CS231: Project 3

Sudoku

**Abstract**

In the project, a Sudoku solver has been coded. The current project reads from the test.txt file and prints the solution out to the terminal, the solved board. In the meantime, the GUI window shows how the solver progresses step by step. Then, I have explored how long the board will take to solve when it starts with different numbers of locked values.

The data structure that has been used to implement this solver is a stack. This data structure works on the principle of LIFO which stands for Last In First Out. The stack saves all the cells that the program has attempted to solve with their values and is helpful in easily manipulating the Cells and their values to go through them until a solution is found.

Calendar

Description automatically generated

A solved Sudoku board

**Exploration**

Values for different numbers of starting values are tabulated below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 10 | 20 | 30 | 40 |
| 1 | 5 | 3 | 1 | 2 |
| 2 | 9 | 2 | 0 | 0 |
| 3 | 0 | 0 | 2 | 1 |
| 4 | 1 | 1 | 1 | 0 |
| 5 | 3 | 0 | 0 | 0 |
| 6 | 1 | 2 | 1 | 0 |
| 7 | 1 | 4 | 2 | 0 |
| 8 | 0 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 0 |
| 10 | 2 | 1 | 0 | 1 |
| Average | 2.3 | 1.3 | 0.7 | 0.4 |

Plotting the data, we get:

As such, it is evident that as the number of starting values goes up, the time taken to solve the board goes down.

**Extension 1**

My first extension explores the difference between the two following versions of the *nextBestCell()* method:

1. Return the first cell (row-wise) cell that has a value of 0.
2. Return the first cell with the least number of possible values.

To do this, I have run 10,000 simulations of both methods for 10 initial values. My results are as follows:

**Extension 2**

For this extension, I increased the number of decks to six. This was done by modifying the *build()* function in the *Deck* class to include 24 each of cards with values 2-9 and 11, and 96 cards with the value 10. This makes it six decks in total

In a table below, I have plotted the results (win %) for 5 games and compared it with the results for using just one deck. An average of the five games has been taken to average out and statistical anomalies.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Serial Number | 1 deck | | | 6 decks | | |
| Player | House | Push | Player | House | Push |
| 1 | 41.7 | 49.7 | 8.6 | 41.2 | 49.6 | 9.2 |
| 2 | 42.2 | 47.3 | 10.5 | 41.8 | 48.0 | 10.2 |
| 3 | 38.9 | 52.8 | 8.3 | 41.4 | 49.0 | 9.6 |
| 4 | 40.3 | 51.9 | 7.8 | 42.7 | 49.1 | 8.2 |
| 5 | 43.3 | 48.4 | 8.3 | 44.4 | 45.8 | 9.8 |
| Average | 41.28 | 50.02 | 8.7 | 42.3 | 48.3 | 9.4 |

**Observations**

The findings from the table above are that even though there has been the difference of a few decimal places between the values of the house winning and the game ending in a push, the player wins approximately the same number of times. It can be concluded from the above data that changing the number of decks from 1 to 6 does not have a substantial effect on the chances of winning/losing/drawing this version of Blackjack.

**References/Acknowledgements**

I consulted both, Prof. Harper and Prof. Al Madi to discuss why my solve method might not be working. I also worked with Quan Phan to go over my code and help me find the bug that was causing my display to be rotated by 90 degrees. The *LandscapeDisplay* class was retrieved from <https://cs.colby.edu/aharper/courses/cs231/f21/labs/lab03/LandscapeDisplay.java> .