Combining Metaheuristics and CSP Algorithms to solve Sudoku

Marlos C. Machado and Luiz Chaimowicz
Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brasil

Abstract

Sudoku is a very popular puzzle game that is played by millions of people everyday. In spite of that, it is a NP-Hard problem that can be very difficult to solve depending on the initial conditions of the board. In this paper, we propose the combination of metaheuristics with techniques from the Constraint Satisfaction Problem (CSP) domain that speed up the solution's search process by decreasing its search space and its processing time. Experiments performed with boards of size 3, 4 and 5 show that this approach allows the resolution of a greater number of instances when compared to an initial baseline.

Keywords:: Constraint Satisfaction Problems, NP-Complete Problems, Metaheuristics, Sudoku

Author's Contact:

{marlos, chaimo}@dcc.ufmg.br

1 Introduction

Sudoku is a puzzle being increasingly widespread in the world and a very interesting problem to the scientific community for its global appeal and for being contained in the class of NP-Complete problems [Ist et al. 2006].

It is organized in a board composed of $n \times n$ squares and each square is composed of $n \times n$ cells, so $n^2 \times n^2$ is the total number of cells. We call n the board order. Normally, puzzles of order 3 are the most common.

The player goal is to fill all board cells with numbers ranging from 1 to n^2 without repeating numbers in a line, column and square. These are the problem constraints. Initially a cell may be blank or pre-defined, *i.e.*, filled with an absolute number. Each different initial board is a specific problem instance.

There are many papers dealing with Sudoku but most of them require instances built in a way that they can be solved only with logical rules [Lewis 2007], in general these restrictions allows these boards to be solvable by humans. We believe this restriction is too strong. A much more flexible approach not having this restriction, *i.e.* aiming to solve the largest number of instances, regardless of the way they were designed is presented in [Lewis 2007] that treats the puzzle as an optimization problem and solves it with *simulated annealing*.

Due to this flexibility allowed by the approach presented in [Lewis 2007] this approach proved itself very interesting for study and improvement. Based on this, our contribution in this work is the proposition of heuristics that are able to solve more puzzle instances in less time.

Our approach consists in the use of constraint satisfaction algorithms allied to the *simulated annealing* approach. The constraint satisfaction algorithms decrease the solutions' search space, increasing the whole heuristic efficiency since it increases the number of solved instances and decreases the time spent to solve them.

This work is organized as follow: we present a deeper discussion about its related works in the next section while we present the *simulated annealing* approach in the third section. In the fourth section we discuss the proposed heuristics. Section 5 presents the experimental setup and Section 6 discuss our results. We conclude this work in Section 7.

2 Related Work

An interesting paper that presents the Sudoku as a constraint satisfaction problem is [Simonis 2005] that compares the different propagation schemes for the Sudoku solving. It also presents other solution approaches without searching. Two more recent papers that see Sudoku as a constraint satisfaction problem and that describe some of the techniques used in our approach are [Crook 2009] and [Russell and Norvig 2009].

Another traditional approach to Sudoku solving is the puzzle modeling as a satisfiability problem (SAT). Two papers which have done this modeling are [Weber 2005] e [Ist et al. 2006]. The first used a theorem prover to get valid solutions while the second used some inference techniques for operations in the conjunctive normal form (CNF). In general these techniques face some difficulties to solve boards without special properties.

There are also approaches based on natural computing as [Moraglio et al. 2006] and [Sato and Inoue 2010]. Both works discussed genetic operators to the problem. Moraglio and Togelius [2007] also proposed the Sudoku resolution with *geometric particle swarm optimization*. In theory these approaches would be able to solve any problem instance but the papers' experimentation did not envision boards with order higher than three. In addition to this restriction, there is no analysis about its execution time (one of our contributions) what made these two works less interesting for our study.

Other interesting works are [Agerbeck et al. 2008] and [Moon et al. 2009]. The first presented a multi-agent approach to the resolution of NP-Complete problems, exemplified by the Sudoku resolution; while the second presented a probabilistic representation of Sudoku, being based on *Sinkhorn balancing* techniques to the puzzle solving.

Finally, the approach we studied and optimized in this paper has been presented in [Lewis 2007] that used metaheuristics to solve the Sudoku. This work is very interesting because it works with boards with different orders and it is able to solve a large number of problems. He also proposed a method for instance generation that allows an experimentation that goes beyond the problems published to human-beings. The author uses *simulated annealing* techniques as a search space solution. This work is deeply discussed on next section. Differently from others, our paper combine different approaches getting the best of each: the search space reduction with constraint satisfaction algorithms and the search with simulated annealing, what increases the number of solved instances and decreases the time spent to solve them.

3 The Simulated Annealing Approach

As we already discussed in the previous section, the simulated annealing approach was chosen because it is more robust than most of other discussed heuristics since it does not depend of instances that cannot be solved by logical rules. It was presented in [Lewis 2007] where the author has modeled the puzzle as a minimization problem.

To formulate the algorithm we need to define an evaluation function, a neighborhood operator and an initial solution (not necessarily correct). We present these concepts below.

The initial solution is obtained with the completion of all board blank cells. "This is done randomly, but in such a way so that when the grid is full, every square contains the values 1 to n^2 exactly once" [Lewis 2007].

While the board is being filled the neighborhood operator is executed. It repeatedly chooses two different cells in the same square and swaps them. It is said in [Lewis 2007] that the goal of these changes is to "stop sampling bias towards squares with less non-fixed values". The neighborhood operator ensures that the third criterion of the Sudoku is always met because the generation of an initial solution satisfies the squares' restriction and this operator keeps it satisfied since it only exchanges cells in the same square.

The evaluation function was defined as the sum of the missing numbers in each column and row (a total of 18 values). Naturally, an acceptable solution has cost equals to 0. Figure 1, extracted from [Lewis 2007] is an example.

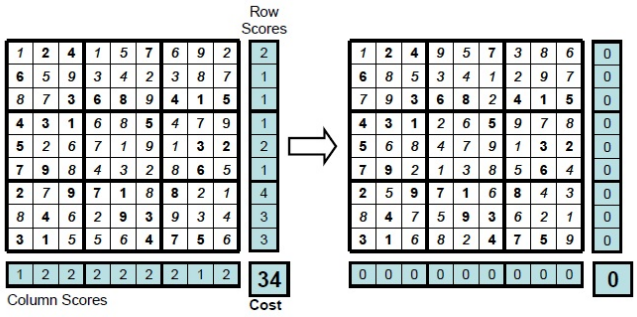

Figure 1: *Cost example for different puzzle instances*

The simulated annealing executes as follow: after its neighborhood operator, initial solution and evaluation function have been defined we apply the neighborhood operator successively. The new states (after the operator have caused a board alteration) are accepted under two different conditions:

- The presented solution has a lower cost than its previous;
- With a probability $e^{-\frac{\delta}{t}}$; with δ being the proposed cost change and t the control parameter, called temperature.

If none of these conditions are true the change is undone.

The temperature choice is an interesting topic to be better discussed. Lewis [2007] presents an extensive discussion about the temperature possibilities. We present here only the author's approach.

Before the algorithm execution, a small number of neighborhood operators is applied and the initial temperature t_0 is defined as being the cost standard deviation during these movements. Throughout the execution the temperature is gradually reduced in a geometric cooling, with the temperature being $t_{i+1} = \alpha \times t_i$. With α being the cooling rate, varying between 0 and 1.

The author defined the problem as a Markov Chain[1]. For each value of t a Markov Chain is generated and t is changed between subsequent chains. The ml size of the Markov Chain was defined with respect to the instance size of the problem.

The ml is calculated with the following formula, where $f(r, c)$ is the non-fixed cells number in the square (r, c).

$$ml = \left(\sum_{r=1}^{n} \sum_{c=1}^{n} f(r, c) \right)^2$$

Finally there is a restart rule: if after a fixed number of chains the solution has not been found, t is restarted to t_0. This restart is called reheating. This approach, described in [Lewis 2007], is our baseline and part of our heuristics.

The next section presents the three proposed heuristics.

[1] A mathematical system that present transitions between states as chains. This is a random process, but dependent of the Markov property: The next state only depends of its current state, not the previous states

4 Proposed Heuristics

We present three different heuristics on this work with all being based on concepts of constraint satisfaction algorithms, more specifically on $ARC-3$ (*Arc Consistency*) and $PC-2$ (*Path Consistency*) algorithms [Russell and Norvig 2009].

We called the three proposed heuristics as $Heuristic-1$, 2 or 3 and they are distinguished by the level of integration with the baseline [Lewis 2007], based on simulated annealing as previously discussed. $Heuristic-1$ does not have any relation with the baseline while $Heuristic-2$ presents an overlapping between the baseline and $Heuristic-1$. $Heuristic-3$ changes the baseline with the ideas of constraint satisfaction algorithms.

We proposed these heuristics based on the hypothesis that they start searching for a solution from a random initial state and this state can be far from solution. Since Sudoku is a NP-Complete problem the solutions' search space is enormous and distant points (start and solution) are almost impossible to be found only with search. Based on this, our heuristics try to reduce the solutions' search space reducing the search space for the search process (if necessary). We will discuss each approach carefully in these following sections.

4.1 Heuristic-1

This first heuristic is the implementation of two traditional constraint satisfaction algorithms: $ARC-3$ (arc-consistency) and $PC-2$ (path consistency). These algorithms were discussed in [Russell and Norvig 2009] where the authors also discussed its applicability to the Sudoku puzzle.

"A variable in a CSP[2] is arc-consistent if every value in its domain satisfies the variable's binary constraints[3]" [Russell and Norvig 2009] and the most popular arc-consistency algorithm is the $AC-3$. The application on the Sudoku is the analysis of the possible values in a cell given the predefined values in the other cells in the same domain. Figure 2, extracted from [Crook 2009], is an example of the set of initial possibilities in a board.

2345	245	1	9	23	237	567	4567	8
6	249	29	1247	8	5	7	3	249
234589	24589	7	234	6	23	1	459	2459
1258	3	4	267	9	2678	568	15678	156
1289	26789	2689	5	23	4	368	16789	1369
589	56789	689	367	1	3678	4	2	3569
2348	2468	5	1236	7	1236	9	1468	1346
239	1	2369	8	4	236	356	56	7
7	468	368	136	35	9	2	14568	13456

Figure 2: *Initial board configuration and its possibilities in each cell*

The path-consistency has a "stronger" notion of consistency than arc-consistency. "Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables" [Russell and Norvig 2009]. The $PC-2$ is a path-consistency algorithm very similar to $ARC-3$ and it was used on Sudoku as follows: Given a specific domain where two cells have α, β as possibilities and a third one has α, β, γ as possibilities, this third cell will be filled with γ even with its set of possibilities

[2] Constraint Satisfaction Problem

[3] "A binary constraint relates two variables (...) A binary CSP is one with only binary constraints (...) "[Russell and Norvig 2009]

having more than one element, since its other elements must be in other cells.

The following algorithm describes our approach. It is a high-level representation and the function names are intuitive and its algorithms will not be presented.

```
HEURISTIC-1(board)
 1  advancedSolution = FALSE
 2  filledCell = FALSE
 3  repeat
 4      UPDATEFILLINGPOSSIBILITIES(board)
        // Fill Cells:
 5      advancedSolution =
            FILLVALUESONLYPOSSIBLEONECELL(board)
 6      filledCell = FILLCELLSWITHONEPOSSIBILITY(board)
        // Evaluate the number of unsolved cells:
 7      unsolvedCells = GETNUMBEROFBLANKCELLS(board)
        // Stop condition:
 8      if unsolvedCells ≤ 0 or TIMEISOVER()
 9          foundSolutionOrWithoutTime == TRUE
10  until (advancedSolution or filledCell) and
                    not foundSolutionOrWithoutTime
```

The UPDATEFILLINGPOSSIBILITIES update all the possible values in each cell while FILLVALUESONLYPOSSIBLEONECELL is the $PC-2$ algorithm and FILLCELLSWITHONEPOSSIBILITY is the $ARC-3$ algorithm. Both of them return TRUE if they have filled any cell and FALSE otherwise. We defined an empty cell having its value equal to -1 and we also represented here the maximum time condition, discussed on Section 5.

We can say that $Heuristic-1$ is the attempt of solving Sudoku with the application of $ARC-3$ and $PC-2$ iteratively, *i.e.*, in each new algorithm application the board is updated, as its sets of possibilities. We then try to fill other cells with these updates. This heuristic has the property of, if after an iteration the board state has not changed it will be "stuck" and will not be able to continue and its execution is aborted. We consider this instance not solved.

Note that unlike the baseline and $Heuristic-2$ and 3 this heuristic does not present a random approach. Nevertheless we did the same tests for it, applying the same replications applied to other heuristics to facilitate comparison.

4.2 Heuristic-2

This heuristic is a hybrid approach between the baseline and $Heuristic-1$. It consists in the application of $Heuristic-1$ until solving the puzzle or it gets "stuck". If the algorithm is not able to change the board state and the board is not complete, this heuristic tries to solve it with the baseline, *i.e.* the simulated annealing approach, in place of aborting execution as $Heuristic-1$.

For sake of completeness this algorithm is presented below. To achieve an easier representation we assumed that HEURISTIC-1 returns TRUE if it solved the Sudoku completely and FALSE otherwise.

```
HEURISTIC-2(board)
1  completelySolvedSudoku = HEURISTIC-1(board)
2  if completelySolvedSudoku == FALSE
3      SIMULATEDANEALING(board)
```

The hunch of this heuristic is that less cells to be filled imply in a lower number of possible value exchanges between cells in the neighborhood operator. It is expected that this reduction of the search space allows this heuristic to be faster (for less search possibilities) and to be able to solve more puzzle instances. All these expectations were confirmed as we will discuss in Section 6.

4.3 Heuristic-3

This last heuristic, as $Heuristic-2$, is a hybrid approach between the baseline and $Heuristic-1$. It is very similar to $Heuristic-2$ and its main structure can be represented by the previous algorithm. The difference between them is in the board initial filling process at the beginning of the simulated annealing execution. While $Heuristic-2$ does not change the initial random filling, as previously discussed, $Heuristic-3$ tries to fill the empty cells with a number that is not in the cell set of possibilities.

This filling consists on the enumeration of the missing values on the grid $n \times n$ followed by the attempt of filling the board as follows: we iterate over the sequence of missing values in the grid looking for an element that is contained in the set of possibilities. If it is found we assign this value to the cell and it is removed from the list. If we find no number that satisfies this condition we select any number from the missing values set. This algorithm is following presented.

```
FILLSQUARE(line, column)
    // Create a vector with all possible numbers
 1  possibleValues = {1 .. (board.order)^2}
    // Search elements that already filled a cell
    // in the square and removes it from the list
 2  for i = 1 to (board.order)^2
 3      if BOARDCONTAINS(board, i)
 4          REMOVEELEMENT(possibleValues, i)
    // Shuffle list of possibilities
 5  SHUFFLE(possibleValues)
    // Fill the square
 6  for i = line × board.order to (line + 1) × board.order
 7      for j = column × board.order to (column + 1) × board.order
 8          if board[i][j].value == −1
 9              if possibleValues ∩ board[i][j].possibilities ≠ ∅
                    // Fill with the first possible value in the list
10                  board[i][j].value = GETVALUE(possibleValues ∩
                                        board[i][j].possibilities)
11                  REMOVEELEMENT(possibleValues,
                                  board[i][j].value)
12              else
                    // Fill with any other value
13                  board[i][j].value = GETVALUE(possibleValues)
14                  REMOVEELEMENT(possibleValues,
                                  board[i][j].value)
```

The initial intuition of this approach was the attempt of reducing the number of iterations/changes done by the simulated annealing heuristic, trying to maximize the chance of a value to be already assigned to its right position, in order to decrease the number of performed iterations. This implementation created a large overhead and degraded the heuristic's performance as we will discuss deeply in Section 6.

The number of iterations was not evaluated since the spent time was bigger, to evaluate it we should first optimize our implementation to see a performance gain.

On the next section we will discuss the methodology of our experiments that validated all of our claims on this paper.

5 Experimentation Methodology

A very important topic to the methodology discussion is how we generate our test instances. The test instances are the initial configuration of the board and were created by the generator presented in [Lewis 2007]. A first concern to this instances generation is that they must be solvable somehow, even if this solution is unknown. We can achieve this from complete boards, removing some of its cells.

We can observe that the variability of the generated instances depends on the way we remove the cells and the complete boards initially used. We can see several ways of obtaining complete boards from an initial solution in [Lewis 2007], they are based on operations that can be applied to the complete board in a way that they do not invalidate the solution. The two most obvious operations that do not invalidate a solution already found are the permutation of complete columns or rows of the board. We could also exchange square rows or columns (this shall be done for all rows/columns),

i.e. , we use the board as it had a size of $n \times n$ in place of $n^2 \times n^2$ and we permute its rows or columns.

All these operations guarantee us $n!^{2(n+1)} - 1$ different solutions from a specific one [Lewis 2007]. Other approaches that do not invalidate the specific solution as the values permutations (1 becomes 4 and vice versa, for example) were not used in the generator.

Once we defined a solution that will be used in the generator we start a cell removal phase where each cell has a probability p of being removed from the board. Note that $p = 0$ implies in a complete board while $p = 1$ implies in an empty board.

After discussing the instance generation process it is possible to present the experiments configuration. For boards with size $n = 3$ we replicated the methodology described by [Lewis 2007], *i.e.*: we generated 20 complete boards and after that we variated p from 0.0 to 1.0 with a step equals to 0.05 for each one, generating 20 test instances for each probability. Each of these instances were executed 20 times generating 400 measurements per instance/probability of removal.

To easiness the execution methodology the following algorithm presents the loop representing all the executions.

RUNEXPERIMENTS

```
1  for probability = 0.0 to probability ≤ 1.0
2      probability = probability + 0.05
3      for i = 0 to i ≤ 20
4          for j = 0 to i ≤ 20
5              EXECUTE(probability, seed[j])
```

Once this procedure was done to $n = 3$ we observed while evaluating its results that the coefficient of variation($C.O.V$)[4] of the execution time to these 20 repetitions had a small variation (maximum C.O.V.: 5%). Because of this, for $n = 4$ and 5 we reduced the number of replications from 20 to 3. This new configuration generated 60 measurements for each instance/probability of removal. As expected we could observe a raise in the C.O.V. (maximum C.O.V.: 12% for $n = 4$) but we judged this cost/benefit interesting for our methodology. In the above algorithm we just replace 20 by 3 in the line 3.

Each execution generated two different information that were used in the analysis: A binary value that answers if after a maximum time t the heuristic found the solution ($t = 5s$ for $n = 3$, $t = 30s$ for $n = 4$ and $t = 350s$ for $n = 5$), as defined by [Lewis 2007]), and the execution time (in seconds). Naturally, if a heuristic does not find a solution in the maximum execution time its execution time will be defined as the maximum possible for this instance. It is also used to calculate the average execution time.

The configuration of the machines used in the experimentation process by [Lewis 2007] was a PC with Linux and a 2.66GHz processor and a RAM of 1GB. We executed these experiments in a PC with Linux with a processor of two 2.13GHz cores and 4GB of RAM. We did not implement any type of parallelism and information like the processor's cache used by [Lewis 2007] could be useful but it is not available (our has 3MB).

We present on the next section the results obtained and its respective analysis.

6 Results

We executed the experiments we described on the previous section to Sudoku instances of size $n = 3$, $n = 4$ and $n = 5$, just like [Lewis 2007].

A first analysis that guarantees our heuristics usefulness is the number of problems solved by each one. It would be pointless to analyze the heuristics' execution time if they are not able to solve a number of instances similar to the baseline. This condition is satisfied by $Heuristic - 2$ and $Heuristic - 3$ that solves, at least, the same percentage simulated annealing solves.

[4] "The ratio of the standard deviation to the mean is called the Coefficient Of Variation (C.O.V.)" [Jain 1991]

Figure 3 presents the percentage of instances that are solved by each heuristic in a board with $n = 3$. In fact these results are consistent with those presented in [Lewis 2007] since the simulated annealing heuristic is able to solve all instances of size 3 in less than $5s$. We can observe that the same happens to the heuristics that mix the constraint satisfaction algorithms to the simulated annealing. This graph is interesting for us because we can observe the performance decay of the heuristic that uses only the constraint satisfaction approaches ($Heuristic - 1$).

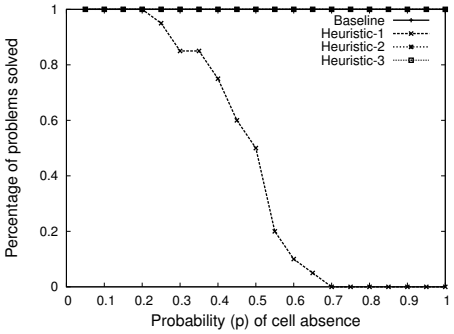

Figure 3: *Percentage of solved instances by heuristic for the Sudoku with size* $n = 3$

The same analysis was done to $n = 4$ (Figure 4) and, as for $n = 3$, the constraint satisfaction algorithms fail from $p = 0.25$ and stop being useful in $p = 0.6$. The final usefulness value for $n = 3$ is $p = 0.7$. These values are very useful in the analysis of the average execution time of each heuristic.

Another important result related to Figure 4 is the percentage of solved instances by the heuristics when $p = 0.6$. $Heuristic - 2$ has solved 95% of the instances while $Heuristic - 3$ and the baseline solved 90%. This temporary decay probably is due to the *phase transition region* also pointed by [Lewis 2007]. Nevertheless is interesting to note that this phase is "softer" and it exists to a higher p value when compared with [Lewis 2007], that was $0.35 \leq p \leq 0.5$ to $n = 4$. This difference must be better studied.

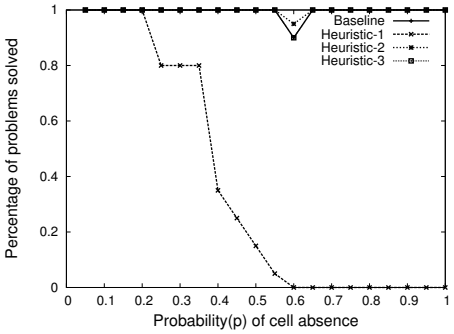

Figure 4: *Percentage of solved instances by heuristic for the Sudoku with size* $n = 4$

Once we analyzed the heuristics' behavior in terms of success we start to analyze its execution time. This measure validates our initial hypothesis that the constraint satisfaction algorithms "speed up" the heuristics' execution decreasing its execution time. Figures 5 and 6 present the average execution time of all heuristics and the baseline to $n = 3$ and $n = 4$, respectively.

To validate the hypothesis that hybrid approaches between the constraint satisfaction algorithms and simulated annealing are more efficient in terms of average execution time than just simulated annealing we performed Paired T-Tests between $Heuristic - 2$, $Heuristic - 3$ and the baseline.

The T-Test show us that, for $n = 3$, in the interval $0.0 \leq p \leq 0.15$ the two heuristics have a statistically equivalent execution time to the baseline while, in the interval $0.20 \leq p \leq 0.75$ the proposed heuristics execute with a statistically lower cost. The interval

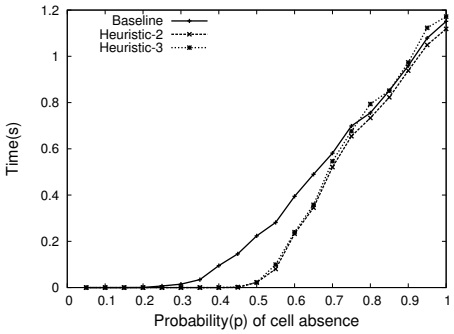

Figure 5: *Average time spent by each heuristic to solve the Sudoku instances of size* $n = 3$

$0.80 \leq p \leq 1.00$ presents a distinct behavior between the two proposed heuristics. While $Heuristic - 2$ has always a lower cost $Heuristic - 3$ has a higher cost in almost the complete interval, except in $p = 0.85$ where it has an equivalent cost. All these evaluations were made with a confidence of 95%.

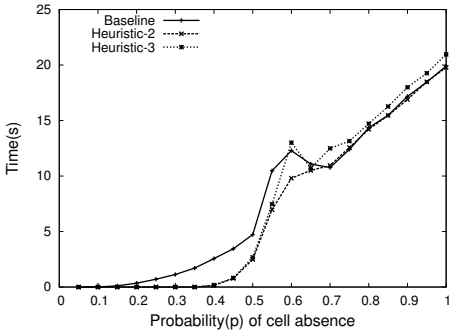

Figure 6: *Average time spent by each heuristic to solve the Sudoku instances of size* $n = 4$

The T-Test applied to the data of $n = 4$ presented the following results: for the intervals $0.0 \leq p \leq 0.05$ and $0.60 \leq p \leq 1.00$ $Heuristic - 2$ runs with a statistically equivalent cost (except in $p = 0.75$ where it is slower). $Heuristic - 2$ and $Heuristic - 3$ are faster between $0.10 \leq p \leq 0.55$. Finally, $Heuristic - 3$ executes with a statistically equivalent cost in the intervals $0.0 \leq 0.05$ e $0.60 \leq p \leq 0.65$. It is slower between $0.70 \leq p \leq 1.00$ (except in $p = 0.80$) where the cost is equivalent. As in the previous paragraph, all these analysis where done for a confidence of 95%.

These results, in addition to the number of solved problems by each heuristic show us clearly that the proposed approach reduces the heuristic execution time when compared with the simulated annealing approach. This happens because for small p values(≤ 0.20) the constraint satisfaction algorithms are sufficient to solve these instances, what makes the proposed heuristics faster in general since the simulated annealing nor even has to be started. We can clearly see that despite the existent differences for larger p values(≥ 0.60) this difference is very small, the curves are very close to each other and this is due to the fact that the constraint satisfaction algorithms are not able to consistently help the simulated annealing solver (they solve zero problems completely, they may just add some elements) and this is why the execution times are very close.

The most interesting interval to be analyzed is the one between the two already described because the heuristics are faster in general an the reason is obvious: the problems already are "difficult" enough to the simulated annealing and the constraint satisfaction algorithms are still useful. In Figures 5 and 6 is in this interval that we can observe the maximum distance between the average execution time of the heuristics and the baseline. With the increasing of p this distance gradually decrease since the constraint satisfaction algorithms start to become useless.

We also executed all approaches in larger boards (order equals to 5) trying to apply the same methodology applied in [Lewis 2007]. We decided to separate this result from the others because it demands a deeper analysis. As the other board orders we also first analyzed the number of instances solved by each heuristic. These results are presented in Figure 7.

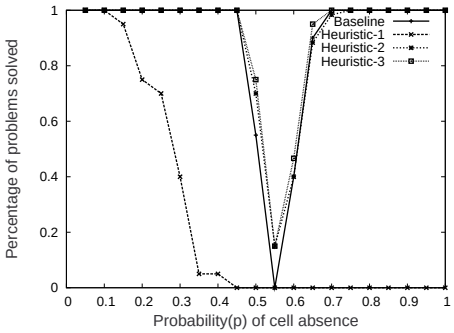

Figure 7: *Percentage of solved instances by heuristic for the Sudoku with size* $n = 5$

It is interesting to observe that we have an interval where all the heuristics fail to solve many of the instances, this is between $0.50 \leq p \leq 0.65$ and as previously discussed we believe this is the phase transition region. As previously observed in boards of lower order we have reached the phase transition region with higher probabilities than [Lewis 2007]. As the previous results we can clearly observe that our heuristics ($Heuristic-2$ and 3) were able to solve more instances than the baseline.

Once we have shown the proposed heuristics ability to solve more Sudoku instances we can now present their execution time compared to the baseline. This analysis is summarized in Figure 8. As we already discussed, with lower probabilities we are able to outperform the baseline because the constraint satisfaction problems are able to solve this type of instance.

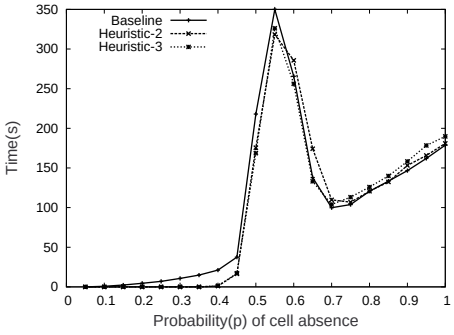

Figure 8: *Average time spent by each heuristic to solve the Sudoku instances of size* $n = 5$

With a confidence of 95% we can say that $Heuristic - 2$ is faster than baseline for probabilities between $0.00 \leq p \leq 0.55$ while it is equal for p equals to $0.60, 0.80, 0.85, 0.95$ and 1.00. $Heuristic - 2$ is slower than baseline for $0.65 \leq p \leq 0.75$ and for p equal to 0.9. For sake of completeness, $Heuristic - 3$ execution time comparison is: it is faster than baseline in $0.00 \leq p \leq 0.55$, equals to it between 0.60 and 0.70 and is slower betweeen $0.75 \leq 1.00$.

After presenting all the results we can deeply discuss them: we can observe that the proposed integration helps the heuristics in general, mainly in more filled boards since the constraint satisfaction heuristic works better in these situations. We can see that for $p > 0.55$ our approach does have similar results to the baseline.

The situations they are slower than the baseline are generated because previously filling the board shifted the phase transition region to the future. This phenomenon was better observed for the large board due to the bigger impact this region caused. Despite this shift we can observe that, in general, our approach improved the baseline since it is faster in most part of the interval.

A final analysis we did was related to the relation between $Heuristic - 2$ and $Heuristic - 3$. We observed that, in general, with a confidence of 95%, $Heuristic - 2$ is faster than $Heuristic - 3$ in most part of the p values. For $n = 3$: $0.45 \leq p \leq 1.00$ and for $n = 4$: $0.35 \leq p \leq 1.00$ (except for $p = 0.45|0.55|0.65$ where they are equivalent).

For $n = 5$ we have seen a much closer performance in many parts of the whole line of probabilities of the execution. $Heuristic - 2$ was faster than $Heuristic - 3$ in the following list of probabilities: $\{0.05, 0.8 \leq p \leq 0.95\}$ and $Heuristic - 3$ is faster in $\{0.10, 0.15, 1.00\}$. The other p values are for statistically equivalent time (95% of confidence). It seems the overhead decreased for larger boards and it appeared again for larger probabilities, what is obvious since the algorithm will process several evaluations that, for a higher board would take much time and would not achieve an answer. All these experiments have shown us that $Heuristic - 3$ is worse than $Heuristic - 2$.

This result show us that the modification of the algorithm for the initial board filling for the simulated annealing does not reduce the execution time, in fact, it generates an overhead that does not worth.

7 Conclusion

In this work we have proposed an approach that "speeds up" the simulated annealing heuristic performance allying it to the constraint satisfaction algorithms reducing the problem search space. We have seen that the use of these algorithms, if used separately, is not interesting since they are not able to solve a great number of problems. We also have seen that, when allied to other approaches the constraint satisfaction algorithms are very useful.

We were able to clearly show through our experiments that the "speed up" of our heuristics is related to the time spent during the solving with the constraint satisfaction algorithms, we were also able to solve a higher number of Sudoku instances than the baseline with one of our approaches.

Much more can be done in this field, an immediate work that can be made is the application of these heuristics in real datasets as the problem set published in the newspaper *The Sun*.

In addition to these future works is also interesting to compare the heuristics proposed here to other methods that are not based on simulated annealing to position it in relation to the other approaches that have been used.

Finally, we could also try to improve our heuristics. Although we have concluded that the overhead caused by the "intelligent filling" does not worth in the $Heuristic - 3$ we still believe that is valid to evaluate the integration of the constraint satisfaction algorithms with the simulated annealing, as the alteration of the neighborhood operator to start using possible values of each cell. Even a less naive implementation of the proposed integration is valid (algorithm FILLSQUARE).

Acknowledgements

We would like to thank Rhyd Lewis that gently made his source code available.

This work is partially supported by CAPES, CNPq and Fapemig.

References

AGERBECK, C., HANSEN, M. O., AND LYNGBY, K. 2008. *A Multi-Agent Approach to Solving NP-Complete Problems*. Master's thesis, Technical University of Denmark.

CROOK, J. F. 2009. A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles. *Notices of the AMS*, 460–468.

IST, I. L., LYNCE, I., AND OUAKNINE, J. 2006. Sudoku as a SAT Problem. In *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics (AIMATH)*.

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley.

LEWIS, R. 2007. Metaheuristics Can Solve Sudoku Puzzles. *Journal of Heuristics 13* (August), 387–401.

MOON, T., GUNTHER, J., AND KUPIN, J. 2009. Sinkhorn Solves Sudoku. *Information Theory, IEEE Transactions on 55*, 4 (april), 1741 –1746.

MORAGLIO, A., AND TOGELIUS, J. 2007. Geometric Particle Swarm Optimization for the Sudoku Puzzle. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO)*, ACM, New York, NY, USA, 118–125.

MORAGLIO, A., TOGELIUS, J., AND LUCAS, S. 2006. Product Geometric Crossover for the Sudoku Puzzle. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 470 –476.

RUSSELL, S., AND NORVIG, P. 2009. *Artificial Intelligence: A Modern Approach (3rd Edition)*, 3 ed. Prentice Hall, Dec.

SATO, Y., AND INOUE, H. 2010. Genetic operations to solve sudoku puzzles. In *Proceedings of the 12th annual conference companion on Genetic and Evolutionary Computation (GECCO)*, ACM, New York, NY, USA, 2111–2112.

SIMONIS, H. 2005. Sudoku as a Constraint Problem. In *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, 13–27.

WEBER, T. 2005. A SAT-based Sudoku Solver. In *Proceedings of 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings*, G. Sutcliffe and A. Voronkov, Eds., 11–15.