Introduction

Sudoku is a game that requires many constraints to be taken into account all at once. Given all these constraints, a strategy of trial-and-error can be used until all of the constraints are satisfied. This leaves the problem to be solved well by a CSP method. The number of trial-and-error iterations can be minimized to reduce the runtime of the solution by optimizing, for example, which grid space should next be assigned.

In this report, we will be exploring optimizing the CSP method for Sudoku by recognizing patterns that lead to unsolvable grids. That is, recognizing grid cell assignments (termed "no-goods") that invariably lead to an eventual violation of the constraints of the problem. Recognizing these no-goods early in the trial-and-error process should save a significant amount of time by preventing redundant further iterations.

Methods

In order to add no-good checking to the Sudoku solver, we will obviously need to track some state of the problem when an unsolvable grid is encountered. On encountering an unsolvable grid, we can track the locations of each assigned grid space and the value of each one so that we do not continue down the same path in the future. These three pieces of information can be stored in the `Q` array of already assigned spaces, if we append the value to the end of the tuple. We can add unsolvable `Q` arrays to a different array, for example `bad_assignments`, to ensure we do not waste time reproducing any of these in the future.

The tracking can be done inside of the `search()` method, since this is where grid spaces are assigned and we check whether the grid is solved. Before we start assigning values to a grid space, we can loop through the `bad_assignments` array and make sure that each member is not contained within the current `Q` array. If it is, we will eventually run into a grid cell that cannot be assigned a value.

When adding a new `Q` array into `bad_assignments`, we can delete all of the items in `bad_assignments` that will be contained within the new `Q` array, since it will be redundant to check those in the future, as that information is included in the appended `Q` array.

Results

The results between the default CSP method, and the method that detects no-good grid states can be seen below in Figure (1). The figure seems to show that in general, for the no-good learning algorithm using FA (first available heuristic) grid cell selection, the runtimes are lower since the points are more concentrated around the origin along the y-axis. The figure also shows that some MRV (smallest domain heuristic) runtimes are higher for the no-good learning algorithm, since the points are more spread out along the x-axis.

The means and standard deviations for both the default and no-good learning algorithms can be found in Table (1). The table shows the same data as Figure (1); for FA, no-good learning improves the runtime and for MRV, no-good learning worsens the runtime.

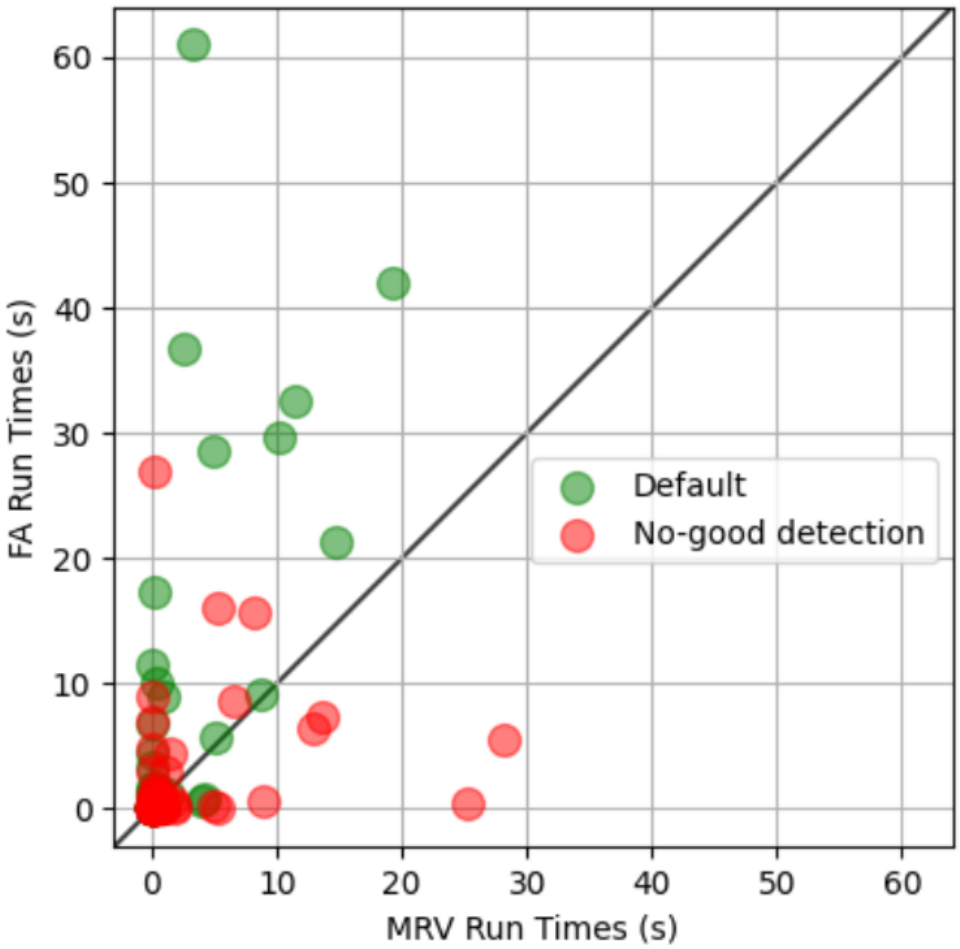

Figure 1 The runtimes for solving 95 Sudoku problems using either a default CSP algorithm or using a CSP algorithm that detects no-goods. Both algorithms either use FA (first available heuristic) selection or MRV (smallest domain heuristic).

	Default (s)	No-good learning (s)	p-value
FA	3.71 ± 10.04	1.41 ± 3.89	0.42
MRV	1.06 ± 3.07	1.51 ± 4.40	0.53

Table 1 The mean and standard deviations for runtimes for solving 95 Sudoku problems using either a default CSP algorithm or using a CSP algorithm that detects no-goods. Both algorithms either use FA (first available heuristic) selection or MRV (smallest domain heuristic).

Using a two-sample z-test between the FA runtimes of the two conditions from Table (1), we find:

$$z = \frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}} = \frac{3.71 - 1.41}{\sqrt{10.04^2 + 3.89^2}} = 0.21,$$

where μ is the mean and σ is the standard deviation of each group. This corresponds to a p-value of 0.42 using a z-table. This means that if there was no true difference between the default and no-good learning FA runtimes, there is a 42% chance we could see the difference in the table just by randomness of the runtimes. Likewise, for the MRV runtime comparison, the p-value is 0.53. Since the p-value threshold for establishing a statistically significant difference between groups is $p < 0.05$, we cannot say for certain that there is a statistical difference between any of the groups, since all $p \geq 0.05$.

Even though there might not be a definitive statistical difference in the data, the data appears to show that no-good learning is beneficial when using FA selection and detrimental when using MRV selection. This is most likely because MRV selection uses a heuristic that minimizes the number of recursions that will occur in the future, and so MRV selection will not waste as much time as FA if it is going down a path that is a dead end. Therefore, in some cases, it could be that checking to see if the current grid leads to something unsolvable in each `search()` call could just slow down MRV rather than speeding it up.

In the future, these results can be improved by implementing more test cases and removing outliers. This would help to reduce the standard deviations seen in Table (1), so that we can establish a certain statistical difference between the runtimes of the default CSP algorithm and the no-good detecting algorithm. Perhaps we could also try a different method of no-good detecting that may be more beneficial for the MRV selection heuristic.

References

1. https://www.z-table.com/
2. http://www.stat.ucla.edu/~cochran/stat10/winter/lectures/lect21.html
3. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.legend.html