Andy Porter

Sean Conklin

Dr. Hooper

**CSP-based Sudoku Solver**

The project we chose to embark on was creating a Sudoku solver. To accomplish this goal, we went through several different ideas and versions of code as well as spending many, many hours with the debugger in PyCharm. What we ended up with was a solver which, when passed the correct data, can determine answers based on constraints given.

In order to explain our code and decisions we must first explain the rules of Sudoku. Sudoku is played on a 9x9 grid which is split into 9 subgrids of 9 cells each. If the puzzle is solved correctly, each column of 9, row of 9, and subgrid of 9 will be filled with the values 1-9 without any repeating numbers.

Sudoku is best understood through a visual so, knowing that fact, one of our first goals was to create a functioning GUI to accurately represent what our puzzle was doing. Some of our goals for the GUI were to allow the user to input their own puzzle found online or created by them and let the solver figure it out as well as letting the GUI initialize a randomly selected puzzle from a built-in dictionary and display it for the user to try and solve. The GUI also has an option to bring up the rules of the game and some basic information on how to navigate the GUI. Lastly the GUI had a solve button which would take the current state of the board and generate a dictionary to send to the solver. There were buttons in the GUI to allow the user to interact with the program and the buttons included; New Game, Rules, Hide Rules, Hint, and Solve. Each button was created by using the tkinter.button object and assigned to a variable. All of the buttons were contained within a frame so that they could stay grouped together even if the output window was resized.

Though we had lofty goals for the GUI and didn’t quite meet them, we did end up accomplishing the stated goal of creating an interactive Sudoku solver. The new game button will only return a specifically stated puzzle from our selection of puzzles. We have a method to generate a random number with which to choose which puzzle to play but what we discovered is that the method was being called twice at different points, once when the board was trying to populate initial values, and once when the solver was looking for the current state of the board. Both times the method was called it returned a different puzzle so the GUI and the solver were not looking at the same puzzle. We ran out of time but our next goal was to create a class variable to hold the value of the puzzle and see if the solver would see the class variable. Furthermore, we had included a hint button which would run the solver method but only return the next single best move. Since we had trouble figuring out our solver method until it was too late, we just didn’t have time to debug that method so we took the button off the GUI.

We wanted the solve button to be able to generate a dictionary based on the current entries of the board, and pass that to the solver to figure out. We never figured out how to read the game board and generate a dictionary so the solver method only looked at the initial state of the board which was passed down through the new game method. Furthermore, we anticipated that there would be unexpected problems in which a user could create an unsolvable puzzle, and we didn’t quite know how to throw a case to stop the infinite loop from occurring. However, seeing as our solver method worked for 11 puzzles that we found from various Sudoku sites on the internet, we hypothesize that if a given puzzle is solvable, a user could pass the puzzle into the randPuzzle method we created. To this end, we did not accomplish our goal to allow the GUI to take user input and create a puzzle out of it, but we effectively had created the same implementation via a workaround.

The game board itself was build using a combination of tkinter objects from Python’s built in tkinter library. The black frame around the puzzle is actually a canvas widget colored black with a raised edge. The canvas was then put inside a frame, and that frame is what hold the entire gameboard. On top of the canvas, the grid was created by creating a total of 8 vertical lines, and 8 horizontal lines and increasing the width of vertical and horizontal lines 3 and 6 to create the 3x3 subgrids. Within each of the 81 boxes in the grid an entry widget was placed and assigned to a variable in order to accept user input as well as allow the system to print values. After all of the widgets were created, a dictionary was built with keys of cell number and values of whichever entry widget was within that cell. For instance: {‘A1’: entryA1, ‘A2’: entryA2,...}. Using this dictionary, we could take a dictionary from the output of the solver and map a value to an entry widget by comparing the dictionary keys.

Given the nature of a Sudoku game being to fulfill a set of conditions such that the numbers 1-9 appear only once in each column, row and square, our goal for our corrected solver was to create a Constraint Satisfaction Problem to limit the total possible solutions to the bare minimum that would make sense given the current boardstate. To conceptually understand what the CSP algorithm itself was doing, we revisited our aima-python CSP assignments and rearranged our colored map projects to be an empty Sudoku board, given the following properties of our domain, neighbors, variables and constraints. The domain was defined as each individual Sudoku input box with an assignable variable of 1-9 each, all connected via a neighborhood listing out each individual boxes’ “constraining boxes”, which can be defined as the boxes with its row, column or square. Given the nature of the colored map to only color any connecting values once, we found that we had no need to change the original constraints from this setup. We found that doing this ended up solving the complete Sudoku puzzle, albeit with horrible efficiency as every value for every box was checked and compared to every neighboring box. From this point on, we attempted to create our own CSP code to eliminate repeat values as they showed up while implementing a search function that would allow our CSP to backtrack if it came to a point where no solutions could be found.

Our solver handled information using python dictionaries. In our initial design, the key for each dictionary was a specific box on the 9x9 grid and the value for each key was an int representing the number in that box. If there was no value in the box it was assigned a value of 0. The columns were counted as numbers 1-9 while the rows were counted as letters A-I. In this way, the box on the top left - most side of the 9x9 grid would be referred to as box A1, while the box just to the right of it would be referred to as box A2, and the box just below it would be referred to as box B1. Our initial goal was to create a constraint satisfaction algorithm ourselves which would eliminate values from the possible answers pool of each row, column, and subgrid as each value was assigned. The theory was that since each row, column, and grid of 9 could only have values 1-9 if we gave cell A1 a value of 1 then 1 would have to be eliminated from the list of possible answers for column A, row 1, and the top left grid of 9. This version of the solver at its peak performance was successful until it encountered a box whose corresponding row, column, and grid had no matching values available in the pool; in this case the code would just loop indefinitely. As we discovered later, this version of the code had two big problems. The first problem was the lack of a backtracking method. Without the backtracking method, the program would just get stuck when it ran out of values on the current path it was following. The second problem was run time. Using this version of the solver, even if we had made it perform at its peak would still have to search through three separate dictionaries and lists every time it encountered a new value just so it could eliminate that value from each list and dictionary.

Given the issues we encountered with drafting our own solver, we decided to scrap the idea and figure out a way to implement the CSP code that we found within the aima-python repository that we had been working in throughout this course. Even though we figured that the runtime was slow, we knew the code itself worked with an empty board, so we theorized that we just had to find a way to input our current boardstate as the domain of the puzzle, setting the variables of the keys of the system-generated boxes to the correct value and changing them to write-only so that our solver method would not change them. Through a lot of trial and error, we were able to accomplish exactly this by creating a dictionary of the generated puzzle and merging it with our current boardstate, which updated the boardstate’s domains and usable variables to everything not currently used on the board. This way the solver could read this new dictionary, correctly solve it and output the answer to the GUI.

It was at this point that we finally had a working solver method, so we just figured that we would work on improving efficiency. We ended up creating a method that looked at the boardstate and found potential values for our empty spots by comparing the boxes to every other box in their row, column and 3x3 square before sending it into the solver method. This ended up allowing the code to look at fewer possibilities, which helped our efficiency a good bit. While looking through the CSP code found in aima-python, we had found a commented-out function for utilizing a Minimum Remaining Value heuristic for the backtracking search algorithm that we could choose to implement, which ended up working perfectly. Our solver is now able to solve puzzles at quick speeds, which was much better than the original implementation that took a couple of minutes for an empty puzzle.

In conclusion, though we feel that we have accomplished our original goal of creating a Sudoku solver ad that is ready for submission, we still want to work out the kinks that we have discovered in our code to polish up our project in our own free time. To this end, our next logical step is finding a way to re-initialize the boardstate such that generating a new game will set our dictionaries back to their original states before generating a new board so that we don’t have to close out the application and re-run it to get it to solve a different puzzle. Furthermore, we hope to eventually develop a way to have the GUI take user input, click a button to set that as the boardstate and then attempt to solve it from there. To accomplish this, we feel that we would also have to come up with some way to check and see if a puzzle is solvable in the first place to where it will either return the fully solved puzzle or an error message telling the user that the boardstate they created is illogical. Lastly, we want to work out the kink where our code generates two instances of our application, which we can only hypothesize being due to a hidden method call somewhere in our code that generates an additional instance of the app.

**Sources:**

1. <http://effbot.org/tkinterbook/entry.htm#Tkinter.Entry.get-method>
2. <http://sebsauvage.net/python/gui/#import>
3. <http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/dictionaries.html#definition-and-use-of-dictionaries>
4. <http://www.python-course.eu/tkinter_canvas.php>
5. <https://docs.python.org/3/contents.html>
6. <https://docs.python.org/2/tutorial/classes.html>
7. <http://www.247sudoku.com/sudokuEasy.php>
8. <http://norvig.com/sudoku.html?HN_repost=true>
9. Norvig, Peter, and Stuart Russell. *Artificial Intelligence: A Modern Approach*(Chapter 6). S.l.: Pearson Education Limited, 2013. Web.