Group Project Report

Sudoku Solver using Simulated Annealing in Python

Francesco Doti

Note on compatibility

This code was written with Python version 3.8.5, numpy 1.23.5

Introduction

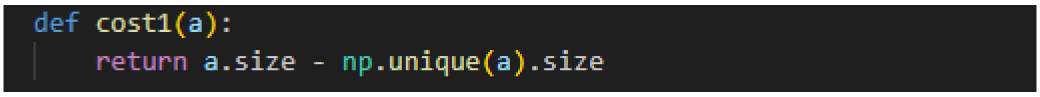
Sudoku is highly engaging and widely utilized in mental exercise and mathematical challenges. The challenge of this puzzle is to fill a grid consisting of nine 3x3 subgrids and 9x9 cells with the complete set of digits from 1 to 9 without any redundancy in each row, column. Despite its simple concept, finding a solution still requires logic and strategic thinking. To tackle Sudoku, several algorithms have been used, including Simulated Annealing, a mathematical method used to find an optimal solution by trying various possible solutions and improving the solution based on previous results.

The primary aim of this project is to develop a Python script that employs simulated annealing to solve Sudoku puzzles efficiently. The project entailed developing a Sudoku generator (Sudoku_v0.py), creating a three-phase Sudoku solver (SimAnn_v0.py;SimAnn_v1.py; SimAnn_v2.py), and crafting an instantiation script (sudokurun_v0; SudokuGUI_v1_c; SudokuGUI_v2_c) that merges these elements to effectively tackle and solve Sudoku challenges. We also developed a graphical user interface that enables the user to try to solve a computer-generated sudoku of dimension n and check whether the solution of the user is the same as the one found by our program.

I. The creation of Sudoku Puzzle [Name of script: Sudoku_v0]

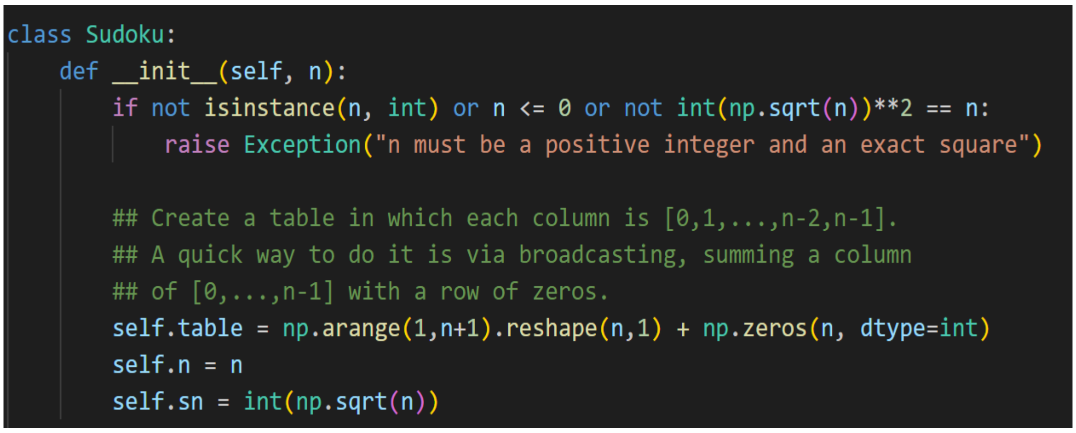
1. Class and Function
   Class: 'Sudoku' this class' is designed to represent and manipulate a Sudoku puzzle. Function: 'cost1(a)' an auxiliary function used in Sudoku class

2. Use of Each Function



```python
def cost1(a):
    return a.size - np.unique(a).size
```

The use of function ‘cost1(a)’ is to return the number of repeated elements in the input array. It does this by subtracting the number of unique elements from the size of the original array.



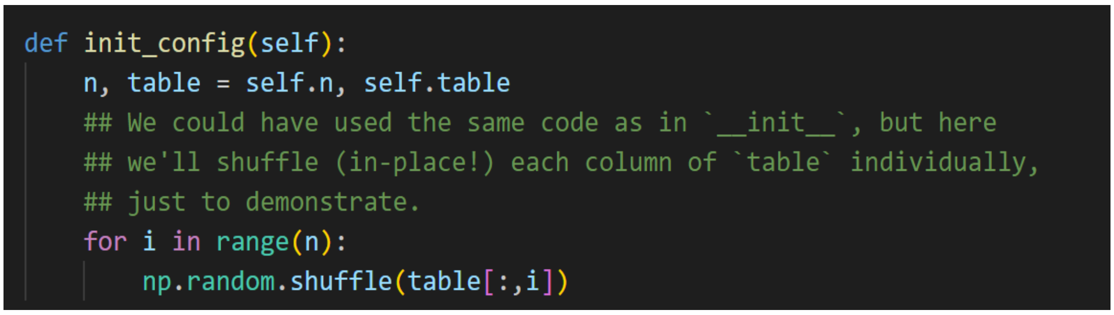
```python
class Sudoku:
    def __init__(self, n):
        if not isinstance(n, int) or n <= 0 or not int(np.sqrt(n))**2 == n:
            raise Exception("n must be a positive integer and an exact square")

        ## Create a table in which each column is [0,1,...,n-2,n-1].
        ## A quick way to do it is via broadcasting, summing a column
        ## of [0,...,n-1] with a row of zeros.
        self.table = np.arange(1,n+1).reshape(n,1) + np.zeros(n, dtype=int)
        self.n = n
        self.sn = int(np.sqrt(n))
```

In ‘Sudoku’ Class

‘__init__(self,n) ’: ‘n’ is the size of a Sudoku grid, which means the number of columns or rows. The first check is to verify if ‘n’ is an integer and greater than zero, and also to check whether n is a perfect square which means its square root should be an integer. If ‘n’ does not fit these criterias, then an exception will be raised with a message that Sudoku can not be created due to invalid dimensions.

In order to create a n-dimension Sudoku, ‘np.arrange(1, n+1)’ can first create a 1-dimensional array starting with 1 to n; then ‘.reshape(n,1) reshape it into 2-dimensional array, which n-rows and 1 column; finally, ‘ +np. zeros(n, dtype=int)’ uses NumPy’s broadcasting feature to sum a column of zeros to reshape the array. The result is a ‘n*n’ 2-dimensional array since zero does not change the value. A n-dimension Sudoku table is then created.

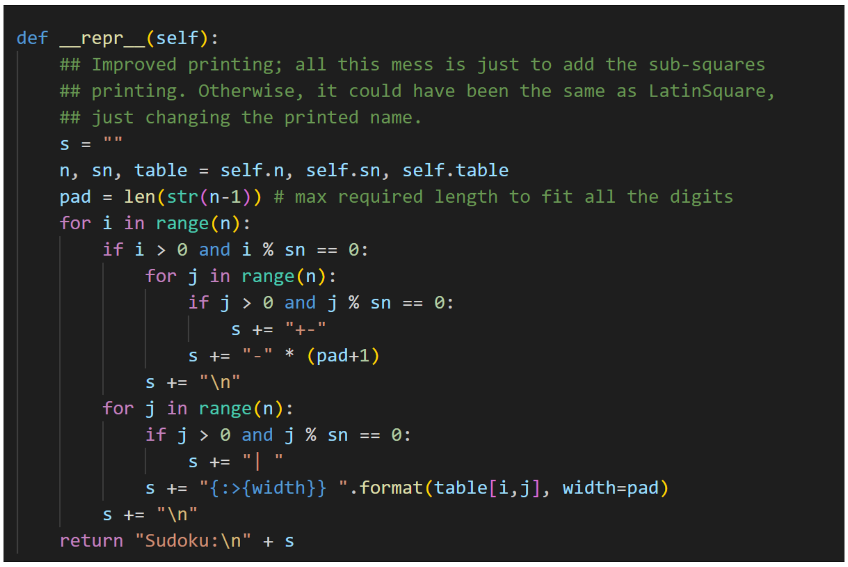


```python
def init_config(self):
    n, table = self.n, self.table
    ## We could have used the same code as in `__init__`, but here
    ## we'll shuffle (in-place!) each column of `table` individually,
    ## just to demonstrate.
    for i in range(n):
        np.random.shuffle(table[:,i])
```

The ‘ init_config(self)’ function is to create an initial configuration for Sudoku by shuffling the number in each column of the Sudoku table. The line ‘n, table = self.n, self.table’ creates two variables ‘n’ and ‘table’ for convenience:

- ‘self.n’ : an instance variable that represents the number of rows and columns, since the Sudoku table can only be square, n is the size of table
- ‘self.table’ : a two-dimensional Numpy array

The loop for i in range(n): np.random,shuffle(table[:,i]) randomly rearranges the element in each ‘i-th’ column, then each column will have different sequences.



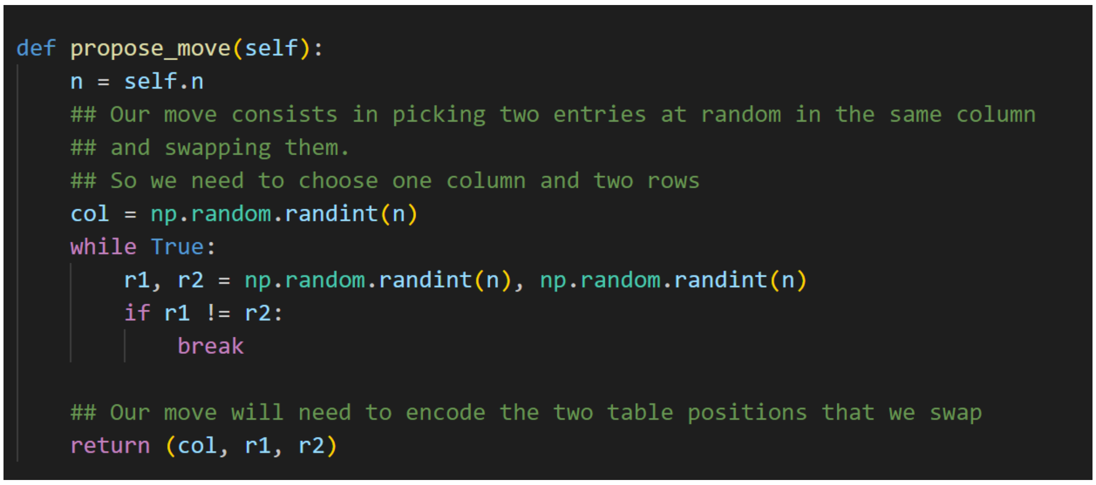
```python
def __repr__(self):
    ## Improved printing; all this mess is just to add the sub-squares
    ## printing. Otherwise, it could have been the same as LatinSquare,
    ## just changing the printed name.
    s = ""
    n, sn, table = self.n, self.sn, self.table
    pad = len(str(n-1)) # max required length to fit all the digits
    for i in range(n):
        if i > 0 and i % sn == 0:
            for j in range(n):
                if j > 0 and j % sn == 0:
                    s += "+-"
                s += "-" * (pad+1)
            s += "\n"
        for j in range(n):
            if j > 0 and j % sn == 0:
                s += "| "
            s += "{:>{width}} ".format(table[i,j], width=pad)
        s += "\n"
    return "Sudoku:\n" + s
```

The goal of the ‘__repr__’ function is to add sub-squares inside the Sudoku table in order to make it in human-readable format.

- Local variables: are ‘n’, ‘ sn’ and ‘table’ respectively, extracting the values from the size of Sudoku ‘self.n’, the size of sub-square ‘self.sn’, and the current state of table ‘self.table’.

- ‘Pad = len(str(n-1)’: the largest length require to fit all the possible digits, which is ‘n-1’ since the dimension of table start from 1

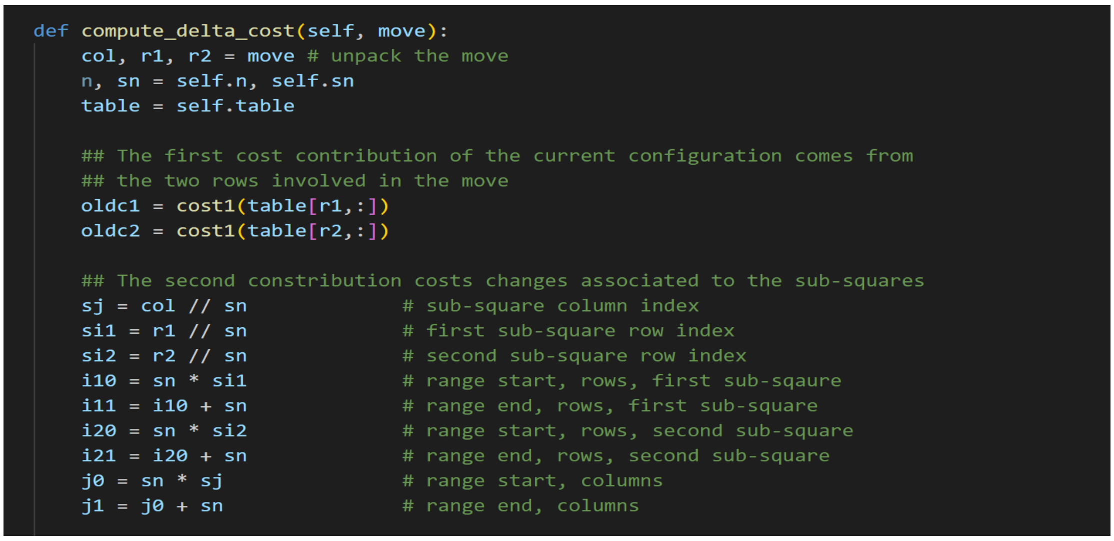
- The loop is to add visual separators between the sub-squares, for example, in a 9x9 Sudoku, there are six 3x3 sub-squares inside. For each row ‘i’, the code ‘i % sn == 0’ checks if the row number is a multiple of sub-square; if it valid the first condition and in the meantime, it is not the first row (‘i’>0), it adds a horizontal line made of dashes (‘-”) and plus signs (‘’+”) to mark the boundary, and then print the number in the row, adding a vertical bar before the number start a new-sub square. Once each number has been printed, it moves to the next line and begins producing subsequent rows.



```python
def propose_move(self):
    n = self.n
    ## Our move consists in picking two entries at random in the same column
    ## and swapping them.
    ## So we need to choose one column and two rows
    col = np.random.randint(n)
    while True:
        r1, r2 = np.random.randint(n), np.random.randint(n)
        if r1 != r2:
            break

    ## Our move will need to encode the two table positions that we swap
    return (col, r1, r2)
```

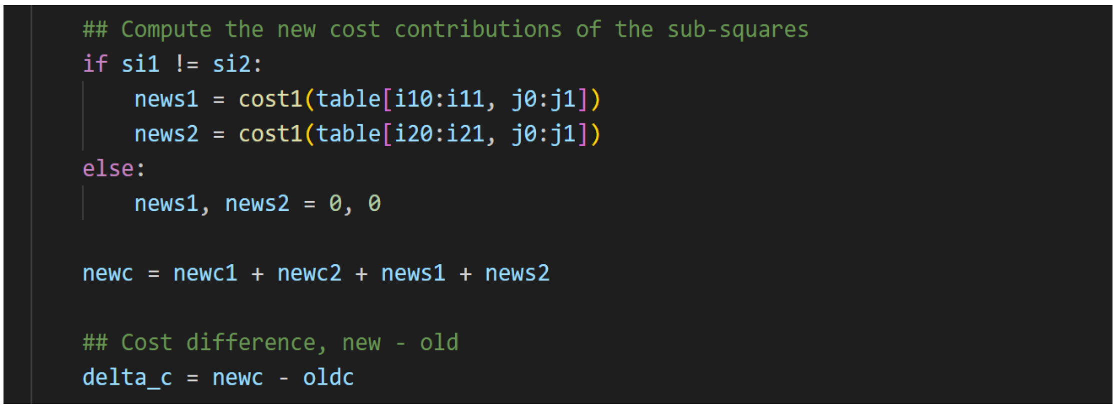
The method ‘propose_move’ involves two numbers in the same column but in different rows (r1! = r2) to ensure a swap can be achieved, once two different rows are found, the loop is finished and returns the tuple.



```python
def compute_delta_cost(self, move):
    col, r1, r2 = move # unpack the move
    n, sn = self.n, self.sn
    table = self.table

    ## The first cost contribution of the current configuration comes from
    ## the two rows involved in the move
    oldc1 = cost1(table[r1,:])
    oldc2 = cost1(table[r2,:])

    ## The second constribution costs changes associated to the sub-squares
    sj = col // sn              # sub-square column index
    si1 = r1 // sn              # first sub-square row index
    si2 = r2 // sn              # second sub-square row index
    i10 = sn * si1              # range start, rows, first sub-sqaure
    i11 = i10 + sn              # range end, rows, first sub-square
    i20 = sn * si2              # range start, rows, second sub-square
    i21 = i20 + sn              # range end, rows, second sub-square
    j0 = sn * sj                # range start, columns
    j1 = j0 + sn                # range end, columns
```

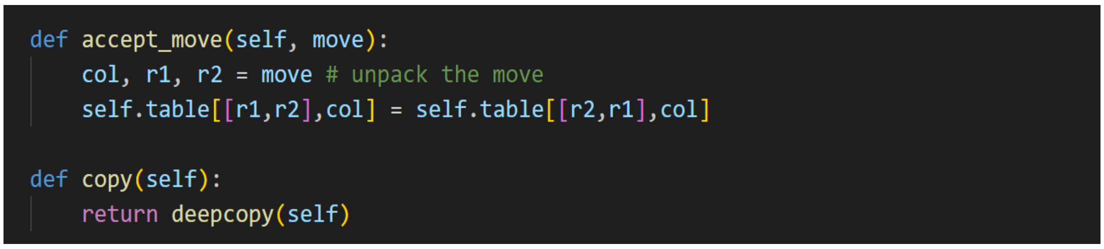


```python
    ## Compute the new cost contributions of the sub-squares
    if si1 != si2:
        news1 = cost1(table[i10:i11, j0:j1])
        news2 = cost1(table[i20:i21, j0:j1])
    else:
        news1, news2 = 0, 0

    newc = newc1 + newc2 + news1 + news2

    ## Cost difference, new - old
    delta_c = newc - oldc
```

The ‘comput_delta_cost’ function is to determine whether the swap brings the grid closer to a valid Sudoku solution, it computes the difference ‘cost’ between the before and after swap.



```python
def accept_move(self, move):
    col, r1, r2 = move # unpack the move
    self.table[[r1,r2],col] = self.table[[r2,r1],col]

def copy(self):
    return deepcopy(self)
```

The function accept_move(self, move) changes the state of the Sudoku table permanently by performing a swap of two values in the same column.
The function copy(self) allows to return a duplicate of the Sudoku instance, which means any modification made to duplication will not affect the original instance.

II. The 3 stages

1. Stage V0

The main goal of the script is to implement a simulated annealing algorithm for solving Sudoku puzzles. Interesting aspects of this are a stochastic optimization approach, the use of the metropolis rule for move acceptance and the integration with the sudoku puzzle representation (Sudoku_v0.py). Our V0 is characterized by 3 scripts: (i) Sudoku_v0 which is described in the previous paragraph and is kept the same in V1 and V2, (ii) SimAnn_v0.py and (iii)sudokurun_v0.py.

1.1 SimAnn_v0.py

```python
def accept(delta_c, beta):
  ## If the cost doesn't increase, we always accept
  if delta_c <= 0:
     return True
  ## If the cost increases and beta is infinite, we always reject
  if beta == np.inf:
     return False
  ## Otherwise the probability is going to be somewhere between 0 and 1
  p = np.exp(-beta * delta_c)
  ## Returns True with probability p
  return np.random.rand() < p
```

The accept function plays a crucial role in the simulated annealing algorithm by determining whether to accept or reject a proposed move based on the Metropolis

rule. Its primary goal is to guide the exploration of the solution space during the annealing process.
The input parameters are:

- delta:c: the change in cost resulting from the proposed move
- beta: the current temperature parameter during the annealing process

In this case we introduce the Metropolis Rule. The Metropolis rule is a fundamental concept in the Metropolis-Hastings algorithm, widely used in Markov Chain Monte Carlo (MCMC) methods, including simulated annealing. Through the explanation of the code we are going to dig deeper into this topic.

- Our function evaluates the metropolis criterion, which is defined as exp(-β*delta_c).
- If the change in cost (*delta_c*) is negative, the move is accepted with a probability based on the temperature (*beta*) and the change in cost
- The higher the temperature, the more likely the algorithm is to accept moves that worsen the solution, facilitating exploration

The output returns True f the move is accepted and False otherwise

The other function in this script is simann:

```
def simann(probl,
        anneal_steps = 10, mcmc_steps = 100,
        beta0 = 0.1, beta1 = 10.0,
        seed = None, debug_delta_cost = False):
```

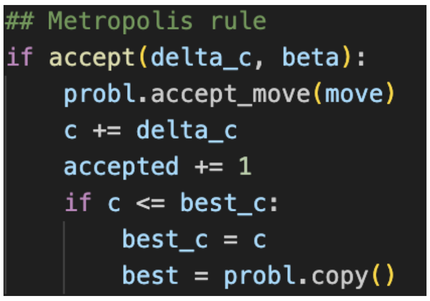
This function encloses the core logic of the simulated annealing algorithm. It moves the exploration of the solution space by iteratively proposing moves and deciding whether to accept or reject them based on the Metropolis rule.
The input parameters are:

- *probl*: represents the problem instance, which in our case is the Sudoku puzzle. The methods required from this parameter are the following:
    - cost(): computes and returns the cost of the current configuration
    - propose_move(): Proposes a symmetric move in the solution space.
    - compute_delta_cost(move): Computes the change in cost associated with a move.
    - accept_move(move): Accepts a proposed move and updates the internal configuration.
    - copy(): Creates a new, independent instance of the problem.
- *anneal_steps*: The number of steps in the simulated annealing process.
- *mcmc_steps*: The number of Markov Chain Monte Carlo (MCMC) steps per annealing step.
- *beta0*: The initial temperature parameter for simulated annealing.
- *beta1*: The final temperature parameter for simulated annealing.
- *seed*: The seed for the random number generator. Optional for reproducibility.

- *debug_delta_cost*: Flag to enable expensive checks for compute_delta_cost functionality.

The function typically contains a loop that iterates through different stages of the annealing process, and within each iteration a move is proposed (e.g. swapping two entries in the Sudoku grid) and the change in cost (*delta_c*) is computed. This is the **annealing loop**.

The **metropolis acceptance** is then called and it decides whether to accept or reject the proposed move.



```python
## Metropolis rule
if accept(delta_c, beta):
    probl.accept_move(move)
    c += delta_c
    accepted += 1
    if c <= best_c:
        best_c = c
        best = probl.copy()
```

The output is the final solution or the best solution found during the annealing process

Design decisions

1. **Symmetric proposal assumed**: In the simulated annealing algorithm, a move is proposed during each iteration. The assumption of symmetric proposals simplifies the acceptance and reversal of moves. Symmetry ensures that the proposal distribution is the same as the reverse proposal distribution, making it easier to apply the Metropolis rule for acceptance.
2. **Metropolis rule applied for move acceptance**: The Metropolis rule is a key aspect of simulated annealing. It provides a probabilistic criterion for accepting or rejecting a move based on the change in cost and the current temperature (controlled by the parameter beta). The rule ensures that moves leading to a lower cost are always accepted, while moves increasing the cost have a probability of acceptance dependent on the temperature.
3. **Beta values spaced between *beta0* and *beta1* during annealing**: The annealing schedule is crucial for the success of simulated annealing. The beta parameter controls the temperature, and annealing involves gradually reducing this temperature. In the provided code, a list of beta values is generated, starting from beta0 and ending at beta1. The last value is set to infinity, ensuring that at the end of the annealing process, only moves leading to a lower cost are accepted.
4. **Cost computation**: the code includes a print statement (print(probl)) to display information about the problem, and the initial cost is computed using probl.cost(). The initial cost represents the objective function value associated

with the initial state of the problem. This information is essential for evaluating the effectiveness of moves during the annealing process.

```
print(probl)

c = probl.cost()

print(f"initial cost = {c}")
```

1.2 sudokurun_v0.py

This is the script used to run the Sudoku game. The puzzle is solved through the simulated annealing algorithm. The script initializes a Sudoku instance (sdk) from the Sudoku_v0 module and imports the simulated annealing solver (SimAnn_v0). The goal is to find the optimal configuration for the Sudoku puzzle using the simulated annealing algorithm.

The code breaks down as the following:

Sudoku Initialization: An instance of the Sudoku class (sdk) is created with a specified puzzle size (in this case, 9x9).
Simulated Annealing Solver: The simann function from the SimAnn_v0 module is then called, passing the Sudoku instance (sdk) as the problem to be solved. Parameters like the number of Monte Carlo steps (mcmc_steps), random seed (seed), initial beta (beta0), and the number of annealing steps (anneal_steps) are set.
Print Final Configuration: After the simulated annealing process, the final configuration with the lowest cost is printed along with its associated cost. The cost represents the quality of the solution, with lower values indicating better solutions.

The main script provides an interface between the Sudoku problem and the simulated annealing algorithm, facilitating the optimization process. The utilization of these modules allows for a modular and organized approach to problem-solving.

Design decisions

1. **Initialization of a Sudoku instance with a specific siz**e: In the script (sudokyrun_v0.py), a Sudoku instance (sdk) is initialized using the Sudoku_v0 module. The Sudoku class is responsible for creating and managing the Sudoku puzzle. In the Sudoku_v0 module, the Sudoku class is defined to

handle the initialization, representation, and manipulation of Sudoku puzzles. It uses a 2D NumPy array to represent the puzzle grid.

2. **Running simulated annealing to solve the Sudoku puzzle**: The core solving mechanism involves applying the simulated annealing algorithm to find the optimal configuration for the Sudoku puzzle. This is accomplished using the simann function from the SimAnn_v0 module. Here, parameters such as the number of Monte Carlo steps (mcmc_steps), random seed (seed), initial beta (beta0), and the number of annealing steps (anneal_steps) are specified. The simann function applies the simulated annealing algorithm to iteratively improve the Sudoku configuration.
3. **Printing the final configuration and cost**: After the simulated annealing process, the final Sudoku configuration with the lowest cost is printed, along with its associated cost. The cost represents the quality of the solution, with lower values indicating better solutions. The np.set_printoptions(threshold=np.inf) is used to ensure that the entire Sudoku grid is printed, even if it exceeds the default display threshold.

This part of the script allows users to visualize the solution obtained through simulated annealing and assess the quality of the Sudoku solution based on the associated cost.

1.3 Evolution from V0 to V1

Key features from the V0 stage which are kept later on are the (i) core simulated annealing logic and the (ii) Sudoku representation and move proposals.

i. Core simulated annealing: In the `SimAnn_v0.py` module, the core simulated annealing logic is implemented. The `simann` function orchestrates the simulated annealing process. It iterates over a range of beta values, each corresponding to a specific annealing step. For each beta, a set number of Monte Carlo steps are executed to explore the solution space.
The acceptance of moves during the annealing process is governed by the Metropolis rule, implemented in the `accept` function. This rule probabilistically accepts or rejects moves based on the change in cost and the annealing temperature (beta).

ii. Sudoku representation and move proposal: The `Sudoku_v0.py` module defines the `Sudoku` class, responsible for representing the Sudoku puzzle and providing methods for proposing moves. The puzzle is represented as a 2D NumPy array, and the `propose_move` method suggests a move by randomly selecting two entries in the same column and swapping them.

During the code development challenges came up that led to the second phase V1. Those challenges were (i) convergence issues and (ii) Unsolvable Sudoku configurations.

i. Convergence issues: One challenge encountered in the initial implementation (`V0`) is related to convergence. Simulated annealing may not always converge to a solution, and there is a need to ensure that the proposed Sudoku configurations are solvable. In `SimAnn_v0.py`, a check for convergence is added. If the best cost remains zero, indicating a solved puzzle, the `converge` variable is set to `True`. This check ensures that the Sudoku puzzle is solvable, and the simulated annealing process converges to a valid solution.

ii. Unsolvable Sudoku configurations: Another challenge is the possibility of generating unsolvable Sudoku configurations during the annealing process. In the absence of constraints, the generated configurations might not adhere to Sudoku rules. The convergence check mentioned earlier helps identify and handle such situations, ensuring that only solvable Sudoku puzzles are presented as solutions.

These features and challenges highlight the complexity and nuances involved in applying simulated annealing to solve Sudoku puzzles, making it necessary to address issues related to convergence and solution validity.

2. Stage V1

Goals of this stage are solving the previous challenges, addressing convergence issues and enhancing the user's interaction by creating a GUI. Scripts in this phase are: (i) Sudoku_v0, (ii) SimAnn_v1.py and (iii) SudokuGUI_v1_c.py.

2.1 SimAnn_v1.py

Modules/classes

- the simann and accept functions are the same as before
- *convergence* global variable: the addition of this variable is what changes between SimAnn_v0.py and SimAnn_v1.py.
  ```
  converge = [False]
  ```
  This variable is a list with a boolean element. It serves as a flag to indicate whether the simulated annealing process has converged to a solvable Sudoku configuration. The global nature of this variables allows it to be accessed and modified across different functions and scripts

Key differences between SimAnn_v0.py and SimAnn_v1.py.

- Convergence check: The most significant difference lies in the introduction of a convergence check. In **SimAnn_v1.py**, the code checks if the best cost (*best_c*) is zero after the annealing process. If it is, the converge flag is set to True. This helps ensure that the Sudoku generated by the annealing process is solvable.

- Global variable usage: The *converge* variable is a global variable, allowing it to be accessed and modified from different parts of the code. In this case, it is used to communicate the convergence status between the simann function and the outer script (SudokuGUI_v1_c.py).

These differences highlight the evolution of the simulated annealing process in `SimAnn_v1.py`. The addition of a convergence check enhances the reliability of the generated Sudoku solutions, ensuring they align with the rules of the puzzle.

2.2 SudokuGUI_v1_c.py
This is the new version of sudokurun_v0.py. It's a GUI script built using Tkinter, importing Sudoku_v0 and SimAnn_v1. Also the random module is imported for this script.

<u>Modules/classes</u>

Tkinter:
- Tkinter is the primary module used for creating the graphical user interface. It provides classes and functions for creating windows, labels, buttons, and other GUI elements to interact with the user.

Sudoku_v0:
- Sudoku_v0 is imported, and an instance of the Sudoku class is created. This class likely represents the Sudoku puzzle and provides methods for initializing and solving it.

SimAnn_v1:
- SimAnn_v1 is imported and utilized for simulated annealing-based optimization to solve Sudoku puzzles. The script appears to run the simulated annealing algorithm until convergence.

<u>Design decisions</u>

Graphical User Interface (GUI):
- The script utilizes Tkinter to create a GUI for interacting with the Sudoku puzzle. This includes labels, buttons, and an interactive grid for user input.

Validation Function:
- The ValidateNumber function is used for validating input in the cells, allowing only digits or empty strings. It is registered with Tkinter for use in validating user input.

Grid Drawing Functions:
- Functions like draw3x3Grid and draw9x9Grid are designed to draw a 9x9 Sudoku grid in the GUI. The grid is divided into 3x3 subgrids, enhancing visual clarity.

User Interaction Functions:

- Functions like initValues, clearValues, and getValues are associated with buttons for initializing, clearing, and solving the Sudoku puzzle, respectively. User actions trigger these functions.

<u>Differences with sudokurun_v0.py</u>

Graphical Interface:

- SudokuGUI_v1_c.py introduces a graphical interface using Tkinter, allowing users to interact with the Sudoku puzzle visually. In contrast, sudokurun_v0.py lacks a graphical interface and relies on console-based output.

Interactive Initialization and Solving:

- In SudokuGUI_v1_c.py, users can initialize the puzzle with random values, clear cells, and solve the puzzle interactively through buttons. sudokurun_v0.py directly prints the final best configuration to the console.

Simulated Annealing Convergence:

- The loop in SudokuGUI_v1_c.py runs the simulated annealing algorithm until convergence. sudokurun_v0.py runs simulated annealing once and directly prints the final configuration.

Validation and User Input:

- SudokuGUI_v1_c.py includes validation of user input in the GUI, ensuring only valid entries are accepted. sudokurun_v0.py assumes proper inputs and directly prints the result.

2.3 Evolution from V1 to V2

From v1 to v2 the main difference is the creation of an additional window to choose the difficulty of the Sudoku and the implementation of 3 different difficulty levels in the SudokuGUI file

Also SimAnn is adapted not to print the initial configuration which would be not interesting for the user and not useful for the game.

Moreover in the code in SudokuGUI the probability of the numbers of the solution appearing when clicking the button start are modified and adjusted for each difficulty such that each level is solvable and not trivial.

3. Stage V2

This is the final stage of the code. It is characterized by 3 scripts: (i) Sudoku_v0.py, (ii) SimAnn_v2.py and (iii) SudokuGUI_v2_c.py.

3.1 SimAnn_v2.py

Modules/classes and design decisions

The same as SimAnn_v1.py, only thing SimAnn is adapted not to print the initial configuration which would be not interesting for the user and not useful for the game, and in SimAnn_v2.py, the line print(probl) was removed.

3.2 SudokuGUI_v2_c.py

Modules/classes

- *set_difficulty*: Sets the difficulty level based on user input.
- *create_difficulty_selection_window*: Creates a difficulty selection window using Tkinter.
- *start_Sudoku*: Initializes and runs the Sudoku GUI based on the selected difficulty level.
- Validation functions (ValidateNumber): Validate user input based on the difficulty level.
- Draw functions (draw3x3Grid, draw9x9Grid, draw5x5Grid, draw25x25Grid, draw4x4Grid, draw16x16Grid): Draw the grids based on the Sudoku size and subgrid size.
- Action functions (clearValues, initValues, getValues): Perform actions related to initializing, clearing, and obtaining values from the solution.

Design decisions

- Dynamic Difficulty Levels:
    - The code introduces a dynamic difficulty level selection feature, allowing users to choose between "Easy," "Medium," and "INSANE."
    - The GUI window changes its size and button positions based on the selected difficulty, providing a more user-friendly experience.
- Modularized Code:
    - The code is well modularized into separate files and functions, making it more readable and maintainable.
    - Each module/class has a specific responsibility, enhancing code organization.
- Improved User Interaction:
    - The use of Tkinter for GUI enables a user-friendly interaction for solving Sudoku puzzles.
    - The introduction of a difficulty selection window enhances the user experience by providing options before starting the game.

Shift from V1 to V2

- GUI Enhancement:

    - V2 introduces a more interactive GUI with difficulty levels, allowing users to choose the complexity of the Sudoku puzzle.
    - The dynamic creation of windows based on user choices enhances the overall user experience.
- Code Refactoring:
    - V2 involves significant code refactoring, introducing separate modules for Sudoku logic, simulated annealing, and GUI.
    - Improved function names and code organization enhance clarity and readability.
- Difficulty Levels:
    - V2 introduces different difficulty levels, requiring adjustments in the drawing, validation, and initialization functions to accommodate varying grid sizes and constraints.
- Simulation Parameters:
    - The simulated annealing parameters, such as annealing steps and MCMC steps, are configurable, providing flexibility in tuning the algorithm for different scenarios.
- Validation Function Changes:
    - Validation functions in the GUI are adapted to accommodate different difficulty levels, considering the number of digits in the generated Sudoku puzzle.

Conclusion

In conclusion, our original code in Sudoku_vo was designed to shuffle numbers in Sudokus rather than to place some blank spots to solve the Sudokus. To accomplish this, we assigned certainty values to each level, denoting the likelihood that a particular number will be present in the initial stage. Additionally, certain Sudoku configurations are unsolvable: In order to verify that the proposed Sudoku is user-solvable, we must include a convergence check in version v_1 of SimAnn. Run the SimAnn call within the Sudoku_GUI script loop until the algorithm converges and the best_cost equals zero. Additionally, the integration of difficulty levels into version 2 necessitated an entire restructuring of the code.