Project Motivation

Sudoku is a well-known logic-based puzzle game that has extensively been used as an example of a problem that can be solved by modelling it as a constraint satisfaction problem (CSP) and utilizing Backtracking Search and Constraint Propagation. In our project, we have modelled a variant of Sudoku, called Samurai Sudoku, as a CSP and have again utilized Backtracking Search and Constraint Propagation to solve each puzzle. We chose Samurai Sudoku because, while there has been an extensive analysis of the standard Sudoku puzzle, Samurai Sudoku has, as far as we are aware, not been analyzed in much rigor. Furthermore, we were interested in determining how the overlapping Sudoku corners in Samurai Sudoku, affected both the modelling of the puzzle as a CSP and the inherent properties of the game itself. Specifically, we wanted to see the effect that initially placed values had on the solvability of the puzzle.

Background

- **Sudoku**

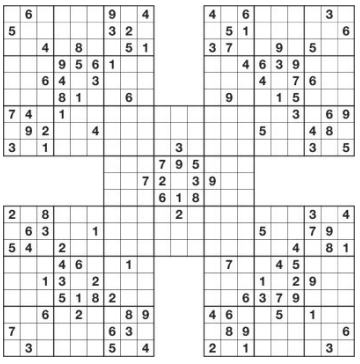
Sudoku is a logic-based number placement puzzle, it consists of 81 cells which are divided into 9 columns, rows and regions. Each region is a 3 by 3 grid.
The object of a Sudoku game, is to fill up the 9 by 9 grid with numbers such that each column, row and region contains all of the digits from 1 to 9; while the same number cannot be appear twice in the same column, row or region.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Example of Sudoku

- **Samurai Sudoku**

Samurai Sudoku consists of 5 overlapping Sudokus; one at the top left, top right, middle, bottom left and bottom right of the game, and the middle Sudoku has its corner regions overlapped with the other 4 Sudokus.
Samurai Sudoku follows the same rules as Sudoku; except, at the region where two Sudokus are overlapped, the row and columns do not go beyond their usual length, but the interlocking regions provides more restriction to the game.



Example of Samurai Sudoku

Methods

The approach we used to solve a given samurai sudoku, given the initial grid configuration was to formulate it as a constraint satisfaction problem, and solve it using backtracking search. To generate the constraints for the CSP, every cell in the sudoku was given a label, with an alphabetical character denoting the row in which the cell is found, a number denoting the column, and an identifier character representing which of the 5 sudokus the cell is located in. For instance, the cells in the top row of the top right sudoku would be labeled ‘A1b’ through ‘A9b’. Here, ‘b’ denotes the top right sudoku grid. The identifier characters given were ‘a’ for the top left sudoku, ‘b’ for the top right, ‘c’ for the bottom left, ‘d’ for the bottom right, and ‘+’ for the remaining squares in the middle sudoku that are not shared with one of the other 4. Thus, while there are 81 cells labeled with each of ‘a’ through ‘d’, there are only 45 cells labeled ‘+’ (81 - 4*9 overlapping cells).

Every label was stored as the key of a dictionary, with the corresponding value a string representing the domain of possible values the cells could take on, initially 1-9. Two more dictionaries were used, also taking the labels as keys. One stored every constraint (called units in the code) of which said label was a member. The other stored all the peers of said label (peers being other cells that share a constraint one another). Being that all constraints are of the ALL-DIFF variety, checking for validity is as simple as checking that a cell does not contain the same value as any of its peers.

Having formulated the CSP and generated all the constraints, all that remains is to solve the samurai sudoku. The solver has 2 conditions on which it will assign a value to a cell. Either a cell has only one value remaining in its domain, or there exists only one location in a unit that can accept a particular value. In either of these situations, the value of the cell is set, and the change is propagated by removing the assigned value from the domains of all the peers of the cell. Sometimes this propagation and setting of values is sufficient to solve the sudoku by itself, but if this is not the case, backtracking search is employed, using the MRV heuristic.

Evaluation and Results

In our evaluation we begin by determining the correctness of the solver against hand-made example puzzles. Once complete, we continue to evaluate correctness by feeding our solver puzzles generated by a random Samurai Sudoku puzzle generator[1]. The solutions outputted in each of these steps are checked for correctness via a brute force checker algorithm.

In total we crafted 8 hand-made example puzzles, both solvable and unsolvable, and have generated 10,000 random puzzles to check for correctness. In each case when the solver finds a solution, the solution is deemed correct by the checker. The checker algorithm is a brute force algorithm that simply goes over each row, column and region of the solved puzzle to see if each ALL-DIFF constraint is satisfied. Notably, the solver timed out on 16 of the 10,000 puzzles generated, these puzzles were then checked by hand and found to be unsolvable.

Next, we looked into how many randomly generated Samurai Sudoku puzzles are solvable. Assuming the solver's correctness. We generated thousands of puzzles with varying degrees of initially assigned squares.[2] The results showed that puzzles with a larger number of *valid initial constraints*[3] were increasingly likely to be unsolvable. However, puzzles with less constraints placed in the center Sudoku were likely to be solvable, even if the outer four Sudoku's had a large number of valid initial constraints.

In order to determine which Samurai Sudoku puzzles were solvable we started collecting data on which squares of a solvable Samurai Sudoku were initially assigned. We aggregated the initially assigned squares of 1500 solvable randomly generated puzzles.[4] From this data we made the heat maps displayed in Figure 1 on the following page.

[1] Programming an algorithm that fairly generated these puzzles proved to be a task as difficult as the CSP itself.
[2] Evaluations can be found in the `eval/` folder.
[3] An initially valid square value is one that satisfies all ALL-DIFF constraints at the start of the puzzle
[4] Data can be found in the `data/` folder

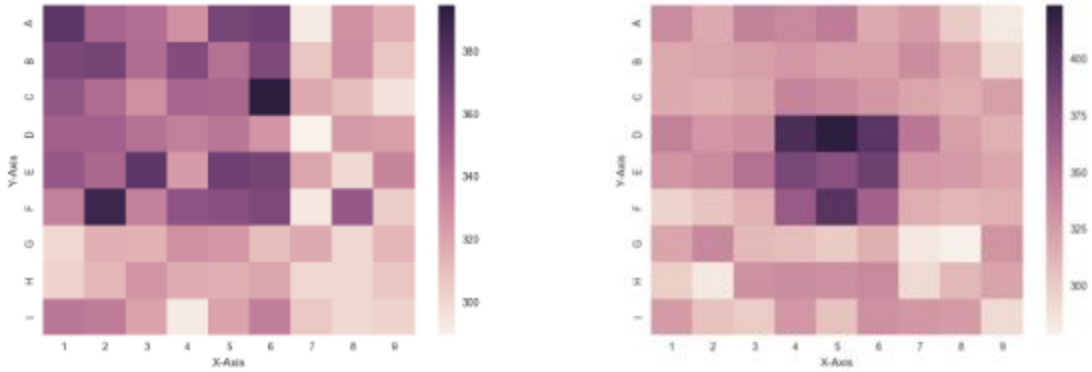


Figure 1. Left: Heatmap of top left Sudoku[5], Right: Heatmap of center Sudoku

The heat maps suggest that most solvable Samurai Sudoku puzzles contain initially assigned valid square values in the darker purple area of the heat maps. Furthermore, initially assigned square values that greatly constrain the overlapping Sudoku regions greatly increase the likelihood of an unsolvable puzzle. This claim follows from darker areas of the heatmap least constraining the overlapping Sudoku regions.

Limitations/Obstacles

Initially we sought to use the starter code provided for Assignment 2 as a base for the samurai sudoku solver. Sudokus, however, represent a far more complex CSP model, with 27 ALL-DIFF constraints each containing 9 variables. Furthermore, Samurai sudokus, are represented by almost 5 times as many constraints. From this we knew that Assignment 2's starter solution's time and space inefficient method of storing and solving CSPs may prove unusable. Nevertheless, having both forward-checking and GAC constraint propagation built into the code made it worthwhile to attempt a solution. Unfortunately, the 'valid tuples' approach used to check constraints results in 362,880 (9!) combinations per constraint, rendering the approach far too slow.

To remedy this, we considered two options. The first was to start from scratch and devise an entirely new method of storing and checking constraints, or to find starter code to use elsewhere. In the end, knowing that the original sudoku is a solved problem with solutions

[5] Heatmaps for each of the outer Sudoku's look similar to the top left Sudoku heat map shown in Figure 1 and can be found in the `heatmaps/` folder

and source code available online, we chose to find a sudoku solver that used constraint propagation and backtracking search, and adapt it to solving the Samurai Sudoku. In this way, the starter code gives us a basis for solving our problem using concepts learned in the course, and allows us to experiment with different ways of representing and solving CSPs.[6] Note that the starter code is written to be compatible with Python 2.5, and so was extensively modified to run on python 3.4. The starter code was then extensively modified to be adapted to solving Samurai Sudoku puzzles instead of the original Sudoku puzzle.

Conclusions

As this report has emphasized, our modelling of Samurai Sudoku as a constraint satisfaction problem has largely been successful. The algorithm has been able to correctly solve all 8 hand-made test cases, and determine which test cases are unsolvable. Furthermore, the algorithm is able to correctly solve almost 10,000 randomly generated Samurai Sudoku's in under 5 minutes. In terms of time efficiently we have deemed this to be satisfactory. However, notably, 16 puzzles took the solver longer than the allotted 10 seconds to solve and were subsequently timed-out before our algorithm could complete, given more time we would thoroughly investigate why these puzzles proved so difficult for the solver. Perhaps, our extensive use of recursion made our algorithm needlessly space inefficient. Exploring an iterative option may be a better alternative.

Finally, our use of our solver to explore the mathematical properties of Samurai Sudoku has proven both fruitful and novel. We have produced evidence indicating that valid initial square values that directly constrain overlapping Samurai Sudoku region's increase the likelihood of creating an unsolvable puzzle. It has recently been discovered that 17 is the lowest number of initially placed squares that can create a Sudoku puzzle with a unique solution.[7] Perhaps, in a further report, researchers could identify the lowest number of initially placed squares that creates a Samurai Sudoku with a unique solution by utilizing the properties discovered in this report.

[6] The starter code we used is available here: http://norvig.com/sudopy.shtml and is also included with our submission.

[7] http://phys.org/news/2012-01-mathematicians-minimum-sudoku-solution-problem.html

Citations and References

While all direct citations have been attributed in the footnotes, we have used various sources as references throughout this assignment. These sources include:

For references of the rules and strategies of Samurai Sudoku Puzzle:

- http://www.samurai-sudoku.com/
- http://www.scanraid.com/BasicStrategies.htm
- http://www.sudokudragon.com/sudokustrategy.htm
- http://www.krazydad.com/blog/2005/09/29/an-index-of-sudoku-strategies/
- http://www2.warwick.ac.uk/fac/sci/moac/currentstudents/peter_cock/python/sudoku/

For help modelling Samurai Solution as a CSP, and help in visualizing the heat maps shown in this report, as well as various other smaller python related issues:

- http://norvig.com/sudopy.shtml
- https://stanford.edu/~mwaskom/software/seaborn/
- Various StackOverflow Questions and Answers for help with general python inquiries