Solving Sudoku Using Probabilistic Graphical Models

Sheehan Khan***, Shahab Jabbari*****, Shahin Jabbari*****, Majid Ghanbarinejad**†
*Department of Computing Science, †Department of Electrical and Computer Engineering
University of Alberta, Edmonton, Canada
{sheehank,jabbaria,sjabbari}@cs.ualberta.ca, madjid@ece.ulaberta.ca

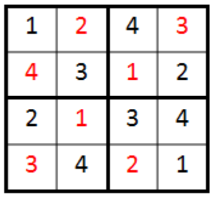
Abstract

Sudoku is a popular number puzzle. Here, we model the puzzle as a probabilistic graphical model and drive a modification to the well-known sum-product and max-product message passing to solve the puzzle. In addition, we propose a Sudoku solver utilizing a combination of message passing and Sinkhorn balancing and show that as Sudoku puzzles become larger, the impact of loopy propagation does not increase.

1 Introduction

1.1 The Problem

Sudoku is a popular puzzle game. An $N \times N$ Sudoku puzzle is a grid of cells partitioned into N smaller blocks of N elements. A solution to the puzzle involves filling in empty cells in the grid in a way that the numbers 1 through N appear once on each row, each column, and each $\sqrt{N} \times \sqrt{N}$ subgrid (*all-different* constraints). Figure 1 shows an example of a 4×4 Sudoku puzzle and its solution.



1	2	4	3
4	3	1	2
2	1	3	4
3	4	2	1

Figure 1: 4×4 Sudoku

Many games similar to Sudoku exist such as Sudoku X, Nonomino Sudoku, Killer Sudoku, Hyper Sudoku, Greater-Than Sudoku, Kakuro, Futoshiki, and KenKen. In Section 3 we will show that our results for Sudoku can also be applied to these games because of the similarity in their constraints.

The interest in treating Sudoku as a probabilistic graphical model (PGM) was largely motivated by an observation in the error-control coding community that, as a PGM, Sudoku takes the form of a low-density parity-check (LDPC) code. LDPC codes have shown great error-control performance, but they still suffer from issues arising from loopy belief propagation. Hence, it was thought that, by observing performance on Sudoku puzzles because of the unique solution insight could be gained on the LDPC decoding problem.

1.2 Related Works

Sudoku puzzles provide an interesting playground for mathematics. It is typically treated as an instance of the graph coloring problem [4], or the quasi-group with holes problem [2]. In a recent work it was shown that Sudoku is NP-complete [14]. Also, Sudoku can be viewed as an LDPC decoding problem over an erasure channel [9]. Other industrial applications that can be modeled by Sudoku are shown in [5, 11].

There have been many reported algorithms for solving Sudoku. The seemingly most efficient is a search based solver presented by [10]. Although the algorithm can easily solve puzzles up to size 16×16, it fails on larger puzzles. The main reason for this is an exponential growth in the search space as the puzzle size increases.

Under the context of LDPC codes [3, 6, 9, 12] present several solvers. In [3, 6, 9] standard belief propagation algorithms were utilized. [3, 6] show that there is no notable difference between sum-product and max-product message passing.

Outside of PGM other probabilistic solvers are presented in [9, 12, 8, 7]. In [9] the Sinkhorn balancing technique for doubly stochastic matrices was used. A bit flipping algorithm similar to WALK-SAT was shown in [12] under the context of LDPC. Standard genetic algorithms and simulated annealing techniques were presented in [8, 7].

Other than search which is garaunteed to eventually solve any Sudoku puzzle given sufficient time, no claims or gaurantees are made for these solvers on Sudoku puzzles larger than 9x9.

1.3 Sudoku as a Probabilistic Graphical Model

Sudoku can be viewed as a bipartite graph using the notations defined in [9]. A bipartite graph is defined as a graph whose vertices can be divided into two disjoint sets $\mathcal{S}$ and $\mathcal{C}$ such that the graph's edges connect vertices in $\mathcal{S}$ only to vertices in $\mathcal{C}$ and vice versa. In this way, all the cells in an $N \times N$ Sudoku puzzle can be mapped to $\mathcal{S}_n \in \mathcal{S}$ by assigning them labels from 1 through N^2 in a row-scan order.

We map all row, column, and subgrid constraints to another set of vertices in the graph, $\mathcal{C}_m \in \mathcal{C}, m \in 1 \cdots 3N$. Edges $\mathcal{E}_{nm}$ are added whenever cell n takes part in the m's constraint. In the rest of this paper, we refer to two sets $\mathcal{S}$ and $\mathcal{C}$ as variable nodes and constraint nodes respectively. In an $N \times N$ Sudoku, variable nodes take values $\mathcal{S}_n = \{1 \cdots N\}$ while constraint nodes take $\mathcal{C}_m = \{0, 1\}$ if the connected variable nodes satisfy the all-different constraint.

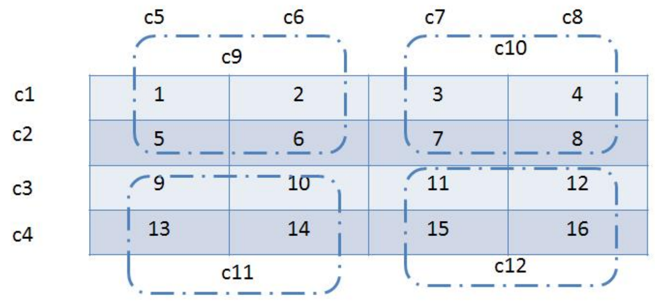


Figure 2: Constraints for a 4×4 Sudoku

Figure 2 illustrates how constraints are defined for a 4×4 Sudoku puzzle. Figure 3 shows the corresponded factor graph. Circles and squares represent variable nodes and constraint nodes respectively.

Each variable node stores a factor of the form

$$p_n = [P(\mathcal{S}_n = 1)\ P(\mathcal{S}_n = 2) \cdots P(\mathcal{S}_n = N)],$$

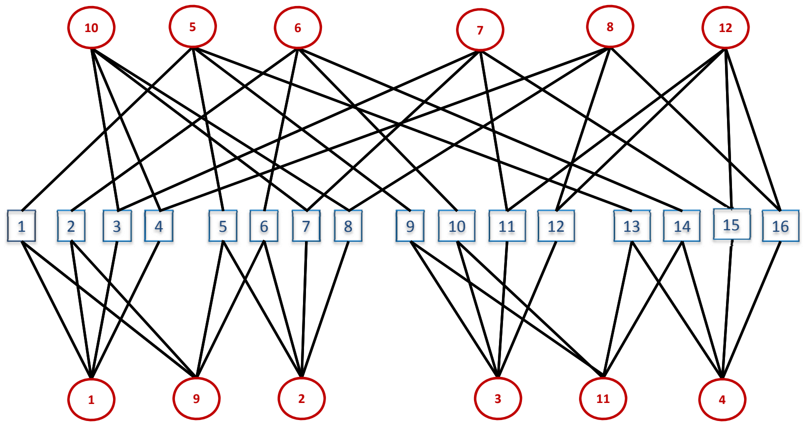


Figure 3: Factor graph associated with a 4×4 Sudoku

which is the probability vector associated with variable node $\mathcal{S}_n$. The vector of a *given* cell filled with value $k, k \in \{1, \cdots, N\}$ is initialized to have a 1 in the kth place and $N-1$ zeros in the other places. For other cells, we initialize the vector uniformly after eliminating the values violating the three constraints associated with that cell. For example, if we know that the first cell can not take value 1 in a 4×4 puzzle, we will initialize its vector as $p_1 = [0\ \frac{1}{3}\ \frac{1}{3}\ \frac{1}{3}]$.

We define the set of all variable nodes connected to the mth constraint node as $\mathcal{N}_m$, and the constraint nodes connected to the nth variable node as $\mathcal{M}_n$. For example, for the puzzle of Figure 2 we have

$$\mathcal{M}_1 = \{1, 5, 9\} \qquad \mathcal{N}_1 = \{1, 2, 3, 4\}.$$

Double script notation are used to indicate sets with removed elements. In this way, $\mathcal{N}_{m,n} = \mathcal{M}_n \backslash m$ denotes the cells involved in the mth constraint except for cell n. For example,

$$\mathcal{N}_{1,1} = \{2, 3, 4\}.$$

In the next two sections, we talk about message passing for Sudoku and difficulties of belief propagation in the loopy structure of Sudoku. The idea of incorporating belief propagation with some other algorithm is presented in Section 4. Section 5 discusses our idea of solution checking. We talk about our experimental results in Section 6. Finally, conclusions are presented in Section 7.

2 Message Passing for Sudoku

In the previous works utilizing PGMs for Sudoku [3, 6, 9], traditional sum-product message passing was used [1]. The following is the sum-product equations in the context of Sudoku with the notation conventions of [9].

$$r_{mn}(x) = \sum_{\substack{\{n' \in \mathcal{N}_{m,n}\} \\ n=x, n'=x_{n'} \text{ all unique}}} \prod_{l \in \mathcal{N}_{m,n}} q_{lm}(x_l) \tag{1}$$

$$q_n(x) = \mathrm{P}(n = x) \prod_{m \in \mathcal{M}_n} r_{mn}(x) \tag{2}$$

$$q_{n,m}(x) = \mathrm{P}(n = x) \prod_{m' \in \mathcal{M}_{n,m}} r_{mn}(x) \tag{3}$$

In the above equations, r_{mn} are the constraint-to-variable messages, $q_{n,m}$ are the variable-to-constraint messages, $\mathrm{P}(n = x)$ are the *a priori* probabilities, and $q_n(x)$ are the a posteriori beliefs. Also, in [3, 6], the max-product algorithm was applied. Max-product simply involves replacing the summation over the product terms in (1) by the maximum term. In other literature it has been shown that for cyclic graphs, in which sum-product suffers from loopy propagation, max-product is optimal over neighbourhoods[1] within the graph [13]. However, results presented in [6] show no discernible difference between sum-product and max-product which is most likely due to the small world topology of the PGM[2].

By examining (1) we can see that computation of both sum-product and max-product message vectors requires a lot of work. For each message value, we must try $N!$ assignments to the variable nodes in the constraint. Thus, to compute a message vector, it takes O($N!N^2$) time and there are $3N^2$ message vectors to compute. This means an iteration of constraint-to-variable messaging takes O($N!N^4$) time.

Here we show a simple alternative that drops the cost back to polynomial time. This is achieved by relaxing the all-different constraint. Consider the equation

$$\mathrm{P}(n = x|\text{constraint } m) \approx \mathrm{P}(n = x) \prod_{n' \in \mathcal{N}_{m,n}} (1 - \mathrm{P}(n' = x)) \tag{4}$$

which is clearly an approximation as it allows for invalid assignments to the other nodes in the constraint while computing the probability. Based on this approximation we modified our belief propagation algorithm to replace (1) with

$$r_{mn}(x) \quad = \prod_{n' \in \mathcal{N}_{m,n}} (1 - q_{n'm}(x)) \tag{5}$$

While we do not show that message passing in this form will actually compute our proposed posterior (4), in Section 6, we will show that we still obtain results comparable to traditional message passing. Note that after our modification, the time cost of a message passing iteration is reduced to O(N^4), which is a huge savings.

3 Loopy Belief Propagation

Loopy belief propagation occurs when message passing is applied in cyclic graphs. The problem is largely due to a double counting of information as we will show specifically in the context of Sudoku. Consider three nodes, A, B, and C that are related under one of the all-different constraints imposed by Sudoku. Node A will send a message to node B about the beliefs it has in taking different values. Node B then uses this information to update its beliefs about its own probabilities and sends a similar message to C. Similarly, C will use the message from B to send a message to A. When A receives this message it will consider it as new information and update itself. However, the message is not entirely new as it contains information that A previously has passed to B. By treating it as new information, A creates a bias in its beliefs. It is these bias terms and their propagation which cause the problems in loopy propagation.

Theorem 3.1. *Whenever a variable node is known* a priori *from the puzzle, there will be no loopy propagation on its associated loops.*

Proof. All belief propagation algorithms presented here share the same variable-to-constraint messages defined in (3). Upon inspection, it is clear that in the case of a node which is known *a priori*, the prior term P($n = x$) will be a Kronecker delta function. Since we normalize the message vectors for numerical stability, this will force all outgoing messages from the node to also be Kronecker delta functions, regardless of the inputs the node receives. Therefore, it is impossible for information sent by another node in the cycle to loop back to itself from within that cycle. □

From this theorem, we can make a second theorem about scaling to larger puzzles, which we then generalize to a larger class of CSP problems. First, we present some lemmas about scaling that will be required for the proof.

[1]Where a neighbourhood is defined as a connected component within the graph with at most one cycle.

[2]After 2 iterations of message passing every node will have received information about every other node in the graph.

Lemma 3.2. *The number of k-cycles in the graph grows proportional to O(N^k).*

Proof. The proof proceeds by providing upper and lower bounds with the same growth rate. Clearly if we select an arbitrary constraint, we can create k-cycles by selecting $k-1$ nodes from within it. There are $3N$ constraints and $\binom{N}{k-1}$ ways to select nodes combining to give O(N^k). This is a lower bound as we have only considered cycles within the same constraint. To get an upper bound, we use the following method to create a path of $k-1$ nodes. Pick a random node and then proceed to select $k-2$ nodes such that each node is connected to the previous node. The first choice can be made in N^2 ways. Assuming that we use a unique constraint each time we select the following node, there is $N-1$ ways to select each of the subsequent nodes. Thus, there are a total of O(N^k) such paths. Clearly, this mechanism will generate all the k-cycles we are interested in, plus many paths which we are not, and thus it is an upper bound. Since both upper and lower bounds have the same growth rate, the proof is complete. □

Lemma 3.3. *The number of k-cycles that contain a given node grows at least as fast as O(N^{k-2}).*

Proof. Similar to the last proof, we can get a quick lower bound by considering only k-cycles that occur within a constraint. After fixing the given nodes as a member of the cycle, there are then $\binom{N-1}{k-2}$ ways to complete it. Thus, there are at least O(N^{k-2}) k-cycles on a given node. □

Lemma 3.4. *A lower bound on the growth rate for the minimal number of given cells required to force a unique solution is O(N^2).*

Proof. The number of variable nodes in Sudoku scales by N^2, but the constraints scale by $3N$. Thus, to enforce a unique solution as the dimension increases, a larger percent of the variable nodes will need to be given. We can then say that the number of givens is equal to the product of some percentage function $p(N)$ and the number of variable nodes N^2 and thus the growth rate is O($p(N)N^2$) $>$ O(N^2). Where we get strict inequality from the fact that $p(N)$ must be monotonically increasing. □

Conjecture 3.5. *The growth rate of the number of k-cycles that contain a given is loosely lower bounded by O(N^k).*

Support. In order to force a unique solution, the given cells in a minimal Sudoku are sparsely distributed. Thus, it is likely there will be little overlap in the loops associated with the given cells so that we can multiply the cycles per node from Lemma 3.3 by the number of given cells from Lemma 3.4 to get the total. We claim this as a loose bound due to the strict inequality in Lemma 3.4. □

Theorem 3.6. *Scaling the dimension of Sudoku does not increase the impact of loopy belief propagation.*

Proof. Since the growth rate from Conjecture 3.5 is the same as the rate from Lemma 3.2, it is clear that the number of cycles without a given cell must not be increasing. Which in turn reduces the percentage of nodes impacted by loopy propagation. □

Based on Theorem 3.6 it is clear that on large enough puzzles that belief propagation will solve Sudoku puzzles. This leads us to a more generalized version of the theorem.

Corollary 3.7. *Belief propagation can be used to solve CSP problems on asymptotically large graphs given that:*

1. *There is a unique solution.*
2. *The variables are discrete.*
3. *The constraints applied are of a combinatorial nature. i.e. "all-different" , "exactly k of ...", and "no more than k ... " etc.*
4. *The number of variables grows faster than the number of constraints.*

Proof. The proof of Theorem 3.6 utilized the regular structure of the graph and the growth rates to force a given variable into all problematic loops such that Theorem 3.1 could be applied. It is possible to visualize the general graphs allowed here as induced subgraphs of a larger regular graph. By conditions 2 and 3, we ensure that Theorem 3.1 holds. Condition 4 is used to force a given variable into the problematic cycles as before. As it is unclear how to properly interpret beliefs if there are multiple solutions condition 1 is required. ☐

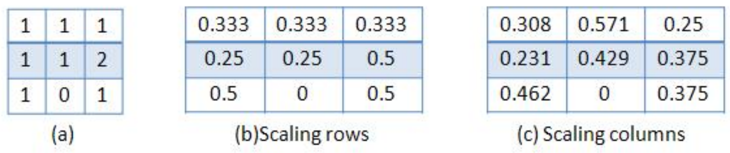
Theorem 3.8. *Belief propagation will detect most, but not all, problems in which based on the given cells no satisfiable solution exists.*

Proof. From examining equations (1,2,3), it is clear that sum/max-product message passing will force the *a posteriori* probabilities of assignments to be zero if they are inconsistent within a constraint. Thus, if all beliefs are forced to zero, then the algorithm can detect that there must have been a contradiction in the given variables. This becomes easier to detect as the problems becomes more constrained. ☐

4 Sinkhorn Balancing

Sinkhorn balancing is the idea of converting an arbitrary matrix to the space of a doubly stochastic matrix. A doubly stochastic matrix is a matrix with nonnegative elements in which the sum of each row and column is equal to 1.

Sinkhorn balancing first normalizes each column in a way that it sums to one. Then, it applies the same normalization to rows. Finally, it repeats this procedure until either the norm of the matrix did not change more than some predefined threshold in two consecutive steps or it reaches maximum number of iterations. Figure 4 shows one iteration of Sinkhorn balancing on a 3×3 matrix.



1	1	1
1	1	2
1	0	1

(a)

0.333	0.333	0.333
0.25	0.25	0.5
0.5	0	0.5

(b)Scaling rows

0.308	0.571	0.25
0.231	0.429	0.375
0.462	0	0.375

(c) Scaling columns

Figure 4: One iteration of Sinkhorn balancing

Sinkhorn balancing can be considered as grouping belief vectors within a constraint. As we stated before, there are N variable nodes in the factor graph connected to each constraint node, each of them having a probability vector of size N that must sum to 1. Considering all such vectors for a constraint as a matrix, the columns correspond to the probabilities of assigning values to a specific node that must also sum to 1.

Moon *et al.* tried to solve Sudoku by only using Sinkhorn balancing [9]. They proved that the KL-distance of the result generated by Sinkhorn balancing to the solution is a monotonically non-increasing function [9]. However, this convergence may be reached in a very large number of iterations.

5 Solution Check

Doing message passing for more iterations is not always helpful in solving Sudoku because we may overfit the problem[3] or lose information due to the biases introduced by the large number of loops. Also, each round of message passing takes $O(N^4)$ time. Based on this intuition, we decide to stop message passing after a certain number of iterations and see whether we can guess the values of cells for which we do not have a deterministic value yet.

One naive way of doing this is to assign the value with the highest probability in the probability vector to each unassigned cell. The main problem with this approach is that there may be more than one cell in a constraint that have the largest probability in their probability vector for a special value. This causes conflict in the puzzle.

[3]By overfit we mean that we have sufficient information to solve the puzzle but spend more time iterating until message passing and Sinkhorn converge.

In our algorithm, which is called solution check, first for each value of every unassigned cell a relative probability is calculated. The relative probability is the probability that a cell takes a certain value, divided by the sum of the probabilities that all the cells in the same constraint take that value. The relative probability is a measure of entropy. The algorithm selects the cell having the the maximum relative probability of a value to take that value. After this assignment, the algorithm deletes that value from all the other cells in the constraint. Then, the algorithm redistributes the deleted probability in those cells. It divides the deleted probability equally into other non-zero probability values. The algorithm repeats the above steps until it assigns a value to all cells.

The output of solution-check is checked. If it is a valid Sudoku solution, the program has successfully solved the puzzle. Otherwise, the function terminates with failure for that puzzle.

6 Experimental Results

In order to verify our theories, we implemented our work in Matlab. We developed an algorithm doing message passing based on (5) in every iteration. Furthermore, we incorporated Sinkhorn balancing every two iterations and checked the generated result with the proposed solution-check method. Our motivation for using Sinkhorn balancing was the idea of simulated annealing. We applied it, with small number of iterations, after every two iterations, because every node will have received information about every other node in the graph after this these iterations. Furthermore: 1) it is an entire different method of dispersing the probability mass, 2) it does not suffer from loops. In this way, it is unlikely that both Sinkhorn balancing and belief propagation get trapped by the same incorrect answer.

Figure 5 shows the results of our experiments on 2356 9×9 puzzle and 44 16×16 puzzles. The puzzles are provided by the Magic Tour website [4]. Note that the performance is examined only in the region that Sudoku puzzles with unique solutions exist [5]. Furthermore, Table 1 shows the

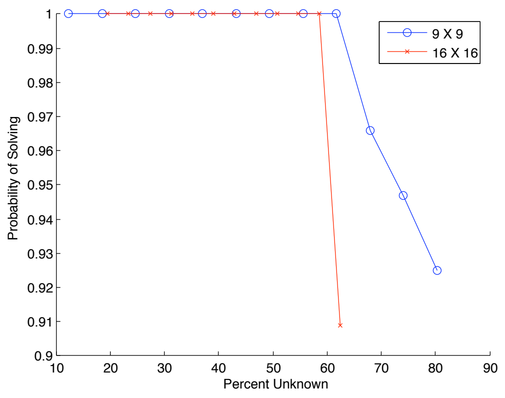


Figure 5: Experimental Results for 9×9 and 16×16 using 250 and 1000 iterations respectively.

comparison of our approach with other approaches presented in related works. As you can see in Figure 5 and also Table 1 the performance of our solver for 16×16 puzzles is nearly the same as the performance for 9×9 puzzles[6], which supports Theorem 3.6.

7 Conclusion

This paper presented Sudoku as a probabilistic graphical model. We drove modifications in the well-known sum-product and max-product message passing that scale well for Sudoku puzzles.

[4]`http://magictour.free.fr/Sudoku.htm`

[5]There is no known unique-solution puzzle with less number of given cells.

[6]The slight difference between the performances is due to the fact that we only have 44 minimal 16×16 puzzles!

Approach	Accuracy for 9×9	Accuracy for 16×16
Sum-Product	53.2%	NA
Max-Product [3]	70.6%	NA
Sum-Product	71.3%	NA
Max-Product	70.7 - 85.6%	NA
Combination [6]	76.8 - 89.5%	NA
Our Approach		
40 Iterations	70%	5%
Many Iterations	95%	90.9%

Table 1: Summary of results for discussed works applied to minimal Sudokus. As it was unclear how to properly scale we capped 9x9 at 200 iterations and 16x16 at 1000 iterations and report the results as *Many Iterations*.

The modification did not impact the ability of the solver to solve the problem, but it provided a large saving in time to solve the puzzle. While it was known that the main deficiency of message passing when applied to Sudoku is due to loopy propagation, we showed that as Sudoku puzzles become larger, the impact of loopy propagation does not increase. Empirical results were provided to support this theory. The results were based on a heuristic algorithm combined of message passing and Sinkhorn balancing.

Acknowledgments

We would like to thank Russell Greiner for insightful discussions and useful suggestions. We also thank Todd K. Moon, Daphne Koller, and Nir Friedman for providing us with advanced copies of their works.

References

[1] *Structured Probabilistic Models: Principles and Techniques*. Daphne Koller and Nir Friedman.

[2] Achlioptas, D., Gomes, C., Kautz, H., and Selman, B. Generating satisfiable instances. *In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)* (2000).

[3] Bauko, H. Passing messages to lonely numbers. *Computing in Science and Engineering 10*, 2 (2008), 32 – 40.

[4] Bondy, J., and Murty, U. *Graph Theory With Applications*. North-Holland, 1982.

[5] Dincbas, M., and Simonis, H. Apache - a constraint based, automated stand allone allocation system. *Advanced Software Technology in Air Transport (ASTAIR'91)* (1991), 267 – 282.

[6] Goldberger, J. Solving sudoku using combined message passing algorithms. *Technical Reort TR-BIU-ENG-2007-05-03,Engineering School, Bar-Ilan Univ.* (2007).

[7] Gottlieb, M. A. The probabilistic sudoku solver (pss). *http://www.feynmanlectures.info/sudoku/pss.html*.

[8] Mantere, T., and Koljonen, J. Solving, rating and generating sudoku puzzles with ga. *EEE Congress on Evolutionary Computation (CEC'07)* (2007), 1382 – 1389.

[9] Moon, T. K., and Gunther, J. H. Multiple constraint satisfaction by belief propagation: An example using sudoku. *Adaptive and Learning Systems, 2006 IEEE Mountain Workshop on* (2006), 122 – 126.

[10] Norvig, P. Solving every sudoku puzzle. *http://norvig.com/sudoku.html*.

[11] Simonis, H. Building industrial applications with constraint programming. *Constraints in computational logics: theory and applications* (2001), 271 – 309.

[12] Soedarmadji, E., and McEliece, R. J. Iterative decoding for sudoku and latin square codes. *Allerton Conference* (2007).

[13] Weiss, Y., and Freeman, W. T. On the optimality of solutions of the max-productbelief-propagation algorithm in arbitrary graphs. *Information Theory, IEEE Transactions on 47*, 2 (Feb 2001), 736 – 744.

[14] YATO, T., and SETA, T. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans Fundam Electron Commun Comput Sci E86-A*, 5 (2003), 1052 – 1060.