Computer Science NEA

**Sudoku**

Table of Contents

[1 ANALYSIS 6](#_txg0hzv2floc)

[1.1 Introduction to Sudoku 6](#_eezviev1z3lw)

[1.1.1 What is Sudoku? 6](#_yfrv9rdujf0v)

[1.1.2 How to play Sudoku? 6](#_7ukmv1tf5om1)

[1.2 Problem Definition 7](#_suqgwcr9lo8r)

[1.2.1 How does Sudoku solve the problem? 7](#_u3g4wejvpkry)

[1.3 Analysis of Current Systems 8](#_zf6ed96dh7b3)

[1.3.1 Sudoku.com 8](#_hn8fuz9b3g86)

[1.3.2 WebSudoku.com 9](#_7xjuqipbjqxt)

[1.3.3 New York Times Games Sudoku 10](#_xynj489tci)

[1.3.4 Current Systems Flowchart 11](#_u1httoqpgthw)

[1.3.5 Current Systems IPSO Chart 11](#_cjizm7nghxdn)

[1.3.6 Current Systems Data Dictionary 12](#_utubxsxs04el)

[1.4 Interview 13](#_ew934t8hbswd)

[1.5 Potential End Users 14](#_odcm2qxrm729)

[1.8 Limitations 15](#_43efp1ies1ww)

[1.9 Proposed Solution 15](#_j7it1oh7i0i0)

[1.6 Objectives 16](#_i5dv7n88mo4m)

[2 DOCUMENTED DESIGN 18](#_y1962r8a3ov7)

[2.1 System Design 18](#_uapmiqg33sop)

[2.1.1 Top Down Diagrams 18](#_9n77evvh7528)

[2.1.2 IPSO Chart for Game 20](#_1woksr4kvyup)

[2.1.3 IPSO Chart for Generator 21](#_bqvdrfhphhsb)

[2.1.3 Overall System Flowchart 22](#_2av0wz4ypoxb)

[2.1.4 Data Dictionaries 23](#_jcwhm51v00zj)

[2.2 Puzzle Class & Generator 24](#_umrsn97qfi6o)

[2.2.1 Solve Algorithm 25](#_8c525owc4ekr)

[2.2.2 Generating Algorithm 26](#_k86344n7ixaq)

[2.2.3 Puzzle Rotator and Flipper 30](#_wyk84cdfsat0)

[2.2.4 Classification and Storage of Generated Puzzles 31](#_o1ncnfgpmhcf)

[2.3 Interface Design 31](#_xsl6r29wiqx5)

[2.3.1 Main Menu 31](#_4ytfr4e1u8sb)

[2.3.2 Classic Sudoku 32](#_q5m4emuyhm5c)

[2.3.3 Multiplayer Sudoku 33](#_1t64hvt0ot8d)

[2.3.4 Account Manager 35](#_244wefmnq3x6)

[2.3.5 Best Times 36](#_kaxirw8va5hp)

[2.3.6 Class Diagram for Screens 37](#_z6oc85dz4olr)

[2.4 Classic Sudoku 38](#_vkq8lq4sqkyv)

[2.4.1 Workings of the Classic gamemode 38](#_nuawmpc9mqg2)

[2.5 Client Server Model 39](#_mq7t2vz4tkw8)

[2.5.1 Workings of the model 39](#_gv8qrmpg9beu)

[2.5.2 Client class for client-side connection module 39](#_qbukti1zkcng)

[2.5.3 Communication between client and server 40](#_12ywargq86h0)

[2.5.4 Flowchart for Client Side 41](#_884pq4fg44kd)

[2.5.5 Flowchart for Server Side 43](#_x127o4dz3x08)

[2.6 Account & Database System 44](#_kvg82omde4g)

[2.6.1 Entity Relationship Diagram 44](#_bpn1ahn3zjgq)

[2.6.2 Register 45](#_karcoohpyogg)

[2.6.3 Login 45](#_yzgiv4mdz4l8)

[2.6.4 Account Database 45](#_d565t1sdf7n2)

[2.6.5 Best Times Database 45](#_e8gk8s6sxg83)

[2.7 Information to save locally in the game's files 45](#_3cxj52teuclw)

[3 TECHNICAL SOLUTION 46](#_h7hiqjtn14hz)

[3.0 Details and Techniques Used 46](#_1oqb2rhvtoa4)

[3.0.1 List of Techniques Used 47](#_wzyvlp22nywx)

[3.0.2 List of Objectives met 49](#_exci0rbitnhu)

[3.1 Puzzle Generator 50](#_4ynkz0jm7bxi)

[3.1.1 Puzzle Class 50](#_y41yvo1yox6m)

[3.1.2 Components of Solve function 51](#_iqxst158k3in)

[3.1.3 Puzzle X String Converters 57](#_7aqeccqmf53i)

[3.1.4 Generator 58](#_7saiq2802v1x)

[3.1.5 Puzzle Rotator and Flipper 62](#_bbkx4wcnvau7)

[3.1.6 Puzzle Generation and Storage 63](#_3pxytipwhbcy)

[3.2 Main Game Overview 1](#_qrbl47d4xuf6)

[3.2.1 Game Class Initialisation + Methods 1](#_tekyl3ohqbt0)

[3.2.2 App Class 1](#_42k54p7zptcc)

[3.2.3 Screens 1](#_j9mcglqor95j)

[3.2.4 Menus 1](#_mvtiiq9b91so)

[3.2.5 Classic Menu 1](#_xl2a96sdxhe6)

[3.2.6 Multiplayer Menu 1](#_x9n5fdgcc7cn)

[3.2.7 Account Menu 1](#_wpzrh76mxm9w)

[3.3 GameScreen 1](#_2ftpcmxxj02j)

[3.3.1 Cell 1](#_kl36zhoc8xb7)

[3.3.2 NumberInput 1](#_96e1d9yozgw2)

[3.3.3 Timer 1](#_11qbuke2zkp)

[3.3.4 GameScreen 1](#_o0fpw0z2b0x3)

[3.3.5 Classic Game 1](#_e1boqxadeeek)

[3.3.6 Multiplayer Game 1](#_14ossvbnejww)

[3.4 Account Management Screens 1](#_vwoienczyox0)

[3.4.1 Login Screen 1](#_wnxu23i8jhf2)

[3.4.2 Register Screen 1](#_44b8sva33zt0)

[3.5 Informative Screens 1](#_aaf9vwwgxo4i)

[3.5.1 Best Times 1](#_v9olsuz23uah)

[3.5.2 How To Play Screen 1](#_d4woscrr507y)

[3.6 Server side (Basic) 1](#_qs0dakwld70k)

[3.6.1 Main Server Setup 1](#_uugq0xfmv9ir)

[3.6.2 Client thread class 1](#_vxnk3hdofedz)

[The 3.6.3 Client handler 1](#_ehk46db967zx)

[3.6.4 Login 1](#_4pk5jvd4tpby)

[3.6.5 Register 1](#_prmkk0mxwzsr)

[3.6.6 Database 1](#_hhnmjyrupndv)

[3.6.7 Update Best Times 1](#_rnu04bei6ie)

[3.6.8 Match Players 1](#_nzal5w3mi4ve)

[3.7 Server side (Multiplayer) 1](#_2dxs39v94al6)

[3.7.1 Create Match Function 1](#_rw0h4o6sqoc)

[3.7.2 Game Class Initialisation 1](#_n8h92nalgwih)

[3.7.3 Game handler method 1](#_gb1aj1w8mbdv)

[3.7.4 Game puzzle handling methods 1](#_yxt8qn242y6q)

[3.8 Client Side 1](#_z6jbvuxdprzk)

[3.8.1 Client Class Initialisation 1](#_ged8jkuzq6vn)

[3.9 Other Algorithms and Techniques 1](#_knmuz0gnxnuo)

[3.9.1 Stack 1](#_tk5ier10e449)

[3.9.2 Circular Queue 1](#_uhlm6nn1y060)

[3.9.3 Hashing 1](#_eaxqa3x4kk2d)

[4 TESTING 1](#_ga4xo1jzizy)

[4.1 Test Plan 1](#_vnnn9dk3tl8o)

[4.2 Testing and Evidence 1](#_wrsdm5ehlnw0)

[4.2.1 Testing Accounts System 1](#_4gmp5uorhru8)

[4.2.2 Testing Classic Mode 1](#_x5tqbmcaav37)

[4.2.3 Testing multiplayer 1](#_c1ul6g7rgdxk)

[4.2.4 Testing the Puzzle Generator 1](#_91lza2wq1fdh)

[4.3 Testing Videos 1](#_ai2teb2qca86)

[4.3.1 Navigating the GUI 1](#_u2t8ypu2xcvm)

[4.3.2 Account System 1](#_igs57kw4iiej)

[4.3.3 Multiplayer Gameplay 1](#_jhwt7vn01wqn)

[5 EVALUATION 1](#_opzftf8ybitf)

[5.1.1 Revisiting the Objectives 1](#_rlaxg98wbffq)

[5.1.2 End User Feedback 1](#_2yd3w6zinun)

[5.14 Improvements to be made 1](#_78o4uxcymnkh)

[Appendix 1, Generator.py 1](#_3grhx8ees4hi)

[Appendix 2, main.py 1](#_8aa1yd35hrh3)

[Appendix 3, Sudoku.kv 1](#_p8o4o6kuuyg)

[Appendix 4,networking.py 1](#_1gfhu9h4bpr)

[Appendix 5, server.py 1](#_1nfvcq9nro5s)

# 1 ANALYSIS

## 1.1 Introduction to Sudoku

### 1.1.1 What is Sudoku?

Sudoku is a puzzle game consisting of 9 3x3 grids which make up a greater 9x9 grid, the grid comes with some cells already filled with numbers. The player must fill the entire grid so that each row, column and 3x3 grid has the entire set of numbers from 1 to 9 without repeating numbers in those sections.

The cells with numbers already given are known as clues, the puzzle on the left has 32 clues.

### 1.1.2 How to play Sudoku? [[1]](#footnote-0)

A starting strategy would be to find a row or column with the fewest cells left to fill and place the remaining numbers in an empty cell without placing a number that is already in that empty cell’s subgrid, row and column.

## 1.2 Problem Definition

The issue of reduced concentration and shorter attention spans, a common symptom of ADHD, has become increasingly prevalent among the general population. The surge in platforms like TikTok, Instagram Reels, and YouTube Shorts has captivated people from all walks of life with their brief content, resulting in addictive behaviours and a decline in attention span. As a consequence, individuals find themselves consistently indulging in these short-form videos for quick moments of instant gratification, often to the detriment of their critical thinking skills, leading to a state of perpetual content consumption.

### 1.2.1 How does Sudoku solve the problem?

Playing Sudoku can help solve that. Sudoku is a game that relies on deductive reasoning, a skill that utilises both memory and logical thinking, which improves as you solve more sudoku puzzles. But its main takeaway for this project is how it trains a person's concentration to be stronger and increases their attention span as playing requires holding one's train of thought to stay focused.

I will be creating an app where players can solve puzzles individually and competitively against other players online as the 1v1 race finishes. Some people may find the idea of solving a sudoku puzzle to be boring, so adding a competitive aspect to it promotes the game to a wider audience who want to experience a more fun way to play sudoku.

Aside from the above additional benefits include but are not limited to the following:

* Improved attention to detail
* Improved pattern recognition
* Improved short-term memory, due to having to remember and take note of the numbers and patterns you encounter
* Entertainment and relaxation, some will find it meditative, it will be challenging but not taxing.

Multiplayer Sudoku will serve as a way for friends to compete with each other mostly for fun, but will also motivate them to play more Sudoku to become better than their friends.

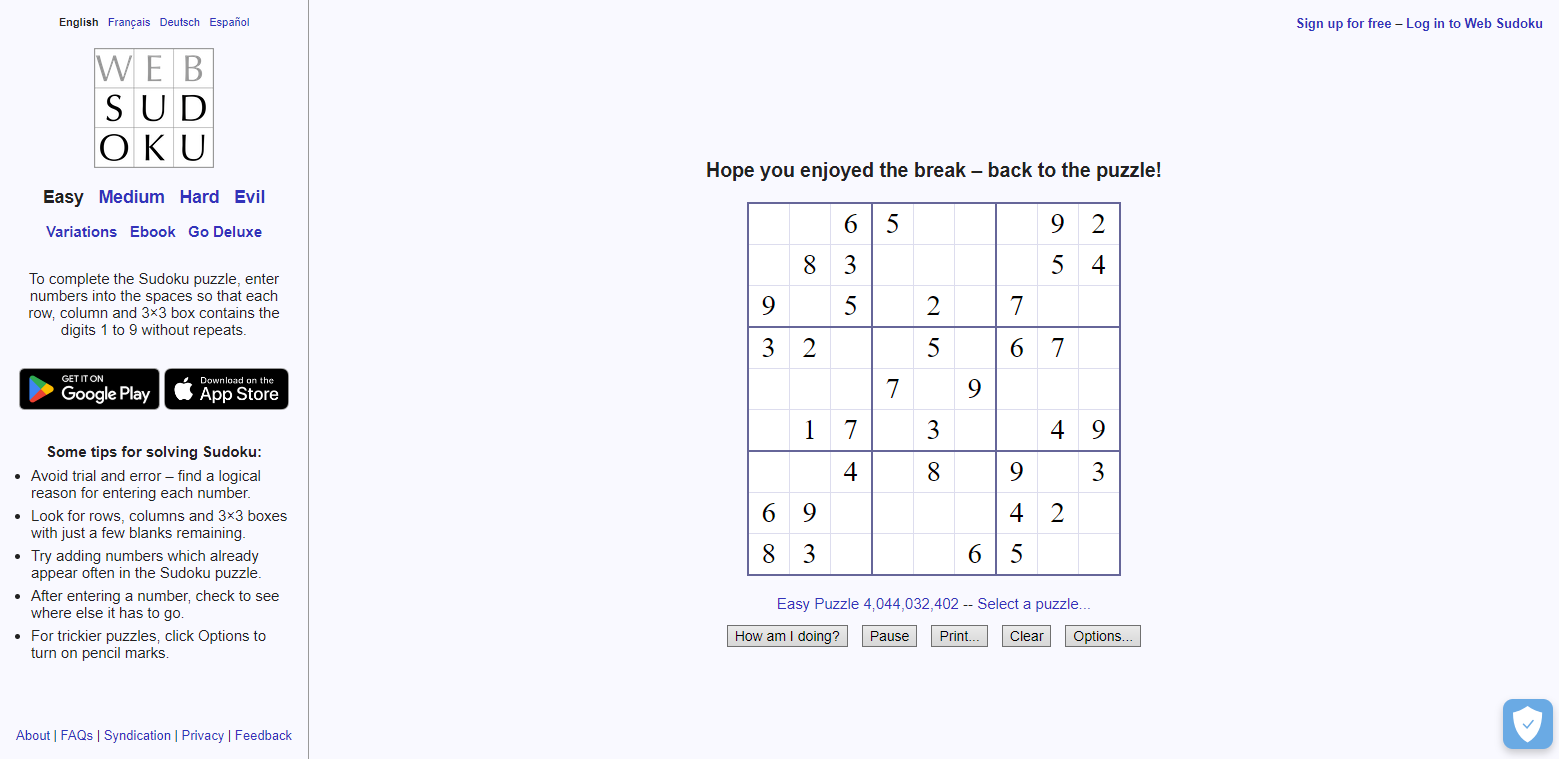
## 1.3 Analysis of Current Systems

### 1.3.1 [Sudoku.com](https://sudoku.com/)

### 

* 5 difficulties, Two game modes (Other is a variant, killer/sumdoku)
* Ability to select the cell and enter a number from the keyboard and select the cell and click on the number pad on the side. (Can be played on a touchscreen), so controls are very accessible
* New game button, Undo and erase options
* Notes option(Tedious have to toggle between it), Hints Option
* Daily challenge and awards system(trophies)
* Killer sudoku(AKA sumdoku) is a somewhat easier variation of the game where numbers within dotted line cages have to sum to the little number noted within the cage.
* Has a mistakes counter that goes up to 3, instead of game over, infinite second chances are provided. I believe this may be too forgiving and should maybe have the option to toggle between infinite chances and not.

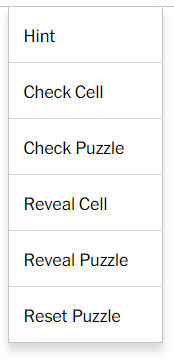
### 1.3.2 [WebSudoku.com](https://www.websudoku.com/)



* Instead of instantly checking and telling the user that a number placed is wrong, there is a ***“how am I doing”*** that when clicked tells the user how many cells are wrong, allowing the user to figure the error out themselves.
* 4 Difficulties
* Daily change of Sudoku variations
* Minimal UI
* Pause option for timer
* Print option to download and print the puzzle to be played on paper.
* Has notes option
* Does not have an undo button
* Has a clear button to erase all numbers but the starting ones
* Has 4 language options
* Has toggleable timer
* Has account system

### 1.3.3 [New York Times Games Sudoku](https://www.nytimes.com/puzzles/sudoku/medium)

### 

* Includes Timer
* 3 difficulties
* Check guesses when entering numbers, toggleable, when on makes the game easier
* The hint button provides the cell at which players should start at
* Ability to check cells individually
* As well as check the whole puzzle
* Ability to reveal cells and puzzle(individually)
* Reset button
* Erase button
* Account system

## 

## 

## 

### 1.3.4 Current Systems Flowchart



This flowchart how most systems set up the game and check the game states and respond to those changes.

### 1.3.5 Current Systems IPSO Chart

| **IPSO** | ***Program section*** | ***Item*** |
| --- | --- | --- |
| *Input* | Account Management | Username  Password |
| Game | Number input  Remove number input  Cell location input  Exit button  Pause button  Toggle Candidates button  Show Solution button |
| *Processing* | Account Management | Check username length  Check password length  Check passwords match  Save details if remember login is true |
| Classic Game | Check cell input for any collisions  Check the puzzle grid state for the solution  Convert time in float to clock format |
| *Storage* | Account player data | Username, password, best times |
| *Output* | Account Management | Display server connection status |
| Classic Game | Display Puzzle grid  Display Number grid  Display Timer  Display Exit button |

### 1.3.6 Current Systems Data Dictionary

| Data item | Data type | Validation | Example |
| --- | --- | --- | --- |
| username | string | length >= 7 | “username” |
| password | string | length >= 7  PW1 == PW2 | “password” |
| Hashed Password | String in hexadecimal | | “ab6f2840c” |
| Difficulties | List of strings | | [“easy”, “medium”, “hard”] |
| Cashed puzzle for resume gameplay | Json file | | '{"puzzle":  "0801000005000391080007000226950000470560800000000000000803007400070000000000050”", "time":30,  "Progress": 100 }' |

#### 

## 1.4 Interview

1. What features do you enjoy in existing versions of the game?

|  |
| --- |

1. What are some negatives in the existing version of the game?

|  |
| --- |

1. What new features do you think would improve user experience?

|  |
| --- |

1. Is there anything else you would like to see implemented in the game?

|  |
| --- |

#### **Summary of answers from potential end users**:

Having a range of difficulty levels for variety in gameplay is especially important for those who want to learn how to play and those looking for a challenge.

Suggested a single-player time attack mode, beat the clock style, whilst another wanted a multiplayer style versus mode to see who can complete the same puzzle the fastest.

Suggests a tutorial should be available for beginners as well as tips, tricks and strategies for intermediate players, could implement interactive tutorials to better engage with the users.

Suggests a public leaderboard to showcase the top players of the game, showing their top time and the puzzle used, could have a feature to contest leaderboard times for the specific puzzle.

Suggest a solver where the user can enter in their own grid and have it solved or declared unsolvable.

## 1.5 Potential End Users

All ages(7+), anyone who likes sudoku and people looking to get started playing Sudoku.

It improves people’s concentration, and focus, which is great for today’s younger generation who spend their time engaging in tons of short-form content, the game can help them improve their attention span.

* Those who wish to play Sudoku on a computer
* Those who wish to learn how to play Sudoku
* Those who wish to play Sudoku in a multiplayer setting

## 

## 1.8 Limitations

* Limited to the English language
* As this is the first application I have developed, the performance may not smooth
* Not enough time to thoroughly learn software development and networking
* Limited multiplayer capabilities won't be able to handle 50+ users for multiplayer

## 1.9 Proposed Solution

An application made in Python using Kivy for the GUI, with a client/server networking build for the multiplayer mode using the socket and threading modules. I will be using Kivy as once the application is built it can be used across many devices including smartphones, as well as being easy to design application with

The puzzles to be used for the games will be generated then categorised by difficulty and stored.

Puzzle patterns can be re-used by rotating and flipping the grid, and swapping the placements of numbers around, so overall from one puzzle generated, we can have 8 different puzzles.

Puzzles will be generated outside of the main application to make sure gameplay is smooth and does not take up too much time. Generated puzzles will be sorted into difficulties by the number of clues(given numbers) they have.

## 

## 1.6 Objectives

| No. | Objective | Performance Criteria |
| --- | --- | --- |
| 1 | Create an app where users can solve sudoku puzzles | Must have a variety of puzzles to play with a variety of difficulties. |
| 2 | The app must be responsive and easy to navigate | Must have easy controls so accessible to all peoples and be able to work smoothly. |
| 3 | Generate and store unique sudoku puzzles for the app. | The puzzle must be sorted into 4 difficulty categories; easy, normal, hard and extra hard. Must be able to recycle the puzzle by flipping and rotating the grid to create more puzzles. |
| 4 | Implement Classic mode | In a single-player game, the user can solve a puzzle of chosen difficulty and be timed for it. |
| 5 | Create a responsive GUI for the game mode | The puzzle grid must respond to number placements, highlight cells for collisions and have a timer that can pause. |
| 6 | Implement Multiplayer mode | Two users must be able to match online and race to see who can finish the puzzle first |
| 7 | Create a server to connect to clients, manage accounts and facilitate the multiplayer mode | Accept requests from clients and deliver appropriate responses. |
| 8 | Create an Account system | Users are to be able to register and login to accounts as well as have payer data saved online. |
| 9 | Create a database to store account information for the account system | Consisting of two tables, Accounts and Best Times.  Accounts ← username, password  BestTimes ← username, easy, normal, hard, extra\_hard |
| 10 | Create a matching system for the multiplayer mode | Players are able to randomly match against other players.  To enter players into queues of chosen game difficulty and match them to other players. |
| 11 | Create the multiplayer game system | To send each player the puzzle grid and to receive each player's progress and relay it to the other player till one of them wins. |
| 12 | Store the user’s username, password and best times in a local file. | A username and password will be used to automatically login to the server. |
| 13 | Create a tutorial for the users | Will explain how the game works and introduce the users to starting strategies |

# 

# 

# 2 DOCUMENTED DESIGN

## 2.1 System Design

### 2.1.1 Top Down Diagrams

#### Main Program

|  |
| --- |

The Main program is split into 4 parts

#### Server

|  |
| --- |

The server is continuously open to listening for connections and accepts connections and requests and proceeds from there.

### 

### 2.1.2 IPSO Chart for Game

| **IPSO** | ***Program section*** | ***Item*** |
| --- | --- | --- |
| *Input* | Account Management | Username  Password  Remember login |
| Classic Game | Number input  Remove number input  Cell location input  Exit button  Pause button |
| Multiplayer Game | Number input  Cell location input  Exit button  Receive opponent puzzle state |
| *Processing* | Account Management | Check username length  Check password length  Check passwords match  Save details if remember login is true |
| Classic Game | Check cell input for any collisions  Check the puzzle grid state for the solution  Convert time in float to clock format |
| Multiplayer Game | Check cell input for any collisions  Check the puzzle grid state for the solution  Apply opponent puzzle state to opponent grid |
| Multiplayer Matching | Check if online  Send a match request to the server |
| *Storage* | Account player data | Username, password, best times |
| *Output* | Account Management | Display server connection status |
| Classic Game | Display Puzzle grid  Display Number grid  Display Timer  Display Exit button |
| Multiplayer Game | Display Puzzle grid  Display Number grid  Display Timer  Display Opponent grid  Display Exit button |
| Best Times | Display best times |

### 

### 2.1.3 IPSO Chart for Generator

| **IPSO** | ***Program section*** | ***Item*** |
| --- | --- | --- |
| *Input* | Production | Number of puzzles to generate |
| Sorting & Storing | Puzzle files |
| *Processing* | Production | Fill empty grid  Remove digits from the grid  Check the number of solutions to the grid |
| Sorting & Storing | Check if the puzzle already exits |
| *Storage* | Sorting & Storing | Write a list of puzzles to the ends of their respective puzzle files |
| *Output* | Production | Number of puzzles generated  Time taken to generate |
| Sorting & Storing | Number of duplicates found  Outliers |

### 

### 2.1.3 Overall System Flowchart

|  |
| --- |

The classic system starts out with setting up the grid and timer and then continuously checks if the grid is full and valid, at the end when the grid is full it logs the time elapsed.

The multiplayer system starts out with connecting the players and sending the puzzle out. The server then waits to check which one of the players finishes first and relays the message to the other player.

### 2.1.4 Data Dictionaries

#### Main App

| Data item | Data type | Validation | Example |
| --- | --- | --- | --- |
| username | string | length >= 7 | “username” |
| password | string | length >= 7  PW1 == PW2 | “password” |
| Hashed Password | String in hexadecimal | | “ab6f2840c” |
| best times | List of Floats | | [1, 2, 3, 4] |

#### Puzzle Generator

| Data item | Data type | Validation | Example |
| --- | --- | --- | --- |
| puzzleString | String of 81 characters | | “080100000500039108  000700002269500004  705608000000000000000803007400070000  000000050” |
| puzzleGrid | 2D list of integers, 0-9 | | [[0, 6, 0, 2, 0, 0, 0, 7, 5], [0, 0, 9, 3, 0, 0, 0, 0, 0], [0, 7, 0, 1, 0, 6, 3, 8, 0], [0, 9, 0, 7, 0, 0, 1, 0, 0], [0, 4, 6, 0, 1, 3, 0, 0, 0], [7, 1, 0, 0, 6, 0, 0, 4, 0], [8, 0, 4, 0, 3, 0, 0, 0, 9], [0, 0, 0, 6, 0, 8, 4, 3, 2], [0, 3, 7, 0, 2, 0, 0, 0, 0]] |
| candidates | 2D list of sets | | [[{3, 4, 5}], [{6, 7, 8}]] |
| Row, Col | Integers | | (3, 2) |
| digitsToRemove | Integer | | 35 |
| digitsRemoved | integer | | 35 |
| solutions | integer | | 2 |
| clues | integer | 81 - digits removed | 49 |
| sortedCandidates | List of tuples that each contain a (set, integer, integer)) | | [ ( {2, 3} , 6 , 7 ) ] |

## 2.2 Puzzle Class & Generator

***This fulfils Objective 3***

Puzzles will be interacted with in the game using a puzzle object, with the grid as an attribute and methods to interact with it.

The grid will be represented and interacted with as a 2D array.

| CLASS Puzzle():  DEFINE \_\_**INIT\_\_**(SELF, string = NONE):  IF STRING:  SELF.GRID ← Translate string into puzzle as 2d array  ELSE:  SELF.GENERATE() |
| --- |

### 

### 

### 2.2.1 Solve Algorithm

The algorithm to solve a sudoku puzzle is a basic depth-first search algorithm that utilises backtracking. This will be a Group A algorithm with recursion.

| SOLVE(GRID):  Pos = FIND\_EMPTY\_SPACE in GRID  IF no empty space is found  RETURN True # Indicating that the puzzle is solved  FOR n ← 1 TO 9  IF CHECK(GRID, Row, Col)  Place n into that empty space  IF SOLVE(GRID) == TRUE # Puzzle solved  RETURN TRUE  GRID[Pos] ← 0 #Puzzle not solved, backtrack  RETURN False # No Solutions |
| --- |

The algorithm works by continuously looking for empty spaces throughout the grid till it has none. When it finds an empty space it tries placing a number in there if it is valid then it recursively calls itself to continue the search. If at any point there is no possible solution at that point in the search the current function call ends and returns back to when the previous function call was made and undos the placement of the cell and tries another number and then goes on to continue the search. The ability to go back and undo the placement of the cell is known as backtracking and this is how I will be solving the grid.

| FIND\_EMPTY\_SPACE(GRID):  FOR Row in RANGE(0, 9) # ~ 0 to 8 inclusive  FOR Col in RANGE(0, 9)  IF GRID[Row][Col] == 0:  RETURN (Row, Col) |
| --- |

| CHECK(GRID, Row, Col, n):  IF n IN GRID[Row]  RETURN FALSE  FOR i ← 0 TO 8  IF n = GRID[i][Col]  RETURN FALSE  boxRow = Row // 3 \* 3 # (//) means integer division  bowCol = Col // 3 \* 3 # Possible values for boxRow/boxCol are 0, 3, 6  FOR i ← boxRow TO boxRow + 2  FOR j ← boxCol TO boxCol + 2  IF n == GRID[i][j]  RETURN FALSE  RETURN TRUE |
| --- |

### 2.2.2 Generating Algorithm

#### Generator

| GENERATE():  GRID = FILL\_GRID()  CLUES = REMOVE\_DIGITS(GRID) |
| --- |

1. Create empty grid
2. Randomly insert numbers 1 to 9 into the grid about 20-25 times
3. Apply Sudoku solver to the grid

#### Grid Filler

| FILL\_GRID():  Solved ← False  While Solved = False  GRID = Make an empty grid  Count = dictionary  FOR i ← 1 to 9  Count[i] = 0 #Numbers cannot appear more than 9 times  N = Randint(20, 25)  WHILE N > 0  X = Randint(1, 9)  Row = Randint(0, 8), Col = Randint(0, 8))  IF Count[X] < 9 AND GRID[Row][Col] AND CHECK(Row, Col, X)  INSERT X into GRID at Row, Col  Count[X] = COUNT[X] + 1  N = N - 1  Solved = SOLVE(GRID)#Valid puzzle has been made  RETURN GRID |
| --- |

1. Randomly remove a number from the grid (45-55) times more or less, removing more numbers makes for a higher difficulty
2. After each removal check that there is only 1 solution to the grid
3. The grid will be satisfactory after it has about 27 - 36 numbers left in the grid, or even lower, with still only one possible solution
4. If more than one solution is found after a removal undo the removal and move on

#### 

#### Solution Counter

The function to count solutions is essentially the same as the solve function however it does not stop once it has found a solution but goes through all of them. However, I have configured the function to stop when it has found more than one solution. This increases the performance overall as the function won't waste time looking for other solutions if the uniqueness of the grid has been proven false.

| COUNT\_SOLUTIONS(GRID, Solutions):  Pos = find empty space on the grid  IF no empty space is found  Solutions += 1  RETURN True # Indicating that the puzzle is solved  FOR n ← 1 TO 9  IF n can be placed into empty space GRID[Pos]  Place n into that empty space  COUNT\_SOLUTIONS(GRID, Solutions)  IF Solutions > 1:  RETURN  GRID[Pos] ← 0 |
| --- |

#### 

#### Digit Remover

This algorithm will remove a given number of digits from the puzzle whilst checking each time that after the removal the puzzle is unique, meaning it has one solution. It will then return the number of digits it did not remove, the clues.

| REMOVE\_DIGITS(GRID):  K = Randint(36, 64)  X = 0    FOR Row, Col through a Randomised list of all cell locations:  N = GRID[Row][Col]  Insert 0 into GRID at Row, Col  Count Solutions of the GRID  IF more than one solution is found  Put N back to where it was in the GRID    ELSE  X = X + 1    IF X == K:  BREAK  RETURN 81 - X #This returns the number of clues left in the grid |
| --- |

#### 

### 2.2.3 Puzzle Rotator and Flipper

I have made this image to represent the possible flips and rotations we can make from a single grid whilst also avoiding duplicates.

|  |
| --- |

So for every puzzle generated via the algorithm, 7 can be made from rotating and flipping the grid. Using the image I know that I just have to use the original grid and a vertical flip of the grid and make rotations of each, instead of including the horizontal flip as well.

To do the vertical flip is easy, as the grid will be stored as a 2d array so all I have to do is reverse the order of the sublists(rows) in the list(grid).

The vertical flip can be made by popping all the rows out of the grid into a stack in order, and then popping each row off the top of the stack to append it to the grid.

To make the rotator, start with an empty 2d array like [ [ ]\*9 ]

To rotate a grid 90 degrees go through the grid by each column and in each column go up looking at each item and appending the item to the row of the grid. So the last item of the 1st col becomes the 1st item of the first row of the new rotated grid.

Instead of coding separate functions for 180 and 270, we can just apply just the 90-degree rotations two times for 180 and three times for 270.

### 2.2.4 Classification and Storage of Generated Puzzles

The starting numbers given in a Sudoku puzzle are known as clues. Below are the puzzle difficulties and the range of typical clues they each contain.

Easy: 36-46 clues

Normal: 32-35 clues

Hard: 28-31 clues

Extra Hard: 17-27 clues

Each difficulty will have its own puzzle file.

Each line will contain a different puzzle and each puzzle will be stored as a string of 81 characters. Generated puzzles are sorted into their respective difficulty groups. For the puzzles in each group, a check is made to make sure that there are no duplicate puzzles being put in. All valid puzzles are then appended to their respective puzzle files.

## 2.3 Interface Design

***This will fulfil Objective 1, 2***

### 2.3.1 Main Menu

The main menu will provide user access to all the modes, the green light at the bottom right indicates that the user is logged in, if not it will be red.

### 2.3.2 Classic Sudoku

#### Difficulty Select



#### Game Screen



Pause button at top right, timer on top left.

#### Pause Screen



### 2.3.3 Multiplayer Sudoku

#### Multiplayer Match Select + Difficulty select



#### 

#### Multiplayer Game Screen



#### Multiplayer Pause Screen



### 2.3.4 Account Manager

|  |
| --- |

#### 

### 

### 2.3.5 Best Times

#### Select Difficulty Screen



#### Best Times Screen

#### 

### 2.3.6 Class Diagram for Screens

***This will fulfil Objectives 1, 2***

The Classes with names in red are ones provided from the kivy module to create the UI. The classes in the dotted line boxes are the screens that the user will see.

### 

## 2.4 Classic Sudoku

***This will fulfil Objectives 1, 2, 4, 5***

### 2.4.1 Workings of the Classic gamemode

* Select difficulty
* From the puzzle files available, randomly select a puzzle file of that difficulty.
* From the selected puzzle file, randomly select a puzzle from the file
* Show puzzle on the screen
* Start the game, start the timer
* Use the mouse to select a cell in the puzzle
* Enter the number by clicking on the cell and then clicking on the available number grid placed on the screen
* Play the game till finished, and stop the timer
* Calculate elapsed time to 2 decimal places
* If elapsed time is lower than previous times, and the new best time is recorded

## 

## 2.5 Client Server Model

***This will fulfil Objectives 6, 7, 8, 11***

### 2.5.1 Workings of the model

The client will connect to the server, and make certain requests such as login and the server will send messages corresponding to the request the client sent.

The server continuously accepts requests from multiple clients. For each client, a thread is made to handle them.

For the multiplayer system, clients will be pushed to a queue till another client is also in the queue available for them to play with, and then be both popped and put into another game thread that lets them interact with each other.

In the game thread, the server will send each client the puzzle as a string, and then receive their progress continuously, to process how complete each is, and send it to the opposite player.

### 2.5.2 Client class for client-side connection module



### 

### 2.5.3 Communication between client and server

| Client | Server |
| --- | --- |
| Will send a Register request | Will accept details to register and check they are valid to insert them into the Accounts table and make a record in the Best Times table. |
| Will send a Login request | Will accept details and check if username and hashed password.  Once verified will send the client a message that the login was verified.  Appends users to a list of online users. |
| Will send an Update Best Times request | Accepts requests and checks if new best times data for users is different from best times data for users in the Best Times table, if it is then the best times record for the user is updated. |
| Sends random match request | Adds a player to match queue, if the queue size is greater than 1 then 2 players are popped off the queue and matched together. |
| Sends specific match request, input username of player to look for | Checks if the username given is in the Accounts table and that user is in the list of online players then will send a prompt to that player to request a match. The difficulty of the match is chosen with a request sent. |
| Sends player lookup request, input username of player to look for | Check if the username given is in the Accounts table, then return the best time for that user if so. |
| Game between players | After two players are popped off a queue and matched together  Create a separate thread that will handle the communication and game handling between these two players |

### 2.5.4 Flowchart for Client Side



The flowchart above shows how the client sends requests to the server to perform the required actions.

To register the user inputs the username once and their password twice and sends them to the server and the server verifies them.

To login the user inputs the username and password and sends them to the server and the server verifies them.

To match players the client sends the wanted difficulty to the server, and the server puts them into that. If the server is empty it notifies the client, if the queue is full it rejects the match request otherwise it pushes the player to the queue.

The play multiplayer part is only initiated when they are popped off the match queue.

### 

### 2.5.5 Flowchart for Server Side

## 

For each request received from a client a thread is made to handle them.

The register function accepts details from the client and verifies them ensuring they aren’t present in the database.

The login function accepts details from the server and makes sure the username and password match to the one in the database.

The match player function checks that the queue is not full, and then inserts the player into the queue if it is not full. If there are two players in the queue then they can be dequeued into a match.

As the game is set up between two players the server sends each of them the puzzle to play and then continuously receives the players game states, processes them and sends the processed states back to the opposite players to let each player know the progress of their opponent. When one player sends a Win message to the server the server tells the other player they lost and ends the game.

## 2.6 Account & Database System

***This will fulfil Objectives 8, 9***

* Users are given the option to register by creating a username and password.
* The username and password will be stored on the server accounts database, the password will be hashed.
* Locally the username and hashed password will also be stored to keep the player logged in if they choose to do so.

### 2.6.1 Entity Relationship Diagram



### 

### 2.6.2 Register

Player’s accounts will consist of a username and a hashed password.

When players register they will enter their username once and their password twice, their password must be a mix of upper and lower case and include a number.

The program when registering will check if the username is already in the database if it is, the program prompts the user to choose a different username, otherwise, the username and hashed password are put into the Accounts table, and the username will be the primary key of this table.

The user's Username will also be added to the Best Times table, and the top scores of the user will be added to the Best Times table.

### 2.6.3 Login

When a user enters their details, the program will check if the username is in the database and if it is, it will check if the hashed password matches the hashed password in the database. Once the details have been verified the server will send a message to the client that the details have been verified, and set a logged-in status in the client.

### 2.6.4 Account Database

Stored online with server

Fields → (username, password)

### 2.6.5 Best Times Database

Stored online with server

Fields → (username, easy, normal, hard, extra\_hard)

For every user, this data will also be stored on their local system and updates the database each time there is a change in the local one and the user is logged in.

## 2.7 Information to save locally in the game's files

***This will fulfil Objective 12***

The username and hashed password will be stored. Best Times for each difficulty.

Example: in a .txt file

| username  f486a1cf7b1f72  10  20  30  40 |
| --- |

# 3 TECHNICAL SOLUTION

## 3.0 Details and Techniques Used

#### Naming Convention

* Variables: **c**amel**C**ase
* Functions: **s**nake**\_C**amel**\_C**ase**\_H**ybrid, note some API methods may not be able to conform
* Classes: **P**ascal**C**ase

#### Folder Structure

| └── 📁Sudoku  └── 📁Main  └── main.py  └── Sudoku.kv  └── networking.py  └── Game Data.txt  └── 📁graphics  └── Sudoku\_App\_Background.png  └── Sudoku\_App\_Border\_Logged\_In.png  └── Sudoku\_App\_Border\_Logged\_Out.png  └──Sudoku\_App\_Deafult\_Button.png  └── Sudoku\_App\_Help.png  └── 📁Generator  └── Generate.py  └── easy.txt  └── normal.txt  └── hard.txt  └── extra\_hard.txt  └── 📁Sever  └── Server.py  └── Sudoku\_Online.db  └── 📁Generator  └── Generate.py  └── easy.txt  └── normal.txt  └── hard.txt  └── extra\_hard.txt |
| --- |

### 3.0.1 List of Techniques Used

From 4.14.3.4.1 Table 1 Example Technical skills

| Group | Technique/Algorithm | Link |
| --- | --- | --- |
| A | Recursive algorithm  Recursive algorithm + Complex user-defined algorithm  Recursive algorithm  Complex user-defined algorithm  Stack  Queue  Complex client-server model + Server-side scripting using request and response objects and server-side extensions for a complex client-server model  Hashing  Dynamic generation of objects based on complex user-defined use of the OOP model  Complex user-defined use of object-oriented programming Inheritance + Polymorphism | [DFS](#_bflaae70t8s0)  [Eliminate](#_dlns36dj4pqs)  [Count Solutions](#_gwwfuj4lemuz)  [Remove Digits](#_2q18jkwh0f9i)  [Stack](#_tk5ier10e449)  [Circular Queue](#_uhlm6nn1y060)  [Client Side](#_z6jbvuxdprzk), [Server Side Basic](#_qs0dakwld70k), [Server Side Multiplayer](#_2dxs39v94al6)  [gaviHash](#_9sptp5lty03o)  [Creating the puzzle grid](#_e3fvknsbmgwp)  [GameScreen Load](#_e3fvknsbmgwp), [GameScreen Cell](#_kl36zhoc8xb7) |
|
| B | Multi-dimensional arrays  Simple user-defined algorithm  Simple OOP model  Simple OOP model  Writing and reading from files  Simple database model  Single table SQL | [Puzzle Grid](#_ukbp4unq44v7)  [Puzzle Candidates](#_4ryddx39sgvl)  [Make More function](#_lzun0b8mpd7y)  [Puzzle Class](#_y41yvo1yox6m)  [PuzzleFile class](#_hsjaisyjmecb)  [Game Class](#_tekyl3ohqbt0)  [App Class](#_42k54p7zptcc)  [PuzzleFile methods](#_hsjaisyjmecb)  [Game data](#_y924od5un185)  [Database](#_hhnmjyrupndv)  [Verify function](#_puszikcz1fma)  [Check function](#_210ixgw23pn6)  [Register function](#_rg0plp6vqhlg)  [Update best times function](#_rnu04bei6ie) |

## 

### 

### 3.0.2 List of Objectives met

| No. | Objective | References |
| --- | --- | --- |
| 1 | Create an app where users can solve sudoku puzzles | [3.2](#_qrbl47d4xuf6), [3.3](#_2ftpcmxxj02j), [3.4](#_vwoienczyox0), [3.5](#_aaf9vwwgxo4i) |
| 2 | The app must be responsive and easy to navigate | [3.3](#_2ftpcmxxj02j) |
| 3 | Generate and store unique sudoku puzzles for the app. | [3.1](#_4ynkz0jm7bxi) |
| 4 | Implement Classic mode | [3.3](#_2ftpcmxxj02j) |
| 5 | Create a responsive GUI for the game mode | [3.3](#_2ftpcmxxj02j) |
| 6 | Implement Multiplayer mode | [3.3](#_2ftpcmxxj02j), [3.7](#_2dxs39v94al6), [3.8](#_z6jbvuxdprzk) |
| 7 | Create a server to connect to clients, manage accounts and facilitate the multiplayer mode | [3.6](#_qs0dakwld70k), [3.7](#_2dxs39v94al6), [3.8](#_z6jbvuxdprzk) |
| 8 | Create an Account system | [3.6](#_qs0dakwld70k), [3.8](#_z6jbvuxdprzk) |
| 9 | Create a database to store account information for the account system | [3.6.6](#_hhnmjyrupndv) |
| 10 | Create a matching system for the multiplayer mode | [3.6.8](#_nzal5w3mi4ve), [3.7.1](#_rw0h4o6sqoc), [3.8.1](#_ged8jkuzq6vn) |
| 11 | Create the multiplayer game system | [3.3.6](#_14ossvbnejww), [3.7](#_2dxs39v94al6) |
| 12 | Store the user’s username, password and best times in a local file. | [3.2.2](#_42k54p7zptcc) |
| 13 | Create a tutorial for the users | [3.5.2](#_d4woscrr507y) |

# 

## 

## 3.1 Puzzle Generator

[Generate.py](#_s1aqnzdbwr6j), ***This fulfils Objective 3***

### 3.1.1 Puzzle Class

***4.14.3.4.1 Table 1 Group B, Simple OOP Class***

#### Constructor

| class Puzzle:  def \_\_init\_\_(self, data = None):  if type(data) == str:#For importing grids  self.string\_To\_Grid(data) # Converts a string representation of a puzzle into a 2d array  self.get\_All\_Candidates() # Sets candidates for all cells  elif type(data) == list:  # For importing grid as 2d array, used for converting grid into string  self.grid = data  else: # Default  self.generate() |
| --- |

#### Show grid

| def show\_grid(self):  for row in self.grid:  for item in row:  print(item, end = " ")  print()#newline  print() |
| --- |

#### Insert

| def insert(self, row, col, n):  self.grid[row][col] = n |
| --- |

### 

### 

### 3.1.2 Components of Solve function

#### Find empty space

| def find\_Empty\_Space(self):  for row in range(9):  for col in range(9):  if self.grid[row][col] == 0:  return (row, col)  return None |
| --- |

#### Check

| def check(self, row, col, num):#return false if there are any mistakes  if num in self.grid[row]:  return False    for i in range(9):  if num == self.grid[i][col]:  return False    boxRow = row // 3 \* 3 #Reduces numbers to either 0, 3 or 6 the  boxCol = col // 3 \* 3 #starting indexes for a cells respective box  for i in range(boxRow, boxRow+3):  for j in range(boxCol, boxCol+3):  if num == self.grid[i][j]:  return False  return True |
| --- |

#### 

The algorithm checks to see if num can be placed in the cell location (row, col).

#### Get all candidates

| def get\_All\_Candidates(self):  self.candidates = [[set() for i in range(9)]for j in range(9)]  #All cells start with empty candidates set  for row in range(9):  for col in range(9):  if self.grid[row][col] == 0:  #If cell is empty then it will get all possible candidates  candidates = set(range(1,10))    candidates -= set(self.grid[row])    candidates -= set(self.grid[i][col] for i in range(9))  boxRow = row // 3 \* 3  boxCol = col // 3 \* 3    candidates -= set(self.grid[i][j]  for j in range(boxCol, boxCol+3)  for i in range(boxRow, boxRow+3))  self.candidates[row][col] = candidates |
| --- |

self.candidates is a 2d array of sets. The numbers in the sets represent the possible number that can be placed into that respective cell. Filled cells are left with empty candidate sets.

#### Sort Candidates

| def sort\_Candidates(self):  self.sortedCandidates = sorted( [(self.candidates[row][col], row, col)  for col in range(9)  for row in range(9)  if len(self.candidates[row][col]) > 0],  key = lambda x: len(x[0]) ) |
| --- |

***4.14.3.4.1 Table 1 Group B, Simple user-defined algorithm***

Uses list comprehension to create a list of tuples that contain (candidates, row, col) of all the cells with candidates and sorts them in ascending order.

#### Update Peers Remove Candidates

| def update\_Peers\_Remove\_Candidates(self, row, col, n):#When a number is inserted, the cells in the same row, column and box will have that number removed from their candidates  self.candidates[row][col] = set()    for i in range(9):  self.candidates[row][i].discard(n)  self.candidates[i][col].discard(n)  boxRow = row // 3 \* 3  boxCol = col // 3 \* 3    for i in range(boxRow, boxRow+3):  for j in range(boxCol, boxCol+3):  self.candidates[i][j].discard(n) |
| --- |

***4.14.3.4.1 Table 1 Group B, Simple user-defined algorithm***

#### 

#### Update Peers Insert Candidates

| def update\_Peers\_Insert\_Candidates(self, row, col, n):  #For adding candidates back to a cell that was previously filled  candidates = set(range(1,10))  candidates -= set(self.grid[row])  candidates -= set(self.grid[i][col] for i in range(9))  boxRow = row // 3 \* 3  boxCol = col // 3 \* 3    candidates -= set(self.grid[i][j]  for j in range(boxCol, boxCol+3)  for i in range(boxRow, boxRow+3))  self.candidates[row][col] = candidates  #For the peers of the cell adds the number back to their candidates  for i in range(9):  if self.grid[row][i] == 0 and self.check(row, i, n):  self.candidates[row][i].add(n)  if self.grid[i][col] == 0 and self.check(i, col, n):  self.candidates[i][col].add(n)    for i in range(boxRow, boxRow+3):  for j in range(boxCol, boxCol+3):  if self.grid[i][j] == 0 and self.check(i, j, n):  self.candidates[i][j].add(n) |
| --- |

### 

***4.14.3.4.1 Table 1 Group B, Simple user-defined algorithm***

#### 

#### Eliminate

| def eliminate(self):  if len(self.sortedCandidates) < 1:  if self.find\_Empty\_Space() is None:  return True  else:  return False    emptySpace = self.sortedCandidates.pop(0)  candidates, row, col = emptySpace    if len(candidates) == 1:  self.insert(row, col, self.candidates[row][col].pop())  self.update\_Peers\_Remove\_Candidates(row, col, self.grid[row][col])  self.sort\_Candidates()  return self.eliminate()  else:  self.sortedCandidates.insert(0, emptySpace)  return False |
| --- |

***4.14.3.4.1 Table 1 Group A, Complex user-defined + Recursive algorithm***

This algorithm solves the grid via a method known as constraint propagation, the constraints being the candidates for each cell, which places numbers into cells that only have one possible candidate, it then updates the candidate sets of the cells that are part of the same subgrid, row and column of the changed cell to “propagate the constraints”. It then recursively calls itself to perform the whole action again. If there are no more cells with only one candidate, then the algorithm will return False, to be directed to be solved via the dfs function. If the sortedCandidates list is reduced to None this is usually an indication that the grid is solved, but we perform and find\_Empty\_Space check to make sure it is, if it is not it is a sign that the grid is invalid/unsolvable, but it is still sent to the dfs function to double check that it is so.

#### Constraint Propagation

| def constraint\_Propagation(self):  self.get\_All\_Candidates()  self.sort\_Candidates()  return self.eliminate() |
| --- |

This function sets up the self.candidates array and the self.sortedCandidates array for the execution of the self.eliminate function.

#### DFS

| def dfs(self) -> bool:#Named after Depth First Search, backtracking algorithm  pos = self.find\_Empty\_Space()#pos is given as a tuple (row, col)  if pos == None:  #Meaning the grid is full and solved  return True    row, col = pos  for n in self.candidates[row][col]:  #if self.check(row, col, n):  self.insert(row, col, n)  self.update\_Peers\_Remove\_Candidates(row, col, n)    if self.dfs():  #Causes all the recursion to unwind  return True  self.insert(row, col, 0)  self.update\_Peers\_Insert\_Candidates(row, col, n)  return False |
| --- |

***4.14.3.4.1 Table 1 Group A, Complex user-defined + Recursive algorithm***

This algorithm also solves the grid via backtracking with constraint propagation involved in searching for solutions by going through the grid and trying all possible number for each cell.

The algorithm goes through the grid finding empty Spaces to place numbers in, if it finds none this means that the grid is filled and solved. After obtaining a space it checks the the available candidates for that cell, inserts the first number in the cell candidates and then updates the candidates for the cell and the cells that are in the same subgrid, row and column.

After that, it recursively calls itself to go through the grid again.

If at any point the code reaches the return False statement this will be because the algorithm has reached a dead end and cannot proceed to solve the grid from that point and so returning False will cause the grid to undo the placement of the number and try inputting a different number into the grid.

#### Solve

| def solve(self) -> bool:  if self.constraint\_Propagation():  print("[SOLVED VIA CONSTRAINT PROPAGATION]")  return True  else:  if self.dfs():  print("[SOLVED VIA BACKTRACKING]")  return True  else:  print("[UNSOLVABLE]")  return False |
| --- |

This function starts by attempting to solve the grid via constraint propagation, if it cannot, it then tries to do so via the backtracking approach using the dfs function, if neither of these methods works, it means that the grid is invalid/unsolvable.

### 3.1.3 Puzzle X String Converters

| def grid\_To\_String(self):  #Will convert the grid into a string that can be saved on a file  return "".join( ["".join([str(item) for item in row]) for row in self.grid] ) |
| --- |

Used during the generation process in sorting and storing to convert the grid into strings that are saved as text.

| def string\_To\_Grid(self, string):  #the if statement accounts for grids that use a dot instead of 0  self.grid = [[int(item) if item != "." else 0  for item in string[9\*row:9\*(row+1)]]  for row in range(9)] |
| --- |

Used to convert puzzleStrings obtained from the puzzle files into the grid that will be used to interact with the gameplay.

### 

### 3.1.4 Generator

#### Fill Grid

| def fill\_Grid(self):  print("[Filling Grid]")  solved = False  while solved == False:    self.grid = [[0 for col in range(9)] for row in range(9)]  count = {}  for i in range(1, 10):  #Will be used to keep track of how many of each number is inserted  count[i] = 0  numberOfCellsToInsert = random.randint(25, 35)  while numberOfCellsToInsert > 0 :  x = random.randint(1, 9)  row = random.randint(0, 8)  col = random.randint(0, 8)  if self.grid[row][col] == 0:  if count[x] < 9 and self.check(row, col, x):  #Ensures that no more than 9 copies of the same number  self.insert(row, col, x)  count[x] += 1  numberOfCellsToInsert -= 1  solved = self.solve() |
| --- |

***4.14.3.4.1 Table 1 Group A, Complex user-defined***

The algorithm fills a random number of cells, ensuring that all the placements are valid and that no more than 9 iterations of the same digit are present in the grid, it then fills the rest of the grid by applying the solver to it. This process is repeated until a valid solution is found.

#### Count solutions

| def count\_Solutions(self):  #print("[COUNTING SOLUTIONS]")  pos = self.find\_Empty\_Space()#pos is given as a tuple (row, col)  if pos == None:#Meaning the grid is full and solved  self.solutions += 1 #Notes that it has found a solution  return #Continues "EXPLORING" grid for more solutions    row, col = pos  for n in range(1, 10):  if self.check(row, col, n):  self.insert(row, col, n)  self.count\_Solutions()  if self.solutions > 1:  #More than one solution is found so we stop searching  return    self.insert(row, col, 0) |
| --- |

***4.14.3.4.1 Table 1 Group A, Recursive function***

The count solutions algorithm is essentially the same as the dfs function, except once it has found a solution it will backtrack till it finds another one, I have modified it to stop after finding more than one solution. It also doesn't.

The algorithm doesn’t use the candidate reduction methods as it can interfere with the search for all possible solutions and result in invalid grids being generated.

#### 

#### 

#### 

#### Remove digits

| def remove\_digits(self) -> int:  print("[Removing Digits]")  self.solutions = 0  digitsToRemove = int(random.triangular(36, 64, 50))# Weighted towards 50  digitsRemoved = 0 #Used to count how many digits have been removed  cells = [(row, col) for col in range(9) for row in range(9)]\*3  #All locations are set three times so that algorithm does not end too soon  random.shuffle(cells)  for row, col in cells:#While all cells haven't been visited thrice    if self.grid[row][col] == 0:  continue    n = self.grid[row][col]  self.insert(row, col, 0)  #count\_Solutions alters the grid so a copy is made  change = copy.deepcopy(self.grid)  self.count\_Solutions()  if self.solutions > 1:  #If more than one solutions are found, undo removal of digit  self.grid = copy.deepcopy(change)  self.insert(row, col, n)  elif self.solutions == 1:  self.grid = copy.deepcopy(change)  digitsRemoved += 1    self.solutions = 0    if digitsRemoved == digitsToRemove:  #if all required number of cells to be removed have been removed  break  self.grid = copy.deepcopy(change)  return 81 - digitsRemoved # = number of clues |
| --- |

***4.14.3.4.1 Table 1 Group A, Complex user-defined algorithm***

The algorithm removes cells from the grid, each time counting the number of solutions after each change to ensure the grid retains a unique solution.

copy.deepcopy() is used to make copies of the grid as the way Python make grid copies is that it makes variables point to the same reference or memory location which means the copies aren’t independent of each other, so a change in one will affect the other, so I used copy.deepcopy to ensure I could make changes to the copies without letting the changes affect the preserved grids.. The count Solutions algorithm changes self.grid, so that is why the copies needed to be made.

#### 

#### Generate

| def generate(self):  self.fill\_Grid()  self.clues = self.remove\_digits()  #print(f"Clues: {self.clues}") |
| --- |

This function creates fully valid solutions to a Sudoku puzzle and then creates a valid playable puzzle by removing digits from it randomly. self.clues is the number given cells in the grid.

### 

### 

### 3.1.5 Puzzle Rotator and Flipper

#### Vertical Flipper

| def flip\_Vertical(grid : list) -> list:  newGrid = Stack(9)  for row in grid:  newGrid.push\_To\_Stack(row)  return [newGrid.pop\_From\_Stack() for i in range(9)] |
| --- |

***4.14.3.4.1 Table 1 Group A, Stack Operations***, This function flips the vertical orientation of the grid that is input.

#### 90-Degree Rotator

| def rotate\_90(grid : list) -> list:  newGrid = [[], [], [], [], [], [], [], [], []]  for col in range(9):#for each column  for row in range(8, -1, -1):  newGrid[col].append(grid[row][col])  return newGrid |
| --- |

This function rotates the grid by 90 degrees by going through each column, through each item in each column from down to up and appends the items from the first column to the first row of the new grid. So then the items of the second column will go to the second row and so on.

#### Make More

| def make\_More(grid : list) -> list:  more = [].append(grid)  flipped = flip\_Vertical(grid)  more.append(flipped)  for i in range(3): # 90, 180, 270  grid = rotate\_90(grid)  more.append(grid)  flipped = rotate\_90(flipped)  more.append(flipped)  return more |
| --- |

***4.14.3.4.1 Table 1 Group B, Simple user-defined algorithm***

As seen in [2.2.3](#_wyk84cdfsat0) we only need to create a vertically flipped copy of the grid and rotate those as well as the original grid to make all the grid copies required.

### 3.1.6 Puzzle Generation and Storage

#### PuzzleFile Class

| class PuzzleFile:  def \_\_init\_\_(self, file : str, mode : str, data : list = []):  self.duplicates = 0  if mode == "read":  self.file = open(file, "r")  self.contents = self.file.readlines()  self.file.close()  elif mode == "append":  self.file = open(file, "a+")  self.contents = self.file.readlines()  for puzzleString in data:  if not f"{puzzleString}\n" in self.contents:  self.file.write(f"{puzzleString}\n")  else:  self.duplicates += 1  self.file.close() |
| --- |

***Group B, Simple OOP model + Writing and reading from files***

#### Puzzle Production: Generation

| if \_\_name\_\_ == "\_\_main\_\_":  numberToGenerate = 50 #n\*8 puzzles will be generated  easy = [], normal = [], hard = [], extra\_hard = []  outliers = []  listOfPuzzles = []  startTime = time.perf\_counter()  for i in range(numberToGenerate):  print(f"[Puzzle {i+1}]")  listOfPuzzles.append(Puzzle())  elapsedTime = time.perf\_counter() - startTime  print(f"{round(elapsedTime, 2)} seconds to generate {numberToGenerate} puzzles") |
| --- |

#### Puzzle Production: Sorting and Storing

| for puzzle in listOfPuzzles:  if puzzle.clues in range(17, 28):  a = make\_More(puzzle.grid)  for i in a:  extra\_hard.append(Puzzle(i).grid\_To\_String())  elif puzzle.clues in range(28, 32):  a = make\_More(puzzle.grid)  for i in a:  hard.append(Puzzle(i).grid\_To\_String())  elif puzzle.clues in range(32, 36):  a = make\_More(puzzle.grid)  for i in a:  normal.append(Puzzle(i).grid\_To\_String())  elif puzzle.clues in range(36, 45):  a = make\_More(puzzle.grid)  for i in a:  easy.append(Puzzle(i).grid\_To\_String())  else:  outliers.append(puzzle.grid\_To\_String())  easyFile = PuzzleFile("Main/Generator/easy.txt", "append", easy)  normalFile = PuzzleFile("Main/Generator/normal.txt", "append", normal)  hardFile = PuzzleFile("Main/Generator/hard.txt", "append", hard)  extra\_HardFile = PuzzleFile("Main/Generator/extra\_hard.txt", "append", extra\_hard)  print("------------------------------")  print(f"easy {len(easy)}")  print(f"normal {len(normal)}")  print(f"hard {len(hard)}")  print(f"extra hard {len(extra\_hard)}")  print(f"Total {numberToGenerate\*8}")  print("------------------------------")  print(f"outliers {len(outliers)}")  print(f"Duplicates {easyFile.duplicates + normalFile.duplicates + hardFile.duplicates + extra\_HardFile.duplicates}")  print(outliers) |
| --- |

## 3.2 Main Game Overview

[main.py + Sudoku.kv](https://docs.google.com/document/d/1CREVJVae8fQst4eSGylUl6tpC7_fLLkmfZf2O7nfvS4/edit?pli=1#heading=h.s1aqnzdbwr6j), ***This fulfils Objectives 1, 2, 4, 5, 6, 8,***

### 3.2.1 Game Class Initialisation + Methods

***This fulfils Objectives 4, 5***

|  |
| --- |

### 

### 3.2.2 App Class

***This fulfils Objective 12***

#### Class definitions and boot method

|  |
| --- |

#### Game data file interaction methods

|  |
| --- |

#### Check match found

|  |
| --- |

***This fulfils Objective 10***

This function will be run in the background once the client has received information that it has been queued into a match.

#### App exit handler method

|  |
| --- |

### 3.2.3 Screens

#### Screens in kv file

***This fulfils Objective 2***

The code on the left sets all the screens to be used as part of the screen manager, the screen manager is used to switch between screens.

| class MenuManager(ScreenManager):  pass |
| --- |

#### 

The size\_hint of a widget is its size relative to the widget contained inside.

#### BaseScreen

|  |
| --- |

StringProperty is a string object that when updated is automatically reflected for all widgets that utilise that string, the two different border styles feature and the little red/green circle used to denote the user’s off/online status.

#### BaseScreen in kv

| <BaseScreen>:  name : "BaseScreen"  canvas.before:  Color :  rgba : 1, 1, 1, 0.07  Rectangle:  size: self.size  source : None      canvas.after:  Rectangle:  size: self.size  source : root.borderFile |
| --- |

### 3.2.4 Menus



| <MainMenu>:  name : "MainMenu"    AnchorLayout:  size\_hint : 0.9, 0.4  pos\_hint : {"center\_x": 0.5, "center\_y":0.65}  BoxLayout:  orientation : "vertical"  BoxLayout:  size\_hint : 1, 0.75  Label:  text: "Sudoku App"  font\_size: 50  color: 0, 1, 0.6, 1    BoxLayout:  size\_hint : 1, 0.25  Button:  text: "Best Times"  font\_size : 30  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  on\_release:  root.manager.transition = SlideTransition()  root.manager.transition.direction = "right"  root.manager.current = "BestTimes"  Button:  text: "Classic"  font\_size : 30  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  on\_release:  root.manager.transition = SlideTransition()  root.manager.transition.direction = "right"  root.manager.current = "ClassicMenu"  Button:  text: "Multiplayer"  font\_size : 30  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  on\_release:  root.manager.transition = SlideTransition()  root.manager.transition.direction = "left"  root.manager.current = "MultiplayerMenu"  Button:  text: "Account"  font\_size : 30  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  on\_release:  root.manager.transition = SlideTransition()  root.manager.transition.direction = "left"  root.manager.current = "AccountMenu"  BoxLayout:  size\_hint : 0.225, 0.1  pos\_hint : {"center\_x": 0.5, "center\_y":0.25}  Button:  text: "Help"  font\_size : 30  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  on\_release:  root.manager.transition = SlideTransition()  root.manager.transition.direction = "up"  root.manager.current = "HowToPlayScreen" |
| --- |



### 3.2.5 Classic Menu

****

****

### 3.2.6 Multiplayer Menu



Match function used to initiate contact with server to request to be matched into a queue, if the match\_Players returns True it means that the player has been matched and so the client can start waiting to be put into a game.

****

### 3.2.7 Account Menu

| class AccountMenu(Menu):  def on\_enter(self):  super().on\_enter()  self.ids.usernameLabel.text = self.get\_Username()    def get\_Username(self):  return app.username  def clickLogout(self):  if app.client and app.online is True:  app.client.disconnect()  app.client = None  app.online = False  self.set\_Border()  app.rememberLogin = False |
| --- |





## 

## 3.3 GameScreen

[main.py + Sudoku.kv](#_s1aqnzdbwr6j), ***This fulfils Objectives 2, 4, 5, 6***

#### Colours variables set

There is no need to have a separate colour for clue cells, as they are set as disabled and the buttons object automatically dull them.

### 3.3.1 Cell

| class Cell(Button):  def \_\_init\_\_(self, row, col, n, \*\*kwargs):  super().\_\_init\_\_(\*\*kwargs)  self.color = TEXT  self.font\_size = 20  self.width = 10  self.height = 10  self.row = row  self.col = col  self.n = n  self.background\_color = NEUTRAL  if self.n > 0:  self.text = str(n)  self.disabled = True  else:  self.text = ""  self.disabled = False    self.clock = Clock.schedule\_interval(self.checkCell, 0.5)  def updateCell(self, n):  self.n = n  self.text = str(n)  game.puzzle.insert(self.row, self.col, self.n)  def checkCell(self, dt):  if self.n != 0:  game.puzzle.insert(self.row, self.col, 0)#THis is so that it does not detect a collison with itself  if game.puzzle.check(self.row, self.col, self.n) is False:  self.background\_color = COLLISION  else:  self.background\_color = NEUTRAL  game.puzzle.insert(self.row, self.col, self.n)    def clearCell(self):  self.n = 0  self.text = ""  game.puzzle.insert(self.row, self.col, 0)  self.background\_color = NEUTRAL    def on\_press(self):  if game.holding\_Number > 0:  self.updateCell(game.holding\_Number)  game.holding\_Number = 0  elif self.last\_touch.button == "right":  self.clearCell() |
| --- |

***4.14.3.4.1 Table 1 Group A, Complex user-defined use of object-oriented programming (OOP) model, Inheritance & Polymorphism***

***Inheritance is demonstrated with inheriting the methods and attributes from Button and the polymorphism with the method overriding of the on\_press method.***

### 

### 3.3.2 NumberInput

| class NumberInput(Button):  def \_\_init\_\_(self, n, \*\*kwargs):  super().\_\_init\_\_(\*\*kwargs)  self.width = 5  self.height = 5  self.n = n  self.text = str(n)    def on\_press(self):  game.holding\_Number = self.n |
| --- |

### 3.3.3 Timer

| class Timer(Label):  def \_\_init\_\_(self, \*\*kwargs):  super().\_\_init\_\_(\*\*kwargs) |
| --- |

### 3.3.4 GameScreen

#### Definition and Startup

| class GameScreen(BaseScreen):  def \_\_init\_\_(self, \*\*kwargs):  super().\_\_init\_\_(\*\*kwargs)  def on\_enter(self):  self.set\_Border()  self.load()  self.clock = Clock.schedule\_interval(self.checks, 0.01) |
| --- |

***4.14.3.4.1 Table 1 Group A, Complex user-defined use of object-oriented programming (OOP) model, Inheritance, Polymorphism***

***Inheritance is demonstrated with inheriting the methods and attributes from BaseScreen and the polymorphism with the method overriding of the on\_enter and on\_leave methods***

***The same inheritance and polymorphism is present in all of the other child screens created from the BaseScreen.***

#### Load

| def load(self):  game.import\_Puzzle(game.difficulty)#assigns puzzle to game.puzzle  game.puzzle.show\_grid()  grid = self.ids.grid  i = 0  j = 0  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box1.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 0  j = 3  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box2.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 0  j = 6  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box3.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 1  j = 0  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box4.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 1  j = 3  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box5.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 1  j = 6  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box6.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 2  j = 0  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box7.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 2  j = 3  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box8.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 2  j = 6  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box9.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  """  for row in range(9):  for col in range(9):  grid.add\_widget(Cell(row, col, game.puzzle.grid[row][col]))  """  numGrid = self.ids.numberGrid  for n in range(1, 10):  numGrid.add\_widget(NumberInput(n))  self.recentTime = time.time()  self.elapsedTime = 0  game.timerOn = True |
| --- |

***4.14.3.4.1 Table 1 Group A, Dynamic generation of objects based on complex user-defined use of OOP model***

Creates cell widgets based on the respective numbers that are also in that place in the grid, Cells that are already filled will be disabled.

The cells have to be grouped by subgrids so that the proper formatting for the UI is implemented so that the difference between the subgrids is made obvious.

#### 

#### UpdateTimer

| def updateTimer(self):  self.elapsedTime += time.time() - self.recentTime  self.recentTime = time.time()  self.ids.timer.text = game.parse\_Timer\_to\_String(self.elapsedTime)  self.saveTime = [round(self.elapsedTime, 2), self.ids.timer.text] |
| --- |

#### Checks

| def checks(self, dt):  if game.timerOn:  self.updateTimer()  else:  self.recentTime = time.time()  game.win = game.puzzle.grid == game.puzzleSolution.grid  if game.win:#check win  print("Player WINS!")  self.clock.cancel()  game.timerOn = False  game.finishTime = self.saveTime  topTime = app.topTimes[difficulties.index(game.difficulty)]  topTime = float(topTime) if len(topTime) > 0 else 0    newRecord = False  if topTime == 0 or topTime > game.finishTime[0]:  newRecord = True  app.topTimes[difficulties.index(game.difficulty)] = str(game.finishTime[0])  game.win = False  p = Popup(title = "Congratulations",  content = Label(text = f"{'New record!\n' if newRecord else ''}Complete Time: {game.finishTime[1]}"),  size\_hint = (0.6, 0.3))  p.open()  self.manager.current = "MainMenu" |
| --- |

#### On leave

| def on\_leave(self):  self.clock.cancel()    for i in self.ids.box1.children:  i.clock.cancel()  for i in self.ids.box2.children:  i.clock.cancel()  for i in self.ids.box3.children:  i.clock.cancel()  for i in self.ids.box4.children:  i.clock.cancel()  for i in self.ids.box5.children:  i.clock.cancel()  for i in self.ids.box6.children:  i.clock.cancel()  for i in self.ids.box7.children:  i.clock.cancel()  for i in self.ids.box8.children:  i.clock.cancel()  for i in self.ids.box9.children:  i.clock.cancel()  game.puzzle, game.puzzleSolution = None, None  self.ids.box1.clear\_widgets()  self.ids.box2.clear\_widgets()  self.ids.box3.clear\_widgets()  self.ids.box4.clear\_widgets()  self.ids.box5.clear\_widgets()  self.ids.box6.clear\_widgets()  self.ids.box7.clear\_widgets()  self.ids.box8.clear\_widgets()  self.ids.box9.clear\_widgets()  self.ids.numberGrid.clear\_widgets() |
| --- |

### 

### 3.3.5 Classic Game

***This fulfils Objectives 2, 4, 5***

| class ClassicGame(GameScreen):  def pauseGame(self):  pause = PauseScreen()  pause.open() |
| --- |

Only the classic game requires a pause screen,

#### ClassicGame Screen



#### 

#### 

#### PauseScreen

| class PauseScreen(Popup):  def on\_open(self):  game.timerOn = False  def on\_dismiss(self):  game.timerOn = True |
| --- |

| <PauseScreen>:  title : "Pause"  size\_hint : 0.4, 0.6  Button:  text: "Resume"  on\_release: root.dismiss() |
| --- |

PauseScreen is a separate widget from the game screen that is activated as the pause button is pressed.



### 

### 3.3.6 Multiplayer Game

***This fulfils Objectives 2, 5, 6***

#### Definition and Startup

| class MultiplayerGame(GameScreen):  def \_\_init\_\_(self, \*\*kwargs):  super().\_\_init\_\_(\*\*kwargs)  def on\_enter(self):  self.set\_Border()  #RECEIVE GRID  #initilaize stuff  self.load()  self.clock = Clock.schedule\_interval(self.checks, 0.01) |
| --- |

#### Load

| def load(self):#########Change this for multiplayer  puzzleString = app.client.receive()  self.opponentUsername = app.client.receive()    game.puzzle = Puzzle(puzzleString)  game.puzzleSolution = Puzzle(puzzleString)  game.puzzleSolution.solve()  game.opponentGrid = "".join([str(0) for i in range(81)]) # Will be revced by server as astring of 1s and 0s, 1s for cells that are complete and 0s for cells that arent  i = 0  j = 0  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box1.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 0  j = 3  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box2.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 0  j = 6  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box3.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 1  j = 0  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box4.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 1  j = 3  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box5.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 1  j = 6  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box6.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 2  j = 0  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box7.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 2  j = 3  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box8.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  i = 2  j = 6  for x in range(3\*i, 3\*i+3):  for y in range(j, j+3):  self.ids.box9.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))  numGrid = self.ids.numberGrid  for n in range(1, 10):  numGrid.add\_widget(NumberInput(n))  opponentGrid = self.ids.opponentGrid  for i in range(81):  opponentGrid.add\_widget(OpponentGridCell(game.opponentGrid[i]))  self.recentTime = time.time()  self.elapsedTime = 0  game.timerOn = True |
| --- |

***4.14.3.4.1 Table 1 Group A, Dynamic generation of objects based on complex user-defined use of OOP model***

#### Checks

| def checks(self, dt):  #GET OPPONENT GRID SOMEHOW  app.client.send(game.puzzle.grid\_To\_String())  game.opponentGrid = app.client.receive()  if game.opponentGrid == "LOSE":  print("Opponent Wins!")  self.clock.cancel()  game.timerOn = False  p = Popup(title = "Unlucky", content = Label(text = "Opponent Wins!"), size\_hint = (0.4, 0.35))  p.open()  self.manager.current = "MainMenu"    elif game.opponentGrid == "OPPONENT QUIT":  self.clock.cancel()  game.timerOn = False  p = Popup(title = "Unlucky", content = Label(text = "Opponent Quit!"), size\_hint = (0.4, 0.35))  p.open()  self.manager.current = "MainMenu"    elif game.opponentGrid is not None:  for i in range(81):  self.ids.opponentGrid.children[80-i].updateCell(game.opponentGrid[i])  print(game.opponentGrid)  if game.timerOn:  self.updateTimer()  else:  self.recentTime = time.time()  game.win = game.puzzle.grid == game.puzzleSolution.grid  if game.win:#check win  app.client.send("WIN")  self.clock.cancel()  print("Player WINS!")  game.timerOn = False  game.finishTime = self.saveTime  game.win = False  topTime = app.topTimes[difficulties.index(game.difficulty)]  topTime = float(topTime) if len(topTime) > 0 else 0  newRecord = False  if topTime == 0 or topTime > game.finishTime[0]:  newRecord = True  app.topTimes[difficulties.index(game.difficulty)] = str(game.finishTime[0])  p = Popup(title = "Congratulations",  content = Label(text = f"{'New record!/n' if newRecord else ''}You Win/nComplete Time: {game.finishTime[1]}"),  size\_hint = (0.6, 0.3))  p.open()  self.manager.current = "MainMenu"  print("Exit") |
| --- |

#### Click Exit

| def clickExit(self):  self.clock.cancel()  game.timerOn = False  app.client.send("QUIT")  self.manager.transition.direction = "down"  self.manager.current = "MultiplayerMenu" |
| --- |

#### On leave

| def on\_leave(self):  self.clock.cancel()  for i in self.ids.box1.children:  i.clock.cancel()  for i in self.ids.box2.children:  i.clock.cancel()  for i in self.ids.box3.children:  i.clock.cancel()  for i in self.ids.box4.children:  i.clock.cancel()  for i in self.ids.box5.children:  i.clock.cancel()  for i in self.ids.box6.children:  i.clock.cancel()  for i in self.ids.box7.children:  i.clock.cancel()  for i in self.ids.box8.children:  i.clock.cancel()  for i in self.ids.box9.children:  i.clock.cancel()  game.puzzle, game.puzzleSolution, game.opponentGrid = None, None, None  self.ids.box1.clear\_widgets()  self.ids.box2.clear\_widgets()  self.ids.box3.clear\_widgets()  self.ids.box4.clear\_widgets()  self.ids.box5.clear\_widgets()  self.ids.box6.clear\_widgets()  self.ids.box7.clear\_widgets()  self.ids.box8.clear\_widgets()  self.ids.box9.clear\_widgets()  self.ids.numberGrid.clear\_widgets()  self.ids.opponentGrid.clear\_widgets() |
| --- |

## 

## 

## 

## 

## 

## 3.4 Account Management Screens

[main.py + Sudoku.kv](#_s1aqnzdbwr6j) ,***This fulfils Objectives 1, 2, 8***

### 3.4.1 Login Screen

| class Login(BaseScreen):  def togglePW(self):  self.ids.password.password = not(self.ids.password.password)  def toggleRememberLogin(self):  app.rememberLogin = not(app.rememberLogin)  def clickLogin(self):  un, pw = self.ids.username.text, self.ids.password.text  app.client = Client()  app.client.connect()  if app.client.connected:  if app.client.login(un, pw, hashed = False):  app.online = True  self.set\_Border()  if app.rememberLogin is True:  app.username = un  app.password = app.client.hash\_Password(pw)  app.save\_Game\_Data()  app.client.update\_BestTimes(app.topTimes)  else:  app.client.disconnect()  app.client = None  p = Popup(title = "Error", content = Label(text = "Invalid Username or Password"), size\_hint = (0.6, 0.3))  p.open()  else:  app.client.disconnect()  app.client = None  p = Popup(title = "Error", content = Label(text = "Could not connect to server, try again otherwise server must be offline"), size\_hint = (0.6, 0.3))  p.open() |
| --- |

#### Login in kv

| <Login>:  name : "Login"  Button:  text: "Return"  size\_hint : 0.07, 0.04  pos\_hint : {"x":0.005, "top": 0.99}  on\_release:  root.manager.transition = FadeTransition()  app.root.current = "AccountMenu"  AnchorLayout:  size\_hint : 0.6, 0.4  pos\_hint : {"center\_x": 0.5, "center\_y":0.45}  BoxLayout:  orientation : "vertical"  BoxLayout:  Label:  size\_hint : 0.25, 1  text : "Username"  color : 0, 1, 1, 1  TextInput:  size\_hint : 0.75, 1  id: username  hint\_text : "username"  multiline : False  BoxLayout:  Label:  size\_hint : 0.25, 1  text : "Password"  color : 0, 1, 1, 1  TextInput:  size\_hint : 0.5, 1  id : password  hint\_text : "password"  multiline : False  password : True  ToggleButton:  size\_hint : 0.25, 1  text : "toggle mask"  on\_press: root.togglePW()  Label:  text : ""  Button:  text:"Login"  font\_size : 25  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  pos\_hint: {"center\_x" : 0.5}  size\_hint : 0.3, 0.8  on\_release : root.clickLogin()  Label:  text : ""  Label:  text : ""  ToggleButton:  text:"Remember Login"  font\_size : 20  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  pos\_hint: {"center\_x" : 0.5}  size\_hint : 0.3, 0.8  on\_release : root.toggleRememberLogin() |
| --- |



### 

### 3.4.2 Register Screen

[main.py + Sudoku.kv](#_s1aqnzdbwr6j) ,

| class Register(BaseScreen):  def clickRegister(self):  print("Butoon Clicked")  un , pw1, pw2 = self.ids.username.text, self.ids.password1.text, self.ids.password2.text  if len(pw1) < 7 or len(pw2) < 7:  p = Popup(title = "Error", content = Label(text = "Password must be at least 7 characters long"), size\_hint = (0.6, 0.3))  p.open()  return  if pw1 != pw2:  p = Popup(title = "Error", content = Label(text = "Passwords do not match"), size\_hint = (0.6, 0.3))  p.open()  return  app.client = Client()  app.client.connect()  if app.client.connected:  if app.client.register(un, pw1):  loggedIn = app.client.login(un, pw1, False)      if loggedIn:  app.online = True  self.set\_Border()  app.username = un  app.password = app.client.hash\_Password(pw1)    p = Popup(title = "Success", content = Label(text = "Account has been created" + "\nAnd you have been logged in" if loggedIn else ""), size\_hint = (0.6, 0.3))  p.open()  else:  p = Popup(title = "Error", content = Label(text = "Username already taken"), size\_hint = (0.6, 0.3))  p.open()  else:  app.client.disconnect()  app.client = None  p = Popup(title = "Error", content = Label(text = "Could not connect to server"), size\_hint = (0.6, 0.3))  p.open() |
| --- |

#### 

| <Register>:  name : "Register"  Button:  text: "Return"  size\_hint : 0.07, 0.04  pos\_hint : {"x":0.005, "top": 0.99}  on\_release:  root.manager.transition = FadeTransition()  app.root.current = "AccountMenu"  AnchorLayout:  size\_hint : 0.6, 0.3  pos\_hint : {"center\_x": 0.5, "center\_y":0.5}  BoxLayout:  orientation : "vertical"  BoxLayout:  Label:  size\_hint : 0.25, 1  text : "Username"  color : 0, 1, 1, 1  TextInput:  size\_hint : 0.75, 1  id: username  hint\_text : "username"  multiline : False  BoxLayout:  Label:  size\_hint : 0.25, 1  text : "Password"  color : 0, 1, 1, 1  TextInput:  size\_hint : 0.75, 1  id : password1  hint\_text : "password"  multiline : False    BoxLayout:  Label:  size\_hint : 0.25, 1  text : "Re-Enter Password"  color : 0, 1, 1, 1  TextInput:  size\_hint : 0.75, 1  id : password2  hint\_text : "password"  multiline : False    Label:  text : ""  Button:  text:"Create Account"  font\_size : 25  color : 0.38, 0.13, 0.97, 1  background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"  pos\_hint: {"center\_x" : 0.5}  size\_hint : 0.3, 0.8  on\_release : root.clickRegister() |
| --- |



## 3.5 Informative Screens

[main.py + Sudoku.kv](#_s1aqnzdbwr6j), ***This fulfils Objectives 1, 2***

### 3.5.1 Best Times

| class BestTimes(BaseScreen):  def on\_enter(self):  self.ids.easyTime.text = self.get\_topTime(0)  self.ids.normalTime.text = self.get\_topTime(1)  self.ids.hardTime.text = self.get\_topTime(2)  self.ids.extraHardTime.text = self.get\_topTime(3)  def get\_topTime(self, index):  topTime = app.topTimes[index]  return game.parse\_Timer\_to\_String(topTime) if len(topTime) > 0 else "N/A" |
| --- |

#### 

|  |
| --- |



### 

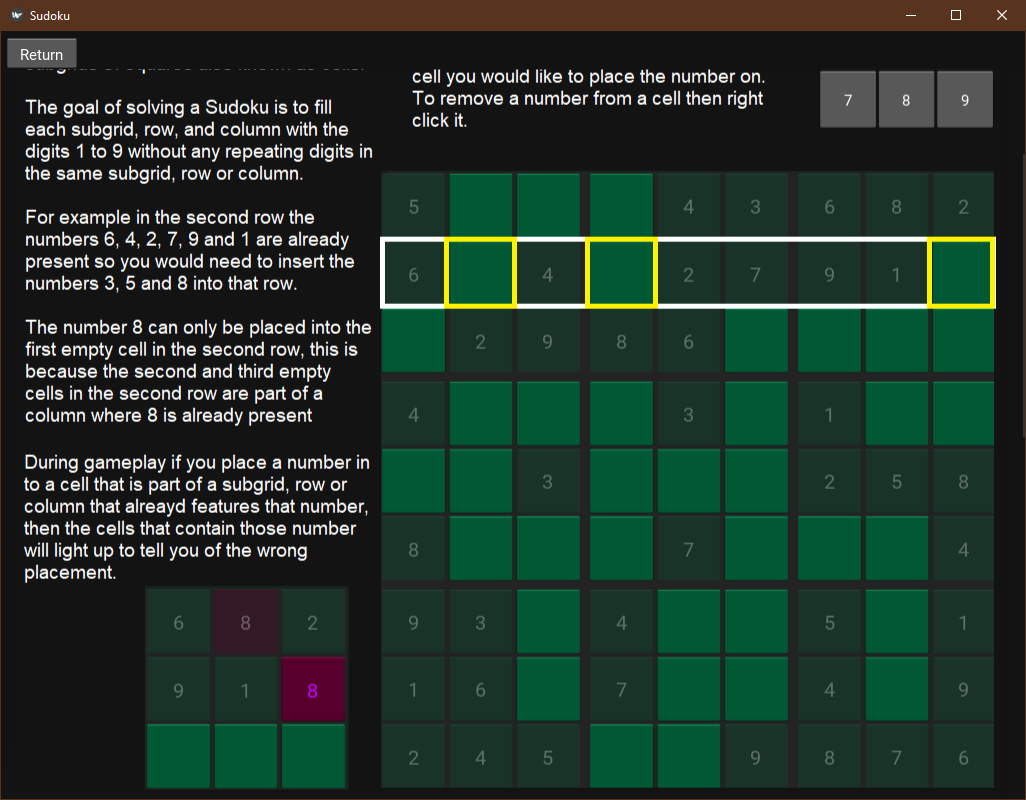
### 3.5.2 How To Play Screen

***This fulfills Objective 13***

| class HowToPlayScreen(BaseScreen):  pass |
| --- |

| <HowToPlayScreen>:  name : "HowToPlayScreen"  Button:  text: "Return"  size\_hint : 0.07, 0.04  pos\_hint : {"x":0.005, "top": 0.99}  on\_release:  root.manager.transition = SlideTransition()  root.manager.transition.direction = "down"  app.root.current = "MainMenu"  AnchorLayout:  size\_hint : 1, 0.96  pos\_hint : {"center\_x": 0.5, "center\_y":0.47}  ScrollView:  size\_hint : 1, 1  do\_scroll\_x : False  do\_scroll\_y : True  Image:  source: "Main\graphics\Sudoku\_App\_Help.png"  size\_hint: None, None  size : self.texture\_size |
| --- |

The How to Play Screen features a scrollable image that provides information to the user on how to play and interact with the game.



## 3.6 Server side (Basic)

[server.py](#_s1aqnzdbwr6j), ***This fulfils Objectives 7, 8, 10***

***A part of 4.14.3.4.1 Table 1 Group A, Complex client-server model + Server-side scripting using request and response objects and server-side extensions for a complex client-server model***

### 3.6.1 Main Server Setup

| host = "127.0.0.1" # IP address of the server  port = 7777 # Port number  DATABASE = "Server\Sudoku\_Online.db"  easyQ = Queue(25)  normalQ = Queue(25)  hardQ = Queue(25)  extraHardQ = Queue(25)  queueDict = {"easy": easyQ, "normal": normalQ, "hard": hardQ, "extraHard": extraHardQ}  def main():  server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # TCP socket  server.bind((host, port))  server.listen() # Ready to accept connections  threading.Thread(target = create\_Match).start() # Will create matches indefinitely  while True:  print("ACCEPTING NEW CONNECTIONS")  client, address = server.accept()  print(f"CONNECTION ACCEPTED FROM {address}")  Client(client, address).start() # put client into their own thread  if \_\_name\_\_ == "\_\_main\_\_":  print("\_\_SERVER STARTED\_\_")  main() |
| --- |

Sets up the server by binding the local address and port number that will be used for communicating. Once the server is listening it will continue accepting connections forever.

### 3.6.2 Client thread class

| class Client(threading.Thread):  def \_\_init\_\_(self, client, address):  super().\_\_init\_\_()  self.client = client  self.address = address  self.username = None  self.bestTimes = [None, None, None, None]  self.options = { "login": self.login,  "register": self.register,  "match\_Players": self.match\_Players,  "update\_BestTimes" : self.update\_BestTimes}  self.difficulty = None  self.matching = False  self.startMatchingTime = 0  self.inGame = False |
| --- |

The keys in the self.login dictionary are the same as the request that would be received from the client, the keys are used for fast access to the functions without any messy if statements in the client handler.

The difficulty, matching, startMatchingTime and inGame attributes are for the multiplayer aspect of the server system.

### The 3.6.3 Client handler

#### What the client thread runs

|  |
| --- |

***4.14.3.4.1 Table 1 Group A, Server-side scripting using request and response objects***

The client-run function continuously waits for requests from the client. If the client is in a game no requests are to be received by the handler as well as if the client is in a matching queue, otherwise the handler receives a request from the client and executes the related function from the request via dictionary mapping.

### 

### 3.6.4 Login

#### Login

| def login(self):  print("doing login stuff on sever")  self.client.send("proceed".encode())  details = self.client.recv(1024).decode().split(",")  conn = sqlite3.connect(DATABASE)  cursor = conn.cursor()  if self.verify(details[0], details[1], cursor):  print(f"Login good by {details[0]}")  self.client.send("valid".encode())  conn.close()  return True  else:  self.client.send("invalid".encode())  #print("Login bad")  conn.close()  return False |
| --- |

Accepts username and password from client and verifies that they are valid.

#### Verify

| def verify(self, username, password, cursor):#Checks username and password match  if self.check(username, cursor):  result = cursor.execute("SELECT Username, Password FROM Accounts WHERE Username = ? AND Password = ?", (username, password)).fetchone()  print(result)  if result is None:  return False  else:  usernameResult, passwordResult = result  if usernameResult == username and passwordResult == password:  return True  else:  return False  else:  return False |
| --- |

***4.14.3.4.1 Table 1 Group B, Single table SQL statements***

Checks that the username is in the database if not then returns False and then checks if the username and password match if so then returns True if not returns False.

### 3.6.5 Register

| def register(self):  print("Doing register stuff on self.client")  self.client.send("proceed".encode())  details = self.client.recv(1024).decode().split(",")  print(details)  conn = sqlite3.connect(DATABASE, check\_same\_thread=False)  cursor = conn.cursor()  if self.check(details[0], cursor):  self.client.send("invalid".encode())#USername already exists  conn.close()  return False    else:  cursor.execute("INSERT INTO Accounts (Username, Password) VALUES (?, ?)", (details))  cursor.execute("INSERT INTO BestTimes (Username, Easy, Normal, Hard, 'Extra Hard') VALUES (?, NULL, NULL, NULL, NULL)", (details[0],))  self.client.send("valid".encode())  conn.commit()  conn.close()  print("INserted")  return True |
| --- |

***4.14.3.4.1 Table 1 Group B, Single table SQL statements***

First checks if the username given is already present in the database if it is not it inserts the details into the database otherwise it returns false.

#### 

#### Check

| def check(self, username, cursor):  result = cursor.execute("SELECT Username FROM Accounts WHERE Username = ?", (username,)).fetchone()  print(f"result: {result}")  if result is None:  print("username not in database")  return False  elif username in result:  print("username in database")  return True |
| --- |

***4.14.3.4.1 Table 1 Group B, Single table SQL statements***

Checks if the username is present in the database.

### 3.6.6 Database

***4.14.3.4.1 Table 1 Group B, Simple database model, two interlinked tables.***

Accounts and BestTimes table, linked by Username

### 

### 

### 3.6.7 Update Best Times

|  |
| --- |

***4.14.3.4.1 Table 1 Group B, Single table SQL statements***

Inserts the user’s latest best times data into the Best Times table

### 3.6.8 Match Players

|  |
| --- |

***4.14.3.4.1 Table 1 Group A, Queue Operations***

Matches players into the queue of their wanted difficulty, if the queue is not full and has two players ready to play. Uses dictionary mapping to access the required queues.

## 

## 3.7 **Server side (Multiplayer)**

[server.py](#_s1aqnzdbwr6j)***, This fulfils Objectives 7, 10, 11***

***4.14.3.4.1 Table 1 Group A, Complex client-server model + Server-side scripting using request and response objects and server-side extensions for a complex client-server model***

### 3.7.1 Create Match Function

|  |
| --- |

***4.14.3.4.1 Table 1 Group A, Queue Operations***

The create match function is to run in the background continuously as long as the server program is on. It will check each match queue and if the queue is not empty and there are at least two players.

### 3.7.2 Game Class Initialisation

| class Game(threading.Thread):  def \_\_init\_\_(self, thread1, thread2, difficulty):  super().\_\_init\_\_()  self.player1 = thread1  self.player2 = thread2  self.puzzleString = self.import\_Puzzle(difficulty)  p = Puzzle(self.puzzleString)  p.show\_Grid()  p.solve()  self.solutionString = p.grid\_To\_String() |
| --- |

Sets players 1 and 2 as the two threads that were matched from whatever queue they were pulled from, and loads a puzzle string of the required difficulty as well the solution of that puzzle string.

## 

### 3.7.3 Game handler method

| def run(self):  finished = False  self.player1.client.send("Match Found".encode())  self.player2.client.send("Match Found".encode())  self.player1.client.send(self.puzzleString.encode())  self.player2.client.send(self.puzzleString.encode())  while finished is False:  p1Grid = self.player1.client.recv(1024).decode()  p2Grid = self.player2.client.recv(1024).decode()  if p1Grid == "WIN":  self.player2.client.send("LOSE".encode())  finished = True  self.player1.inGame = False  self.player2.inGame = False  elif p2Grid == "WIN":  self.player1.client.send("LOSE".encode())  finished = True  self.player1.inGame = False  self.player2.inGame = False    elif p1Grid == "QUIT":  self.player2.client.send("OPPONENT QUIT".encode())  finished = True  self.player1.inGame = False  self.player2.inGame = False    elif p2Grid == "QUIT":  self.player1.client.send("OPPONENT QUIT".encode())  finished = True  self.player1.inGame = False  self.player2.inGame = False  else:  self.player1.client.send(self.compare\_Puzzles(p2Grid).encode())  self.player2.client.send(self.compare\_Puzzles(p1Grid).encode()) |
| --- |

Continuously receives the game states from the clients, encodes them and sends them back to the opposite client, also checks for any wins and quits to handle them.

### 3.7.4 Game puzzle handling methods

| def import\_Puzzle(self, difficulty):  with open(f"Main\Generator\{difficulty}.txt", "r") as file:  puzzles = file.readlines()  return random.choice(puzzles)  def compare\_Puzzles(self, puzzleString):  #Makes a string of 1s and 0s, 1s showing correctly filled cells and 0s showing empty and incorrect cells  return "".join(["1" if puzzleString[i] == self.solutionString[i] else "0" for i in range(81)]) |
| --- |

A Comparison puzzle creates a string of 1s and 0s, where 1s show where the puzzle is correctly filled and 0s where it's empty or wrong.

## 

## 3.8 Client Side

[networking.py](#_s1aqnzdbwr6j), ***This fulfils Objectives 6, 7, 8, 10***

### 3.8.1 Client Class Initialisation

| class Client:  def \_\_init\_\_(self):  self.\_\_host = "127.0.0.1"  self.\_\_port = 7777  self.\_\_client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  self.connected = False  self.username = ""  def connect(self):  try:  self.\_\_client.connect((self.\_\_host, self.\_\_port))  self.connected = True  print("[Connection with server established]")  except:  print("[Connection could not be made]")  return False  def disconnect(self):  self.\_\_client.send("Logout".encode())  self.\_\_client.close()  self.connected = False  print("[Connection with server closed]")  def send(self, message):  self.\_\_client.send(message.encode())  def receive(self, n = 1024):# n = bytes to receive  return self.\_\_client.recv(n).decode()  def hash\_Password(self, password):  return gaviHash(password) |
| --- |

This is the interface that will be used in the main app to interact with the server.

#### Register

| def register(self, username, password):#Password here is unprocessed  print("Doing register stuff on client")  self.\_\_client.send("register".encode())  proceed = self.\_\_client.recv(1024).decode()  valid = False  if proceed == "proceed":  self.\_\_client.send((username+","+self.hashPW(password)).encode())  if self.\_\_client.recv(1024).decode() == "valid":  valid = True  else:  print("No proceed")  return valid |
| --- |

Sends the user’s username and hashed password to the server to register their details.

#### Login

| def login(self, username, password, hashed):  print("doing login stuff on client")  self.\_\_client.send("login".encode())  proceed = self.\_\_client.recv(1024).decode()  valid = False  if proceed == "proceed":  if hashed is True:  self.\_\_client.send((username+","+password).encode())  else:  self.\_\_client.send((username+","+self.hashPW(password)).encode())  if self.\_\_client.recv(1024).decode() == "valid":  valid = True  self.username = username  return valid |
| --- |

Sends the user’s username and hashed password to the server to login.

#### Update Best Times

| def update\_BestTimes(self, times):  print("Doing update best times stuff on client")  self.\_\_client.send("update\_BestTimes".encode())  proceed = self.\_\_client.recv(1024).decode()  if proceed == "proceed":  self.\_\_client.send((f"{self.username}," + ",".join(times)).encode()) |
| --- |

Sends the player’s latest best times to update the ones on the server’s database.

#### Enter Matchmaking

| def match\_Players(self, difficulty):  print("Doing login stuff on client")  self.\_\_client.send("match\_Players".encode())  proceed = self.\_\_client.recv(1024).decode()  if proceed == "proceed":  self.\_\_client.send(difficulty.encode())  message = self.\_\_client.recv(1024).decode()  if message == "Enqueued":  return True  elif message == "Queue full":  return False  else:  return #connection no go |
| --- |

Sends the required difficulty to the server to enqueue the player to the queue of respective difficulty.

#### hash\_Password

| def hash\_Password(self, password):  return gaviHash(password) |
| --- |

***4.14.3.4.1 Table 1 Group A, Hashing***

Hashes the password using the [gaviHash](#_9sptp5lty03o) algorithm.

## 3.9 Other Algorithms and Techniques

### 3.9.1 Stack

[Generate.py](http://generate.py)

| class Stack():  def \_\_init\_\_(self, size = 10):  self.\_\_stack = [None for i in range(size)]  self.\_\_top = -1  self.\_\_maxSize = size  def isFull(self):  return True if self.\_\_top == self.\_\_maxSize - 1 else False  def isEmpty(self):  return True if self.\_\_top == -1 else False    def pushToStack(self, item):  if self.isFull():  print("Stack Full")  else:  self.\_\_top += 1  self.\_\_stack[self.\_\_top] = item  def popFromStack(self):  if self.isEmpty():  print("Stack Empty")  return None  else:  item = self.\_\_stack[self.\_\_top]  self.\_\_top -= 1  return item  def show(self):  print(self.\_\_stack) |
| --- |

***4.14.3.4.1 Table 1 Group A, Stack model, helps achieve objective 3***

The stack follows the Last In First Out principle, so I coded the push and pop function to abide by it, where pushing to the stack will place an item at the top of the stack and popping from the stack will take an item from the top of the stack.

### 3.9.2 Circular Queue

[Server.py](#_s1aqnzdbwr6j)

| class Queue():  def \_\_init\_\_(self, maxSize=10):  self.\_\_queue = [None for i in range(maxSize)]  self.\_\_front = 0  self.\_\_rear = -1  self.\_\_size = 0  self.\_\_maxSize = maxSize  self.lock = threading.Lock()    def isFull(self):  return self.\_\_size == self.\_\_maxSize    def isEmpty(self):  return self.\_\_size == 0    def isEven(self):  with self.lock:  return self.\_\_size % 2 == 0    def enQueue(self, item):  with self.lock:  if self.isFull():  return item  else:  self.\_\_rear = (self.\_\_rear + 1) % self.\_\_maxSize  self.\_\_queue[self.\_\_rear] = item  self.\_\_size += 1  return True  def deQueue(self):  with self.lock:  if self.isEmpty():  return False  else:  data = self.\_\_queue[self.\_\_front]  self.\_\_front = (self.\_\_front + 1) % self.\_\_maxSize  self.\_\_size -= 1  return data  def show(self):  return [self.\_\_queue[(self.\_\_front + i) % self.\_\_maxSize]for i in range(self.\_\_size)] |
| --- |

***4.14.3.4.1 Table 1 Group A, Queue, helps achieve objective 10***

The queue follows the First In First Out principle. This specific queue structure is for queues of a fixed size. The circular queue model does not circle the items in the queue around as one may initially think but it circles the pointers around. This allows for the re-use of empty spaces in the queue made by dequeuing times.

The queues that are intended to be made with this model are going to be accessed by multiple threads, so the threading.lock enables a thread lock that enforces a rule that the queue can only be accessed by one thread at a time.

### 3.9.3 Hashing

[networking.py](#_s1aqnzdbwr6j)

#### gaviHash

| def gaviHash(data):  hash\_value = 0  x = len(data)  for i, char in enumerate(data):  # Mix operations: XOR, multiplication, bit-shifting  hash\_value ^= (ord(char) \* (x\*\*i))  hash\_value ^= (ord(char) << ((x-i) % 3)) \*\* ((x-i))    return hex(hash\_value)[2:] |
| --- |

***4.14.3.4.1 Table 1 Group A, Hashing, helps achieve objective 8***

The hashing algorithm performs operation on the hash based on its current character and index throughout the iteration. The algorithm takes the unicode of the current character and multiplies it by the length of the string to the power of the current index and then XORs it with the hash value. Then again it takes the unicode of the current character and shifts it by the length of the string minus the index mod 3 to the power of the length minus the index. This hashing algorithm will be used for hashing passwords of length greater than or equal to 7.

# 

# 4 TESTING

## 4.1 Test Plan

BlackBox Testing

| Will test the account creation and login process.  Will test the single-player win experience.  Will test the multiplayer win experience. |
| --- |

White Box Testing

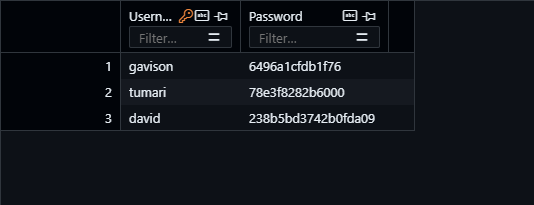
| Will test the puzzle solve and generation of puzzles, ensuring puzzles are valid. |
| --- |

## 4.2 Testing and Evidence

### 4.2.1 Testing Accounts System

| Test No | 1 |
| --- | --- |
| Objective | 8, 9, 12 |
| Purpose | Testing registering a new user, logging in and updating their best times, this process can work all in one go, so that is how we will be testing it. |
| Description | Register a new account  Auto Login  Auto update best times  Check the database for changes |
| Test Data | Username: “sherwin”  Password: “gaming8” |
| Expected Result | Successful register  Logged-in status on, green circle at bottom right  Correct best times from game data to database |
| Actual Result | Successful Register  Successful Login  Successful login status updated  Successful addition of username, and password to the Accounts table  Successful addition of username and best times to Best Times table |

Database Before





Database After

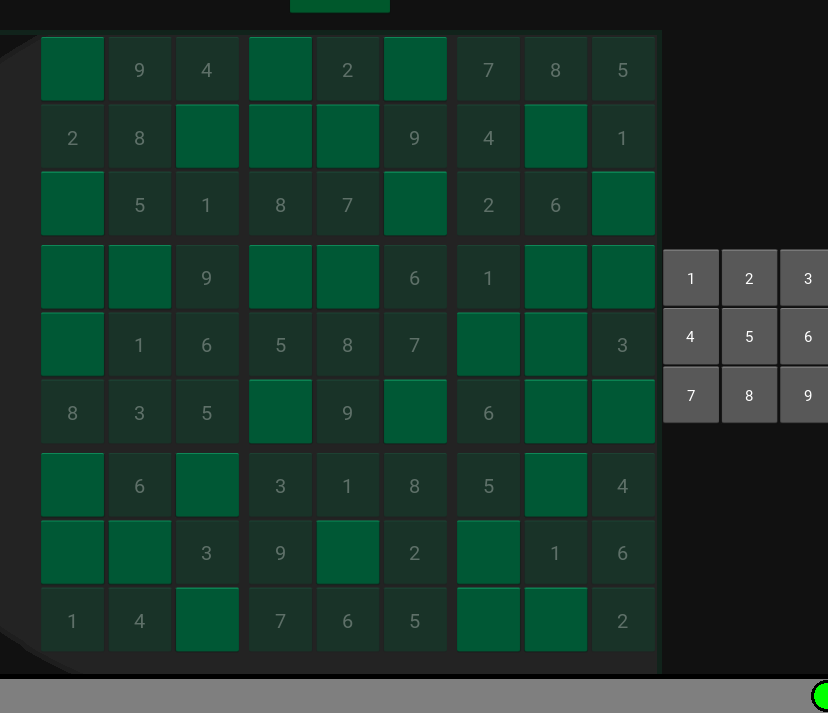




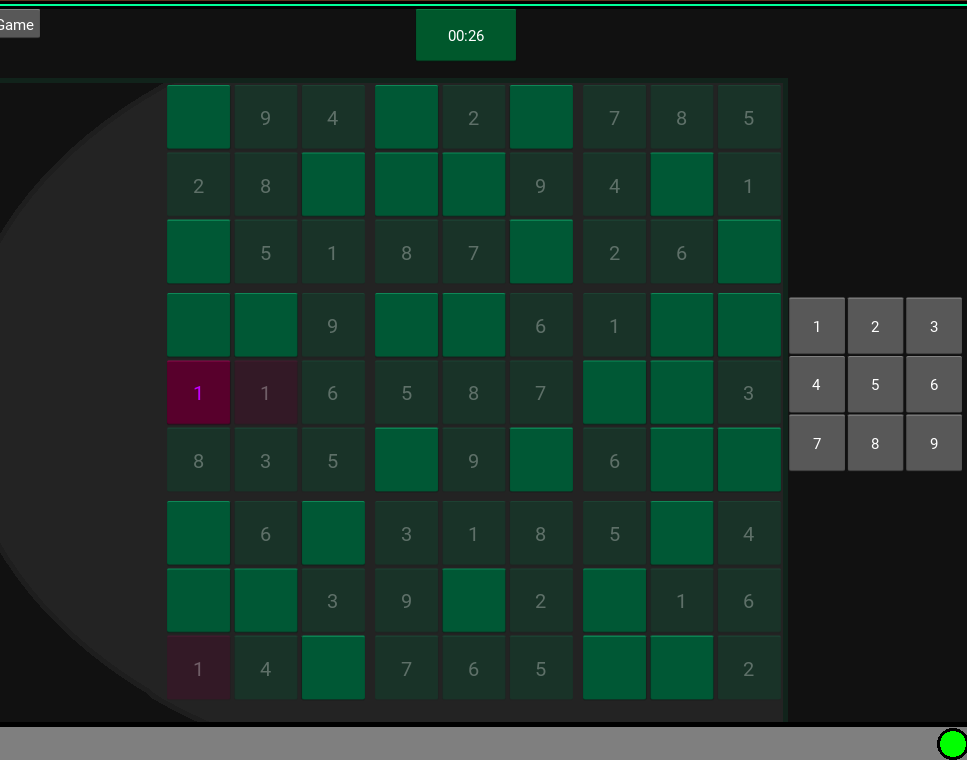
### 4.2.2 Testing Classic Mode

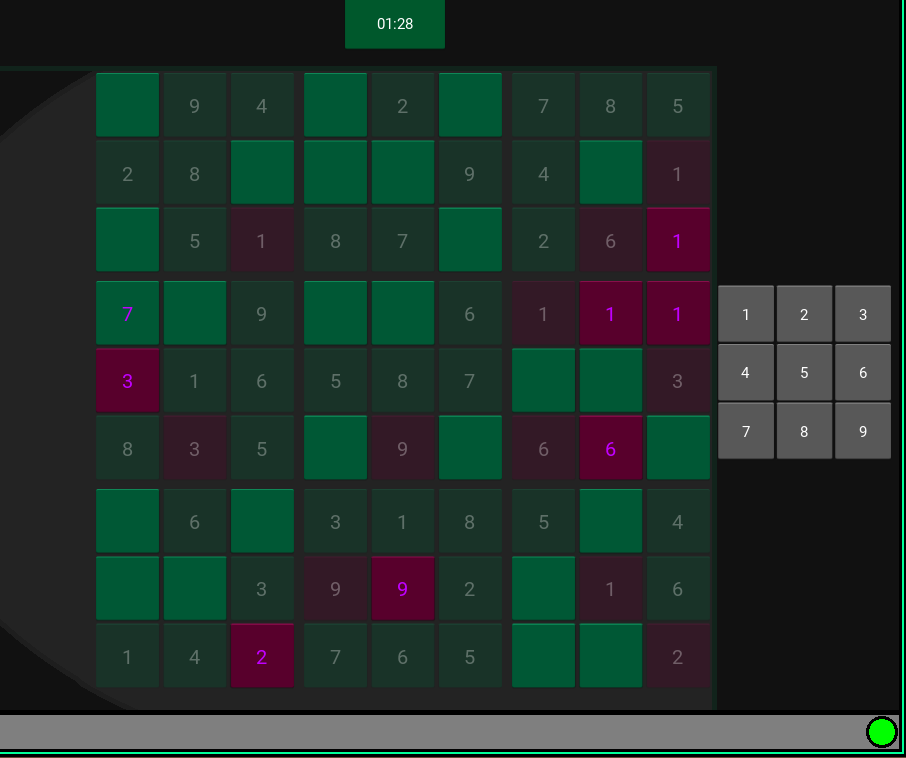
| Test No | 2 |
| --- | --- |
| Objective | 5 |
| Purpose | Test whether program detects collisions in cell placements |
| Description | Add a number into a cell, row, or grid that has the number number already inside of it, colliding cells should light up |
| Test Data | Given a grid, take a number from it that is already present in the subgrid, row, or column and add it to a space init |
| Expected Result | All colliding cells should light up |
| Actual Result | Colliding cells did indeed light up |

Puzzle grid before



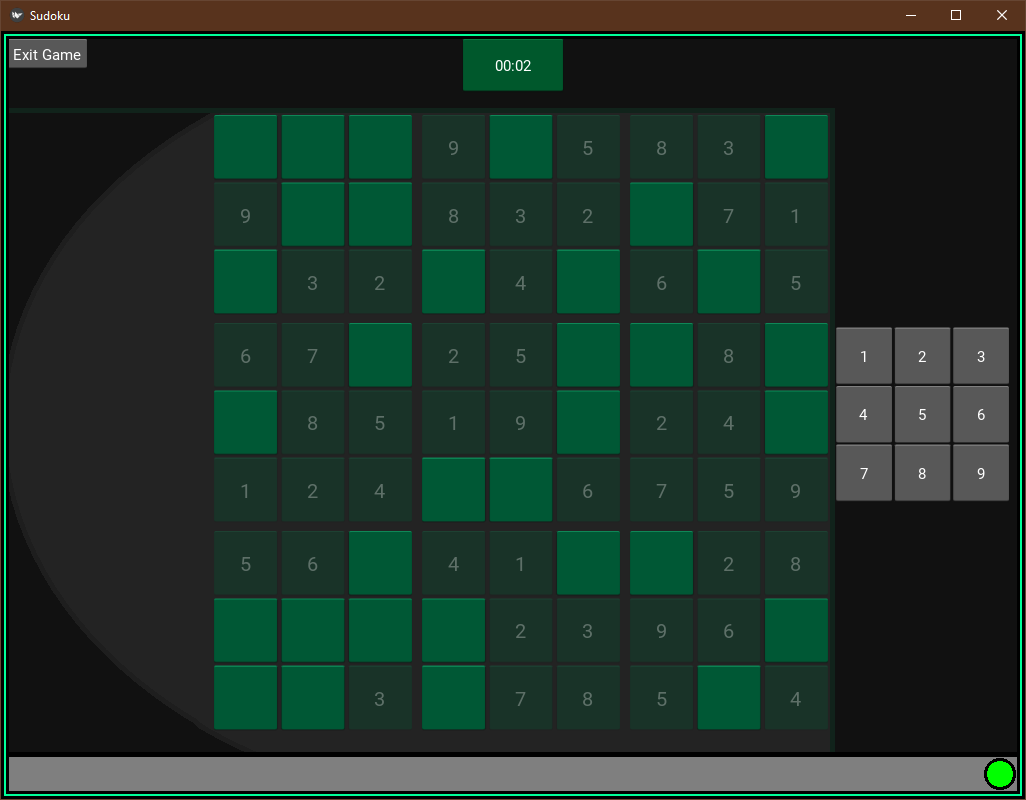
Puzzle grid after:



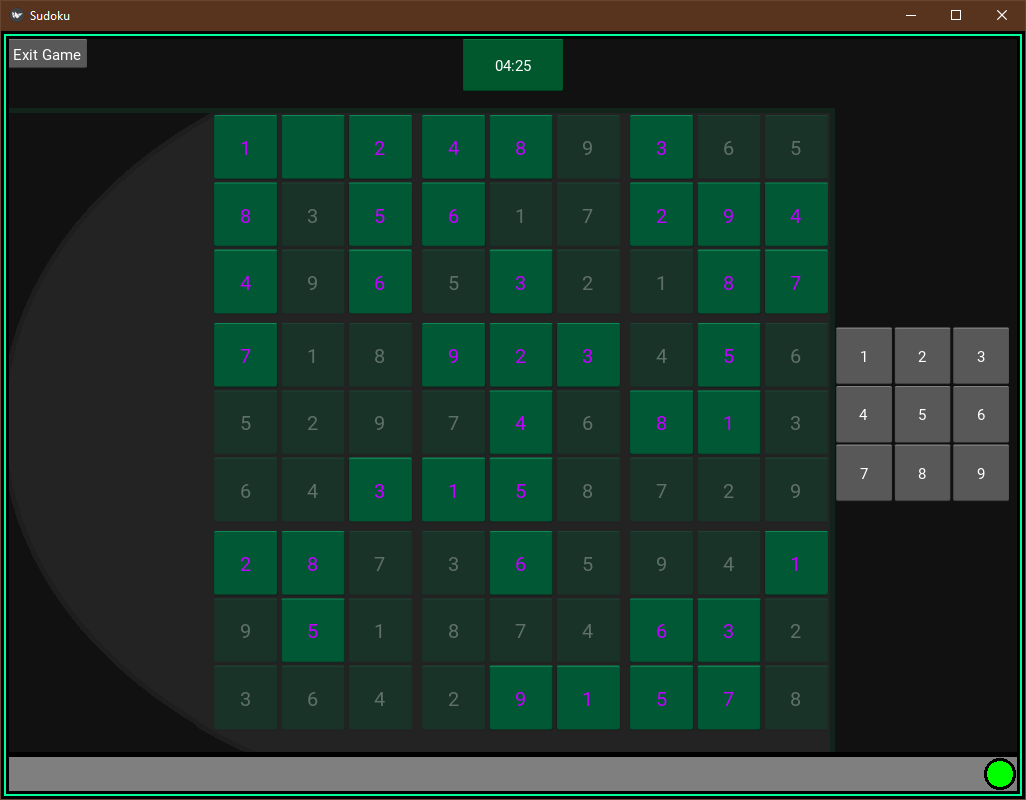


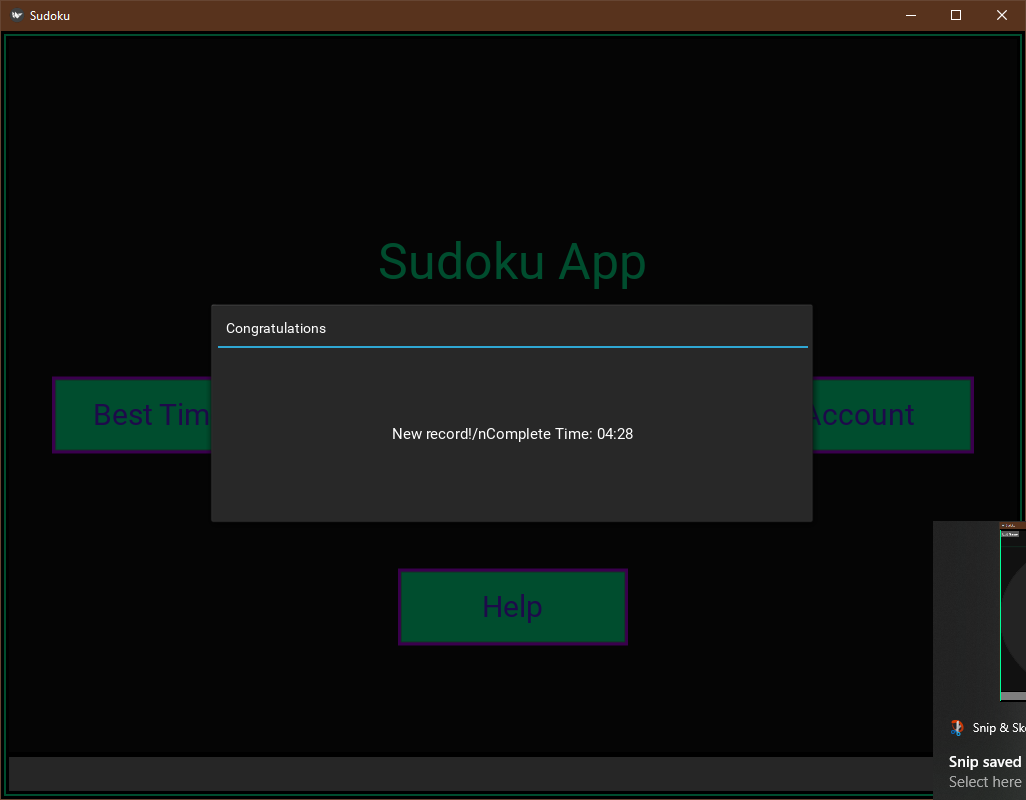
| Test No | 3 |
| --- | --- |
| Objective | 4, 5 |
| Purpose | Test game wins check that the game is won, without crashing and display an appropriate message. |
| Description | Load game  Finish puzzle correctly |
| Test Data | 000009065030017000090502100018000406529706003640008729007305940901874002364200008 is the puzzle grid  Set initial best time to 10 minutes |
| Expected Result | Win message for correct solution Finish puzzle before 10 mins to get a new best time |
| Actual Result | Win message for correct solution, logs new best time |

Before:



After:



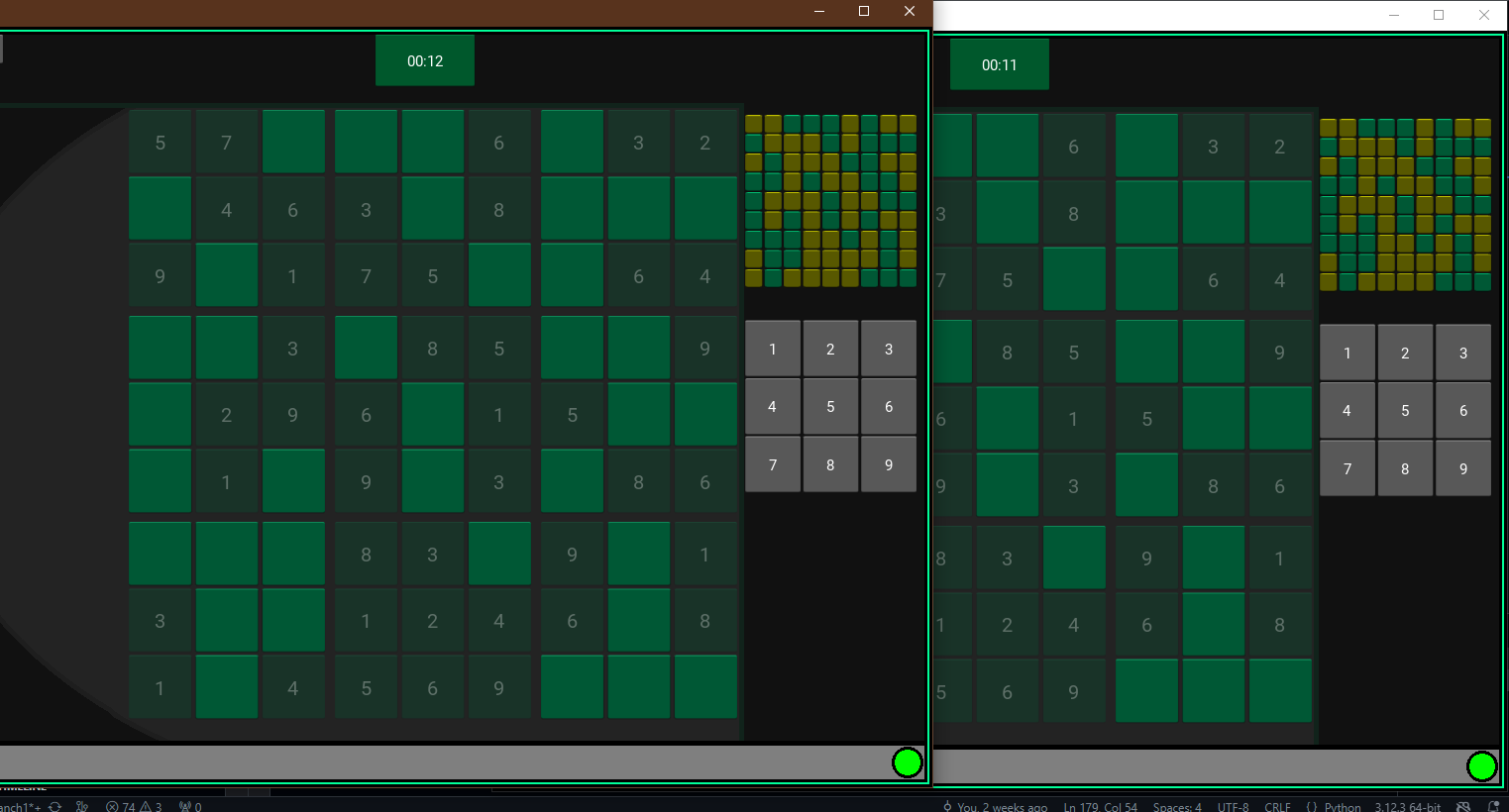


### 

### 4.2.3 Testing multiplayer

| Test No | 4 |
| --- | --- |
| Objective | 6, 7, 10, 11 |
| Purpose | Testing that the multiplayer game system acts accordingly |
| Description | Players login  They both go into matching for easy mode  They get put into the same match  A correct placement in player 1 should be reflected onto player 2’s opponent grid  On a player1 win player2 should receive a message that they lost |
| Test Data | Not required |
| Expected Result | Correct placement on player 1 reflected on player 2.  Player 1 win, Player 2 lose  Player 2 is alerted that they have lost |
| Actual Result | Correct placement on player 1 reflected on player 2.  Player 1 win, Player 2 lose  Player 2 is alerted that they have lost |

PLayer 1 on left, Player 2 on right



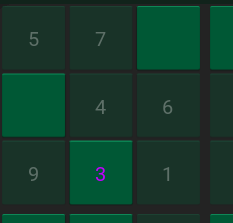
Changes are to be made on PLayer 1’s grid so we don't need to look at Player 2’s main grid, just its opponent grid.

Player 2 opponent grid before:



Look at cell location (row:3, column:2)

Player 1 places the correct number onto the top left subgrid on their grid



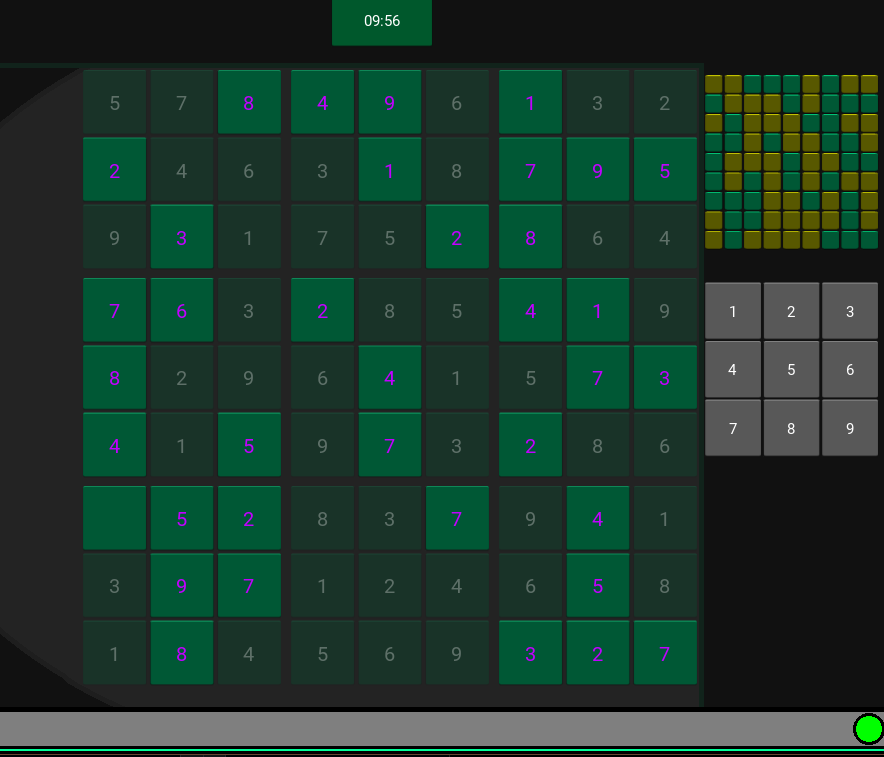
Player 2 opponent grid was updated to match the progress of Player 1’s grid



Look at cell location (row:3, column:2)

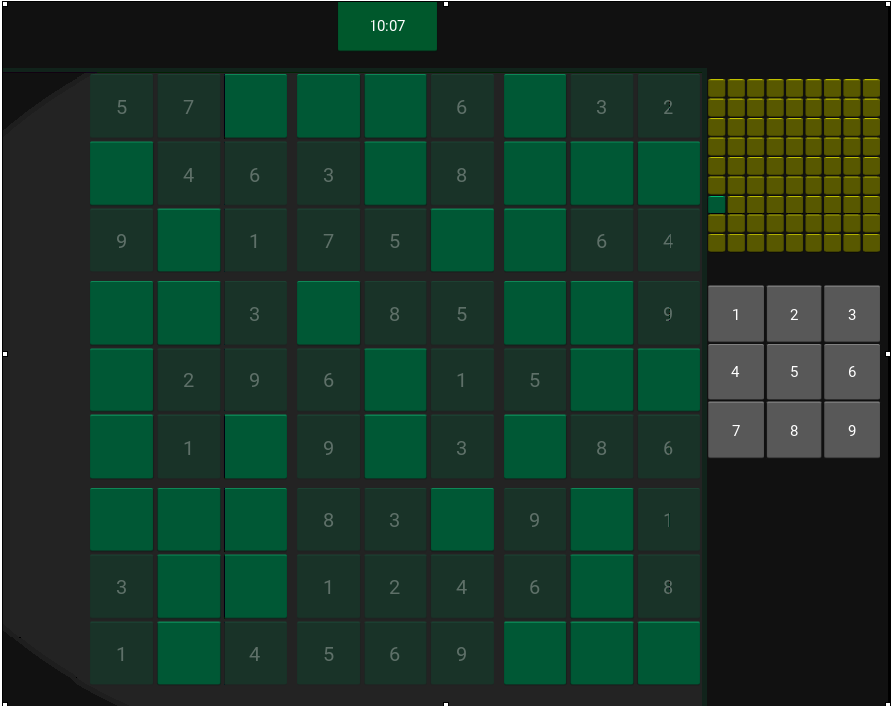
A yellow cell indicates a correct placement

Player 1 screen:



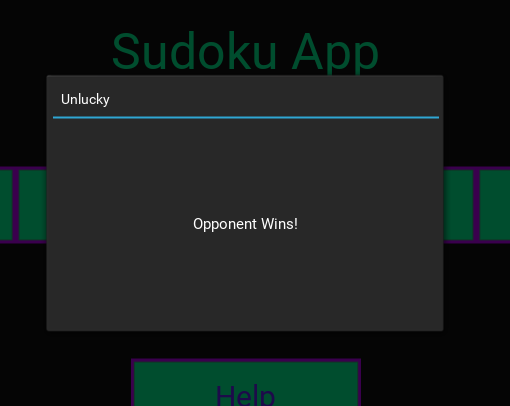
As you can see player 1’s grid is almost full and if you look below at player 2’s screen you can see their opponent's grid also shows that player 1’s grid is full.

Player 2 screen:



Player2’s opponent grid correctly reflects the progress of player 1.

PLayer 2 Screen after player 1 win:



### 4.2.4 Testing the Puzzle Generator

For testing the validity of the puzzles and their solutions we will be using a website called

<https://www.thonky.com/sudoku/solution-count>

| Test No | 5 |
| --- | --- |
| Objective | 3 |
| Purpose | To test the validity of the puzzles generated |
| Description | Generate an amount of puzzles and put those through the website linked above |
| Test Data | Puzzle string generated |
| Expected Result | All strings should have one solution |
| Actual Result | Puzzle generated “000000000000000000000000000000000000000000000000000000000000000000000000000000000” this is obviously wrong |

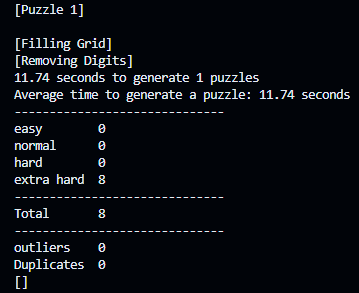
After some consideration I figured that the problem was due to the fact that the count solution was going through only some possible search space, which as a solution counter it needs to do. This is because I had designed the solution counter off of the optimised puzzle solver, however the puzzle solver is intended for puzzle with only one solution so the candidate reduction from the puzzle solver that was applied to the solution counter reduced the search space, and so the the solution was not able to count up all the solution and verify the validity of the puzzle generated properly, resulting in invalid puzzle.

Below is the wrong implementation of the solution counter.

| def count\_Solutions(self):  #print("[COUNTING SOLUTIONS]")  pos = self.find\_Empty\_Space()#pos is given as a tuple (row, col)  if pos == None:#Meaning the grid is full and solved  self.solutions += 1 #Notes that it has found a solution  return #Continues "EXPLORING" grid for more solutions    row, col = pos  for n in self.candidates[row][col]:  if self.check(row, col, n):  self.insert(row, col, n)  self.update\_Peers\_Remove\_Candidates(row, col, n)    self.count\_Solutions()  if self.solutions > 1:  #More than one solution is found so we stop searching  return    self.insert(row, col, 0)  self.update\_Peers\_Insert\_Candidates(row, col, n) |
| --- |

Below is the correct implementation of the solution counter

| def count\_Solutions(self):  #print("[COUNTING SOLUTIONS]")  pos = self.find\_Empty\_Space()#pos is given as a tuple (row, col)  if pos == None:#Meaning the grid is full and solved  self.solutions += 1 #Notes that it has found a solution  return #Continues "EXPLORING" grid for more solutions    row, col = pos  for n in range(1, 10):  if self.check(row, col, n):  self.insert(row, col, n)  self.count\_Solutions()  if self.solutions > 1:  #More than one solution is found so we stop searching  return    self.insert(row, col, 0) |
| --- |

****

**OUTPUT RESULT AFTER CHANGE :**

004200005300400207070300900000037020000500300000600400090705000010900004708046000

708046000010900004090705000000600400000500300000037020070300900300400207004200005

700000030019000700800000004097650342400003000605007000000430920000002000040000075

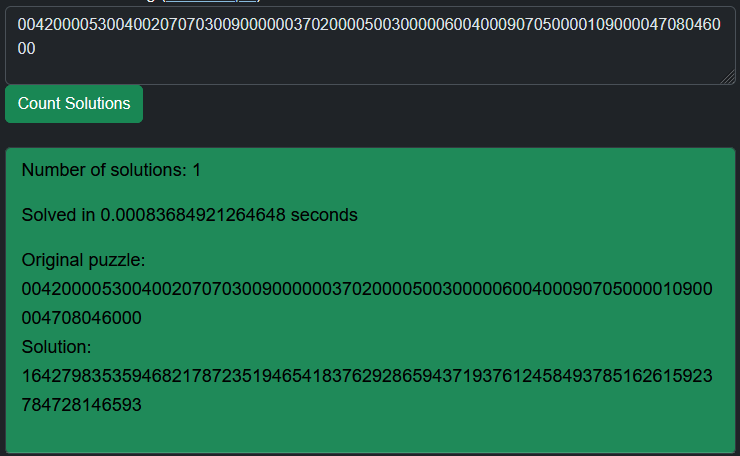
030000007007000910400000008243056790000300004000700506029034000000200000570000040

000640807400009010000507090004006000003005000020730000009003070702004003500002400

500002400702004003009003070020730000003005000004006000000507090400009010000640807

570000040000200000029034000000700506000300004243056790400000008007000910030000007

040000075000002000000430920605007000400003000097650342800000004019000700700000030



The puzzle generated is confirmed to be unique.

## 4.3 Testing Videos

Look into 4.3\_Testing Videos Folder

### 4.3.1 Navigating the GUI

***Testing Objectives: 1, 2, 4, 5***

At times 0:28 to 0:50 you can see the user going through all the screens.

At times 0:50 to end you can see the user interacting with the classic mode.

At 0:53 you can see the user pausing the game and onwards can see the grid responding to incorrect placements.

At 1:06 and onwards you can see that the user exits their current game then goes into another one where the grid refreshes and the timer resets..

### 4.3.2 Account System

***Testing Objectives: 1, 7, 8, 9, 12***

At time 0:37 you can see the user create a new account and login.

At time 1:06 you can see that the new account created has been added to the database.

At time 1:34 you can see the user logging in with remember login then they close the app.

At time 1:46 you can see the user opening the app and it automatically logging them in.

At time 1:58 you can see that the “Game Data.txt” has been updated to include the user’s details.

### 4.3.3 Multiplayer Gameplay

***Testing Objectives: 1, 6, 7, 10, 11***

Let player1 be left and player2 be right

At time 0:21 you can see both players open and logged in.

At time 0:28 you can see them both matching into the easy mode for multiplayer.

At time 0:50 you can see player1 p[lace a 1 into a spot on the grid and on player2’s screen you can see their mini opponent grid have a change where the respective cell turns yellow, this is to show that player1 has made a correct placement in that cell.

At time 1:20 you can see that fast forward player1 is almost finished and that is also reflected on player2’s mini grid.

At time 1:24 player1 finishes their grid and beats player2, player1 and player2 receive appropriate messages and are put back into the main menu.

# 

# 5 EVALUATION

## 5.1.1 Revisiting the Objectives

| No. | Objective | Performance Criteria | Evaluation |
| --- | --- | --- | --- |
| 1 | Create an app where users can solve sudoku puzzles | Must have a variety of puzzles to play with a variety of difficulties. | Achieved, users are able to solve puzzles of difficulties that they choose out of 4. |
| 2 | The app must be responsive and easy to navigate | Must have easy controls so accessible to all peoples and be able to work smoothly. | Achieved, the application smoothly transitions between screens and has clear back buttons. |
| 3 | Generate and store unique sudoku puzzles for the app. | The puzzle must be sorted into 4 difficulty categories; easy, normal, hard and extra hard. Must be able to recycle the puzzle by flipping and rotating the grid to create more puzzles. | Achieved  Able to generate various puzzles and takes an average of 7s to generate each puzzle. |
| 4 | Implement Classic mode | In a single-player game, the user can solve a puzzle of chosen difficulty and be timed for it. | Achieved |
| 5 | Create a responsive GUI for the game mode | The puzzle grid must respond to number placements, highlight cells for collisions and have a timer that can pause. | Achieved  Grid automatically highlights placement errors and instantly detects completion. |
| 6 | Implement Multiplayer mode | Two users must be able to match online and race to see who can finish the puzzle first | Achieved  Users are able to connect to each other and view each other's progress via a special mini grid. |
| 7 | Create a server to connect to clients, manage accounts and facilitate the multiplayer mode | Accept requests from clients and deliver appropriate responses. | Achieved  Able to create persistent connections between the client and server and interact with it using requests. |

## 

## 

| 8 | Create an Account system | Users are to be able to register and login to accounts as well as have payer data saved online. | Achieved  Able to store username and password as well as best times in the database and be able to login using the username and password. |
| --- | --- | --- | --- |
| 9 | Create a database to store account information for the account system | Consisting of two tables, Accounts and Best Times.  Accounts ← username, password  BestTimes ← username, easy, normal, hard, extra\_hard | Achieved |
| 10 | Create a matching system for the multiplayer mode | Players are able to randomly match against other players.  To enter players into queues of chosen game difficulty and match them to other players. | Achieved  Matches players into queues and puts them into matches appropriately. |
| 11 | Create the multiplayer game system | To send each player the puzzle grid and to receive each player's progress and relay it to the other player till one of them wins. | Achieved  Controls the communication between players and the server during a multiplayer game flawlessly. |
| 12 | Store the user’s username, password and best times in a local file. | A username and password will be used to automatically login to the server. | Achieved |

## 

## 5.1.2 End User Feedback

| Great UI, easy to navigate. The little green dot at the bottom is a great way to tell if I am logged in or not. I find it easy to add and remove numbers from cells and it is good that wrong cell placements are made obvious by the highlighting. However, you could implement a note-taking feature to keep track of possible candidates for cells. The help screen is good but it could have more content such as guidance to more complex strategies. |
| --- |

## 5.14 Improvements to be made

* Add in specific match requests for the matchmaking system
* Include the ability to view candidates/ take notes on the empty cells in the classic game mode.
* Add more info on the tutorial screen
* Include the ability to insert numbers via the keyboard.
* Improve the look of the Best Items screen.
* Improve the look of the toggleable buttons when they are toggled.
* Refine the puzzle-generating process, to generate puzzles of higher difficulty and sort puzzles also based on the strategies required to solve them.
* Save more than just the best times, like the top 10 times and note the exact puzzles that were solved with those times, this could also be stored on the database

# Appendix 1, Generator.py

from multiprocessing.spawn import prepare

import random

import time

import copy

class Puzzle:

def \_\_init\_\_(self, data = None):

if type(data) == str:#For importing grids from a string

self.string\_To\_Grid(data) # Converts a string representation of a puzzle into a 2d array

elif type(data) == list: # For importing grid as 2d array, used for converting grid into string

self.grid = data

else: # Default

self.generate()

def show\_Grid(self):

for row in self.grid:

for item in row:

print(item, end = " ")#continues on same line

print()#newline between rows

print()

def insert(self, row, col, n):

self.grid[row][col] = n

def find\_Empty\_Space(self) -> tuple | None:

for row in range(9):

for col in range(9):

if self.grid[row][col] == 0:

return (row, col)

return None

def check(self, row, col, num) -> bool:#return false if there are any mistakes

if num in self.grid[row]:

return False

for i in range(9):

if num == self.grid[i][col]:

return False

boxRow = row // 3 \* 3 #Reduces numbers to either 0, 3 or 6 the

boxCol = col // 3 \* 3 #starting indexes for a cells respective box

for i in range(boxRow, boxRow+3):

for j in range(boxCol, boxCol+3):

if num == self.grid[i][j]:

return False

return True

"""

def solve(self):#Basic dfs Solve function

pos = self.find\_Empty\_Space()

if pos == None:

return True

row, col = pos[0], pos[1]

for n in range(1, 10):

if self.check(row, col, n):

self.insert(row, col, n)

if self.solve():

return True

self.insert(row, col, 0)#if a solve does not happen then position goes back to zero in case further backtracking needed

return False

"""

def get\_All\_Candidates(self):

self.candidates = [[set() for i in range(9)]for j in range(9)]

#All cells start with empty candidates set

for row in range(9):

for col in range(9):

if self.grid[row][col] == 0:

#If cell is empty then it will get all possible candidates

candidates = set(range(1,10))

candidates -= set(self.grid[row])

candidates -= set(self.grid[i][col] for i in range(9))

boxRow = row // 3 \* 3

boxCol = col // 3 \* 3

candidates -= set(self.grid[i][j]

for j in range(boxCol, boxCol+3)

for i in range(boxRow, boxRow+3))

self.candidates[row][col] = candidates

#Cells that are not empty will have empty sets

def sort\_Candidates(self):

self.sortedCandidates = sorted( [(self.candidates[row][col], row, col)

for col in range(9)

for row in range(9)

if len(self.candidates[row][col]) > 0],# Ensures that only cells with candidates are considered

key = lambda x: len(x[0]) )# Sorted in ascending order in number of candidates

###########################\_EXPERIMENTAL\_CODE\_NOT\_NECESSARY\_############################################################################################

def update\_Peers\_Remove\_Candidates(self, row, col, n):

#When a number is inserted, the cells in the same row, column and box will have that number removed from their candidates

self.candidates[row][col] = set()#Number is inserted therefore no candidates

#No need to check if it is present, discard removes the number from the set if it is

for i in range(9):

self.candidates[row][i].discard(n)

self.candidates[i][col].discard(n)

boxRow = row // 3 \* 3

boxCol = col // 3 \* 3

for i in range(boxRow, boxRow+3):

for j in range(boxCol, boxCol+3):

self.candidates[i][j].discard(n)

def update\_Peers\_Insert\_Candidates(self, row, col, n):

#For adding candidates back to a cell that was previously filled

candidates = set(range(1,10))

candidates -= set(self.grid[row])

candidates -= set(self.grid[i][col] for i in range(9))

boxRow = row // 3 \* 3

boxCol = col // 3 \* 3

candidates -= set(self.grid[i][j]

for j in range(boxCol, boxCol+3)

for i in range(boxRow, boxRow+3))

self.candidates[row][col] = candidates

#For the peers of the cell adds the number back to their candidates

for i in range(9):

if self.grid[row][i] == 0 and self.check(row, i, n):

self.candidates[row][i].add(n)

if self.grid[i][col] == 0 and self.check(i, col, n):

self.candidates[i][col].add(n)

for i in range(boxRow, boxRow+3):

for j in range(boxCol, boxCol+3):

if self.grid[i][j] == 0 and self.check(i, j, n):

self.candidates[i][j].add(n)

#############################################################################################################################

def constraint\_Propagation(self):

self.get\_All\_Candidates()

self.sort\_Candidates()

return self.eliminate()

def eliminate(self):

if len(self.sortedCandidates) < 1:

if self.find\_Empty\_Space() is None:

return True#Completly solved using constraint propagation

else:

return False#Likely unsolvable, resort to backtracking to double check

emptySpace = self.sortedCandidates.pop(0)#items ast the fornt of the list have fewer candidates

candidates, row, col = emptySpace

if len(candidates) == 1:

#Meaning there is only one possible number that can be placed into the grid,

self.insert(row, col, self.candidates[row][col].pop())

self.update\_Peers\_Remove\_Candidates(row, col, self.grid[row][col])

self.sort\_Candidates()

return self.eliminate()

else:#No more single candidate sets, will resort to backtracking

self.sortedCandidates.insert(0, emptySpace)

return False

def dfs(self) -> bool:#Named after Depth First Search comes with basic backtracking

pos = self.find\_Empty\_Space()#pos is given as a tuple (row, col)

if pos == None:

#Meaning the grid is full and solved

return True

row, col = pos

for n in self.candidates[row][col]:

self.insert(row, col, n)

self.update\_Peers\_Remove\_Candidates(row, col, n)#Removes the number from the candidates of the peers

if self.dfs():#Branches off

#Causes all the recursion to unwind

return True

self.insert(row, col, 0)#If the branch does not lead to a solution, the number is removed

self.update\_Peers\_Insert\_Candidates(row, col, n)#Adds the number back to the candidates of the peers

return False

def solve(self) -> bool:

if self.constraint\_Propagation():

#print("[SOLVED VIA CONSTRAINT PROPAGATION]")

return True

else:

if self.dfs():

#print("[SOLVED VIA BACKTRACKING]")

return True

else:

#print("[UNSOLVABLE]")

return False

##############################################################################################################################

def fill\_Grid(self):

print("[Filling Grid]")

solved = False

while solved == False:

self.grid = [[0 for col in range(9)] for row in range(9)]

count = {}

for i in range(1, 10):

#Will be used to keep track of how many of each number is inserted

count[i] = 0

numberOfCellsToInsert = random.randint(25, 35)

while numberOfCellsToInsert > 0 :

x = random.randint(1, 9)

row = random.randint(0, 8)

col = random.randint(0, 8)

if self.grid[row][col] == 0:

if count[x] < 9 and self.check(row, col, x):

#Ensures that no more than nine iterations of the same number

self.insert(row, col, x)

count[x] += 1

numberOfCellsToInsert -= 1

solved = self.solve()#Fills the rest fo the grid by trying to solve it

def count\_Solutions(self):

#print("[COUNTING SOLUTIONS]")

pos = self.find\_Empty\_Space()#pos is given as a tuple (row, col)

if pos == None:#Meaning the grid is full and solved

self.solutions += 1 #Notes that it has found a solution

return #Continues "EXPLORING" grid for more solutions

row, col = pos

for n in range(1, 10):

if self.check(row, col, n):

self.insert(row, col, n)

self.count\_Solutions()

if self.solutions > 1:

#More than one solution is found so we stop searching

return

self.insert(row, col, 0)

def remove\_digits(self) -> int:

print("[Removing Digits]")

self.solutions = 0

digitsToRemove = int(random.triangular(36, 64, 50))# Weighted towards 50

digitsRemoved = 0 #Used to count how many digits have been removed

cells = [(row, col) for col in range(9) for row in range(9)]\*3

#All locations are set three times so that algorithm does not end too soon

random.shuffle(cells)

for row, col in cells:#While all cells haven't been visited thrice

if self.grid[row][col] == 0:

continue

n = self.grid[row][col]

self.insert(row, col, 0)

#count\_Solutions alters the grid so a copy is made

change = copy.deepcopy(self.grid)

self.count\_Solutions()

if self.solutions > 1:

#If more than one solutions are found, undo removal of digit

self.grid = copy.deepcopy(change)

self.insert(row, col, n)

elif self.solutions == 1:

self.grid = copy.deepcopy(change)

digitsRemoved += 1

self.solutions = 0

if digitsRemoved == digitsToRemove:

#if all required number of cells to be removed have been removed

break

self.grid = copy.deepcopy(change)

return 81 - digitsRemoved # = number of clues

def generate(self):

self.fill\_Grid()

self.clues = self.remove\_digits()

#print(f"Clues: {self.clues}")

##############################################################################################################################

def grid\_To\_String(self) -> str:

#Will convert the grid into a string that can be saved on a file

return "".join( ["".join([str(item) for item in row]) for row in self.grid] )

def string\_To\_Grid(self, string : str):

#the if statement accounts for grids that use a dot instead of 0

self.grid = [[int(item) if item != "." else 0

for item in string[9\*row:9\*(row+1)]]

for row in range(9)]

##############################################################################################################################

class PuzzleFile:

def \_\_init\_\_(self, file : str, mode : str, data : list = []):

self.duplicates = 0

if mode == "read":

self.file = open(file, "r")

self.contents = self.file.readlines()

self.file.close()

elif mode == "append":

self.file = open(file, "a+")

self.contents = self.file.readlines()

for puzzleString in data:

if not f"{puzzleString}\n" in self.contents:

self.file.write(f"{puzzleString}\n")

else:

self.duplicates += 1

self.file.close()

################################################################################################

class Stack():

def \_\_init\_\_(self, maxSize = 10):

self.\_\_stack = [None for i in range(maxSize)]

self.\_\_top = -1

self.\_\_maxSize = maxSize

def is\_Full(self):

return True if self.\_\_top == self.\_\_maxSize - 1 else False

def is\_Empty(self):

return True if self.\_\_top == -1 else False

def push\_To\_Stack(self, item):

if self.is\_Full():

print("Stack Full")

else:

self.\_\_top += 1

self.\_\_stack[self.\_\_top] = item

def pop\_From\_Stack(self):

if self.is\_Empty():

print("Stack Empty")

return None

else:

item = self.\_\_stack[self.\_\_top]

self.\_\_top -= 1

return item

def show(self):

print(self.\_\_stack)

###############################################################################################################################

"""puzzle1 = Puzzle()

#puzzle1.get\_All\_Candidates()

#print(puzzle1.candidates)

x = time.time()

puzzle1.solve()

y = time.time()

print("1", y-x)

puzzle2 = Puzzle()

x = time.time()

puzzle2.solve()

y = time.time()

print("2", y-x)

"""

"""puzzle3 = puzzle()

print(puzzle3.solveH())

puzzle3.show\_Grid()"""

def flip\_Vertical(grid : list) -> list:

newGrid = Stack(9)

for row in grid:

newGrid.push\_To\_Stack(row)

return [newGrid.pop\_From\_Stack() for i in range(9)]

def rotate\_90(grid : list) -> list:

newGrid = [[], [], [], [], [], [], [], [], []]

for col in range(9):#for each column

for row in range(8, -1, -1):#for each item in column, going down to up, this transposes the first column to the first row, with the bottom of the column aligned with the start of the row

newGrid[col].append(grid[row][col])

return newGrid

def make\_More(grid : list) -> list:

more = []

more.append(grid)

flipped = flip\_Vertical(grid)

more.append(flipped)

for i in range(3): # 90, 180, 270

grid = rotate\_90(grid)

more.append(grid)

flipped = rotate\_90(flipped)

more.append(flipped)

return more

if \_\_name\_\_ == "\_\_main\_\_":

numberToGenerate = 5 #n\*8 puzzles will be generated

easy = []

normal = []

hard = []

extraHard = []

outliers = []

listOfPuzzles = []

times = []

startTime = time.perf\_counter()

for i in range(numberToGenerate):

print(f"[Puzzle {i+1}]")

print()

listOfPuzzles.append(Puzzle())

elapsedTime = time.perf\_counter() - startTime

print(f"{round(elapsedTime, 2)} seconds to generate {numberToGenerate} puzzles")

print(f"Average time to generate a puzzle: {round((elapsedTime)/numberToGenerate, 2)} seconds")

for puzzle in listOfPuzzles:

if puzzle.clues in range(17, 28):

a = make\_More(puzzle.grid)

for i in a:

extraHard.append(Puzzle(i).grid\_To\_String())

elif puzzle.clues in range(28, 32):

a = make\_More(puzzle.grid)

for i in a:

hard.append(Puzzle(i).grid\_To\_String())

elif puzzle.clues in range(32, 36):

a = make\_More(puzzle.grid)

for i in a:

normal.append(Puzzle(i).grid\_To\_String())

elif puzzle.clues in range(36, 45):

a = make\_More(puzzle.grid)

for i in a:

easy.append(Puzzle(i).grid\_To\_String())

else:

outliers.append(puzzle.grid\_To\_String())

easyFile = PuzzleFile("Main/Generator/easy.txt", "append", easy)

normalFile = PuzzleFile("Main/Generator/normal.txt", "append", normal)

hardFile = PuzzleFile("Main/Generator/hard.txt", "append", hard)

extraHardFile = PuzzleFile("Main/Generator/extra\_hard.txt", "append", extraHard)

print("------------------------------")

print(f"easy {len(easy)}")

print(f"normal {len(normal)}")

print(f"hard {len(hard)}")

print(f"extra hard {len(extraHard)}")

print("------------------------------")

print(f"Total {numberToGenerate\*8}")

print("------------------------------")

print(f"outliers {len(outliers)}")

print(f"Duplicates {easyFile.duplicates + normalFile.duplicates + hardFile.duplicates + extraHardFile.duplicates}")

print(outliers)

print()

"""

#p = Puzzle("008034610043052897056187000007040560091063700465801009000000953539720040600390070")

p = Puzzle("8..........36......7..9.2...5...7.......457.....1...3...1....68..85...1..9....4..")

p.show\_Grid()

st = time.perf\_counter()

p.solve()

et = time.perf\_counter() - st

print(et)

p.show\_Grid()

print(p.grid\_To\_String())

"""

# Appendix 2, main.py

import random

import threading

from kivy.app import App

from kivy.uix.label import Label

from kivy.uix.screenmanager import ScreenManager, Screen

from kivy.uix.popup import Popup

from kivy.uix.button import Button

from kivy.clock import Clock

from kivy.properties import StringProperty

from kivy.config import Config

Config.set("graphics", "width", "1024")

Config.set("graphics", "height", "768")

Config.set("graphics", "reszizable", False)

Config.set('input', 'mouse', 'mouse,disable\_multitouch')# Without this red dot appears at right click

"""

Inside box SIZe

1008, 713 within border

0.984, 0.928 ratio

return button

0.0078, 0.9375 ratio

"""

import time

import random

from Generator.Generate import Puzzle

from networking import Client

difficulties = ["easy", "normal", "hard", "extra\_hard"]

class Game:

def \_\_init\_\_(self):

self.difficulty = "normal" # dafault

self.puzzle = None

self.puzzleSolution = None

self.holding\_Number = 0

self.finishTime = 0

self.timerOn = False

self.opponentGrid = None

self.awaitingMatch = False

def import\_Puzzle(self, difficulty):

with open(f"Main/Generator/{difficulty}.txt", "r") as file:

puzzles = file.readlines()

puzzle = random.choice(puzzles)

print(puzzle)

game.puzzle = Puzzle(puzzle)

game.puzzleSolution = Puzzle(puzzle)

game.puzzleSolution.solve()

def parse\_Timer\_to\_String(self, timeFloat):

timeFloat = float(timeFloat)

hours = str(int(timeFloat // 3600))

minutes = str(int(timeFloat % 3600 // 60))

seconds = str(int(timeFloat % 60))

#print(hours, minutes, seconds)

#centiSeconds = str(int(round(self.elapsedTime % 60 - int(self.elapsedTime % 60), 2)\*100))

return ((f"{'0'\*(2-len(hours))+hours}:" if int(hours) > 0 else "")

+

f"{'0'\*(2-len(minutes))+minutes}:{'0'\*(2-len(seconds))+seconds}")

########################

########################

game = Game() ####

########################

########################

class SudokuApp(App):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.online = False

self.rememberLogin = False

self.client = None

self.boot()

self.awaiting\_Match = None

def boot(self):

self.load\_Game\_Data()

if len(self.username) > 0 and len(self.password) > 0:

self.client = Client()

self.client.connect()

if self.client.connected is True:

if self.client.login(self.username, self.password, hashed = True):

self.online = True

self.client.update\_BestTimes(self.topTimes)

else:

print("Disconnected")

self.client.disconnect()

self.client = None

self.rememberLogin = True

def check\_match\_Found(self, dt):

print("[Chekcing math recieve found]")

matchFound = app.client.receive()

if matchFound == "Match Found":

return True

elif matchFound == "Match Not Found":

return False

def load\_Game\_Data(self):

with open("Main/Game Data.txt", "r") as file:

self.username = file.readline().replace("\n", "")

self.password = file.readline().replace("\n", "")

self.topTimes = [file.readline().replace("\n", "") for i in range(4)]

print(self.username)

print(self.password)

print(self.topTimes)

def save\_Game\_Data(self):

with open("Main/Game Data.txt", "w") as file:

if self.rememberLogin is True:

data = f"{self.username}\n{self.password}\n"

else:

data = ("\n\n")

data += "\n".join([time if len(time) > 0 else "\n" for time in self.topTimes])

file.write(data)

def on\_stop(self):

self.save\_Game\_Data()

print("Goodbye World")

class MenuManager(ScreenManager):

pass

class BaseScreen(Screen):

borderFile = StringProperty("graphics/Sudoku\_App\_Border\_Logged\_Out.png")

def \_\_init\_\_(self, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

def on\_enter(self):

self.set\_Border()

def set\_Border(self):

if app.online is True: # Ignore red underline code works

self.borderFile = "graphics/Sudoku\_App\_Border\_Logged\_In.png"

else:

self.borderFile = "graphics/Sudoku\_App\_Border\_Logged\_Out.png"

class MainMenu(BaseScreen):

pass

class Menu(BaseScreen):

pass

class ClassicMenu(Menu):

def setDifficulty(self, difficulty):

game.difficulty = difficulty

class MultiplayerMenu(Menu):

def setDifficulty(self, difficulty):

game.difficulty = difficulty

def match(self):

if app.client and app.online is True:

print("About to match")

if app.client.match\_Players(game.difficulty):

print("matching good")

self.ids.returnButton.disabled = True

game.awaitingMatch = True

matchingPopup = Popup(title = "Matching", content = Label(text = "Waiting for opponent"), size\_hint = (0.7, 0.7))

matchFound = Clock.schedule\_once(app.check\_match\_Found, 0.1)

matchingPopup.open()

if matchFound:

self.ids.returnButton.disabled = False

self.manager.current = "MultiplayerGame"

else:

p = Popup(title = "Unsuccessful", content = Label(text = "No match found, retry"), size\_hint = (0.6, 0.6))

p.open()

self.ids.returnButton.disabled = False

print("Matching not good")

else:

print("Matching not good")

else:

popup = Popup(title = "Error", content = Label(text = "You need to login"), size\_hint = (0.5, 0.5))

popup.open()

class AccountMenu(Menu):

def on\_enter(self):

super().on\_enter()

self.ids.usernameLabel.text = self.get\_Username()

def get\_Username(self):

return app.username

def clickLogout(self):

if app.client and app.online is True:

app.client.disconnect()

app.client = None

app.online = False

self.set\_Border()

app.rememberLogin = False

#############################

#Cell Colours

NEUTRAL = (0, 1, 0.6, 1)

COLLISION = (1, 0, 0.5, 1)

#CLUE = (0, 1, 0.6, 1)

TEXT = (0.76, 0, 1, 1)

#############################

class Cell(Button):

def \_\_init\_\_(self, row, col, n, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.color = TEXT

self.font\_size = 20

self.width = 10

self.height = 10

self.row = row

self.col = col

self.n = n

self.background\_color = NEUTRAL

if self.n > 0:

self.text = str(n)

self.disabled = True

else:

self.text = ""

self.disabled = False

self.clock = Clock.schedule\_interval(self.checkCell, 0.5)

def checkCell(self, dt):

if self.n != 0:

game.puzzle.insert(self.row, self.col, 0)#THis is so that it does not detect a collison with itself

if game.puzzle.check(self.row, self.col, self.n) is False:

self.background\_color = COLLISION

else:

self.background\_color = NEUTRAL

game.puzzle.insert(self.row, self.col, self.n)

def updateCell(self, n):

self.n = n

self.text = str(n)

game.puzzle.insert(self.row, self.col, self.n)

def clearCell(self):

self.n = 0

self.text = ""

game.puzzle.insert(self.row, self.col, 0)

self.background\_color = NEUTRAL

def on\_press(self):

if game.holding\_Number > 0:

self.updateCell(game.holding\_Number)

game.holding\_Number = 0

elif self.last\_touch.button == "right":

self.clearCell()

class NumberInput(Button):

def \_\_init\_\_(self, n, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.width = 5

self.height = 5

self.n = n

self.text = str(n)

def on\_press(self):

game.holding\_Number = self.n

class Timer(Label):

def \_\_init\_\_(self, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

class GameScreen(BaseScreen):

def \_\_init\_\_(self, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

def on\_enter(self):

self.set\_Border()

self.load()

self.clock = Clock.schedule\_interval(self.checks, 0.01)

def load(self):

game.import\_Puzzle(game.difficulty)#assigns puzzle to game.puzzle

game.puzzle.show\_Grid()

#grid = self.ids.grid

i = 0

j = 0

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box1.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 0

j = 3

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box2.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 0

j = 6

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box3.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 1

j = 0

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box4.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 1

j = 3

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box5.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 1

j = 6

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box6.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 2

j = 0

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box7.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 2

j = 3

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box8.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 2

j = 6

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box9.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

"""

for row in range(9):

for col in range(9):

grid.add\_widget(Cell(row, col, game.puzzle.grid[row][col]))

"""

numGrid = self.ids.numberGrid

for n in range(1, 10):

numGrid.add\_widget(NumberInput(n))

self.recentTime = time.time()

self.elapsedTime = 0

game.timerOn = True

def checks(self, dt):

if game.timerOn:

self.updateTimer()

else:

self.recentTime = time.time()

game.win = game.puzzle.grid == game.puzzleSolution.grid

if game.win:#check win

game.puzzleSolution.show\_Grid()

print("Player WINS!")

self.clock.cancel()

game.timerOn = False

game.finishTime = self.saveTime

topTime = app.topTimes[difficulties.index(game.difficulty)]

topTime = float(topTime) if len(topTime) > 0 else 0

newRecord = False

if topTime == 0 or topTime > game.finishTime[0]:

newRecord = True

app.topTimes[difficulties.index(game.difficulty)] = str(game.finishTime[0])

game.win = False

p = Popup(title = "Congratulations",

content = Label(text = f"{'New record!/n' if newRecord else ''}Complete Time: {game.finishTime[1]}"),

size\_hint = (0.6, 0.3))

p.open()

self.manager.current = "MainMenu"

def updateTimer(self):

self.elapsedTime += time.time() - self.recentTime

self.recentTime = time.time()

self.ids.timer.text = game.parse\_Timer\_to\_String(self.elapsedTime)

self.saveTime = [round(self.elapsedTime, 2), self.ids.timer.text]

def on\_leave(self):

self.clock.cancel()

for i in self.ids.box1.children:

i.clock.cancel()

for i in self.ids.box2.children:

i.clock.cancel()

for i in self.ids.box3.children:

i.clock.cancel()

for i in self.ids.box4.children:

i.clock.cancel()

for i in self.ids.box5.children:

i.clock.cancel()

for i in self.ids.box6.children:

i.clock.cancel()

for i in self.ids.box7.children:

i.clock.cancel()

for i in self.ids.box8.children:

i.clock.cancel()

for i in self.ids.box9.children:

i.clock.cancel()

game.puzzle, game.puzzleSolution = None, None

self.ids.box1.clear\_widgets()

self.ids.box2.clear\_widgets()

self.ids.box3.clear\_widgets()

self.ids.box4.clear\_widgets()

self.ids.box5.clear\_widgets()

self.ids.box6.clear\_widgets()

self.ids.box7.clear\_widgets()

self.ids.box8.clear\_widgets()

self.ids.box9.clear\_widgets()

self.ids.numberGrid.clear\_widgets()

class ClassicGame(GameScreen):

def pauseGame(self):

pause = PauseScreen()

pause.open()

class OpponentGridCell(Button):

def \_\_init\_\_(self, cellType, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.width = 3

self.height = 3

self.text = ""

if cellType == "1":

self.background\_color = (1, 1, 0, 1)

else:

self.background\_color = NEUTRAL

def updateCell(self, cellType):

if cellType == "1":

self.background\_color = (1, 1, 0, 1)

else:

self.background\_color = NEUTRAL

class MultiplayerGame(GameScreen):

def \_\_init\_\_(self, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.opponentUsername = ""

def get\_Opponent\_Username(self):

return self.opponentUsername

def on\_enter(self):

self.set\_Border()

#RECEIVE GRID

#initilaize stuff

self.load()

self.clock = Clock.schedule\_interval(self.checks, 0.01)

def checks(self, dt):

#GET OPPONENT GRID SOMEHOW

app.client.send(game.puzzle.grid\_To\_String())

game.opponentGrid = app.client.receive()

if game.opponentGrid == "LOSE":

print("Opponent Wins!")

self.clock.cancel()

game.timerOn = False

p = Popup(title = "Unlucky", content = Label(text = "Opponent Wins!"), size\_hint = (0.4, 0.35))

p.open()

self.manager.current = "MainMenu"

elif game.opponentGrid == "OPPONENT QUIT":

self.clock.cancel()

game.timerOn = False

p = Popup(title = "Unlucky", content = Label(text = "Opponent Quit!"), size\_hint = (0.4, 0.35))

p.open()

self.manager.current = "MainMenu"

elif game.opponentGrid is not None:

for i in range(81):

self.ids.opponentGrid.children[80-i].updateCell(game.opponentGrid[i])

print(game.opponentGrid)

if game.timerOn:

self.updateTimer()

else:

self.recentTime = time.time()

game.win = game.puzzle.grid == game.puzzleSolution.grid

if game.win:#check win

app.client.send("WIN")

self.clock.cancel()

print("Player WINS!")

game.timerOn = False

game.finishTime = self.saveTime

game.win = False

topTime = app.topTimes[difficulties.index(game.difficulty)]

topTime = float(topTime) if len(topTime) > 0 else 0

newRecord = False

if topTime == 0 or topTime > game.finishTime[0]:

newRecord = True

app.topTimes[difficulties.index(game.difficulty)] = str(game.finishTime[0])

p = Popup(title = "Congratulations",

content = Label(text = f"{'New record!/n' if newRecord else ''}You Win/nComplete Time: {game.finishTime[1]}"),

size\_hint = (0.6, 0.3))

p.open()

self.manager.current = "MainMenu"

def updateTimer(self):

self.elapsedTime += time.time() - self.recentTime

self.recentTime = time.time()

self.ids.timer.text = game.parse\_Timer\_to\_String(self.elapsedTime)

self.saveTime = [round(self.elapsedTime, 2), self.ids.timer.text]

def load(self):#########Change this for multiplayer

puzzleString = app.client.receive()

self.opponentUsername = StringProperty(app.client.receive())

game.puzzle = Puzzle(puzzleString)

game.puzzleSolution = Puzzle(puzzleString)

game.puzzleSolution.solve()

game.opponentGrid = "".join([str(0) for i in range(81)]) # Will be revced by server as astring of 1s and 0s, 1s for cells that are complete and 0s for cells that arent

i = 0

j = 0

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box1.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 0

j = 3

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box2.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 0

j = 6

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box3.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 1

j = 0

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box4.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 1

j = 3

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box5.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 1

j = 6

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box6.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 2

j = 0

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box7.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 2

j = 3

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box8.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

i = 2

j = 6

for x in range(3\*i, 3\*i+3):

for y in range(j, j+3):

self.ids.box9.add\_widget(Cell(x, y, game.puzzle.grid[x][y]))

numGrid = self.ids.numberGrid

for n in range(1, 10):

numGrid.add\_widget(NumberInput(n))

opponentGrid = self.ids.opponentGrid

for i in range(81):

opponentGrid.add\_widget(OpponentGridCell(game.opponentGrid[i]))

self.recentTime = time.time()

self.elapsedTime = 0

game.timerOn = True

def clickExit(self):

self.clock.cancel()

game.timerOn = False

app.client.send("QUIT")

self.manager.transition.direction = "down"

self.manager.current = "MultiplayerMenu"

def on\_leave(self):

self.clock.cancel()

for i in self.ids.box1.children:

i.clock.cancel()

for i in self.ids.box2.children:

i.clock.cancel()

for i in self.ids.box3.children:

i.clock.cancel()

for i in self.ids.box4.children:

i.clock.cancel()

for i in self.ids.box5.children:

i.clock.cancel()

for i in self.ids.box6.children:

i.clock.cancel()

for i in self.ids.box7.children:

i.clock.cancel()

for i in self.ids.box8.children:

i.clock.cancel()

for i in self.ids.box9.children:

i.clock.cancel()

game.puzzle, game.puzzleSolution, game.opponentGrid = None, None, None

self.ids.box1.clear\_widgets()

self.ids.box2.clear\_widgets()

self.ids.box3.clear\_widgets()

self.ids.box4.clear\_widgets()

self.ids.box5.clear\_widgets()

self.ids.box6.clear\_widgets()

self.ids.box7.clear\_widgets()

self.ids.box8.clear\_widgets()

self.ids.box9.clear\_widgets()

self.ids.numberGrid.clear\_widgets()

self.ids.opponentGrid.clear\_widgets()

def pauseGame(self):

pass

class PauseScreen(Popup):

def on\_open(self):

game.timerOn = False

def on\_dismiss(self):

game.timerOn = True

class Login(BaseScreen):

def togglePW(self):

self.ids.password.password = not(self.ids.password.password)

def toggleRememberLogin(self):

app.rememberLogin = not(app.rememberLogin)

def clickLogin(self):

print("button clicked")

un, pw = self.ids.username.text, self.ids.password.text

print(un, pw)

if app.online:

app.client.disconnect()

app.client = Client()

app.client.connect()

if app.client.connected:

if app.client.login(un, pw, hashed = False):

app.online = True

self.set\_Border()

if app.rememberLogin is True:

app.username = un

app.password = app.client.hash\_Password(pw)

app.save\_Game\_Data()

app.client.update\_BestTimes(app.topTimes)

else:

app.client.disconnect()

app.client = None

p = Popup(title = "Error", content = Label(text = "Invalid Username or Password"), size\_hint = (0.6, 0.3))

p.open()

else:

app.client = None

p = Popup(title = "Error", content = Label(text = "Could not connect to server, try again otherwise server must be offline"), size\_hint = (0.6, 0.3))

p.open()

class Register(BaseScreen):

def clickRegister(self):

print("Buton Clicked")

un , pw1, pw2 = self.ids.username.text, self.ids.password1.text, self.ids.password2.text

if len(pw1) < 7 or len(pw2) < 7:

p = Popup(title = "Error", content = Label(text = "Password must be at least 7 characters long"), size\_hint = (0.6, 0.3))

p.open()

return

if pw1 != pw2:

p = Popup(title = "Error", content = Label(text = "Passwords do not match"), size\_hint = (0.6, 0.3))

p.open()

return

if app.online:

app.client.disconnect()

app.client = Client()

app.client.connect()

if app.client.connected:

if app.client.register(un, pw1):

loggedIn = app.client.login(un, pw1, False)

if loggedIn:

app.online = True

self.set\_Border()

app.username = un

app.password = app.client.hash\_Password(pw1)

p = Popup(title = "Success", content = Label(text = "Account has been created" + "\nAnd you have been logged in" if loggedIn else ""), size\_hint = (0.6, 0.3))

p.open()

else:

p = Popup(title = "Error", content = Label(text = "Username already taken"), size\_hint = (0.6, 0.3))

p.open()

else:

app.client.disconnect()

app.client = None

p = Popup(title = "Error", content = Label(text = "Could not connect to server"), size\_hint = (0.6, 0.3))

p.open()

class BestTimes(BaseScreen):

def on\_enter(self):

self.ids.easyTime.text = self.get\_Top\_Time(0)

self.ids.normalTime.text = self.get\_Top\_Time(1)

self.ids.hardTime.text = self.get\_Top\_Time(2)

self.ids.extraHardTime.text = self.get\_Top\_Time(3)

def get\_Top\_Time(self, index):

topTime = app.topTimes[index]

return game.parse\_Timer\_to\_String(topTime) if len(topTime) > 0 else "N/A"

class HowToPlayScreen(BaseScreen):

def on\_enter(self):

return super().on\_enter()

#############################

#############################

########################

########################

####

app = SudokuApp() ####

app.run() ####

####

########################

########################

# Appendix 3, Sudoku.kv

#: import WipeTransition kivy.uix.screenmanager.WipeTransition

#: import SwapTransition kivy.uix.screenmanager.SwapTransition

#: import SlideTransition kivy.uix.screenmanager.SlideTransition

#: import FadeTransition kivy.uix.screenmanager.FadeTransition

MenuManager:

MainMenu:

ClassicMenu:

ClassicGame:

MultiplayerMenu:

MultiplayerGame:

AccountMenu:

Login:

Register:

BestTimes:

HowToPlayScreen:

<BaseScreen>:

name : "BaseScreen"

canvas.before:

Color :

rgba : 1, 1, 1, 0.07

Rectangle:

size: self.size

source : None

canvas.after:

Rectangle:

size: self.size

source : root.borderFile

<DefaultButton>:

size\_hint : 0.1, 0.1

pos\_hint : {"center\_x": 0.5, "center\_y":0.5}

text : "Default Button"

on\_release: print("Default Button Pressed")

source : "Main\graphics\DefaultButton.png"

<Menu>:

Button:

name : "ReturnToMainMenu"

text: "Return"

id: returnButton

size\_hint : 0.07, 0.04

pos\_hint : {"x":0.005, "top": 0.99}

on\_release:

root.manager.transition = FadeTransition()

root.manager.current = "MainMenu"

<MainMenu>:

name : "MainMenu"

AnchorLayout:

size\_hint : 0.9, 0.4

pos\_hint : {"center\_x": 0.5, "center\_y":0.65}

BoxLayout:

orientation : "vertical"

BoxLayout:

size\_hint : 1, 0.75

Label:

text: "Sudoku App"

font\_size: 50

color: 0, 1, 0.6, 1

BoxLayout:

size\_hint : 1, 0.25

Button:

text: "Best Times"

font\_size : 30

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.manager.transition = SlideTransition()

root.manager.transition.direction = "right"

root.manager.current = "BestTimes"

Button:

text: "Classic"

font\_size : 30

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.manager.transition = SlideTransition()

root.manager.transition.direction = "right"

root.manager.current = "ClassicMenu"

Button:

text: "Multiplayer"

font\_size : 30

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.manager.transition = SlideTransition()

root.manager.transition.direction = "left"

root.manager.current = "MultiplayerMenu"

Button:

text: "Account"

font\_size : 30

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.manager.transition = SlideTransition()

root.manager.transition.direction = "left"

root.manager.current = "AccountMenu"

BoxLayout:

size\_hint : 0.225, 0.1

pos\_hint : {"center\_x": 0.5, "center\_y":0.25}

Button:

text: "Help"

font\_size : 30

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.manager.transition = SlideTransition()

root.manager.transition.direction = "up"

root.manager.current = "HowToPlayScreen"

<HowToPlayScreen>:

name : "HowToPlayScreen"

Button:

text: "Return"

size\_hint : 0.07, 0.04

pos\_hint : {"x":0.005, "top": 0.99}

on\_release:

root.manager.transition = SlideTransition()

root.manager.transition.direction = "down"

app.root.current = "MainMenu"

AnchorLayout:

size\_hint : 1, 0.96

pos\_hint : {"center\_x": 0.5, "center\_y":0.47}

ScrollView:

size\_hint : 1, 1

do\_scroll\_x : False

do\_scroll\_y : True

Image:

source: "Main\graphics\Sudoku\_App\_Help.png"

size\_hint: None, None

size : self.texture\_size

<GameScreen>:

name : "GameScreen"

canvas.before:

#Color :

#rgba : 1, 1, 1, 0.07

Rectangle:

size: self.size

source : "Main\graphics\Sudoku\_App\_Background.png"

Button:

id : timer

text: "00:00"

background\_color : 0, 1, 0.5, 1

size\_hint : 0.1, 0.07

pos\_hint : {"center\_x" : 0.5, "top": 0.99}

on\_release: root.pauseGame()

AnchorLayout:

anchor\_x: "center"

anchor\_y: "center"

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : "600dp"

height : "600dp"

id : grid

cols: 3

rows: 3

spacing : 8

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box1

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box2

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box3

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box4

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box5

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box6

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box7

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box8

cols: 3

rows: 3

spacing : 2

GridLayout:

size\_hint\_x : None

size\_hint\_y : None

width : 200

height : 200

id : box9

cols: 3

rows: 3

spacing : 2

GridLayout:

width: 175

height: 175

size\_hint\_x : None

size\_hint\_y : None

pos\_hint : {"center\_x": 0.9, "center\_y":0.5}

id : numberGrid

cols : 3

rows : 3

spacing : 0.5

<PauseScreen>:

title : "Pause"

size\_hint : 0.4, 0.6

Button:

text: "Resume"

on\_release: root.dismiss()

<ClassicMenu>:

name : "ClassicMenu"

AnchorLayout:

size\_hint : 0.9, 0.1

pos\_hint : {"center\_x": 0.5, "center\_y":0.5}

BoxLayout:

Button:

text: "Easy"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.setDifficulty("easy")

root.manager.transition = FadeTransition()

root.manager.current = "ClassicGame"

root.manager.transition.direction = "right"

Button:

text: "Normal"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.setDifficulty("normal")

root.manager.transition = FadeTransition()

root.manager.current = "ClassicGame"

root.manager.transition.direction = "right"

Button:

text: "Hard"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.setDifficulty("hard")

root.manager.transition = FadeTransition()

root.manager.current = "ClassicGame"

root.manager.transition.direction = "right"

Button:

text: "Extra Hard"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.setDifficulty("extra\_hard")

root.manager.transition = FadeTransition()

root.manager.current = "ClassicGame"

root.manager.transition.direction = "right"

<MultiplayerMenu>:

name : "MultiplayerMenu"

AnchorLayout:

size\_hint : 0.9, 0.3

pos\_hint : {"center\_x": 0.5, "center\_y":0.5}

BoxLayout:

orientation : "vertical"

BoxLayout:

ToggleButton:

text: "Easy"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

group : "difficulty"

state : "down"

on\_release:

root.setDifficulty("easy")

ToggleButton:

text: "Normal"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

group : "difficulty"

on\_release:

root.setDifficulty("normal")

ToggleButton:

text: "Hard"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

group : "difficulty"

on\_release:

root.setDifficulty("hard")

ToggleButton:

text: "Extra Hard"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

group : "difficulty"

on\_release:

root.setDifficulty("extra\_hard")

Label:

text : ""

Button:

text: "Match"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

id : matchButton

size\_hint : 0.5, 1

pos\_hint : {"center\_x": 0.5}

on\_release: root.match()

<AccountMenu>:

name : "AccountMenu"

AnchorLayout:

size\_hint : 0.9, 0.3

pos\_hint : {"center\_x": 0.5, "center\_y":0.6}

BoxLayout:

orientation : "vertical"

Label:

id : usernameLabel

text : root.get\_Username()

font\_size : 25

Label:

text:""

BoxLayout:

Button:

text: "Login"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.manager.current = "Login"

Button:

text: "Register"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release:

root.manager.current = "Register"

Button:

text: "Logout"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

on\_release: root.clickLogout()

<BestTimes>:

name : "BestTimes"

Button:

text: "Return"

size\_hint : 0.07, 0.04

pos\_hint : {"x":0.005, "top": 0.99}

on\_release:

root.manager.transition = FadeTransition()

app.root.current = "MainMenu"

AnchorLayout:

size\_hint : 0.8, 0.5

pos\_hint : {"center\_x": 0.5, "center\_y":0.5}

#orientation : "horizontal"

BoxLayout:

background\_color : 0, 0, 0, 0.5

spacing : 5

BoxLayout:

color: 1, 0, 0, 1

orientation : "vertical"

Label:

text: "Easy"

Label:

id : easyTime

text : ""

BoxLayout:

orientation : "vertical"

Label:

text: "Normal"

Label:

id : normalTime

text : ""

BoxLayout:

orientation : "vertical"

Label:

text: "Hard"

Label:

id : hardTime

text : ""

BoxLayout:

orientation : "vertical"

Label:

text: "Extra Hard"

Label:

id : extraHardTime

text : ""

<ClassicGame>:

name : "ClassicGame"

Button:

text: "Exit Game"

size\_hint : 0.08, 0.04

pos\_hint : {"x":0.005, "top": 0.99}

on\_release:

root.manager.transition.direction = "down"

root.manager.current = "ClassicMenu"

<MultiplayerGame>:

name : "MultiplayerGame"

Button:

id : exitButton

text: "Exit Game"

size\_hint : 0.08, 0.04

pos\_hint : {"x":0.005, "top": 0.99}

on\_release:

root.clickExit()

Label:

id: opponentLabel

text : root.get\_Opponent\_Username()

size\_hint : 0.3, 0.3

pos\_hint : {"x":0.1, "top": 0.5}

GridLayout:

id : opponentGrid

width: 175

height: 175

size\_hint\_x : None

size\_hint\_y : None

pos\_hint : {"center\_x": 0.9, "center\_y":0.77}

cols : 9

rows : 9

spacing : 0.1

<Login>:

name : "Login"

Button:

text: "Return"

size\_hint : 0.07, 0.04

pos\_hint : {"x":0.005, "top": 0.99}

on\_release:

root.manager.transition = FadeTransition()

app.root.current = "AccountMenu"

AnchorLayout:

size\_hint : 0.6, 0.4

pos\_hint : {"center\_x": 0.5, "center\_y":0.45}

BoxLayout:

orientation : "vertical"

BoxLayout:

Label:

size\_hint : 0.25, 1

text : "Username"

color : 0, 1, 1, 1

TextInput:

size\_hint : 0.75, 1

id: username

hint\_text : "username"

multiline : False

BoxLayout:

Label:

size\_hint : 0.25, 1

text : "Password"

color : 0, 1, 1, 1

TextInput:

size\_hint : 0.5, 1

id : password

hint\_text : "password"

multiline : False

password : True

ToggleButton:

size\_hint : 0.25, 1

text : "toggle mask"

on\_press: root.togglePW()

Label:

text : ""

Button:

text:"Login"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

pos\_hint: {"center\_x" : 0.5}

size\_hint : 0.3, 0.8

on\_release : root.clickLogin()

Label:

text : ""

Label:

text : ""

ToggleButton:

text:"Remember Login"

font\_size : 20

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

pos\_hint: {"center\_x" : 0.5}

size\_hint : 0.3, 0.8

on\_release : root.toggleRememberLogin()

<Register>:

name : "Register"

Button:

text: "Return"

size\_hint : 0.07, 0.04

pos\_hint : {"x":0.005, "top": 0.99}

on\_release:

root.manager.transition = FadeTransition()

app.root.current = "AccountMenu"

AnchorLayout:

size\_hint : 0.6, 0.3

pos\_hint : {"center\_x": 0.5, "center\_y":0.5}

BoxLayout:

orientation : "vertical"

BoxLayout:

Label:

size\_hint : 0.25, 1

text : "Username"

color : 0, 1, 1, 1

TextInput:

size\_hint : 0.75, 1

id: username

hint\_text : "username"

multiline : False

BoxLayout:

Label:

size\_hint : 0.25, 1

text : "Password"

color : 0, 1, 1, 1

TextInput:

size\_hint : 0.75, 1

id : password1

hint\_text : "password"

multiline : False

BoxLayout:

Label:

size\_hint : 0.25, 1

text : "Re-Enter Password"

color : 0, 1, 1, 1

TextInput:

size\_hint : 0.75, 1

id : password2

hint\_text : "password"

multiline : False

Label:

text : ""

Button:

text:"Create Account"

font\_size : 25

color : 0.38, 0.13, 0.97, 1

background\_normal : "Main\graphics\Sudoku\_App\_Deafult\_Button.png"

pos\_hint: {"center\_x" : 0.5}

size\_hint : 0.3, 0.8

on\_release : root.clickRegister()

# 

# Appendix 4,networking.py

import socket

#import hashlib

#Make connection with server

#Select if you want to play against a random

#get added to queue of players

#get put into a lobby

#play against player win lose

#continue

#if dont continue exit mulitplayer

#close connection

def gaviHash(data):

hash\_value = 0

x = len(data)

for i, char in enumerate(data):

# Mix operations: XOR, multiplication, bit-shifting

hash\_value ^= (ord(char) \* (x\*\*i))

hash\_value ^= (ord(char) << ((x-i) % 3)) \*\* ((x-i))

return hex(hash\_value)[2:]

class Client:

def \_\_init\_\_(self):

self.\_\_host = "127.0.0.1"

self.\_\_port = 7777

self.\_\_client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # TCP connection

self.connected = False

self.username = ""

def connect(self):

try:

self.\_\_client.connect((self.\_\_host, self.\_\_port))

self.connected = True

print("[Connection with server established]")

except:

print("[Connection could not be made]")

return False

def disconnect(self):

self.\_\_client.send("Logout".encode())

self.\_\_client.close()

self.connected = False

print("[Connection with server closed]")

def send(self, message):

self.\_\_client.send(message.encode())

def receive(self, n = 1024):# n = bytes to receive

return self.\_\_client.recv(n).decode()

def hash\_Password(self, password):

return gaviHash(password)

def register(self, username, password):#Password going through here is here unproccessed

print("[Registering from client]")

self.\_\_client.send("register".encode())

proceed = self.\_\_client.recv(1024).decode()

valid = False

if proceed == "proceed":

self.\_\_client.send((username+","+self.hash\_Password(password)).encode())

if self.\_\_client.recv(1024).decode() == "valid":

valid = True

else:

print("No proceed")

return valid

def login(self, username, password, hashed : bool):

print("[Login from client]")

self.\_\_client.send("login".encode())

proceed = self.\_\_client.recv(1024).decode()

valid = False

if proceed == "proceed":

if hashed is True:

self.\_\_client.send((username+","+password).encode())

else:

self.\_\_client.send((username+","+self.hash\_Password(password)).encode())

if self.\_\_client.recv(1024).decode() == "valid":

valid = True

self.username = username

print("Login success")

return valid

def update\_BestTimes(self, times):

print("[Updating best times from client]")

self.\_\_client.send("update\_BestTimes".encode())

proceed = self.\_\_client.recv(1024).decode()

if proceed == "proceed":

self.\_\_client.send((f"{self.username}," + ",".join(times)).encode())

if self.\_\_client.recv(1024).decode() == "valid":

return True

else:

return False

def match\_Players(self, difficulty):

print("[Entering player for matchmaking from client]")

self.\_\_client.send("match\_Players".encode())

proceed = self.\_\_client.recv(1024).decode()

if proceed == "proceed":

print("proceed = true sending difficulty")

self.\_\_client.send(difficulty.encode())

message = self.\_\_client.recv(1024).decode()

print(message)

if message == "Enqueued":

return True

elif message == "Queue full":

return False

if \_\_name\_\_ == "\_\_main\_\_":

while True:

print(gaviHash(input(">")))

# Appendix 5, server.py

import socket

import threading

import sqlite3

import random

import time

from Generator.Generate import Puzzle

class Queue():

def \_\_init\_\_(self, maxSize=10):

self.\_\_queue = [None for i in range(maxSize)]

self.\_\_front = 0

self.\_\_rear = -1

self.\_\_size = 0

self.\_\_maxSize = maxSize

self.lock = threading.Lock()

def isFull(self):

return self.\_\_size == self.\_\_maxSize

def isEmpty(self):

return self.\_\_size == 0

def isEven(self):

#with self.lock:

return self.\_\_size % 2 == 0

def enQueue(self, item):

#with self.lock:

if self.isFull():

return item

else:

self.\_\_rear = (self.\_\_rear + 1) % self.\_\_maxSize

self.\_\_queue[self.\_\_rear] = item

self.\_\_size += 1

return True

#print(self.\_\_front, self.\_\_rear)

def deQueue(self):

#with self.lock:

if self.isEmpty():

return False

else:

data = self.\_\_queue[self.\_\_front]

self.\_\_front = (self.\_\_front + 1) % self.\_\_maxSize

self.\_\_size -= 1

return data

#print(self.\_\_front, self.\_\_rear)

def show(self):

return [self.\_\_queue[(self.\_\_front + i) % self.\_\_maxSize]for i in range(self.\_\_size)]

######################################################################

class Client(threading.Thread):

def \_\_init\_\_(self, client, address):

super().\_\_init\_\_()

self.client = client

self.address = address

self.username = None

self.bestTimes = [None, None, None, None]

self.options = { "login": self.login,

"register": self.register,

"match\_Players": self.match\_Players,

"update\_BestTimes" : self.update\_BestTimes}

self.difficulty = None

self.matching = False

self.startMatchingTime = 0

self.inGame = False

def run(self):

while self.client:

try:

if self.inGame:

continue

elif self.matching == True:

if time.time() - self.startMatchingTime > 300:#After 5 minutes removes player from match queue

self.matching = False

self.client.send("Match Not Found".encode())

print(f"[Matchmaking timed out for {self.username} from {self.address}]")

else:

print(f"[Waiting for request from {self.username}, {self.address}]")

request = self.client.recv(1024).decode()

print(f"[Request accepted: {request}]")

if request == "Logout":

print(f"[Logged out has {self.username} from {self.address}]")

break

elif request in self.options:

self.options[request]()

except Exception as e :

print(f"[Inavlid request {e}]")

break

print(f"[Closing connection {self.username} from {self.address}]")

self.client.close()

def check(self, username, cursor):

result = cursor.execute("SELECT Username FROM Accounts WHERE Username = ?", (username,)).fetchone()

if result is None:

return False

elif username in result:

return True

def verify(self, username, password, cursor):#Checks if username an dpassword match

if self.check(username, cursor):

result = cursor.execute("SELECT Username, Password FROM Accounts WHERE Username = ? AND Password = ?", (username, password)).fetchone()

if result is None:

return False

else:

usernameResult, passwordResult = result

if usernameResult == username and passwordResult == password:

return True

else:

return False

else:

return False

#######################################################################

#Request Redirections

#######################################################################

def register(self):

print(f"[Register request from {self.address}]")

self.client.send("proceed".encode())

details = self.client.recv(1024).decode().split(",")

conn = sqlite3.connect(DATABASE, check\_same\_thread=False)

cursor = conn.cursor()

if self.check(details[0], cursor):

self.client.send("invalid".encode())#USername already exists

conn.close()

print(f"[Account not created, username already taken from {self.address}]")

return False

else:

cursor.execute("INSERT INTO Accounts (Username, Password) VALUES (?, ?)", (details)) # Tuple unpacking if not obvious

cursor.execute("INSERT INTO BestTimes (Username, Easy, Normal, Hard, 'Extra Hard') VALUES (?, NULL, NULL, NULL, NULL)", (details[0],)) # Tuple unpacking if not obvious

self.client.send("valid".encode())

conn.commit()

conn.close()

print(f"[Account created with username: {details[0]} from {self.address}]")

return True

def login(self):

print(f"[Login request from {self.address}]")

self.client.send("proceed".encode())

details = self.client.recv(1024).decode().split(",")

conn = sqlite3.connect(DATABASE)

cursor = conn.cursor()

if self.verify(details[0], details[1], cursor):

print(f"[Successful login by {details[0]} from {self.address}]")

self.client.send("valid".encode())

conn.close()

return True

else:

self.client.send("invalid".encode())

print(f"[Unsuccessful login by {details[0]} from {self.address}]")

conn.close()

return False

def update\_BestTimes(self):

print(f"[Update best times request from {self.address}]")

self.client.send("proceed".encode())

times = self.client.recv(1024).decode().split(",")

conn = sqlite3.connect(DATABASE, check\_same\_thread=False)

cursor = conn.cursor()

try:

cursor.execute("UPDATE BestTimes SET Easy = ?, Normal = ?, Hard = ?, 'Extra Hard' = ? WHERE Username = ?",

(times[1], times[2], times[3], times[4], times[0]))

self.client.send("valid".encode())

print(f"[Succesfully updated best times from {self.address}]")

except Exception as e:

print(f"Error updating best times {e} from {self.address}")

self.client.send("invalid".encode())

self.username = times[0]

self.bestTimes = times[1:]

conn.commit()

conn.close()

def match\_Players(self):

print(f"[Matching request from {self.address}]")

self.client.send("proceed".encode())

self.difficulty = self.client.recv(1024).decode()

if queueDict[self.difficulty].isEmpty():

pass

#client.send("Queue empty".encode())

#prompt use that queue is empty so they mayhave to wait a while

if queueDict[self.difficulty].enQueue(self):

print(f"[Enqueued into {self.difficulty} queue {self.username} from {self.address}]")

self.client.send("Enqueued".encode())

self.matching = True

self.startMatchingTime = time.time()

#wait

else:

print(f"[{self.difficulty} queue is full for {self.username} from {self.address}]")

self.client.send("Queue full".encode())

#return False

#######################################################################

class Game(threading.Thread):

def \_\_init\_\_(self, thread1, thread2, difficulty):

super().\_\_init\_\_()

self.player1 = thread1

self.player2 = thread2

self.puzzleString = self.import\_Puzzle(difficulty)

p = Puzzle(self.puzzleString)

p.show\_Grid()

p.solve()

self.solutionString = p.grid\_To\_String()

def run(self):

finished = False

self.player1.client.send("Match Found".encode())

self.player2.client.send("Match Found".encode())

self.player1.client.send(self.puzzleString.encode())

self.player2.client.send(self.puzzleString.encode())

self.player1.client.send(self.player2.username.encode())

self.player2.client.send(self.player1.username.encode())

while finished is False:

p1Grid = self.player1.client.recv(1024).decode()

p2Grid = self.player2.client.recv(1024).decode()

print(f"Player1's grid: [{p1Grid}]")

print(f"Player2's grid: [{p2Grid}]")

print()

if p1Grid == "WIN":

self.player2.client.send("LOSE".encode())

finished = True

self.player1.inGame = False

self.player2.inGame = False

elif p2Grid == "WIN":

self.player1.client.send("LOSE".encode())

finished = True

self.player1.inGame = False

self.player2.inGame = False

elif p1Grid == "QUIT":

self.player2.client.send("OPPONENT QUIT".encode())

finished = True

self.player1.inGame = False

self.player2.inGame = False

elif p2Grid == "QUIT":

self.player1.client.send("OPPONENT QUIT".encode())

finished = True

self.player1.inGame = False

self.player2.inGame = False

else:

self.player1.client.send(self.compare\_Puzzles(p2Grid).encode())

self.player2.client.send(self.compare\_Puzzles(p1Grid).encode())

def import\_Puzzle(self, difficulty):

with open(f"Main\Generator\{difficulty}.txt", "r") as file:

puzzles = file.readlines()

return random.choice(puzzles)

def compare\_Puzzles(self, puzzleString):

#Makes a string of 1s and 0s, 1s showing correctly filled cells and 0s showing empty and incorrect cells

return "".join(["1" if puzzleString[i] == self.solutionString[i] else "0" for i in range(81)])

####################################################################################################################################

def create\_Match():

while True:

time.sleep(1)

for difficulty in queueDict:

if not queueDict[difficulty].isEmpty() and queueDict[difficulty].isEven():

player1 = queueDict[difficulty].deQueue()

player2 = queueDict[difficulty].deQueue()

player1.matching, player2.matching = False, False

player1.inGame, player2.inGame = True, True

Game(player1, player2, difficulty).start()

print(f"[Have matched some players {player1.username} from {player1.address} and {player2.username} from {player2.address}]")

###########################################################################################################################################

######################################################################################

################\_\_\_INITIALISATION\_\_\_##################################################

host = "127.0.0.1" # IP address of the server

port = 7777 # Port number

DATABASE = "Server\Sudoku\_Online.db"

easyQ = Queue(25)

normalQ = Queue(25)

hardQ = Queue(25)

extraHardQ = Queue(25)

queueDict = {"easy": easyQ, "normal": normalQ, "hard": hardQ, "extraHard": extraHardQ}

#######################################################################################

def main():

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # TCP socket

server.bind((host, port))

server.listen() # Ready to accept connections

threading.Thread(target = create\_Match).start() # Will create macthes indefinitely

while True:

print("ACCEPTING NEW CONNECTIONS")

client, address = server.accept()

print(f"CONNECTION ACCEPTED FROM {address}")

Client(client, address).start() # puts accepted client into their own thread

if \_\_name\_\_ == "\_\_main\_\_":

print("\_\_SERVER STARTED\_\_")

main()

1. Image from <https://www.britannica.com/story/will-we-ever-run-out-of-sudoku-puzzles> [↑](#footnote-ref-0)