CS221 Final project [p-final]

SUNet ID: angelo, quaizarv
Team: Angelo Scorza and Quaizar Vohra

By turning in this assignment, we agree by the Stanford honor code and declare that all of this is our own work.

Introduction

A Sudoku puzzle consists of a 9x9 grid. The objective is to fill each square with a single digit from 1 to 9 such that no column, row or box (3x3 square region) has a digit more than once. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution. The input and output format is simply a text string of digits with '0' or '.' used for representing blank squares. We can also parse nicely formatted grid as shown in the figure (which is just digits separated by '|', '-' , '+' and newline character.

```
Input format:  .7.........12......53..1.....................6.....5...7.46...........7439.24.....
Output format: 276594318891236475453871692645129837329748156187365924714653289568912743932487561
```

```
    Input Format                    Output Format

. 7 . |. . . |. . .             2 7 6 |5 4 9 |8 3 1
. . 1 |2 . . |. . .             4 9 1 |2 8 3 |5 6 7
. 5 3 |. . 1 |. . .             8 5 3 |7 6 1 |4 9 2
------+------+------           ------+------+------
. . . |. . . |. . .             3 8 5 |1 9 6 |2 7 4
. . . |. . . |. . 6             1 2 9 |8 7 4 |3 5 6
. . . |. . 5 |. . .             6 4 7 |3 2 5 |1 8 9
------+------+------           ------+------+------
7 . 4 |6 . . |. . .             7 1 4 |6 3 8 |9 2 5
. . . |. . . |7 4 3             5 6 8 |9 1 2 |7 4 3
9 . 2 |4 . . |. . .             9 3 2 |4 5 7 |6 1 8
```

The scope as presented in the proposal had 2 parts: a) Minimize the time required to solve Sudoku puzzles by identifying constraint propagation rules and heuristics for narrowing the search space. In our case the search space consists of all possible combinations of 0 to 9 across all the squares of the grid. b) Explore statistical methods for ranking the exploration possibilities. So far we have focused only on part a).

We had proposed using Peter Norvig's Sudoku Solver as our baseline. Since we model our problem as a CSP, the evaluation metric we had proposed measures the effectiveness of our Constraint Propagation rules and Heuristics in reducing the total search space. We will measure this as the number of backtracking searches required by our algorithm, i.e.the total # of variable (Sudoku Squares) and their values (digits 1 to 9) explored before finding the solution. Note that our data consists of 1683 hardest puzzles selected from a million randomly generated puzzles (they are harder than the hardest posed by mathematicians - as our solver can solve these much faster than what we have generated).

Model

We model Sudoku as a Constraint Satisfaction Problem. Actually we have implemented 2 models. First model uses the concepts from the class. The second model extends our baseline which is written by Peter Norvig. These models are similar except in the way they do constraint propagation (i.e. evaluation of constraint potentials for domain reduction). We first describe the common parts and then the differences.

Variables: Each square in the Sudoku grid is a variable. There are in all 9x9=81 variables. We name the rows from A to I and columns as 1 to 9. So the squares variables are named as {A1, …, A9, B1, …, B9, ……., I1, …., I9}.

Domains: Domain of each square variable is a digit from 1 to 9 except for those squares for which the puzzle has already specified a single digit. In the later case, the square has that single digit as its domain (and its assignment is complete)

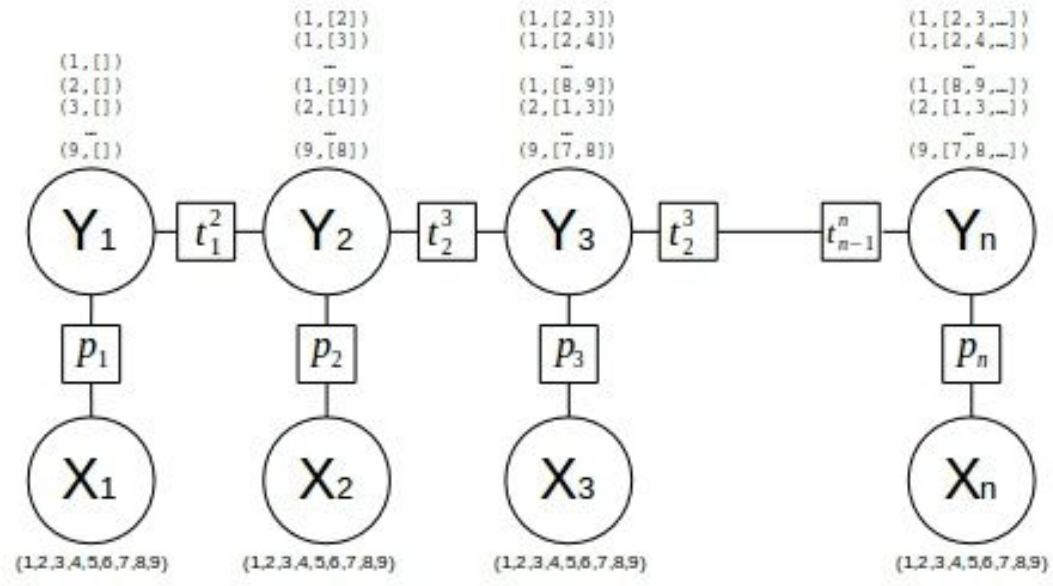
Constraints: Two square variable are peers if they share a unit. For the purpose of further discussion we will use the term **Unit** to mean a **row, column or group**. Conceptually a square has binary constraint (potential) with each of its peers which says that the square and its peers cannot have the same digit. For example if square has square y as peer, then they share a potential f such that $f(x,y): x! = y$

Using binary constraints causes an explosion of constraints and is also not the most optimal way of propagating constraints (reducing domains). So, for both models we chose to implement an n-ary constraint which ensures that no 2 squares in a unit have the same value.

Model 1

We define a n-ary operator known as `all_different` defined as follows:

Input:	CSP , list of variables names $X = (x_1, x_2, x_3 ... x_n)$, list of values to assign $V = (v_1, v_2, v_3 ... v_n)$.
Output:	CSP that includes: variables $y_1, y_2, y_3 ... y_n$ potentials $p_1, p_2, p_3 ... p_n$ potentials $t_1^2, t_2^3, t_3^4 ..., t_{n-1}^n$ All together ensure that all the $x_1, x_2, x_3 ... x_n$ have unique values.



Where:

X_i is a variable such that $domain(X_i) \supseteq V$

Y_i is a variable such that each value in the domain is a tuple $(w_a, [w_0, w_1, ..., w_{i-1}])$, where $w_a \in V$ and $w_0, w_1, ..., w_{i-1}$ are any combination of $i-1$ elements from V and $\neq w_a$.

P_i define a function $f(x_i, y_i): x_i == y_i[0]$

T_i^{i+1} define a function $f(y_i, y_{i+1}): (y_i[0] \cup y_i[1]) == y_{i+1}[1]$

Model 2

The baseline model also has the notion of variables and their domains. It does not use the concept of potentials formally but does constraint propagation manually whenever a variable is assigned a value, by implementing a recursive version of AC3 (which is described in the Algorithm Section below).

Algorithm (CSP approach)

Our basic algorithm (for both models) alternates between doing arc-consistency checking and backtracking search (as described in the class). We start by initializing the domains of each of the squares with digits 1 to 9. Next we parse the puzzle and for variables which are given in the puzzle, we set their domain to their puzzle assigned values. This triggers the first wave of constraint propagation using AC3. Once this settles, we start backtracking search which picks a square and an option (digit) to assign to that square. This triggers the next round of AC3 and this continues until either the puzzle is solved (or we realize that the puzzle has no solution when all options for a square have been exhausted). Following are the details of our generalization of AC3 followed by the explanation of backtracking search and the heuristics we use.

AC3: We use a similar arc-consistency checking algorithm as implemented for our scheduling assignment in the class for propagating constraints (i.e. eliminating incompatible options from domains in the process) . Our baseline has implemented a recursive algorithm for AC3 which is called everytime the domain of a square is modified as a result of value assignment to variables or by the application of AC3 itself.

Our arc consistency is much more generalized compared to what was taught in the class as well as our baseline. It consists of the following constraint propagation strategies. The first 2 were already implemented in the baseline. The last 3 were added by us on top of the baseline.

1. **Single Choice**: If a square is reduced to only one value, then eliminate that value from peers of that square.
2. **Single Possibility**: if a unit has only one square left as a possibility for a value, then put it there.
3. **Subregion Exclusion** - if a number is restricted to a row (column) of a single box (i.e the 3 squares of the row or column lying within that box) then it can be removed as a choice from other squares in that row (column) (the trigger here is CP within the box). The other way round is also true, i.e. if a number is restricted to 2 or 3 squares in a row (column) and those squares happen to be in the same box then the number can be removed as a choice from the rest of the box (here the trigger is CP within a row/column).
4. **Naked twins**: if there are a set S of n < 9 numbers which are the only possibilities in exactly n squares of a unit, then these numbers can be discarded as option from remaining squares in the unit (as these options will be completely used up by squares in S). Note that this is a generalized version of the naked twin strategy.
5. **Hidden twins:** if there are a set S of n numbers (n < 9) which are restricted as options to exactly n squares in a unit, then any other options (beside these n numbers) in the squares in S can be discarded (because these options have no where else to go and so they will completely take up the squares in S allowing no other options to be used in S). Note that this is a generalized version of Hidden Twins Strategy.

Backtracking Search: Our backtracking search is very similar to what has been used in the class except for enhancements for collecting performance statistics like number of squares and options explored.

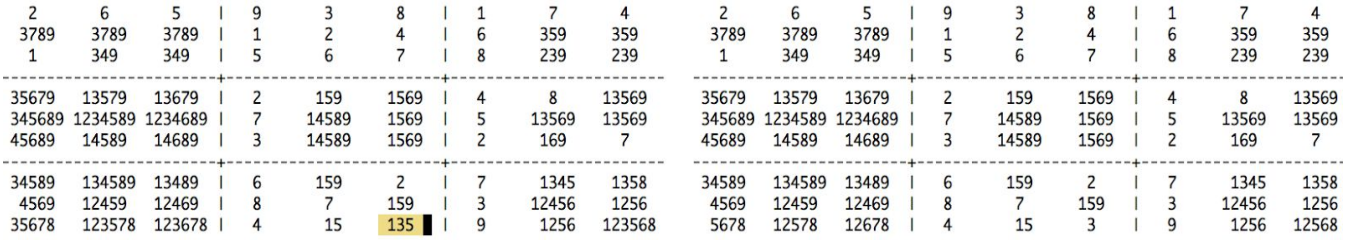
Heuristics:We have experimented with several Heuristics.

1. **MCV (Most constrained variable heuristic)**: We use the square which has the smallest domain (least number of digits available as options after constraint propagation).
2. **LCV (Least constrained value)**: Having chosen a square variable using MCV, we select the option (digit) which is present in the fewest peers of that square.
3. **Most Promising Square**: This is a heuristic for selecting the next square to explore for our Backtracking Search. The square which once filled lead us to fewest squares left after AC3 is selected.

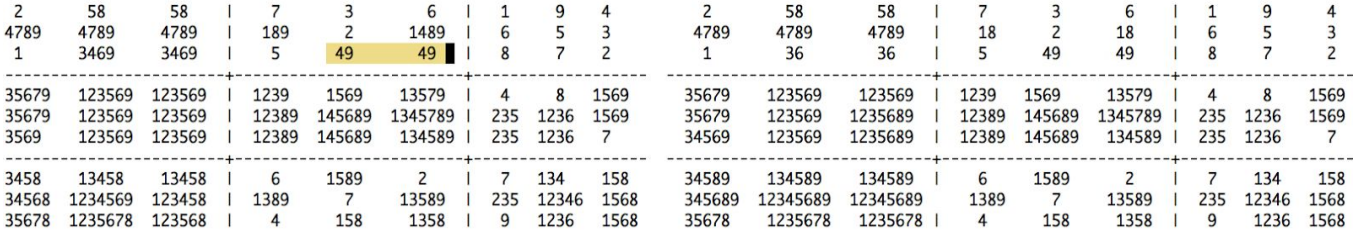
Concrete Example

Here we present some examples of our generalized AC3 strategies. These are not complete and remaining can be found in the Appendix. For discussion, we will refer to rows as A to I going top to bottom and columns as 1 to 9 going from left to right.

Single Possibility: if a unit has only one square left as a possibility for a value, then put it there - The figure below illustrates this case. The highlighted square I6 is the only square in its box (G-I, 4-6) which has 3 in its domain (no other squares in the box have 3 in their domains). So I6 is the only possible square for 3 in this box and so 3 can be safely discarded as an option from other squares in the box (the result of this elimination is shown on the left).



Naked Twins. Figure below illustrates this case. Here the twin squares C5 and C6 in row C have exactly 2 options, i.e 4 & 9. Thus both 4 & 9 must get assigned to these 2 squares and can be safely discarded from the domains of their peers (shown by the grid on the right - look at row C in the top middle box).



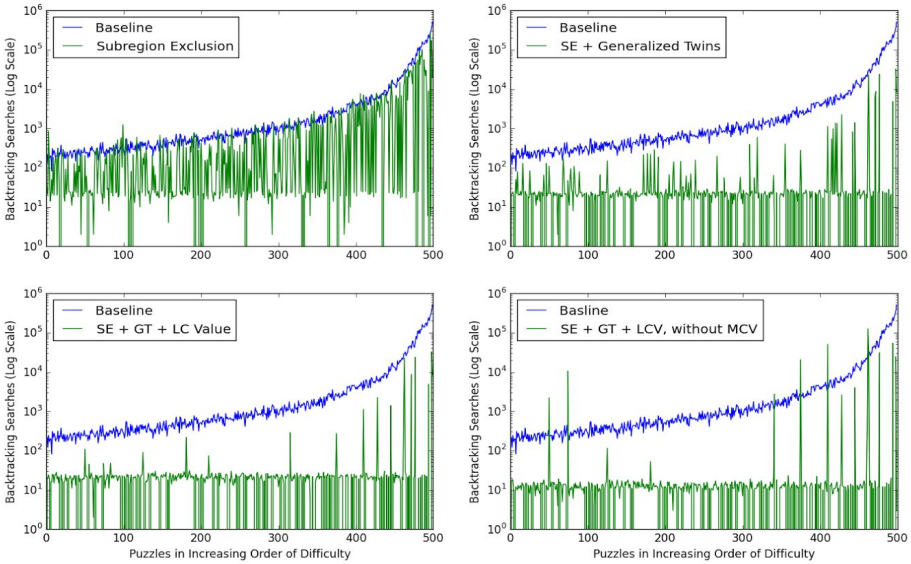
Results

Some of our generalized AC3 strategies for constraint propagation have done very well. The main reason is that they are able to reduce the search space to linear in the size of the grid because of the aggressive domain elimination they achieve.The results shown here are for the hardest 1683 puzzles selected from a million randomly generated puzzles (these are harder than some of the hardest puzzles posed by mathematicians). The table below compares various logic strategies & heuristics to the baseline in terms of raw time as well as the number of options explored (i.e. calls to backtracking search). Run times for our best combination of logic strategies and heuristics are 6-7 times faster than baseline and we explore 40 times less search space than the baseline.

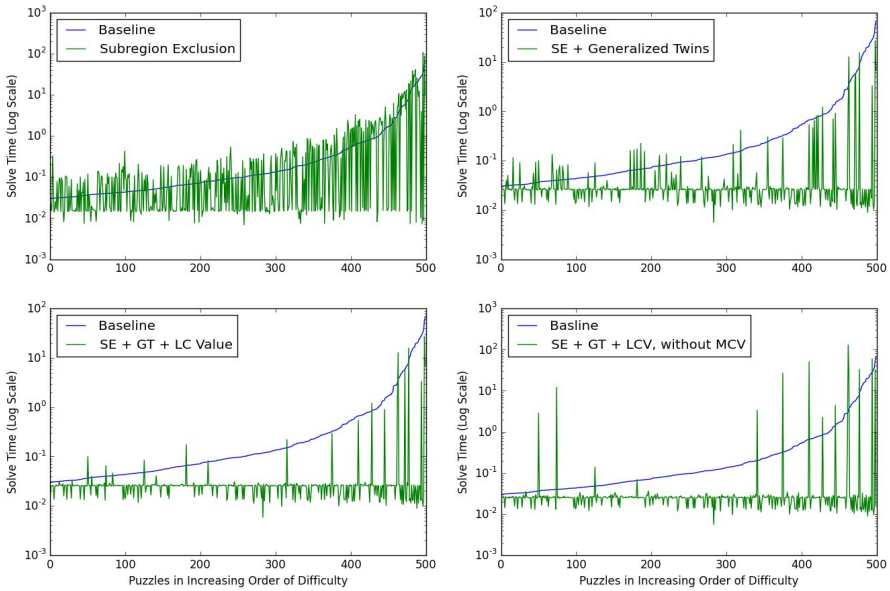
	Time (Seconds)	Time to Solve Hardest Puzzle (Seconds)	Total Squares Explored	Total Options Explored (calls to backtracking search)
Baseline (includes MCV)	775.99	67.74	2801249	5590249
Sub-Region Exclusion (SE)	754.54	108.54	911987	1803241
SE + Generalized Twins (GT)	128.84	26.88	94000	157544
SE + GT + LC Value	118.89	27.13	86361	141492
SE + GT + LC Value, No MCV	748.75	274.87	196923	646630
Opportunistic MCV	116.11	26.84	69096	121229

The results for hardest 500 puzzles (of the 1683) are compared in the plots below. The first set of 4 plots use number of backtracking searches as the metric. The next set of 4 plots use time as the metric. Note that the Y axis is plotted in log scale. Note that our heuristics on most puzzles including hard ones explores only linear amount of search space while the baseline seems to explore the exponentially growing tree space. The 4-plots compare the effect of various strategies and heuristics on the run time. Note that Subregion-exclusion by itself does not work very well though it is better than baseline on average. Generalized twins strategy is key to improving efficiency and gives a very significant performance gain. An

interesting observation (bottom right plot in the top set) is that the solver is efficient without MCV for the most part except on a few hard problems. This allowed us to derive a hybrid strategy called “opportunistic MCV” where we run the solver with MCV disabled by default and if the solver explores options beyond a threshold (50 worked best as a threshold) then we enable MCV are restart the solver. This helps us improve the efficiency by 15% on an already very efficient solution.



Comparison of Various Domain Reduction Strategies & Heuristics by # of Backtracking Searches



Comparison of Various Domain Reduction Strategies & Heuristics by Time

Machine Learning for Learning to Predict Next Best Move

For our CSP approach, during our backtracking search, we need to decide which square and what value in that square to explore next. In this section, we present our ML approach for learning to predict the next Square/Value to explore.

We used two strategies: ranking SVM as well as linear regression. Ranking SVM compares moves pairwise. Comparing two moves only makes sense if they are for the same partial grid, i.e. one node in the backtracking search tree.

To build our training set, we created a modified version of backtracking search. In this version, at randomly selected nodes in the search tree (which maps to a partially filled puzzle grid), we explore multiple Squares with multiple Values in each square irrespective of whether a solution has been found or not. Note that we do not explore every square and value at every node in the search tree because the run time to solve a single puzzle is exponential in the size of the grid. This prevents collection of a rich variety of data spanning a large number of puzzles in a reasonable time. Thus we randomly select points in the search tree (i.e partial grids) for exploration of comparable moves. The probability of random selection is also dependent on the depth of the tree because tree levels closer to the root have far fewer nodes than tree levels farther aways from the root. So we sample lot more aggressively at levels closer to the root than at levels away from the root.

For ranking SVM we take difference between every possible pair of moves for a given grid to create the final training set which is fed to the SVM algorithm. While for linear classifier we do not do this. Note that SVM compares a pair of moves while linear regression predicts a score.

The following list enumerates the features used in training as well as for prediction of best potential move at runtime. Except for tree depth, all our features are binary. The same features were used in training the ranking SVM as well as the linear classifier. The data sets were different as in the case of SVM we do pairwise differencing of moves while in the linear classifier we use the collected data as is.

- Empty squares in a unit (row/column/box) - discretized to test for = 2, <=4 and >4.
- Number of available digits (i.e. size of the domain of each square) summed across all the empty squares in a unit. Again discretized into three buckets (<=5, <= 10 and >10)
- Number of available digits for the square corresponding to the move under consideration (i.e. the one for which we are computing a ranking score)
- Is this the most constrained square on the Grid (i.e. with the smallest domain)
- Number of times the digit corresponding to the move under consideration occurs in the unfilled peers of the square under consideration for the next move.
- Is this the Least Constraining Value among all the available Values for the square under consideration for the next move (i.e. one with fewest occurrences across the domains of all the peers of the square under consideration).
- Tree depth of a node in the backtracking search.

The results were not very encouraging. We got 59% accuracy on both training as well as test set using SVM as well as Linear Regression. Also the time it takes to solve the puzzle is much worse due to the time it takes to extract the features across different potential moves and predict a score based on the learned predictor and then compare the result across multiple moves.

Algorithm (Genetic approach)

In this approach, we will use the strategy of assigning a random value from 1 to 9 to each unknown variables in the model, then in each iteration we will select, reproduce and maintain a population until we get solution, where the fitness function is equal to zero.

Fitness function

It is a particular type of objective function that is used to summarise, as a single figure of merit, how close a given design solution is to achieving the set aims. For the sudoku problem, we will define it as follows:

$$fitness(x) = \alpha \cdot sum(x) + \beta \cdot prod(x) + \gamma \cdot unique(x)$$

Where:

$sum(x) = \sum_{unit \in Units(x)} \left| 45 - \sum_{v \in unit} v \right|$, every row, column and box only allows summation of 45.

$prod(x) = \sum_{unit \in Units(x)} \left| 9! - \prod_{v \in unit} v \right|$, product of every row, column and box must be equal to 9!.

$unique(x) = \sum_{unit \in Units(x)} | \{1,2,3,4,5,6,7,8,9\} - \{unit\} |$, every row, column and box only consists of numbers between 1 and 9.

The parameters α, β and γ are weights that need to be calibrated in order to find the solution faster.

Steps

1. **Initialization:** Read the initial state of the Sudoku and create a template of genes. Each given value in the input (value different to '.') will set a gene in the template to an invariant value, so no mutation or crossover will change them. Example:

```
Input    :  .7.........12......53..1.....................6.....5...7.46...........7439.24.....
```

We process and create a template of genes.

```
Template:  .7.........12......53..1.....................6.....5...7.46...........7439.24.....
```

> **Note:** A different way to see the idea of this template is to think it as the concatenation of all the variables as: `{A1,…,A9,B1, …,B9,…,I1,…,I9}`. In the context of genetic algorithms, we will denote this sequence as the sequence of genes, so in this example the gene A1 has no value assigned and the gene A2 is fixed to 7.

Then, we create a population of N individuals using the template of genes, where each individual randomly assigns the non-invariant genes. Example:

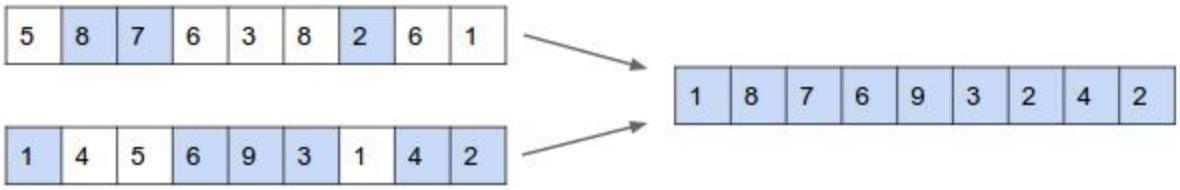
```
Individual 1:  176341098231223434653471976598074365982374566234365257774698164359786743962434531
Individual 2:  373746976891291076553671541978236345356748756197825323784646767097827743962456095
Individual 3:  477346592831276891053601197823641023674673866236745097774642396859683743942468352
...
Individual N:  575987346561278236453911746976898245762451076107655396724665312780423743972423968
```

2. **Parent Selection:** Using the value of the fitness function for each individual, we select the set of individuals (randomly selected or cherry-picked from the best fitness values).

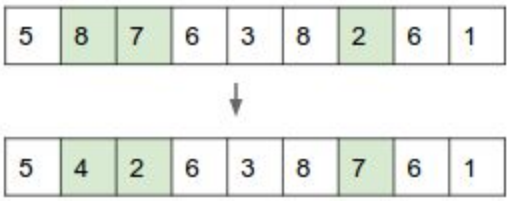
3. **Grow population:**

   We defined two operators, Crossover and Mutation.

   - **Crossover:** we go through each chromosome and then randomly select from each of the parents for each individual gene (some references call this uniform crossover).



   - **Mutation:** each gene in the child is iterated through and at each iteration, a random number r where $0 < r < 1.0$ is generated. If r is less than the parameter provided `mutate_rate`, then that gene is randomly changed to another valid value.



   We generate new individuals in the following ways.
   - By crossover between parents (previous step).
   - By mutating individuals of the current population.
   - A mix of both. (it applies a mutation to a result of a crossover).

4. **Selection:** We sort the new individuals using the fitness function and get the best N individuals to be the new population. If there is an individual with fitness value equal to 0, we are done, otherwise we go back to step 2.

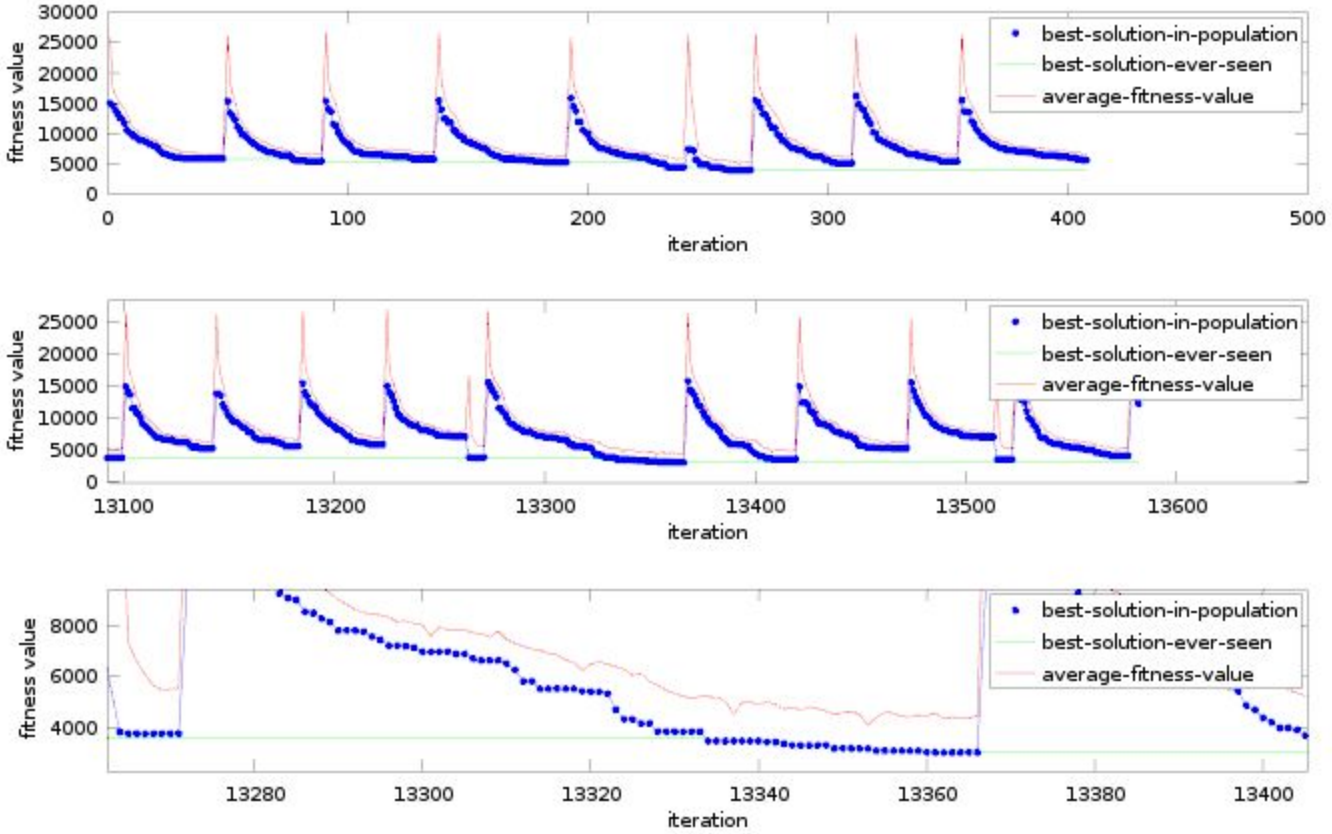
Considerations:

- Size of the population is very important. If it is too small, it does not allow enough diversity, and if it is too large, it cuts performance and keeps bad individuals longer.
- A too high mutation-rate disables the algorithm to perform specific changes to find the solution, while too small mutation-rate disables the algorithm to overcome local minimums. The perfect balance is to start with a small mutation-rate and progressively increase it as we identify a local minimum.
- When a local minimum is found, the complete population must die, otherwise individuals of the previous population could 'contaminate' the new population and reach the same local minimum.
- However, keeping the best candidates that were identified as local minimums could be use to create a new population (Example: after killing the complete population 10 consecutive times without beating the local minimum, we use all the previous local minimums). This helps in some cases to overcome the local minimum.

Partial Results (included in poster):

The best fitness value of the current population is shown in blue, the average fitness value is shown in red and the best found fitness value so far in green.

Size of population: **500**
Only top 20% of individuals gets the chance to reproduce.
Mutation rate: **0.005**, then increases by **0.005** per iteration if without improvement.
Kill entire population after **10** iterations if without improvement in the best fitness value.



Observations: the algorithm is continuously finding local minimum and trying to overcome them by killing the complete populations without success. After 6 hours of computing these implementation was not able to find the solution to the sudoku problem.

What was wrong?

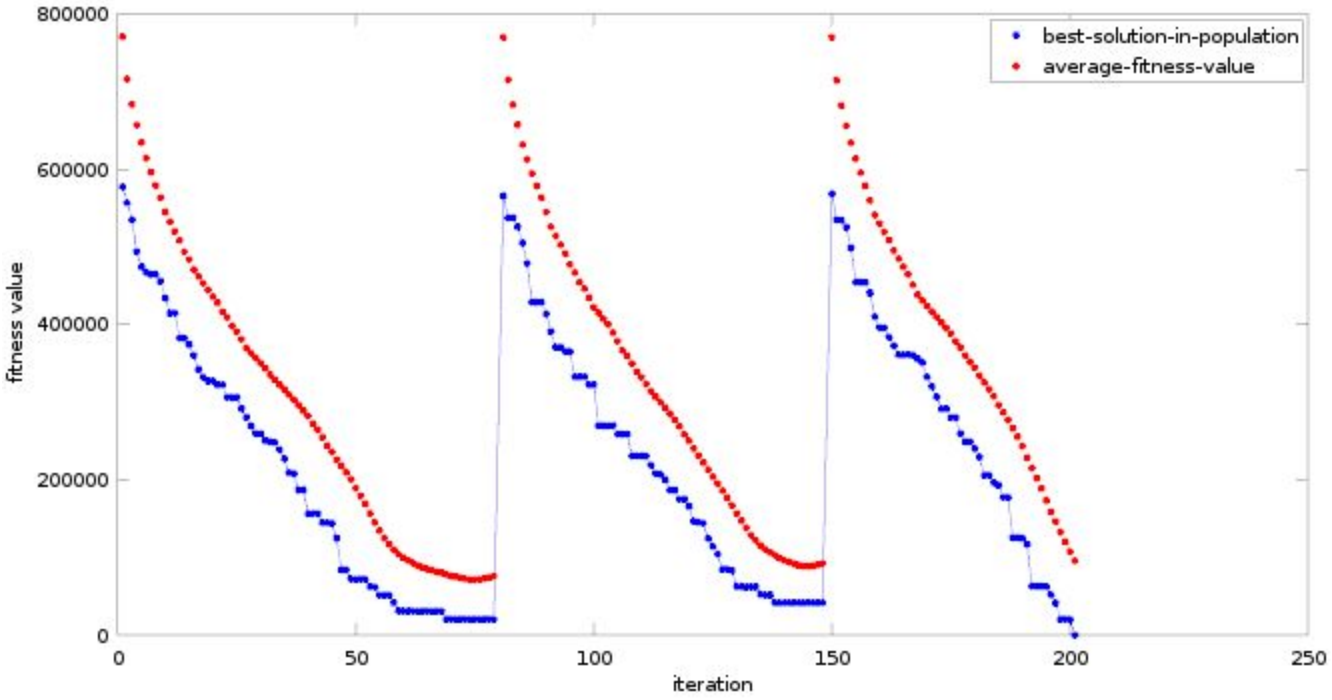
We identified that the population was not big enough, so we increased the population to 15000. We also noticed that reproducing only the best candidates was not very effective, so we decided to choose randomly the parents of the new generation.

Another thing that showed a huge improvement was to change the weights α, β and γ. For the results in the poster, we used $\alpha = 1,\ \beta = 1,\ \gamma = 50$. Notice that γ is bigger because it is just counting how many numbers between 1 and 9 are missing, so to give it more importance needed to be larger. However, we found that 50 was not large enough, so we increased it to 10000.

Final Results:

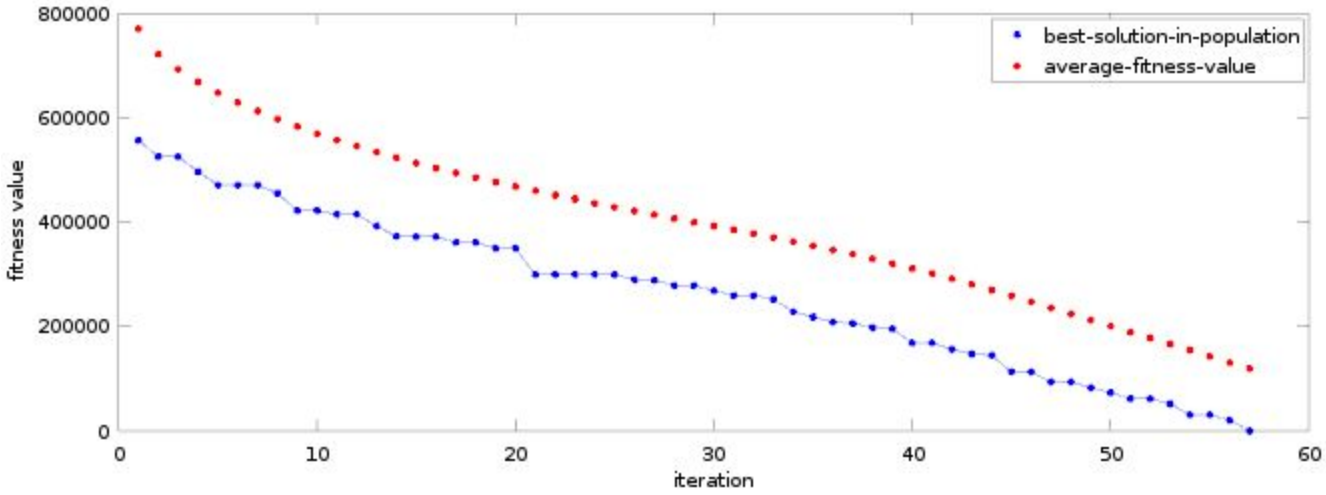
The best fitness value of the current population is shown in blue, while the average fitness value is shown in red.

Size of population: **5000**
Mutation rate: **0.005**, then increases by **0.005** per iteration without improvement.
Kill entire population after **10** iterations without improvement in the best fitness value.



Observations: the algorithm found the solution in **46 min** and killed twice the entire population.

Size of population: **20000**
Mutation rate: **0.005**, then increases by **0.005** per iteration without improvement.



Observations: the algorithm found the solution in **23 min** and there was no need to kill the population.

References

[1] http://norvig.com/sudoku.html