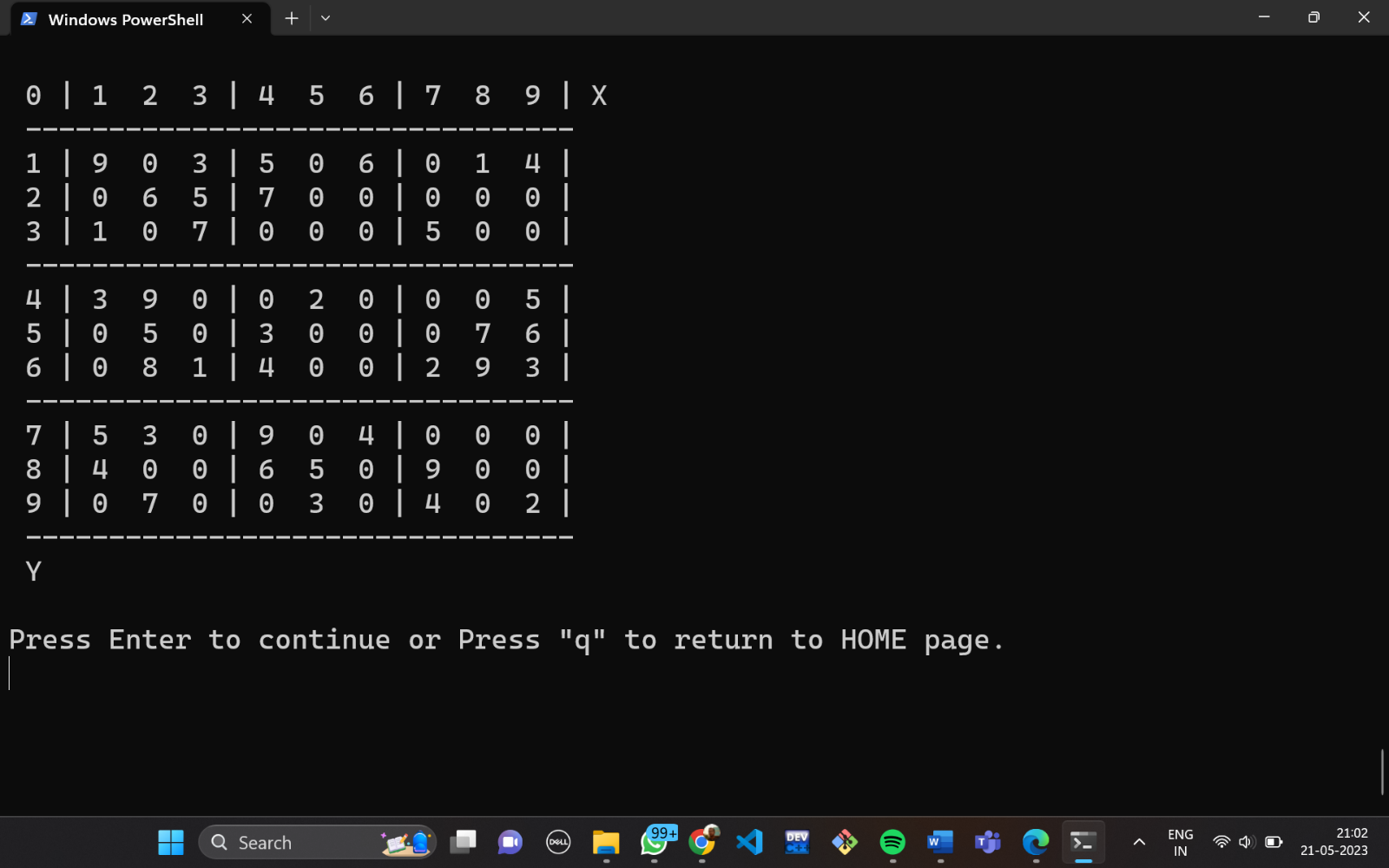
**Sudoku game in C**



**Subject Code :** ES CS – 201

**Tam Name** :CODEX

**Project :** Sudoku Game in C

**Team Member (Roll) :** Abir Mukherjee (01)

Debanjan Roy (31)

Abhik Ghosh (34)

Bikram Pal (35

**PROJECT INTRODUCTION :**

Sudoku is a popular number puzzle game that involves filling a 9x9 grid with numbers from 1 to 9, with each row, column, and 3x3 sub grid containing all the digits from 1 to 9. Solving a Sudoku puzzle can be a challenging task that requires logical reasoning and problem-solving skills.

In this project we are working on Solving a sudoku puzzle which can provided by the user as input or the machine can generate itself and store it in a file.One approach to solving Sudoku puzzles is to use graph theory, which involves representing the puzzle as a graph where each cell in the grid is a node and each edge connects nodes that are in the same row, column, or sub grid. By modelling the puzzle as a graph, we can use backtracking algorithms to find the solution efficiently.

The main problem in solving a Sudoku puzzle using a graph is finding an algorithm that can efficiently search the graph to find the correct solution. This requires identifying the most effective algorithmic techniques and optimizing them for the specific problem of solving Sudoku puzzles. Additionally, the algorithm must be able to handle the various complexities that can arise in Sudoku puzzles, such as multiple solutions or incomplete information in the initial grid.

Overall, solving a Sudoku puzzle using graph theory presents an interesting and challenging problem that requires a combination of mathematical, computational, and problem-solving skills.

**REQUIREMENTS OF THIS PROBLEM:**

***1.*** ***Input:***

a. The program should accept a Sudoku puzzle as input, either through a file or user input.

b. The input should be in the form of a 9x9 grid, where empty cells are represented by a placeholder value (e.g., 0 or blank space).

c. The solver should be capable enough to generate a puzzle of its own.

***2. Graph Representation:***

a. The Sudoku puzzle should be represented as a graph, where each cell represents a node.

b. The graph should capture the relationships between cells (nodes) in terms of rows, columns, and 3x3 sub grid.

c. The graph should be constructed such that each node is connected to its corresponding row, column, and sub grid nodes.

***3. Constraint Propagation:***

a. The program should implement constraint propagation techniques to reduce the search space and increase efficiency.

b. Constraint propagation involves applying Sudoku rules (no repetition in rows, columns, and sub grids) to update the possible values for each cell based on the values of its neighbours.

***4. Backtracking Algorithm:***

a. If constraint propagation alone is not sufficient to solve the puzzle, the program should incorporate a backtracking algorithm.

b. The backtracking algorithm should explore possible solution paths, making educated guesses and backtracking when a contradiction is encountered.

***5. Output:***

a. The program should provide the solution to the Sudoku puzzle as output.

b. The solution should be displayed in the form of a completed 9x9 grid.

c. The program should be able to store the file in a file if needed.

***6. User Interface (optional):***

a. If desired, the program may include a user interface to display the initial puzzle and the solved puzzle.

b. The user interface should allow users to input their Sudoku puzzles, view the solution, and potentially interact with the solving process.

c. The user interface should be clean and tidy.

***7. Performance and Scalability:***

a. The program should be designed to handle Sudoku puzzles of varying difficulties, ranging from easy to hard.

b. It should be efficient and able to solve puzzles within a reasonable amount of time.

c. The program should be scalable and capable of solving large sets of Sudoku puzzles efficiently.

***8. Error Handling:***

a. The program should handle input errors, such as invalid puzzle formats or values outside the valid range (1-9).

b. It should provide appropriate error messages or prompts to guide the user in correcting any mistakes.

***9. Code Quality and Documentation:***

a. The program should be well-structured, modular, and follow good coding practices.

b. It should include clear and concise documentation explaining the implementation details, algorithms used, and any assumptions made.

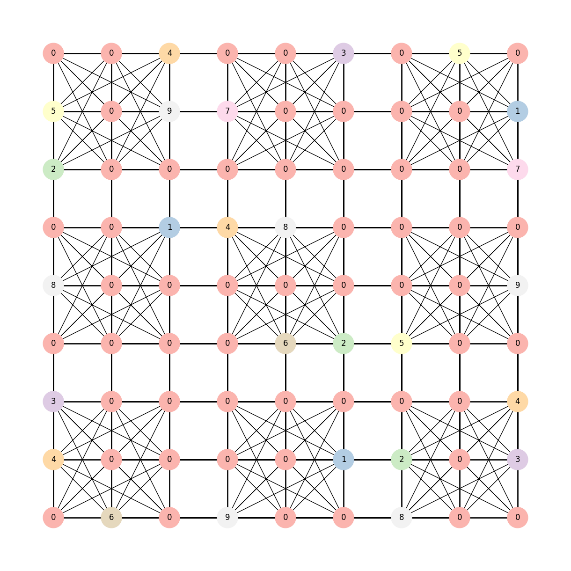
***10. Testing:***

a. The program should include a comprehensive set of test cases to validate the correctness of the implemented solution.

b. Test cases should cover various puzzle configurations, including both solvable and unsolvable puzzles.

**BRIEF SOLUTION:**

Sudoku is a logic-based number puzzle where a 9x9 grid is divided into nine 3x3 sub-grids, and the goal is to fill in the grid with digits from 1 to 9 so that each column, each row, and each sub-grid contains all the digits exactly once. Graph theory provides an effective approach to solving Sudoku puzzles. Here's a brief solution using graph theory:



***1. Graph Representation:***

* Create a graph where each cell in the Sudoku grid is a node.
* Connect nodes that belong to the same row, column, or sub-grid with edges.

***2. Encoding the Puzzle:***

* Convert the initial Sudoku puzzle into a graph representation.
* Assign weights to the edges based on the given or inferred constraints.
* For given clues, set the edge weight to a high value (e.g., 1,000) to prioritize preserving them.
* For unknown cells, set the edge weight to a lower value (e.g., 1).

***3. Constraint Satisfaction:***

* Apply constraint satisfaction techniques to solve the puzzle.
* Use the concept of graph colouring to assign values to nodes (cells) in a way that satisfies Sudoku rules.
* Start with a backtracking algorithm that explores possible assignments for each cell while checking for conflicts.

***4. Backtracking Algorithm:***

* Choose an empty cell and assign a number that is not conflicting with the filled cells in its row, column, and sub-grid.
* If a conflict arises (same number in the same row, column, or sub-grid), backtrack to the previous cell and try a different number.
* Repeat this process recursively until the puzzle is solved or no more valid assignments are possible.

***5. Optimization Techniques:***

* Apply additional optimization techniques like constraint propagation and inference to reduce the search space.
* Techniques such as naked singles, hidden singles, and advanced strategies can help eliminate possibilities and provide more direct assignments.

***6. Solution Validation:***

* Once a solution is found, verify that it satisfies all the Sudoku rules.
* Check that each row, column, and sub-grid contains all digits from 1 to 9 without any repetition.

***7. Multiple Solutions and Difficulty Levels:***

* Sudoku puzzles can have multiple valid solutions. If the goal is to find a unique solution, additional constraints or techniques may be necessary.
* For generating puzzles of varying difficulty levels, start with a completed grid and remove numbers while ensuring that the puzzle remains solvable through unique logical steps.

By applying graph theory concepts, encoding the puzzle, using constraint satisfaction techniques, and employing optimization methods, Sudoku puzzles can be effectively solved. The use of graph theory provides a structured approach and allows for efficient exploration of the solution space, making it an excellent tool for solving Sudoku puzzles.

**THE OBJECTIVE OF THE SOLUTION:**

The objective of the solution is to solve a given Sudoku puzzle by applying graph theory concepts, constraint satisfaction techniques, and optimization methods. The ultimate goal is to fill the 9x9 grid with digits from 1 to 9, ensuring that each column, each row, and each 3x3 sub-grid contains all the digits exactly once.

To achieve this objective, the solution involves the following steps:

***1. Graph Representation***: Create a graph where each cell in the Sudoku grid is a node, and connect nodes that belong to the same row, column, or sub-grid with edges. This graph representation helps in visualizing the relationships and dependencies among the cells.

***2. Encoding the Puzzle:*** Convert the initial Sudoku puzzle into a graph representation. Assign weights to the edges based on the given clues or inferred constraints. Higher weights are assigned to preserve the given clues, and lower weights are assigned to unknown cells.

***3. Constraint Satisfaction:*** Apply constraint satisfaction techniques, such as graph colouring, to assign values to nodes (cells) in a way that satisfies Sudoku rules. The solution starts with a backtracking algorithm that explores possible assignments for each cell while checking for conflicts.

***4. Backtracking Algorithm:*** Choose an empty cell and assign a number that does not conflict with the filled cells in its row, column, and sub-grid. If a conflict arises, backtrack to the previous cell and try a different number. Repeat this process recursively until the puzzle is solved or no more valid assignments are possible.

***5. Optimization Techniques:*** Apply additional optimization techniques, such as constraint propagation and inference, to reduce the search space and improve efficiency. Techniques like naked singles, hidden singles, and advanced strategies help eliminate possibilities and provide more direct assignments.

***6. Solution Validation:*** Once a solution is found, verify that it satisfies all the Sudoku rules. Check that each row, column, and sub-grid contains all digits from 1 to 9 without any repetition.

***7. Multiple Solutions and Difficulty Levels:*** Sudoku puzzles can have multiple valid solutions. If the goal is to find a unique solution, additional constraints or techniques may be necessary. For generating puzzles of varying difficulty levels, start with a completed grid and remove numbers while ensuring that the puzzle remains solvable through unique logical steps.

By following these steps and leveraging the power of graph theory, the solution aims to effectively solve Sudoku puzzles, providing a systematic and structured approach for achieving the puzzle's objective.

**APPLICATION OF THE PROJECT:**

The application of the project **"Solving a Sudoku Puzzle Using Graph Theory"** can be useful in various scenarios, including:

***1. Puzzle Solving:*** The primary application of the project is to solve Sudoku puzzles efficiently. Users can input a Sudoku puzzle through the interface, and the system will employ graph theory-based algorithms to find a valid solution.

***2. Puzzle Generation***: The project can be extended to generate Sudoku puzzles of varying difficulties. By implementing a puzzle generator module, the system can create new puzzles for users to solve.

***3. Puzzle Verification:*** The project can be utilized to verify the correctness of manually solved Sudoku puzzles. Users can input their solution, and the system will check if it adheres to Sudoku rules and provides feedback on any errors.

***4. Puzzle Teaching and Learning:*** The project can serve as an educational tool for teaching and learning Sudoku solving strategies. Users can input partially filled puzzles and observe the system's solution, which can help them understand the logical steps involved in solving Sudoku puzzles.

***5. Sudoku Competitions:*** The project can be employed in Sudoku competitions or events where participants are required to solve Sudoku puzzles within a specified time frame. The system can act as a timer, accept puzzle inputs, and validate the solutions provided by the participants.

***6. Puzzle Solving Algorithms Research:*** The project can serve as a platform for experimenting with and comparing different graph theory-based algorithms for solving Sudoku puzzles. Researchers and developers can use the system to analyse the performance and efficiency of various techniques.

***7. Integration with Other Applications:*** The Sudoku puzzle-solving component developed using graph theory can be integrated into other applications or systems. For example, it can be incorporated into mobile puzzle games, online puzzle platforms, or puzzle-solving robots.

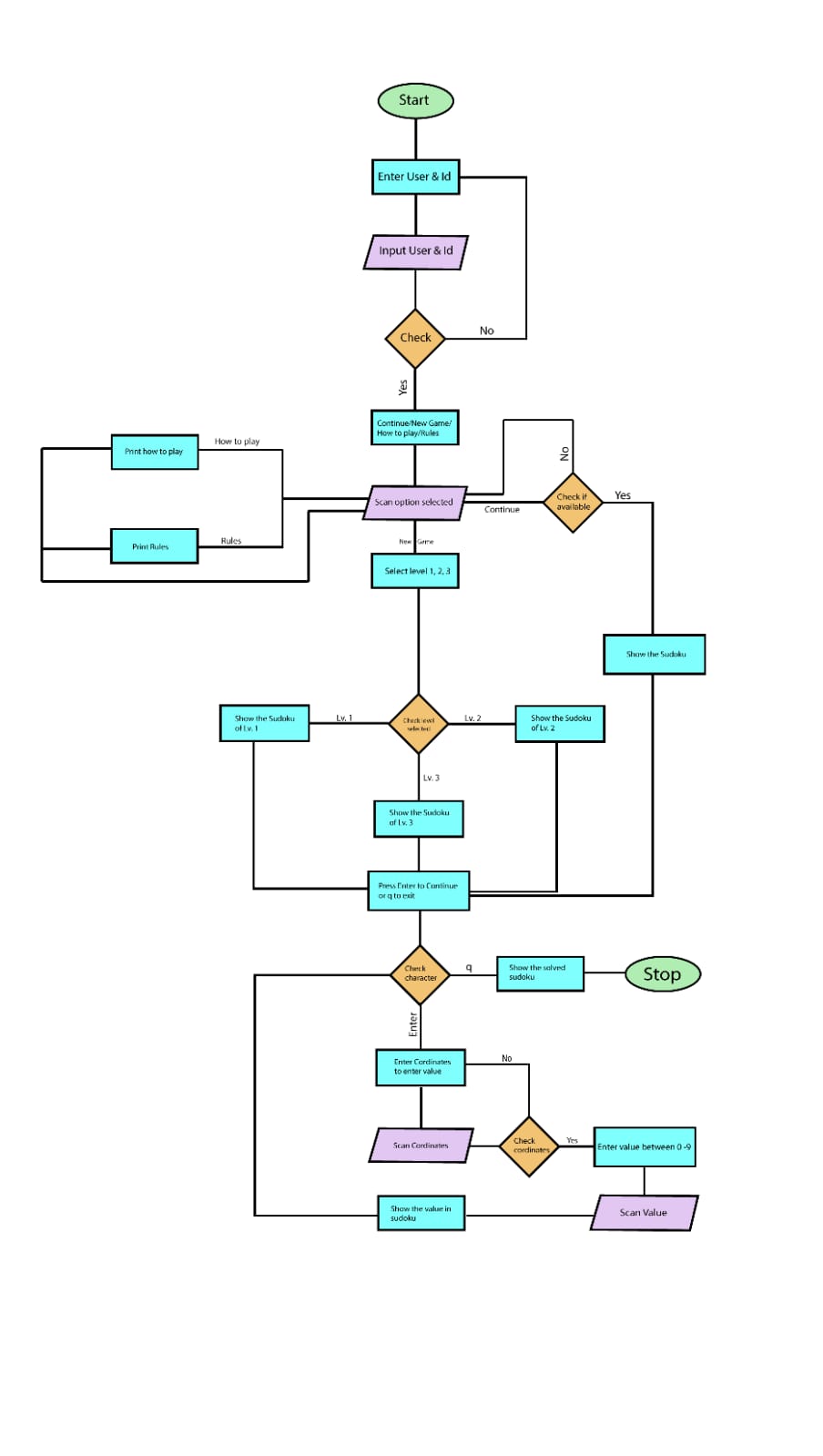
***8. Algorithm Optimization:*** The project can be used to explore and refine the efficiency of the Sudoku-solving algorithms. By analysing performance metrics and identifying areas for improvement, developers can optimize the algorithms to handle larger and more complex puzzles.

***9. Problem-solving Techniques:*** The Sudoku puzzle-solving project can be used as a demonstration of problem-solving techniques using graph theory. It can serve as a case study for understanding how to model real-world problems as graphs and apply appropriate algorithms to find solutions.

***10. Research and Analysis:*** The project can be used for research purposes, such as analysing the complexity of Sudoku puzzles, studying solving strategies, or exploring the relationships between Sudoku puzzles and graph theory concepts.

Overall, the application of **"Solving a Sudoku Puzzle Using Graph Theory"** extends beyond solving individual puzzles and can be beneficial in educational, competitive, research, and integration contexts.

**flOW Chart :**



**CODE*:***

MAIN FILE:

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <stdbool.h>

*#include* "sudoku.h"

*#include* "ui.h"

*#include* "settings.h"

*static* *int* *call=*0;

*int* *main*()

{

*int* *choice*;

*char* *enter*;

*int\*\** *puzzle*;

*int\*\** *userPuzzle*;

*int\*\** *tempPuzzle*;

*int\*\** *array*;

*int* *size*;

*int* *flag*;

*if* (*call==*0){

*while* (*flag=password*()*!=*1){

*continue*;

        }*call* *=* 1;

    }

*clear*(); *// clearing the screen*

*home*(); *// displays the home page*

*printf*("Enter your choice: ");

*scanf*("%d",*&choice*); *// input the user's choice*

*fflush*(*stdin*);

*switch* (*choice*)

    {

*case* 1:

*size* *=* *sizeofhistory*(); *// checks weather the file continue.txt is empty or not*

*if* (*size* *==* 0) { *// if file is empty show the message*

*printf*("\n\nYou Must start a new game to continue. \n\n");

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*main*();

            }

*else* { *// if puzzle is present*

*clear*();

*puzzle* *=* *createcontinuePuzzle*(); *// used to import the puzzle from file and store it in variable*

*userPuzzle* *=* *copyPuzzle*(*puzzle*); *// creating a copy of file for future reference*

*tempPuzzle* *=* *copyPuzzle*(*puzzle*); *// creating a copy of file for future reference*

*clear*();

*printPuzzle*(*userPuzzle*); *//prints the puzzle*

*userChoice*(*userPuzzle*,*tempPuzzle*,*puzzle*); *// this funtioon is used to update the puzzle*

*// free the memory allocated*

*free*(*puzzle*);

*free*(*userPuzzle*);

*free*(*tempPuzzle*);

            }

*break*;

*case* 2:

*clear*();

*puzzle* *=* *createPuzzle*();

*userPuzzle* *=* *copyPuzzle*(*puzzle*);

*tempPuzzle* *=* *copyPuzzle*(*puzzle*);

*clear*();

*printPuzzle*(*userPuzzle*);

*userChoice*(*userPuzzle*,*tempPuzzle*,*puzzle*);

*// free the memory allocated*

*free*(*puzzle*);

*free*(*userPuzzle*);

*free*(*tempPuzzle*);

*break*;

*case* 3:

*clear*();

*howtoplay*(); *// display the how to play screen*

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*main*();

*break*;

*case* 4:

*clear*();

*rules*(); *// display the rules screen*

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*main*();

*break*;

*case* 5:

*settings*(); *// displays settings screen*

*break*;

*case* 6: *// exit the program*

*printf*("\n\nThank You for Playing... \n\n");

*printf*("Press Enter to quit... \n\n");

*scanf*("%c",*&enter*);

*exit*(1);

*break*;

*default*: *// default case for wrong choice*

*printf*("\n\nEnter your choice correctly !! \n");

*printf*("Press Enter to continue... \n\n");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*main*();

    }

*return* 0;

}

SETTING FILE:

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <time.h>

*#include* "ui.h"

*void* *settings*();

*// clears the contents of the history file*

*void* *clearhistory*()

{

*char* *enter*;

    FILE*\** *fptr=fopen*("history.txt","w");

*fclose*(*fptr*);

*printf*("\n\nProgress deleted...\n ");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

}

*//add level automatically to the file entered by the user*

*void* *addautolevel*()

{

*// this function initializes random number with current time every sec*

*srand*(*time*(*NULL*));

*int\*\** *sudoku*;

*int\*\** *tempsudoku*;

*char* *enter*;

*int* *choice*;

*int* *cor\_x*,*cor\_y*;

*int* *val*,*a*,*flag=*0,*b*,*i*,*j*;

*int* *difficulty*;

*sudoku* *=* *createemptypuzzle*(); *// this creates apuzzle with only 0's*

    FILE *\*fptr*;

*clear*();

*printf*("\t\t\t\*\*  SELECT DIFFICULTY  \*\*\n\n\n");

*printf*("1. Easy \n");

*printf*("2. Medium \n");

*printf*("3. Hard \n\n");

*printf*("Press any key to return to home \n");

*scanf*("%d", *&difficulty*); *// user provides the level as input*

*fflush*(*stdin*);

*switch* (*difficulty*)

    {

*case* 1:

*a* *=* 41 ;

*b* *=* 5 ;

*fptr* *=* *fopen*("easy.txt","a");

*break*;

*case* 2:

*fptr* *=* *fopen*("medium.txt","a");

*a* *=* 48 ;

*b* *=* 8 ;

*break*;

*case* 3:

*fptr* *=* *fopen*("hard.txt","a");

*a* *=* 56 ;

*b* *=* 11 ;

*break*;

*default*:

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*main*();

*break*;

    }

*while* (*flag* *<* *b*)

    {

*// generates random number with rand()*

*cor\_x* *=* (*rand*() *%* 9) *+* 1 ;

*cor\_y* *=* (*rand*() *%* 9) *+* 1 ;

*val* *=* (*rand*() *%* 9) *+* 1 ;

*// filling the puzzle with random number*

*if* (*checkBox*(*sudoku*,*cor\_y-*1,*cor\_x-*1,*val*)){

*sudoku*[*cor\_y-*1][*cor\_x-*1] *=* *val* ;

        }*else*{

*continue*;

        }

*flag++*;

    }

*// solves the sudoku*

*solvePuzzle*(*sudoku*);

*flag=*0;

*while*(*flag* *<* *a*){

*// generating random coordinates for removal of digit and changing to 0*

*cor\_x* *=* (*rand*() *%* 9) *+* 1 ;

*cor\_y* *=* (*rand*() *%* 9) *+* 1 ;

*if* (*sudoku*[*cor\_y-*1][*cor\_x-*1]*!=*0){

*sudoku*[*cor\_y-*1][*cor\_x-*1] *=* 0 ;

        }*else*{

*continue*;

        }

*flag++*;

    }

*tempsudoku* *=* *copyPuzzle*(*sudoku*);

*clear*();

*if* (*solvePuzzle*(*tempsudoku*)){*// if puzzle is solvable*

*printf*("\n\nPuzzle Generated is : \n\n");

*printPuzzle*(*sudoku*);

    }*else*{

*printf*("\n\nSome unknown error occoured !!\n\n");

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*return*;

    }

*printf*("\n\nWant to add this puzzle to the file ?\n\n");

*printf*("\nType '1' to make changes or anything to cancel and return to settings page \n\n");

*scanf*("%d",*&choice*);

*fflush*(*stdin*);

*// adding puzzle to file*

*if* (*choice* *==* 1){

*if* (*fptr* *==* *NULL*) { *//checks if file is open or not*

*printf*("Error opening file!\n");

*exit*(0);

        }

*for* (*i* *=* 0; *i* *<* 9; *i++*) {

*for* (*j* *=* 0; *j* *<* 8; *j++*) {

*fprintf*(*fptr*, "%d ", *sudoku*[*i*][*j*]);

*if* (*j==*2 *||* *j==*5){

*fprintf*(*fptr*,"  ");

                }

            }

*fprintf*(*fptr*, "%d", *sudoku*[*i*][8]);

*fprintf*(*fptr*, "\n");

*if* (*i==*2 *||* *i==*5){

*fprintf*(*fptr*,"\n");

            }

        }

*fprintf*(*fptr*,"\n\n");

*fclose*(*fptr*);

*printf*("\n\nPuzzle added successfully...");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*return*;

    }*else* {

*printf*("\n\nProcess Terminated ");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*return*;

    }

*return*;

}

*// clears the content of file and resets all 3 file*

*void* *cleardata*(){

*int* *ch*, *temp*;

*char* *enter*;

    FILE*\** *fptr1*;

    FILE*\** *fptr2*;

*fptr1* *=* *fopen*("easy.txt", "r");

*fptr2* *=* *fopen*("replica1.txt", "w");

*temp* *=* 1;

*while* (*temp* *!=* 607){

*ch* *=* *fgetc*(*fptr1*);

*fputc*(*ch*, *fptr2*);

*temp++*;

    }

*fclose*(*fptr1*);

*fclose*(*fptr2*);

*remove*("easy.txt");

*rename*("replica1.txt", "easy.txt");

*fptr1* *=* *fopen*("medium.txt", "r");

*fptr2* *=* *fopen*("replica2.txt", "w");

*temp* *=* 1;

*while* (*temp* *!=* 607){

*ch* *=* *fgetc*(*fptr1*);

*fputc*(*ch*, *fptr2*);

*temp++*;

    }

*fclose*(*fptr1*);

*fclose*(*fptr2*);

*remove*("medium.txt");

*rename*("replica2.txt", "medium.txt");

*free*(*fptr1*);

*free*(*fptr2*);

*fptr1* *=* *fopen*("hard.txt", "r");

*fptr2* *=* *fopen*("replica3.txt", "w");

*temp* *=* 1;

*while* (*temp* *!=* 607){

*ch* *=* *fgetc*(*fptr1*);

*fputc*(*ch*, *fptr2*);

*temp++*;

    }

*fclose*(*fptr1*);

*fclose*(*fptr2*);

*remove*("hard.txt");

*rename*("replica3.txt", "hard.txt");

*free*(*fptr1*);

*free*(*fptr2*);

*printf*("\n\nGame Reset Successful...\n ");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

}

*//  reference matrix for adding new level*

*void* *reference*()

{

*int* *x=*0,*i*,*j*,*a*;

*clear*();

*printf*("\n\nThe matrix should be of this format \n\n");

*printf*("\n");

*printf*(" 0 | 1  2  3 | 4  5  6 | 7  8  9 | X\n");

*printf*(" ---------------------------------\n");

*for* (*i* *=* 0, *a* *=* 1; *i* *<* 9; *i++*, *a++*){

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*if* (*j* *==* 0)

*printf*(" %d |", *a*);

*else* *if* ((*j*) *%* 3 *==* 0)

*printf*("|");

*printf*(" %d ", *x*);

*if* (*j* *==* 8)

*printf*("|");

        }

*printf*("\n");

*if* ((*i* *+* 1) *%* 3 *==* 0)

*printf*(" ---------------------------------\n");

    }

*printf*(" Y\n");

}

*// this dunction is used to add new level to the file*

*void* *addlevel*()

{

*int* *difficulty*;

*int* *i*,*j*,*a*;

*char* *enter*;

*int\*\** *array*;

*int* *arr*[9][9];

*int* *val*;

*int\*\** *settemppuzzle*;

    FILE*\** *fptr*;

*clear*();

*printf*("\t\t\t\*\*  SELECT DIFFICULTY  \*\*\n\n\n");

*printf*("1. Easy \n");

*printf*("2. Medium \n");

*printf*("3. Hard \n\n");

*printf*("Press any key to return to settings \n");

*scanf*("%d", *&difficulty*);

*fflush*(*stdin*);

*switch* (*difficulty*)

    {

*case* 1:

*fptr=fopen*("easy.txt","a");

*break*;

*case* 2:

*fptr=fopen*("medium.txt","a");

*break*;

*case* 3:

*fptr=fopen*("hard.txt","a");

*break*;

*default* :

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*settings*();

    }

*reference*();

*printf*("Press Enter to continue... \n");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*for* (*i=*0; *i* *<* 9 ; *i++*){

*for* ( *j* *=* 0 ; *j* *<* 9 ; *j++*){

*printf*("\n\nEnter a number or '0' for empty place \n");

*printf*("Enter the value of %d and %d in the format (X Y) :\n",*j+*1,*i+*1);

*scanf*("%d",*&val*);

*fflush*(*stdin*);

*arr*[*i*][*j*]*=val*;

        }

    }

*array* *=* (*int\*\**)*malloc*(*sizeof*(*int\**) *\** 9);

*for* (*i* *=* 0; *i* *<* 9; *i++*){

*array*[*i*] *=* (*int\**)*malloc*(*sizeof*(*int*) *\** 9);

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*array*[*i*][*j*] *=* *arr*[*i*][*j*];

        }

    }

*printf*("\n\nThe matrix you entered is : \n\n");

*printf*("\n");

*printf*(" 0 | 1  2  3 | 4  5  6 | 7  8  9 | X\n");

*printf*(" ---------------------------------\n");

*for* (*i* *=* 0, *a* *=* 1; *i* *<* 9; *i++*, *a++*){

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*if* (*j* *==* 0)

*printf*(" %d |", *a*);

*else* *if* ((*j*) *%* 3 *==* 0)

*printf*("|");

*printf*(" %d ", *array*[*i*][*j*]);

*if* (*j* *==* 8)

*printf*("|");

        }

*printf*("\n");

*if* ((*i* *+* 1) *%* 3 *==* 0)

*printf*(" ---------------------------------\n");

    }

*printf*(" Y\n");

*settemppuzzle* *=* *copyPuzzle*(*array*);

*if* (*solvePuzzle*(*settemppuzzle*)){

*for* (*int* *i* *=* 0; *i* *<* 9; *i++*) {

*for* (*int* *j* *=* 0; *j* *<* 8; *j++*){

*fprintf*(*fptr*, "%d ", *array*[*i*][*j*]);

*if* (*j==*2 *||* *j==*5){

*fprintf*(*fptr*,"  ");

                }

            }

*fprintf*(*fptr*, "%d", *array*[*i*][8]);

*fprintf*(*fptr*, "\n");

*if* (*i==*2 *||* *i==*5){

*fprintf*(*fptr*,"\n");

                }

        }

*fprintf*(*fptr*,"\n\n");

*printf*("\n\nValues Entered into the file successfully.. ");

    }*else*{

*printf*("\n\nEnter a valid puzzle....\n");

    }

*fclose*(*fptr*);

*printf*("\n\nPress Enter to continue... \n\n");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

}

*// display settings sceen*

*void* *settings*()

{

*char* *enter*;

*int* *schoice*;

*clear*();

*printf*("\t\t\*\*  SETTINGS  \*\*\n\n");

*printf*("\n\n1. Add a Level by auto input \n");

*printf*("2. Add a Level by manual input \n");

*printf*("3. Reset Progress \n");

*printf*("4. Reset Game Data \n");

*printf*("5. Return Home \n\n");

*printf*("Enter your choice..\n");

*scanf*("%d",*&schoice*);

*fflush*(*stdin*);

*switch* (*schoice*) *//user choice*

    {

*case* 1:

*addautolevel*();

*settings*();

*break*;

*case* 2:

*addlevel*();

*settings*();

*break*;

*case* 3:

*clearhistory*();

*settings*();

*break*;

*case* 4:

*cleardata*();

*settings*();

*break*;

*case* 5:

*main*();

*break*;

*default*:

*printf*("\n\nEnter your choice correctly !! \n");

*printf*("Press Enter to continue... \n\n");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*settings*();

*break*;

    }

}

GAME FILE:

*#ifndef*  *SUDOKU\_H\_*

*#define*  *SUDOKU\_H\_*

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <stdbool.h>

*#include* "ui.h"

*int* *main*();

*void* *addautolevel*();

*// this function creates an empty puzzle with only 0's*

*int\*\** *createemptypuzzle*()

{

*int\*\** *puzzle*;

*int* *i*,*j*;

*puzzle* *=* (*int\*\**)*malloc*(*sizeof*(*int\**) *\** 9);

*for* (*i* *=* 0; *i* *<* 9; *i++*){

*puzzle*[*i*] *=* (*int\**)*malloc*(*sizeof*(*int*) *\** 9);

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*puzzle*[*i*][*j*] *=* 0;

        }

    }

*return* *puzzle*;

}

*// this function is used to count the length of continue.txt*

*int* *sizeofhistory*(){

*int* *len*;

    FILE*\** *fp* *=* *fopen*("history.txt", "r"); *//opens the file in read mode*

*fseek*(*fp*, 0, *SEEK\_END*);

*len* *=* *ftell*(*fp*);

*fclose*(*fp*); *//closes the file*

*return* *len*; *//len returns the size*

}

*void* *storeinhistory*(*int\*\** userPuzzle){ *//stores the sudoku in the file for future access*

    FILE *\*fp* *=* *fopen*("history.txt", "w"); *//opens the file in write mode*

*if* (*fp* *==* *NULL*) { *//checks if file is open or not*

*printf*("Error opening file!\n");

*exit*(0);

    }

*// Write 2D array to file*

*for* (*int* *i* *=* 0; *i* *<* 9; *i++*) {

*for* (*int* *j* *=* 0; *j* *<* 8; *j++*) {

*fprintf*(*fp*, "%d ", userPuzzle[*i*][*j*]);

*if* (*j==*2 *||* *j==*5){

*fprintf*(*fp*,"  ");

            }

        }

*fprintf*(*fp*, "%d", userPuzzle[*i*][8]);

*fprintf*(*fp*, "\n");

*if* (*i==*2 *||* *i==*5){

*fprintf*(*fp*,"\n");

            }

    }

*fprintf*(*fp*,"\n\n");

*fclose*(*fp*); *//closes the file*

}

*// this function copies the data from continue.txt and stores un puzzle*

*int\*\** *createcontinuePuzzle*()

{

*int* *i*,*j*;

*int\*\** *puzzle*;

*int* *arr*[9][9];

    FILE *\*fptr=fopen*("history.txt", "r");; *//opens the file in read mode*

*if* (*fptr* *==* *NULL*) {

*printf*("The file is not opened. The program will now exit.");

*exit*(0);

    }

*for* (*i=*0;*i<*9;*i++*){

*for* (*j=*0;*j<*9;*j++*){

*fscanf*(*fptr*,"%d",*&arr*[*i*][*j*]);

        }

    }

*puzzle* *=* (*int\*\**)*malloc*(*sizeof*(*int\**) *\** 9);

*for* (*i* *=* 0; *i* *<* 9; *i++*){

*puzzle*[*i*] *=* (*int\**)*malloc*(*sizeof*(*int*) *\** 9);

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*puzzle*[*i*][*j*] *=* *arr*[*i*][*j*];

        }

    }

*// returns the puzzle*

*return* *puzzle*;

}

*// this function is used to define the 9x9 sudoku puzzzle*

*int\*\** *createPuzzle*()

{

*int* *i*, *j*,*difficulty*,*enter*,*lvlc*,*val*,*nolvl*;

*int\*\** *puzzle*;

*int* *arr*[9][9];

*int* *choice*;

    FILE *\*fptr*;

*printf*("\t\t\t\*\*  SELECT DIFFICULTY  \*\*\n\n\n");

*printf*("1. Easy \n");

*printf*("2. Medium \n");

*printf*("3. Hard \n\n");

*printf*("Enter number for difficulty or Press any key to return to home \n");

*scanf*("%d", *&difficulty*); *// user provides the level as input*

*fflush*(*stdin*);

*// used for defining file pointer*

*if* (*difficulty==*1){

*fptr* *=* *fopen*("Easy.txt", "r");

*fseek*(*fptr*, 0, *SEEK\_END*);

*nolvl* *=* *ftell*(*fptr*);

*lvlc* *=* *level*(*nolvl*);

    }*else* *if* (*difficulty==*2){

*fptr* *=* *fopen*("Medium.txt", "r");

*fseek*(*fptr*, 0, *SEEK\_END*);

*nolvl* *=* *ftell*(*fptr*);

*lvlc* *=* *level*(*nolvl*);

    }*else* *if* (*difficulty==*3){

*fptr* *=* *fopen*("Hard.txt", "r");

*fseek*(*fptr*, 0, *SEEK\_END*);

*nolvl* *=* *ftell*(*fptr*);

*lvlc* *=* *level*(*nolvl*);

    }*else*{

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*main*();

    }

*if* (*lvlc==*0){

*printf*("\n\nCreate a level to continue\n");

*printf*("\n\nWant to add new level ?\n\n");

*printf*("Press 1 to continue or eny key to return to home page..\n");

*scanf*("%d",*&choice*);

*if* (*choice==*1){

*addautolevel*();

        }*else*{

*main*();

        }

*lvlc=*1;

    }

*if* (*fptr* *==* *NULL*) {

*printf*("The file is not opened. The program will now exit.");

*exit*(0);

    }

*val=*(*lvlc-*1)*\**215; *// used to find the position to import puzzle*

*fseek*(*fptr*, *val*, *SEEK\_SET*); *// used to seek the file pointer to desired position*

*// reading puzzle from the file*

*for* (*i=*0;*i<*9;*i++*){

*for* (*j=*0;*j<*9;*j++*){

*fscanf*(*fptr*,"%d",*&arr*[*i*][*j*]);

        }

    }

*puzzle* *=* (*int\*\**)*malloc*(*sizeof*(*int\**) *\** 9);

*for* (*i* *=* 0; *i* *<* 9; *i++*){

*puzzle*[*i*] *=* (*int\**)*malloc*(*sizeof*(*int*) *\** 9);

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*puzzle*[*i*][*j*] *=* *arr*[*i*][*j*];

        }

    }

*// returns the new puzzle*

*return* *puzzle*;

}

*//prints the sudoku puzzle and the co-ordinates to the screen*

*void* *printPuzzle*(*int\*\** puzzle)

{

*int* *i*, *j*, *a*;

*printf*("\n");

*printf*(" 0 | 1  2  3 | 4  5  6 | 7  8  9 | X\n");

*printf*(" ---------------------------------\n");

*for* (*i* *=* 0, *a* *=* 1; *i* *<* 9; *i++*, *a++*){

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*if* (*j* *==* 0)

*printf*(" %d |", *a*);

*else* *if* ((*j*) *%* 3 *==* 0)

*printf*("|");

*printf*(" %d ", puzzle[*i*][*j*]);

*if* (*j* *==* 8)

*printf*("|");

        }

*printf*("\n");

*if* ((*i* *+* 1) *%* 3 *==* 0)

*printf*(" ---------------------------------\n");

    }

*printf*(" Y\n");

}

*//checks if the cpuzzle has any empty places or not*

*bool* *checkAvailable*(*int\*\** puzzle, *int\** row, *int\** column)

{

*int* *i*, *j*;

*for* (*i* *=* 0; *i* *<* 9; *i++*){

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*if* (puzzle[*i*][*j*] *==* 0){ *// checks if the coordinates has 0 in the position or not*

*\**row *=* *i*;

*\**column  *=* *j*;

*return* *true*;

            }

        }

    }

*return* *false*;

}

*// used to check weather value is present in row , column , or 3x3 matrix*

*bool* *checkBox*(*int\*\** puzzle, *int* row, *int* column, *int* val)

{

*int* *i*, *j*;

*int* *squareRow*, *squareColumn*;

*//CHECK VERTICAL*

*for*(*i* *=* 0; *i* *<* 9; *i++*){

*if* (puzzle[*i*][column] *==* val)

*return* *false*;

    }

*//CHECK HORIZONTAL*

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*if* (puzzle[row][*j*] *==* val)

*return* *false*;

    }

*//CHECK SQUARE*

*squareRow* *=* row *-* row *%* 3;

*squareColumn* *=* column *-* column *%* 3;

*for*(*i* *=* 0; *i* *<* 3; *i++*){

*for*(*j* *=* 0; *j* *<* 3; *j++*){

*if*(puzzle[*squareRow* *+* *i*][*squareColumn* *+* *j*] *==* val)

*return* *false*;

        }

    }

*return* *true*;

}

*//this function solves the puzzle and checks the value if not matched*

*bool* *solvePuzzle*(*int\*\** puzzle)

{

*int* *i*, *j*, *val*;

*// checks weather the position is vacant or not*

*if*(*!checkAvailable*(puzzle, *&i*, *&j*))

*return* *true*;

*// assigning values to the position*

*for*(*val* *=* 1; *val* *<* 10; *val++*){

*// checking if the value is alredy present or not*

*if*(*checkBox*(puzzle, *i*, *j*, *val*)){

            puzzle[*i*][*j*] *=* *val*;

*if*(*solvePuzzle*(puzzle))

*return* *true*; *//solution available*

*else*

                puzzle[*i*][*j*] *=* 0;

        }

    }

*return* *false*; *//puzzle not solved*

}

*//copies values from existing puzzle to new puzzle*

*int\*\** *copyPuzzle*(*int\*\** puzzle){

*int* *i*, *j*;

*int\*\** *newPuzzle*;

*newPuzzle* *=* (*int\*\**)*malloc*(*sizeof*(*int\**) *\** 9);

*for* (*i* *=* 0; *i* *<* 9; *i++*){

*newPuzzle*[*i*] *=* (*int\**)*malloc*(*sizeof*(*int*) *\** 9);

*for*(*j* *=* 0; *j* *<* 9; *j++*){

*newPuzzle*[*i*][*j*] *=* puzzle[*i*][*j*];

        }

    }

*return* *newPuzzle*;

}

*// function to enter values to the puzzle*

*void* *userChoice*(*int\*\** userPuzzle, *int\*\** tempPuzzle, *int\*\** puzzle)

{

*int* *i*, *j*, *enter*;

*int* *positionX*, *positionY*, *userVal*;

*char* *c*;

*while*(1){

*// checks completion of the puzzle*

*if*(*!checkAvailable*(userPuzzle, *&i*, *&j*)){

*printf*("\nGood Job You Solved the Puzzle!!!\n");

*return*;

        }

*while*(1){

*printf*("\nPress Enter to continue or Press \"q\" to return to HOME page.\n"); *//prompt after each successful entry*

*c* *=* *getchar*();

*if*((*c* *==* 'q') *||* (*c* *==* 'Q')){

*getchar*();

*main*();

*return*;

            }

*else* *if*((*c* *!=* '\n') *&&* (*c* *!=* 'q'))

*getchar*();

*else*

*break*;

        }

*// takes the user desired co-ordinate as input*

*printf*("\nPlease Enter Coordinate for the square you want to insert the value to in the following format \"X Y\":\n");

*scanf*("%d %d",*&positionX*, *&positionY*);

*while*(1){

*// checking range of coordinate*

*if* ((*positionX* *>* 9) *||* (*positionX* *<* 1) *||* (*positionY* *>* 9) *||* (*positionY* *<* 1) *||* (userPuzzle[*positionY* *-* 1][*positionX* *-* 1] *!=* 0))

            {

*printf*("\nYou can't insert value to this Coordinate, please pick a new Coordinate\n");

*scanf*("%d %d",*&positionX*, *&positionY*);

            }

*else* {

*positionX* *-=* 1;

*positionY* *-=* 1;

*break*;

            }

        }

*printf*("\nPlease insert value from 1 to 9\n");

*scanf*("%d", *&userVal*); *// takes the user desired value for the already provided co-ordinate*

*while*(1){

*if*((*userVal* *>* 9) *||* (*userVal* *<* 1)){

*printf*("\nPlease insert valid value:\n");

*scanf*("%d", *&userVal*);

            }

*else* *break*;

        }

*// returns true if value is not present in the puzzle*

*if* (*checkBox*(userPuzzle, *positionY*, *positionX*, *userVal*)){

            userPuzzle[*positionY*][*positionX*] *=* *userVal*;*// storing val to user puzzle*

        }

*else*{

*printf*("\nIncorrect value for the X = %d Y = %d coordinate, please try again\n",*positionX* *+* 1, *positionY* *+* 1); *//display the value is already present*

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

        }

*// changes value of userval to temp puzzle*

*for* (*i* *=* 0; *i* *<* 9; *i++*){

*for*(*j* *=* 0; *j* *<* 9; *j++*){

                tempPuzzle[*i*][*j*] *=* userPuzzle[*i*][*j*];

            }

        }

*// checks weather the value is correct or not*

*if*(*!solvePuzzle*(tempPuzzle)){

*//checks temp puzzle and changes the value of temppuzzle*

*printf*("\nIncorrect value for the X = %d Y = %d coordinate, please try again\n",*positionX* *+* 1, *positionY* *+* 1);

            userPuzzle[*positionY*][*positionX*] *=* 0;

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

        }

*getchar*();

*clear*();

*printPuzzle*(userPuzzle);

*if* (puzzle*!=*userPuzzle)

            {

*storeinhistory*(userPuzzle);

            }

    }

*return*;

}

*#endif*

UI FILE:

*#ifndef*  *UI\_H\_*

*#define*  *UI\_H\_*

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <string.h>

*struct* User

{

*char* *name*[20];

*int* *password*;

};

*void* *clear*();

*// password() is used to for all necessary inputs and ui of login page*

*int* *password*()

{

*char* *name1*[20] *=* "Abir\0";

*int* *id1* *=* 2001;

*char* *name2*[20] *=* "Debanjan\0";

*int* *id2* *=* 2031;

*char* *name3*[20] *=* "Abhik\0";

*int* *id3* *=* 2034;

*char* *name4*[20] *=* "Bikram\0";

*int* *id4* *=* 2035;

*char* *c*,*enter*;

*int* *i*;

*struct* User *user*;

*clear*();

*printf*("\t\t\*\*  LOGIN PAGE  \*\*\n\n");

*printf*("\nUser Name : ");

*scanf*("%s", *&user*.*name*);

*printf*("\nPassword : ");

*scanf*("%d",*&user*.*password*);

*fflush*(*stdin*);

*// checking for username and password*

*if* (*strcmp*(*user*.*name*,*name1*)*==*0 *&&* *user*.*password==id1*)

    {

*printf*("\n\nWelcome.....\n");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*return* 1;

    }*else* *if*(*strcmp*(*user*.*name*,*name2*)*==*0 *&&* *user*.*password==id2*)

    {

*printf*("\n\nWelcome.....\n");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*return* 1;

    }*else* *if*(*strcmp*(*user*.*name*,*name3*)*==*0 *&&* *user*.*password==id3*)

    {

*printf*("\n\nWelcome.....\n");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*return* 1;

    }*else* *if*(*strcmp*(*user*.*name*,*name4*)*==*0 *&&* *user*.*password==id4*)

    {

*printf*("\n\nWelcome.....\n");

*printf*("\n\nPress Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*return* 1;

    }*else*{ *//if the username and password not matching to what is stored*

*while*(1){

*printf*("\nPress Enter to try again or press \"q\" to quit.\n");

*c* *=* *getchar*();

*if*((*c* *==* 'q') *||* (*c* *==* 'Q')){

*printf*("\n\n");

*exit*(0);

            }*else*{

*return* 0;

            }

        }

    }

}

*// this unction is used to print empty lines to clear up the interface*

*void* *clear*()

{

*int* *i=*0;

*while* (*i<*45){

*printf*("\n");

*i++*;

    }

}

*// this function is used to display the level selection screen*

*int* *level*(*int* nolvl)

{

*int* *cas*,*i*,*lvlc*;

*char* *enter*;

*cas* *=* nolvl*/*215;

*if* (*cas==*0){

*clear*();

*printf*("\n\nFirst create a level to continue.\n\n");

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*return* 0;

    }

*while* (1){

*i=*1;

*clear*();

*printf*("\t\t\*\*  SELECT LEVEL  \*\*\n\n");

*while* (*i* *<=* *cas*){

*printf*("%d. Level %d\n",*i*,*i*);

*i++*;

        }

*i--*;

*printf*("\n\nSelect level to continue...\n");

*scanf*("%d",*&lvlc*);

*fflush*(*stdin*);

*if* (*lvlc* *>* *i*){

*printf*("\n\nEnter your choice correctly.....\n\n");

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*continue*;

        }*else* *if* (*lvlc* *<* 1){

*printf*("\n\nEnter your choice correctly.....\n\n");

*printf*("Press Enter to continue... ");

*scanf*("%c",*&enter*);

*fflush*(*stdin*);

*continue*;

        }*else*{

*break*;

        }

    }

*return* *lvlc*;

}

*// this function is used to print the home page*

*void* *home*()

{

*printf*("\t\t\t\*\*  SUDOKU GAME  \*\*\n\n\n");

*printf*("1. Continue \n");

*printf*("2. New Game \n");

*printf*("3. How To Play\n");

*printf*("4. Rules \n");

*printf*("5. Settings\n");

*printf*("6. Exit \n\n");

}

*// this function is used to print the how to play screen*

*void* *howtoplay*()

{

*printf*("\t\t\*\*  HOW TO PLAY  \*\*\n\n");

*printf*("\n\n1. Press Enter to start the game \n");

*printf*("2. Select the level \n"); *// use only if required*

*printf*("3. Select |X| and |Y| cordinate to enter the value at that position \n");

*printf*("4. Check if your given answer is write or not \n");

*printf*("5. Play as many times you want \n\n");

}

*// this function is used to printthe rules page*

*void* *rules*()

{

*printf*("\t\t\*\*  RULES  \*\*\n\n");

*printf*("\n\n1. Sudoku Rule 1: Use Numbers 1-9 \n");

*printf*("2. Sudoku Rule 2: Don't Repeat Any Numbers \n");

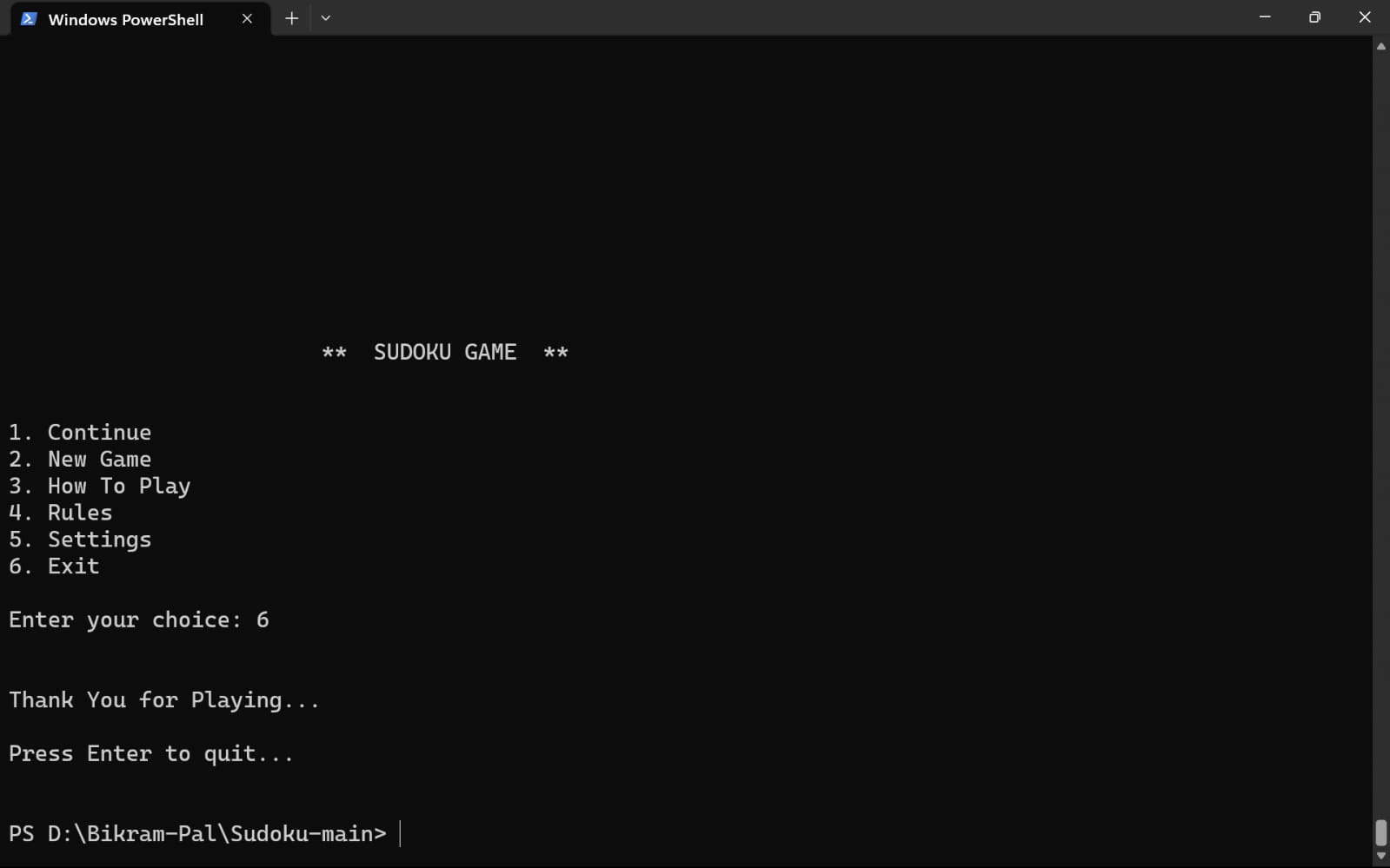
*printf*("3. Sudoku Rule 3: Don't Guess \n");

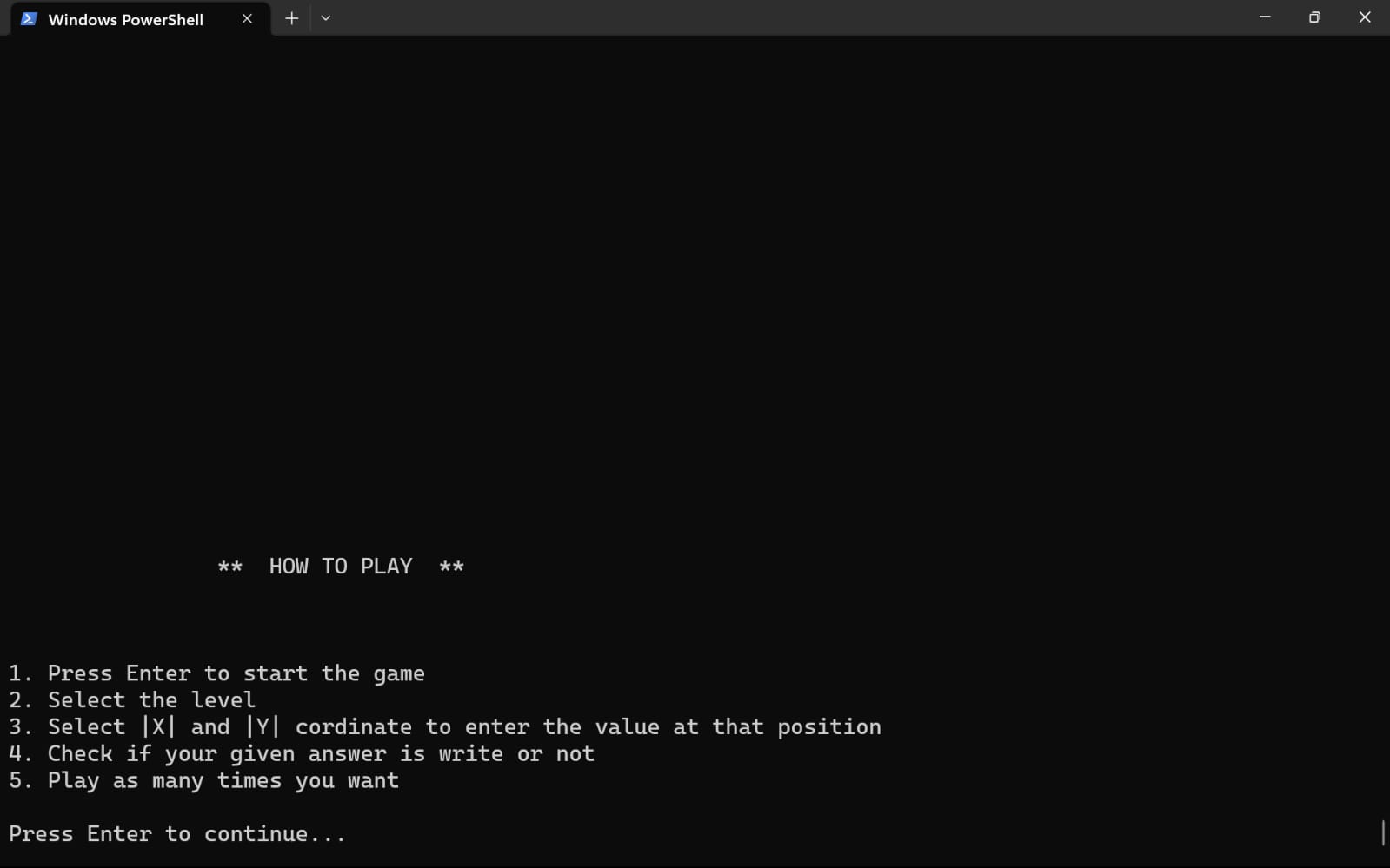
*printf*("4. Sudoku Rule 4: Use Process of Elimination \n\n");

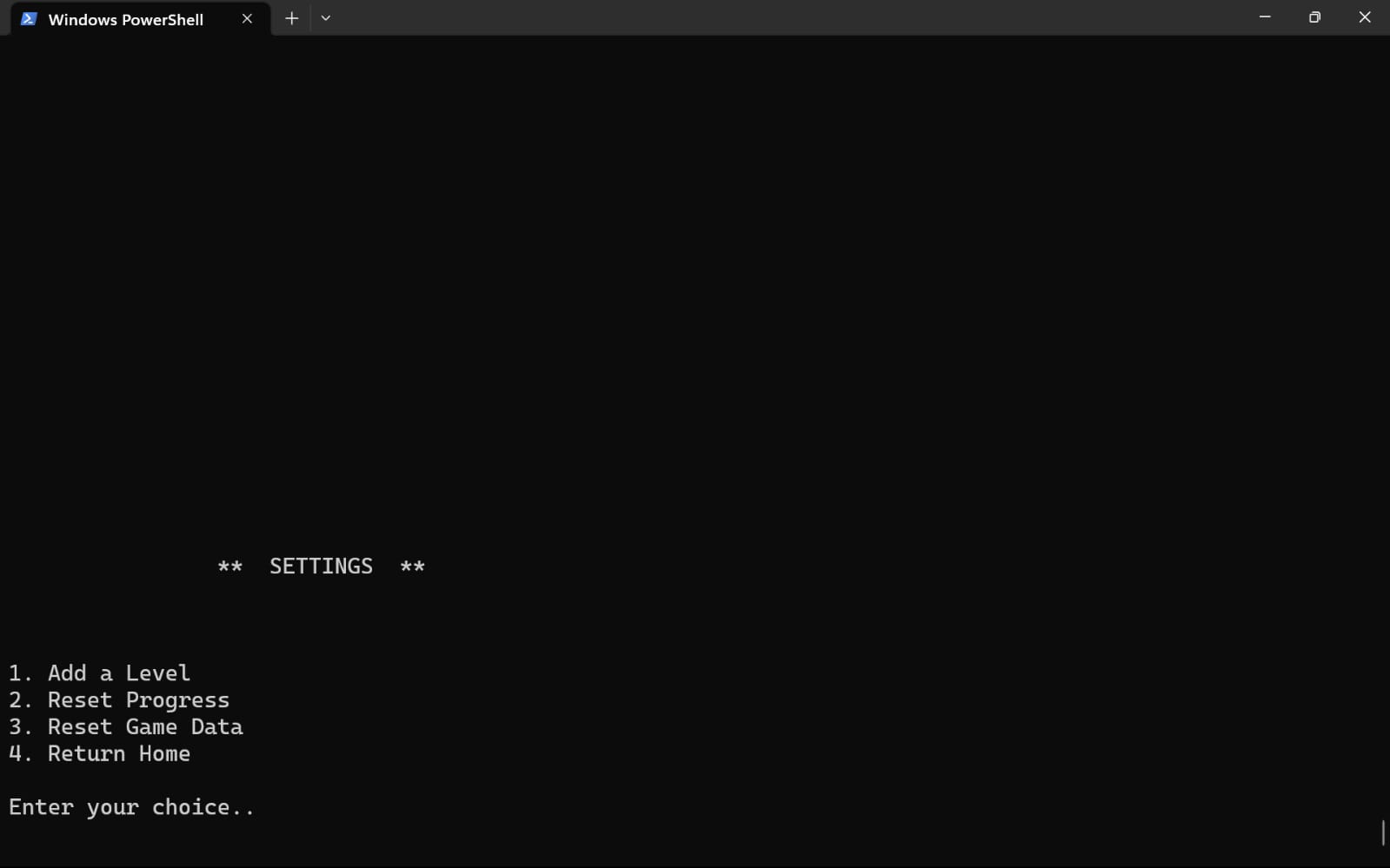
}

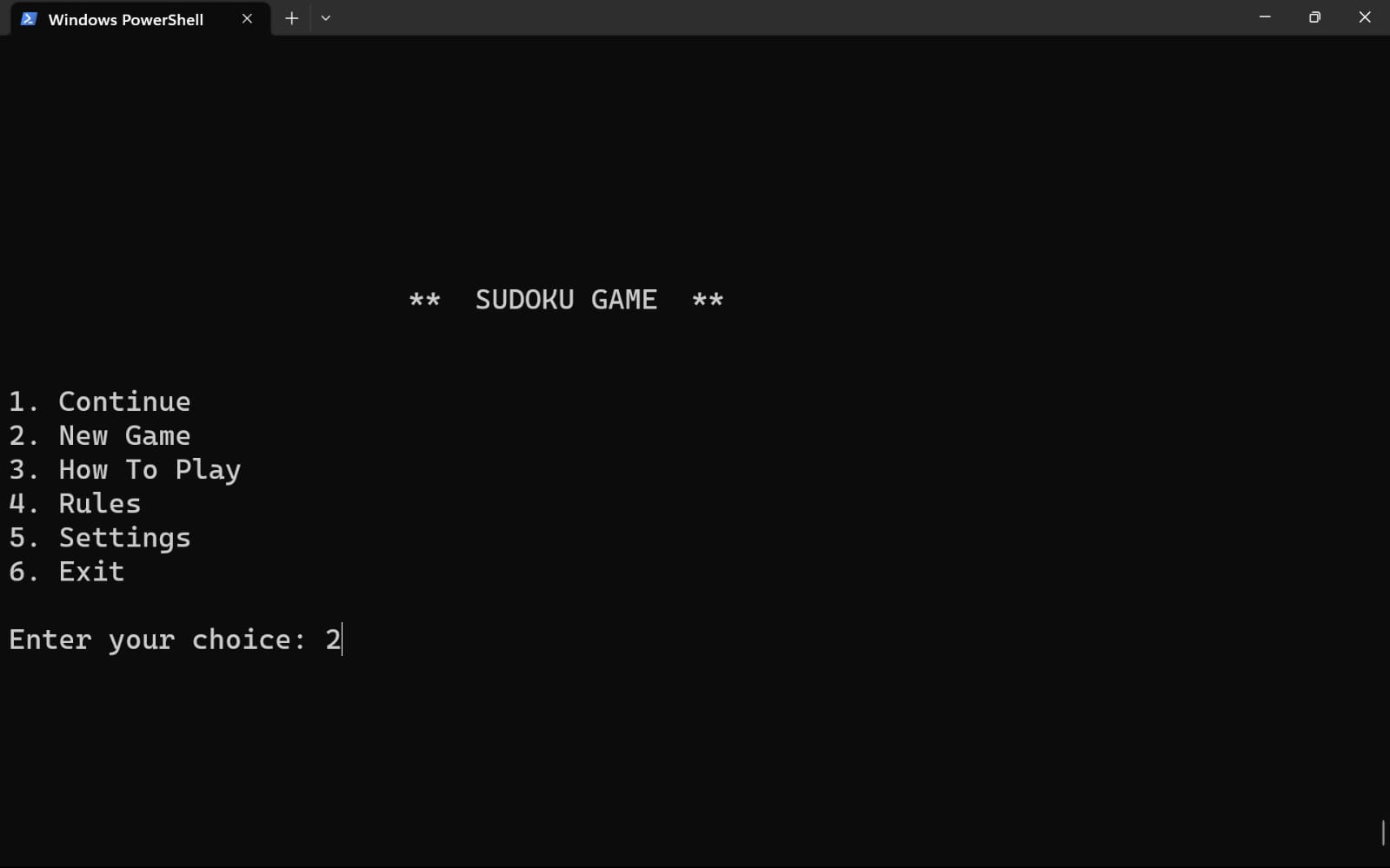
*#endif*

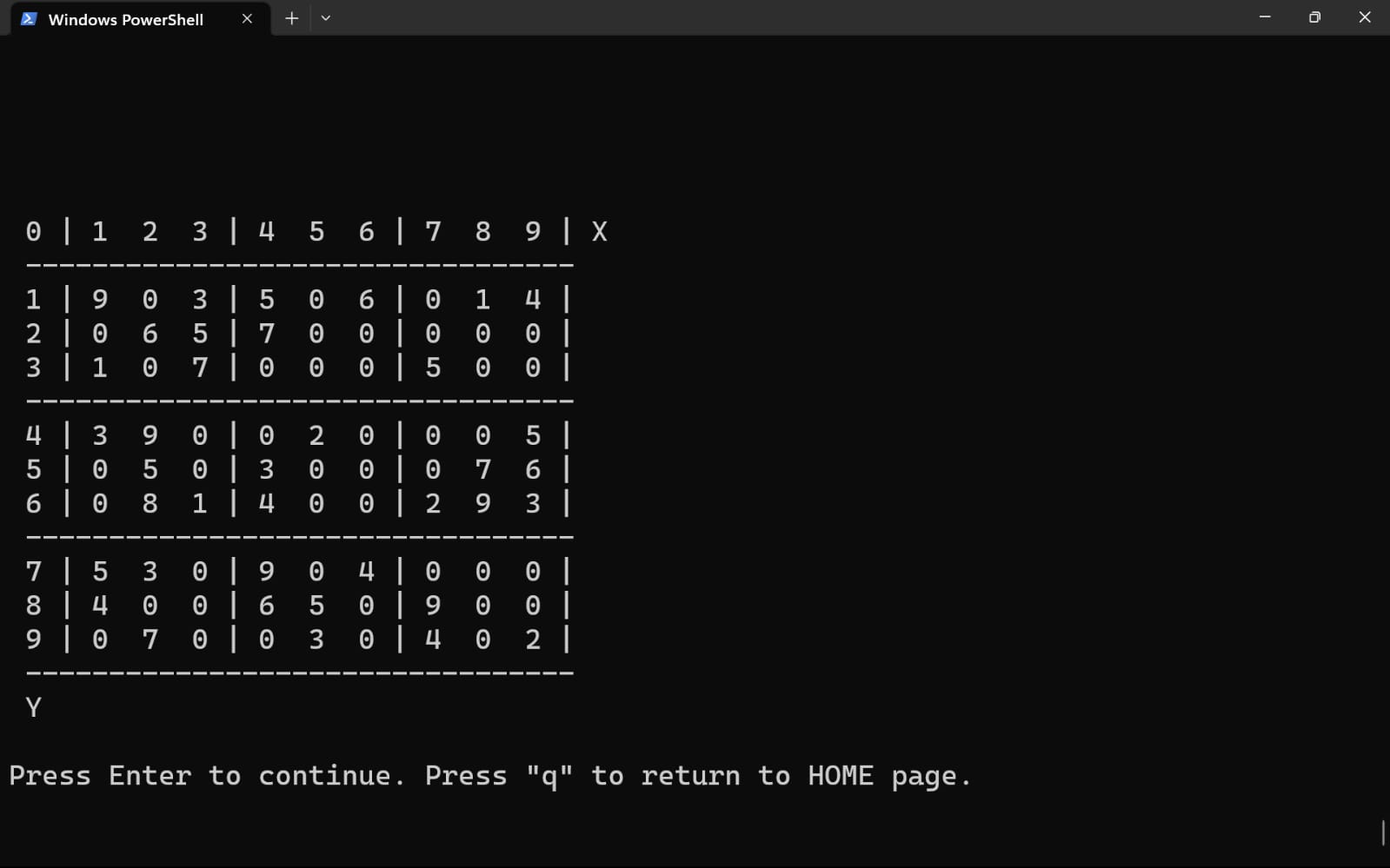
***OUTPUT:***

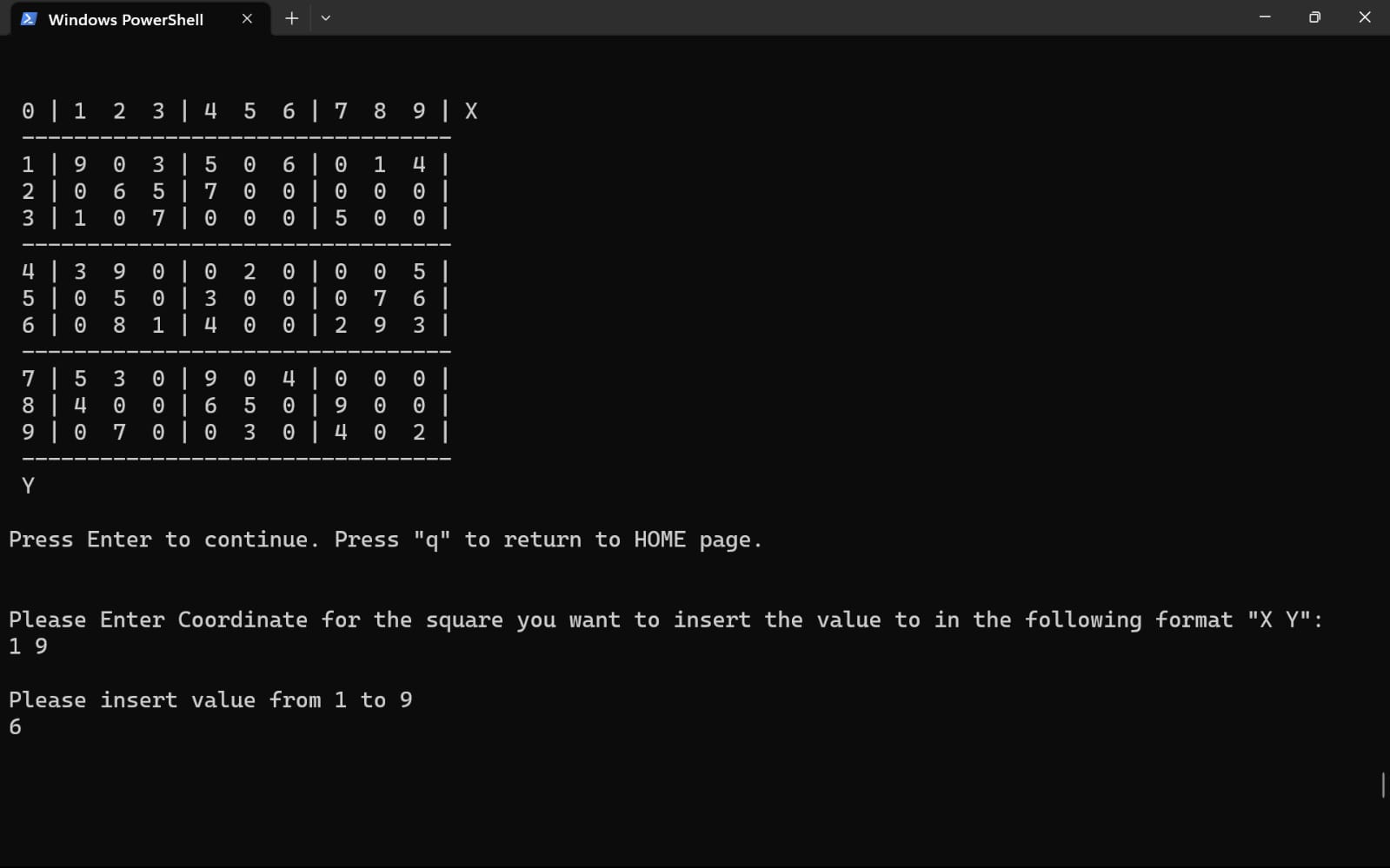


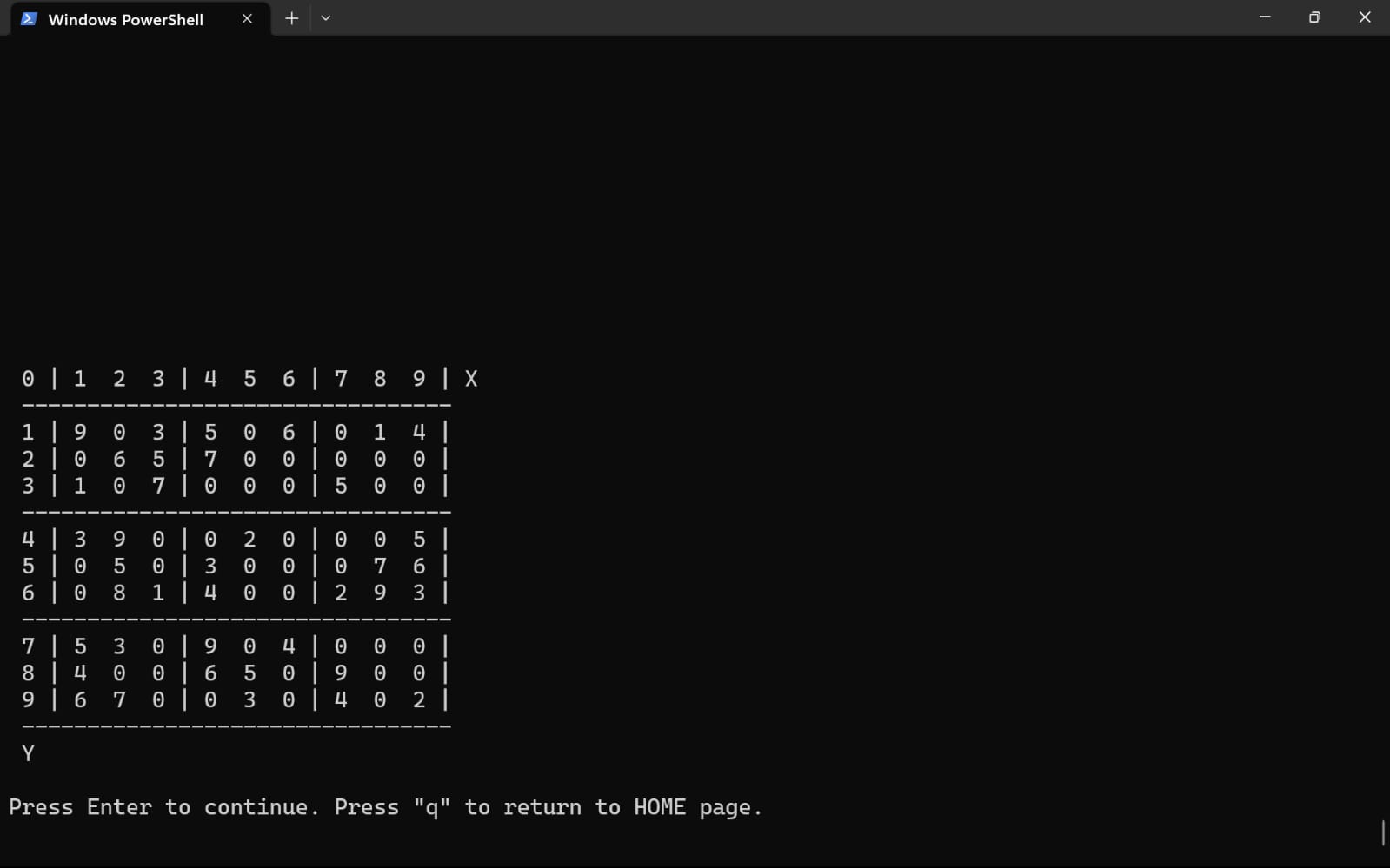


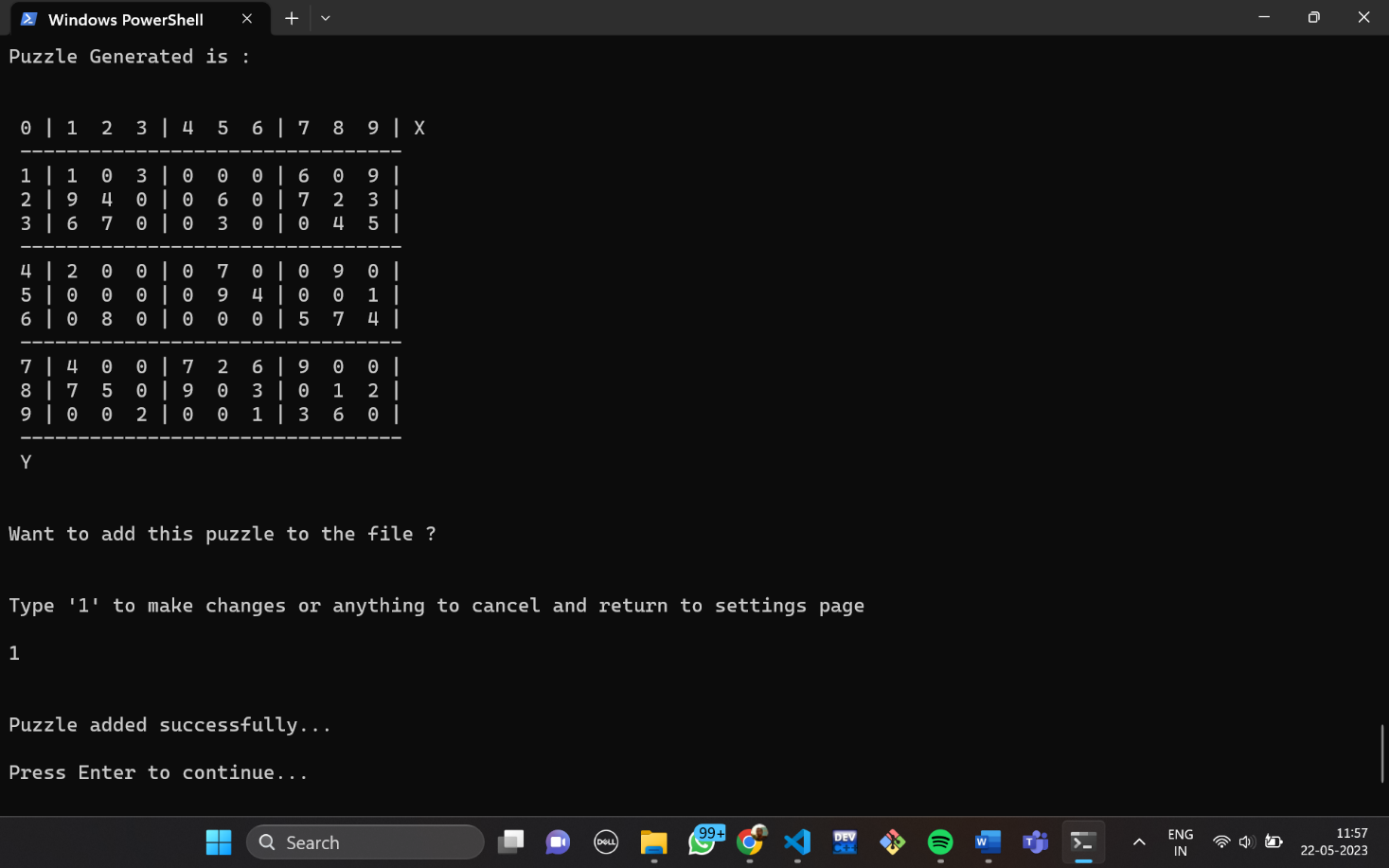










******

**ANALYSIs**

The code includes necessary header files such as **stdio.h**, **stdlib.h**, and **stdbool.h** , as well as user-defined header files "sudoku.h", "ui.h", and "settings.h". All the uses of the function present in the respective file is explained below.

File “ui.h”

* **password()** : In this function a struct is used to store an array of user name and password. Then with if else statement the values are matched with the hardcoded values. If it matches the function returns 1 else it returns 0 with which the function gets called again.
* **clear()** : this function is used to print empty lines 45 times just to clear up the screen .
* **level()** : This function first checks weather the file is empty or not else calculates the number of puzzle present in it and displays the number of level. It then takes the user choice and returns the choice.
* **home()** : This function displays the main home page in which all the choice are present.
* **howtoplay()** : this function prints all the instructions.
* **rules()** : this function prints all the rules.

File “settings.h”

* **clearhistory(**) : This function opens the file in write mode and all the content of the file gets deleted.
* **addautolevel()** : This function is used to generate a puzzle of its own and then print it into the file. The srand(time(NULL) function is used to initialise random number with current time. The “createemptypuzzle()” generates an empty puzzle with 0’s and stores it in the double pointer variable sudoku. Then after the user enters difficulty the file pointer, the number of empty values, the number of random values is generated. Then in the while loop the random variables are inserted in the randomly generated coordinate by checking the availability of the coordinate by using “checkbox()” function. With “solvepuzzle()” function the puzzle stored in variable sudoku is solved. Then using rand() function again random coordinates are generated and the values in sudoku are changed to 0. Then the copy of the sudoku is stored in double pointer variable tempsudoku. If “solvepuzzle(temppuzzle)” is true the program continues else displays unexpected error occoured. Then it aska the user whether the user wants to add this in a file or not if yes it writes it in the file else displays process terminated.
* **cleardata()** : This function opens file in read mode and another file in write mode. Then in the original file it reads each byte and stores it in the other file till 3 puzzles are stored. After completion it renames the original file to the replica file and deletes the original file. Free() function Is used to free the file pointer location.
* **reference()** : This function is used to print the structure of the sudoku puzzle.
* **addlevel()** : This function is used to add new level by manual input. The user selects the difficulty in which the user will store the puzzle. Then a switch case isused to initialise the file pointer. “reference()” function is called to print the sample structure of the puzzle with 0’s. Then using nested for loop the user can enter each value manually representing empty values by 0’s. Then dynamically storing it in the double pointer variable array. Then display the puzzle entered by the user. Using else if check weather the puzzle is solvable or not. If “solvepuzzle(array)” returns true write into file else print not possible.
* **settings()** : This function is used to display all the settings menu option and calls respective function with the help of a switch case.

File “sudoku.h”

* **createemptypuzzle()** : This function is used to create an empty puzzle with 0’s. The memory of the double pointer variable puzzle is allocated dynamically and using nested for loop puzzle is accessed value by value and 0 is stored in it. After that this function returns the value to the double pointer variable in which the function is being called.
* **sizeofhistory()** : This function is used to find the size of the history.txt file. File pointer is initialised first and the file is open in read mode. The “fseek(fp,0,SEEK\_END)” is used to seek the file pointer from 0 to the last of the file. Then “ftell()” is used to find the pointer location in bytes and store it in len variable and return the len variable.
* **storeinhistory()** : This function is used to store the userpuzzle passed as parameter to file history.txt. Using nested for loop print the values of userpuzzle into the file till 8th element in a row separated by ‘ ’ and store the last element ending with ‘\n’. with if check if row no ==2 or 5 the print ‘\n’. After completion printf(‘\n’) to ensure proper structure.
* **createcontinuepuzzle()** : This function takes the puzzle stored in history.txt and returns it to the double pointer variable where it is being called. The puzzle is first taken from file and stored in static 2d array arr. Then puzzle gets memory assigned dynamically and values are stored in puzzle from arr using nested for loop. The function returns puzzle.
* **createpuzzle()** : This function returns the puzzle stored in the file. The user selects the difficulty of the game. Then according to the file level is displayed using “level()” function and number of levels returned is stored in a variable. If there is no level the user Is directed to “settings()” page else the file pointer is set to the desired location. Then the puzzle is first taken from file and stored in static 2d array arr. Then puzzle gets memory assigned dynamically and values are stored in puzzle from arr using nested for loop. The function returns puzzle.
* **printpuzzle()** : This function is used to print the puzzle passed as parameter. The coordinates are displayed first then using nested for loop all the values along with the separators “|, ,\n” using if else statement.
* **checkavailable()** : This function is used check weather there are any 0’s present in the puzzle or not. It returs true is present else returns false.
* **Checkbox()** : This function is used to check whether the value passed as parameter is already present in the puzzle or not. The first for loop checks whether the value is present in the vertical column. The second for loop checks whether the value is present in the horizontal row or not. The nested for loop is used to check whether the val is present in the 3x3 grid. In all cases the function returns true if value not found else returns false.
* **solvepuzzle()** : This function is used to solve the puzzle automatically and also return true if solvable else return false. The “checkavaiable()” function checks whether the position is available or not. The for loop is used to take values from 1 to 9 and check whether value Is already present in the row, column or 3x3 box. If yes check puzzle solvable or not else set value to 0. After completion return false.
* **copypuzzle()** : This function is used to copy the puzzle passed as parameter to a newpuzzle and return the new puzzle. The newpuzzle is allocated memory dynamically and then the value of puzzle is stored in it using nested for loop.
* **userpuzzle()** : This is ysed to take user input and check its validity to store it in userpuzzle. Firstly it checks whether there are any empty places present in the userpuzzle or not by “checkavailable()”. Using for loop check whether user want to enter value or quit. Using while loop take coordinates and input and check its validity. Take input for value to be placed in the userpuzzle and check its validity with while loop. Check weather the value entered by the user is already present in the puzzle in row, column, or 3x3 box with “checkbox()”. If it returns true the assign the value if it returns false print incorrect value. Then copy the userpuzzle to temppuzzle. Now check using “solvepuzzle()” whether the puzzle is solvable or not. If it returns false print incorrect value. Then “printpuzzle()” is used to print the userpuzzle. At last check whether userpuzzle is same as puzzle which was taken from the file. If yes call “storeinhistory()” .

1. Flow of the Program is discussed below.

2. And static integer call is defined globally for executing the “password()” function only once.

3. The functions ‘**clear()’** and **‘home()’** are called to clear the screen and display the home page of the Sudoku game.

4. The user is prompted to enter their choice, and the input is stored in the variable ‘**choice’** using ‘**scanf’.**

5. A switch statement is used to perform different actions based on the value of ‘**choice’**:

**Case 1:**

When this case is in use the “sizeofhistory()” function checks weather the history.txt file is empty or not by seeking the file pointer to the last and returning the bite size. If the file is empty the user has to start a new game first else the entire puzzle is stored in the double pointer variable puzzle using “createcontinuepuzzle()”.Then the userchoice() function is being executed which performs the main action discussed further. Then the dynamically allocated memory is cleared using free() function.

**Case 2:**

When this case is in use “createpuzzle()” is executed in which the user has to select the level of difficulty. After the input the file pointer opens the file accordingly using a switch case. If the file is empty the “addautolevel()” function gets executed through which the program generates a sudoku template and the user can add it in the file. If there are puzzles present in the file the user selects the level which is executed by “level()” function. A copy of the puzzle is stored in userpuzzle and temppuzzle double pointer variables for further use. The “printpuzzle()” function prints the puzzle which is passed in the argument. The “userchoice()” function is executed next. If the user wants to make changes then the “checkavailable()” function checks weather any empty places is available in the puzzle or not else displays puzzle complete and breaks the loop. Take user input of the coordinate the user want to insert value in. check for validity of the coordinate if not possible enter again. Take value to be inserted if not valid take again. The “checkbox()” function is used to check if the value is already present in the row or column or 3x3 box. If already present display message if not change the value to the coordinate in the userpuzzle double pointer variable. Update the temppuzzle as same as userpuzzle. Then the “solvepuzzle()” is executed if it returns true then value if correct else the value has to be entered again. If value is inserted to the puzzle then store the value in history.txt using “storeinhistory()”.

**Case 3:**

When this case in In use the “howtoplay()” function is executed it prints all the instructions.

**Case 4**:

When this case in In use the “rules()” function is executed it prints all the rules.

**Case 5:**

When this case is in use the “settings()” function is executed. Which displays the options. The user can add a new level by auto input or can manually insert each value or can reset progress or can reset game data.

**Case 6:**

Exit game.

**Default Case** :

Calls the main function again.

6. Finally, the main function returns 0, indicating successful execution.

Overall, this code represents a Sudoku game with options for continuing a saved game, starting a new game, viewing instructions and rules, adjusting settings, and exiting the game.

**CONCLUSION:**

In conclusion, the application of graph theory in solving Sudoku puzzles has proven to be a highly effective and efficient approach. By representing the Sudoku grid as a graph and utilizing various graph algorithms, we were able to devise a systematic method to find the solution to even the most challenging Sudoku puzzles.

The key advantage of using graph theory lies in its ability to capture the intricate relationships between the cells in the puzzle. By modelling the puzzle as a graph, we were able to leverage well-established graph algorithms, such as depth-first search and backtracking, to systematically explore the solution space and eliminate invalid possibilities.

Additionally, the graph-based approach provided a clear and structured framework for tackling Sudoku puzzles of varying difficulty levels. It allowed us to implement algorithms that could adapt to different scenarios, taking advantage of constraints and logical deductions to reduce the search space and find solutions efficiently.

Moreover, the graph theory approach to Sudoku puzzle solving opens up possibilities for further exploration and improvement. Researchers can delve deeper into graph-based strategies, such as constraint propagation and graph colouring techniques, to enhance the efficiency and accuracy of solving Sudoku puzzles.

Overall, the successful application of graph theory in solving Sudoku puzzles showcases the power of mathematical modelling and algorithmic techniques in tackling complex problems. By leveraging the inherent structure and relationships within Sudoku puzzles, we can continue to develop innovative approaches and algorithms that not only solve Sudoku efficiently but also contribute to advancements in other related domains, such as combinatorial optimization and constraint satisfaction problems.