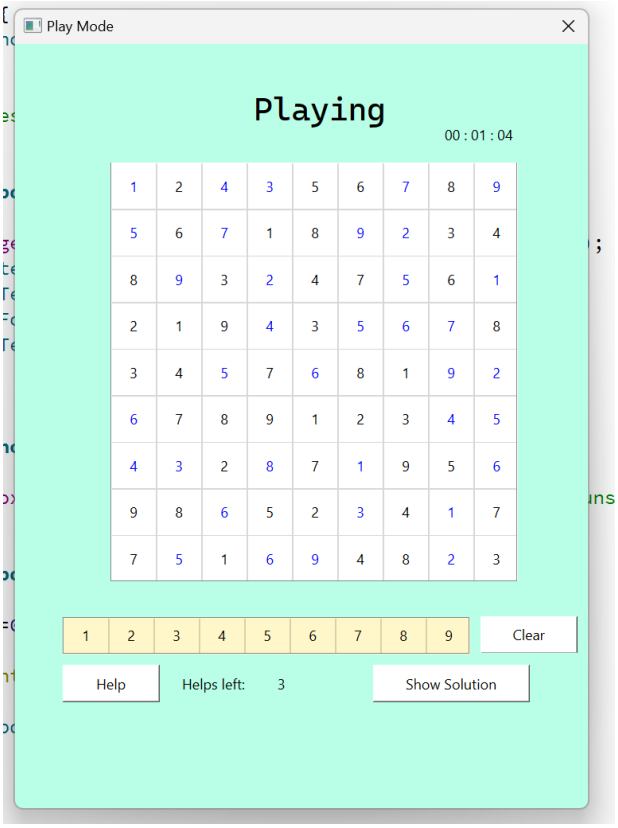


Play mode:



- “Show Solution” pressed:

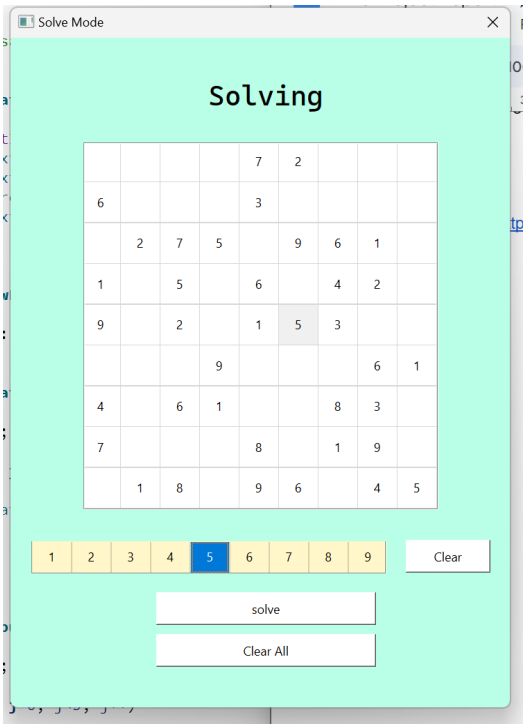




Experimental result:

Solve Mode:

- Inserting some values to be solved by the program
- Input:

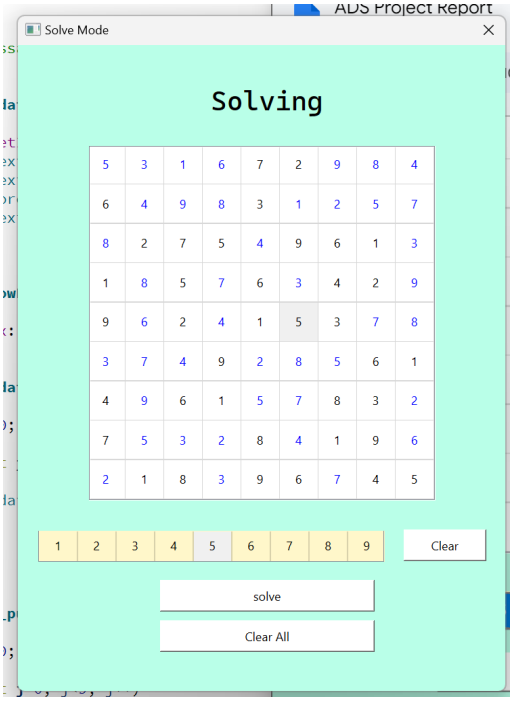


- Expected Output (from https://sudokuspoiler.com/sudoku/sudoku9)

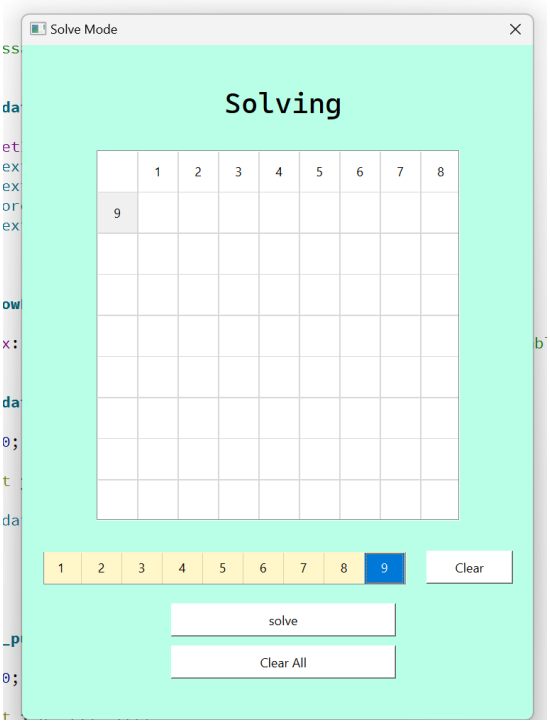
5	3	1	6	7	2	9	8	4
6	4	9	8	3	1	2	5	7
8	2	7	5	4	9	6	1	3
1	8	5	7	6	3	4	2	9
9	6	2	4	1	5	3	7	8
3	7	4	9	2	8	5	6	1
4	9	6	1	5	7	8	3	2
7	5	3	2	8	4	1	9	6
2	1	8	3	9	6	7	4	5

Solution 1 of 1

-

- Program Output:
  - 

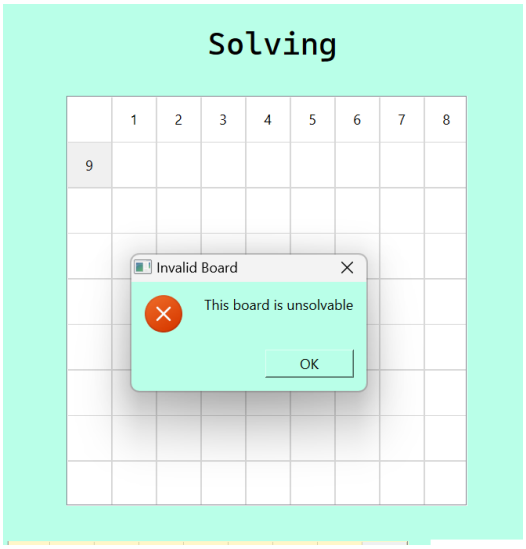
- Inserting an invalid board:
  - Input:



- Expected output:

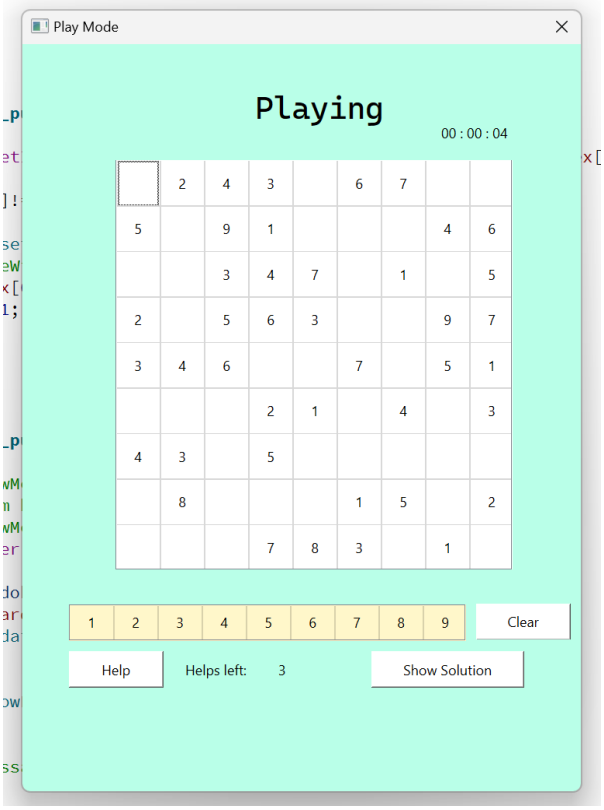
Error message

- Program output:



Play Mode:

- Output:



- Expected Solution (from https://sudokuspoiler.com/sudoku/sudoku9)

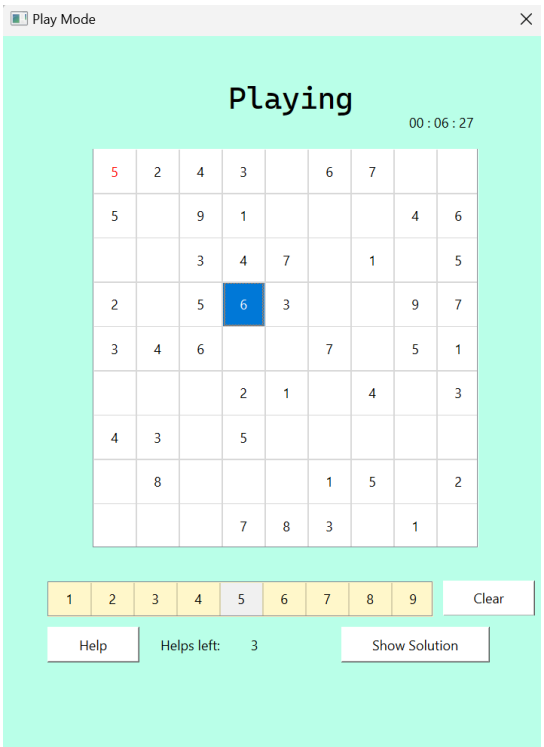
1	2	4	3	5	6	7	8	9
5	7	9	1	2	8	3	4	6
8	6	3	4	7	9	1	2	5
2	1	5	6	3	4	8	9	7
3	4	6	8	9	7	2	5	1
7	9	8	2	1	5	4	6	3
4	3	1	5	6	2	9	7	8
6	8	7	9	4	1	5	3	2
9	5	2	7	8	3	6	1	4

Solution 1 of 1

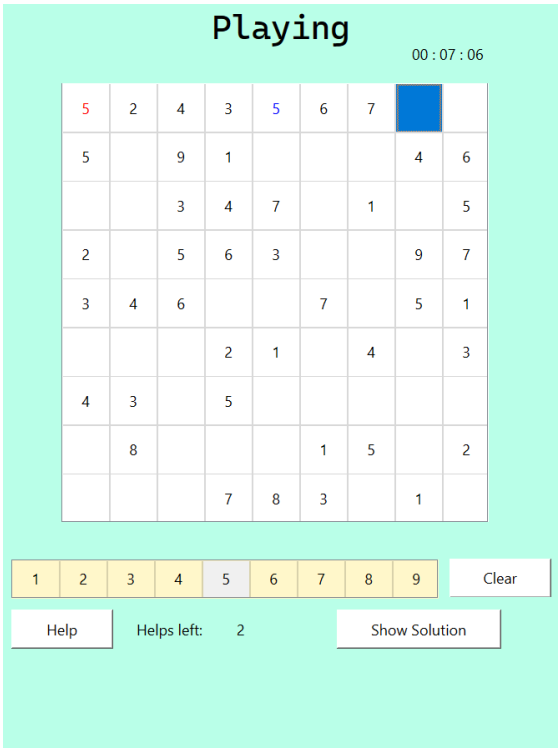
(Note that only one solution is available)

- Inserting correct value in cell [1][1]; expected output=blue color:

- Inserting incorrect solution in position [1][1]; expected output=red color:



- Asking for help for position [5][1]; expected output=5:



- Displaying solution:

Analysis and Critique:

The sudoku solver applied in this research paper has two primary algorithms. The first of which is backtracking, in which it systematically tries different techniques to evaluate their validity through the isValid function and then recursively backtracks previous steps to ensure that no solution was found. The backtracking technique is standard for solving problems that need constraint techniques like Sudoku, and it is easy for application. However, its time complexity is exponential, which is relatively high. In our case, the average case is O(9^(n^2)), where n is the grid size (9 for standard Sudoku). The overall complexity can be worse for more complex and more extensive data due to the rapid growth of the exponential complexity. The primary concern with the backtracking algorithm is its time complexity as it takes a considerable amount of time with the increase of difficulty, which may lead to performance issues when the conflicting data is discovered later on, and the program has to keep restarting from the very beginning.Moving to the second algorithm, unique board generation is a specific algorithm that generates a sudoku board with only one possible solution. This algorithm also has exponential complexity because of the repetition of solving required to ensure that each board has only one solution.

Now, let's have a more qualitative look at the time complexity of the main functions in the code:

SudokuSolver::SudokuBoard class (Solver. cpp):

- **isValid** is the function that checks if the number entered is aligned with the sudoku rules in which it is not repeated in the same column, row, or grid, and its complexity O(n) where n is the board size.
- **findEmptyPosition** is the function that searches for an empty position on the board, and its complexity is O(n^2).
- **solveBoard** is the function that runs the backtracking algorithm in exponential time, and its time complexity is O(9^(n^2)) in which n is the board size.

Play and Solve classes (play. cpp and solve. cpp):

- The complexity of this class is related to the user input and board manipulation with linear time complexity O(n) where n is the number of cells on the board.

In comparison to backtracking, the rule-based algorithm performs much better in terms of time complexity, and its main idea is that it uses deterministic logic to eliminate possibilities in cells based on the Sudoku rules (Berggren& Nilsson, 2012). By applying these rules, the possible values for each cell are reduced without the waste of complexity needed to explore every possible combination of every cell in exponential time in the backtracking algorithm. The average time complexity of this algorithm ranges from O(n^2) to O(n^4), where n represents the size of the sudoku board, which is mostly (9*9) for the standard puzzle. O(n^2) is the best time complexity for basic strategies like checking rows, columns, and grids, and O(n^3) is the complexity when more advanced strategies are used, such as naked pairs, triples, hidden singles, and pointing pairs. O(n^4) is the worst-case scenario, and it is applied when the sudoku puzzle reaches the highest level of difficulty.

In comparison with the Boltzmann Machines algorithm, the backtracking algorithm performed much better than the Boltzmann Machines algorithm (Berggren& Nilsson, 2012). Each cell in the Sudoku grid could be represented as a node in the network, and each cell is either filled or unfilled, and its potential value will vary from 1 to 9. The idea of Boltzmann Machines is that it decides the state of the node depending on the state of the neighboring nodes. It involves adjusting the weights between nodes to maximize the probability of observed configurations and minimizing the energy of the system. The time complexity depends on various factors, such as the size of the network, the number of nodes, the convergence criteria, and the learning algorithm used. For large networks and complex data, the training time can be considerable and might scale poorly with the size of the network, making Boltzmann Machines less practical for certain large-scale applications.

Conclusions

In conclusion, this paper explored the intricacies of Sudoku puzzle solving and our implementation of a program capable of solving and generating unique puzzles using the QT framework. The program's dual functionality has two modes, a solver and a play mode, for both Sudoku creators and enthusiasts to enjoy the user-friendly program. The paper covers a comprehensive analysis as it examines the theoretical aspects of Sudoku, including its rules, complexities, and data structures. The algorithmic implementations, which are backtracking, and unique board generation, were also explained in detail in the paper and specimens of the output were provided.

Finally, the paper also evaluates the time complexity of the used algorithm backtracking and compares it to more and less efficient algorithms that are also used in Sudoku games.

References:

A. K. Maji and R. K. Pal, "Sudoku solver using minigrid based backtracking," *2014 IEEE International Advance Computing Conference (IACC)*, Gurgaon, India, 2014, pp. 36-44, doi: 10.1109/IAdCC.2014.6779291.

Berggren, P., & Nilsson, D. (2012). A study of Sudoku solving algorithms. *Royal Institute of Technology, Stockholm*.

Doc Brown (Stack Overflow username). Answer to "How to generate Sudoku boards with unique solutions", Stack Overflow. 5 December, 2023. <https://stackoverflow.com/a/7280517>.

Appendix: (Please note that these are the codes used in QT)

Headers:

Play.h

```
#ifndef PLAY_H
#define PLAY_H
#include <QTimer>
#include <QDialog>
#include <vector>

namespace Ui {
class Play;
}

class Play : public QDialog
{
   Q_OBJECT

public:
   explicit Play(QWidget *parent = nullptr);
   ~Play();

private slots:

   void on_tableWidget_numbers_cellClicked(int row, int column);

   void on_tableWidget_board_cellClicked(int row, int column);

   void on_pushButton_Clear_clicked();

   void on_pushButton_help_clicked();

   void on_pushButton_Display_clicked();

   void finished();

   void Timer();

private:
```

```
    Ui::Play *ui;
    QTimer* timer;
    int Time=0;
    QString timeText;
    int index[2]={-1,-1};
    std::vector<std::vector<int>> board;
    std::vector<std::vector<int>> solution;
    //update with number of filled in values;
    //get them
    int answers;
    void displaySolution();
    bool solved=false;
};

#endif // PLAY_H
```

Solve.h

```
#ifndef SOLVE_H
#define SOLVE_H

#include <QDialog>
#include <QStatusBar>
#include <vector>


namespace Ui {
class Solve;
}

class Solve : public QDialog
{
    Q_OBJECT

public:
    explicit Solve(QWidget *parent = nullptr);
    ~Solve();

private slots:
```

```
    void on_tableWidget_numbers_cellClicked(int row, int column);

    void on_tableWidget_board_cellClicked(int row, int column);

    void on_pushButton_Clear_clicked();

    void on_pushButton_solve_clicked();

    void on_pushButton_clearAll_clicked();

private:
    Ui::Solve *ui;
    int index[2]={-1,-1};
    std::vector<std::vector<int>> board={{0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0},
                                         {0,0,0,0,0,0,0,0,0}};
    //QStatusBar* bar;
    void updatevalue(int,int,int);
    void updateBoard();
    void showError();
};

#endif // SOLVE_H
```

Start.h

```
#ifndef START_H
#define START_H

#include <QMainWindow>
#include "solve.h"
#include "play.h"
```

```
QT_BEGIN_NAMESPACE
namespace Ui { class Start; }
QT_END_NAMESPACE

class Start : public QMainWindow
{
    Q_OBJECT

public:
    Start(QWidget *parent = nullptr);
    ~Start();

private slots:
    void on_start_clicked();

private:
    Ui::Start *ui;
    Solve* solve;
    Play* play;
};
#endif // START_H
```

Source Code:

Solver.cpp

```
#include <iostream>
#include <vector>

namespace SudokuSolver {

    const int BOARD_SIZE = 9;

    class SudokuBoard {
    private:
        std::vector<std::vector<int>> board;
```

```
bool isValid(int row, int col, int num) {
    // Check if the number already exists in the same row or column
    for (int i = 0; i < BOARD_SIZE; i++) {
        if (board[row][i] == num || board[i][col] == num) {
            return false;
        }
    }

    // Check if the number already exists in the same 3x3 subgrid
    int subgridStartRow = (row / 3) * 3;
    int subgridStartCol = (col / 3) * 3;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[subgridStartRow + i][subgridStartCol + j] == num) {
                return false;
            }
        }
    }

    return true;
}


bool findEmptyPosition(int& row, int& col) {
    for (row = 0; row < BOARD_SIZE; row++) {
        for (col = 0; col < BOARD_SIZE; col++) {
            if (board[row][col] == 0) {
                return true;
            }
        }
    }
    return false;
}


bool solveBoard() {
    int row, col;
```

```
        // Find the next empty position on the board
        if (!findEmptyPosition(row, col)) {
            // If no empty position is found, the board is already solved
            return true;
        }

        // Try placing numbers from 1 to 9 at the empty position
        for (int num = 1; num <= 9; num++) {
            if (isValid(row, col, num)) {
                // Place the number at the empty position
                board[row][col] = num;

                // Recursively solve the board
                if (solveBoard()) {
                    return true;
                }

                // If the board cannot be solved with the current number, backtrack
                board[row][col] = 0;
            }
        }

        // If no number can be placed at the empty position, the board is unsolvable
        return false;
    }

public:

    SudokuBoard(const std::vector<std::vector<int>>& initialBoard) {
        board = initialBoard;
    }


    bool solve() {
        return solveBoard();
    }


    std::vector<std::vector<int>> printBoard() {
        std::vector<std::vector<int>>solution(9,std::vector<int>(9));
```

```
        for (int i = 0; i < BOARD_SIZE; i++) {
            for (int j = 0; j < BOARD_SIZE; j++) {
                solution[i][j]= board[i][j];
            }
        }
        return solution;
    }
};
}
```

GenerateBoards.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include<stdlib.h>
#include<ctime>
#define BOARD_SIZE 9

namespace SudokuSolver {
    //const int BOARD_SIZE = 9;

    class GenerateBoards {
    private:
        struct position {
            int r, c;
        };
        std::vector <position>list;

        std::vector<std::vector<int>> board{ {0,0,0,0,0,0,0,0,0}
                                            ,{0,0,0,0,0,0,0,0,0}
                                            ,{0,0,0,0,0,0,0,0,0}
                                            ,{0,0,0,0,0,0,0,0,0}
                                            ,{0,0,0,0,0,0,0,0,0}
                                            ,{0,0,0,0,0,0,0,0,0}
                                            ,{0,0,0,0,0,0,0,0,0}
                                            ,{0,0,0,0,0,0,0,0,0}
```

```
                                ,{0,0,0,0,0,0,0,0,0} };

bool isValid(int row, int col, int num) {
    // Check if the number already exists in the same row or column
    for (int i = 0; i < BOARD_SIZE; i++) {
        if (board[row][i] == num || board[i][col] == num) {
            return false;
        }
    }

    // Check if the number already exists in the same 3x3 subgrid
    int subgridStartRow = (row / 3) * 3;
    int subgridStartCol = (col / 3) * 3;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[subgridStartRow + i][subgridStartCol + j] == num) {
                return false;
            }
        }
    }

    return true;
}


bool isValid(std::vector<std::vector<int>> check, int row, int col, int num) {
    // Check if the number already exists in the same row or column
    for (int i = 0; i < BOARD_SIZE; i++) {
        if (check[row][i] == num || check[i][col] == num) {
            return false;
        }
    }

    // Check if the number already exists in the same 3x3 subgrid
    int subgridStartRow = (row / 3) * 3;
    int subgridStartCol = (col / 3) * 3;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
```

```
            if (check[subgridStartRow + i][subgridStartCol + j] == num) {
                return false;
            }
        }
    }

    return true;
}


bool solveSudoku() {
    for (int row = 0; row < BOARD_SIZE; row++) {
        for (int col = 0; col < BOARD_SIZE; col++) {
            if (board[row][col] == 0) {
                for (int num = 1; num <= BOARD_SIZE; num++) {
                    if (isValid(row, col, num)) {
                        board[row][col] = num;
                        if (solveSudoku())
                        {
                            return true;
                        }
                        board[row][col] = 0;
                    }
                }
                return false;
            }
        }
    }
    return true;
}

bool solveSudoku(std::vector<std::vector<int>> board) {
    for (int row = 0; row < BOARD_SIZE; row++) {
        for (int col = 0; col < BOARD_SIZE; col++) {
            if (board[row][col] == 0) {
                for (int num = 1; num <= BOARD_SIZE; num++) {
                    if (isValid(row, col, num)) {
                        board[row][col] = num;
                        if (solveSudoku())
                        {
```

```
                        return true;
                    }
                    board[row][col] = 0;
                }
            }
            return false;
        }
    }
    return true;
}

public:
  GenerateBoards() {
      // Initialize board with zeros
     //  board.assign(BOARD_SIZE, std::vector<int>(BOARD_SIZE, 0))

  }


  void printBoard() {
      for (int i = 0; i < BOARD_SIZE; i++) {
          if (i > 0 && i % 3 == 0) {
              std::cout << "---------------------" << std::endl;
          }
          for (int j = 0; j < BOARD_SIZE; j++) {
              if (j > 0 && j % 3 == 0) {
                  std::cout << "| ";
              }
              std::cout << board[i][j] << " ";
          }
          std::cout << std::endl;
      }
  }

  void printBoard(std::vector<std::vector<int>> board) {
      for (int i = 0; i < BOARD_SIZE; i++) {
          if (i > 0 && i % 3 == 0) {
              std::cout << "---------------------" << std::endl;
          }
```

```
            for (int j = 0; j < BOARD_SIZE; j++) {
                if (j > 0 && j % 3 == 0) {
                    std::cout << "| ";
                }
                std::cout << board[i][j] << " ";
            }
            std::cout << std::endl;
        }
    }

    //std::vector<std::vector<int>>
    void Solvingboard(std::vector<std::vector<int>>& OurBoard ,
std::vector<std::vector<int>> & Solution, int & answers)
    {
        answers=81;
        OurBoard = generateValidSudoku();
        Solution=OurBoard;
        //std::cout << "\n\n\n\n";
        int l, count = 0;
        list.resize(81);
        for (int i = 0; i < 81; i++) {
            list[i].r = i / 9;
            list[i].c = i % 9;
            /*
              for (int j = 0; j < 9;j++)
              {
                 for (int k = 0; k < 9;k++)
                 {
                    list[i].r = j;
                    list[i].c = k;

                 }
              }*/
        }
        std::shuffle(list.begin(), list.end(), std::mt19937{ std::random_device{}() });
        for (int i = 0; i < 81; i++)
        {
            count = 0;
            l = OurBoard.at(list[i].r).at(list[i].c);
            for (int w = 1; w < 10; w++)
```

```
        {
            OurBoard.at(list[i].r).at(list[i].c) = 0;
            //SudokuSolver::SudokuBoard sudokuBoard(OurBoard);

            if (isValid(OurBoard, list[i].r, list[i].c, w))
            {
                count++;
            }

        }
        if (count != 1)
        {
            OurBoard.at(list[i].r).at(list[i].c) = l;
        }
        else answers--;
        /* else
         {
             OurBoard.at(list[i].r).at(list[i].c) = l;

         }*/
    }

    //printBoard(OurBoard);
    //return OurBoard;
}
std::vector<std::vector<int>> generateValidSudoku() {

    // Solve an empty board using a backtracking algorithm

    srand(time(NULL));
    int x;
    int y;
    int z;
    z = (rand() % 9);
    x = (rand() % 9);
    y = (rand() % 9);

    board[x][y] = 1;
    board[y][z] = 6;
    board[z][x] = 9;
```

```
            board[x][x] = 3;

            solveSudoku(board);
            // Shuffle the values of the entire board
            std::vector<int> boardValues;

            for (int i = 0; i < BOARD_SIZE; i++)
            {
                for (int j = 0; j < BOARD_SIZE; j++)
                {
                    boardValues.push_back(board[i][j]);
                }
            }
            int idx = 0;
            for (int i = 0; i < BOARD_SIZE; i++)
            {
                for (int j = 0; j < BOARD_SIZE; j++)
                {
                    board[i][j] = boardValues[idx++];
                }
            }
            return board;
        }
    };
}
```

Play.cpp

```
#include "play.h"
#include "ui_play.h"
#include "GenerateBoards.cpp"
#include <QTabWidget>
#include <QTableWidgetItem>
#include <QDialog>
#include <QMessageBox>
#include <QTimer>
#include <QDateTime>

Play::Play(QWidget *parent) :
    QDialog(parent),
```

```
    ui(new Ui::Play)
{
    ui->setupUi(this);
    setWindowTitle("Play Mode");

    SudokuSolver::GenerateBoards newBoard;
    newBoard.Solvingboard(board,solution,answers);
    //initialize the table with board values
    QTableWidgetItem* item;
    for(int i=0; i<9; i++)
    {
        for(int j=0; j<9;j++)
        {
            item= new QTableWidgetItem();
            if (board[i][j]!=0)
            {
                item->setText(QString::number(board[i][j]));
                item->setForeground(Qt::black);
                item->setTextAlignment(Qt::AlignCenter);
            }
            else item->setText("");
            ui->tableWidget_board->setItem(i,j,item);
        }
    }

    //update Timer

    timer= new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(Timer()));
    timer->start(1000);

}

Play::~Play()
{
    //delete board
    for(int i=0; i<9; i++)
    {
        for(int j=0; j<9; j++)
        {
```

```
            QTableWidgetItem* item = ui->tableWidget_board->item(i,j);
            delete item;
        }
    }
    delete ui;
}



void Play::on_tableWidget_board_cellClicked(int row, int column)
{
    //copy index of selected cell
     index[0]=row; index[1]=column;
}


void Play::on_pushButton_Clear_clicked()
{
    QTableWidgetItem* item = ui->tableWidget_board->item(index[0], index[1]);

    //if a cell was selected, clear its content if it wasn't prefilled
    if(index[0]!=-1 && index[1]!=-1 && item->foreground()!=Qt::black&& !solved)
    {
        item->setText("");
   // ui->tableWidget_board->setItem(index[0],index[1],item);
    board[index[0]][index[1]]=0;
    }

    index[0]=-1; index[1]=-1;
}


void Play::on_tableWidget_numbers_cellClicked(int row, int column)
{
    //copy value of the button
    int value= column+1;
    QTableWidgetItem* item = ui->tableWidget_board->item(index[0],index[1]);

    //if an empty cell in the board was selected, update its value
```

```
    if(index[0]!=-1 && index[1]!=-1 && item->foreground()!=Qt::black)
    {
        item->setText(QString::number(value));
        item->setTextAlignment(Qt::AlignCenter);

    //ui->tableWidget_board->setItem(index[0],index[1],item);
    board[index[0]][index[1]]=value;

    //if the value was incorrect, set the color to red
    if(board[index[0]][index[1]]!=solution[index[0]][index[1]])
        item->setForeground(Qt::red);

    //else set the color to blue
    else
    {
        item->setForeground(Qt::blue);
        answers++;
        if(answers==81)
        {
            finished();
        }
    }



    }
    index[0]=-1; index[1]=-1;
}


void Play::on_pushButton_help_clicked()
{
    //get the number of helps
    int num=ui->label_helpsNum->text().toInt();

    QTableWidgetItem* item = ui->tableWidget_board->item(index[0],index[1]);
    //if there are still helps available and the selected cell was empty, reveal the selected
cell
    if(num>0)
    {
```

```
        if(index[0]!=-1 && index[1]!=-1 && item->foreground()!=Qt::black &&
item->foreground()!=Qt::blue)
        {
            item->setText(QString::number(solution[index[0]][index[1]]));
            item->setForeground(Qt::blue);
            item->setTextAlignment(Qt::AlignCenter);
            board[index[0]][index[1]]=solution[index[0]][index[1]];
            num--;
        }
        ui->label_helpsNum->setText(QString::number(num));
    }

    else QMessageBox::information(this,"","You have reached the maximum number of
helps");
}


void Play::on_pushButton_Display_clicked()
{
    if(!solved)
    {
   QMessageBox::StandardButton reply=
QMessageBox::question(this,"","Are you sure you want to display the final solution?",
                          QMessageBox::Yes | QMessageBox::No);
   if(reply==QMessageBox::Yes)
   {displaySolution();
       answers=81;
       solved=true;
   }
    }
}

void Play::finished()
{
    timer->stop();
    solved=true;
    //could display best time here
    QString message="Congratulations! You Have Solved the board\nYour time is:
"+timeText;
    QMessageBox::about(this, "Finished",message);
```

```
}

void Play::Timer()
{
   Time++;
   int sec,min,hr;
   hr=Time/3600;
   min=Time%3600/60;
   sec=Time%3600%60;
   QTime time;
   time.QTime::setHMS(hr,min,sec);
   timeText=time.toString("hh : mm : ss");
   ui->label_Time->setText(timeText);

}

void Play::displaySolution()
{
    timer->stop();
    for(int i=0; i<9; i++)
        for(int j=0; j<9; j++)
        {
            if(board[i][j]!=solution[i][j])
            {
            ui->tableWidget_board->item(i,j)->setText(QString::number(solution[i][j]));
            ui->tableWidget_board->item(i,j)->setTextAlignment(Qt::AlignCenter);
            ui->tableWidget_board->item(i,j)->setForeground(Qt::blue);
            board[i][j]=solution[i][j];
            }
        }
}
```

Solve.cpp

```
#include "solve.h"
//#include "qmainwindow.h"
#include "ui_solve.h"
#include <QPixmap>
```

```
#include <QTabWidget>
#include <QTableWidgetItem>
#include <QDialog>
#include <QMessageBox>
#include "Solver.cpp"
//#include <QStatusBar>

Solve::Solve(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Solve)
{
    ui->setupUi(this);
    setWindowTitle("Solve Mode");

    QTableWidgetItem* item;
    for(int i=0; i<9; i++)
    {
        for(int j=0; j<9;j++)
        {
            item= new QTableWidgetItem();
            item->setText("");
            ui->tableWidget_board->setItem(i,j,item);
        }
    }


}

Solve::~Solve()
{
    for(int i=0; i<9; i++)
    {
        for(int j=0; j<9; j++)
        {
            QTableWidgetItem* item = ui->tableWidget_board->item(i,j);
            delete item;
        }
    }
    delete ui;
}
```

```
void Solve::on_tableWidget_board_cellClicked(int row, int column)
{
    index[0]=row; index[1]=column;
}

void Solve::on_tableWidget_numbers_cellClicked(int row, int column)
{
    int value= column+1;
    QTableWidgetItem* item = ui->tableWidget_board->item(index[0],index[1]);
    if(index[0]!=-1 && index[1]!=-1)
    {
        item->setText(QString::number(value));
        item->setTextAlignment(Qt::AlignCenter);

    //ui->tableWidget_board->setItem(index[0],index[1],item);
    board[index[0]][index[1]]=value;
    index[0]=-1; index[1]=-1;
    }
}




void Solve::on_pushButton_Clear_clicked()
{
    QTableWidgetItem* item = ui->tableWidget_board->item(index[0], index[1]);

    if(index[0]!=-1 && index[1]!=-1)
    {
        item->setText("");
   // ui->tableWidget_board->setItem(index[0],index[1],item);
    board[index[0]][index[1]]=0;
    index[0]=-1; index[1]=-1;
    }
}
```

```
void Solve::on_pushButton_solve_clicked()
{
    //bar->showMessage("Solving...");
    //algorithm here
    //bar->showMessage("");
    SudokuSolver::SudokuBoard sudokuBoard(board);

        if (sudokuBoard.solve()) {
            board=sudokuBoard.printBoard();
            updateBoard();
        }
        else {
            showError();
        }

    //error message if unsolvabe
}

void Solve::updatevalue(int row, int col, int val)
{
    QTableWidgetItem * item= ui->tableWidget_board->item(row,col);
    if(item->text()==""){
    item->setText(QString::number(val));
    item->setForeground(Qt::blue);
    item->setTextAlignment(Qt::AlignCenter);
}
}

void Solve::showError()
{
    QMessageBox::critical(this, "Invalid Board", "This board is unsolvable");
}

void Solve::updateBoard()
{
    for(int i=0; i<9; i++)
    {
        for(int j=0;j<9;j++)
        {
```

```
            updatevalue(i,j,board.at(i).at(j));
        }
    }
}


void Solve::on_pushButton_clearAll_clicked()
{
    for(int i=0; i<9;i++)
    {
        for(int j=0; j<9; j++)
        {
            QTableWidgetItem* item = ui->tableWidget_board->item(i, j);
            {
                item->setText("");
            board[i][j]=0;
            }
        }
    }
                index[0]=-1; index[1]=-1;
}
```

Start.cpp

```
#include "start.h"
#include "ui_start.h"
#include <QMessageBox>

Start::Start(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::Start)
{
    ui->setupUi(this);
}

Start::~Start()
{
    delete ui;
}


void Start::on_start_clicked()
{

//https://stackoverflow.com/questions/35887523/qmessagebox-change-text-of-standard-
button
    //https://forum.qt.io/topic/58214/solved-qmessagebox-buttons

  QMessageBox box;

  box.setIcon(QMessageBox::Question);
  box.setWindowTitle("Mode");
  box.setText("Play Game or Solve Board?");
  box.setStandardButtons(QMessageBox::Yes | QMessageBox::No);
  box.button(QMessageBox::Yes)->setText("Solve");
  box.button(QMessageBox::No)->setText("Play");
  box.setDefaultButton(QMessageBox::Yes);
 box.exec();


{
}
```

```
 QMessageBox::StandardButton reply=box.standardButton(box.clickedButton());


 //what to do here
 //if (!box.escapeButton())
{
 if(reply== QMessageBox::Yes)
 {
   solve= new Solve(this);
   solve->setModal(true);
   solve->exec();
 }
 else if (reply== QMessageBox::No)
 {
    play= new Play(this);
    play->setModal(true);
    play->exec();
 }
}
}
```