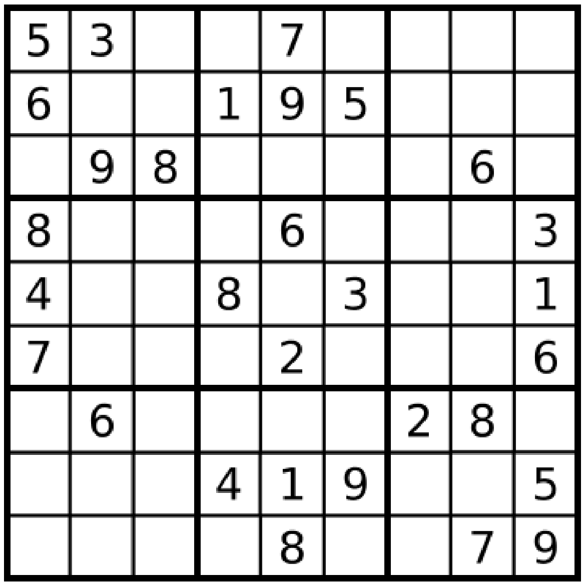
CS 440: Project 1
A Sudoku Puzzle Solving Agent

Jun Jiang, Priya Narayana Subramanian, Peng Wang, Ri Zhang

Abstract

This paper discusses solving and generating Sudoku puzzles with evolutionary algorithms. The most natural formal description of a Sudoku puzzle is to express it as a constraint satisfaction problem. The board presents the user with a number of cells, some already preassigned, and asks to fill in symbols from a finite domain such that each puzzle segment (row, column, or block) does not contain twice the same symbol. Backtracking and Constraint Propagation algorithms have been utilized to solve the problem. The naive approach to solve a Sudoku is to try out potential candidate entries for the cells; if a conflict is detected, we backtrack and try out a different value. There also exists a number of constraint propagation techniques which can solve most Sudokus without performing any search. In this paper we report on the implementation of a simple Backtracking solver, a Pure CSP solver, and two versions of CSP solver, that integrates standard backtracking search with three different specialized constraint propagation techniques for Sudoku. We study the puzzle from a constraint point of view, show models to solve and generate puzzles and give an objective measure of the difficulty of a puzzle instance. The time and memory performance of each method has been simulated and compared in this study.



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

Figure 1: A 9x9 Sudoku Puzzle

Introduction

Sudoku is a puzzle invented by the American Howard Garns in 1979 at IBM, and popularized in Japan during the 80s, before rising to world fame in the early 2000s. The game consists of a grid of 9x9 cells, divided in 9 boxes of size 3x3, which have to be filled with numbers from 1 to 9. The constraints by which the numbers can be placed (hence the English name for Sudoku, Number Place) are as follows:

- Each column, row, and box must contain the numbers 1 to 9.
- No column, row, or box may contain the same number multiple times.

Usually a Sudoku puzzle has a unique solution, which is achieved by pre-filling some of the cells (see fig- ure 1 for a possible starting layout); with these givens and the constraints, the other cells have to be filled.

Sudoku puzzles that appear in books or newspapers always have exact one unique solution. Even an 9x9 Sudoku grid can have a large number of different combinations. According to Felgenhauer and Jarvis, the number of possible 9 by 9 Sudoku grids is N=6670903752021072936960 which is approximately 6.6711021.Various algorithms have been applied to solve this combinatorial problem. In this study, we will try four different approaches and compare them. According to the textbook, Artificial Intelligence- A Modern Approach, S. Russell, P. Norvig; the introduced environment is considered to be: Discrete, Fully Observable, Static, Deterministic, Sequential, and Single- Multi agent.

1. Backtracking

The naive solution method for constraint satisfaction problems is search with backtracking. One can obtain a sound and complete solving algorithm by picking unassigned variables and assigning them a value from their domain, and repeats the procedure recursively, moving through the search tree. If a square has no possible values, then return to the previously assigned square and change its value. This method of search is known as a backtracking search. It is guaranteed to find a solution if there is one, simply because it will eventually try every possible number in every possible location. This algorithm is very effective for size two puzzles. Unfortunately for size three puzzles there are nine possibilities for each square. This means that there are roughly 981n possible states that might need to be searched, where n is number of given values. Obviously this version of backtracking search is not going to work for size 3 puzzles. Fortunately there are several means by which this algorithm can be improved: constraint propagation, forward checking and choosing most constrained value first.

2. Specialized CSP

Sudoku can be considered as a typical Constraint Satisfaction Problem (CSP) with the following formulation of variables, domains and constraints.

- **Variables:** 81 variables, one for each square;
- **Domains:** The empty squares have the domain 1,2,3,4,5,6,7,8,9 and the prefilled squares have a domain consisting of a single value.
- **Constraints:** There are in total 27 different constraints: one for each row, column, and box of 9 squares. These constraints ensure each row, each column, and each box contains all of the digits from 1 to 9, i.e. no digit appears twice in a so called **unit**.

Similar to the textbook, we use the variable names *11* through *19* for the top row from left to right, down to *91* through *99* for the bottom row. In addition to **units**, we also define **peers** of a specific square to be those squares that are in the same units with it. Every square has 3 **units** and 20 **peers**. For example, the square *32* has 3 units – (31, 32, 33, 34, 35, 36, 37, 38, 39), (12, 22, 32, 42, 52, 62, 72, 82, 92) and (11, 12, 13, 21, 22, 23, 31, 32, 33) and 20 peers – (11, 12, 13, 21, 22, 23, 31, 33, 34, 35, 36, 37, 38, 39, 42, 52, 62, 72, 82, 92).

As opposed to searching, constraint propagation prunes the domains of the variables and assigns values to them by mere reasoning, by using various procedures to enforce consistency within constraints. Sudoku puzzles are usually solvable by pure constraint propagation, by adding some specializations. We now describe the three techniques used in our implementation and experiments.

1. Only choice rule: If the possible values of a square has been reduced to one, we eliminate that value from its peers' remaining values.
2. Only square rule: In a unit, if there is only one square for a value, we put the value into this square and do propagation to its peers according to the above strategy.
3. Naked Twin exclusion rule: This rule is a little bit more advanced. We look for two squares in the same unit with the same two possible digits. For example, given {'26':'34','27':'34'}, we can easily conclude that the numbers 3 and 4 must be in the square 26 and 27. Then we can eliminate 3 and 4 from every other square in this row unit.

During our research, we also found other more sophisticated strategies, such as *Two out of three rule*, *Sub-group exclusion rule* and *General permutation rule*.

a) Two out of three rule: Also referred as 'slicing and slotting'. The rule is to take groups of three rows and columns in turn, looking for all the 1s then all the 2s, 3s etc. During the process of scanning rows and columns in groups of three, if you find two out of three where the number being scanned has been allocated, then the missing number can only be put in one of three squares in the left one row or column.

b) Sub-group exclusion rule: A **sub-group** is used to describe three squares in a row or column that intersect a Sudoku **unit**. Every row and column has three sub-groups in the three units it crosses. We will not cover details here.

c) General permutation rule: The Naked Twin exclusion rule is actually a particular example of this one. We can try also Naked Triplet exclusion rule to look for three squares in the same unit that have the same three possible digits and do constraint propagation likewise.

Due to time limit, we weren't able to implement these strategies.

3. Optimized CSP

We optimized the Specialized CSP by doing the following.

- **Initial round of elimination** Not start with all possible values for all the cells. For example, if the input contains a set of cells with pre-existing values, we immediately use that to remove the value from the peer cells.

- **Hidden single elimination** When all elimination is done, we identify and remove hidden singles i.e., cases where each units can have a particular value in only one position. In this case, even if the cell has multiple data possibilities, we assign the hidden single value and move on.

- **Minimum Remaining Values** The minimum remaining values heuristic is used to alter the order in which squares are guessed in order to reduce the number of branches at each level. Basically instead of choosing the first empty square, the square with the least number of possible values is chosen. The heuristic can allow us to find the correct digit for that square with the best chance. For instance, starting with a square with only two possible digits gives us 50% chance to guess the right number, bigger than the possibility to start with a square with 5 possibilities with 20% chance to make the correct guess. In value ordering, we don't apply a specific policy. Instead we just try all possible digits in their numeric order.

4. CSP with search

The specialized and the optimized constraint solvers, are able to solve all puzzles with easy and medium level of difficulties. In order to solve puzzles with even more difficult levels such as hard and evil we need to apply search algorithms, in particular, depth-first search to complete the algorithm. A human player solves the puzzle by using simple techniques. If the puzzle is not solvable by using the techniques the player then tries to fill the rest of the empty squares by guessing.
We applied constraint propagation algorithm on the Sudoku instance at the initialization and then write down the values which are the only options for the given case. After updating the Sudoku puzzle we applied Backtracking algorithm on the resulting Sudoku problem. In this way, most of the time we did not need to run Backtracking since after applying Constraint Propagation at the initialization the problem is totally solved. The search is implemented using the *minimum remaining values* heuristic.

Evaluation

Following figures show the runtime of each algorithm for various levels of difficulty. We used a set of approximately 50 easy sudoku problems, 100 hard sudoku problems and about a couple of dozen of hardest problems. We could not get runtimes for hard and hardest problem using backtracking as they took forever to run. Here are the results and observations.

CSP with heuristic is better in performance compared to the specialized CSP, both of them, however are not complete algorithms. When an algorithm returns without finding a solution, we mark the runtime as -1. we can see that CSP without DFS is often incomplete. use of heuristiccs make it better but not by a lot. Backtracking guarantees a result but is often costlier.

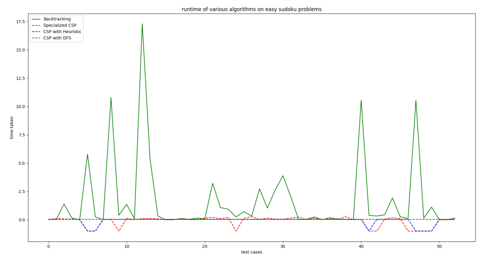


Figure 2: Comparison of time consumed by each algorithm for easy puzzles

When it comes to hard problems, backtracking does not scale at all. we could not include the results due to the long running nature of the general brute force method. As we can see here, CSP and CSP with heuristic often fail to find a goal state. CSP combined with DFS produces consistent and complete results.

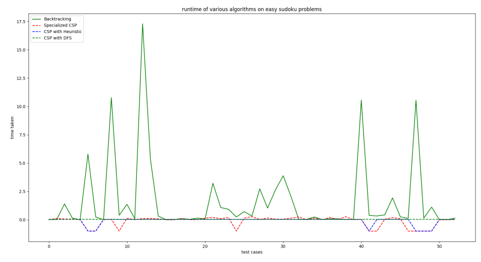


Figure 3: Comparison of time consumed by each algorithm for easy puzzles

When we move the difficulty level up a notch, the inability of CSPs to produce results are much more visible in the absence of an accompanying DFS approach. However, CSP combined with DFS seems to be doing reasonably well.

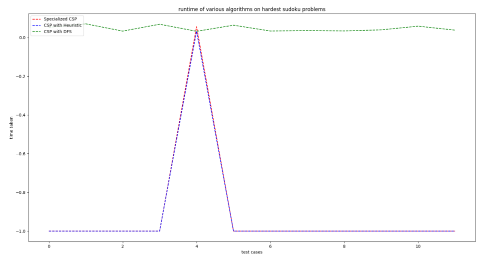


Figure 4: Comparison of time consumed by each algorithm for hardest puzzles

Conclusions

In this work we investigated the use of CSP solving techniques for Sudoku. This study has shown that the CSP with Search algorithm is a feasible method to solve any Sudoku puzzle. The CSP with Search algorithm is also an appropriate method to find a solution faster and more efficient compared to the Pure CSP and Optimised CSP algorithm. The CSP with Search algorithm is able to solve such puzzles with any level of difficulties in a short period of time. The testing results have revealed that the performance of the CSP with Search algorithm is better than the Pure, Optimized CSP algorithms with respect to the computing time to solve any puzzle.

Future Work

We have implemented the Sudoku solver for larger puzzles (e.g., n = 4 or n = 5) by using alpha-numeric characters for digits. Further research needs to be carried out in order to optimize the algorithms. A possible way could be applying backtracking to the Optimised CSP algorithm. Other alternatives might be to establish whether it is feasible to implement an parallelized sudoku solving algorithm.