Sudoku Solver

Colin Carruzzo
University of Lausanne
colin.carruzzo@unil.ch

Maxime Zahler
University of Lausanne
maxime.zahler@unil.ch

Arnaud Castellana
University of Lausanne
arnaud.castellana@unil.ch

***Abstract*—Sudoku games have become one of the most famous logic games in the world, but people often judge them as being too complicated, often making them lose interest in solving them until the end. This paper will present an application enabling people to solve Sudoku games more easily by guiding them with assistance along their solving of the game. We will first discuss the algorithm used to solve a Sudoku game, then, we will concentrate on the description of the application which relies on a graphical user interface.**

***Index Terms*—Sudoku, grid, matrix, cell, graphical user interface, application**

I. Introduction

The Sudoku game found its origin in Switzerland, when in 1783 the mathematician Leonhard Euler developed Latin Squares, in which the digits were only allowed to appear once in each row and column of a grid block, describing the squares as "a new kind of magic squares". The grid invented by Leonhard Euler is similar to the nowadays traditional Sudoku games which we all know, expect that the modern Sudoku games contain nine 3x3 subdivisions. The principle, however, has not seen many changes, since having the same number or symbol in a row or column is not allowed.

It is in New York, during the late 1970s, that the logic game found all its glory and became famous worldwide. The need to fill these grids spread across cities and countries and people started to develop their love and interest for this game. The Japanese then took over the game and presented it under the now well-known name of Sudoku. It does not require advanced mathematical knowledge to solve a Sudoku game and even though some grids require more reflection time, most of them can be finished in a reasonable amount of time, which explains the success of this game since people from different backgrounds can start solving Sudoku games and improve their skill level with time. This is the reason, why we would like to present an application that will help the beginners, the intermediate and the already more experienced players to take their skills to the next level, in order to solve any type of Sudoku grid they can find, be it in a newspaper, online or in Sudoku specialised books.

Before continuing, let us quickly remind the basic rules and principles of the Sudoku game: We are given a 9x9 grid consisting of nine 3x3 squares leading to a total of eighty-one cells. The goal of the Sudoku game is quite simple: we have to fill in every cell with a number from 1 to 9, with the following constraints:
- Each row should contain the numbers 1 to 9
- Each column should contain the numbers 1 to 9
- Each 3x3 sub-grid should contain the numbers 1 to 9

II. Research question

A. Problem

Even though it is not complicated to find Sudoku grids nowadays, some players are still confronted to some difficulties during their process of solving the game and all they can find is the entire correction of the Sudoku game. This clearly is not adequate since it removes their pleasure of solving the game themselves and thus not improving their skills.

To complete the application, we are first confronted to the obvious problem of solving the grid, which we want to solve as quickly as possible.

We also need to make it possible and practical for the users to use the application with a Sudoku potentially not provided in a digital format. For this, we will need a graphical user interface which intends to make it easy and understandable for the user to input the Sudoku needing to be solved and then to solve it with the application.

B. Objective

We want to provide an application based on an interactive graphical user interface that can react to user inputs and give the user hints when they encounter difficulties. The aim is to keep the users interested in solving the actual Sudoku game even when they reach a stage, in which they think they cannot solve it any further because they think it is too difficult for them.
We also want to give the user the opportunity to check his inputed values whenever he wants. This will be especially helpful for the beginners because they will be able to solve a Sudoku grid step by step and to make sure they are not going in the wrong direction.

Python is the best way to in order to fully solve the problem, since it gives us all the necessary tools to create the needed application. Two modules are going to be of use for us: Numpy and Pygame. We therefore recommend having a minimal amount of knowledge in both of these libraries

in order to be able to follow the code being presented. All necessary information is available online on Numpy.org and Pygame.org.

C. Scope

The first objective of our project is to obtain an efficient algorithm which is able to solve any Sudoku. The algorithm will be composed of four different functions, each being necessary in order to obtain the required results.

The second objective is the design of a user-friendly application. The possibilities offered by Pygame are too many to count, and we will therefore concentrate on having a straightforward path for the users of the application.

III. Methodology and algorithm applied

Our code can be found in a folder, named SudokuHelper. It is composed of a total of eight different .py files, each of them containing a part of the overall codebase. We are going to reference these files in this paper. You can also find another folder named Police in the SudokuHelper folder, in which the fonts used in the graphical user interface can be found, all of them free of rights. The last folder, called Pycache, is the ine containing specific files that are automatically created once the Pygame module is engaged.

We first present the solving algorithm and then the implementation of the application.
The main principle used in this context is the idea of backtracking solving, which is based on the method of recursions. In contrary to a person who would solve the Sudoku by looking for the easiest cell to fill and then proceed to the next easiest cell, a backtracking algorithm can solve the Sudoku cell by cell, going from the top left to the bottom right. Important to note is that even if this procedure is completely valid, it is not the most efficient way to solve the Sudoku grid.

As our aim is to implement an algorithm that solves the Sudoku game in the fastest possible way, we thought about different possible techniques. We came to the conclusion that the fastest way to solve the Sudoku grid is to follow the human intuition.

This means that our algorithm does not only need to be able to tell if a number is valid in a specific cell and to tell wether a grid is finished or not. We require different functions to be able to manage the solving of the Sudoku grid in the best possible way. Let us now see what needs to be done in the correct order: We first need to find the cell with the smallest amount of possibilities and we then need to retrieve its coordinates. Since our program was coded in Python, we have to be cautious that the indexing system of the program starts in 0. It is therefore important to notice that the coordinate of the top left cell is (0,0), and not (1,1). Once we have determined the coordinate, we would have the possibility to test each value from 1 to 9 in the increasing order, but as already mentioned before, this would not be the most efficient way. For a specific cell, we only want to try the possible values and not all the values from 1 to 9. We therefore need a function that has the aim of not only retrieving the coordinates of the cell with the least amount of possibilities that would fit in the Sudoku, but this function should also be able to determine the possible values that could fit in this cell. The procedure described above corresponds to one of the four functions that are going to be introduced in this part needed to solve the Sudoku grid. In order to find the corresponding cell we first need to determine the number of possibilities for each cell in the Sudoku grid. This task is done by another function. Another important aspect needed in the solving of the Sudoku grid is the possibility to tell when a grid is finished, when the completely filled grid is impossible, meaning that one or more cells have been wrongly placed and when a grid completion is on the right track, meaning that a grid has not completely been filled but that no mistakes have appeared in the placed cells yet. We yet need another function for this task. We then introduce the last of the four functions, which will forge the bridge between our three functions described above. It also is the function which will engage the recursion being the game changer in our procedure of solving the Sudoku. The four mentioned functions above are called GetPoss(), GetChoice(), CheckSolved() and SolveSudok(). Find below a flowchart describing the connection between the four functions aimed at solving the Sudoku grid.

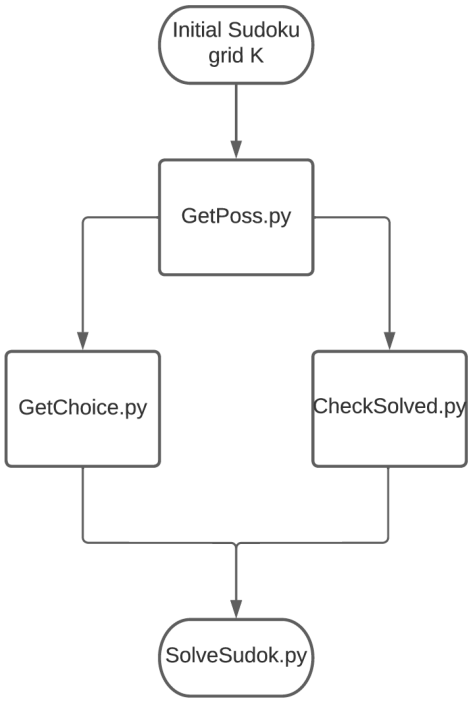

Fig. 1. Graphical representation of the solving algorithm

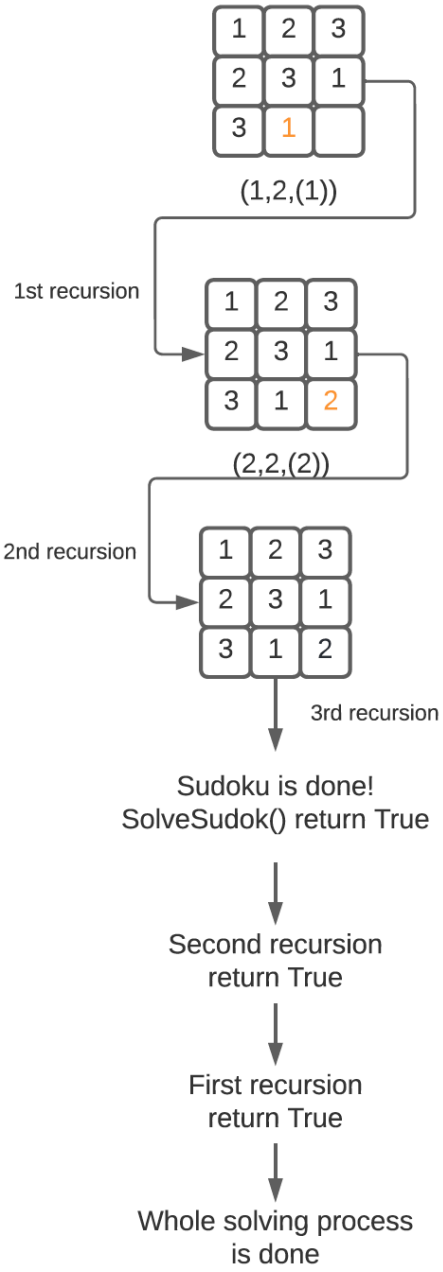

Fig. 2. Flowchart of the recursion process

The first function we describe is GetPoss(). The function takes an initial Sudoku grid as an input. By initial Sudoku grid, we mean a grid that is filled with some cells, but that is not completely finished. The output will be helpful in order to find the cell with the smallest amount of possibilities, as it is a 9x9x9 matrix. There exist nine layers where each layer consists in a Sudoku grid for a particular input number from one to nine. Each matrix is a boolean matrix, with zeros meaning a certain value cannot be placed in this cell and 1 meaning a certain number can indeed be placed in this cell. Let us take an example with the third layer being the one for the number three. If in a cell of this matrix we find a 1, it means that the number can indeed be place in this cell. Let us call this 9x9x9 matrix the matrix M. Having defined the matrix M, we can now engage in the next function, which is as already mentioned above the function GetChoice(). This function reads the matrix M and sums over each of its layers leading to a 9x9 matrix. The number in each cell represents the number of possibilities that can be inserted in this particular cell in our initial Sudoku grid. Notice that this is what we initially were looking for, but we still need to find these possible values. In order to do so, we check in M whether the values are possible and if this is the case we add them to a list called S. We also need to retrieve the coordinate of the cell we initially wanted to fill. This can be done by summing over the matrix M, which indicates us the cell with the smallest amount of possibilities. We then end up with the total output of GetChoice(), which consists in a list composed of the coordinate of the cell with the smallest amount of possibilities and the list S itself, as seen in Figure 2. Next, we define the function CheckSolved(), which is quite simple in its composition. This function is used in order to check the state of a Sudoku grid, so checking whether it is completely finished, still in progress or if we have achieved a stage in which there is an error, meaning that we have to go back a certain amount of steps and start again from there. Let us first think about how do we know a Sudoku grid is finished. We could simply check that all the cells are not empty, but this would not be adapted for the case in which some cells would be filled with the wrong number, meaning that we need another way of checking the above condition. We decided to opt for a quite simple and elegant method which consists in counting the sum of all the entered values. When this sum attains the number of 405, this means that the Sudoku grid is finished. In order to obtain the number 405, we simply solve the following calculation:

$$9\sum_{i=1}^{9} i = 405$$

which can be solved by applying the fundamental theorem of summation.

We now have all the required elements to solve the initial Sudoku grid and can use all these outputs of the before mentioned functions in the final function named SolveSudok(). Let us start by filling in the first cell, which is going to be the first recursion in our solving. By using the function CheckSolved(), we first check whether or not the Sudoku of the grid has been solved. We also need to find the wanted cell via the function GetChoice(), i.e. the cell with the smallest amount of possibilities. Given that we now know the cell that has to be filled and also know the different possible values that can be inserted in this cell, we can start looping over these possibilities. After having tried the first possible values, we call the recursion, i.e we once again call the function SolveSudok(), but this time with the updated grid. This process will go on for

every cell until CheckSolved() returns the desired result: the completed Sudoku grid. This process will go on for every cells until CheckSolved() returns that the Sudoku is completed. In order to dive a bit deeper in the recursion process, let us look at Figure 2. This corresponds to the simplest case in which each cell has only one possible value. Let us also mention, that in order to simplify the understanding of the graphical representation, we just represented one of the nine smaller grids in the 9x9 matrix. In this case we only have to pay attention to the rule of not having the same number appear in the same line or in the same column.

With an initial grid containing only two cells to be filled and each of these cells only having one possible value that fits in it, we first fill the most upper left cell, which is given by the GetChoice() function. Once the cell has been filled, we call the recursion and go on to the second grid, in which we fill the last empty cell. The recursion is then called once again, but this time immediately returning the boolean True since CheckSolved() returns that the grid is solved. Now that the last recursion is completed, the recursion before the last one can attain its end and also return the boolean True, enabling the first one to also end and therefore ending the whole process which leads to having the Sudoku grid solved with all the cells filled with the correct values.

```
1  def SolveSudok(K):
2      '''
3      Main solving function that
4      will use the others function
5      to solve a grid K
6      '''
7      global recurs
8      fill = GetChoice(K)
9
10      if CheckSolved(K)==1:
11          recurs = 0
12          return True
13
14      for f in range(0,len(fill[2])):
15          if CheckSolved(K) == 0:
16              K[fill[0]][fill[1]] = fill[2][f]
17
18              if SolveSudok(K):
19                  recurs += 1
20                  return (True, K, recurs)
21          K[fill[0]][fill[1]] = 0
22      return False
```

We really want to point out, that each recursion is only ended when the next recursion returns the boolean True (see line 18 in code above). That is why we need to reach the end of the solving in order to get back to the first recursion. It is indeed only the last recursion that will return directly the boolean True in the CheckSolved part (see line 12 in code above). Now that the basic example is clear, let us see what happens when a cell has more than one possible value. The algorithm will try the first possible value and then call the next recursion. This recursion will do the same thing, until either the end, or until a stage is reached, in which a cell has no more possible values. This will create a loop from the values zero to zero which means that it will not do anything here. The said recursion will then return the boolean False, and it will then make the previous recursion try the next possible value.

To summarise this part on recursion, we observe that the recursion process, although complicated to understand at first, allows us to end up with a short code, that performs well and leads to a solved Sudoku. We also added a counter counting the number of recursions in the algorithm, which corresponds to the number of cells that are filled. This will give us an interesting statistic about how many cells a user filled for the summary part in the GUI.py file.

Now that we know how to solve our initial Sudoku by using our function SolveSudok(), we want to create an interface that will help the user. The aim of the graphical user interface is to make our code more user-friendly and easier to understand for someone who is not familiar with coding. Indeed, instead of giving a (9,9) np.array which is long to modify and painful to read, the graphical user interface allows the user to implement a Sudoku in a smooth way and to solve it in a graphical way. To implement this graphical user interface we immediately thought about the python library called Pygame. Indeed the graphical user interface can be seen as a video game that the user will use in order to help him solve the most difficult Sudoku they will face. Pygame is a free cross-platform library that facilitates the development of real-time video games with the Python programming language. This library is ideal for our purpose.
After having installed pygame with the command:

```
pip install pygame
```

we have to import it and initialise it with the command pygame.init(). This will allow us to initialise all the imported modules that are going to be needed. We could initialise them one by one but this command is a convenient way to get everything started and ready to work. The idea of Pygame is that we can set up the drawing window in which our graphical user interface will evolve. For this, Pygame has a useful built-in function called set_mode(), in which we define the desired width and height of the screen. The user then needs to define a “running” variable as a boolean True. This variable will literally be the running variable of our program. As long as this variable is equal to True, the game will continue and the screen will continue to be displayed. Since the running variable in our while loop is a boolean True we can display our main menu screen, defined by a function discussed later. The idea of Pygame is to draw and display a static screen at a given point in time in our loop. Then if you want to change something, like for example go to another menu, you have to create a condition (press on the keyboard or click on a button as an outline). If this condition is fulfilled you can call a new menu, defined in a total different function that will display a different text and other layout on the screen. The conditions in Pygame are handled by the pygame.event family. From this family, the module pygame.event.get() is the most used in the code. Used

in an infinite loop, it will basically tell Pygame that we want it to check if one type of event in its library has been called. The type of event, declared with event.type could be, as in our code, the event where the user pressed on the backspace button on their keyboard. Since Pygame is displaying a static screen we have to tell Pygame to flip the display, meaning to update the screen, at the end of the running While loop. This can be done using the Pygame module pygame.display.flip(). More generally the pygame.display module family is used to control everything linked to display, update window and screen. To stop the game, thus to close the window and stop Pygame, we need to specify it in an event (usually this event is clicking the Windows close button). As we have seen, the strength of Pygame is that it allows its users, through built in functions directly integrated in the library, to draw, display and have a limitless interaction with the screen. Another very used module in the code is pygame.mouse.get_pos() which directly gives the position of the mouse cursor returned in a tuple. This function, if not included in the Pygame library, would be difficult to implement. Pygame being full of multiple useful functions as the ones presented earlier, it will clearly help us in our case. The last Pygame module families we often use in the code are the pygame.draw and pygame.font modules. They are used to draw shapes like the lines of our Sudoku and text such as the Sudoku numbers and text on buttons and menus.

Let us now see some examples from our codebase with the mainmenu() function below, which is the first screen the user will encounter.

```
1  def mainmenu():
2      '''
3      This is the mainmenu screen
4      User can choose different other menus
5      '''
6      ecran =
7      pygame.display.set_mode((1000,600))
8      pygame.display.set_caption('mainmenu')
9      ecran.fill((0,0,0))
10
11      title =
12      text((255,0,0),500,100,80,'Sudoku Helper')
13      title1 =
14      text((255,0,0),500,170,50,'Main Menu')
15
16      startbutt =
17      button((0,0,0),350,300,300,50,'Start')
18      controlsbutt =
19      button((0,0,0),350,350,300,50,'Controls')
20      creditsbutt =
21      button((0,0,0),350,400,300,50,'Credits')
22      quitbutt =
23      button((0,0,0),350,450,300,50,'Quit')
24
25      running = True
26      while running:
27          pygame.display.flip()
28          title.printtxt(ecran,
29          Police/8-bit-operator
30          /8bitOperatorPlus8-Bold.ttf)
31          title1.printtxt(ecran,Police
32          /8-bit-operator/
33          8bitOperatorPlus8-Bold.ttf)
34          startbutt.printbutt(ecran)
35          controlsbutt.printbutt(ecran)
36          creditsbutt.printbutt(ecran)
37          quitbutt.printbutt(ecran)
38          for event in pygame.event.get():
39              if event.type == pygame.QUIT:
40                  pygame.quit()
41              if event.type ==
42              pygame.MOUSEBUTTONUP and
43              quitbutt.mouseon():
44                  pygame.quit()
45              if event.type ==
46              pygame.MOUSEBUTTONUP and
47              startbutt.mouseon():
48                  startmenu()
49              if event.type ==
50              pygame.MOUSEBUTTONUP and
51              controlsbutt.mouseon():
52                  Controls()
53              if event.type ==
54              pygame.MOUSEBUTTONUP and
55              creditsbutt.mouseon():
56                  Credits()
57          pygame.display.update()
```

In the code above we can observe the first part from line 11 to 23, in which we define the different objects we will use and display on the screen, which is generated by the code between line 6 and 9.
Having now defined everything we need, we will trigger the function with the infinite loop coded between line 25 and 26. Now that everything is in motion, we update the display at every loop run with pygame.display.flip() and pygame.display.update(). Inbetween, we use the printing methods to showcase our objects, in this function being buttons and texts. At line 38, we then start to look for every user input in order to respond properly to the users needs. Let us also note the use, once again, of the methods mouseon(), which allows to know when the user clicks on a certain cell.

We can observe the whole process of our application in figure 3. We are going to get more in detail in this section. The first screen is the mainmenu() screen, in which the user can choose between starting a game, seeing the controls or seeing the credits. The controls and credits menus are trivial ones, in which the users can only read information. The start menu, however, allows the user to choose the game mode: either user or random generating.

When selecting random generating, the user is then asked to choose a difficulty level, that will get him to the rndgen() screen, in which the whole game will take place. In this section the user can fill cells in the Sudoku, place a marker on specific cells, ask for a hint, validate already entered input or solve the whole Sudoku with the algorithm. Once the Sudoku is completed, the last screen, named summary, appears with all the summary statistics of the game.

If the user selects the user generating game mode, he is first shown an empty grid, in which he can fill the Sudoku grid with values of his choice. We infer that the user has found

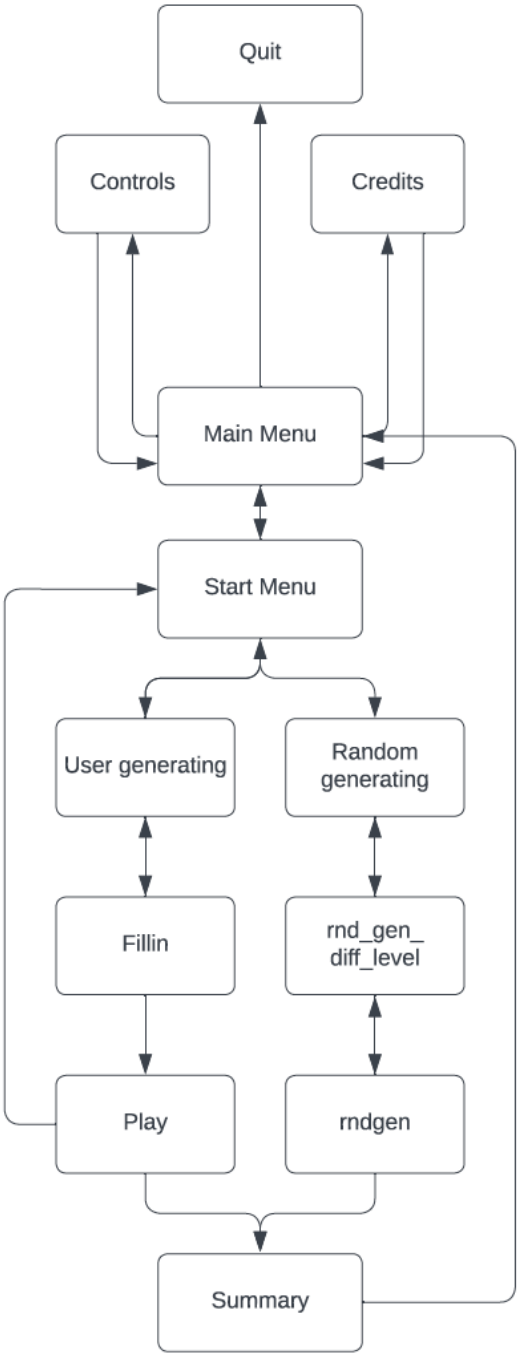

Fig. 3. Process of the application

a Sudoku grid in a newspaper for example, and wants to use the application to solve it. Once the Sudoku is ready to be solved, the user presses the val button that brings him to the play screen, in which the options are exactly the same than the before mentioned rndgen screen. Once again, when the Sudoku is finished, the last screen, summary, appears with all the summary statistics of the game.

Let us now talk about the file classes_file which contains the different classes that are used all along the GUI file. In this file, we define classes such as button, text that will help us in the graphical user interface. We will use methods, such as printbutt for example that will simply print a button on the screen. Another method, called mouseon tells us whether or not the mouse of the user is located on a button.

We also define two classes, namely rndgrid and usergrid, that represent the whole grid, once for the user generating mode and once for the random generating mode. The class cell, which represents each cell in a Sudoku grid, inherits from the two parent classes mentioned above. The inheritance here is important because when the method printgrid is used from the class user/rndgrid, the method from the class cell printnum_incell is called, which allows us to print values in the grid. Let us also present the method from user/rndgrid called mouseon(), that does not only tell whether the mouse is on the grid or not, but also returns the coordinate of the cell the mouse is located on.

You can find below a sample of the classes_file.py code in which you see the methods described above, with the classes usergrid and cell:

```
class usergrid():
    '''
    class used to generate Sudoku grids
    '''
    def __init__(self, width, height):
        self.ugrille =
         np.zeros((9,9),dtype = int)
        self.tgrille =
         np.zeros((9,9),dtype = int)
          self.cells =
          [[Cell(self.ugrille[i][j],i,j,
          width,height)
          for j in range(9)] for i in range(9)]
          self.width = width
          self.height = height
          self.errpron = 0

    def printgrid(self, surface):
        '''
        printgrid prints the grid
        on the screen,with the help function
        printnum_incell for the numbers
        in cells.
        It also print the counter of errors
        '''
        saut = self.width / 9
        for i in range(9+1):
            if i % 3 == 0 or i == 0:
                trait = 4
            else:
                trait = 1
            pygame.draw.line(surface,(0,0,0),
            (100, 30+i*saut),
            (100+self.width, 30+i*saut), trait)
            pygame.draw.line(surface,(0,0,0),
            (100+i*saut, 30),
            (100+i*saut, 30+self.height), trait)
        for i in range(9):
            for j in range(9):
                self.cells[i][j].
```

```
41                  printnum_incell(surface)
42
43          ffont =
44          pygame.font.Font(Police/
45          8-bit-operator/
46          8bitOperatorPlus-Regular.ttf,20)
47          fill =
48          ffont.render('Number of errors:'
49          +str(self.errpron),
50          True, (0,0,0))
51            surface.blit(fill,(730,450))
52
53      def mouseon(self, pos = (0,0)):
54          '''
55          mouseon returns the coordinate
56          of the cell in the Sudoku grid
57          where the mouse is,
58          if the mouse is on the Sudoku grid,
59          returns (-10,-10) otherwise
60          '''
61          pos = pygame.mouse.get_pos()
62          saut= self.width / 9
63          if 100 < pos[0] < 100+self.width
64          and 30
65          < pos[1] < 30+self.height:
66              return (int((pos[0]-100)//saut),
67              int((pos[1]-30)//saut))
68          else:
69              return (-10,-10)
70
71  class Cell():
72      '''
73      class used to represents cells and
74      number in it
75      '''
76      def __init__(self,value,row,col,
77      width,height):
78          self.value = value
79          self.row = row
80          self.col = col
81          self.width = width
82          self.height = height
83
84      def printnum_incell(self,surface,
85      col = (0,0,0)):
86          '''
87          printnum_incell prints numbers
88          in cells
89          '''
90          font = pygame.font.Font(Police/
91          8-bit-operator/
92          8bitOperatorPlus-Regular.ttf,30)
93          saut = self.width / 9
94          x = self.row * saut+109
95          y = self.col * saut+40
96          if self.value != 0:
97              txt = font.render(str(self.value),
98              True,col)
99              surface.blit(txt,(x+10,y))
```

IV. Implementation

For the solving algorithm, each function was coded in a separated .py file and since every file is located in the same folder, it is not a problem to import necessary functions from one file to another one.

Concerning the application code, it was decided to create one file for the classes, called classesfile.py, one file for the outside functions, named rndgen.py, which are called in the graphical user interface and one file with all the graphical user interface codebase, named GUI.py. Since each screen in the graphical user interface is a function in itself, it was discussed to create a file for each function, but we quickly came to the conclusion that it would not be feasible, since some functions call another function, which themselves call the previous function (think of the mainmenu that calls credits, that itself calls mainmenu) which would then lead to circular function calling.

Concerning the performance of our application, the vast majority of the codebase is efficient and there is no waiting time for the user. The solving algorithm is quite fast, and after testing it on a variety of different Sudokus, we end up with an average solving time of less than half a second, which is fast enough in our application.

Let us however mention that a special Sudoku invented by the finish mathematician Arto Inkala, called the “AI Escargot”, which has the reputation to be the hardest Sudoku grid ever invented, indeed proved to be pretty difficult to solve with our algorithm too, since it approximately took 47 seconds to solve. However, we want to point out that this is an exception and that the vast majority of the Sudoku grids available are solved in less than half a second.

Concerning the application itself, we encounter no waiting time in the user experience, except when generating a random grid. Even if most of the time it takes less than three seconds to generate a Sudoku grid, this process can sometimes take up to a minute. This part is worth spending some time on it, and assessing whether or not the generating process should be completely changed. However, the way of doing it in our codebase seems to be one of the most efficient ways possible. Another solution to this problem would be to use parallel programming. When starting the application, another computing resource could immediately start to generate at least two grids of each level of difficulty and then continue to generate supplementary grids when one grid is ‘used’ by the player. The objective would be to always have two grids that would be ready to be used.

V. Maintenance

The code base accessible for all three developers of the application is located in a GitHub repository. This allows every member of the team to easily have access to the base code and furthermore implement new functionalities to the application.

Let us note that the algorithm for solving the Sudoku in its current state works as expected and would therefore not require any changes in the short term. However, in the case we would like to add new functionalities or improve the efficiency, this could be done easily.

Obviously, the updating of the codebase will concern in its majority the application SudokuHelper.

The application and its graphical user interface are only in an alpha phase, which is a feature complete exploratory phase.

The objectives set in the introduction are now fulfilled and that is the reason we can affirm that the alpha phase is concluded.

The next phase of our project will consist of getting user feedback. This will be done by sending the application to a set of selected people and getting to understand their points about the parts that are already working well and the parts that are not working well and which would need to be changed in the codebase. We will call these points user requirements and will evaluate them, first by urgency of implementation.

Feedback has already been collected in a small close group, and the main aspects that are worth mentioning in this paper, in their order of urgency, are the followings:

- Remark 1: *Problem with selecting the level of difficulty in 'random generating'*:

This user requirement was already accepted by the developing team and changed in circumstance in the codebase. To understand what was done, it is required to know that initially, the part where the user could choose the difficulty of their Sudoku was coded in the following sense: the user needed to press a key on its keyboard, either 1 for easy, 2 for medium or 3 for hard. Some users were stuck on the screen in the part where they needed to select one of these options since they did not understand how to proceed to the next step. The idea of having the user use his keyboard was judged as a wrong decision by our first group of users, as it was not aligned with the overall way of using the application, which is mainly done by using the mouse and clicking on the button. After having checked that the user requirement was properly understood, the developing team proceeded to modify the codebase in order to meet the user's requirement. This is now considered as a completed task since the user who initially made this request is now satisfied with the new version.

Note that this requirement was considered "very urgent", due to the problematic nature of the situation. This task was considered as not complex.

- Remark 2: *Impossible to select the cell in which we want the hint to appear in*

This user requirement was, in a sense, not a surprise for the developing team, since it was already discussed before the coding phase. The developing team reached a consensus, that having the opportunity to choose the cell for a hint would make the solving process too easy, and therefore the team chose to not implement this functionality. Instead, it was decided to give the user a random hint.

As some user mentioned that the possibility to select a cell for the hint was lacking in the current application, the developing team is now reconsidering implementing such a functionality. This user requirement is defined as not urgent and not complex, and the final decision will be taken after receiving more user feedback, in order to see whether it is mostly encouraged by our users.

- Remark 3: *"Well done!" message even though user clicks on Solve Button*

This user requirement is defined as not urgent and somewhat complex by the developing team. Some users have pointed out the fact that the last screen (summary function in GUI.py), a congratulation message appears even when the user clicked on the Solve button and thus did not really complete the Sudoku themself. The developing team is setting the complexity level for this task as somewhat complex since it would require creating a new function for a summary screen, when the user clicks on solve and does not intend to solve the Sudoku themself. Furthermore, this will require more feedback in order to take a final decision.

VI. Results

On the overall, we end up with about eight hundreds lines of code, which is quite compact for the complete application and solving algorithm.

As a reminder, the goals we fixed ourselves in the introduction were to help beginners to solve a Sudoku game and make their user experience more friendly and less frustrating. This can be done by using the val button, which will tell the user if his already inputed values are correct or not. After the different feedbacks we received from our users, it was confirmed that the val button was a very helpful option in our game. The val button will thus save a lot of time to the users and it will avoid them to have to finish filling all the cells of the Sudoku in order to check if their input was correct. Even if more experienced players told us that this feature should not be inserted, since they found that the val button eliminated the logical thinking part of playing Sudoku, we decided to keep it in our implementation, since as already mentioned in the introduction, the idea was to encourage new players to not give up this game too quickly. The rest of the functionalities are also well implemented and allow the user to have an enjoyable user experience.

VII. Conclusion

To conclude this paper, and more globally our overall project, we are proud of the final application and think that all the set goals have been fulfilled. We are aware that some changes are required in order to make the overall user experience more enjoyable, but think that the results are in alignment with our expectations.

This project could be continued with the introduction of optical character recognition (OCR) instead of the user filling every cell separately (screen where the user fills his Sudoku grid), which would be more user-friendly and also more time efficient. This part was not introduced in our project since OCR is a topic in machine learning, which is not the content of this Advanced programming course.

The modules Numpy and Pygame also were of great help for the implementation of the code.

The development of the application was done over a period of three months, with the most complex part consisting in the understanding and development of the recursion process. The most time consuming part clearly was the development of the application, with first the development of the flow chart and the design of each screen and then also the coding part.

We are also glad to have had the opportunity to let five of our friends test our application and we are satisfied with their overall feedback.