Sudoku Fighter

AI Project

Marzieh AshrafiAmiri
Negin Shariat
Amir Hosein Afandizadeh Zargari

December 2019

Table of Contents

Table of Figures

Table of Tables

1. Describe Problem

Classical Sudoku refers to a puzzle in which a player starts to insert numbers between 1 and 9 in a grid space. The space defined for the play is a 9*9 grid, subdivided into 9 smaller squares with the size of 9.

There is only one simple rule in a sudoku puzzle. All numbers in the same row, the same column or the same small square should be unique. Meaning that each number between 1 and 9 should be present in one of the grids existing in a row, column or small square.

Solving sudoku means finding the correct placing of numbers for the grid space which is consistent with each other and the initial settings of the grid space. This puzzle can be expressed as a constraint satisfaction problem (CSP). The set of implicit constraints can be written as follows:

$$\begin{cases} All\ diff\ in\ each\ row \\ All\ diff\ in\ each\ column \\ All\ diff\ in\ each\ small\ square \end{cases}$$

In this project, we are trying to create an agent that can solve different sudoku puzzles. The existing difference between our project and classical sudoku is that we are having a grid space 25*25. Hence, it needs 25 distinct numbers for each row, column, and a small square. The numbers used in this project as values cover 0 to 24. Since sudoku is a CSP, we have to define variables and domains in our project as well as values. Each cell in the grid space is represented as the variables for a CSP and domains are like values which are an integer between 0 and 24. In the following sections, you can read about our approach, algorithms, and heuristics used in the project and the proposed models used for evaluation.

2. Describe the Approach

Solving sudoku can be a bit tricky. The easiest solution that comes to mind is using a backtracking algorithm meaning that traversing all the possible states that can be made with the grid space. This algorithm is like Depth First Search (DFS) in which one child is generated at a time for each value of a variable. Eventually, it will iterate over the domain values of that variable. If no legal values left, it will backtrack to the variable's parent and check another value for that. Among these possible states, of course, most of them do violate the existing constraints. So why should we traverse them to the endpoint?

As a starting point of improving the backtrack search, it is important to find an algorithm for finding the order of variables (cells) to be assigned. This order can be fixed in time; so, at the first moment, you will run the algorithm to find the order and you will use it to end. But this approach is like a blind search that won't seem to work fine. Hence, it is better to have a wiser one which can be considered as an informed search. In this project, two famous heuristic algorithms are used for variable selecting, Minimum Remaining Values (MRV) and Degree Heuristic (DH). Further information about these heuristics can be found in sections 3.1 and 3.2.

Now that the problem of selecting one variable among others has been solved, it is important to pay attention to the values. A selected variable could have multiple values for the assignment. Therefore, an algorithm should exist to find the value to select. A naive method can be assigning the values in increasing order. Since we are dealing with integer numbers, we could pick the minimum number first. In the case of dead-end, we could choose the next minimum. Obviously, this method is not wise enough to solve 25*25 sudoku. The famous heuristic we used in this project is the Least Constraining Value (LCV). You can find details of this heuristic in section 3.3.

The heuristics mentioned above will try to solve a sudoku puzzle but they are not enough. Due to the existence of exponential state space search for the sudoku puzzle, it is not feasible to iterate nearly all possible values. Hence, we have to try to reduce the domain space for cells. The heuristics mentioned above will try to reduce the state-space search, leading to a sooner solution. Due to the existence of exponential state space search for the sudoku puzzle, pruning some branches might not result in a noticeable effect in 25*25 sudoku.

One of the useful improvements can be a wise combination of forward checking and backtracking with heuristics mentioned above. So, whenever a value is chosen for the variable, named variable X (which is selected by MRV or DH) by a heuristic like LCV, you can start pruning that value from the domains of all other variables which have constraints with the assigned variable X. If the assigned value results in zero choices for one of the other variables, the value assigned in the first place for variable X was incorrect. Hence, backtracking to the variable X has to be done in this step. By backtracking, the value will be removed temporarily from possible choices (domains) of variable X and another value will be assigned. This procedure will take place until a solution is found (desired state, goal state) or no value remains for variable X. In the case of empty domains for variable X, another level of backtracking is needed. So that we could find another value for the assignment of the variable before X.

But there still exist some problems with forward checking for sudoku of size 25*25. It won’t reduce the state space as needed. Therefore a much powerful reduce strategy should be taken

into consideration. The algorithm we used is arc consistency which has a detailed explanation in section 3.4.

To improve the performance of the combined structure, in this project we try to use some other methods for pruning domain for each cell based on others. These methods are also useful when solving sudoku examples in the real world. The main concept of these methods is assigning a value to some variable. This assignment will be a sure and true assignment to a variable, based on the previous assigned values for other variables. We have implemented three different update methods which are explained in sections 3.5 and 3.6.

3. Define Algorithms and Heuristics

The first three following sections, refer to heuristics used in Sudoku Fighter for selecting a variable among all existing variables and assign a value among the domain of the variable to it. Section 3.4 is trying to define the constraint propagation which is based on forward checking as a weaker representation of arc consistency. The arc consistency algorithm will be defined in section 4.1. The three remaining sections describe the update functions which are highly used in the real world when solving sudoku puzzles.

3.1. Minimum Remaining Value

This heuristic is used for choosing the next variable to assign in search. It tries to select the variable most probable to fail soon, so it can prune more branches in the search process. The method is to determine the number of the domain for all unassigned variables and choose the one with the least number. In the case of a tie, it is assumed this algorithm will select the variable in upper rows and more left columns.

Table 1 - Properties of MRV

Input	Output
The domain of all variables	Variable selected in the current level of search

3.2. Degree Heuristic

This heuristic is also used for variable selection. For each variable V, this method counts the number of variables connected to variable V. As the output of the function, it will return the

variable that has the biggest number of connected variables. In the case of a tie, it is assumed this algorithm will select the variable in upper rows and lefter columns.

Table 2 - Properties of DH

Input	Output
The domain of all variables	Variable selected in the current level of search

3.3. Least Constraining Value

This heuristic is used for choosing the next value for the given variable. The functionality of this algorithm can be divided into these 6 steps:

1. Gets a variable V
2. Check the variable's domain and choose any value N
3. Consider domains of all the variables that have constraint with variable V
4. Among those domains finds the minimum domain size (with excluding N from them) (Min)
5. Does step 3 and 4 for all values on variable V 's domain and find Min for each value
6. Select the value with maximum Min number

Table 3 - Properties of LCV

Input	Output
The domain of all variables One selected variable	A value from the domain of the input variable

3.4. Constraint Propagation using Forward Checking

This algorithm is used for reducing the state space search. By assigning a value to a variable, it is obvious that the particular value should be removed from the domain of all variables which have constraint with the first assigned variable. In the other words, choosing a value V for a variable "Var" will result in eliminating that value V for the variables which are in the same row, the same column, and the same small square as variable Var.
We will go through all the cells (variables) in the grid space. If it was assigned to one variable, we will check the domains for other variables that have constraints with the assigned one. We will continue iterating over all variables until nothing is changed in one step.

Table 4 - Properties of Constraint Propagation

Input	Output
The domain of all variables	Reduced domain for all variables

3.5. Update by Single Value

This punning method uses the fact that all numbers (0 to 24) should be used exactly one time in each constraint groups (each row, each column, and each small square). It checks the domains of all variables in a constraint group and if one value only exists in the domain of one variable, named “v” in that group, it assigns that value to the variable “V” that only has the possibility to be assigned to this value (ignoring how many other values that variable can take). This helps the assignment of some variables without using the search process.

Table 5 - Properties of Update by Single Value

Input	Output
The domain of all variables	A reduced form of the domain

3.6. Update by 2 Variables with 2 Values

When there are two variables in one constraint group (each row, each column, each square) that have exactly 2 same possible values, it means that those 2 values have to be assigned to these variables and they can be deleted from the domain of other variables in that constraint group. It is a more complicated form of "Update by single value". As an e.g. if D11={2,3} , D18={2,3} , we can delete value 2 and 3 from all D1i where i∈ [0,24] and i ≠ 1 or 8 (not the same variable).

Table 6 - Properties of Update by 2 Variables with 2 Values

Input	Output
The domain of all variables	A reduced form of the domain

4. Experimental Setup and Results

In the previous section, we have defined several functions that may help us to get the result faster during the backtracking search. We can divide these functions into three different classes which is illustrated in ..... . We put Minimum Remaining Value and Degree Heuristic in the first class, Least Constraining Value in the second class, and also Update by Single Value, Update by 2 Variables with 2 Values, and Constraint propagation in the third class.

In every life cycle of each search in the backtracking search, we need to use at least one function of each class. Functions in the first class are used to pick the variable which has a higher probability to fail in search. In the second class, we want to assign a value to the variable that was picked in the first class which has a lower probability to fail in search.

Now we can guess if we use only the first and the second class it is still hard to find the solution in reasonable time. So we can apply the functions of the third class to reduce domain space before and also during the search.

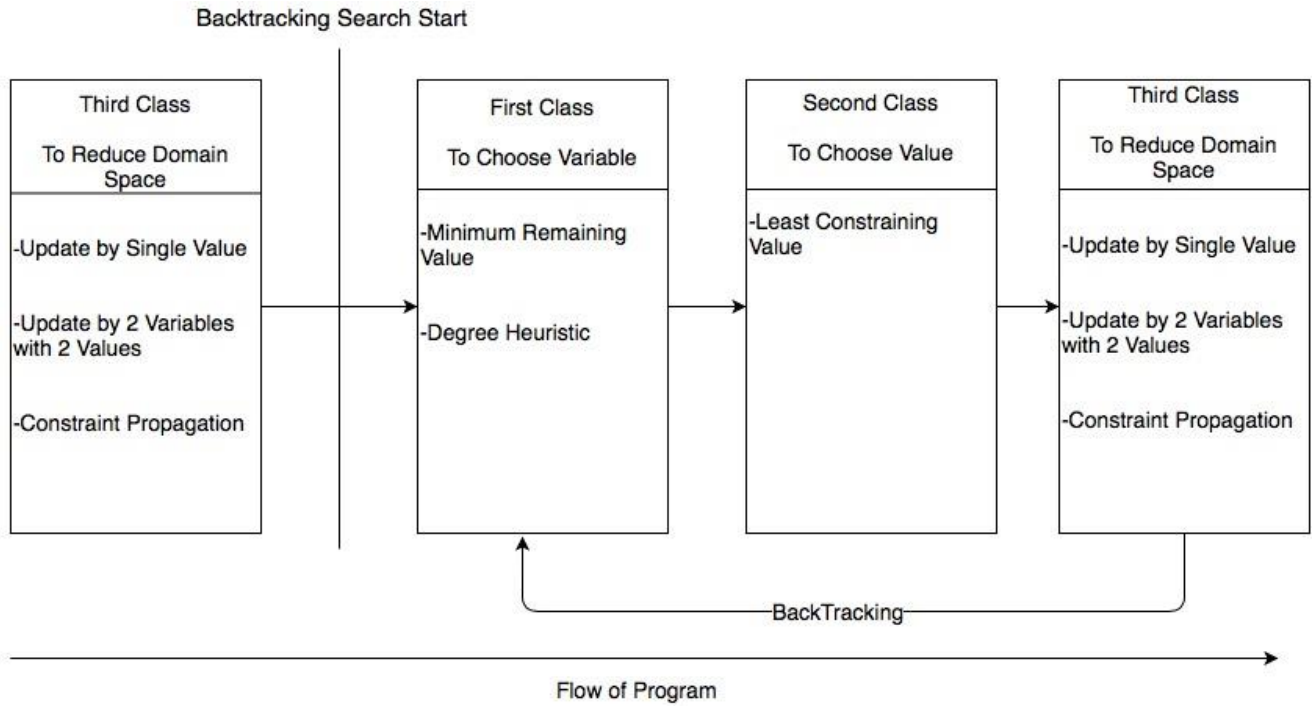

Figure 1 - Different Functions in Classes

To obtain a fast agent for solving sudoku puzzles, there are some ambiguities that are necessary to be answered about choosing at least a function in each class. Which heuristic should we use to pick a variable? Does it worth adding a third class to decrease domain space although it increases some computation penalty during the search?

We tried to compare heuristic functions and domain pruning functions with the test cases which released in canvas for 16*16 sudoku. In the following sections, first, we will describe the mixture of all pruning (update) algorithms mentioned in the section 4.1. Also, we will show the fantastic effect of this mixture for the 16*16 test cases. After that, MRV and DH are being compared using the same test cases.

4.1. Pruning Mixture

There were different methods explained in sections 3.5 and 3.6 for pruning each of which with some overhead of running the method and some improvement in the run time due to the pruning. These algorithms are supposed to provide arc consistency for the sudoku puzzle. The important factor is how we can combine these methods to get maximum improvement. Our proposed way is illustrated in *Figure 2*. First of all, we will use the constraint propagation method to update the domains of all variables based on each other. Second of all, "Update by Single Value" will be called. As a result of this function call, we will have a smaller state-space search. In the end, "Update by 2 Variables with 2 Values" will be used to eliminate domain based on equal domains of 2 variables. These three steps will be continuously running until the domain will remain fixed. In other words, if we consider the domain which is the input of these steps as D1 and consider the domain which is the output after running these three steps as D2, we will stop pruning mixture if D1 is equal to D2.

Now that we find the best mixture for pruning, it is time to decide where to put the pruning. The easy solution is to use this pruning once before search. But the more efficient way is to call the pruning before and during the search. Using it during the search is tricky. Because, we will first assign a value to a variable, then use pruning and if it was a dead-end we have to understand that the value assigned in the first place was wrong. Due to soundness of pruning all of the pruning will become false and we have to go back to the domain before pruning. Results for finding the place of pruning is shown in *Table 7*. It is good to know that for these test cases, we use MRV heuristic in the search.

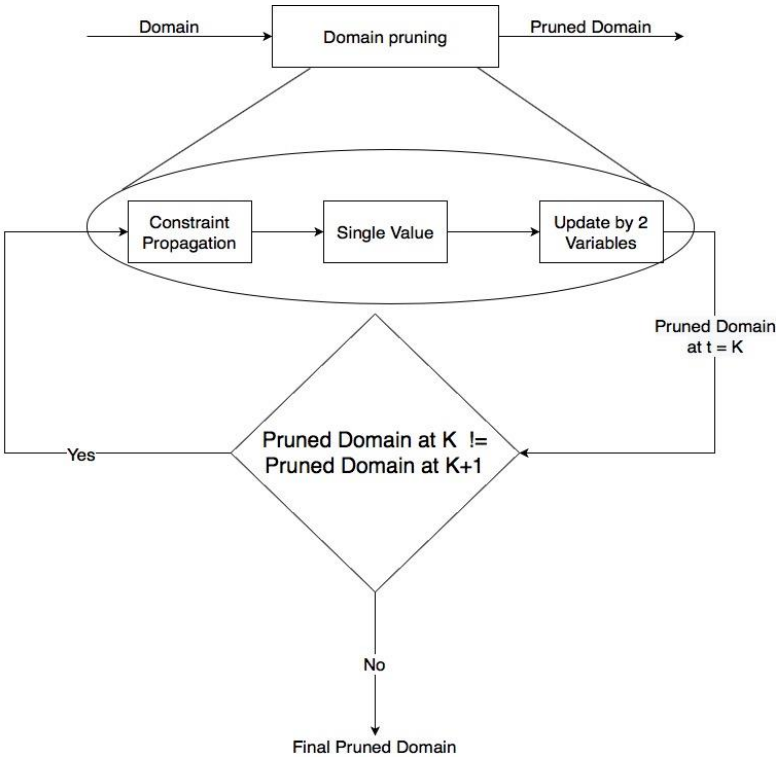

Figure 2 - Pruning Mixture Structure

Table 7 - Results for Finding the Place of Pruning Mixture

	Without Pruning	Pruning Before Search	Pruning Before and During Search
Hexa-81.txt	No answer in 5 minutes	3.380 sec	1.358 sec
Hexa-82.txt	No answer in 5 minutes	10.469 sec	7.202 sec
Hexa-101.txt	1.027 sec	3.390 sec	18.557 sec

In general examples and a few test cases that we showed, pruning before and during search improves the performance of search. But in some rare cases like the “hexa-101.txt” test case, you can see that without using pruning the performance is better. This is the place that the tradeoff shows the influence meaning that the complexity that pruning will add to the running time is more than the effect of pruning during search.

4.2. MRV or DH? Big Question…

Two famous heuristics are explained in sections 3.1 and 3.2. Let’s see how they will perform on different test cases. We have to pay attention that in these test cases for different algorithms, based on the previous result of pruning mixture, the pruning is called before and during search.

Table 8 - MRV vs. DH

	MRV	DH
Hexa-81.txt	1.358 sec	7.470 sec
Hexa-82.txt	7.202 sec	136.856 sec
Hexa-101.txt	18.557 sec	No answer in 5 minutes

4.3. The best Algorithm

Based on the result in Table 7 and Table 8, it is obvious that the best heuristic to use is the MRV algorithm; due to its better solution in every test case. For the perfect place of pruning, we propose using it before the search to start the search with fewer domain space and also during the search. Although it can add complexity, the improvement in reducing the domain space can be more effective. Hence, it will result in achieving a solution sooner.

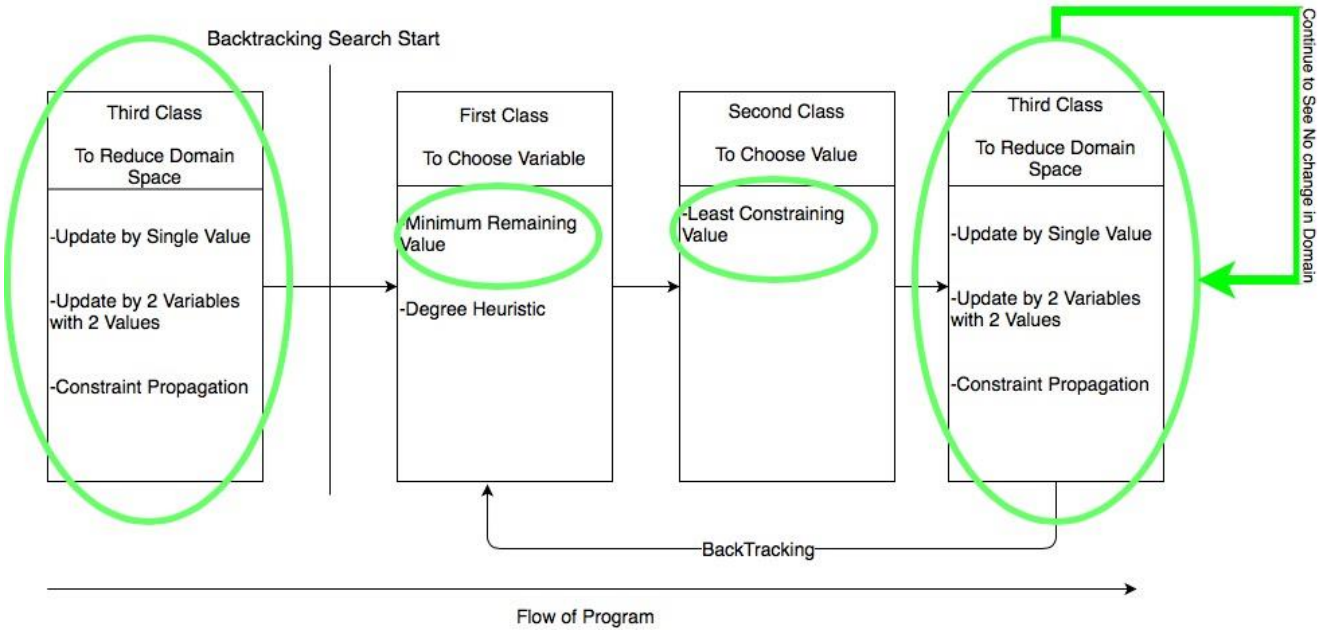

Figure 3 - The Flow of Proposed Method

Reference

For this project, we only used the lectures of class to fully understand the concepts of Constraint Satisfaction Problem (CSP), Backtracking Search, Heuristic Methods (Most Remaining Values, Degree Heuristic, and Least Constraining Value), Forward Checking, and Arc Consistency. All the algorithms provided for different updates were totally gained of our real-life experiments of solving sudoku problems.

Appendix

For testing the code, please check the instructions in README.txt. After having an input file just like the one that was mentioned in the project, the output will be shown in your terminal as Figure *1*. The output in the figure shows the solution. As you can see, there is a float number after the table which shows running time of the program.

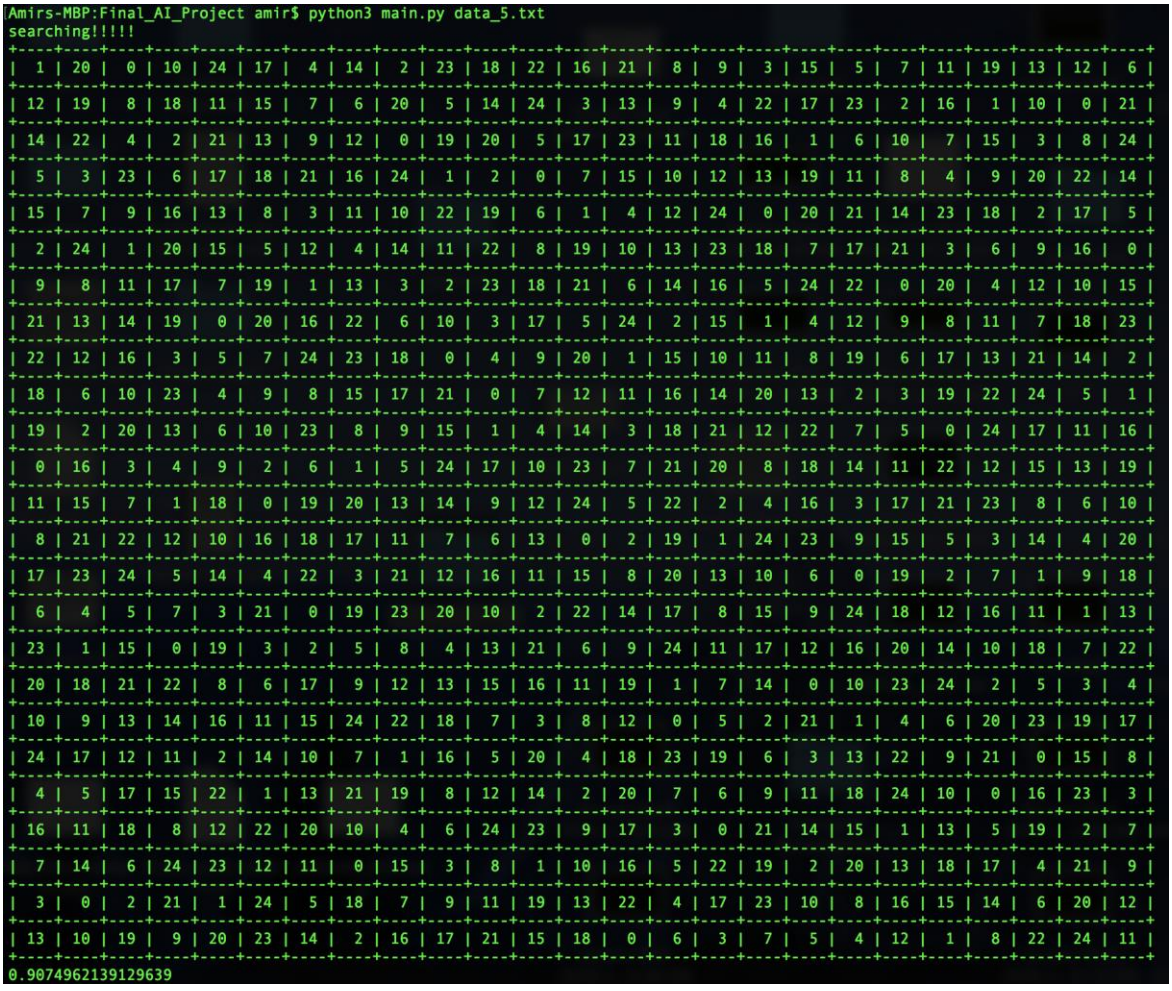

Figure 4 - Output Sample

Based on the proposed method for the Sudoku Fighter, the benchmarks were tested and the results are shown in the *Table 9*.

Table 9 - Benchmark Results

Benchmark	Time
Sudoku_1_1	No result in one hour (7725.790 sec)
Sudoku_1_2	No result in one hour
Sudoku_2_1	1132.983 sec
Sudoku_2_2	245.402 sec
Sudoku_3_1	266.124 sec
Sudoku_3_2	241.797 sec
Sudoku_3_3	251.133 sec
Sudoku_3_4	505.061 sec
Sudoku_3_5	264.073 sec
Sudoku_3_6	219.728 sec