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

Nysret Musliu and Felix Winter, *Vienna University of Technology*

Sudoku is a challenging constraint satisfaction problem. Current methods provide solutions for small puzzles, but a new search technique based on the min-conflicts heuristic exploits constraint programming as a perturbation technique for larger instances.

Sudoku is a popular logic puzzle in which you have to fill in a grid with numbers that typically lie between one and nine, without duplicating numbers in columns or rows. From a scientific viewpoint, it's a typical constraint satisfaction problem, with one study[1] finding that the decision problem asking for a solution to a given Sudoku instance is NP-complete.

Although Sudoku might not seem to be a relevant problem at first glance—after all, it appears in daily newspapers—many large instances of it still aren't solved satisfactorily, meaning they serve as very challenging benchmarks to test the robustness of new methods. Indeed, in this article, we show that our efforts to solve challenging Sudoku problems result in methods that can also be useful in other problem areas such as employee scheduling.

Formally, a Sudoku puzzle instance can be described as an $n^2 \times n^2$ grid divided into n^2 distinct squares that in turn divide the whole grid into $n \times n$ subgrids. 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:

- In every row, the numbers 1 to n^2 appear exactly once.
- In every column, the numbers 1 to n^2 appear exactly once.
- In every $n \times n$ subgrid, the numbers 1 to n^2 appear exactly once.

A typical puzzle contains a number of prefilled cells, which are considered fixed. The variable n determines the size and also to some degree the difficulty. This is sometimes referred to as the puzzle's *order*, and we'll use this term going forward. Problems that are meant to be solved by the human mind usually have an order of three—most instances published in newspapers have this size.

In the literature, different techniques have been proposed to solve Sudoku puzzles. Two large groups of methods and techniques

1541-1672/17/$33.00 © 2017 IEEE
Published by the IEEE Computer Society

have been repeatedly applied: exact methods and stochastic search-based heuristics. These two classes differentiate in several properties, but the most crucial difference lies in the fact that exact methods will always find the best solution available if given enough time, while stochastic search-based approaches are nondeterministic and can't guarantee they'll 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, one author[2] introduced a formal model of Sudoku as a constraint satisfaction problem; another study[3] investigated a similar method, in which the authors compared different variable- and value-selection heuristics using backtracking search with constraint propagation.

In all these 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. Additional improvements to solve puzzles of this size have been published elsewhere[4,5] and focused on solving the hardest 9×9 problem instances by applying hybrid search techniques. Modeling Sudoku as a satisfiability problem has been proposed elsewhere as well.[6] The authors in that particular study presented two encoding variants that can be used to solve puzzles by utilizing Boolean Satisfiability (SAT) solvers and corresponding inference techniques. Experiments were conducted on a 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 (corresponding to an order of four or five, respectively) introduce new challenges. Exact methods that try to find a solution by applying intelligent enumeration mechanisms come to their limits here: the search space is simply too large to enumerate all solutions in a feasible amount of time. To solve larger instances, one author[7] introduced the first metaheuristic-driven approach for Sudoku puzzles with an implementation that uses simulated annealing. The study concluded that puzzles of higher order can be tackled by heuristic techniques and could outperform exact methods when solving such problems. These ideas and problem formulations have since been extended[8,9] in papers introducing constraint propagation techniques to reduce the search space before applying the metaheuristic-driven search process. To the best of our knowledge, these works set the state of the art for solving large Sudoku puzzles. But even though they perform significantly better than exact methods, there's still room for improvement. Sudoku instances that consist of a grid with 25×25 cells and 55 percent of their cells filled in form the hardest class of problems considered in the literature. There's a significant drop in the success rate when using the simulated annealing-based solver in all the published results, so we focus here on large problem instances.

A typical puzzle contains a number of prefilled cells, which are considered fixed. The variable n determines the size and also to some degree the difficulty. This is sometimes referred to as the puzzle's *order*.

Specifically, this article proposes a new method for solving the Sudoku problem based on iterated local search that uses the min-conflicts heuristic. Additionally, our method includes CP 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 here regarding min-conflicts and our perturbation methods during iterated local search are innovative and haven't been considered before. With the use of a random instance generator that has been proposed in the literature, we randomly generated a total of 1,200 puzzle benchmark instances. We experimentally evaluated our methods by comparing our results with state-of-the-art methods for Sudoku.

A New Approach for Solving the Sudoku Problem

To describe our problem formulation, we use the terminology introduced by Rhydian Lewis,[7] who defines the notion of a *square* for each of the $n \times n$-sized subgrids 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 $r, c \in \{1, \ldots, n\}$. In a similar fashion, the 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 $i, j \in \{1, \ldots, n^2\}$. A cell that has its value

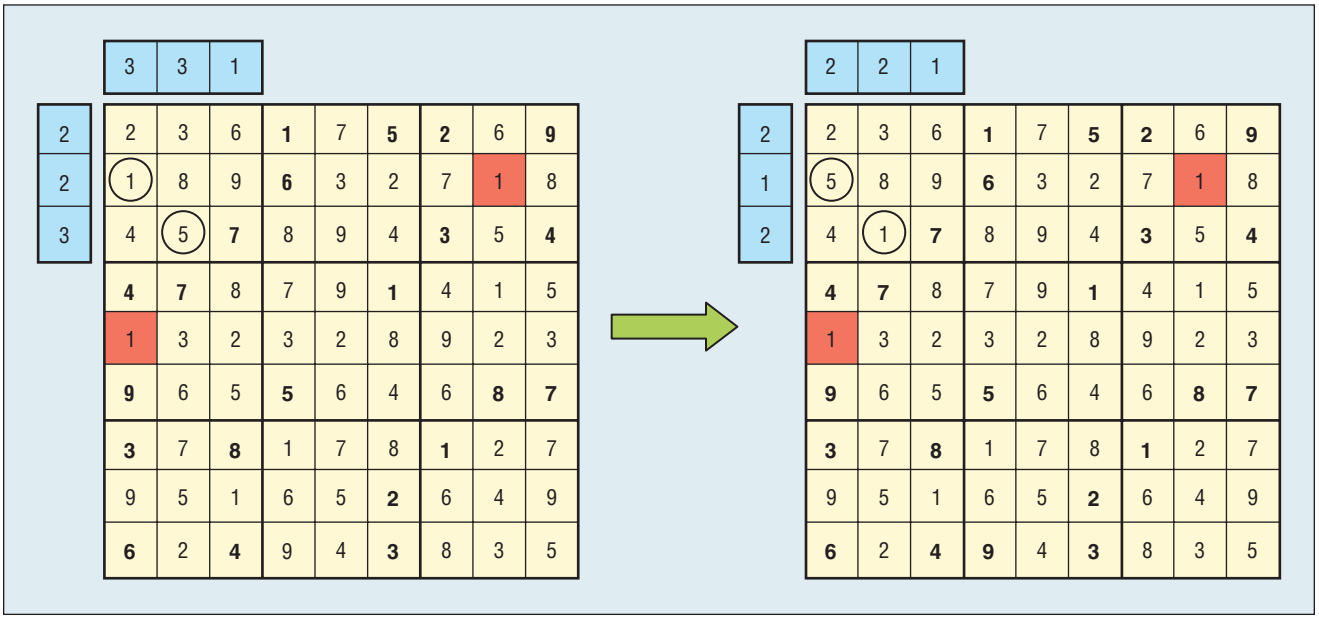

Figure 1. An example neighborhood move applying the min-conflicts heuristic. (a) The grid as it appears before the cell swap. The circled value 1 in the upper left subgrid represents the randomly selected cell that's in conflict (the two cells with the red background highlight these conflicts). (b) The grid after the swap has been performed. The number of conflicts for the affected rows and columns has been decreased successfully.

predefined in the puzzles is called *fixed*, whereas a cell that's initially empty and has to be filled by the solver is referred to as *unfixed*. Finally, a grid that's complete and fulfills all of the problem's constraints is referred to as *optimal*.

Representation and Neighborhood

In our local search techniques, we use a direct representation of the Sudoku grid. To generate an initial solution, all the puzzle's unfixed cells are filled randomly in such a way that the third constraint of the puzzle won't 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. 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 a solution's objective value.

Evaluating Candidate Solutions

Because two cells lying inside the same square can never contain the same number throughout a 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 Lewis's work,[7] 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

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

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.

To keep the required time consumed during the calculation of the cost function as low as possible, we apply 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.

Applying the Min-Conflicts Heuristic

To achieve an effective local search for the Sudoku problem, we use a variant of the min-conflicts heuristic. The general idea behind this heuristic lies in concentrating on variables that cause conflicts and removing those conflicts by swapping the affected cells to better positions. The procedure works in two steps: a conflicting cell is selected randomly, and then a good swap partner is determined.

Figure 1 illustrates the use of the min-conflicts heuristic. The numbers outside the Sudoku grid (highlighted with the blue background) give information about the number of missing values in the considered

```
Input: puzzle, iterationLimit, acceptanceProbability
   initialize tabu list

    iterationCounter←0
    bestCost← MAX
    currentCost← MAX

    while bestCost> 0 ∧iterationCounter < iterationLimit do
       randomly select cell which is in conflict

       generate all possible swaps with the selected cell

       bestSwap ← Find the best swap which minimizes total conflicts
       bestSwapNotTabu ← Find the best swap which minimizes total conflicts and is not tabu

       if bestSwap ≠ bestSwapNotTabu then
              if evaluate(bestSwap) < bestCost then
                     currentCost ← evaluate(bestSwap)
                     perform swap
                     go to update tabu list
              end if
       end if
       if evaluate(bestSwapNotTabu) < currentCost ∨random() <= acceptanceProbability then
              currentCost ← evaluate(bestSwapNotTabu)
              perform swap
       end if

       update tabu list
       if currentCost< bestCost then
              bestCost ← currentCost
              iterationCounter ← 0
       else
              iterationCounter ← iterationCounter + 1
       end if
    end while
   Output: best solution
```

Algorithm 1. Min-conflicts heuristic with tabu list for Sudoku.

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. In this case, value 5, which is also circled, is selected (note that a swap with any other cell wouldn't move the value 1 to a good position). The right side of the figure shows 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 avoid this problem, we considered the combination of a tabu list and the min-conflicts heuristic, which has also been used in other problem domains.[10] The tabu list stores recently performed swaps 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 won't 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 is fulfilled, a swap will be allowed even if it's 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 that 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 that 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. Algorithm 1 shows the overall local search procedure.

Using Constraint Programming Methods in Iterated Local Search

Although basic local search can often produce satisfying results, it has been shown in the literature[8,9] that the introduction of CP methods can bring significant improvements to the algorithm.

One simple variant[8] 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.

In this article, we propose to include a CP approach based on forward checking (FC) with dynamic variable ordering that's applied in between iterated phases of local search and has the goal of intensifying the search of promising areas in the search space. Whenever this procedure is called during search, all unfixed cells that cause any conflicts plus some additional unfixed cells are emptied, and the solver tries to find a solution by filling the missing cells with CP methods.

CP approaches based on backtracking for the Sudoku problem have been examined in the literature.[2,3] Our variant basically performs a backtracking search using FC, 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 metaheuristic methods through a perturbation mechanism for iterated local search (ILS) hasn't 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 metaheuristic 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 enters its perturbation phase, in which a further examination of the nearby search space using CP takes place. The perturbation process is conducted by emptying several unfixed cells and then performing FC search. This can lead to three different outcomes: the optimal solution could have been found using CP, FC could detect that there's no possible solution for this particular candidate instance, or 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 that also contains cells that have been filled in the perturbation phase. The algorithm will then fill the remaining cells randomly 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 that are causing conflicts influence the search space by our FC procedure, so 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 percent of the cells will be emptied before the FC procedure is started. To change this factor during the overall search, we iteratively reduce the reset factor after every perturbation phase via multiplication with a parameter α that 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. Algorithm 2 describes the overall search process based on iterated local search.

Experimental Environment

We contacted the authors of various studies[3,8,9] for the source code of their implementations, so that we could compile the solvers and conduct a fair comparison of the results. We first experimented with a set of 9×9 Sudoku puzzles from one study[11] that were known to be challenging. However, our algorithm could solve each of these within one second, and because these instances haven't been shown to be challenging for our solver, we considered the generation of harder instances with the use of a random instance generator,[7] which created puzzles of any size by simply removing some randomly selected cells of a pre-solved puzzle. We followed Lewis's experimental approach[7] and created many puzzles

```
Input: puzzle, timeLimit, resetFactor, α
   1. fixCellsUsingArcConsistency(puzzle)
   2.
   3. fillRemainingCellsRandomly(puzzle)
   4.
   5. bestPuzzle ← puzzle
   6. bestCost ← evaluate(puzzle)
   7.
   8. while bestCost > 0 ∧timeLimit not passed do
   9.        puzzle ← minConflictsWithTabuList(puzzle)
   10.
   11.       cost ← evaluate(puzzle)
   12.
   13.       if bestCost > cost then
   14.              bestCost ← cost
   15.              bestPuzzle ← 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 ← resetFactor· α
   29.       end if
   30. end while
  Output: best puzzle
```

Algorithm 2. Iterated local search for Sudoku.

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, generating puzzle instances with 0 percent fixed cells, 5 percent fixed cells, 10 percent fixed cells, and so on. To provide a large number of problems, 20 instances were created per category, totaling 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 1,200 instances for our experiments. Because 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: the other considered algorithms didn't produce competitive results, even for instances with an order of four. All tests were run on an Intel Xeon E5345 2.33 GHz with 48 Gbytes RAM. The instances used in this article as well as the sources of our implementation can be found at www.dbai.tuwien.ac.at/research/project/arte/sudoku/.

We used two metrics to compare algorithms: the average solving time and the success rate for each puzzle category. We followed Lewis's approach[7] to determine the required values, with the success rate representing 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 that were able to find an optimal solution were considered. The time limits differ depending on the order of the given puzzle instance. We restricted the runtime of

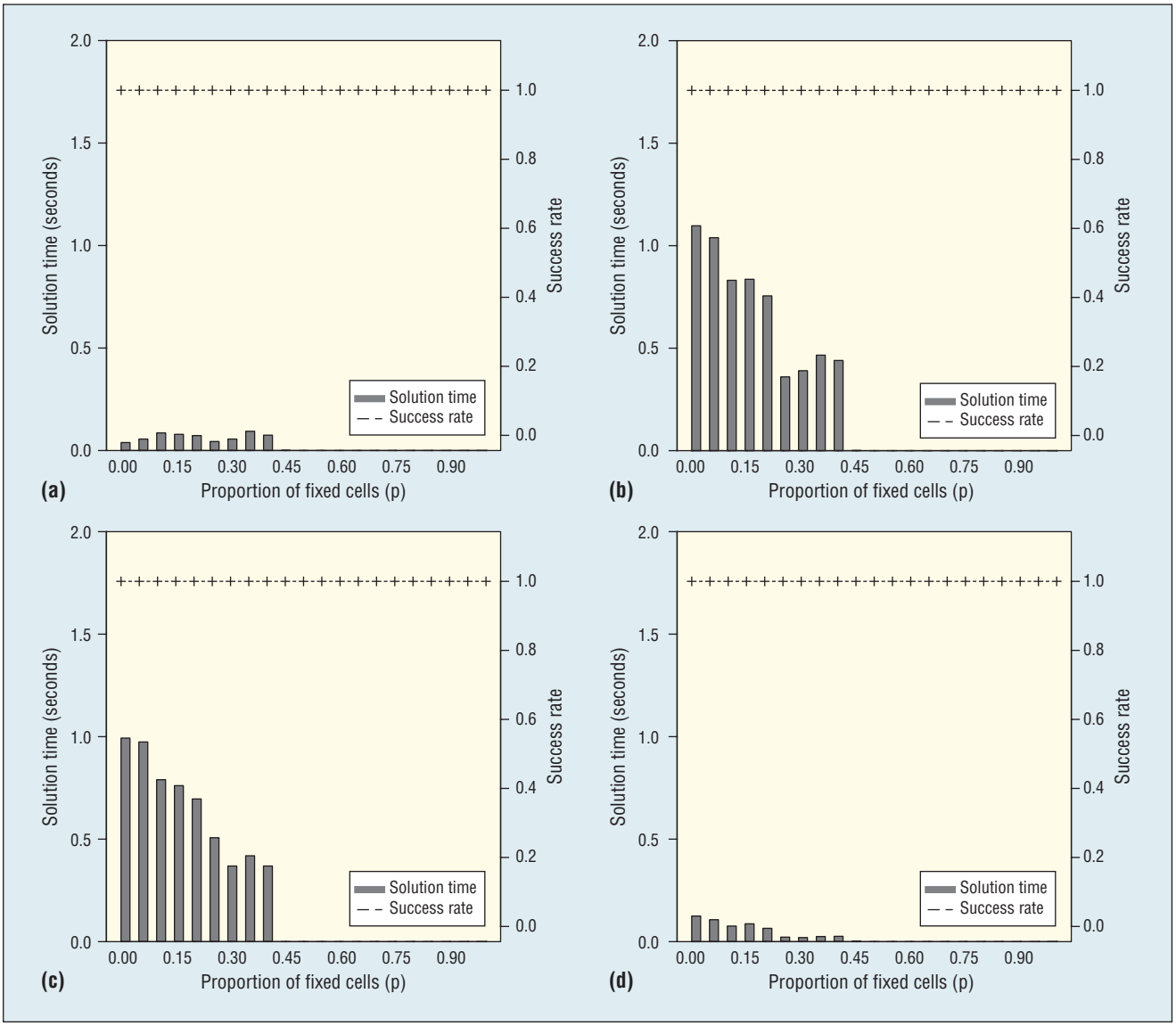

Figure 2. Results for Sudoku puzzles of order three: outcomes for (a) the algorithm based on our solver, (b) a simulated annealing-based algorithm,[8] and (c) and (d) an alternative approach[9] and a CP-based algorithm.[3]

our experiments to 5 seconds for a Sudoku of order three, 30 seconds for order four, and 350 seconds for order five. Time limits were chosen based on Lewis's experiments.[7]

Algorithm Configuration

Parameters for the considered algorithms from the literature[3,8,9] were configured as described in the corresponding papers. To configure our algorithm, we ran experiments with different values on some of the hardest puzzles from the benchmark instances. There's an "easy-hard-easy" phase transition depending on the relative number of prefilled cells, so we focused on the hardest problems (which have between 40 and 45 percent of the cells fixed initially) when experimenting with different parameter settings.

Because 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 by 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 25×25 grid, the *iterationLimit* was set to $625 \times 20 = 12{,}500$ in our experiments. We set the *forwardCheckingTimeLimit* parameter to a maximum of 5 seconds so that the algorithm wouldn't spend too much time in the perturbation phase.

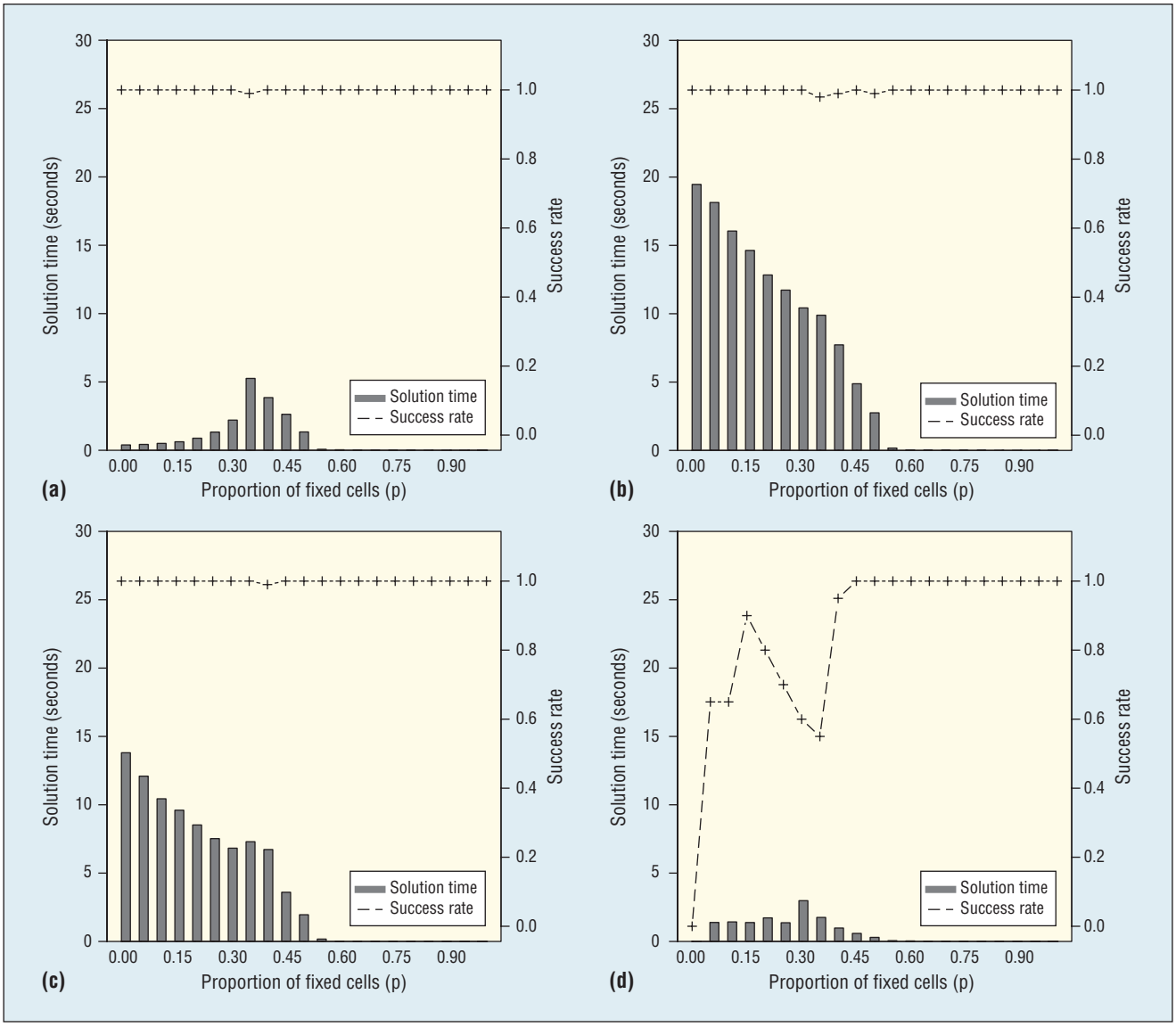

Figure 3. Results for Sudoku puzzles of order four: outcomes for (a) the algorithm based on our solver, (b) a simulated annealing-based algorithm,[8] and (c) and (d) an alternative approach[9] and a CP-based algorfithm.[3]

The initial value for the *resetFactor* was set to 1.0, so all the unfixed cells would be removed in the first perturbation phase and the algorithm would get a chance to solve the puzzle solely by FC. In later iterations, this value could be stepwise reduced so that only conflicting cells would be perturbed. This restricts FC search to smaller areas of the search space in later perturbation phases.

To determine good values for the *tabuListSize*, the *acceptanceProbability*, and the α parameter, we applied automatic parameter configuration using the irace-package.[12] We kept all the irace default settings and limited the tuning budget (maximum number of runs) for the algorithm to 1,000. Twenty of the puzzles that have an order of 5 and 40 percent 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 percent, and a value for α of around 0.5. A run with the parameters determined by irace yielded good results, but we were able to achieve additional improvements by some manual tuning trials with these parameters on the tuning instances. Via further manual tuning, we found that a *tabuListSize* of 0.05, an *acceptanceProbability* of 15 percent, and an α of about 0.8 produced even better results, so we used those parameter values in our final experiments.

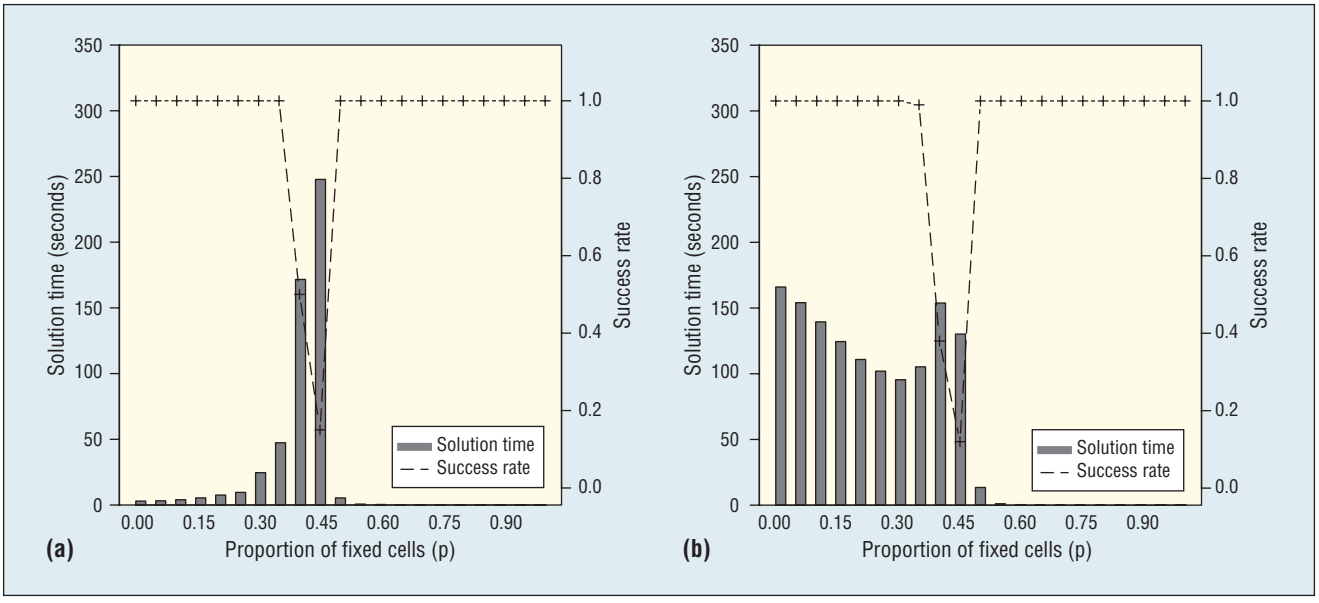

Figure 4. Results for Sudoku puzzles of order five: outcomes for (a) the algorithm based on our solver and (b) a simulated annealing-based algorithm.[8]

Results

We conducted experiments for puzzles of order three and four using four different algorithms: a simulated annealing-based approach[8] and a variation,[9] a CP-based solver,[3] and our algorithm. For Sudoku instances with an order of five, experiments were only conducted for our solver and the simulated annealing-based algorithm,[8] which produced better results compared to the two other algorithms from the literature for Sudoku with an order of four.

Figures 2, 3, and 4 give 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 Broderick Crawford and colleagues[3] produces very good results when it comes to solving 9×9 Sudoku puzzles. However, as soon as the search space gets larger, the CP-based algorithm can't compete with the other approaches: the large drop in the success rate for Sudoku puzzles with an order of four shows that for larger problems, the metaheuristic approaches using simulated annealing combined with constraint propagation techniques deliver better results. This is clearer in Figures 3b and 3c, which visualize results from the algorithms by Lewis[7] and Marlos C. Machado.[9] When comparing these two approaches, Lewis's implementation slightly outperforms the other algorithm in terms of success rate, so we only compare our approach to the former when considering benchmark instances with an order of five.

The hybrid algorithm presented here provided very good results in all 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. As we can see, for the hardest instances ($p = 0.4$ and $p = 0.45$), Lewis's success rates[8] are 38 and 12 percent, whereas our algorithm has success rates of 57 and 13 percent.

Application of Our Method in Scheduling

To the best of our knowledge, the hybridization of min-conflicts with iterated local search using a CP-based perturbation hasn't been considered before, but as we've shown, this hybrid algorithm gives very good results for the Sudoku problem. To investigate the generalizability of the proposed approach, we applied our method on a practical problem from the employee scheduling area.[13,14]

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 can either work in a single shift or have a day off on each day in a given scheduling period spanning multiple weeks. The employees and shift types considered in this problem are specified by a list of unique names connected with 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

Table 1. The results of different approaches to the employee scheduling problem.*

	Iterated local search		Iterated local search & constraint programming		Iterated local search & constraint programming*		Ejection chain[13]	
Instance	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	1,001	1,003	1,001	1,001	1,001	1,001	1,003	1,003
Instance 4	1,721	1,718	1,722	1,717	1,716	1,716	1,719	1,718
Instance 5	1,244	1,237	1,237	1,235	1,150	1,147	1,439	1,358
Instance 6	2,254	2,159	2,245	2,165	2,145	2,050	2,344	2,258
Instance 7	1,176	1,178	1,078	1,072	1,090	1,084	1,284	1,269
Instance 8	-	1,886	1,549	1,446	1,548	1,464	2,529	2,260
Instance 9	466	475	455	455	454	454	474	463
Instance 10	4,960	4,875	4,769	4,750	4,660	4,667	4,999	4,797
Instance 11	3,578	3,494	3,459	3,462	3,470	3,457	3,967	3,661
Instance 12	4,538	4,768	4,629	4,216	4,338	4,308	5,611	5,211
Instance 13	3,568	2,801	3,461	2,767	3,157	2,961	8,707	3,037
Instance 14	-	-	1,668	1,512	1,430	1,432	2,542	1,847
Instance 15	-	-	4,861	4,737	4,871	4,570	6,049	5,935
Instance 16	4,057	-	3,869	3,636	3,754	3,748	4,343	4,048
Instance 17	6,902	6,916	7,035	6,606	6,720	6,609	7,835	7,835
Instance 18	5,525	5,509	5,944	5,604	5,400	5,416	6,404	6,404
Instance 19	6,654	4,748	6,551	4,573	4,780	4,364	6,522	55,31
Instance 20	-	-	-	-	8,763	6,654	23,531	97,50
Instance 21	-	82,541	-	-	33,163	22,549	38,294	36,688
Instance 22	-	-	-	-	192,946	48,382	-	516,686
Instance 23	488,156	320,788	480,064	321,094	189,850	38,337	-	54,384
Instance 24	1,208,465	940,803	1,202,862	942,501	519,173	177,037	-	156,858

Columns 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 specifies hard and soft constraints to set up a corresponding rule set: hard constraints are always strict and have to be fulfilled to generate a feasible solution, and soft constraints can be violated (although violations can lead to an integer valued penalty). For example, a hard constraint could specify the minimum and maximum amount of time an employee can work during the entire scheduling horizon, and personal shift requests of employees can be formulated as soft constraints. 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 in which the optimal solution is a feasible schedule with the lowest possible objective value.[13]

To use our approach on this problem, we applied three different search neighborhoods that have been proposed elsewhere.[14] These 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 cause constraint violations. Additionally, we devised a CP approach that uses an FC 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 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 cause 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 1

THE AUTHORS

Nysret Musliu is a senior scientist in the Institute of Information Systems at the 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. Musliu has a PhD in computer science from the Vienna University of Technology. Contact him at musliu@dbai.tuwien.ac.at.

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

displays the best results and compares them with results obtained by a state-of-the-art heuristic based on ejection chains.[13]

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-art algorithm, is 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 1).

To the best of our knowledge, our solver 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. For future work, we'll consider the application of min-conflicts-based search that exploits constraint programming as a perturbation technique for other optimization problems that arise in various areas such as planning and scheduling. ■

Acknowledgments

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 Trans. Fundamentals of Electronics, Communications and Computer Sciences*, vol. 86, no. 5, 2003, pp. 1052–1060.
2. H. Simonis, "Sudoku as a Constraint Problem," *Workshop Modeling and Reformulating Constraint Satisfaction Problems*, 2005, pp. 13–27.
3. B. Crawford et al., "Using Constraint Programming to Solve Sudoku Puzzles," *Proc. Int'l Conf. Convergence Information Technology*, vol. 2, 2008, pp. 926–931.
4. R. Soto et al., "A Hybrid AC3-Tabu Search Algorithm for Solving Sudoku Puzzles," *Expert Systems with Applications*, vol. 40, no. 15, 2013, pp. 5817–5821.
5. R. Soto et al., "A Hybrid alldifferent-Tabu Search Algorithm for Solving Sudoku Puzzles," *Computational Intelligence and Neuroscience*, 2015, article no. 286354.
6. I. Lynce and J. Ouaknine, "Sudoku as a SAT problem," *Int'l Symp. Artificial Intelligence and Mathematics*, 2006; http://anytime.cs.umass.edu/aimath06/proceedings/P34.pdf.
7. R. Lewis, "Metaheuristics Can Solve Sudoku Puzzles," *J. Heuristics*, vol. 13, no. 4, 2007, pp. 387–401.
8. R. Lewis, "On the Combination of Constraint Programming and Stochastic Search: The Sudoku Case," *Hybrid Metaheuristics*, LNCS 4771, Springer, 2007, pp. 96–107.
9. M. Machado and L. Chaimowicz, "Combining Metaheuristics and CSP Algorithms to Solve Sudoku," *Proc. Brazilian Symp. Games and Digital Entertainment*, 2011, pp. 124–131.
10. T. Stützle, "Lokale Suchverfahren für Constraint Satisfaction Probleme: Die Min Conflicts Heuristik und Tabu Search" (in German), *Künstliche Intelligenz*, vol. 11, no. 1, 1997, pp. 14–20.
11. T. Mantere and J. Koljonen, "Solving and Analyzing Sudokus with Cultural Algorithms," *IEEE Congress on Evolutionary Computation*, 2008, pp. 4053–4060.
12. M. López-Ibánez et al., *The irace Package, Iterated Race for Automatic Algorithm Configuration*, tech. report, TR/IRIDIA/2011-004IRIDIA, Univ. Libre de Bruxelles, 2011.
13. T. Curtois and R. Qu, *Computational Results on New Staff Scheduling Benchmark Instances*, tech. report, Univ. Nottingham, 2014.
14. E.K. Burke and T. Curtois, "New Approaches to Nurse Rostering Benchmark Instances," *European J. Operational Research*, vol. 237, no. 1, 2014, p. 7181.