End to End Sudoku Solver

Iddo Ziv | Amitsour Egosi | Yehuda Mishaly | Avihay Malul

August 16, 2020

Abstract

In this report we present our attempt at creating a end-to-end program that is able to see an image of a Sudoku puzzle and solve it. We implemented a fully connected neural network that recognizes the digits extracted from the board's image. Then we compare several different approaches to solve the puzzle including an original algorithm. Finally, we discuss other usages to our project.

Contents

Part I
Describing the problem

1 Introduction

The question we asked that sparked the idea of the project is: “Say someone created an intelligent robot that lives life as we humans do, and one day he encounters a Sudoku puzzle in the newspaper. What would go inside his brain in the process of solving the puzzle?”

This is somewhat a philosophical question but it inspired us to try and create a program that is able to solve a Sudoku puzzle with all the steps along the way. From seeing the image with the robots eyes, to eventually placing the digits of the solution in the grid on the newspaper.

Part II
Solving the problem and workflow

2 Image recognition

At the beginning of the program, we receive as input an image containing a board of sudoku. From this image we wish to extract all the information necessary to solve the puzzle. The first step in successfully translating the given input into a representation that enables us to solve the puzzle, is to find the board in the image and where are the digits in the board. After we find the digits we’ll crop them out of the board and collect them in a set of 28×28 images to be passed to classification. This image processing part is composed of the following stages

2.1 Cleaning the input image

In order to be able to successfully find the board within the image we’ll need to get rid of the 3 color channels in the image (RGB) and convert it to gray scale. This is done by multiplying the channels vector by a weights vector defined as the international standard. Then we must clean the image from all the irrelevant noise that a typical image has. We do so by smoothing the image in the following manner: we set the value of each pixel to be the weighted average of its neighboring pixels, this way we are able to remove any outliers and irregular disturbances from the image that would make the task of finding the boarder of the board difficult. Lastly we use adaptive Gaussian thresholding in order to turn the image from gray scale to black and white. For every pixel in the image, we mark it as black only if it is darker than the pixels around it, using a Gaussian weight function on its neighborhood. Below are the results of the measures we took in order to clean the image. From RGB to gray scale, smoothing and adaptive Gaussian thresholding, from left to right.

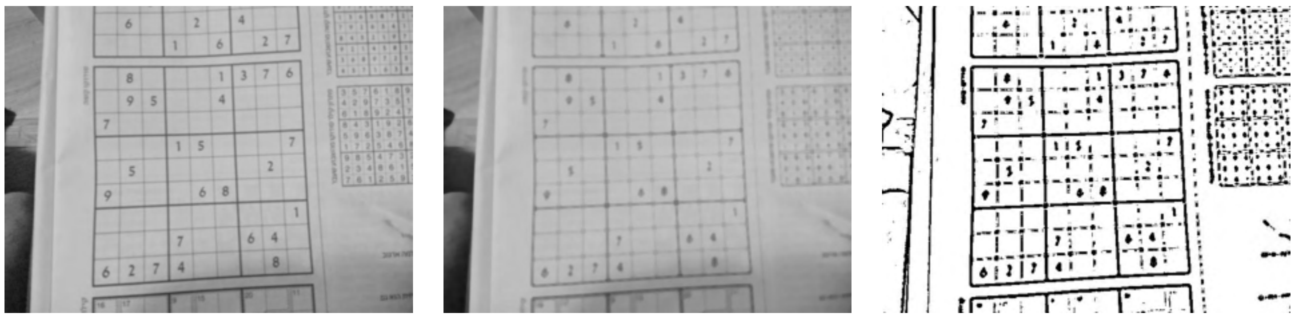
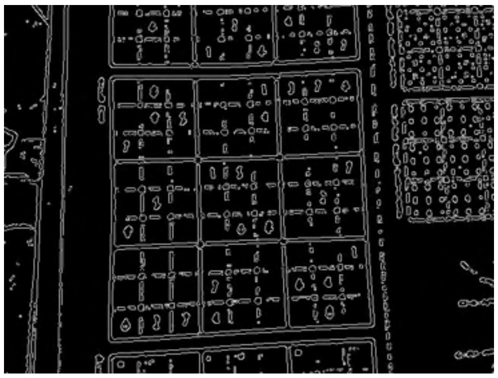


Figure 1: Cleaning the image

2.2 Finding the board

Since every Sudoku is written inside a grid like board, we used the assumption that the grid is the largest connected shape in the image, and so we'll look for the biggest shape in the image. To do this, we first find all the contours in the image by following edge pixels (black pixel with white neighbor). Next, out of all the contours we found, we pick the contour with the largest area contained inside of it, as by our assumption this is the board's border. To find the board itself we simply find a quadrilateral that is wrapped in this contour. Below, on the left, we see the edge pixels that we found and on the right the all the contours in the image such that the contour with the largest area within it is marked with green.



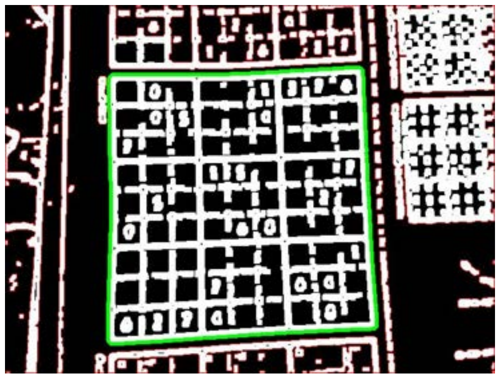


Figure 2: Finding the grid

2.3 Transforming the grid into a square

Now that we know where is the board in the image, we want to cut it along the grid lines. However since the image might have been taken from an angle, the area we found, in which the board resides, is not necessarily a square but rather the result of an affine transformation on the square, turning it into some quadrilateral. We therefore find the affine transformation and transform the quadrilateral we found back into a square using Perspective Transform. Note that we apply this transformation on the original colored image, to keep information. The result of this transformation can be seen below

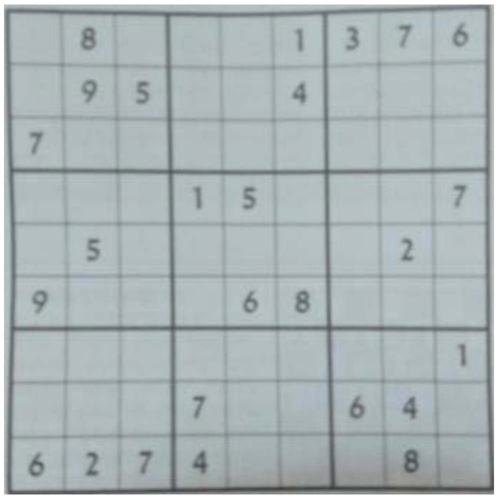


Figure 3: Transforming the board into a squared shape

2.4 Cleaning and cutting the digits

At this point we successfully found a squared shaped image of the board. To extract the images of the digits, we again clean the image, this time without smoothing so that we don't distort the digits, then we invert the image's colors so that the background is black and the digits are white (easier to learn this way). We now guess the lines of the grid (cells are equally spaced and we know the size of the board) and so we create a mask to delete the lines from the image, and afterwords we simply divide the resulting image into 81 cells. Now to make things easier for the classifier, we center the content of every cell. To do that we look for the largest connected shape in the cell using flood fill algorithm. The algorithm runs on all the pixels in the cell, and when encountering a white pixel, if it is not part of a shape, the algorithm selects a new shape id. Afterwords it searches for all the connected white pixels using dfs, and then we just find the shape id that takes most of the space of the cell (by finding its furthest pixels), and cutting around it. In the figure below we show the steps in this part; from left to right: cleaning the image, masking the lines and extracting images of cells with centered digits.

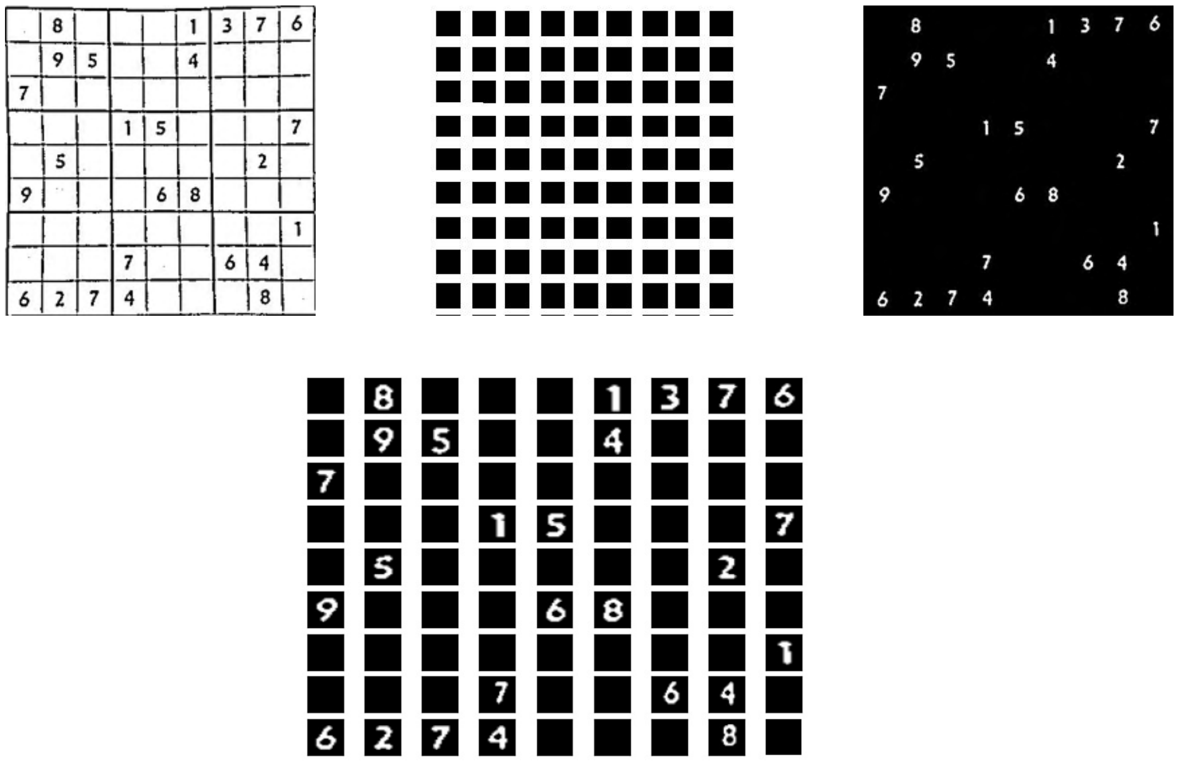


Figure 4: Cutting the digits

After cutting all the cells of the board we use a threshold on the number of white pixels in the cell to determine if a cell is empty, and then we are ready to send images of digits for prediction.

3 Classifying images of digits

Now that we have cropped the image containing a board of sudoku into black and white 28×28 images of digits we have to correctly identify the digits so that we could work with the digital representation of the board.

In order to achieve this multi-class classification task we decided to create a fully connected neural network.

3.1 Network architecture

Since the images we work with have the size of 28×28 pixels, and since there are 10 different classes to classify to (corresponding to the different 10 digits), the first (input) and last (output) layers have size of $28 \cdot 28 = 784$, 10 neurons respectively. In order to choose the number of hidden layers and their size we had to balance between our wish to enlarge the model for better expressiveness, and the need to prevent over-fitting and comply to our resources limitations. Comparison between different choices are discussed below. This being a classification problem we chose our Loss function to be the Cross Entropy Loss, namely

$$L_x(A, Y) = \sum_{(a,y) \in A,Y} - yln(a) - (1-y)ln(1-a)$$

where A, Y are the output of the net and the real label for an input x. We chose this function since it has the property of not being dependent on the derivative of the activation function which ensures that the more erroneous we are the faster we learn. This Loss function cannot receive 0 values and therefore the activation function to the last layer must be positive, and so we chose the Sigmoid function. for the other layers we decided to try both the Sigmoid and the ReLU functions.

3.2 Implementation

In order to build the network we implemented the SGD algorithm that updates the weights and biases of the model in the direction that minimizes the loss. Our SGD implementation uses the Back-Propagation algorithm on the training data partitioned into mini-batches, and does this for a given number of epochs. Each epoch we split the data to mini batches then apply forward pass on the net and backprop to compute the gradient of the loss with respect to the weights and biases. For the update of the weights we added a regularization parameter in order to better achieve better generalization on real time data. Additionally, we step-decay the learning rate by half every 5 epochs to help the system converge into the minima in a less chaotic manner.

To create the model we had to choose between different values to the hyper-parameters. These parameters include the number of epochs, the mini-batch size, the regularization parameter and the learning rate. Additionally we had to decide how to initialize the weights and we tried $\sqrt{1/n} \cdot randn$ or $\sqrt{2/n} \cdot randn$ where n is the number of neurons in the layer the weight is going into, as they have been found in the literature to be quite useful. For the biases we saw no reason to have a special initialization because weights account for that, and so we initialized them with the 0 vector.

3.3 Data

Sudoku boards have their digits printed on them and so it only made sense to find a labeled data with images of printed digits. After looking online we found a csv file containing 270 labeled images 30 for every digit. In order to make the available data for training larger, make the model more robust for anomalies, and to train the model on a more realistic and diverse data we had to augment the data. The different transformation we considered are: original image, stretch, zoom, shift, blur, random noise surrounding the image, Gaussian noise, Salt & Pepper noise, Poisson noise, Speckle noise, as shown in the figure below from top to bottom left to right:

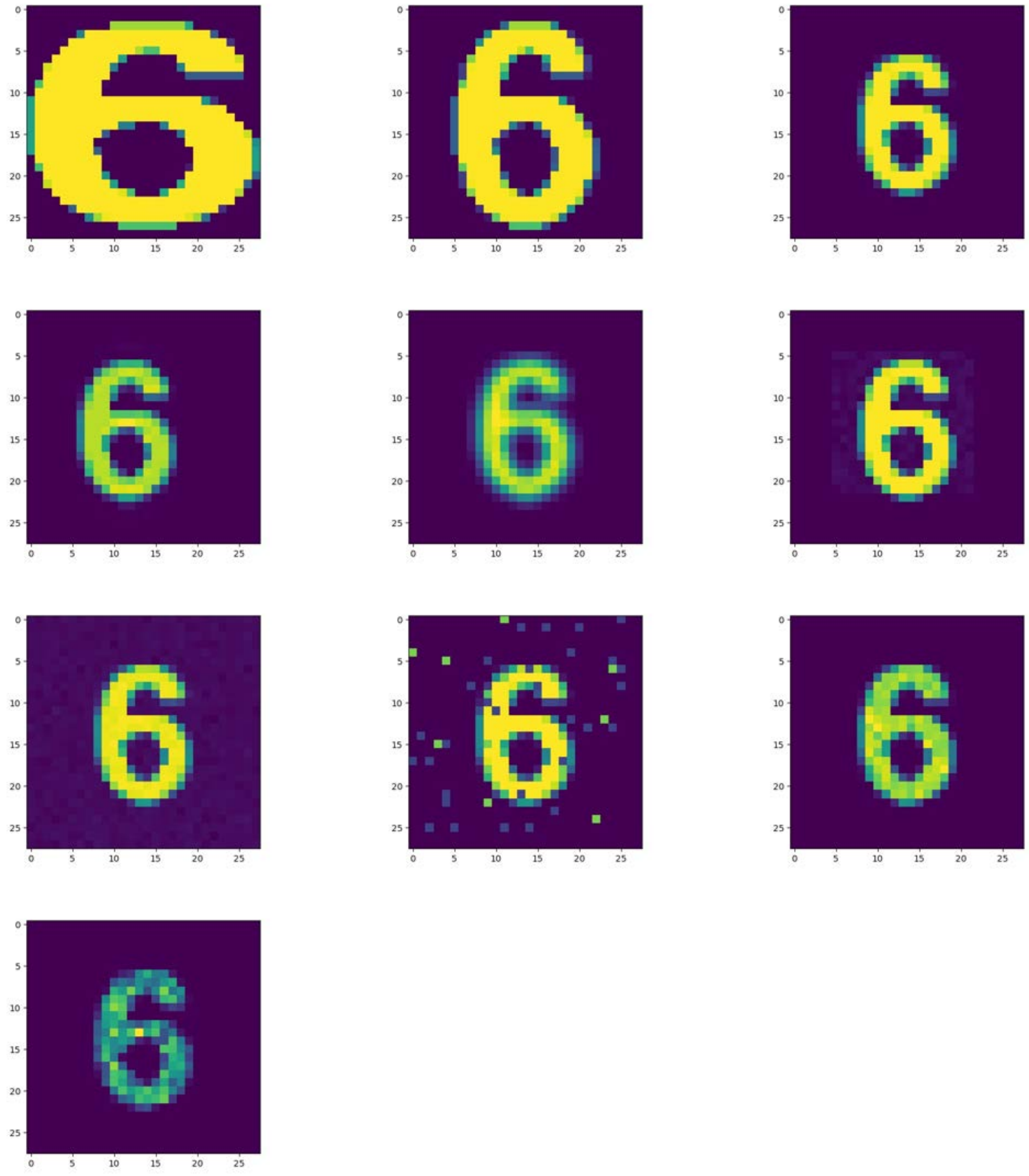


Figure 5: Different transformation applied on the top left image

3.4 Performance Comparison

Now that we have our model and data ready for training we have to try different types of data augmentation and different values to the hyper-parameters, and check which combination yields better accuracy on the test data. We used stretching, zooming, shifting and most of the noise types above, together with composition of them to create the data. After reading some of the literature we decided to use the ReLU activation function on all but the last layer where we use the Sigmoid activation. Preforming cross-validation we observed that the best values for regularization and learning rate are around $15, 0.5$ respectively. Below is a table with the different experiments we made

Table 1: models

Table	**Results of different models**						
No.	***shape***	***reg***	***learning rate***	***epochs***	***batch***	***train acc***	***val acc***
1	$[784, 168, 10]$	0	0.5	7	10	0.9983	0.986
2	$[784, 128, 30, 10]$	5	0.5	5	10	1	0.988477
3	$[784, 128, 30, 10]$	5	0.5	10	10	1	0.98913
4	$[784, 128, 30, 10]$	10	0.5	5	10	0.9999	0.9895
5	$[784, 128, 30, 10]$	15	0.5	5	10	0.9999	0.98966
6	$[784, 128, 36, 18, 10]$	15	0.5	7	10	0.9967	0.9899
7	$[784, 128, 64, 30, 10]$	15	0.5	7	10	0.997	0.9899
8	$[784, 128, 40, 20, 10]$	15	0.5	5	10	0.9998	0.9902
9	$[784, 128, 40, 20, 10]$	5	0.5	5	10	0.9983	0.9926
10	$[784, 128, 40, 20, 10]$	5	0.5	7	10	1	0.993
11	$[784, 128, 40, 20, 10]$	15	0.5	7	10	0.9998	0.9958

As can be seen from the table the best combination we could find has the architecture $[784, 128, 40, 20, 10]$, regularization 15 and learning rate of 0.5, and so we chose this model as our model.

Now we have a model that when given an image of a printed digit we run it through the net and then apply the Softmax function in order to get class probabilities as prediction.

Upon using our model on images of Sudoku boards we saw that the program failed to identify all the board more often than we wished. Assume that we have model with 99% accuracy, and assume the reasonable assumption that there are 30 digits in the board to classify (out of the 81 possible cells) then the probability that the model would classify all the digits correctly, and thus allow us to presume with the program, is $0.99^{30} = 0.7397...$, meaning that the model would fail to classify the board with probability of $1 - 0.99^{30} = 0.26$. This is not negligible at all!

3.5 Committee

To deal with this problem we closely investigated the misclassified digits and noticed that most of time if we change the input image by a tiny amount, using one of the transformation discussed above, then the model suddenly succeeds (this is probably due to the fact that the original data we started from was very small and so over-fitting is a grate risk, as there is a limitation to the ability of the augmentation to truly mimic large data). Therefore we decided to create some sort of a committee method in which we make several tiny changes to the image and take the weighted majority vote of the model's predictions on them (weighted by the certainty the model has in its prediction). This improved the success drastically and allowed the program to work better (but still not perfect since as we mentioned we need to be correct on **all** the predictions which isn't trivial) on relatively nice and clean images of Sudoku boards. Now, given images of the digits present in the board we are able to identify them and pass the board as an array of number to the next part of the program; finding a solution.

3.6 Handwritten digits

One question for further research that naturally arise is regarding the case in which some of the digits on the board are not printed but rather handwritten by a human (as is the case when you start solving a sudoku puzzle and give up midway). An immediate attempt would be using the model we have just built, but doing so we reached very low accuracies. After some thinking we hypothesized that the reason for that is the fact that some digits have a different shape when switching from printed fonts to handwritten. As an example consider the images below

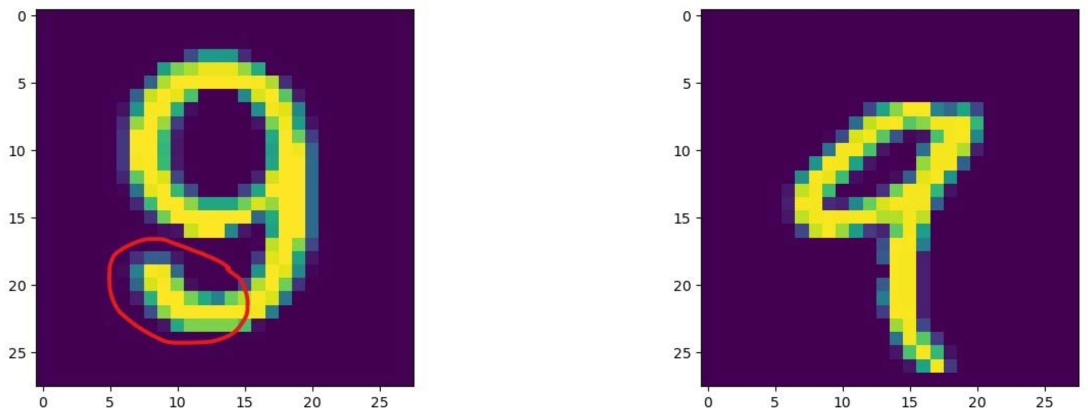


Figure 6: printed vs written

Notice the difference in the bottom part of the digit. All the printed digits the network was trained on have this “tail”, but handwritten digits don’t have this part (at least not when Americans write, as is in the MNIST data). These kinds of differences made it impossible for our model to properly predict. There are therefore two possible options: either train one model on data with both printed and handwritten digits, or train two models one for printed and one for handwritten and then another classifier that determines the type and which model should we use. Playing around with the network and the combined data (our printed data and MNIST) we concluded that this task cannot be solved with our tools especially with such unbalanced data sizes. The limitations of a fully connected network in recognizing and dealing with images that contain some degree of complexity in their structure, are well known, and we are confident that such problem could be easily solved by a Convolutional Neural Network. But we leave it to for future work.

4 Solving the Sudoku puzzle

4.1 Stochastic Search - Simulated Annealing

As a main part of our project we designed and implemented 2 different stochastic algorithms in order to solve any given Sudoku puzzle - one is a simulated annealing algorithm and the other is a genetic algorithm. In this chapter we'll discuss the different design considerations, Implementation details, problems & insights that arose from this process, and the general performance of these algorithms.

4.1.1 Introduction

Simulated annealing algorithms are probabilistic techniques for finding a good approximation for a global max/min point of a function, and are typically used to search in a large (finite) search space. These algorithms are inspired by the process of annealing in metallurgy (a technique for controlled heating and cooling of crystals that is used to increase their strength).

Note - a similar, but more well-known algorithm to SA is gradient decent. We choose to use SA because it is superior in finding a **global** optimum.

4.1.2 Algorithm design

The algorithm works in the following manner:

1. We start with a given board - curr_board. We then create a successor to that board, called next_board, by choosing (*) two entries (=blank squares in the original board) and swapping them.
2. We give a score to each board by counting how many unique, non-repeating elements (=any single digit square in the board) we have in a 3-by-3 block, row or column. after we finish counting these, our score will be -(count). Since there are 9 such blocks/rows/columns and up to 9 unique elements in each of these, we get a maximum count of 243. And since that we can only get to this maximum in a **solved** Sudoku board (by the game's definition), We concluded that a global minimum of -243 is also our stop condition.
3. if we find that next_board has a lower score than curr_board. We define curr_board as next_board. check for the stop condition, And iterate back from (1) (**) if needed.
4. in case next_board has an equal or greater score than curr, we may still switch between them. the **probability** for this switch is $e^{\Delta S/T}$. where e is Euler's constant, ΔS is the difference in score between curr and next, And T is the current temperature.

Finally, we iterate this 4 stages until we reach our stop condition (score = -243) or our predetermined number of iteration (itr_limit).

Notes

(*) Most of the time we choose these two entries by randomly choosing a 3-by-3 block, and than randomly choosing 2 entries. Since we noticed we get Stuck in a local minimum with the board that are harder when using this method, We chose to switch to just randomly picking two entries and filling them with random numbers (from 1-9) if we reached half of the number of iterations that is our iteration limit.

(**) In every iteration our temperature T parameter slightly decreases, reaching a number close to zero when we reach our iterations limit. This is a crucial step in the algorithm design, since that as T approaches zero, our search space gets narrower and narrower, and will tend to get stuck in a local minimum if we don't design a proper cooling schedule.

4.1.3 Implementation

The parameters that decide our algorithm's behavior are the temperature, rate of cooling (cooling schedule),

As it is can be very difficult to calculate the optimal values for the parameters of this algorithm in the design phase, We choose to calculate those values by trial & error - we ran the program on 9 different Sudoku boards, with 10 different sets of parameters, and averaged the results over multiple runs as such. Eventually we chose the best preforming set of parameters (differences between 'functioning' sets of parameters was not critical, so we chose not to optimize any further).

4.1.4 Problems & Insights

The main problem with this algorithm is that, on relatively hard Sudoku boards, it tends to get stuck in a local minimum and not find a solution. We tried to solve this problem in several ways:

1. Inserting a random board as a successor every x iterations
2. Experimenting with the cooling schedule (we tried different exponential coefficients, and a linear approach as well).
3. Different variations on the successor generating function (randomly swapping 2 squares, putting random integers in squares that are not unique in their row/column).

These approaches didn't work eventually, They only gave incremental improvement. The main conclusion for us was to chose a better overall performing algorithm for this project.

4.2 Stochastic Search - Genetic Algorithm

Another way to find a solution to the Sudoku problem is using a genetic algorithm. Using this approach we model our problem using terminology and principles inspired by the field of evolutionary biology. In this model an individual is a unique Sudoku board determined by the entries in the board, and a population is a set of different individuals. One particular individual is considered to be the perfect individual, representing the solution of the problem, and we hope it would emerge from the population after s certain amount of generations. In the begging, the given input induces the first generation, then, as a result of reproduction and the principle of natural selection that arises from individual's fitness, we obtain a new, and hopefully a fitter generation.

4.2.1 Implementation

In the process of implementing the above procedure several key components were considered:

- Population initialization - To initialize a population we create a set of boards such that every board is a different and random filling of the input unfilled board in a way that doesn't violate the rules of Sudoku with respect to the given unfilled input board. This way we are restricting the search space and avoid knowingly allowing illegal individuals with bad genes to contaminate the population.
- Fitness evaluation - In order to assess the fitness of a given board we tried two approaches:
    1. In the first approach we simply used the rules of the game and count for every row, column and sub-box the number of unique numbers in it and then reduce this number from 9. Indeed, the sum of all these differences amounts to 0 if and only if the board comply to the rules - is the solution. Conversely, the bigger the number the more violations there are. We negate the value returned in order to turn the problem into one of finding the global maximum of the fitness function.
    2. The second approach adds to the first one two additional constrains. First we calculate the difference between $\sum_{x=1}^{9} x = 45$ and the sum of elements in the row/column/sub-box, and second we calculate the difference between $\prod_{x=1}^{9} x = 9! = 362880$ and the product of elements in the row/column/sub-box. The

idea being that in the solution these are the values of the sum and product in every row/column/sub-box. We then apply a weighted sum of all the differences and return the value with a negative sign.

Comparing these two approaches we saw no advantage in using the second one. The two additional constrains are not exclusive to the solution. For example a row filled entirely with 5 would have a sum of 45 wrongly implying it is somewhat better then a row filled with 4. As a result we decided to use the simple and interpretable function in the first approach.

- Parent selection - We chose each parent in the following manner: Two competitors were drawn uniformly and the fittest one was chosen to be a parent with some small probability that the weaker would win.
- Reproduction - Given two different parents we create from them two children using *multi-point whole-row crossover,* such that random rows in the parent's boards are swapped to create two new individuals.
- Mutation - After a new offspring is created, and with some probability, we apply a mutation on the child's chromosome. The mutation we chose is a *gene-swap mutation* in which we draw a random row in the board and we swap two random values in it (while keeping the rules with respects to the input board).
- Running process - In every generation an *elite* is kept aside to be passed to the next generation. The elite is composed of the top individuals determined by a given threshold, the rest of the population in the new generation is created by reproduction, with the fittest individual defined as the best board so far. If during the allotted number of generation a solution is not found we return the fittest individual in the last generation.

4.3 Backtracking with dfs

The backtracking algorithm is an algorithm that that is used in order to solve search problems. The algorithm treats a problem as if it was a decision tree in the following manner:

- The root of the tree is the initial state (board)
- For every node in the tree (board) each legal assignment to the board would be considered as a child node.
- A leaf in the tree is either a full and legal board or a board such that for at least one of its cells there is no valid assignment.
- Every node along the path from the root to a leaf would be a partial solution.

We are looking for the path between the root and the leaf representing the solution for the puzzle. In order to do that the algorithm traverse the tree from the root as DFS does, visiting the first child node:

1.
    (a) If the child represent a valid solution we return the child board.
    (b) If the child represent a board with at least one cell with no valid assignment, we return that no solution exists.
2. Otherwise - we try to assign one of the valid assignments to one of the empty cells, and call the method again.
3. If we got a legal solution we return it. Else we return that no solution exists, we delete the last assignment and try another one (back to step 2).
4. In the case where for every possible assignment we didn't find a solution we return that no solution exists.

This way we can check every possible assignment for a given board and determine whether it has a solution or not, and when a solution exists what does it look like.

4.4 Arc-Consistency Algorithm

This is another method we tried in order to achieve an elegant and efficient algorithm. The mechanism behind the algorithm is inspired by the way humans apply their intuition to approach the task of solving the Sudoku problem. In order to explain the way by which the algorithm works, we lay here some helpful definitions:

- $A = \{1, ..., 9\}$
- Cube - We define a cube to be a 3×3 part of the original board such that in the upper left corner (x, y) it holds $x\%3 = 0$ and $y\%3 = 0$.
- Variable - A variable in our problem is each cell in the Sudoku board. We denote the variables as a concatenation of the form xy where $x \in \{A, B, ..., J\}$ and $y \in A$. For example "F5" refers to the cell in the sixth row and fifth column.
- Neighbor - Variable x_i would be called a neighbor of x_j iff they both reside in the same row/col/cube.
- Valid assignment - A valid assignment with respect to (x_i, x_j) and a given board is a tuple (a, b) such that $a, b \in A$ and if (x_i, x_j) are neighbors then:
  - $a \not= b$.
  - for all x_k neighbor of x_i, if x_k was assigned with c then $a \not= c$.
  - for all x_k neighbor of x_j, if x_k was assigned with c then $b \not= c$.
- Domain - The domain of a variable x_i with respect to a given board is a subset of A.
- *arc-consistent in relation to* - A variable x_i is *arc-consistent in relation to* another variable x_j with respect to a given board if for every value a in its domain there exists a variable b in $x_j's$ domain such that (a, b) is a valid assignment with respect to (x_i, x_j).
- *arc-consistent* - A variable x_i is called arc-consistent with respect to a given board if it is *arc-consistent in relation to* all the other variables in the board.

4.4.1 Remarks

1. (from the definitions) let x_i be a variable that isn't arc-consistent $\implies$ exists another variable x_j such that x_i is not arc-consistent in relation to x_j $\implies$ exists a number $a_0$in x_i's domain such that for all b in x_j's domain the assignment (a_0, b) is not valid with respect to (x_i, x_j) $\implies$ We can remove a_0 from x_i's domain without missing a possible solution.
2. (from 1) If we reduce the domain of every variable using the principle discussed in the previous remark, Then if we reach a state in which there is a variable who's domain is empty we can assert than no solution exists. Additionally if we reach a state in which every variable has one value in its domain, Then said value is the only valid assignment to its corresponding variable with relation to all the other variables $\implies$we can extract a solution from this state.

4.4.2 Implementation

- Initialization - We define a list of edges (x_i, x_j) for every pair of neighbors, and an initial dictionary holding the domains of the variables such that if an assignment doesn't exist in the board (cell is empty), the domain is A, else the domain is a set of size 1 containing the assignment (value).
- Domain subtraction - We traverse the list of edges and for each edge (x_i, x_j) we check if we can reduce the domain of x_i by applying the principle discussed in remark 1. If we can do so, we reduce. Note that in this scenario, remark 1 holds and therefore there exists a value a_0 in x_i's domain such that x_j's domain is $\{a_0\}$. Indeed, we use this observation as a condition to the reduction of the domain of x_i. Now, after reducing, x_i

is arc-consistent with respect to x_j. Having modified the domain of x_i we must repeat the above procedure with respect to x_j (as the reduction could have possibly violated the arc-consistency (if there was one) of x_j in relation to one of its neighbors).

- Solution extraction - We traverse the domains of all the variables. One of the following 3 situations necessarily holds:
  - There exists a variable with an empty domain, and therefore we can conclude that no solution exists (using remark 2)
  - All the variables have exactly one value in their domain, and therefore we can conclude that its value is the correct assignment (using remark 2)
  - Otherwise, the domain of every variable has at least one value and at least one variable has more than one, and in this case we cannot extract a solution.

4.5 Arc-Consistency With Backtracking (Forward Checking)

In order to address situations in which the arc-consistency algorithm didn't find a solution, we tried to interweave the principles from the previous algorithm together with the classical Backtracking algorithm. The idea we had in mind is that whenever the arc-consistency algorithm has no definite assertion after its run (this occurs when at the end of the run there is a variable (cell in the board) with more than one value in its domain), we would try to guess a value from the set of possible assignment to the cell (domain), and in doing so we could continue in the elimination of values from variable's domains.

4.5.1 Implementation

1. We run the arc-consistency algorithm on the input board. If at the end of the run there is a variable (cell) with more than one value in its domain we proceed to step 2. Otherwise we either return the found solution if it exists.
2. While we can we choose an empty cell and guess its value from the set of valid value assignments, and return to step 1. If step 1 returned a solution we return it, and if it failed we backtrack (we turn back to our last decision and try again with another value).

During the implementation we tried to preform the assignment, *during the run*, to every cell in the board whose domain reduced to size one. Since the characteristics of the backtrack procedure don't allow for the deletion of such assignment in the case that an error was found, Then in some cases the output would be wrong. Therefore we decided to perform these kind of assignments at the end of the run where we reach a full solution. This way we benefit both from the assured correctness of the backtrack algorithm and the efficiency of the arc-consistency.

4.5.2 Heuristics

Initially we tried to choose the empty cells according to positional ordering (using the method **by_order**). Afterwards, in order to improve the performance of the aforementioned algorithm, we tried several different heuristics from class that we deemed as helpful to solve our problem:

1. **Minimum remaining values** - In this heuristic, the next empty cell chosen by the algorithm would be the cell with the smallest sized domain among all the existing empty cells.
2. **Degree heuristic** - In this heuristic, we have done a variation of the classical degree heuristic in which the next empty cell chosen by the algorithm would be the cell that is involved in the least amount of constrains applied on other neighboring empty cells.
3. **Least constraining values** (heuristic on the values) - In this heuristic, for a given cell the next value that the algorithm would choose is the value that rules out the fewest choices for the neighboring variables (the value that appears the least amount of times in the domains of the neighboring variables).

4.6 Knuth's Algorithm X

Knuth's Algorithm X is an algorithm to solve the exact cover problem using linked list in order to increase search efficiency is a sparse search space.

The Exact cover problem is as follows: Given a Finite Universe U and a group $S \subset 2^U = \{A_i | A_i \subset U\}$ of subgroups of U, find a subset S^* of S so such that each element in U is contained in exactly one subset S^* i.e. $\bigcup_{A \in S} A = U \wedge \forall A, B \in S^* \quad A \cap B = \emptyset$.

The Exact cover is NP-complete, and can be solve using search. One way to look at the problem is using incidence matrix M, using the rows for the groups in S and the columns as the elements in U. $M[i,j] = \begin{cases} 1 & j \in A_i \\ 0 & j \notin A_i \end{cases}$.

In many cases the table U is sparse, meaning the number of zeros is much greater than the ones, making searching for ones hard. To solve this problem Knuth suggested a modified version of the data structure doubly linked list. The Data structure is called Dancing Links, and it is a 2d linked list, meaning each node has 4 neighbors. When a node is removed he is not deleted, and can be easily restored to it's place thanks to the double links, and that gives this data structure great advantage in back tracking. In order to speed up the search each node represents a one in the matrix m, and zeros are skipped, each cell is connected to its closest neighbors in the row and column, allowing to traverse a row in O(n) time where n is the number of ones, which is much smaller than its total size (more on that later). The solution is still a search, using the least possible assignments heuristic, we select a row to add to S^* and the delete conflicting rows, backtracking if needed.

Sudoku as an Exact Cover Problem: In order to convert Sudoku to an exact cover problem we first look at It as the following CSP: each cell must contain exactly one number, each row column and block must contain each number exactly once. So for the 9×9 Sudoku we get $81 + 3 \times 9 \times 9 = 324$ constrains. The constrains are our Universe since each of them must be fulfilled exactly once. Next our groups, based on the choices we have in the game, each group in S represents the assignment of a number to a cell, for example marking 4 in the cell (3,5). Each assignment fulfills 4 constrains, in our example, the constrain (3,5) must have a number, there must be a 4 in row 3, there must be a 4 in column 5 and there must be a 4 in block (1,2).

Analysis: The dancing links algorithm creates the 2 search heuristics discussed in the CSP section, if there is one possible assignment to a constrain it means that ether a cell has only on legal value left, or there is only one legal location left for a number in the cell group.

4.7 Pencil Marks Algorithm

We decided to try to write our own original solver based on the way human solves Sudoku. When solving I solve a Sudoku some people mark in each cell i test all remaining possible values, in order both to save the calculation result and find unique valid location for a value in an area in the board.
Our algorithm expend on this idea, converting the board into a collection of cells and areas(rows, columns and boxes).

Every time a change is made on the board, it is propagated from the changed cell to its areas and from them back to all the cells in each area. every cell counts for each value how many areas blocks him from using it, so when an undo event propagates to him it knows whether it is now a valid value or not.

The board is updated in one of three ways:

- If a cell has only one valid value, we update it.
- If in an area the is only one legal place for a number, we place the number there.
- We choose a cell to guess a value in according to a heuristic (more on that later), and update it to the next valid value.
- Undo when back tracking.

The first two ways run on a loop until the board is full or No changes where made in a full cycle. Then we guess based on heuristic and use backtracking to solve the harder puzzle, more on back tracking above.

The guess heuristics we tried:

- First empty cell, also the null heuristic, just find a cell to guess in (FE).
- Most constrained variable, i.e the cell with the least valid numbers (LO).
- A tie breaker for the above using the call that has the least empty neighbors, the total number of empty cells in all of its areas (LO_MIN).
- Most empty neighbors, as control for the previous heuristic, same as before just sorted in reverse (LO_MAX).

When reaching an illegal board we backtrack to the last guess and try the next available value in the same cell. Because The code is directed to Sudoku we were able to restore both the board and all the "pencil marks" i.e. the valid assignment lists without recalculating them, saving time in the process. On big improvement was saving the lists as Boolean arrays, and also keeping track of the number of valid values in each area and in each cell, allowing both fast calculation of the heuristics and fast catching of illegal board states.

4.8 Performance Comparison

After describing all the different approaches we have implemented in order to solve the sudoku puzzle, we now want to assess their performance. To do that we proceed in two steps. First we investigate each algorithm separately and compare its performance under different degrees of freedom, meaning we find out what are the best parameter choices for the algorithm. Then after we have the best version of each approach, we compare them with each other and see which are the more successful ones.

For brevity we will denote the names of the different algorithms using the following abbreviations: classic Backtracking using dfs - DFS, Simulated Annealing - SA, Genetic Algorithm - GA, Arc-Consistency - AC, Arc-Consistency with Backtracking (AC_BT), Knuth's Algorithm X (DLX) and the Pencil Mark Algorithm (PEN).

There are two datasets of sudoku boards with solutions that we used. The 1 million sudoku games dataset, and another dataset generated from the "qqwing" website. The first dataset is considerably easier than all the difficulty levels in the second one. we used the first one for the local search algorithms, and the second one for the backtracking algorithms.

4.8.1 Finding the best hyper-parameters and heuristics for each algorithm

Simulated Annealing Let us first analyze the effects of the cooling coefficient $c$on the performance of the algorithm.

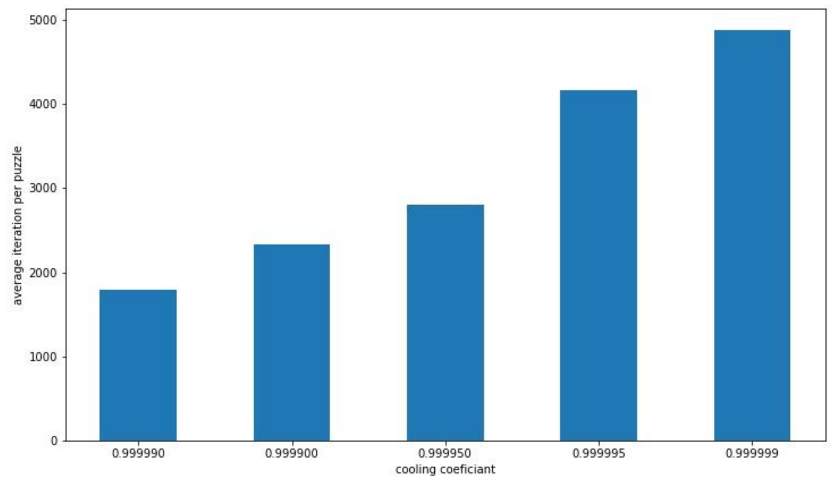


Figure 7: Avg. number of iterations as a function cooling factor

Our cooling schedule is derived from the formula $T_i = c \cdot T_{i-1}$ where T_i stands for temperature at the i'th iteration, and c is our cooling coefficient. As you can see from the graph, the best performance we got is with $c = 0.99999$. More then that, when using values lower than that we stat seeing a decrease in the algorithm success rate - as should be expected since it means our "cooling" is faster, Which in turn means we narrow the search space too fast for the algorithm to find the global minimum, So it tends to get stuck in local ones.

Next we see how changes in the temperature parameter T affects the algorithm's performance.

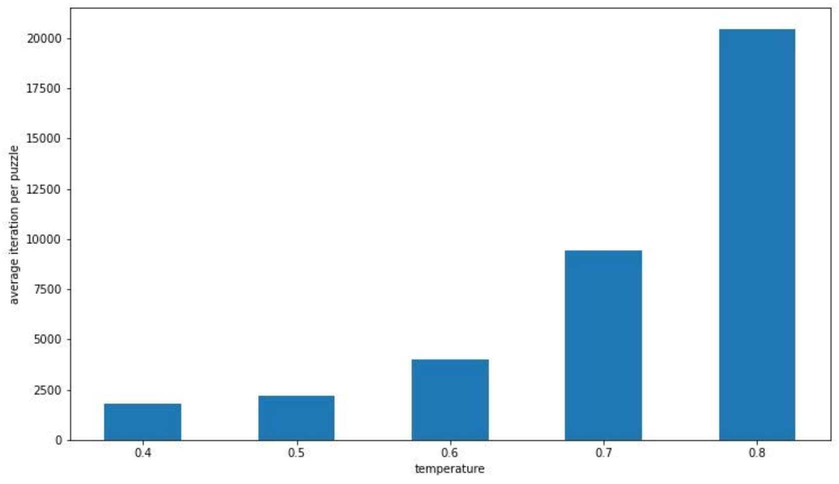


Figure 8: Avg. number of iterations as a function temperature

As you can see, out of the values we tried, performance is best with $T = 0.4$ as initial temperature. That being said, it is important to mention that for any values lower then $T = 0.4$ for the initial temperature we start seeing a decline in the algorithm's success rate (all of the values in the graph has a success rate of 100% on low difficulty

Soduko boards), and not much of a gain in speed, Thus we choose to omit them from this comparison.

The reason for lower success rate is similar in this case as well, As the algorithm starts from a narrower search space, It tends to miss the global minimum more often and to get stuck in a local minimum, Which means it won't find a solution to the board.

We thus use these parameters to create our optimal SA.

Genetic Algorithm The parameters we can tune and compare their effect on the algorithm performance are the fitness function, elite size and mutation probability. As we said when discussing the GA, the first fitness function preforms better and is much more interpretable. Indeed, after running the algorithm on 50 different sudoku boards, changing only the fitness function used, we get that the average number of generations needed to reach a solution is 16.5 using the first fitness function, and 91 using the second fitness function.

Using the first fitness function we now compare between different elite sizes. For every sudoku in a set of 50 random boards, we calculate the number of generations needed to reach a solution for each elite size in the range $1\%, 3\%, 5\%, 10\%$ of the population.

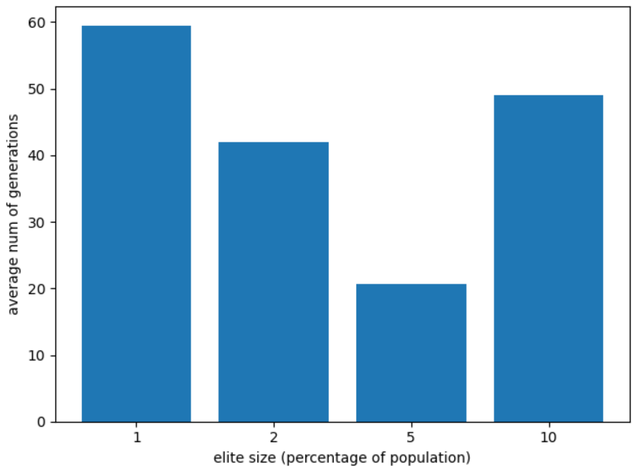


Figure 9: Avg. number of generations as a function of elite size

As can be seen, the best elite size is 5%.

For the choice of mutation probability we tried the values $0, 0.04, 0.06, 0.1, 0.2$. Again, for every sudoku in a set of 50 random boards, we calculate the number of generations needed to reach a solution for each probability in the range.

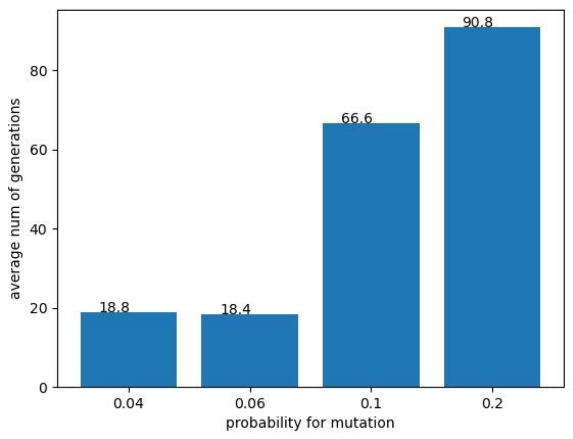


Figure 10: Avg. number of generations as a function of elite size

And as can be seen, the optimal value for the probability is 0.06.
Overall we used these parameters to create the best genetic solver.

Backtrack with dfs This classical algorithm has a straight-forward implementation that doesn't involve hyper-parameters or hypothesis, and so we don't have any comparisons to make in order to create the model.

Arc-Consistency As we said when discussing this algorithm, since it doesn't make use of backtracking it fails to solve some of the puzzles. Therefore we decided not to use it and so we are not making a comparison analysis on it.

Arc-Consistency with Backtracking The "qqwing" Data came with difficulty level, which roughly translates to number of look ahead moves needed to solve the puzzle. In this algorithm our choice is between the different heuristics we discussed above. For brevity we'll use the abbreviations DH - degree heuristic, MRV - minimum remaining values, LCV - least constraining values, where the first two compete on the best way to suggest to the backtrack which cells to fill, and the goal of the last one is to improve the way we decide which value to assign a given cell.

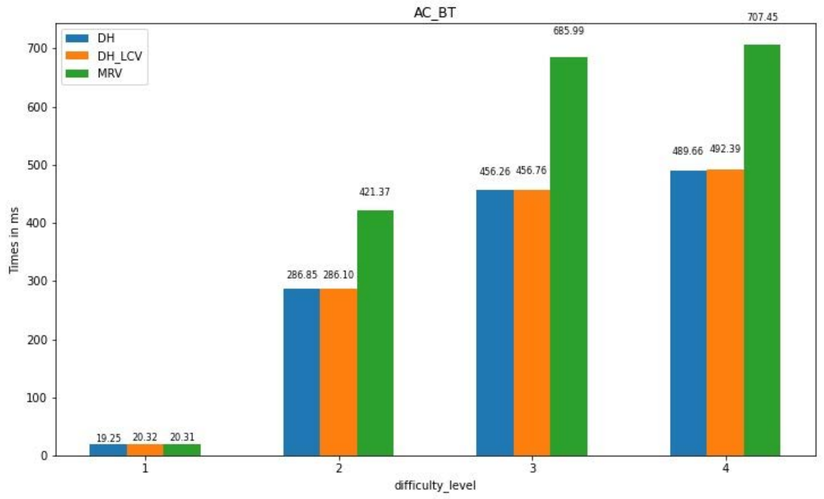


Figure 11: Results for arc consistency with backtracking

As can be seen from the graph above, in all the difficulty levels the performances of DH are superior to those of MRV with respect to cells choosing. The main reason for that could be the fact that even though using MRV we choose the cell for which the backtrack has the least amount of guesses to make (and so less possible mistakes), still there are situations in which some variables have equally sized domains. In this case MRV would return the same value for both of these variables, and the choice would be arbitrary (according to the ordering in the board). On the other hand, in DH we take in consideration the constrains of the cell with respect to its neighbors, and therefore the likelihood that using DH, both cells (variables) would be assigned with the same value is much smaller. In short, when the backtrack chooses its next variable using DH it makes a greater use of existing knowledge of the current state, and so the chances that the decision was correct are greater.

After concluding that for the variable choice it's better to use DH, we examined the influence LCV has on the choice of values for assignment. As can be seen in the graph, LCV doesn't guarantee an improvement in running time, and sometimes even results in a slowdown. This result could stem from the fact that after we choose a correct variable for assignment, the choice of the value has a lesser importance, and as a result the computation time of LCV for every cell we chose cancels out with the advantages LCV offers.

Knuth's Algorithm X This algorithm has a closed form implementation, and thus no further analysis is required to determine the solver.

Pencil Mark Algorithm We now compare the different heuristics that we consider in the goal of optimally solving the puzzle using PEN, using the abbreviations we presented above

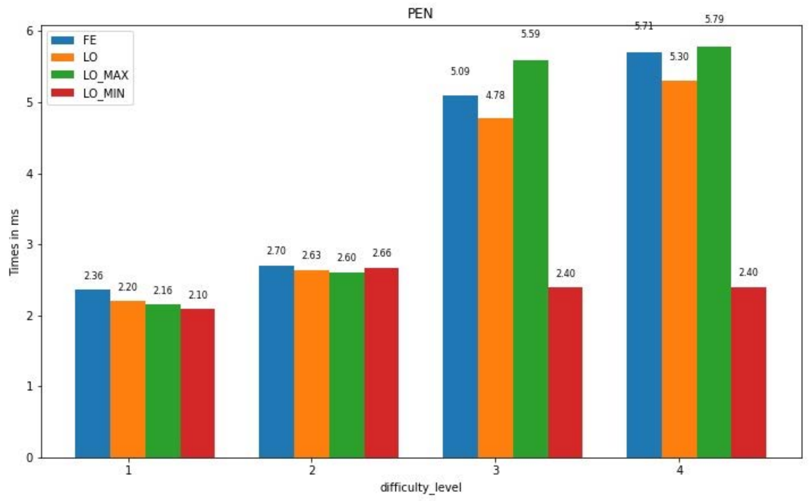


Figure 12: Results for pencil algorithm heuristics

In the first two difficulty levels the heuristic almost never came into play, because no almost guesses were needed. next we see that the least valid options heuristic was better than the first empty (null) heuristic. The least empty neighbors heuristic was best, and most empty neighbors that we ran as control was even worst than just first empty heuristic. It is in line with the search heuristics we learned in class for Constrain Satisfaction Problems, the least empty neighbors is like the degree heuristic from class.

4.8.2 Comparing between the algorithms

After finding the best version of each algorithm, we now want compare their performances in several aspects. We make a distinction between two families of algorithms. One is the local search algorithms, containing the Algorithms SA and GA. The other family is the set of algorithms that are based on the idea of searching using backtracking, these include DFS, AC, AC_BT, DLX and PEN. There are several reasons as to why this distinction is necessary and useful. The main one being the different nature of the underlying principles of these Algorithms that causes a big difference in time performance. Local search algorithms relay on randomness and so take considerably more time to solve a given board if at all.

Stochastic Local Search Algorithms The two stochastic local search algorithms we have are SA and GA. Both are methods that relay heavily on randomness in order to find the global extremum of some score function. They do not take full advantage of the available knowledge in every step and therefor preform much worse than backtracking algorithms. The only interesting thing left to do is compare between them. We ran both algorithms on on the same 50 boards and observed that the average time taken to solve a single board is 34.551 seconds for the GA and 1.76454 for the SA. We also saw that the variance of the average generations needed for the GA to solve is very high, so the above results may vary but imply that SA works faster than GA. These are not suited for our problem but rather for problems where the structure of the constrains is unknown.

Backtracking Algorithms Comparing all backtracking algorithms:

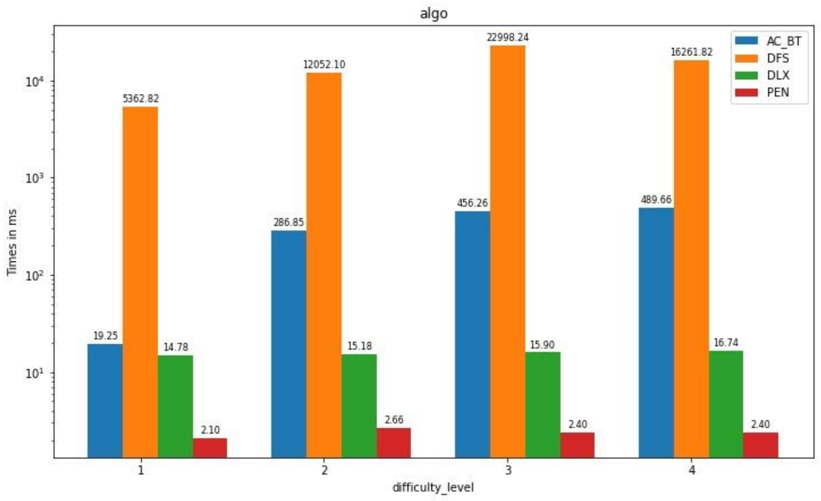


Figure 13: Comparing all backtracking algorithms (logarithmic scale)

The average run-time difference between the algorithms is in orders of magnitude, showing just how important it is to run an efficient algorithm. The dfs came worse, not surprising since it doesn't use the any information presented to it and recalculates all of the constrains every time it checks them. The Arc Consistency algorithm came third, with his easiest difficulty close to dlx, but then falling behind when backtracking is needed. The pencil algorithm is custom made for Sudoku and so it is able to run as efficiently as possible. The dlx algorithm has some of the heuristics used in the pencil algorithm as emergent heuristics, we can see that it helped him keep his times even for harder puzzles, that ac took longer to solve. In the literature we found claims that dlx is the fastest algorithm, we can explain the fact that we could not reproduce this result by the our implementation choices in the dlx algorithm using OOP for readability. For future research in bigger boards dlx should gain an advantage thanks to the sparsity of the search space.

4.8.3 Conclusion

To conclude, we see that to our great satisfaction out of all the algorithms we learned in class, our pencil mark algorithm works better than all the other algorithms, and so we chose it to be the algorithm the program uses.

5 Output Image

To return the solution we found to the puzzle, we chose to display the results back on the original image we received as input. This is done by first writing the numbers on the squared image of the board, and then transforming the square back into its original quadrilateral shape using the reverse transformation of the affine transformation we saved earlier. Finely, we paste the transformed square back on the original colored image, to obtain a neat presentation of the result, as shown below:

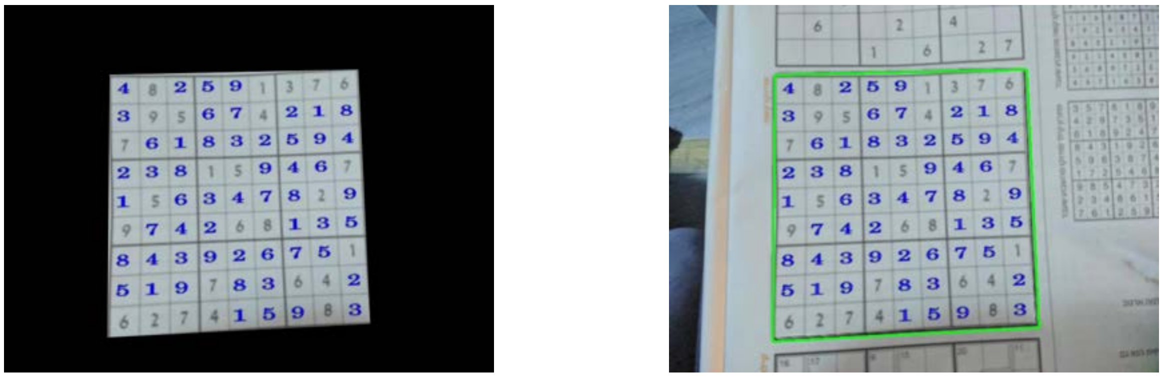
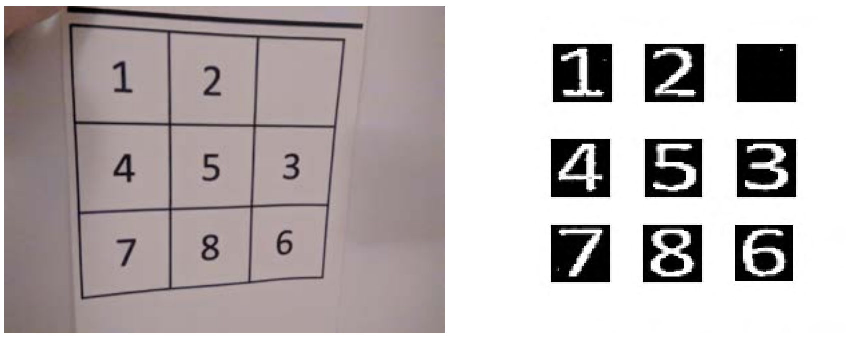
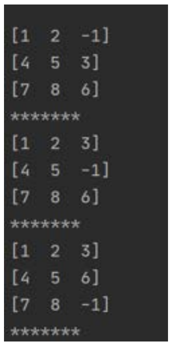


Figure 14: final display of the solution

6 Further work (8-Puzzle) and final remarks

The work we've done in our project can be used to solve a variety of other puzzles that are characterized by a grid and digits. As an example we show here how we used our program to solve an 8-Puzzle. 8-puzzle is an interesting puzzle we saw in class that like the Sudoku involves a grid and digits inside. We ran the same image recognition steps (extraction and classification), but later passed the digital representation of the puzzle to an A* solver and printed the results as text.





```
[1  2  -1]
[4  5   3]
[7  8   6]
*******
[1  2   3]
[4  5  -1]
[7  8   6]
*******
[1  2   3]
[4  5   6]
[7  8  -1]
*******
```

Figure 15: solving the Eight puzzle

Many problems in life are presented to us as written media and our program bridges the gap from it to the algorithm, in a way that your average fully-integrated-in-the-world-robot would probably do it.