Ali Noor & Christian Simons
CSCI-3202
AI Practicum Write-up
May 1st, 2019

The problem that will be solved for the practicum is the game of sudoku. Sudoku is a puzzle where the player inserts a number between 1 and 9 in a partially complete grid of 9 squares which is also subdivided into 9 smaller squares. The constraints of the game consists of not repeating a number in a vertical line, horizontal line, and the subdivision that the number is placed or exists in.

We will solve sudoku using the constraint satisfaction problem algorithm. *CSPs are composed of variables with possible values which fall into ranges known as domains. Constraints between the variables must be satisfied in order for constraint-satisfaction problems to be solved[1].* There will be 5 key components to solving sudoku which will consist of the variables, domain, constraints, worlds, and models. The variables will represent the empty squares in the game and the domain will be the range of values from 1 to 9. The world will be a possible combination of values that can take up a given section, while the model will be the final version of the worlds in which all constraints are satisfied. There will be 3 constraints, the first being no 3x3 grid should have the same number existing within it, the second being no horizontal row should have duplicates, and the third also should have no duplicates in its column.

We will be using a sudoku game generator to create the game boards that we will be testing[2].In order to find the optimal solution we will be using backtracking to build a potential candidate for a given solution, while abandoning whatever path that cannot find a valid solution. *Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree)[3].* To search the tree we will be using depth first search to evaluate what node is failing the constraints.

Once the failing node is found, it will be pruned and that will reduce the size of our search space. A stack is used to run the depth first search. This is done by trying every possible value in the domain that satisfies the given constraints. We first will begin with the smallest value in that domain and continue filling in the empty cells, until we run into a failed cell and that's when

the process of backtracking and pruning begins. As we iterate over our domain list, we ensure we look forward and pick the next smallest value, while discarding our previous value.

Some key classes we have set up is the *sudokuBoard* and the *agentCSP*. The *sudokuBoard* class is used to generate the game board each time a new value is set in a cell and it is also used to call helper functions such as *getRows*(), *getCols*(), *getGrid*(), and so on. The *agentCSP* class performs important checks to satisfy the constraints. This includes a *isGoalState*() function which checks if the board is filled, which can only happen if the constraints are not violated and a *getDomain*() function which takes in a single row, column, or cell as a parameter and returns a list of the possible values that do not violate the constraints.

The implementation was a failure. We ran into a logical problem that we could not implement through code properly, as well as some coding errors which were minor compared to our logic issue. Our code would first find an empty cell (which would be the highest up and left-most cell) and call the *domain function* (a function that finds what values can go into a cell given the constraints of the game of sudoku) to see what possible values can be placed into the empty cell. We used a *min function* to pick the smallest value in the domain to start with. Once the value was placed in the cell, that number would be added to our stack frame.

The code would continue to run finding the closest empty cell to its right (finding an empty cell from left to right and top to bottom). This process would repeat on the given empty cell until it reached a domain that had a length of 0 and it failed the *isGoalState* function. When this flag is triggered the backtracking process begins where the current cell that is being looked at would be popped from the stack frame and the next possible domain, if any, would be placed into that cell and into the frame. The code continues to look at the next empty cell and fill in a potential value given the domain until the board is complete.

The logical issue we had was a case when we backtracked. For example if we backtracked and there was a value occupying that cell, lets call it 7 and the domain of that cell was {7, 8, 9}, the next possible value to put in would be 8. Once the 8 is placed into that cell, the stack has a memory of the previous value that was used. So the stack has a tuple of [value occupying the cell, (last number used)] or [8, (7)] for this example. As the code continues and backtracks again to the same position, it picks the next value which is 9 and keeps track of

the 8 that was previously occupying the cell. Code continues and backtracks to the same position again and picks the value 7, even though it picked that value before. We get stuck in an infinite loop of these same 3 numbers.

We know in order to solve it we need a list of the numbers that we have used and we should iterate over that to see if we used it before. We're just having a hard time implementing that logic into our code. Since our original implementation plan failed, we've decided to take a different approach to the problem. In this new implementation plan we search the game board for a cell that has the most constraints satisfied, thus the smallest domain which would be of length 1. If the domain is of length 1 then that value will be placed into the empty cell. The code continues to search for an empty cell that has a domain of length 1 and fills in the game board as it goes until the entire board is filled. Once the board is filled a check is made if all the constraints are satisfied and once that passes the game is completed successfully. Below you can see how easy this implementation was since it only took about 6 lines to do everything.

```
314
315         def searchCSP(self, game):
316                 while not (self.isGoalState(game)):
317                         posX, posY = self.findMostConstrainedSpace(game)
318                         val = self.getDomain(posX, posY, game)
319                         game.setNewGridValue(posX, posY, val[0])
320
321                 self.printBoard(game)
322                 return game
323
```

We also knew we were successful because of how little time it took to run and solve the game. One of the games we tested only took 0.04 seconds to solve!

```
(base) C:\Users\chris\OneDrive\Documents\CSCI3202 Practicum>python SudokuSolver.py ./sudoku.sdk
['5', '3', '4', '6', '7', '8', '9', '1', '2']
['6', '7', '2', '1', '9', '5', '3', '4', '8']
['1', '9', '8', '3', '4', '2', '5', '6', '7']
['8', '5', '9', '7', '6', '1', '4', '2', '3']
['4', '2', '6', '8', '5', '3', '7', '9', '1']
['7', '1', '3', '9', '2', '4', '8', '5', '6']
['9', '6', '1', '5', '3', '7', '2', '8', '4']
['2', '8', '7', '4', '1', '9', '6', '3', '5']
['3', '4', '5', '2', '8', '6', '1', '7', '9']
Time to Solve (in seconds): 0.043301

(base) C:\Users\chris\OneDrive\Documents\CSCI3202 Practicum>
```

This was a great project to really learn and hammer down the many ways a constraint satisfaction problem can be solved. We had three different implementations that we have tested and failed one of them, discarded another for being too complex, and the third we passed. The implementation that we failed we discussed in our original plan. The complex implementation that we attempted had us making a tree of every possible game state, and then traversing through that until the *isGoalState* flag passed. The implementation that passed was the search the game board for a cell that has the smallest domain approach and fill in the numbers as you go. After many hours put into this practicum there is no portion of the class we understand more than constraint satisfaction problems.

References:

1. Anon. Constraint-Satisfaction Problems in Python. Retrieved May 2, 2019 from https://freecontent.manning.com/constraint-satisfaction-problems-in-python/
2. Anon. Python Sudoku. Retrieved May 2, 2019 from https://sourceforge.net/projects/pythonsudoku/
3. Anon. Backtracking Algorithms. Retrieved May 2, 2019 from https://www.geeksforgeeks.org/backtracking-algorithms/