A Hybrid Approach for the Sudoku problem: Using Constraint Programming in Iterated Local Search

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

***Abstract*—Sudoku is not only a popular puzzle but also an interesting and challenging constraint satisfaction problem. Therefore, automatic solving methods for this problem have been the subject of several publications in the last two decades. Although current methods provide good solutions for small sized puzzles, larger instances remain challenging. This paper introduces a new local search technique based on the min-conflicts heuristic for Sudoku. Furthermore, we propose an innovative hybrid search technique that exploits constraint programming as perturbation technique within the iterated local search framework. We experimentally evaluate our methods on challenging benchmarks for Sudoku and report improvements over state of the art solutions. To show the generalizability of the proposed approach, we also applied our method on another challenging scheduling problem. The results show that the proposed method is also robust in another problem domain.**

***Index Terms*—Heuristic methods, Sudoku, Iterated Local Search, Constraint programming, Min-conflicts heuristic, Hybrid techniques**

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 is a typical constraint satisfaction problem. In [1] it has been shown that the decision problem which asks if there is a solution to the given Sudoku instance is NP-complete.

Although Sudoku may seem not to be a relevant problem on the first look, many large instances of this problem are still not solved satisfactorily. Therefore, they serve as very challenging benchmarks to test the robustness of new methods. Indeed such problems can contribute to the development of innovative techniques that can be applied in other areas of high practical relevance. In this paper we show that our efforts on solving challenging Sudoku problems resulted in methods that can also be useful in other problem areas like employee scheduling.

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.

A typical puzzle contains a number of cells already prefilled, which are considered fixed. 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 use this term 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 have 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: Exact methods and stochastic search based heuristics. Those two classes differentiate in several properties, but the most crucial difference lies within the fact that exact methods will always find the best solution available if given enough time, while stochastic search based approaches are non-deterministic and cannot guarantee to find the optimal solution.

Exact Sudoku solving techniques based on constraint programming (CP) have been well studied and proposed for instances that consist of grids with 9×9 cells. For example in [2], 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 [3]. A similar solving method is investigated in [4], where the authors conduct a comparison of different variable- and value-selection heuristics using backtracking search with constraint propagation.

In all of those publications it has been shown that approaches relying on CP work well on 9×9 puzzles and are also able to classify the difficulty of puzzles seen from a human's perspective. The latter feature becomes especially interesting when creating problems that are meant to be solved by humans and is further discussed in [2]. Additional improvements to solve puzzles of this size have been published in [5] and [6]. These papers focus on solving the hardest 9×9 problem instances by applying hybrid search techniques.

Modeling Sudoku as a satisfiability problem has been proposed in [7]. The authors present two encoding variants that can be used to solve puzzles by utilizing SAT-solvers and corresponding inference techniques. Experiments were conducted on 9×9 Sudoku and have shown comparable results to CP based approaches.

Larger Sudoku grids that consist of 16×16 or even 25×25 cells (that would correspond to an order of 4 or 5

			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.

respectively) and may have more than one possible solution, 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 [8] 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 [9] and [10]. Those articles introduce constraint propagation techniques to reduce the search space before applying the meta-heuristic driven search process. To the best of our knowledge, they set the state of the art for solving large Sudoku puzzles. However, although these techniques perform significantly better than exact methods, 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 considered in the literature. This shows in a significant 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 method for solving the Sudoku problem based on iterated local search which uses the min-conflicts heuristic. Additionally, our method includes constraint programming techniques that are applied during the perturbation phases of an iterated local search based procedure. Although local search techniques in hybridization with CP have been previously proposed in the literature, to the best of our knowledge the ideas used in this paper regarding min-conflicts and our perturbation methods during iterated local search are innovative and have not been considered before. With the use of a random instance generator that has been 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 for Sudoku, where we report improvements regarding the success rate. Additionally, we report results of using the proposed methods on another problem from the scheduling area and thereby show the generalizability of our algorithm.

In the following section we will give background information on constraint programming and local search. Section III will then give a detailed description of our approach. Information about conducted experiments and the configuration of our algorithm will then be given in sections IV and V. We will present our results and make a comparison with existing approaches in Section VI. Finally, the application of our method on a scheduling problem is shown in VII. Concluding remarks will be given in the last section.

II. Background

The majority of approaches that have been proposed to solve the Sudoku problem can be grouped in two classes of techniques: Exact solution methods based on constraint programming and meta-heuristic approaches based on local search. In the following, we will shortly introduce the fundamental concepts regarding those solution strategies.

Methods using constraint programming [11] describe a given problem by a set of variables, their domains and constraints over these variables. The general approach to find a solution with CP lies in intelligently enumerating variable assignments and checking their feasibility. In order to reduce the potentially large number of possible assignments to check, different techniques like backtracking tree search and constraint propagation have been proposed.

Large NP-complete problems usually cannot be solved in feasible time by exact techniques (like it is the case for Sudoku). For such problems typically heuristic techniques are used that give some solutions in reasonable time, but do not guarantee optimality. In this paper we apply heuristic techniques based on local search [12]. Those search methods start from an initially generated solution and then try to improve it by iteratively applying small modifications.

III. A new approach for solving the Sudoku problem

In order to describe our problem formulation we will use the terminology introduced by Lewis [8] which defines the notion of a *square*. A *square* refers to each of the $n \times n$ sized sub-grids that form the Sudoku puzzle.

Furthermore, $square_{r,c}$ denotes the *square* in row *r* and column *c*, considering an instance as a grid of *squares* and $r, i \in \{1, ..., n\}$. In a similar fashion a value of a cell in row *i* and column *j* of the overall $n^2 \times n^2$ sized grid is referred to as $cell_{i,j}$, where $i, j \in \{1, ..., n^2\}$. 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*.

A. Representation and neighborhood

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 then in each search step choose two different unfixed cells in the same square and swap them. Details on how those cells are selected will be 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. Therefore, we use the cost function proposed in [8], which looks at each row and column individually. For each row/column all missing numbers from 1 to n^2 are counted and summed up. An optimal solution without any constraint violations will therefore have a cost of 0. The objective function f for a candidate solution S is defined as:

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

where $r(i)$ and $c(i)$ represent the number of missing values in row i or column i respectively. Obviously a conflict necessarily arises wherever a single number appears multiple times in a row or column.

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

C. Applying the min-conflicts heuristic on Sudoku

In order to achieve an effective local search for the Sudoku problem we use a variant of the min-conflicts heuristic [13]. The general idea behind this heuristic lies in concentrating on variables that cause conflicts and to remove those conflicts by swapping the affected cells to better positions. The procedure works in two steps: 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 red background highlight these conflicts). The numbers outside the Sudoku grid (highlighted 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 if 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 can get stuck in local optima easily. To work against this issue, we additionally make use of a data structure called tabu list, that stores recently performed swaps in order to prevent cyclic changes to a solution. All swapping moves that are contained in this list, are considered to be tabu for a number of forthcoming iterations, which means that they will not be considered as potential cell swaps. One exception to this rule are swaps that would lead to a cost decrease that could beat the best found solution so far. If this so called aspiration criterion [14] is fulfilled, a swap will be allowed even if it is considered to be tabu. As soon as the best swap candidates have been determined, usually the change would just be performed and the search would proceed to the next iteration. Other local search variants only accept it if evaluation yields a decrease of the cost function. 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 swaps 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 we use is described in Algorithm 1.

D. Using constraint programming methods in iterated local search

Although basic local search often can produce satisfying results, it has been shown in the literature ([9], [10]) that the introduction of constraint programming methods can bring significant improvements to the algorithm.

One simple variant which was first described in [9] uses constraint propagation to reduce the domains for each cell variable until all variables are arc consistent before performing local search. Any unfixed cell that has only one possible domain value left can then be considered as a prefixed cell containing that value. The problem's search space can often be significantly reduced by using this technique.

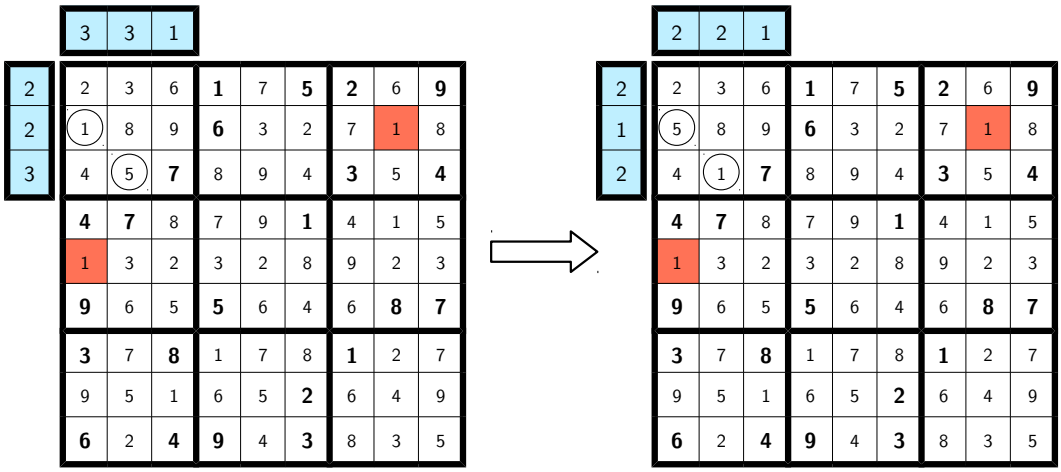


Fig. 2. In this figure an example neighborhood move applying the min-conflicts heuristic is illustrated.

Algorithm 1 Min conflicts heuristic with tabu list for Sudoku

Input: puzzle, iterationLimit, acceptanceProbability
1: initialize tabu list
2:
3: $iterationCounter \leftarrow 0$
4: $bestCost \leftarrow$ MAX
5: $currentCost \leftarrow$ 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

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 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 any 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 approaches based on backtracking for the Sudoku problem have been examined in the literature ([2] and [4]) and we implemented a variant based on the results presented in these two publications for our purpose. 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. Although there exists work on solving Sudoku with similar CP methods, to the best of our knowledge a combination with meta-heuristic methods through a perturbation mechanism for iterated local search (ILS) [15] has not been applied.

The main idea behind iterated local search is to examine the search space by iteratively calling an embedded local search. After a local optimum has been found the best known solution so far is perturbed to provide a good starting point for the next run of the meta-heuristic procedure.

We utilize iterated local search in our algorithm as follows: 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. This can lead to three different outcomes: Firstly, the optimal solution could have been found using CP. 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 continue with local search from this solution. Iterated local search keeps repeating this overall process until a given time limit is reached.

In our perturbation method the cells that should be emptied additionally to the ones which are causing conflicts influence the search space by our FC procedure. 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 is described.

Algorithm 2 Iterated local search for Sudoku

Input: puzzle, timeLimit, resetFactor, α
1: FIXCELLSUSINGARCCONSISTENCY($puzzle$)
2:
3: FILLREMAININGCELLSRANDOMLY($puzzle$)
4:
5: $bestPuzzle \leftarrow puzzle$
6: $bestCost \leftarrow$ EVALUATE($puzzle$)
7:
8: **while** $bestCost > 0 \wedge$ timeLimit not passed **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

E. 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 the *iterationLimit* is set to 1000, the cost of the best solution found so far is 15, and no solution with a lower cost than 15 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 swap candidate has been selected with min-conflicts, it has to be determined whether or not the corresponding change 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 this parameter for example is set to 15%, then in around 15 out of 100 cases a change which does not introduce any improvement will be accepted.

The *resetFactor* parameter is a floating point number within the range from 0.0 to 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 procedure. 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 iterative reduction of the *resetFactor* and is applied at the end of each perturbation phase. It also is given to the algorithm as a 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 total search 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 this limit, the algorithm is forced to end the exact search.

IV. EXPERIMENTAL ENVIRONMENT

In this section we compare our proposed hybrid solver to the state of the art algorithms for Sudoku. Additionally, we have a closer look on the effects of using constraint programming as perturbation mechanism for iterated local search.

We contacted the authors of [9], [10] and [4] 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 [9] and [10]. The CP parts from [9] had to be reimplemented. Regarding [4], we implemented the algorithm based on the instructions from the authors. We first experimented with a set of 9×9 Sudoku puzzles from [16] which are known to be challenging. However, our algorithm could solve each of those within one second and since those instances have not been shown to be challenging for our solver we considered the generation of harder instances with the use of a random instance generator from [8]. This program creates puzzles of any size by simply removing some randomly selected cells of a presolved puzzle. We followed Lewis' experimental approach (described in [8]) and created many puzzles in 20 different categories, 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. We applied this generation procedure for instances with an order of three, four and five. With 400 puzzles per order we generated a sum of 1200 instances for our experiments. Since most of the discussed algorithms rely on stochastic search, we performed 20 repeated test runs on each puzzle instance. Note that for puzzle instances with an order of five, experiments have been conducted exclusively for the simulated annealing based algorithm by Lewis and our min-conflicts based algorithm, since the other considered algorithms did not produce competitive results even for instances with an order of four.

All the tests were run on an Intel Xeon E5345 2.33GHz with 48GB RAM. The instances used in this paper as well as the sources of our 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 [8] 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 restricted the runtime of our experiments to five seconds for Sudoku of order three, 30 seconds for order four and a 350 seconds for order five Sudoku. Time limits were chosen based on the experiments in [8]. We agree that those limits fit the experimental environment well, since most of the puzzles can be solved within this time.

V. Algorithm configuration

Parameters for the considered algorithms from the literature ([9], [4] and [10]) 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 from the benchmark instances. As mentioned in [8], there is an 'easy-hard-easy' phase transition [17] depending on the relative number of prefilled cells. We focused on the hardest problems (which have between 40 % and 45 % of the cells fixed initially) when experimenting with different parameter settings.

Since the *iterationLimit* parameter limits the number of trials for swapping two cells during local search, we decided to set its value relative to the number of cells in the grid. We experimented multiplying 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* was 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* was set to 1.0. Therefore, 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, so that only conflicting cells will be perturbed. 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 [18]. We kept all of the irace default settings and limited the tuning budget (maximum number of runs) for the algorithm to 1000. 20 of the puzzles that have an order of 5 and 40% of their cells fixed served as tuning instances. The elite candidates produced by irace 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 even better results. Therefore those parameter values were used in our final experiments.

VI. Results

We conducted experiments for puzzles of order three and four using four different algorithms: The simulated annealing based approach from [9] and its variation from [10], the constraint programming based solver from [4] and finally our algorithm proposed in this paper. For Sudoku instances with an order of five, experiments were only conducted for our solver and the algorithm from [9], which produced better results compared to the two other algorithms from the literature for Sudoku with an order of four.

Figures 3, 4, 5 display a graphical representation of the results, for Sudoku of order three, four and five respectively. All of them show the average running times of successful runs for each category in the form of bars. The corresponding success rates are shown as punctuated lines.

We can see that the approach from Crawford et. al [4] produces 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 four. It shows that for larger problems the meta-heuristic approaches using simulated annealing combined with constraint propagation techniques deliver better results.

This can be seen in Figures 4b and 4c that visualize results from the algorithms by Lewis [8] and Machado et al. [10]. When comparing those two approaches, Lewis' implementation slightly outperforms the algorithm from Machado et al. regarding the success rate. Therefore, we only compare our approach to the former when considering benchmark instances with an order of five.

The hybrid algorithm 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 with an order of four and five. This can be seen in 4 and 5. Detailed values regarding the success rates and running times for instances of order five 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 [9] are 38% and 12%, whereas our algorithm has success rates of 57% and 13%.

VII. Application of our method in scheduling

To the best of our knowledge the hybridization of min-conflicts with iterated local search using a constraint programming based perturbation has not been considered before. As we have shown, this hybrid algorithm gives very good results for the Sudoku problem. In order to investigate the generalizability of the proposed approach, we applied our method on a practical problem from the employee scheduling area ([19], [20]).

The overall goal of the considered employee scheduling problem is to find an optimal roster for a number of given employees and shift types, where every employee may either work in a single shift or have a day off on each day of a given scheduling period spanning over multiple weeks. The employees and shift types which are considered in this problem are specified by a list of unique names which are connected with a number of constraints that restrict all possible shift assignments. Some employees might for example be only allowed to work in certain shift types and patterns of consecutive working shifts might be prohibited or requested. Each problem instance specifies hard- and soft-constraints to set up a corresponding rule set. Hard constraints on the one hand are always strict and have to be fulfilled in order to generate a feasible solution. Soft constraints on the other hand may be violated, but will in case of a violation lead to an integer valued penalty. For example one of the hard constraints specifies the minimum and maximum amount of time that an employee can work during the whole scheduling horizon. Personal shift requests of employees are formulated as soft constraints. Finally, the objective function of a candidate solution is defined as the sum of penalties caused by the violated soft constraints. We therefore deal with an optimization problem, where the optimal solution is a feasible schedule with the lowest possible objective value. The complete definition for this problem is given in [19].

In order to use our approach on this problem we applied three different search neighborhoods which have been proposed in [21]. Those neighborhoods make local changes to the schedule by swapping blocks of shifts horizontally and vertically or by directly reassigning blocks of shifts. We implemented a local search procedure based on min conflicts that generates the corresponding neighborhoods by selecting cells that are causing constraint violations. Additionally, we devised a constraint programming approach that uses a forward checking search to solve partial schedules. Both local search as well as the CP based solution techniques were then combined within iterated local search to perturb solutions in a similar way as it has been proposed for the Sudoku problem.

To show the benefits of using CP as a perturbation mechanism for iterated local search, we compared our method with a classical iterated local search that performs a simple perturbation by randomly reassigning cells that are causing constraint violations. Experiments were then conducted with 24 different instances using five repeated runs per instance within a time limit of 10 minutes and two repeated runs per instance within a time limit of 60 minutes. Table III displays the best results and also compares them with results obtained by a state of the art heuristic that is based on ejection chains [19].

The algorithm that makes use of a CP based perturbation produces the best schedules for 16 of the 24 instances within the time limit of 10 minutes and for 17 of the 24 instances within the time limit of 60 minutes when compared with existing methods. These results show the robustness of our method in this domain. The ejection chain based approach, which also includes a construction method that creates an initial solution at the start of the algorithm, shows to be better only for the five largest instances. However, with the inclusion of a construction heuristic for the generation of initial solutions, our method was able to reach better results for 23 of the 24 instances (see columns 6 and 7 of Table III).

VIII. Conclusion

In this paper we proposed a novel iterated local search algorithm that exploits CP techniques in the perturbation phase to solve the Sudoku problem. We compared our approach to state of the art methods from the literature and 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.

Additionally, experiments on instances of a well known scheduling problem have shown the generalizability of our approach.

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

References

[1] Y. Takayuki and S. Takahiro, "Complexity and completeness of finding another solution and its application to puzzles," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 86, no. 5, pp. 1052–1060, 2003.

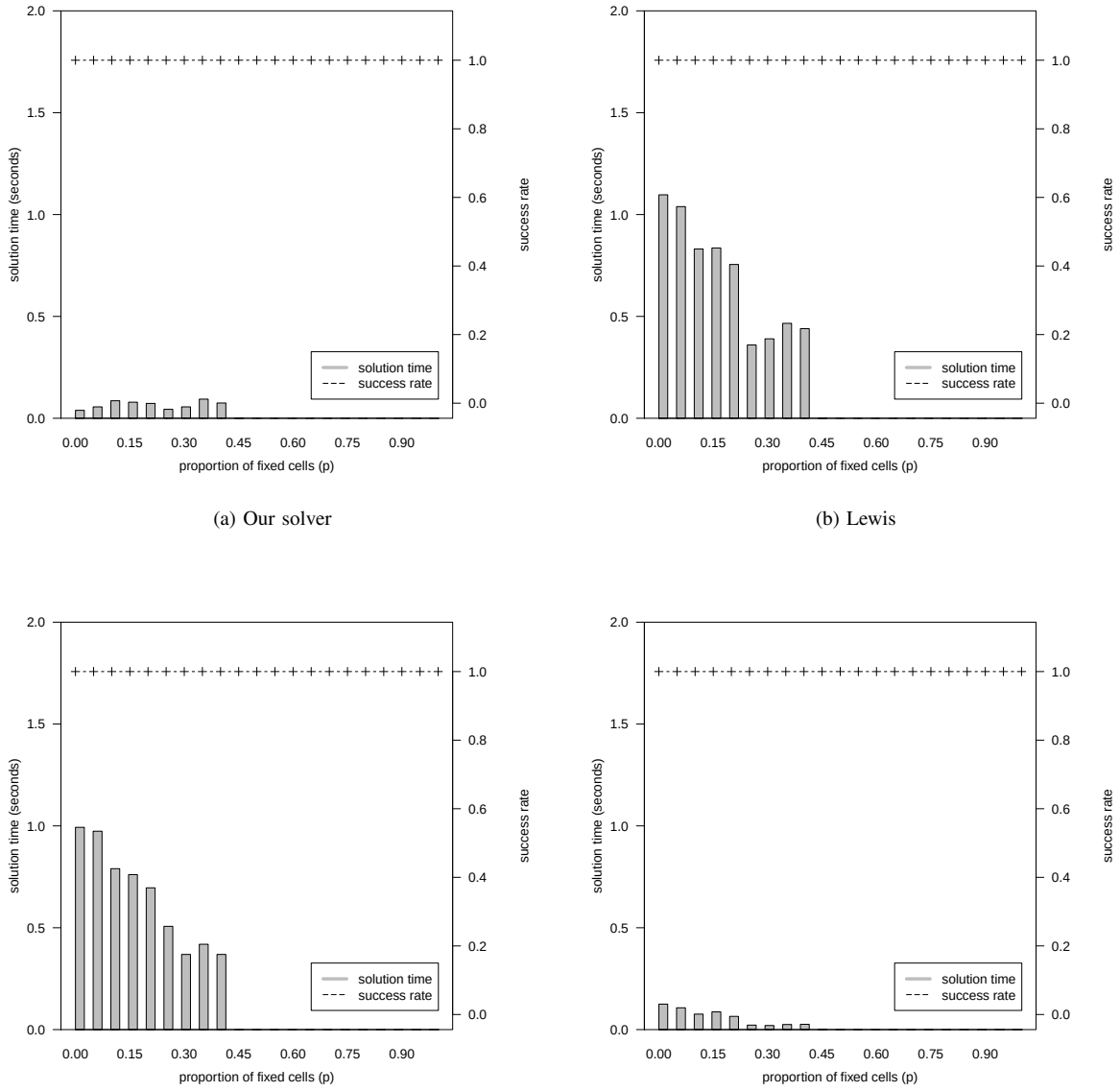
[2] H. Simonis, "Sudoku as a constraint problem," in *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, 2005, pp. 13–27.

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

[4] 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.

[5] 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

[6] 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



(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 [9] and 3c and 3d show the results for the approaches from [10] and the constraint programming based algorithm from [4].

[7] I. Lynce and J. Ouaknine, "Sudoku as a SAT problem," in *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2006, Fort Lauderdale, Florida, USA, January 4-6, 2006*, 2006. [Online]. Available: http://anytime.cs.umass.edu/aimath06/proceedings/P34.pdf

[8] 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

[9] ——, "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

[10] 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.

[11] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence. Elsevier, 2006, vol. 2. [Online]. Available: http://www.sciencedirect.com/science/bookseries/15746526/2

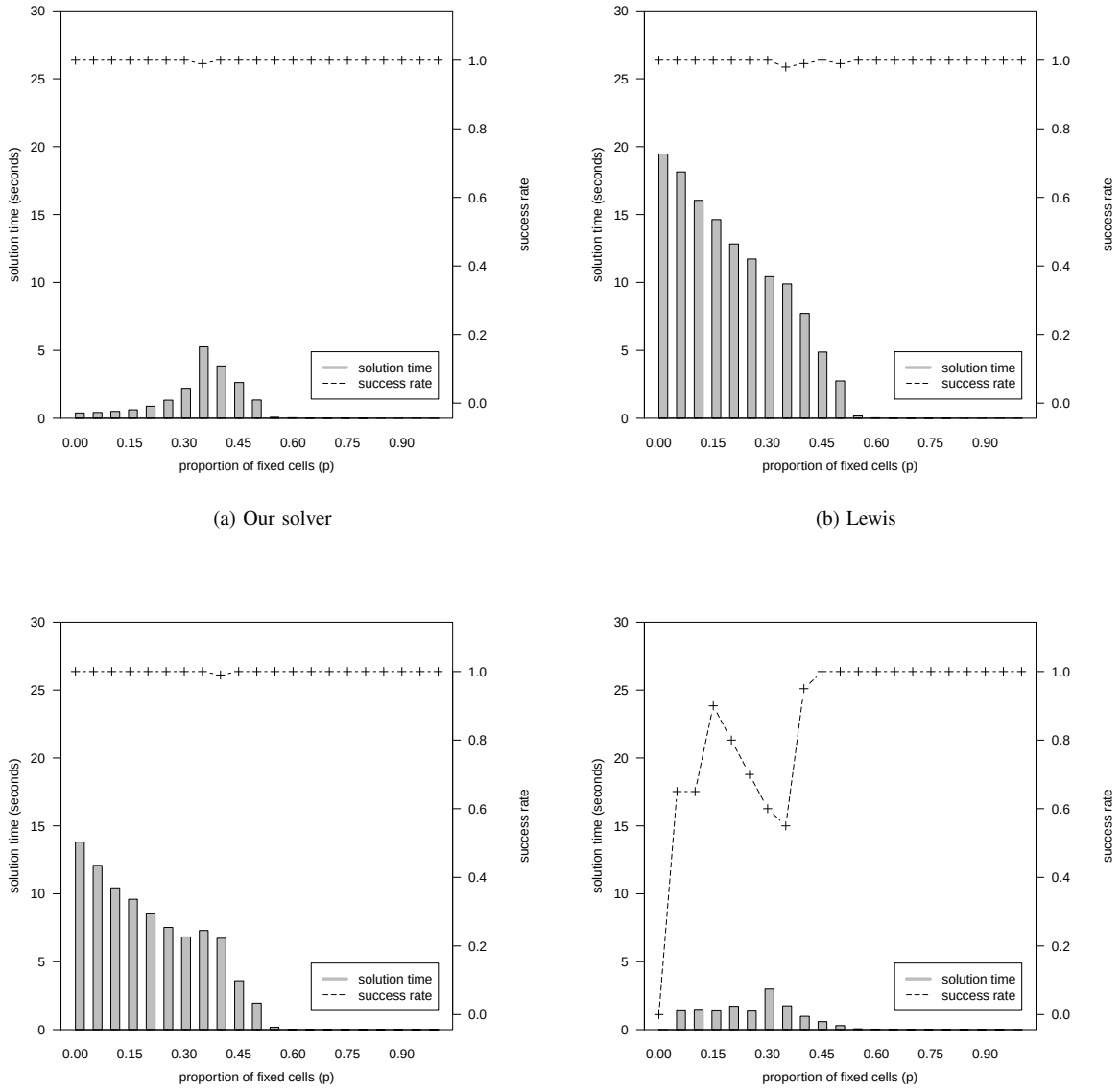
[12] H. H. Hoos and T. Stützle, *Stochastic local search: Foundations & applications*. Elsevier, 2004.

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

[14] 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

[15] 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.

[16] 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*



(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 [9] and 4c and 4d show the results for the approaches from [10] and the Constraint Programming based algorithm from [4].

on, June 2008, pp. 4053–4060.

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

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

[19] T. Curtois and R. Qu, "Computational results on new staff scheduling benchmark instances," ASAP Research Group, School of Computer Science, University of Nottingham, NG8 1BB, Nottingham, UK, Tech. Rep., October 2014.

[20] E. K. Burke and T. Curtois, "New approaches to nurse rostering benchmark instances," *European Journal of Operational Research*, vol. 237, no. 1, p. 7181, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221714000605

[21] E. K. Burke, T. Curtois, R. Qu, and G. V. Berghe, "A scatter search methodology for the nurse rostering problem," *JORS*, vol. 61, no. 11, pp. 1667–1679, 2010. [Online]. Available: http://dx.doi.org/10.1057/jors.2009.118

Nysret Musliu is a Priv.Dozent/Senior Scientist at 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.

TABLE I

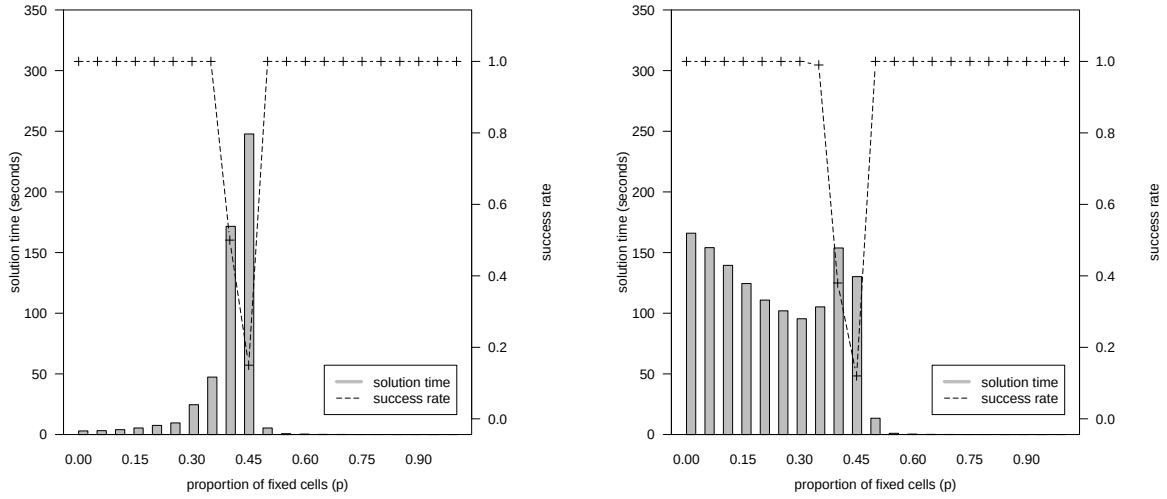
OVERVIEW OF AVERAGE RUNNING TIMES FOR THE TESTED ALGORITHMS CATEGORIZED BY THE PROPORTION OF FIXED CELLS (p) WITH PUZZLES OF ORDER 5. NOTE THAT ONLY RUNS WHICH LED TO A CORRECT OPTIMAL SOLUTION WERE CONSIDERED. (*Our solver*: THE ALGORITHM PRESENTED IN THIS PAPER, *Lewis*: ALGORITHM PROPOSED BY LEWIS [9])

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	

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 [9])

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 [9].

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

TABLE III

THIS TABLE COMPARES THE RESULTS FROM DIFFERENT APPROACHES ON THE EMPLOYEE SCHEDULING PROBLEM. A - MEANS THAT NO FEASIBLE SOLUTION COULD BE FOUND WITHIN THE GIVEN TIME LIMIT. *COLUMN 6 AND 7 (ILS & CP*) PRESENT THE RESULTS OF ITERATED LOCAL SEARCH WITH A CP BASED PERTURBATION AND AN ADDITIONAL CONSTRUCTION HEURISTIC FOR THE GENERATION OF AN INITIAL SOLUTION.

Instance	ILS		ILS & CP		ILS & CP*		Ejection Chain [19]	
	10 min	60 min	10 min	60 min	10 min	60 min	10 min	60 min
Instance 1	607	607	607	607	607	607	607	607
Instance 2	828	828	828	828	828	828	923	837
Instance 3	1001	1003	1001	1001	1001	1001	1003	1003
Instance 4	1721	1718	1722	1717	1716	1716	1719	1718
Instance 5	1244	1237	1237	1235	1150	1147	1439	1358
Instance 6	2254	2159	2245	2165	2145	2050	2344	2258
Instance 7	1176	1178	1078	1072	1090	1084	1284	1269
Instance 8	-	1886	1549	1446	1548	1464	2529	2260
Instance 9	466	475	455	455	454	454	474	463
Instance 10	4960	4875	4769	4750	4660	4667	4999	4797
Instance 11	3578	3494	3459	3462	3470	3457	3967	3661
Instance 12	4538	4768	4629	4216	4338	4308	5611	5211
Instance 13	3568	2801	3461	2767	3157	2961	8707	3037
Instance 14	-	-	1668	1512	1430	1432	2542	1847
Instance 15	-	-	4861	4737	4871	4570	6049	5935
Instance 16	4057	-	3869	3636	3754	3748	4343	4048
Instance 17	6902	6916	7035	6606	6720	6609	7835	7835
Instance 18	5525	5509	5944	5604	5400	5416	6404	6404
Instance 19	6654	4748	6551	4573	4780	4364	6522	5531
Instance 20	-	-	-	-	8763	6654	23531	9750
Instance 21	-	82541	-	-	33163	22549	38294	36688
Instance 22	-	-	-	-	192946	48382	-	516686
Instance 23	488156	320788	480064	321094	189850	38337	-	54384
Instance 24	1208465	940803	1202862	942501	519173	177037	-	156858