A Hybrid Heuristic Approach for the Sudoku problem

Nysret Musliu and Felix Winter*
Vienna University of Technology
*corresponding author

***Abstract*—Sudoku is not only a popular puzzle but also an interesting constraint satisfaction problem. Therefore automatic solving methods for this problem have been the subject of several publications in the last decade. While some authors propose algorithms which make use of constraint programming techniques, others propose meta-heuristics and stochastic search to tackle larger instances of the puzzle. Although current methods provide good solutions for many Sudoku problems, some instances remain challenging. This paper introduces a new local search technique based on the min-conflicts heuristic. Furthermore we will introduce a hybrid approach that will additionally apply constraint programming methods in the search process. We experimentally evaluate our methods where we report improvements over state of the art solutions.**

***Index Terms*—Heuristic methods, Constraint satisfaction, Sudoku**

I. Introduction

Sudoku is a logic puzzle where one has to fill a grid with numbers that typically lie between one and nine. The challenge of solving these problems became popular among people all over the world during the last decades, and it can be found in a wide variety of newspapers nowadays. From a scientific point of view Sudoku belongs to the class of constraint satisfaction problems. This and the fact that it belongs to the class of NP-complete problems make the task of finding an efficient solver not only interesting, but also very challenging when dealing with large puzzles.

Formally a Sudoku puzzle instance can be described as an $n^2 \times n^2$ grid which is divided into n^2 distinct squares. These squares divide the whole grid into $n \times n$ sub-grids. To solve a Sudoku, each cell must be filled with a number in the range of 1 to n^2. Additionally, three constraints must be fulfilled to achieve a valid solution:

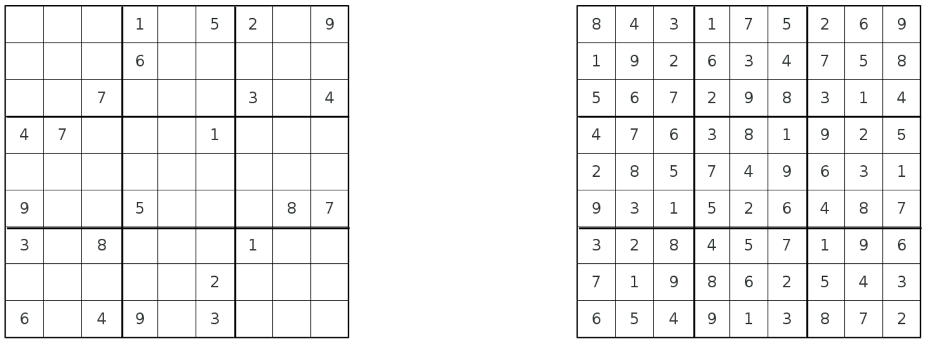
1) In every row the numbers 1 to n^2 appear exactly once.
2) In every column the numbers 1 to n^2 appear exactly once.
3) In every $n \times n$ sub-grid the numbers 1 to n^2 appear exactly once.

The variable n determines the size and also to some degree the difficulty of a Sudoku. This is sometimes referred to as the puzzle's *order* and we will also call it this way from now on. Problems that are meant to be solved by the human mind usually have an order of three, and most of the instances which are published in newspapers are this size. An example of such a puzzle can be seen in Figure 1.

In the literature different techniques have been proposed to solve Sudoku puzzles. Especially two large groups of methods and techniques have been repeatedly applied: Constraint programming (CP) and stochastic search based methods. These two classes differentiate in several properties, but the most crucial difference lies within the fact that CP methods are exact search methods, while stochastic search approaches are non-deterministic and cannot guarantee to find an optimal solution. Sudoku solving techniques based on constraint programming have been well studied and proposed for instances that consist of grids with 9×9 cells. For example in [1], the author introduces a formal model of Sudoku as a constraint satisfaction problem. Additionally the paper provides an implementation which is based on backtracking search and uses the eclipse framework [2]. A similar solving method is also investigated in [3], where the authors conduct a comparison of different variable- and value-selection heuristics using intelligent backtracking techniques.

In all of this publications it is shown that approaches relying on CP work well on 9×9 puzzles and are able to classify the difficulty of puzzles, seen from a human's perspective. This feature becomes interesting when creating problems that are meant to be solved by humans and is further discussed in [1]. Additional improvements to solve puzzles of this size have been published in [4] and [5]. These papers focus on solving the hardest 9×9 problem instances by applying hybrid search techniques.

Larger Sudoku grids that consist of 16×16 or even 25×25 cells (that would correspond to an order of 4 or 5 respectively), introduce new challenges. Exact methods which try to find a solution by applying intelligent enumeration mechanisms come to their limits here, since the search space is simply too large to enumerate all solutions in feasible time. For solving also larger instances, the author from [6] introduces the first meta-heuristic driven approach for Sudoku puzzles with an implementation that uses simulated annealing. The paper concludes that puzzles of higher order can be tackled by heuristic techniques and are able to outperform exact methods when solving such problems. The ideas and problem formulation presented in this paper have since then been extended in [7] and [8]. These articles introduce constraint propagation techniques to reduce the search space before applying the meta-heuristic driven search process. To our best knowledge, they set the state of the art for solving large Sudoku puzzles. However, although these techniques perform significantly better than exact techniques, there still is place for improvement. Sudoku instances that consist of a grid with 25×25 cells and have about 55% of their cells filled initially, form the hardest class of problems. This shows in a significant



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

(a) Problem

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

(b) Solution

Fig. 1. This figure shows a Sudoku problem of order 3 with its initial prefilled cells on the left and its corresponding solution on the right.

drop of the success rate when using the simulated annealing based solver in all of the published results. Therefore in this article we focus on large problem instances.

In this paper we propose a new hybrid method of solving the Sudoku problem using a combination of iterated local search, the min-conflicts heuristic and tabu search. Additionally our method includes constraint programming techniques that are applied during the perturbation phases of an iterated local search based procedure which calls the underlying local search methods. Although local search techniques in hybridization with constraint programming have been previously proposed in the literature, to the best of our knowledge the ideas used in this paper regarding min-conflicts, iterated local search are innovative and have not been considered before. Furthermore we use constraint programming in a unique way as a perturbation mechanism of iterated local search.

By using a random instance generator proposed in the literature we randomly generate a total of 1200 puzzle benchmark instances. We experimentally evaluate our methods as we compare our results with state of the art methods, where we report improvements regarding the success rate.

In the following we will shortly present the problem formulation in Section II and then give a detailed description of our approach in Section III. Information about our conducted experiments and the configuration of our algorithm will then be given in Sections IV and V. Finally we will present our results and make a comparison with existing approaches in Section VI. Conclusion remarks will be given in the last section.

II. Problem formulation

In order to describe the problem formulation we will use the terminology introduced by Lewis which defines the notion of a *square*. A *square* refers to each of the $n \times n$ squares that form the Sudoku puzzle. Furthermore a $square_{r,c}$ denotes the *square* in row *r* and column *c*, considering an instance as a grid of *squares*. In a similar fashion a value of a cell in row i and column j is referred to as $cell_{i,j}$. A cell which has its value predefined in the puzzles is called *fixed*, whereas a cell that is initially empty and has to be filled by the solver is referred to as *unfixed*. Finally, a grid that is complete and fulfills all of the problem's constraints is referred to as *optimal*.

III. A new approach for solving the Sudoku problem

In this section we will propose a meta-heuristic driven approach for the Sudoku problem based on the Min-conflicts heuristic, which introduces a, to our best knowledge, new method of solving this puzzle.

A. Representation and Neighborhood operator

In our local search techniques we use a direct representation of the Sudoku grid. To generate an initial solution all of the puzzle's unfixed cells are filled randomly in such a way that the third constraint of the puzzle will not be violated. In other words every *square* contains the values from 1 to n^2 exactly once. The neighborhood operator will choose two different unfixed cells in the same square and swap both of them in each search step. Details on the selection of the two cells are given in section III-C. The way the initial solution and the neighborhood operator are defined has the positive side effect that the third constraint of the puzzle is always fulfilled. This leads to a reduced overhead when calculating the objective value of a solution.

B. Evaluation of candidate solutions

Since two cells lying inside the same square can never contain the same number throughout search it makes sense to only consider potential conflicts per row and column in the evaluation function. The cost function that we used is taken from [6] and looks at each row and column individually. In each row/column all missing numbers from 1 to n^2 are counted and summed up. An optimal solution will therefore have a cost of 0. The objective function f for a candidate solution S therefore is:

$$f(S) = \sum_{i=1}^{n^2} r(i) + \sum_{i=1}^{n^2} c(i) \tag{1}$$

where $r(i)$ and $c(i)$ represent the number of missing values in row i or column i respectively.

In order to keep the required time consumed during evaluation as low as possible, we applied delta-evaluation. Each single search step in fact influences the number of conflicts for at most two rows and the two columns of the swapped cells. Therefore only the affected row/column costs are updated.

C. Min-conflicts for Sudoku

In order to achieve an effective technique for the Sudoku problem we used the min-conflicts heuristic [9] during local search. The general idea behind this heuristic is to concentrate on variables which cause conflicts and to find good neighbors by performing cell swaps which reduce a lot of existing conflicts in the grid. The procedure works as follows: First a conflicting cell is selected randomly and then a good swap partner is determined.

Figure 2 illustrates the use of the min-conflicts heuristic: On the left we see the grid as it appears before the cell swap. The circled value 1 in the upper left sub-grid represents the randomly selected cell which is in conflict (the two cells with the blue background highlight these conflicts). The numbers outside the Sudoku grid (also with blue background) give information about the number of missing values in the considered rows and columns. In the search for a good swap partner the algorithm selects the value that would lead to the lowest possible number of conflicts when swapped with. In this case value 5 which is also circled is selected (Note that a swap with any other cell would not move the value 1 to a good position). On the right side we can see the grid after the swap has been performed. The number of conflicts for the affected rows and columns has been decreased successfully.

One drawback of using the min-conflicts heuristic in local search lies in the fact that it easily can get stuck in local optima. We additionally use a tabu list storing recently performed swaps. Moves are marked as tabu for a number of forthcoming iterations so that the search is able to escape from local optima if necessary. One exception to this rule are swaps that would generate a solution with a fitness that beats the best found solution so far. If this so called aspiration criterion [10] is fulfilled, a swap will be allowed even if it is marked as tabu. As soon as the best swap candidates have been determined the question remains if this move should be accepted or not. Common possible variants would be to either accept the candidate in any case or to only accept it if evaluation yields an improvement of the fitness. We decided to use a combination of both approaches: Candidates which lead to a higher or equal cost are accepted only under a certain acceptance probability which is given as a parameter to the program. Candidates which lead to a lower solution cost however will always be accepted. The whole process of generating and selecting neighborhood moves is repeated until either the optimal solution is found, or no improvement can be achieved for a given number of iterations. This iteration limit is also defined through a program parameter. The overall local search procedure which used in my approach is shown in Algorithm 1.

Algorithm 1 Min conflicts heuristic with tabu list for Sudoku

```
Input: puzzle, iterationLimit, acceptanceProbability
 1: initialize tabu list
 2:
 3: $iterationCounter \leftarrow 0$
 4: $bestCost \leftarrow \mathrm{MAX}$
 5: $currentCost \leftarrow \mathrm{MAX}$
 6:
 7: while $bestCost > 0 \wedge iterationCounter < iterationLimit$ do
 8:     randomly select cell which is in conflict
 9:
10:     generate all possible swaps with the selected cell
11:
12:     $bestSwap \leftarrow$ Find the best swap which minimizes total conflicts
13:     $bestSwapNotTabu \leftarrow$ Find the best swap which minimizes total
        conflicts and is not tabu
14:
15:     if $bestSwap \neq bestSwapNotTabu$ then
16:         if EVALUATE($bestSwap$) $< bestCost$ then
17:             $currentCost \leftarrow$ EVALUATE($bestSwap$)
18:             perform swap
19:             go to 27
20:         end if
21:     end if
22:     if EVALUATE(bestSwapNotTabu) $< currentCost \vee random() <=$
        $acceptanceProbability$ then
23:         $currentCost \leftarrow$ EVALUATE($bestSwapNotTabu$)
24:         perform swap
25:     end if
26:
27:     update tabu list
28:     if $currentCost < bestCost$ then
29:         $bestCost \leftarrow currentCost$
30:         $iterationCount \leftarrow 0$
31:     else
32:         $iterationCount \leftarrow iterationCount + 1$
33:     end if
34: end while
Output: best solution
```

D. Constraint programming techniques

Although local search alone often can produce good results when dealing with constraint optimization problems like Sudoku, it has been shown in the literature ([7], [8]) that the introduction of constraint programming methods can bring significant improvements to the algorithm. Therefore we decided to also make use of CP in our algorithm.

One simple variant which was first described in [7] uses constraint propagation to reduce the search space before performing local search. Therefore the initial domains for each cell variable are modified until all variables are arc consistent at first. Any unfixed cell that has only one possible domain value left can then be considered as fixed when performing local search. The problem's search space can often be significantly reduced by using this technique.

In this paper we propose to further include a CP approach based on forward checking (FC) with dynamic variable ordering which is applied in between iterated phases of the local search and has the goal to intensify the search of promising areas in the search space. Whenever this procedure is called during search, all unfixed cells which cause conflicts plus some additional unfixed cells are emptied and the solver tries to find a solution by filling the missing cells with constraint programming methods. CP based approaches for the Sudoku problem have been examined in the literature ([1] and [3]) and we implemented a variant based on the results presented in these two publications. It basically performs a backtracking search using forward checking, makes use of a *minimum domain first* variable selection heuristic and a *smallest value first* value selection heuristic. Details about this method and its

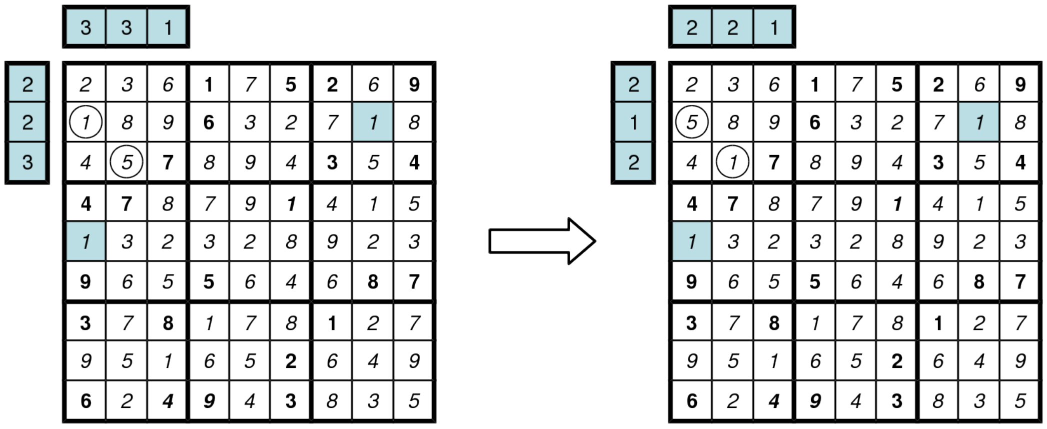


Fig. 2. In this figure an example neighborhood move applying the min-conflicts heuristic is illustrated.

parameters can be found in [3]. To the best of our knowledge this method has not been applied as a perturbation mechanism for iterated local search (ILS) [11].

E. Iterated local search for Sudoku

Further we propose a new iterated local search algorithm to solve the Sudoku problem. The main idea of this method is to examine the search space by iteratively calling an embedded heuristic and make use of perturbation after each iteration. Our solver repeatedly calls the min-conflicts based local search procedure with an instance of the problem where all the unfixed cells have been filled randomly in such a way that every value appears once per square. If local search fails to find the optimal solution after a given number of iterations, the program then enters its perturbation phase, where a further examination of the nearby search space using CP takes place. The perturbation process is conducted by emptying a number of unfixed cells and then performing forward checking search. It can lead to three different outcomes: Firstly, the optimal solution could have been found. Secondly, FC could detect that there is no possible solution for this particular candidate instance and thirdly the FC procedure could run out of time. In the first case, the algorithm has found the optimal solution and can exit. If one of the other two cases occurs, the procedure returns a partially filled Sudoku grid which also contains a number of cells that have been filled in the perturbation phase. The algorithm then will fill all the remaining cells randomly again and begin with a new iteration and continue with local search. Iterated local search keeps repeating this overall process until a given time limit is reached.

In the perturbation phase the number of cells that should be emptied additionally to the ones which are causing conflicts, play an important role. Therefore we introduced the *reset factor* parameter in our algorithm. Depending on the factor (a real value between 0.0 and 1.0) a relative amount of the puzzle's unfixed cells will be emptied. For example if the value is 0.8, 80% of the cells will be emptied before the FC procedure is started. To change this factor during the overall search we also decided to iteratively reduce the reset factor after every perturbation phase. This is done by multiplication with a parameter α which also lies between 0.0 and 1.0. The idea behind this stepwise reduction of cell resets is that the search increases the level of intensification with every processed perturbation phase.

In algorithm 2 the overall search process based on iterated local search utilizing all mentioned methods is given.

Algorithm 2 Iterated local search for Sudoku

```
Input: puzzle, timeLimit, resetFactor, $\alpha$
 1: FIXCELLSUSINGARCCONSISTENCY($puzzle$)
 2:
 3: FILLREMAININGCELLSRANDOMLY($puzzle$)
 4:
 5: $bestPuzzle \leftarrow puzzle$
 6: $bestCost \leftarrow$ EVALUATE($puzzle$)
 7:
 8: while $bestCost > 0 \vee$ time limit reached do
 9:     $puzzle \leftarrow$ MINCONFLICTSWITHTABULIST($puzzle$)
10:
11:     $cost \leftarrow$ EVALUATE(puzzle)
12:
13:     if $bestCost > cost$ then
14:         $bestCost \leftarrow cost$
15:         $bestPuzzle \leftarrow puzzle$
16:     end if
17:
18:     if $cost > 0$ then
19:         Empty all unfixed cells in $puzzle$ which are in conflict
20:
21:         Additionally empty relative amount of
22:         remaining unfixed cells defined by $resetFactor$
23:
24:         FORWARDCHECKINGSEARCH($puzzle$)
25:
26:         FILLREMAININGCELLSRANDOMLY($puzzle$)
27:
28:         $resetFactor \leftarrow resetFactor \cdot \alpha$
29:     end if
30: end while
Output: bestPuzzle
```

F. Algorithm parameters

A number of parameters which are given to the program, are used to configure our proposed algorithm. To give a short overview about their use and function, all of them are shortly described in this section.

The *iterationLimit* parameter controls how many steps without overall improvement are allowed inside the embedded min-conflicts based local search procedure. For example if it is set to 1000, the cost of the best solution found so far is 15, and no solution with a lower cost is found during the following 1000 iterations, local search will stop. ILS will then continue with the perturbation phase.

Determining the length of the used tabu list depends on the *tabuListSize* parameter. It is given to the algorithm as a floating point number between 0.0 and 1.0. The length is then calculated by rounding the product of the number of unfixed cells of the puzzle with the *tabuListSize* parameter. For example if it is set to 0.05 the tabu list length will be 5 if the puzzle counts 100 unfixed cells.

After a neighborhood candidate has been selected with min-conflicts, it has to be determined whether or not the corresponding move should be accepted. If it does not bring any improvement in cost it will only be conducted under a certain probability that is defined by the *acceptanceProbability* parameter. If the parameter for example is set to 15%, then in around 15 out of 100 cases neighbor which does not introduce any improvement will be accepted.

The *resetFactor* parameter is a floating point number within the range $[0.0, 1.0]$ and sets a relative value which determines how many unfixed cells should be reset (additionally to all conflicting cells which are reset in any case) at the start of the perturbation phase of iterated local search. For example if the *resetFactor* is set to 0.2 then around 20% of the remaining unfixed cells are emptied.

The parameter α is responsible for the iteratively reduction of the *resetFactor* and is applied at the end of each perturbation phase. It also is given to the algorithm as floating point number between 0.0 and 1.0.

Finally there are two time limits which control the algorithms runtime. An overall *timeLimit* restricts the overall searching time. If it passes, the algorithm will output the best found solution so far and quit. The *forwardCheckingTimeLimit* on the other hand will limit the time that is given to the FC procedure in each perturbation phase. Whenever time spent within FC search passes the limit, the algorithm is forced to end the exact search.

IV. Experimental Environment

In this section we compare our min-conflicts based hybrid solver to the state of the art algorithms for Sudoku. We contacted the authors of [7], [8] and [3] for the source code of their implementations, so that we would be able to compile the solvers and conduct a fair comparison of the results. All of them responded to our request and we have been provided with the sources for the simulated annealing based programs from [7] and [8]. The CP parts from [7] had to be reimplemented. Regarding [3], we implemented the algorithm based on the instructions from the authors. Regarding the instances we first experimented with state of the art 9×9 Sudoku puzzles from [12]. Our algorithm could solve each of those instances within one second. As those instances have not been shown to be challenging for our solver we considered the generation of harder instances by the random instance generator from [6]. The program creates a puzzle by simply removing some randomly selected cells of a presolved puzzle. We furthermore followed Lewis' experimental approach (described in [6]) and created puzzles in 20 different categories for each order, categorized by the proportion of fixed cells (p) in the Sudoku grid. Those categories used values for p starting from 0.0 up to 1.0 using steps of 0.05. Therefore puzzle instances with 0% fixed cells, 5% fixed cells, 10% fixed cells and so on were generated. To provide a large number of problems, 20 instances were created per category, totaling in a number of 400 instances per order. Since most of the discussed algorithms rely on stochastic search in some way, we performed 20 repeated test runs on each puzzle instance. Note however that for the puzzle instances with order 5, experiments have been conducted exclusively for the simulated annealing based algorithm from Lewis and our min-conflicts based algorithm, since the other two did not perform well even on instances with an order of 4.

All the tests were run on an Intel Xeon E5345 2.33GHz with 48GB RAM. The used instances in this paper as well as the sources of our algorithm implementation can be found at http://www.dbai.tuwien.ac.at/research/project/arte/sudoku/.

We used two metrics for the comparison of the algorithms: The average solving time and the success rate for each puzzle category. We followed the approach of Lewis in [6] to determine the required values: The success rate represents the percentage of successfully solved instances. The average time taken refers to the average runtime that was necessary to correctly solve a puzzle over 20 runs. Note that for the calculation of the average runtime only test runs which were able to find an optimal solution have been considered. The time limits differ depending on the order of the given puzzle instance. We used a time limit of five seconds for Sudoku of order three, 30 seconds for order four and a 350 seconds for order five Sudoku.

V. Algorithm configuration

Parameters for the considered algorithms from the literature ([7], [3] and [8]) were configured as described in the corresponding papers. In order to configure our algorithm, we ran experiments with different values on some of the hardest puzzles in our benchmark instances. As already mentioned by Lewis in [6] there is an 'easy-hard-easy' phase transition [13] depending on the relative number of prefilled cells. We focused on the harder problems (which have between 40 % and 45 % of the cells fixed initially) when experimenting with different parameter settings.

Since the *iterationLimit* parameter limits the min-conflicts heuristic number of trials to swap cells during Local Search, we decided to set its value relative to the number of cells in the grid. We experimented multiplicating the instance with factors of 10, 20 and 50, with 20 turning out to be the most suitable. Following this calculation for example for Sudokus with a 25x25 grid, the *iterationLimit* has been set to $625 \cdot 20 = 12500$ in our experiments.

We set the *forwardCheckingTimeLimit* parameter to a maximum of five seconds, so that the algorithm will not spend too much time in the perturbation phase.

The initial value for the *resetFactor* is 1.0. This way all of the unfixed cells will be removed in the first perturbation phase and the algorithm gets a chance to solve the puzzle solely by forward checking. In later iterations this value will then be stepwise reduced by the factor of α, which was tuned automatically. This will restrict forward checking search to smaller areas of the search space in later perturbation phases.

In order to determine good values for the *tabuListSize*, the *acceptanceProbability* and the *alpha* parameter we applied automatic parameter configuration using the irace-package [14]. We kept all of the irace default settings and only limited the tuning budget (maximum number of runs) for the algorithm to 1000. As tuning instances for irace we provided 20 puzzles with an order of 5 and that have 40% of their cells fixed. The elite candidates produced by the automated tuner suggested a *tabuListSize* of around 0.03, an *acceptanceProbability* of around 75% and a value for *alpha* of around 0.5. A run with the parameters determined by irace yielded good results, however we were able to achieve additional improvements by some manual tuning trials with these parameters on the tuning instances. By further manual tuning we found out that a *tabuListSize* of 0.05, an *acceptanceProbability* of 15% and an *alpha* of about 0.8 produced the better results. Therefore those values were used in the final experiments.

VI. Results

Experiments for puzzles of order 3 and 4 were conducted on four different algorithms: The simulated annealing based approach from [7] and its variation from [8], the constraint programming based solver from [3] and finally our algorithm proposed in this paper. For Sudoku instances of order 5, experiments were also conducted in the same form, but only for our solver and the algorithm from [7], which produced better results compared to the two other algorithms from the literature for Sudoku of order 4.

Figures 3, 4, 5 show a graphical representation of the results, for Sudoku of order 3, 4 and 5 respectively. Each figure shows average running times of successful runs per category in the form of bars. The success rate is shown as a punctuated line in the chart.

As we can see Crawford et. al [3] give very good results when it comes to solving 9×9 Sudoku puzzles, however as soon as the search space gets larger the constraint programming based algorithm cannot compete with the other approaches. This shows in the large drop of the success rate for Sudoku puzzles with an order of 4. For larger problems metaheuristic approaches using simulated annealing combined with constraint propagation techniques deliver better results regarding success rate.

This can be seen in Figures 4b and 4c that visualize results from the algorithms by Lewis [6] and Machado et al. [8]. When comparing those two approaches, Lewis implementation slightly outperforms the algorithm from Machado et al. regarding the success rate. Therefore for order 5 Sudoku we only compare to this algorithm.

The hybrid algorithm based on min-conflicts which is presented in this paper turned out to provide very good results in all of the tested categories. Based on these results we can conclude that our algorithm is very efficient and provides the best success rates on harder instances of order 4 and 5. This can be seen in 4 and 5. Detailed values regarding the success rates and running times for the hardest problems of order 5 are presented in tables I and II. As we can see, for the hardest instances ($p = 0.4$ and $p = 0.45$) the success rates of [7] are 38% and 12%, whereas our algorithm has success rates of 57% and 13%.

VII. Conclusion

In this paper we proposed a new approach to solve the Sudoku problem. Our method includes a novel min-conflict based heuristic and a new iterated local search algorithm that exploits a CP approach in the perturbation phase. We compared our approach to state of the art methods from the literature. Experimental results show the robustness of our algorithm on solving puzzles of different levels of difficulty. To the best of our knowledge our solver currently delivers the best results for Sudoku problem instances with an order of four and five.

Acknowledgement. The work was supported by the Austrian Science Fund (FWF): P24814-N23.

References

[1] H. Simonis, "Sudoku as a constraint problem," in *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, 2005, pp. 13–27.

[2] K. R. Apt and M. Wallace, *Constraint Logic Programming Using Eclipse*. New York, NY, USA: Cambridge University Press, 2007.

[3] B. Crawford, M. Aranda, C. Castro, and E. Monfroy, "Using constraint programming to solve sudoku puzzles," *Convergence Information Technology, International Conference on*, vol. 2, pp. 926–931, 2008.

[4] R. Soto, B. Crawford, C. Galleguillos, E. Monfroy, and F. Paredes, "A hybrid ac3-tabu search algorithm for solving sudoku puzzles," *Expert Systems with Applications*, vol. 40, no. 15, p. 58175821, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417413003059

[5] R. Soto, B. Crawford, C. Galleguillos, F. Paredes, and E. Norero, "A hybrid alldifferent-tabu search algorithm for solving sudoku puzzles," *Computational Intelligence and Neuroscience*, vol. 2015, 2015. [Online]. Available: http://dx.doi.org/10.1155/2015/286354

[6] R. Lewis, "Metaheuristics can solve sudoku puzzles," *Journal of Heuristics*, vol. 13, no. 4, pp. 387–401, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10732-007-9012-8

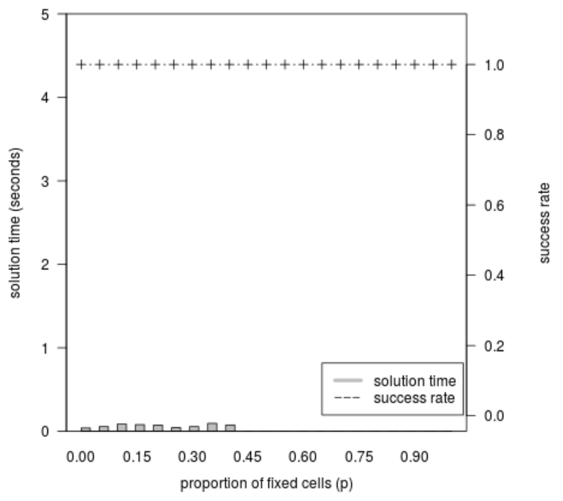
[7] ——, "On the combination of constraint programming and stochastic search: The sudoku case," in *Hybrid Metaheuristics*, ser. Lecture Notes in Computer Science, T. Bartz-Beielstein, M. Blesa Aguilera, C. Blum, B. Naujoks, A. Roli, G. Rudolph, and M. Sampels, Eds. Springer Berlin Heidelberg, 2007, vol. 4771, pp. 96–107. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75514-2_8

[8] M. Machado and L. Chaimowicz, "Combining metaheuristics and csp algorithms to solve sudoku," in *Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on*, Nov 2011, pp. 124–131.

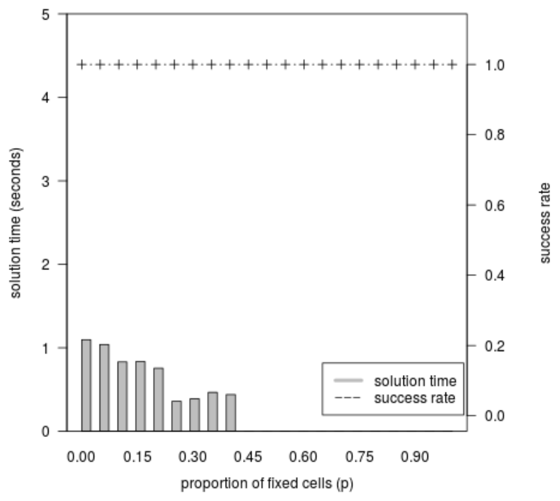
[9] T. Stützle, "Lokale suchverfahren für constrain satisfaction probleme: die *min conflicts* heuristik und tabu search," *KI*, vol. 11, no. 1, pp. 14–20, 1997.

[10] F. Glover, "Tabu search Part I," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989. [Online]. Available: http://dx.doi.org/10.1287/ijoc.1.3.190

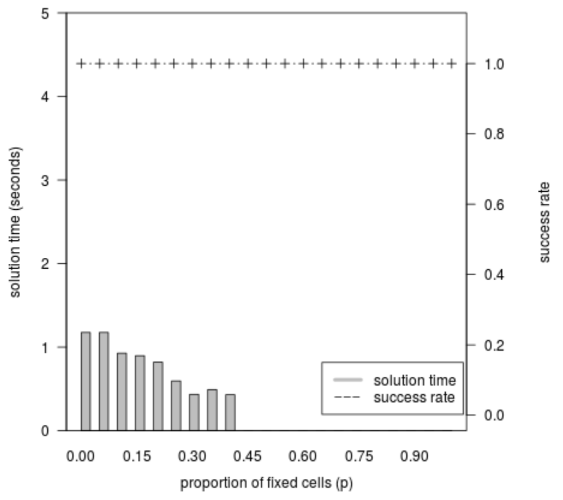
[11] H. R. Lourenço, O. Martin, and T. Stützle, "A beginners introduction to iterated local search," in *Proceedings of MIC*, vol. 2, Porto, Portugal, July 2001, pp. 1–6.



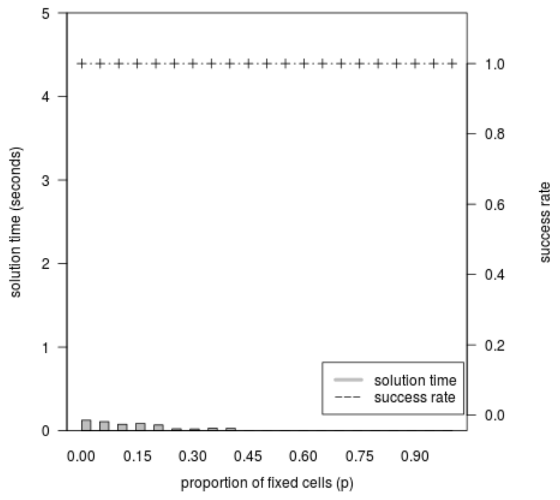
(a) Our solver



(b) Lewis



(c) Machado et al.



(d) Crawford et al.

Fig. 3. This figure compares the results for Sudoku puzzles of order 3. 3a shows the outcomes for the algorithm based on our solver which is presented in this paper, 3b shows the outcomes for the simulated annealing based algorithm from [7] and 3c and 3d show the results for the approaches from [8] and the constraint programming based algorithm from [3].

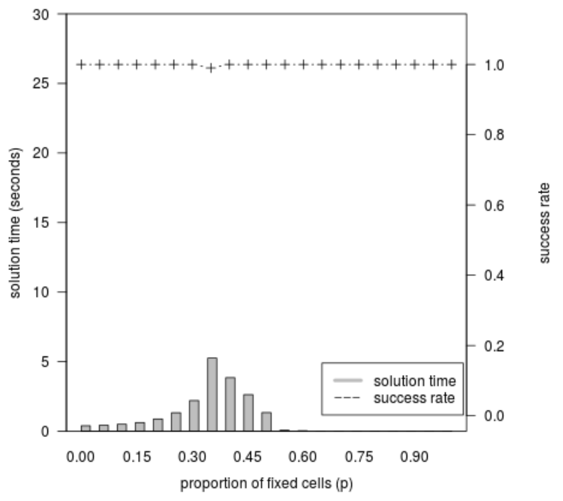
TABLE I
OVERVIEW OF AVERAGE RUNNING TIMES FOR THE TESTED ALGORITHMS CATEGORIZED BY THE PROPORTION OF FIXED CELLS (p) WITH PUZZLES OF ORDER 5. NOTE THAT ONLY RUNS WHICH LED TO A CORRECT OPTIMAL SOLUTION WERE CONSIDERED. (*Our solver*: THE ALGORITHM PRESENTED IN THIS PAPER, *Lewis*: ALGORITHM PROPOSED BY LEWIS [7])

Algorithm	**p=0.0**	**p=0.05**	**p=0.1**	**p=0.15**	**p=0.2**	**p=0.25**	**p=0.3**	**p=0.35**	**p=0.4**	**p=0.45**	**p=0.5**
Our solver	2.933s	3.368s	4.02s	5.403s	7.635s	11.652s	24.051s	35.826s	157.639s	178.723s	6.135s
Lewis	165.985s	154.067s	139.474s	124.474s	110.832s	101.967s	95.445s	105.23s	153.794s	130.172s	13.444s
Algorithm	**p=0.55**	**p=0.6**	**p=0.65**	**p=0.7**	**p=0.75**	**p=0.8**	**p=0.85**	**p=0.9**	**p=0.95**	**p=1.0**	
Our solver	0.85s	0.391s	0.163s	0.098s	0.05s	0.039s	0.028s	0.02s	0.01s	0.0s	
Lewis	1.027s	0.387s	0.16s	0.094s	0.047s	0.035s	0.027s	0.015s	0.005s	0.0s	

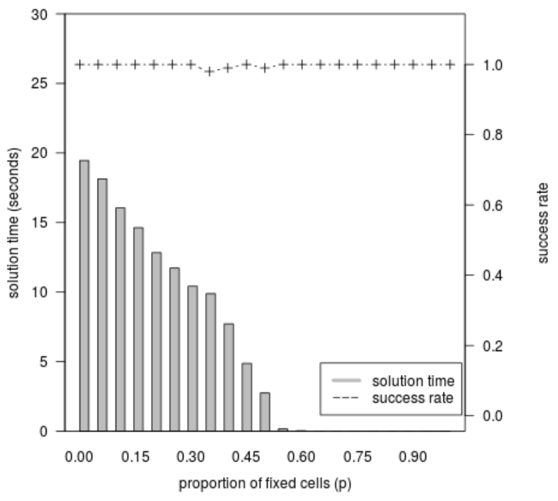
[12] T. Mantere and J. Koljonen, “Solving and analyzing sudokus with cultural algorithms,” in *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, June 2008, pp. 4053–4060.

[13] B. M. Smith, “The phase transition in constraint satisfaction problems: A closer look at the mushy region,” in *Artificial Intelligence*, 1993.

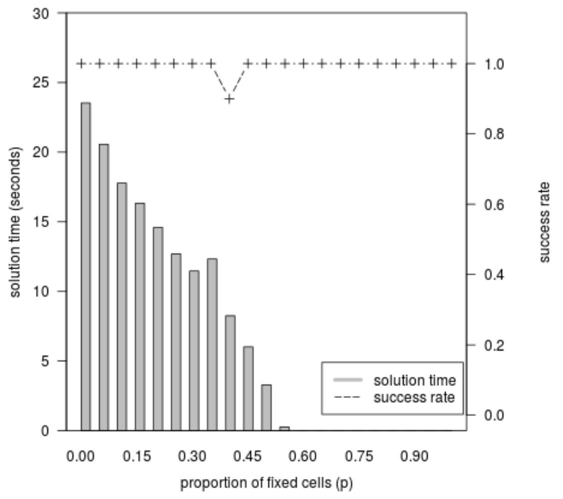
[14] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, “The irace package, iterated race for automatic algorithm



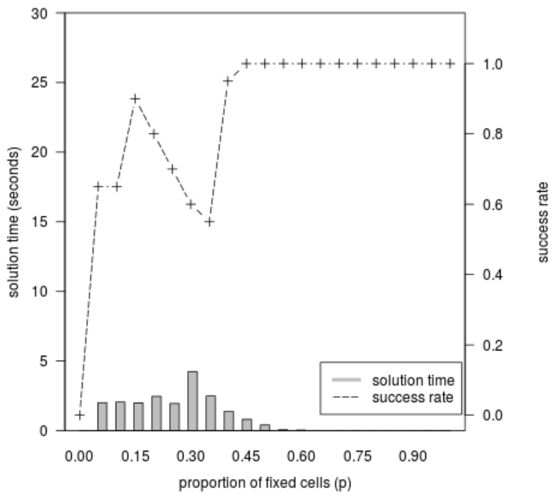
(a) Our solver



(b) Lewis



(c) Machado et al.



(d) Crawford et al.

Fig. 4. This figure compares the results for Sudoku puzzles of order 4. 4a shows the outcomes for the algorithm based on our solver which is presented in this paper, 4b shows the outcomes for the Simulated Annealing based algorithm from [7] and 4c and 4d show the results for the approaches from [8] and the Constraint Programming based algorithm from [3].

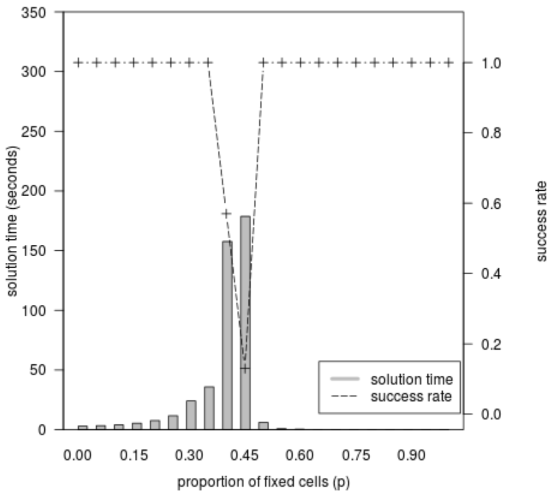
configuration," IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2011-004, 2011. [Online]. Available: http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-004.pdf

Nysret Musliu is a Priv.Dozent/Senior Scientist in the Institute of Information Systems at Vienna University of Technology. His research interests include AI problem solving and search, metaheuristic techniques, constraint satisfaction, machine learning and optimization, hypertree and tree decompositions, scheduling, and other combinatorial-optimization problems. He has a PhD in computer science from Vienna University of Technology. Contact him at musliu@dbai.tuwien.ac.at.

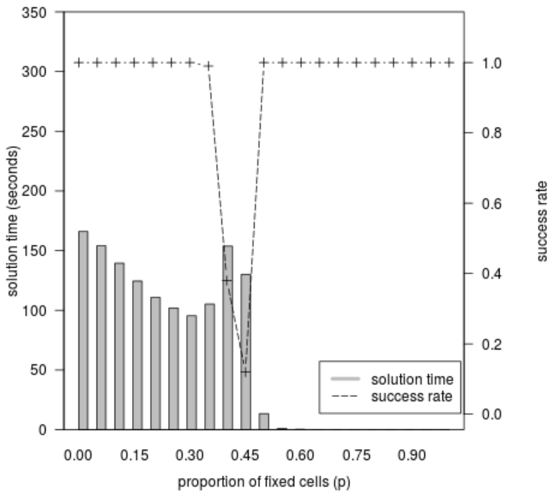
Felix Winter is a project assistant in the Institute of Information Systems at Vienna University of Technology. His research interests include constraint satisfaction problems, metaheuristics, hybrid approaches for solving optimization problems. He is currently a master student in computer science from Vienna University of Technology. Contact him at winter@dbai.tuwien.ac.at.

TABLE II
OVERVIEW OF THE SUCCESS RATES FOR THE TESTED ALGORITHMS CATEGORIZED BY THE PROPORTION OF FIXED CELLS (p) WITH PUZZLES OF ORDER 5. (*Our solver*: THE ALGORITHM PRESENTED IN THIS PAPER, *Lewis*: ALGORITHM PROPOSED BY LEWIS [7])

Algorithm	**p=0.0**	**p=0.05**	**p=0.1**	**p=0.15**	**p=0.2**	**p=0.25**	**p=0.3**	**p=0.35**	**p=0.4**	**p=0.45**	**p=0.5**
Our solver	100%	100%	100%	100%	100%	100%	100%	100%	57%	13%	100%
Lewis	100%	100%	100%	100%	100%	100%	100%	99%	38%	12%	100%
Algorithm	**p=0.55**	**p=0.6**	**p=0.65**	**p=0.7**	**p=0.75**	**p=0.8**	**p=0.85**	**p=0.9**	**p=0.95**	**p=1.0**	
Our solver	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	
Lewis	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	



(a) Our solver



(b) Lewis

Fig. 5. This figure compares the results for Sudoku puzzles of order 5. 5a shows the outcomes for the algorithm based on our solver which is presented in this paper, 5b shows the outcomes for the simulated annealing based algorithm from [7].