Sudoku Solver

Felipe Vital Cacique[1]

I. INTRODUCTION

Sudoku is a number based logic game that is very popular around the world. The default configuration consists of a table 9x9 which goal is filling all cells/units in a row, column and 3x3 blocks with numbers from 1 to 9, without repetition. The decision problem version of this puzzle is known to be in NP-complete [8], which is a class of hard problems. Thus the problem of solving a Sudoku is as hard or even harder than this, being unpractical for today's computers to compute. Just to exemplify, we would need to check O(9^{81}) combinations for solving the 9x9 Sudoku, and as the puzzle size gets bigger, the scenario is much worse. The inviability of using hard coded approach create a need for finding better solutions with lower time complexity.

This project proposed efficient algorithms to solve any Sudoku puzzle of size nxn, where $n = a^2$ and $a = \{1, 2, 3, 4...\}$. They were implemented and compared in terms of time complexity to a baseline algorithm.

II. SUDOKU RULES

The standard Sudoku is composed by 81 cells organized in a table with size 9x9 (Figure 1), which is divided in 9 smaller tables 3x3. The goal is to attribute numbers from 1 to 9 to the empty unities, satisfying the following rules [4]:

- Each row contains the numbers $1, 2, 3, 4, 5, 6, 7, 8, 9$, without repetition.

- Each column contains the numbers $1, 2, 3, 4, 5, 6, 7, 8, 9$, without repetition.

-Each small table 3x3 contains the numbers $1, 2, 3, 4, 5, 6, 7, 8, 9$, without repetition.

Fig. 1: 9x9 Sudoku.

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

[1]F.V. Cacique is with Faculty of Computer Science, Federal University of Minas Gerais, 6627 Antonio Carlos Ave, Brazil

III. BACKTRACKING (BASELINE)

Backtracking was used as a baseline algorithm to solve the Sudoku puzzle. According to Russell [3] "backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign". In other words, it fills each unit with a value from the domain, one by one, always checking at each step if it satisfies the problem constrains. In case it does not, the algorithm backtracks, changing the previous values attributed to be another candidate that satisfies the conditions. This process repeats until it finds a solution. The backtrack search is similar to a combinatorial approach, however it does not check all possible combinations that exists. It is because that once the algorithm get in a situation where the constrain is not satisfied, it does not need to investigate further. The code implemented is below.

Listing 1: Backtracking pseudo-code obtained from xx.

```
function BACKTRACKING_SEARCH(csp) returns a
    solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a
    solution, or failure
  if assignment is complete then
    return assignment
  var = SELECT–UNASSIGNED_VARIABLE(csp)
  for each value in ORDER_DOMAIN_VALUES(var ,
      assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences = INFERENCE(csp, var , value)
      if inferences != failure then
        add inferences to assignment
        result = BACKTRACK(assignment, csp)
        if result != failure then
          return result
    remove {var = value} and inferences from
        assignment
  return failure
```

The time complexity of the algorithm is O($n^{|b|}$), where $|b|$ is the number of the blank spaces, that can assume values from 1 to n^2. The worse situation would happen if all variables x are blank, and if the answer is the last combination the backtracking would try, so that it would try all existent sequences before finding the solution. However, in average it would take less time, being $|b|$ for the best case, if all of the first value assignments are correct and there is no need for backtracking.

There are some situations that we can combine the Backtracking to other techniques, and the blank cells domain will not be the same among all of them, some larger and other smaller. Thus we can also represent the complexity time as

O($\prod_{i=1}^{b} |d(b)|$), where $|d(b)|$ is quantity of numbers in the blank cells domains. It is a closer upper bound than our previous definition. From the formula, we can see that the Backtracking time is expected to increase exponentially in function of the number of blank spaces and the blank space domains (called by 'blanks' and 'r domain' respectively in future chapters)

IV. Sudoku as a Constrain Satisfaction Problem and AC-3

Sudoku puzzle can be modeled as a Constrain Satisfaction Problem (CSP). Basically it is a problem defined as a set of objects that needs to satisfy some constrains. The main idea for solving these problems is to eliminate large portions of the search space all at once by identifying value combinations that violate the constraints [3]. To formulate this puzzle as a CSP, firstly we need to define the variables as being the position of each unit of the Sudoku table (S) nxn, where $n = a^2$ and $a = \{1, 2, 3, 4...\}$.

$$X = \{S[1,1], S[1,2], S[1,3], ...S[n,n]\}$$

Secondly, we define the domain of each variable, which is the set $D = \{1, 2, 3, ...n\}$.

Finally, we explicit the constrains. There will be 3xn constrains:

One by each n columns:
AllDiff $(S[1,1], S[1,2], S[1,3], ...S[1,n])$
AllDiff $(S[2,1], S[2,2], S[2,3], ...S[2,n])$
...
AllDiff $(S[n,1], S[n,2], S[n,3], ...S[n,n])$

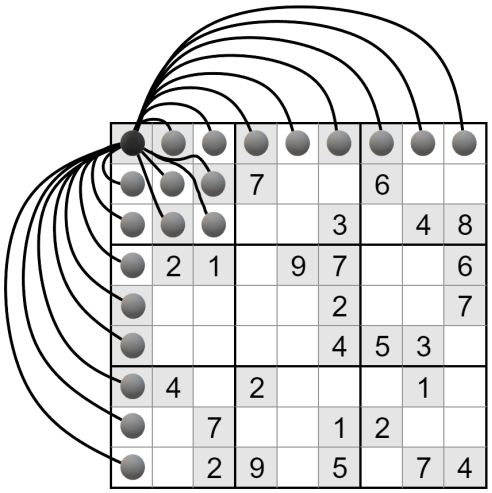
One by each n rows:
AllDiff $(S[1,1], S[2,1], S[3,1], ...S[n,1])$
AllDiff $(S[1,2], S[2,2], S[3,2], ...S[n,2])$
...
AllDiff $(S[1,n], S[2,n], S[3,n], ...S[n,n])$

One by each n sub matrix of size a x a:
AllDiff $(S[1,1], S[1,2], S[1,3], ...S[1,a]$,
$S[2,1], S[2,2], S[2,3], ...S[2,a]$,
...
$S[a,1], S[a,2], S[a,3], ...S[a,a])$

The notation AllDiff means that the variables given in the set has to be all different from each other (same notation as used by Russell [3]).
The CSP can be visualized by the constrain graph. The vertex of the graph corresponds to the variables X, while the links connects the variables that participate in the constrain (Figure 2).

There is a general purpose algorithm that can be used for CSP problems, that is called Arc Consistency algorithm (AC-3). Russel [3] explained it as follows:

Fig. 2: Sudoku constrain graph visualization for a standard configuration. The black vertex in the corner is connected by edges to all the vertex in its row, column and sub matrix 3x3. The connections for all the other vertex follow the same idea.



The most popular algorithm for arc consistency is called AC-3 (see Figure 6.3). To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. (Actually, the order of consideration is not important, so the data structure is really a set, but tradition calls it a queue.) Initially, the queue contains all the arcs in the CSP. AC-3 then pops off an arbitrary arc (Xi, Xj) from the queue and makes Xi arc-consistent with respect to Xj . If this leaves Di unchanged, the algorithm just moves on to the next arc. But if this revises Di (makes the domain smaller), then we add to the queue all arcs (Xk, Xi) where Xk is a neighbor of Xi. We need to do that because the change in Di might enable further reductions in the domains of Dk, even if we have previously considered Xk. If Di is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP - they both have the same solutions - but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains

Listing 2: Arc Consistency (AC-3) pseudo-code obtained from xx with complexity analysis

```
function AC_3(csp) returns false if an
    inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D
      , C)
  local variables: queue, a queue of arcs,
      initially all the arcs in csp
```

```
while queue is not empty do // O(n*n)
  (Xi, Xj ) = REMOVE_FIRST(queue)
  if REVISE(csp, Xi, Xj ) then // O(n)
    if size of Di = 0 then return false
    for each Xk in Xi.NEIGHBORS - {Xj} do // O
        (3*n-1)
      add (Xk, Xi) to queue
return true // Total = O(n*n*n*n)

function REVISE(csp, Xi, Xj ) returns true if we
    revise the domain of Xi
  revised = false
  for each x in Di do // O(n)
    if no value y in Dj allows (x ,y) to satisfy
        the constraint between
    Xi and Xj then
      delete x from Di
      revised = true
  return revised
```

Analyzing the code above we can obtain the AC-3 time complexity that is O(n^4). It is a polynomial complexity, which is much faster than the exponential Backtracking complexity. Unfortunately AC-3 alone is not able to completely solve all Sudoku puzzles we might give (only easy ones), however the AC-3 can drastically reduce the variable domains for each variable x and complete many of the blank xs. Thus we clearly see that combining the AC-3 with Backtracking at the end is expected to be much more efficient than only using Backtracking.

V. A FEW STRATEGIES (HEURISTICS)

Solving a Sudoku requires a significant amount of time and techniques, which increase with the difficulty level. Solving hard Sudoku may require the use of different set of strategies compared to easy ones [7]. The best approach is starting by using the basic techniques in order to reduce the most of the variable domains, and then use the advanced strategies as needed. When all the technique sets were used, start from the basic ones again, repeating all the process over and over again. Many of the Sudoku puzzles can be solved already by using this approach, and if it does not, it will, at least, reduce substantially the state space [7].

[6] and [7] shows a few strategies for solving Sudoku. This work used 3 of them, which will be shown next, by order of simplicity:

1) OCCULT SOLE CANDIDATE: Sometimes, if we look very carefully, we can find an "occult sole candidate". This number is the unique possible candidate in a row, column or block, but it appears mixed with other numbers. See the image bellow.

Fig. 3: OCULT SOLE CANDIDATE

In the image, we can see that the number 1 appears only once in its respectively block. This indicates that the number 1 must be placed in its position.

Listing 3: OCULT SOLE CONDIDATE pseudo-code

```
function OCULT_SOLE_CONDIDATE(csp) // Total = O(n
    *n*n)
  groups = columns + rows + bloks
  for each value in domain do // O(n)
    for each group in groups do // O(n + n + n)
      if CHECK_IF_OCULT_SOLE_CONDIDATE(group) ==
          true then // O(n)
        REVISE(value, group) // O(n)
```

From the code above we can see that the algorithm identify all OCULT SOLE CONDIDATES in O(n^3).

2) NAKED SUBSET: If at any moment there is a same pair of numbers in the cells of a group (row, column or block), then this pair must necessarily appear in those 2 cells, as we can see in the image bellow.

Fig. 4: Identifying the NAKED SUBSET

In this image, the numbers 1 and 3 appears alone in 2 cells, thus they must be placed there. However, we do not know which number will be place in which cell, but we know that the numbers 1 and 3 cannot appear in the other blank cells. Thus, we can remove these numbers from their domains.

Fig. 5: Removing NAKED SUBSET numbers from other set domains

Listing 4: NAKED SUBSET pseudo-code

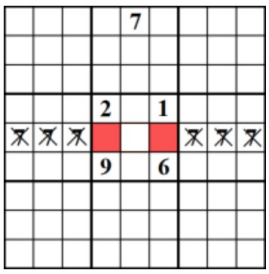
```
function NAKED_SUBSET(csp) // Total = O(n*n*n*n)
  groups = columns + rows + bloks
  for each group in groups do // O(n + n + n)
    naked_subset, is_naked_subset =
        CHECK_IF_NAKED_SUBSET(group) // O(n*n*n)
    if is_naked_subset == true then
      REVISE(naked_subset, group) // O(n)
```

```
function CHECK_IF_NAKED_SUBSET(naked_subset, group
    )
  for each pair of domain numbers do // O(n*n)
    if COUNT_PAIR_IN_GROUP(pair, group) == 2 then
        // O(n)
      naked_subset = pair
      return naked_subset, true
  return none, false
```

The complexity of time of the OCULT SOLE CONDIDATE heuristic is O(n^4).

3) BLOCK AND COLUMN/ROW INTERACTION: This method helps with removing numbers from the domains of the blank cells. The example bellow shows that number 7 can be inserted in the red calls of the middle row. Thus we can remove 7 as a possible candidate from the rest of row.

Fig. 6: BLOCK AND COLUMN/ROW INTERACTION



In a more general way we can state this technique in the following way: Let set A intersect the set B, and value is a number from the domain. If value is not in B - A, then value must be in A∩B. Thus, as value is in A∩B, value cannot be in A - B. Therefore we can remove value from wherever value appears in A - B. The pseudo-code of the implementation is below.

Listing 5: BLOCK AND COLUMN/ROW INTERACTION pseudo-code

```
function BLOCK_COLUMN_ROW_INTERACTION(csp) //
    Total = O(n*n*n*n)
  As = columns + rows
  for A in As do // O(2*n)
    for B in blocks do // O(n)
      if A intersects B // O(n)
        for each value in domain do // O(n)
          if value not in B–A then // O(n)
            remove value from A–B // O(n)
```

The complexity of time of the BLOCK AND COLUMN/ROW INTERACTION strategy is O(n^4).

VI. METHODOLOGY

The project consists in the implementation of three different approaches for solving Sudoku puzzles in Python Language. They were tested in Sudoku of sizes 9x9, 16x16 and 25x25, with 5 different difficulty levels, which are Very Easy, Easy, Medium, Hard and Very Hard, that were acquired in the Planet Sudoku website[5]. Larger Sudoku sizes than those ones are not easily found.

The thee approaches implemented and evaluated were:

1) Backtracking (B): Backtracking algorithm applied only (explained in section III).
2) AC-3 with Backtracking (AC3B): Firstly we use the AC-3 to reduce the numbers of the blank cells domain, and then we apply the backtracking for solving the remaining blank cells.
3) AC-3 with Heuristics and Backtracking (AC3HB): Use not only the AC3, but also three strategies for solving Sudoku, which were 'Occult Sole Candidate', 'Naked Subset' and 'Block and Column/Row Interaction' (Section V). Combined to AC-3, they were applied in this order over and over again, for as many iterations as needed, until they did not do any better. We noticed that doing that is sufficient for solving many of Sudoku puzzles. For the situation where it is not enough, we apply the backtracking for finishing to solve the puzzle. Surely the AC-3 and the heuristics takes some amount of time, but it is considerably faster in comparison to applying the Backtracking only. The greater the % of reduction, the better.

The algorithms were evaluated in terms of three metrics. Firstly, is the time complexity, that is what we are more interested in. In the tests, we took the best out 5 measurements (except when time was above 10 min). In practical experiments the time might not be easy to measure for some examples, because it grows exponentially. Thus, in addition we are measuring two others metrics, that are the number of blank cells ('blanks') and the sum of all numbers in the remaining domain of all blank cells (remaining domain or 'r domain'). Using the last 2 metrics we also calculated the % of reduction by %reduction = 100%*blanks(reduced by AC3 or AC3+Heuristics)/blank(initial), that is how much of the blank cells and the remaining domain were reduced by the AC3 and other heuristics before applying backtracking.

In Appendix B we showed a simple Genetic Algorithm implementation in order to test its performance compared to Backtracking. It is not in the body text of this project because that are only the first tests and results using GA. There are still a lot more to experiment.

VII. TESTS AND ANALYSIS OF RESULTS

For tests, the algorithms Backtracking (B), AC-3 with Backtracking (AC3B) and AC-3 with Heuristics and Backtracking (AC3HB) were applied to solve Sudoku of sizes 9x9, 16x16 and 25x25 for 5 different levels. The experiments limit time was 30 minutes for each puzzle. The results is shown in the Table I and some of the solved Sudoku is in Appendix A.

The 'blanks' and 'r domain' in the Backtracking B are the initial values of blank cells and remaining domain of the untouched Sudoku. Notice that 'blanks' corresponds to around 50 to 60% of the total number of cells, growing proportional to the Sudoku size, and these variables in a Sudoku of fixed size are close, independent of the difficulty level. The table shows that B easily solved any given 9x9 Sudoku, with time varying from 0.05s (Very Easy) to 0.73s (Medium). As expected the Sudoku hardness does not

TABLE I: Results for the algorithms Backtracking (B), AC-3 with Backtracking (AC3B) and AC-3 with Heuristics and Backtracking (AC3HB), solving Sudoku of sizes 9x9, 16x16 and 25x25 for 5 different levels. The columns 'blank' and 'r domain' stands, respectively, for the number of blank cells and the remaining domain (sum of all remaining domain over all blank cells) before applying the backtracking. Notice that the time complexity of the backtracking is function of the blank cells and the remaining domain, so the lower those values, the backtracking is expected to perform exponentially faster. The total time spent for solving the puzzle for the 3 different techniques is also shown below. The measures that exceeded a limit time of 30 min was represented by '–'. The '% reduction' is how much of the blank cells and the remaining domain were reduced before applying backtracking, calculated by % reduction = blanks(reduced by AC3 or AC3+Heuristics)/blank(initial).

Sudoku		**B**			**AC3B**			**AC3HB**		
Size	**Level**	**blanks**	**r domain**	**Time(s)**	**blanks**	**r domain**	**Time(s)**	**blanks**	**r domain**	**Time(s)**
	Very Easy	52	468	**0.05**	0	0	0.08	0	0	0.09
9x9	*Easy*	48	432	0.31	48	164	0.15	0	0	**0.11**
81	*Medium*	54	486	0.73	54	202	0.33	0	0	**0.22**
Cells	*Hard*	51	459	0.24	51	172	0.15	0	0	**0.13**
	Very Hard	53	477	0.11	48	162	**0.09**	42	122	0.11
	Very Easy	150	2400	–	0	0	**0.92**	0	0	0.93
16x16	*Easy*	148	2368	–	142	586	1761.01	0	0	**1.14**
256	*Medium*	148	2368	–	146	645	–	95	300	**14.27**
Cells	*Hard*	154	2464	–	148	700	–	98	316	**35.44**
	Very Hard	164	2624	–	164	853	–	138	599	–
	Very Easy	323	8075	–	0	0	**5.27**	0	0	5.63
25x25	*Easy*	328	8200	–	324	1610	–	0	0	**9.16**
625	*Medium*	327	8175	–	321	1658	–	137	447	**608.95**
Cells	*Hard*	329	8225	–	326	1675	–	0	0	**15.88**
	Very Hard	339	8475	–	339	1844	–	169	569	**36.84**
Average		177.87	3713.07	–	140.73	684.73	–	**45.27**	**156.87**	–
% Reduction		0%	0%	–	21%	82%	–	**75%**	**96%**	–

interfere in time, because the Backtracking is function of 'blanks' and 'r domain'. B was not able to solve any of the 16x16 and 25x25 sizes.

The AC-3 in AC3B reduced the number of blank cells in 21% and the remaining domain in 82%, reducing consequently the time spent for solving all 9x9, compared to B. We can see that the AC-3 was able to solve the Very Easy level for all Sudoku sizes (AC-3 reduced 'blanks' and 'r domain' to 0) without needing backtracking. However the AC3B still does not solve all games.

The AC3HB is the best between all of the three. It reduced the number of blank cells in 75% and the remaining domain in 96%, being able to solve completely 14 out 15 games. Notice that in 9 of them it solved without even needing to perform backtracking, which is thanks to the AC-3 combined to the heuristics. We can see that for the AC3HB, the hardest the puzzle, higher is the time for solving it. It turns out that the metrics 'blanks', 'r domain' and 'time' can be possible used to estimate a difficulty level of a Sudoku of an unknown level. To confirm this hypothesis we would need, however, more tests and statistic validation.

VIII. CONCLUSION

The project consisted on the implementation an comparison of the algorithms of Backtracking (B), AC-3 with backtracking (AC3B) and AC-3 with heuristics and backtracking (AC3HB) for solving Sudoku puzzles of sizes 9x9, 16x16 and 25x25, with 5 different difficulty level. The AC3HB was the best of the three, solving 14 out 15 puzzles in comparison to 5 out 15 for B. In general, the experiment results were consistent to the theoretical analysis of time complexity. Future work would be adding more strategies to the heuristics of the AC3HB and verify the possibility of implement Genetic Algorithm to replace the backtracking.

REFERENCES

[1] Mantere, T. & Koljonen, J. (2007). Solving, rating and generating Sudoku puzzles with GA.. IEEE Congress on Evolutionary Computation (p./pp. 1382-1389), : IEEE.

[2] Perez, M., & Marwala, T. (2008). Stochastic Optimization Approaches for Solving Sudoku. CoRR, abs/0805.0697.

[3] Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[4] Regra de Sudoku, available via http://pt.sudokupuzzle.org/ (cited 21/04/2018).

[5] Planet Sudoku, available via http://planetsudoku.com/ (cited 23/05/2018).

[6] Geniol. Como jogar Sudoku, available via https://www.geniol.com.br/logica/sudoku/tutorial/ (cited 23/05/2018).

[7] Kristanix. Tips on Solving Sudoku Puzzles - Sudoku Solving Techniques, available via https://www.kristanix.com/sudokuepic/sudoku-solving-techniques.php (cited 24/06/2018).

[8] Yato. (2003). COMPLEXITY AND COMPLETENESS OF FINDING ANOTHER SOLUTION AND ITS APPLICATION TO PUZZLES. Master Thesis.

IX. Appendix A

In this section we will show a few Sudoku puzzles solved by the algorithm AC3HB.

TABLE II: Very Hard Sudoku 9x9 solved by AC3HB in 0.11s. The bold characters are the initial filled cells.

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

TABLE III: Medium Sudoku 16x16 solved by AC3HB in 14s. The bold characters are the initial filled cells.

E	**G**	2	**1**	A	4	F	6	B	7	3	8	**5**	9	**D**	C
9	5	**3**	A	**8**	**2**	G	B	E	F	**C**	**D**	6	**4**	1	7
4	**F**	**7**	6	1	3	C	**D**	**2**	A	5	9	E	**B**	**G**	**8**
B	8	D	**C**	**9**	5	**7**	**E**	**4**	**1**	6	**G**	**2**	F	3	A
A	**6**	1	B	7	**G**	E	**4**	**F**	D	**8**	5	3	C	**9**	2
5	4	8	**3**	**6**	D	**1**	9	A	**2**	G	**C**	**F**	E	7	**B**
C	E	**G**	F	**2**	A	B	**3**	**1**	6	9	**7**	4	**5**	8	D
2	D	9	**7**	C	**F**	5	**8**	**3**	B	**E**	4	**1**	6	A	**G**
1	3	A	**5**	G	**E**	8	**C**	**D**	4	**2**	F	**9**	7	B	**6**
G	B	**C**	8	**3**	7	6	**F**	**9**	5	1	**E**	A	**D**	2	4
F	9	E	**4**	**D**	B	**2**	A	8	**3**	7	**6**	**G**	1	C	**5**
D	**7**	6	2	5	**9**	4	**1**	**C**	G	**A**	B	8	3	**F**	E
8	A	5	**9**	**B**	6	**3**	**7**	**G**	**C**	4	**1**	**D**	2	E	F
7	**2**	**F**	G	E	C	A	**5**	**6**	9	D	3	B	**8**	**4**	**1**
6	1	**4**	D	**F**	**8**	9	G	7	E	**B**	**2**	C	**A**	5	3
3	**C**	B	**E**	4	1	D	2	5	8	F	A	**7**	G	**6**	9

TABLE IV: Very Hard Sudoku 25x25 solved by AC3HB in 37s. The bold characters are the initial filled cells.

A	9	**8**	4	**I**	7	2	3	1	**5**	B	**K**	O	G	**P**	**M**	F	C	**6**	**H**	L	J	**D**	**E**	**N**
M	D	**F**	6	7	I	**L**	H	**C**	**P**	**J**	E	**1**	3	**9**	**K**	8	G	4	N	A	5	**2**	O	**B**
J	**3**	**N**	L	5	4	E	G	**6**	**M**	**2**	A	I	8	**F**	**B**	**1**	**O**	D	7	**H**	**P**	K	C	**9**
1	O	**E**	**B**	H	F	K	8	**9**	D	**C**	M	**5**	**N**	6	A	J	**2**	L	**P**	**I**	4	3	7	G
K	G	P	C	2	**A**	**N**	**J**	**B**	O	**H**	**D**	**4**	L	**7**	3	**E**	I	9	5	6	F	**8**	1	M
L	6	2	**P**	**C**	3	**O**	**I**	5	**J**	F	**B**	D	A	G	7	4	M	**1**	**E**	9	8	**H**	**N**	K
N	**M**	**G**	**A**	**O**	D	**1**	E	L	B	8	C	**H**	**J**	5	**2**	**9**	P	**K**	6	7	**I**	4	**3**	**F**
H	**J**	**4**	1	**D**	**9**	**6**	**P**	F	**8**	**M**	**I**	**7**	**K**	N	**5**	G	**L**	3	A	O	2	**C**	B	E
9	**5**	**I**	8	3	G	7	**K**	N	2	**L**	4	E	**P**	O	**D**	**C**	**H**	**B**	**F**	J	**6**	**A**	M	**1**
E	F	B	7	**K**	H	**C**	**4**	M	**A**	**6**	**2**	3	9	**1**	O	**N**	J	I	8	**D**	G	**5**	**P**	**L**
I	C	6	9	1	**L**	5	**2**	**7**	E	**G**	**3**	**P**	D	B	F	**O**	K	8	J	N	A	**M**	**4**	H
F	**K**	7	H	**G**	**8**	M	**O**	4	**N**	**1**	**5**	J	C	E	P	**D**	**A**	2	9	**3**	**B**	I	L	6
5	P	**O**	M	**L**	**K**	D	9	**A**	I	4	N	F	6	8	G	**B**	7	H	**3**	**E**	1	**J**	2	**C**
8	2	3	**N**	**J**	1	H	**B**	**P**	F	A	9	K	**I**	**L**	**6**	M	**4**	E	**C**	**G**	7	O	**5**	D
4	**A**	**D**	E	B	6	J	C	**G**	3	O	H	**M**	**7**	**2**	I	**L**	**N**	5	**1**	P	K	9	F	**8**
D	**4**	**5**	K	**8**	N	B	1	**H**	L	**I**	O	6	**F**	**A**	**E**	P	**3**	**C**	2	**M**	9	G	J	**7**
B	I	**C**	**O**	A	**J**	**G**	**M**	**2**	**6**	3	**1**	8	E	**H**	N	5	**9**	7	K	4	L	**F**	**D**	P
3	1	**H**	G	E	O	P	**7**	8	**K**	9	**J**	**L**	**B**	**D**	**4**	A	**F**	**M**	**I**	**5**	C	**N**	**6**	2
2	**7**	L	**J**	M	E	**I**	F	**3**	**9**	N	**P**	**C**	5	4	8	6	B	**G**	D	**K**	**O**	**1**	**H**	**A**
6	**N**	**9**	F	P	**5**	**4**	A	D	C	K	7	G	**2**	M	**L**	H	**1**	**J**	O	**B**	**3**	E	8	I
C	E	**K**	5	F	B	3	L	**O**	4	**P**	6	**9**	**H**	**I**	1	**2**	**D**	**N**	**G**	8	M	7	A	J
O	L	1	I	**6**	**P**	A	**D**	J	H	7	**G**	**N**	4	**3**	C	**K**	8	F	M	2	**E**	**B**	9	5
7	H	M	**2**	**9**	C	8	**6**	**E**	**G**	**D**	L	A	1	**K**	**J**	**3**	5	O	B	F	N	**P**	**I**	4
G	8	**A**	3	4	2	9	N	I	**1**	**5**	F	**B**	M	**J**	**H**	**7**	E	**P**	L	C	D	**6**	K	O
P	**B**	**J**	D	N	**M**	**F**	5	K	**7**	**E**	8	2	**O**	C	**9**	I	6	A	4	**1**	H	**L**	G	**3**

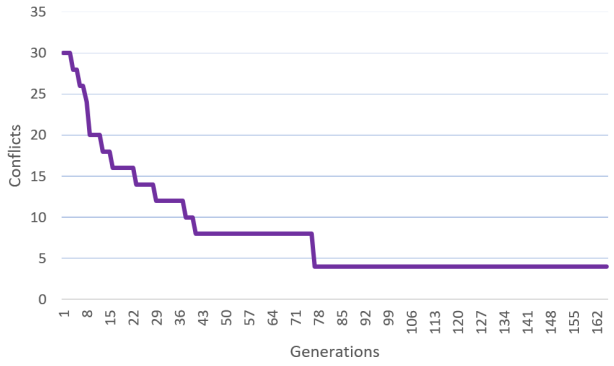
X. Appendix B

This section show our first implementation of a Genetic Algorithm to solve Sudoku as follow.

1) Creates a population of 40 individuals, where the individuals are the puzzle with random initialization of the variable, using only values that is in their domains. Each genome is composed by the set of empty cells.
2) The crossover is done in 80% of the population. It exchanges the variable xi from the same position in the table, with probability of 50%.
3) The mutation gives a random value from the xi domain, for all xs, with probability of 8%.
4) The fitness function is calculated in terms of the number of conflicts between the variables, when they do not respect the constrains. Notice that if it is zero, then all constrains are satisfied and the game is successfully solved.

The graph of learning using GA is bellow.

Fig. 7: GA solving a Very Hard 9x9. Number of conflicts decrease with time, and find its local minimum after 80 generations.



Any of the tests performed could not find the optimal solution. They all stacked in a local minimum, which is one of the characteristics of a GA. For example, in the graph above, the best solution found after 80 generations has 4 conflicts. To solve this problem we can try to encode the genomes in another way, such as Mantere [1], or implement something such as a hybrid of Genetic Algorithm and Simulated Annealing (HGASA) (Perez [2]).

TABLE V: Approximate solution for the Very Hard Sudoku 9x9 solved using a simple Genetic Algorithm implementation. See Table II from Appendix A for comparison.

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