SUDO-QU-ICK, A GAME PLAYING AGENT

Dipak Sairamesh
Northeastern University, USA

Abstract

The problem statement of the final project for Foundations of Artificial Intelligence seeks to explore an application of AI on a real-life problem. For the sake of making this interesting, a sudoku game-playing agent will visually solve puzzles from a million quizzes Kaggle dataset or by fetching the puzzle from Sugoku. Exploration combined with knowledge of search algorithms and constraint satisfaction, in addition to Python 3.7 and PyCharm IDE are used to accomplish visually appealing resolutions as well as bringing each step to life.

Introduction

Throughout this document, the sudoku puzzle is referred to as the **grid**. In this report, the detailed development and implementation of a simple Sudoku game-playing agent, ***Sudo-Qu-ick*** is presented. It will serve as a framework to visually analyze the performance of search algorithms in terms of benefits, speed, and efficiency based on the difficulty level of the grid. Additional resources implemented can fetch our grids from the web using JSON responses.

Sudo-Qu-ick consists of a graphical user interface, solving algorithm and puzzle generator compiled from various sources, implemented using python's pygame package. In more words, this project will quash the hindrances in trying to solve Sudoku on a newspaper or a cereal box. There's no need to be worried about making mistakes and botching up the paper entirely while trying to erase!

Background

Sudoku is a logic-based puzzle consisting of a partially solved n x n grid where numbers from 1 to n are placed in the missing spaces such that they follow the given constraints:

1. Every row in the grid must contain all numbers 1 to n
2. Every column in the grid must contain all numbers 1 to n
3. Every box in the grid must contain all numbers 1 to n

The number of given clues determine the difficulty level of the game. A 9 x 9 grid can be classified as follows based on the given number of clues:

Easy: 36 to 49
Medium: 32 to 35
Hard: 28 to 31

Constraint Satisfaction

A constraint satisfaction problem (CSP) is a problem that requires its solution within some limitations or conditions also known as constraints. It consists of the following:

- A finite set of variables which stores the solution (V = {V1, V2, V3,....., Vn})
- A set of discrete values known as domain from which the solution is picked (D = {D1, D2, D3,.....,Dn})
- A finite set of constraints (C = {C1, C2, C3,....., Cn})

Brute Force

In general, brute force method fills in numbers from existing options in empty squares or removes unsuccessful choices if a "dead-end" is reached. Brute force, for example, can solve a problem by placing the numeral "1" in the first square. If the digit is permitted by verifying the row, column, and box, the agent advances to the next square and places the numeral "1" there. When the agent realizes that the "1" isn't permitted, the digit is incremented by one, becoming 2. When the computer notices a square where none of the numbers (1 to 9) are allowed, it goes back to the previous square. The value of that square then rises by one. This process repeats until all 81 squares display the correct solution.

Backtracking

Backtracking is an algorithmic strategy in which the goal is to use brute force to find all solutions to a problem. It consists of gradually compiling a set of all possible solutions. It finds a solution set by developing a solution step by step, increasing levels over time, using recursive calling. A backtracking algorithm employs the depth-first search strategy.

When it begins exploring the solutions, a boundary function is applied so that the algorithm can determine whether the thus-far built solution satisfies the restrictions. If it does, it will continue to look. If it does not, the branch is removed, and the algorithm returns to the previous level.

Depth First Search

The Depth First Explore (DFS) technique uses backtracking to traverse or search tree or graph data structures. It traverses all the nodes, either forward or backward, if possible. The time complexity of Depth First Search if the entire tree is traversed O(V)O(V) where *V* is the number of nodes.

In the case of a graph, the time complexity is O(V+ E)O(V+E) where *V* is the number of vertexes and *E* is the number of edges.

Related Work

This endeavor aimed at exploring character recognition through the *MNIST handwritten digits dataset for an Optical Character Recognition Neural Network*. The model was unfortunately inefficient against non-stock images, introspecting the idea of implementing API call requests for fetching grids from the web directly. An additional feature is provided to load a puzzle from a million-puzzle dataset directly.

Various other in-game features could be introduced in this project which explores the solving rate of various difficulties of puzzles based on different search algorithms. Various other widget features can be created to provide more user functionality and game options. Some non-conventional and human-techniques on solving Sudoku could also be important parameters. The solver in this project will conduct a depth-first search of the possible solutions to the Sudoku grid. In the worst-case situation, it will go through all the available grid combinations, eliminating some as it fills the grid. As a result, it is an exponential algorithm. This solver should technically take too long to solve harder Sudoku puzzles because of the exponential growth in complexity. Before trying the recursive step, the grid combinations could be filtered more effectively to reduce compute time further.

Approach

Pygame will form the basis of the game-playing agent and pygame-widgets is employed to visualize and update events.

DrawGrid and **DrawModes** functions are defined to output the layout of the game as a pop-up window. DrawGrid goes line by line for a 9x9 sudoku in a nested loop and draws the cells of the grid that will form as the foundation of the sudoku game. Custom RGB values are parameterized to provide a visually appealing solver. The function also checks and draws thicker lines to denote the 9 different 3x3 sub-grids. DrawModes on the other hand provides the layout for user to select game options and game modes, all denoted by their respective choice of characters and representation.

A **SetGridMode** function is deployed to be called to instantiate the DrawModes function and calls upon the user to input the source and difficulty of the random grid that will be solved by the agent. This function has access to SugokuHeroApp puzzles as well a 1,000,000 puzzles local dataset in the form of a CSV file. Various Modes are defined to map the user's keyboard input and perform the respective function calls. The various Modes are described as:

Mode 0 : Clears the grid to hold an empty grid
Mode 1 : Uses API GET request to fetch an EASY grid
Mode 2 : Uses API GET request to fetch a MEDIUM grid
Mode 3 : Uses API GET request to fetch a HARD grid
Mode 4 : Uses API GET request to fetch a RANDOM grid
Mode 5 : Fetch a random grid from ***sudoku.csv*** (1,000,000)

HandleEvents function is used to update events based on the user input. Here, game options and game modes are mapped to keyboard inputs of the player. The keys are described as:

C : Clears Grid – Sets an empty starting grid
E : Easy Grid – Mapped to Mode 1
M : Medium Grid – Mapped to Mode 2
H : Hard Grid – Mapped to Mode 3
R : Random Grid – Mapped to Mode 4
D : Dataset Grid – Mapped to Mode 5
S : Solves the Grid that has been chosen
Q : Quit Game

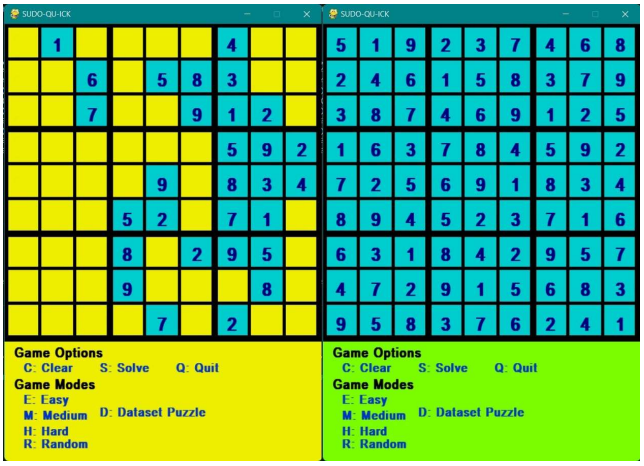
SolveGrid function is the brains behind this game-playing agent as it contains the logic to perform backtracking search in a manner that allows the user to witness the magic. The agent tries all possible values ranging from 1-9 inclusive and checks if the value is valid using the **IsValueValid** function that takes (***gridArray, i, j***) as its parameters to check if that number is valid at the current position. This function is called recursively and a pygame pump event assists in noticing these flickers of forward and backward tracking. **InitializeComponent** and **GameThread** are also used to update our game screens constantly.

RandomGrid function employs pandas libraries to read inputs from CSV datasets. This function then converts our dataset components by integer-location indexing to populate a starting grid with the numbers provided. Zeros in the starting grid are mapped to empty positions that need to be filled.

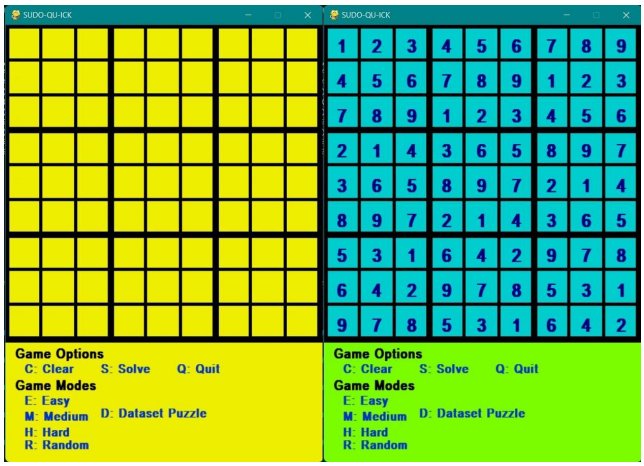
Experiments and Results

```python
# Solving using Backtracking Algorithm
def SolveGrid(gridArray, i, j):
    global IsSolving
    IsSolving = True
    while gridArray[i][j] != 0:  # cell is not empty
        # this loop goes through the entire grid
        if i < 8:
            i += 1
        elif i == 8 and j < 8:  # go back to the first column and next row
            i = 0
            j += 1
        elif i == 8 and j == 8:  # go through all rows and columns
            return True
    pygame.event.pump()  # called once every loop
    for V in range(1, 10):  # trying values from 1->9 inclusive
        if IsValueValid(gridArray, i, j, V):  # if the value is correct, add it to the grid
            gridArray[i][j] = V
            if SolveGrid(gridArray, i, j):  # if the value is correct, keep it
                return True
            else:  # else keep the box empty
                gridArray[i][j] = 0
        screen.fill((124, 252, 0))
        DrawGrid()
        DrawModes()
        pygame.display.update()
    return False
```

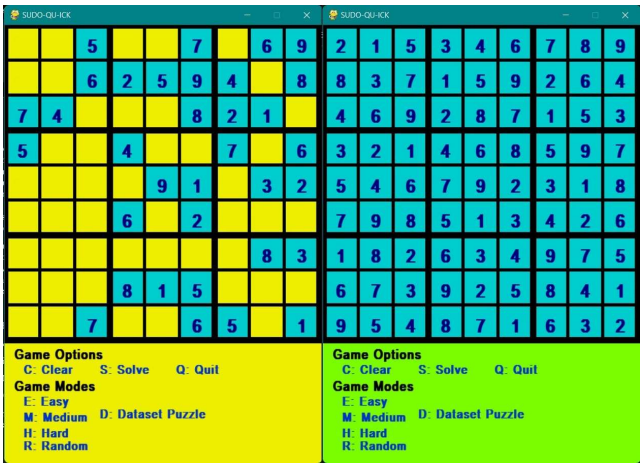
Python Sudoku Backtracking Algorithm



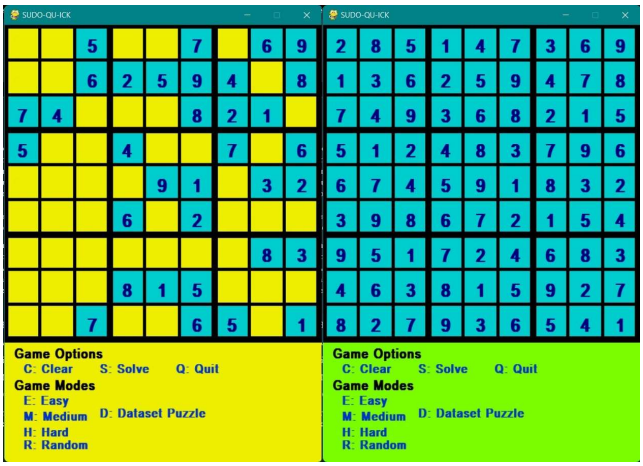
Game Modes : Medium (left) and Solve for Medium grid (right)



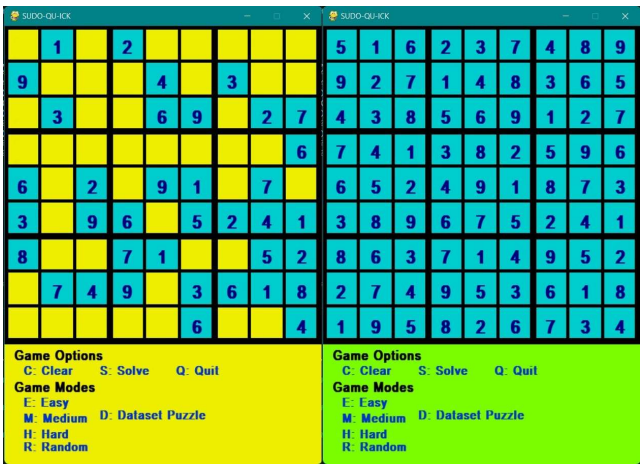
Game States : Clear (left) and Solve for empty grid (right)



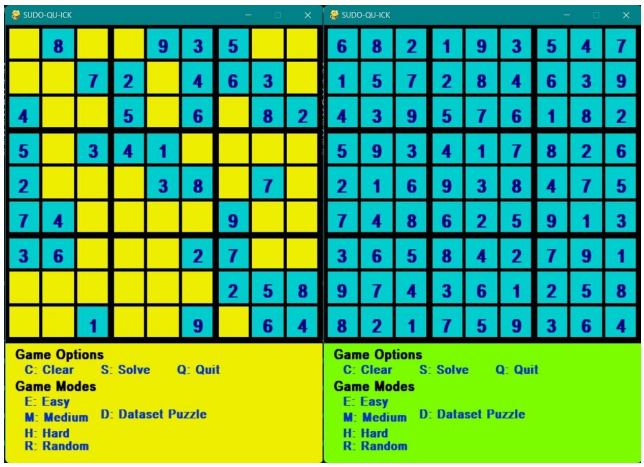
Game Modes : Hard (left) and Solve for Hard grid (right)



Game Modes : Easy (left) and Solve for Easy grid (right)



Game Modes : Random (left) and Solve for Random grid (right)



Game Modes : Dataset Puzzle (left) and Solve for Data (right)

The implementation of the sudoku game-playing agent was completed successfully. The agent was optimized to find solutions quicker and display the procedures without delays. The agent can optimally solve any puzzle that forms the starting grid. The Sudo-Qu-ick agent uses Depth First Search and by exploiting the constraints of not picking numbers that are already present in the sub-grid, row, or column, solves the sudoku grid in seconds!

Conclusion

This final project built on the experience gained in classroom and previous coding assignments to make use of knowledge of search algorithms and python applications to create a game application that involves an agent capable of self and user-exploration and retracing its own steps.

Although the goal of the problem statement was mostly satisfied in the experiment, it is definite that there are various possible steps that could be incorporated to enable a more efficient outcome. This technique is a good way to find a solution quickly and efficiently.

In a short period of time, the suggested algorithm can solve such puzzles of any difficulty level (Easy, Medium, Hard). Certain function optimizations were performed to reduce compute time. Irrelevant of the difficulty of the puzzle, the agent behaves optimally and comparably for all modes of difficulty. Further research in optimizing the search algorithm to employ more constraints or some different approach towards solving sudoku will certainly be of interest.

References

The-Assembly. (2021, January 24). *Code an AI Sudoku solver in Python*. https://www.youtube.com/watch?v=GX4c13SSBrs

The mnist database. MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges. (n.d.). http://yann.lecun.com/exdb/mnist/

Radcliffe. "Exploring the 3M Sudoku Puzzle Dataset." *Kaggle*, Kaggle, 18 Sept. 2020, https://www.kaggle.com/code/radcliffe/exploring-the-3m-sudoku-puzzle-dataset/data.

Sugoku, https://sugoku.herokuapp.com/

Teaching Artificial Intelligence across the Computer Science ... - Aaai.org. https://www.aaai.org/Papers/FLAIRS/2007/Flairs07-066.pdf.

"Building and Visualizing Sudoku Game Using Pygame." *GeeksforGeeks*, 4 Sept. 2021, https://www.geeksforgeeks.org/building-and-visualizing-sudoku-game-using-pygame/.

"A Sudoku Solver Using Python." *One Step! Code*, 9 Dec. 2020, https://onestepcode.com/sudoku-solver-python/.