INFO-H410 - Techniques of Artificial Intelligence

Lemahieu Antoine, Muller Noëmie, Tribel Pascal

I. Introduction

The Sudoku is a grid number-based game invented in 1979. It has ever since been a standard in computer game solving and lots of artificial intelligence implementations have been done [1] [2]. This game consists in a grid of $d * d$ (traditionally, $9 * 9$), where each cell can contain one number between 1 and d (1 and 9 in its traditional form), and where every row, column and $\sqrt{d} * \sqrt{d}$ square can contain each number exactly once.

We can define an higher level game, where the goal is not only to fill the entire board according to the constraints but also to try the fewest guesses to solve the game.

In computer sciences, the traditional approach (besides the brute force one) is using backtracking. This method does not take into account any *order of exploration* of the cells. This method hence does not try to reduce the number of attempts and therefore, we need an improvement of the technique. This is where the $A*$ algorithm takes place. In this method, the *order of exploration* is determined by a certain *heuristic* that we need to define.

We defined four different heuristics:

Basic: the simplest one, the heuristic of a cell is defined by the number of remaining possibilities.

Small crossed: the heuristic of a cell is equal to the sum of the remaining possibilities for the row, the column and the square the cell belongs to

Big Crossed: the heuristic of a cell is the sum of the remaining possibilities for the three rows and the three columns starting from the square the cell belongs to (cf. figure 1)

Weighted: the heuristic is equal to the sum of the remaining possibilities, weighted according to a given weight graph, that we determine in a prior learning process (cf. section II-C).

After having presented our implementation of the $A*$ algorithm and the four different heuristics, we will detail its execution, and then benchmark the four methods.

II. Implementation

A. A* algorithm

1) Sudoku solving as graph exploration

Since A* is a graph traversal and path search algorithm, we have to represent (at least, theoretically) a Sudoku grid as a graph in order to apply this kind of algorithm to solve it. We can see a Sudoku as a graph where each node represents a partially valid grid, and each vertex between two nodes is the insertion of a number in a cell.

The statement of the Sudoku determines the initial node where our search starts. The aim of the search is to reach a node representing a full grid, respecting one important rule in the travel: as the initially filled cells cannot be changed, the vertices in the graph are directional (they only mean *filling an empty cell* and never *emptying a pre-filled cell*).

2) Search algorithm

Finding a solution of the Sudoku (one travel in the graph from the initial node to a destination) is a thing, but we want this travel to be as short as possible. A* is a search algorithm using an heuristic to improve the efficiency of the searching process. The heuristics we used are described below in section II-B, but let's first take a look to the algorithm A* applied on the Sudoku problem. The pseudo-code shown in this section is recursive, and one could see A* is an improvement of the backtracking algorithm.

Algorithm 1: Algorithm A*

```
if board is not full then
    compute the possibilities for each cell;
    compute the heuristic for each cell;
    get one of the cells with the lowest heuristic;
    for each possible number for this cell do
        fill this cell with the current number;
        execute the A* algorithm on the board
          newly obtained;
        if the recursive call returns True then
            return True;
        end
    end
    if the recursive call returns False for all
      possible numbers then
        backtrack;
        return False;
    end
end
return True;
```

The heuristic of each cell depends on the chosen heuristic method.

B. Heuristics

In this section we describe the different heuristics we implemented for the A* algorithm. Let's first define some important concepts regarding heuristic. First, we define a matrix H where every element $H_{i,j}$ denotes the heuristic associated with the cell (i, j). We also use a function $V(i, j)$, a function giving the set of possible numbers to fill the cell (i, j).

We decide to explore first the cells with the lowest non-null heuristic.

1) Mode 0 : Basic heuristic

The basic heuristic we have defined is as simple as:

$$H_{i,j} = |V(i, j)| \tag{1}$$

This means that the heuristic associated with the cell (i, j) is basically the number of remaining possibilities for this cell. In other terms, in this mode, the algorithm explores first the cells having the fewest possibilities.

2) Mode 1 : Small crossed heuristic

The second heuristic we introduce is the sum of the remaining possibilities of the cells in the line i, the column j, and in the square the cell belongs to. In the following formula, the % denotes the modulo operator.

$$\begin{aligned} H'_{i,j} &= \sum_{k=0}^{k=9} |V(i,k)| + |V(k,j)| \\ &+ \sum_{m=(i-(i\%3))}^{m=(i-(i\%3))+3} (\sum_{n=(j-(j\%3))}^{n=(j-(j\%3))+3} |V(m,n)|) \end{aligned} \tag{2}$$

3) Mode 2 : Big crossed heuristic

The third heuristic is pretty similar to the second one excepted we don't only consider the line i and the column j of the cell, but all the squares $3 * 3$ crossed by the line i and the column j. We call all these squares the "Big cross". For example, in the figure 1, the heuristic of the cell (i, j) in blue will take into account all the cells in red. All the red cells are what we call the "Big cross".

This can be computed by the formula :

$$\begin{aligned} H''_{i,j} = &\sum_{m=(i-(i\%3))}^{m=(i-(i\%3))+3} (\sum_{n=0}^{n=9} |V(m,n)|) \\ &+ \sum_{m=0}^{m=9} (\sum_{n=(j\%3)}^{n=(j\%3)+3} |V(m,n)|) \\ &+ \sum_{k=0}^{k=9} |V(i,k)| + |V(k,j)| \end{aligned} \tag{3}$$

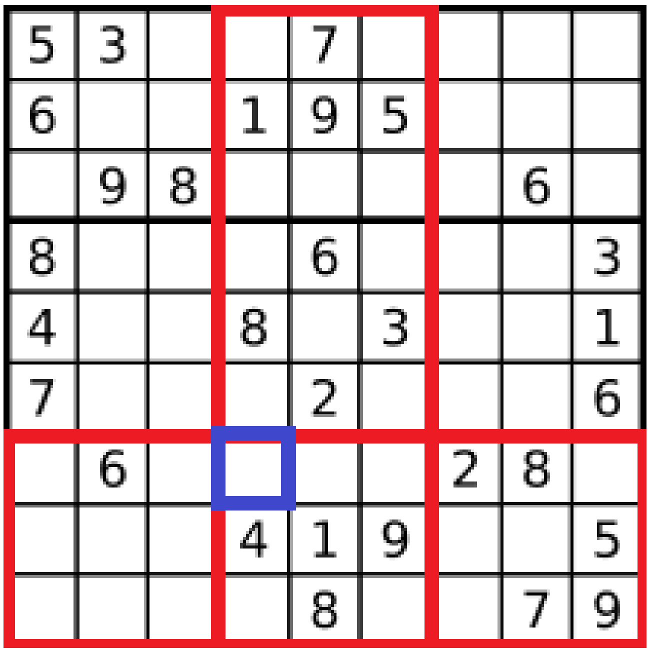


Figure 1. Example of a Sudoku with heuristic of the "Big cross"

4) Mode 3 : Weighted heuristic

The previous section introduced the notion of *Big Cross*. This notion comes from an idea of quantity of information underlying each cell. We assume that there is a link between the filling of a cell and the diminution of the number of possibilities of all other cells in the board.

In order to formalize this idea, we introduce, for each cell x of the board, the notion of **entropy** S. We can define the **entropy** S as the quantity of information brought to any other cell y when filling a cell x. This **entropy** S is determined by the average variation of heuristic of each cell when successfully filling any other cell.

$$S_{x,y} = \lim_{m \to +\infty} \frac{\sum_{n=0}^{n=m} H_x^{n-1} - H_x^n}{m} \tag{4}$$

Of course, this formula depends on the computation of the n^{th} heuristic H_x^n and its previous value $(n-1)^{th}$ heuristic H_x^{n-1}. Those value depends on the method used in the training process, which can be one of the three method previously presented.

Then, when those entropies are estimated for every pair of cells, we introduce our fourth method.

a) Graph

We define a **graph** W of dimension n^4 for n being the size of the Sudokus. In this graph, the node (i, j, k, l) defines the entropy between the cell (i, j) and the cell (k, l).

Finally, to compute the heuristics using this fourth method, we use the following formula:

$$H'''_{i,j} = \sum_{k=0}^{k=n} \sum_{l=0}^{l=n} \frac{V(k,l)}{W_{i,j,k,l}} \quad (5)$$

One way to understand this formula is to see the heuristic of a cell as the quantity of change of information its filling would bring to the rest of the board, according to the interest of its filling would bring to all the other cells.

5) Threshold

While the first heuristic only takes into account the number of remaining possibilities for each cell, the three other methods bring additional information which can cause noise when there are only a few options left for a given cell. That's why we use a threshold τ for the three last heuristic computations. In our model, $\tau = 3$. This threshold determines from how many remaining possibilities we take into account the additional information in order to compute the heuristic. When the number of remaining possibilities is strictly smaller than the threshold, we ignore the additional information in order to prevent diluting the most valuable cells in the additional brought information.

C. Learning phase for weights generation

For the use of the fourth mode explained in section II-B4, all the weights must be computed following the algorithm 2. For this, we generated 10.000 Sudoku and gave them to the 3 first modes for the generation of the weights, then we trained again the graphs three different times on the last mode, based on the previously generated entropies.

In the algorithm 2, we assume that the A^* algorithm has found a solution, and that we are going up in the recursion. $H_{i,j,k,l}$ denotes the heuristics of each cell (k,l) before the cell (i,j) is filled, and $H'_{i,j,k,l}$ denotes the heuristics of the cell (k,l) once the cell (i,j) has been filled. The function $G(x)$ creates a graph (see II-B4a) of dimension x filled with 0's.

III. Dataset

We have trained and tested our different algorithms on 10.000 randomly generated Sudokus, with an average number of holes of $67,3\%$.

IV. Results

A. Learning of the weights

For the *Weighted heuristics* method, we have to determine the entropies as explained in section II-B4. We have defined six different graphs generated with the different other heuristics :

- One graph averaged on the changing of entropy in mode 0, see II-B1
- One graph averaged on the changing of entropy in mode 1, see II-B2

Algorithm 2: Learning

```
$N = G(n^4)$;
$M = G(n^4)$;
for each resolved Sudoku do
    for every filled cell (i, j) do
        for every cell (k, l) do
            $d = H_{i,j,k,l} - H'_{i,j,k,l}$;
            $N_{k,l} = N_{k,l} + d$;
            $M_{k,l} = M_{k,l} + 1$;
        end
    end
    for every filled cell (i, j) do
        for every cell (k, l) do
            $N_{k,l} = \frac{N_{k,l}}{M_{k,l}}$;
        end
    end
end
```

- One graph averaged on the changing of entropy in mode 2, see II-B3
- Three graphs averaged on the changing of entropy in mode 3, see II-B4, based on the three just generated graphs by the three first method.

The figures 2, 3, 4 and 5 shows the weights attributed to the vertices of the graph of each mode at the end of the training phase on 10.000 Sudokus.

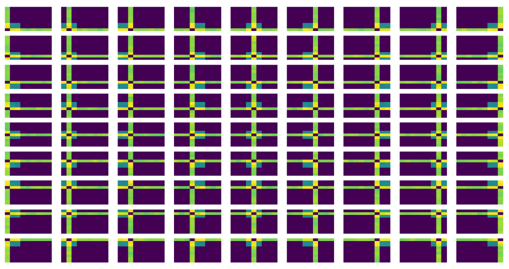


Figure 2. Plot of the weights at the end of the training phase for mode 0

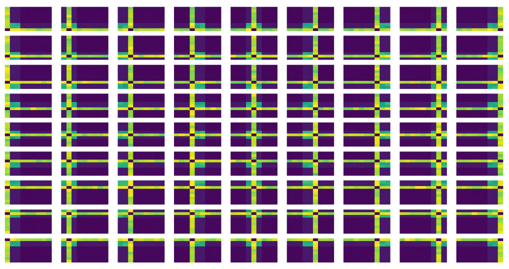


Figure 3. Plot of the weights at the end of the training phase for mode 1

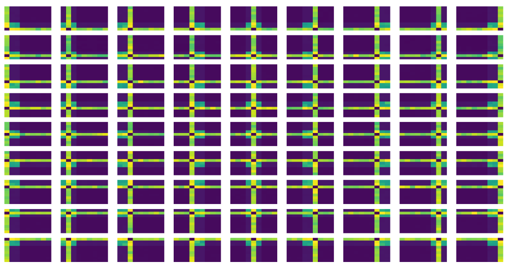
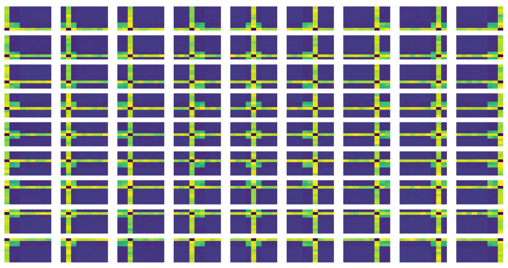
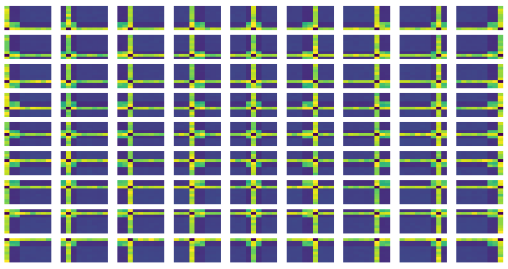


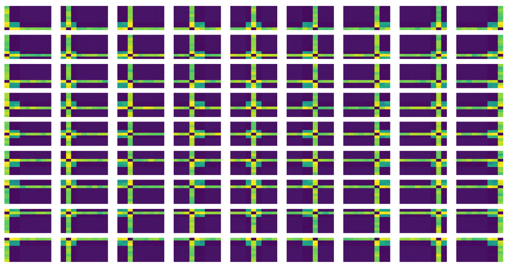
Figure 4. Plot of the weights at the end of the training phase for mode 2



(a) Mode 3 on weights from mode 0



(b) Mode 3 on weights from mode 1

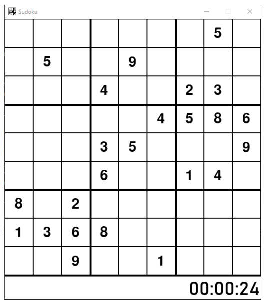


(c) Mode 3 on weights from mode 2

Figure 5. Plot of the weights at the end of the training phase for mode 3

B. *Print screens of the execution*

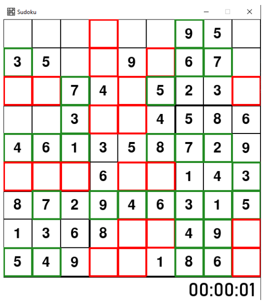
The GUI chosen for the project comes from a public repository [1], and one can simply try to solve the



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

00:00:24

(a) Example of Sudoku generated



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

00:00:01

(b) Example of a Sudoku being solved by the A^* algorithm

Figure 6. Examples of Sudokus [3]

Sudoku by filling the cells with the right numbers, or press the *space bar* to launch the automatic solve. Every time the GUI is launched, a random Sudoku like the one in Figure 6(a) is generated.

When the A^* algorithm is started, all the visited cells are marked in green, and they are marked in red if the algorithm followed a wrong path and needed to backtrack. At the end, all the cells are filled and the Sudoku is complete (cf. Figure 6(b)).

C. *Benchmarking*

1) Results

We have, in total, 9 methods to compare, by averaging their number of trials on 10.000 Sudokus. We measured the time taken by each mode to solve a subset of 500 Sudokus. Finally, we also evaluated the number of trials of a traditional backtracking algorithm on 10 Sudokus to put those results in perspective.

Heuristics		Trials	Time (s)
Backtracking		43507	
Mode 0		206.59	2848.56
Mode 1		205.44	3024.63
Mode 2		205.44	3199.53
Mode 3	Weights 0	206.72	3219.14
	Weights 1	204.18	
	Weights 2	203.84	
	Weights 3/0	204.23	
	Weights 3/1	206.03	
	Weights 3/2	204.35	

2) Discussion

As we can see, the benchmarking of section IV-C1 shows no remarkable difference between the nine methods. However, those results show that $A*$ helps to solve the Sudoku way faster than the traditional backtracking algorithm[2]. It is obvious that the $A*$ algorithm implemented is by far better than the traditional backtracking approach.

[1]See https://github.com/Mercrist/Sudoku-GUI

[2]An average on 10 Sudokus is not really relevant of the overall performance of the algorithm, but we only want to show here that the simple backtracking method has a really poor performance compared to $A*$

The averaged performances for the different heuristics show that the information brought by the different methods do not help to solve the Sudoku faster. We also see that the weights obtained after the training do not differ a lot (which might explain why the statistic in the third method do not vary a lot).

Another information that can be relevant is the fact that the information about the choices to make in order to solve a Sudoku as fast as possible is almost entirely contained in the row, the column and the square of each cell, because the rest of the grid does not help to find faster solutions.

From the benchmarking we can also see that the time needed for the first heuristic is the smallest of all, even though the number of trials needed are pretty close. This is obvious since the simpler heuristics needs less computation and thus takes less time in the end.

V. Conclusion

From the results of section IV-C1, we can clearly see that mode 0 (see II-B1) is the fastest method, in time, even though the necessary number of trials is the largest. We assume that the difference in terms of number of trial between the highest score (206.59 for mode 0) and the smallest (203.84 for mode 3 with weights 2) is not significant enough compared to the difference of computation time (2848.56 s / 3219.14 s) to assume this last mode gives significantly more interesting results.

We can then infer that the information brought by the entire grid gives no remarkable indication on the path to follow and that simpler techniques should be privileged. To go further, the impact of the size of the grid on the power of each method could be explored. The variation of the sparsity in the used grids might also make the results vary.

References

[1] Jad Matta. Brute force approach algorithm for sudoku brute force approach algorithm for sudoku brute force approach algorithm for sudoku. 06 2020.

[2] Eric Pass. Solving sudoku using artificial intelligence. http://www.ericpass.com/assets/AI_project_report_ecp89.pdf, year.

[3] Mercrist. Sudoku gui. https://github.com/Mercrist/Sudoku-GUI, 2020.