Polynomial Algorithm for Solving Cross-matching Puzzles

Josef Hynek(✉)

Faculty of Informatics and Management, University of Hradec Kralove, Rokitanskeho 62, 500 03 Hradec Králové, Czech Republic
josef.hynek@uhk.cz

Abstract. The aim of this paper it to analyze the cross-matching puzzle and to propose a fast and deterministic algorithm that can solve it. Nevertheless, there is a bigger goal than designing an algorithm for a particular problem. We want to show that while AI researchers constantly look for new constraint-satisfaction problems that could be utilized for testing various problem-solving techniques it is possible to come up with the problem that can be solved by much simpler algorithms. We would like to stress that there is an important misconception related to NP class that a huge number of potential solutions to the specific problem almost automatically implies that the relevant problem belongs to the class of NP. Such a misunderstanding and misclassification of the particular problem leads to false impression that there is no chance to design a simple and fast algorithm for the problem. Therefore, various heuristics or general problem-solving techniques are unnecessarily employed in order to solve it. And moreover, the wrong impression that the problem is difficult is further supported. We believe that our paper can help to raise the awareness that not all the problems with immense search spaces are hard to be solved and the polynomial algorithm to tackle the cross-matching puzzle that is described here is a good example of such an approach.

Keywords: Cross-matching puzzle · Efficient algorithm · Time complexity

1 Introduction

It is a very common approach that various games and puzzles are utilized in order to demonstrate the power of specific problem-solving techniques. There is a great advantage hidden in the fact that simple rules that can be easily understood could generate a huge space of potential candidate solution. There are many traditional types of puzzle like, for example, 15-puzzle (or Loyd's puzzle), Sudoku or jigsaw puzzles, while tic-tac-toe, Othello (reversi), checkers, chess or some specific tasks involving individual pieces of chess (like knights or queens) are often used to present specific problem-solving techniques or algorithms. For example, the classic 9×9 Sudoku is well-known and extremely popular amongst general public. The general problem of solving $N \times N$ Sudoku has been proved to be NP-complete [1] while there are really fast algorithms solving Sudoku of small sizes including 9×9 grid (see, for example, [2]). Furthermore,

L. Rutkowski et al. (Eds.): ICAISC 2021, LNAI 12855, pp. 257–266, 2021.
https://doi.org/10.1007/978-3-030-87897-9_24

various of these puzzles and games are also very often used by teachers and lectures in programming courses as it is quite easy to define the problem precisely and then to show how to tackle it algorithmically (see, for example, [3]).

On the other hand, AI researchers have always looked for new challenges and new constraint-satisfaction or constraint-optimization problems that could be utilized as testbeds for various approaches and techniques. If there is not at hand some real and practical problem to be solved, it is a nice challenge to design an artificial problem and then show the way how to solve it. On one side this approach is understandable as there are, for example, too many papers devoted to jigsaw puzzles or the safe placement of N-queens on the chessboard, while the newly designed problem might look not too common, more attractive and it could also possess some specific features that make the search for the solution somehow different or even more difficult. On the other hand, the design of new artificial problems brings along numerous questions or even risks. First of all, the new problem could be exactly the same one as the already known and well described but this time it is only defined in a different way. Secondly, while the classic problems are well-known and correctly classified as belonging to the class of NP problems, the new problem can be easily misunderstood and misclassified as being more difficult that it actually is. Finally, if some general problem-solving technique is unnecessarily employed in order to solve it, it is then clearly a worthless waste of computational resources while, at exactly the same time, the wrong impression that the problem is difficult is further supported.

2 Problem Description

We have decided to illustrate the problem on a simple cross-matching puzzle that was described by Kesemen and Özkul in [4] and the again in [5]. Their cross-matching puzzles consist of three tables with the size of M × N. For the sake of simplicity, we will consider squared tables of the size N × N in this paper but the algorithm presented below works with the size M × N as well.

The cross-matching puzzle is represented by three tables that are shown in Fig. 1. The table in the center is the solution table whose content is to be found. The table to the right is the detection table and the control table is located below. We adopted the same terminology as is used in [5]. The detection and control table represent the constraints under which the solution is to be sought.

The easiest way to describe the cross-matching puzzle is to show how to create it. At the beginning of this process, the solution table is filled by randomly generated symbols (letters or numbers). Then the symbols of the *i*-th row of the selection table are sorted and put into *i*-th row of the detection table.

We can see in Fig. 2a that the first row of the solution table containing the letters {D,M,I,B,E} was transformed into the sorted set {B,D,E,I,M} that is placed in the first row of the detection table. The same principle is applied to the creation of the control table and that is why the first column of the solution table comprising symbols {D,E,L,F,N} was converted into {D,E,F,L,N} in the first column of the control table. As soon as the detection as well as the control table are filled in, all the symbols in the solution table are erased and the puzzle is ready to be solved (Fig. 2b). In order to not confuse the reader,

we have utilised exactly the same example (assignment of letters) as it was presented in [5] and this puzzle will be used throughout this paper to show how to solve it by the algorithm we are going to propose here.

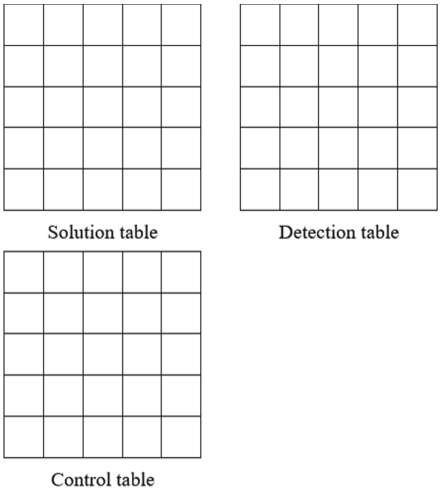

Fig. 1. The cross-matching puzzle tables.

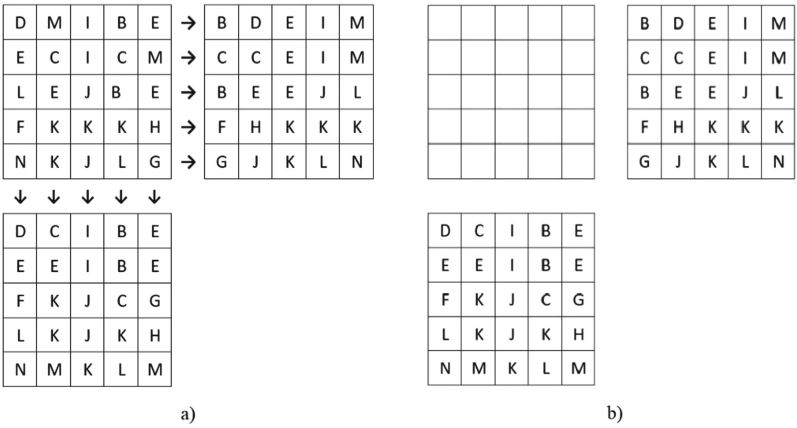

Fig. 2. The process of cross-matching puzzle generation – a) formation of the detection and control table, b) created cross-matching puzzle.

The aim of the puzzle is to re-construct the original solution table by using the clues given by the content of the detection and control tables. Should S be a N × N matrix representing the solution table, it is obvious that the sought symbol S_{ij} has to be present in the i-th row of the detection table and the j-th column of the control table. More specifically, if we convert the symbols of the i-th row of the detection table into a set of symbols and label it as D_i and if we do the same with the j-th column of the control

table and label it as C_j, it is clear that the symbol we are looking for must lie in the intersection of these two sets. Formally written:

$$S_{ij} \in D_i \cap C_j \tag{1}$$

Based on this formula, it is evident that if the puzzle consists of N^2 symbols that are different from each other, the cardinality of the intersection (1) is always equal to one and the solution of the puzzle is absolutely straightforward. Should there be a repeated occurrence of some symbols there, the cardinality of the intersection (1) is greater than one and there is more than one candidate symbol for the placement into the given position. Then we have to wait for the other positions to be filled in order to get a clue which symbol should be selected there.

Finally, the random generation of symbols during the process of the puzzle creation does not guarantee that there is a unique solution of the puzzle. There must be at least one but depending on the frequency of the letters and their specific position there might very easily exist multiple solutions. We will address this issue later when discussing the functioning and performance of our algorithm.

Of course, the puzzle can be solved using backtracking when the positions of the solution table are consecutively assigned starting from right-top corner and trying all the possibilities from the appropriate row in the detection table while checking the constraints given by the relevant column in the control table. This algorithm provides the exact solution but its time complexity is $O(N^N)$. Therefore, due to the combinatorial explosion it can be used for small instances of the cross-matching puzzles only. Kesemen and Özkul in [5] accepted this fact as an argument that the solution of the puzzle belongs to NP-class and hence a genetic algorithm (or another stochastic search method) is needed to tackle it. We are not going to give more details on their approach using multi-layer genetic algorithm here as the details can be found in their paper and we will directly skip to their so called intelligent genetic algorithm [5].

Their main improvement there is based on their observation that it is possible to fix some elements of the solution table because in some cases the intersection of the relevant row and the column provides only one symbol to be placed there. Using this simple idea they were able to generate partial solutions to the cross-matching puzzle where some positions were fixed (the wanted symbol has been found) and the other positions were assigned randomly using the remaining available symbols. This approach has been utilized to generate the initial population of individuals and thus all of these individuals presented partial solutions where the "already known" positions were fixed.

The only remaining concern for Kesemen and Özkul [5] was to make sure that the genetic operators employed there (crossover and mutation) would not damage the already fixed parts of the partial solution represented by chromosomes. They managed to solve this obstacle easily and as they significantly reduced the size of the search space (because of the already fixed positions) their intelligent genetic algorithm works rather nicely. However, they still reported that the algorithm needed nearly 6 s to solve 10×10 cross-matching puzzle and larger instances were not attempted.

Their paper raised our curiosity and the certain similarity of cross-matching puzzles with Sudoku inspired us to analyze the problem in order to devise a heuristic algorithm that would be capable of solving it quickly. We have realized that such an algorithm exists, it is really fast and very simple.

3 Proposed Solution

It is a well-known fact that a huge number of potential solutions to the specific problem does not automatically mean that the relevant problem is hard to be solved and that it belongs to the class of NP [6]. For example, the problem of computing the shortest path between two vertices in a complete graph with positive edge weights can be solved in polynomial time despite the fact that there are exponentially many possible paths between two vertices in a complete graph. The same applies to the minimal spanning tree problem and many others. The trick is that there is no need to asses all potential candidate solutions. Utilizing the specific features of the problem we can design an efficient algorithm that finds the optimal solution without having to traverse the whole search space. Therefore, if the representation of the candidate solutions is wrongly designed causing that the search space is even bigger than it is necessary (as it was discussed, for example, in [7]) and then a brute-force approach to check the whole search space is employed, it cannot be taken as a proof that the problem is impossible to be solved efficiently by a polynomial algorithm.

It was exactly the argument concerning the huge search-space of the cross-matching puzzle that attracted our interest. Moreover, as Kesemen and Özkul [5] realized, the search-space could be rather easily narrowed by the fixation of the symbols that were unique for the particular position within the solution table. Therefore, the first step of the algorithm we would like to present here is also the calculation of the intersections between the relevant row and the relevant column using the formula (1) above. Taking step by step the symbols from the row of the detection table and browsing for them within the control table, we will reach the stage depicted in Fig. 3.

Using the example presented above we can see that there are fifteen positions in the solution table where the cardinality of the just executed intersection is one and these cells are ready to be fixed. Nevertheless, we can see that there are several more places in the table, where the cardinality of the intersection is higher than one but some symbols appear repeatedly there.

For example, the content of the cell on the second row and the second column is {C,C,M} which can be simplified to {C,M}, because only these two symbols are eligible to stand here. Similarly, the content of the cell on the fourth row and the second column is {K,K,K} which without any doubt can be simplified to {K} only. This repeated occurrence of some symbols is due to the procedure that was used to perform the intersection operation and we can either tailor it or simply check the output for uniqueness of the symbols contained within each cell. Then we reach the situation depicted in Fig. 4. There are N^2 cells, the intersection can be done in $O(N^2)$, the uniqueness of each cell in $O(N^2)$ and therefore the overall time of reaching the initial stage of the algorithm shown in Fig. 4 is $O(N^4)$.

{D,E}	{E,M}	{I}	{B}	{E,M}
{E}	{C,C,M}	{I}	{C,C}	{M}
{L}	{E,E}	{J}	{B}	{E,E}
{F}	{K,K,K}	{K,K,K}	{K,K,K}	{H}
{N}	{K}	{J}	{L}	{G}

B	D	E	I	M
C	C	E	I	M
B	E	E	J	L
F	H	K	K	K
G	J	K	L	N

D	C	I	B	E
E	E	I	B	E
F	K	J	C	G
L	K	J	K	H
N	M	K	L	M

Fig. 3. The puzzle after the application of the intersection operator.

Now it is the right time to fix all the positions where the set containing only one candidate symbol exists. Whenever a symbol is fixed in cell S_{ij} we have to delete this symbol from the i-th row in the detection table as well as from the j-th column in the control table. Keeping in our mind that some symbols occur there multiple times it is necessary to make sure that only one symbol is deleted from the respective row and column each time. There are N^2 cells and therefore the fixation including the removal of the fixed symbol from the row and the column can be done in $O(N^3)$.

{D,E}	{E,M}	{I}	{B}	{E,M}
{E}	{C,M}	{I}	{C}	{M}
{L}	{E}	{J}	{B}	{E}
{F}	{K}	{K}	{K}	{H}
{N}	{K}	{J}	{L}	{G}

B	D	E	I	M
C	C	E	I	M
B	E	E	J	L
F	H	K	K	K
G	J	K	L	N

D	C	I	B	E
E	E	I	B	E
F	K	J	C	G
L	K	J	K	H
N	M	K	L	M

Fig. 4. The puzzle after the application of the intersection operator including the uniqueness of symbols in individual cells.

We can see in Fig. 5 that in our illustrative example nearly all the positions within the solution table are fixed right now (21 out of 25). Moreover, we can see that the rest of the puzzle will be solved quickly using the same process based on the row and column intersections that will be performed only for the positions that are unfixed yet.

Naturally, symbol D will occupy the top left corner position in the solution table as the result ofintersection between {D} and {D,E,M}, the next position has to be M, because {D,E,M} ∩ {C,M} = {M}, etc. Therefore, in this specific case it is clear that after the second cycle of computing the intersections and fixing the positions where only one candidate symbol exists, the algorithm will reach the solution in Fig. 6.

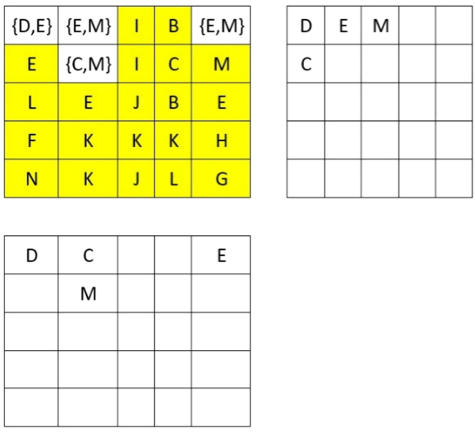

Fig. 5. The puzzle after the fixation of the already known positions.

Of course, depending on the size of the cross-matching puzzle and the number of symbols utilised there could be more cycles needed. If at the end of the cycle all the positions of the solution table are known (it means fixed), we have reached the solution and the program can terminate. The other option is to check the content of the detection or the control table, because at this stage of the computation both of them must be empty (all the symbols were placed and thus deleted from these tables). Secondly, for each cycle we calculate the number of symbols that were fixed within the cycle (*Fixed*). Positive value of *Fixed* indicates that at least one symbol was fixed and removed from the detection and control table and we can safely continue with another cycle. If there is an unique (single) solution to the cross-matching puzzle, our algorithm finds it and terminates.

However, there are situations (especially when the cross-matching puzzle has been generated randomly) that there are several solutions there. In Fig. 7a) we can see the assignment that leads to multiple solution. Utilising the algorithm described above we will reach after two cycles the situation depicted in Fig. 7b). From this stage of calculation no further improvement is possible because the intersection in all four corners always contains two symbols {A,C}. Nevertheless, as no further symbol can be fixed, the value of the above defined indicator *Fixed* is equal to zero and algorithm terminates as well. The output here is a partial solution as one described in Fig. 7b) and in this particular case it indicates that there are two solutions to the cross-matching puzzle depending whether the symbol A or symbol C is selected for the upper left corner in the solution table. However, it is easy to find a solution (or even to generate all the solutions) as whenever we select the particular symbol from the set, this symbol is deleted from the

relevant row as well as the relevant column and the maximum number of choices to be made is less or equal to $(N^2\text{-}N)/2$, where N is the size of the puzzle. We can see that in our example from Fig. 7 only one decision is needed in order to obtain one of the two existing solutions.

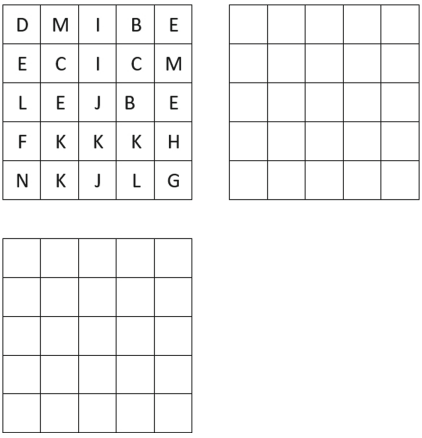

Fig. 6. The solution of the cross-matching puzzle.

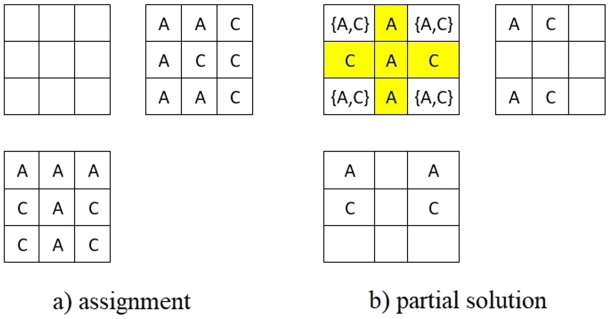

Fig. 7. The cross-matching puzzle with two different solutions.

We can conclude that we have designed a simple fast polynomial deterministic algorithm that solves cross-matching puzzle efficiently. If there is a unique solution to the puzzle, our algorithm will find it. If there are multiple solutions the algorithm will reach the stage when all the positions that could be determined are fixed and the remaining cells are assigned the relevant set of possible symbols. The solution is then to be find by making the choices for these cells one-by-one and by deleting the chosen (and therefore fixed) symbols from the detection and decision tables. Once again, this process will terminate when all the symbols are fixed. The pseudocode of our algorithm is given in Table 1.

Table 1. Cross-matching puzzle algorithm.

```
Input:  N > 0   ... size of the puzzle
        S(N,N)  ... solution table (empty)
        D(N,N)  ... detection table (given)
        C(N,N)  ... control table (given)
Output: S(N,N)  ... filled solution table

for i := 1 to N do
  for j := 1 to N do
     GotIt(i,j):=false;   % no fixed symbol in this cell yet (auxiliary table)
repeat
  Fixed:=0;
  for i := 1 to N do
    for j := 1 to N do
      begin
        if GotIt(i,j)=false then    % still unknown/unfixed cell
        begin
          ISC(i,j):=(i-th row of D) ∩ (j-th column of C);    % intersection
          if ISC(i,j)={X} then    % single option only
            begin
              S(i,j):=X;
              GotIt(i,j):=true;
              remove X from the i-th row of D;
              remove X from j-th column of C;
              Fixed := Fixed + 1;
            end
        end
until Fixed = 0;          % there was no change (no further symbol was fixed)
return S;
```

4 Conclusion

The deterministic algorithm for solving the cross-matching puzzle was presented in this paper. The algorithm works in polynomial time and therefore even large instances of the cross-matching puzzle can be tackled and solved. If there is a unique solution only, this solution is found deterministically. Should there be multiple solutions, the algorithm presented here will terminate at the situation where all the symbols that could be fixed unambiguously were placed in the respective position while there are the sets of the candidate symbols amongst which is it necessary to make the choice elsewhere. Therefore, even in this situation a solution to the cross-matching puzzle (or all of them) can be easily found.

Nevertheless, the main aim of this paper was not to design an algorithm for a specific problem. We wanted to show that while AI researchers constantly look for new constraint-satisfaction problems that could be utilized for testing various problem-solving techniques it is possible to come up with the problem that can be solved by much

simpler algorithms. Moreover, it is important to repeat that one of the top misconceptions related to NP class is that a huge number of potential solutions to the specific problem does not automatically mean that the relevant problem inevitably belongs to the class of NP. Such a misunderstanding and misclassification of the particular problem leads to false impression that there is no chance to design a simple and fast algorithm for such a problem. Consequently, various heuristics or general problem-solving techniques are unnecessarily employed in order to solve it and the wrong impression that the problem is difficult is further supported. There are too many problems in the class of NP anyway and so there is no need to waste our effort as well as unnecessary computational resources on problems that could be solved using polynomial deterministic algorithms.

References

1. Yato, T., Seta, T.: Complexity and completeness of finding another solution and its application to puzzles. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **86**, 1052–1060 (2003)
2. Chatterjee, S., Paladhi, S., Chakraborty, R.: A Comparative Study on the performance characteristics of Sudoku solving algorithms. IOSR J. Comput. Eng. **1**, 69–77 (2014)
3. Slabý, A., Ševčíková, A.: Chess as a motivational tool in education. In: 29th Annual Conference of the European Association for Education in Electrical and Information Engineering, EAEEIE, pp. 1–6 (2019)
4. Kesemen, O., Özkul, E.: Solving crossmatching puzzles using multi-layer genetic algorithms. In: First International Conference on Analysis and Applied Mathematics, 18–21 Oct 2012 Gumushane (2012)
5. Kesemen, O., Özkul, E.: Solving cross-matching puzzles using intelligent genetic algorithms. Artif. Intell. Rev. **49**(2), 211–225 (2016)
6. Mann, Z.A.: The top eight misconceptions about NP-hardness. Computer **50**, 72–79 (2017)
7. Hynek, J.: Genetic algorithms for the N-queens problem. In: Arabnia, H.R., Mun, Y. (Eds.): Proceedings of the 2008 International Conference on Genetic and Evolutionary Methods, pp. 64–68. CSREA Press (2008)