Limiting Backtracking in Fast Sudoku Solvers

Ileana-Diana V.D. Nicolae*, Anca-Iuliana P.M. Nicolae**

**Department of Computers and Information Technology*
(e-mail:nicolae_ileana@software.ucv.ro)
University of Craiova, Decebal Blv. No. 107
***Department of Cybernetics and Economic Statistics*
(e-mail:ancaiuliana.nicolae@gmail.com)
Doctoral School of Bucharest, University of Economic Studies

Abstract: The paper deals with the logic and implementation of an efficient Sudoku solver, successfully tested for classic grids of small and medium difficulty. It relies on an efficient 3-d array (9x9x10) containing information on grid's content and on possible candidates for its unsolved cells at every computation step. The use of auxiliary (9x9) matrices containing flags related to restrictions associated with lines, columns and zones, along with execution branches that implement strategies from gamers' world (known as „x-wing", „y-wing", „hiden/naked pairs/triples", „box-to-line/column restrictions" etc.) and with a minimal (ideally zero) segment of backtracking code, exhibited very good runtimes (not exceeding 0.4 s for classic grids of medium difficulty and respectively 0.15 s for easy puzzles) as compared to usually runtimes of seconds reported by literature. The solver can be easily adapted to solve grids with additional restrictions (e.g. restrictions corresponding to diagonals, symmetric grids etc.).

Keywords: Constraing satisfaction problems, Backtracking, Binary arrays, Computer programming

1. INTRODUCTION

Sudoku puzzle with 9 rows and columns containing digits from 1 to 9, originated in Japan, whose popularity is especially remarkable among gamers all over the world since 2005. The rows/columns are divided into 9 3x3 zones (marked with thicker frames on the grid). In its original (classic) form, restrictions are imposed such as no duplicates are allowed along a line, a column and respectively inside a zone.

In one of their studies on Sudoky puzzles (Ercsey-Ravasz and Toroczkai , 2012) the authors began by reminding that the mathematical structure of these puzzles is akin to hard constraint satisfaction problems lying at the basis of many applications (including ground-state problem of glassy spin systems , protein folding etc.) and proved that the difficulty of Sudoku translates into a transient chaotic behavior exhibited by continuous-time dynamical systems.

Mervyn King (King, 2010), used a Sudoku grid to illustrate why "the mess in the world economy is unlikely to get any better". Its explanation on chosing this way of modelling is that the high-saving countries (e.g. China, Japan) and the low-saving countries (e.g. USA, UK, Spain) are dependent on the choices the other group of countries make. Other relevant examples for Sudoku puzzles' practical applications in Economics are provided by Academia (University of Missouri, 2012).

As a general rule, the fewer clues are given, the harder the puzzle is, but this is not universally true (Rosenhause and Taalman, 2011). One of the most famous exceptions for this rule is the so called "Platinum Blonde" grid, (Sudoku Players Forum, 2009) where the apparently confortable number of initial clues – 21, is in contradiction with is ultra-hard level of difficulty. Fig. 1 reveals that more than 7 minutes were necessary to solve the above mentioned grid with an automatic solver (available at tirl.org/software/sudoku) which otherwise for medium difficulty and easy grids exhibited maximal runtimes under 100 seconds.

A systematic study on this aspect was made in (Ercsey-Ravasz and Toroczkai , 2012) where it was proved that the escape rate κ, an invariant of transient chaos, provides a scalar measure of the puzzle's hardness that correlates well with human difficulty ratings.

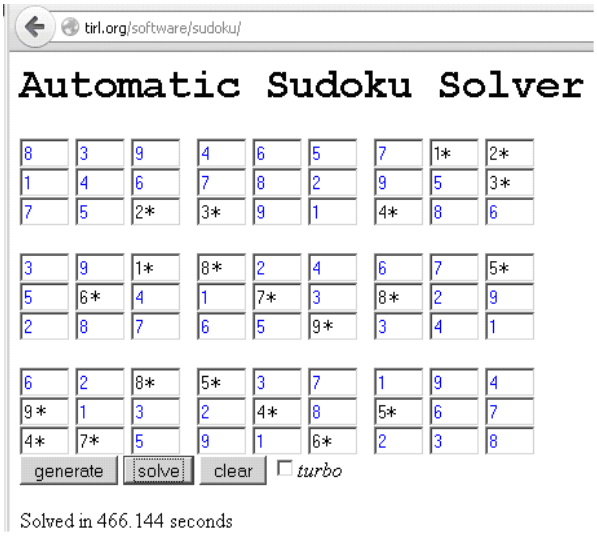

Fig. 1. A Sudoku grid famous for its difficulty (Platinum Blonde). The clues are marked with asterisks.

Accordingly, $\eta = -\log_{10}\kappa$ can be used to define a "Richter"- type scale for puzzle hardness, with easy puzzles having $0 < \eta \leq 1$, medium ones $1 < \eta \leq 2$, hard with $2 < \eta \leq 3$ and ultra-hard with $\eta > 3$. The example from Fig. 1 falls into the last category.

2. ALGORITHMS – STATE OF THE ART

2.1. Fundaments and Theoretic Support

If less then 17 clues are provided, then one can say with certitude that the grid has more than one possible solution. The total number of possible combinations of digits on a standard Sudoku grid is 6,670,903,752,021,072,936,960. Because many of these combinations could be the same as another, only backwards or rotated, the number of real possible combinations drops to 3,359,232. This is essentially the total number of possible Sudoku puzzles (wiki.answers, 2005).

Due to the significantly large number of applicability domains of this puzzle (in both its classic and respectively modified forms – when additional restrictions are imposed to solutions), many theorists contributed to the mathematic support required by various Sudoku-related models (Lynce and Quaknine, 2006), (Simonis, 2005), (Shan and Yap , 2008), (Rosenhause and Taalman, 2011), (Eppstein, 2005) a.o.

Backtracking techniques represent a common approach, often used as study case for computing techniques in universities (Stanford University , 2012) when intending to solve classic grids with no aditional restrictions. These techniques guarantee the detection of all possible solutions, but often involve significant runtimes.

The huge interest in solving Sudoku puzzles all over the world with backtracking techniques is revealed by a great number of programs and routines (many available online), developed in various programming environments: in Java (Armstrong Atlantic State University , 2013), in PHP (Eferanto, 2008), in JavaScript (Detar, 2012), in Python (Python Fiddle, 2012) etc.

2.2. Boolean Representations

A boolean representation which shares the same principles with the one used by our algorithms is depicted by Fig. 2 (Ercsey-Ravasz and Toroczkai , 2012). The difference consists in flaging the unrefined possibility of a certain cell to host a certain digit : our algorithms use 1-s instead of 0-s firstly and change 1-s in 0-s as soon as the algorithm detects restrictions.

2.3. Useful Schemas for Indirect Restrictions Detection

The huge popularity of Sudoku puzzles explaines the efforts behind numerous papers and web pages dedicated to various techniques used to solve them. We found at (Sudokuwiki, 2011) very useful, documented and well-structured information about various schemas which can provide additional restrictions, needed to unblock the process of solutions' deduction when direct eliminating techniques become temporarily unusable. If correctly implemented in performing code, these techniques can refine very much the set of "candidates" – ideally up to the point when backtracking techniques are no longer needed, therefore saving significant computer resources (especially runtime). An interesting classification of these techniques can be found at http://www.sudokuwiki.org, where the strategies' suggestive names ("tough", "diabolical" and respectively "extreme") are used to reveal how difficult the grids to which they are addressed are.

For the beginning we implemented some of the basic techniques and tough strategies in our algorithms, namely those required to solve classic grids with small or medium degrees of difficulty very fast, with the price of acceptable increased algorithm complexity and additional memory consumptions. The elaborated algorithms were also successfully used on grids with additional restrictions.

3. IMPLEMENTATION OF ORIGINAL ALGORITHMS

3.1. Main Data Structures

The main structure used by algorithm consists in a 3-D array c, whose first two indices are used for the identification of a certain location ("place") from the grid.

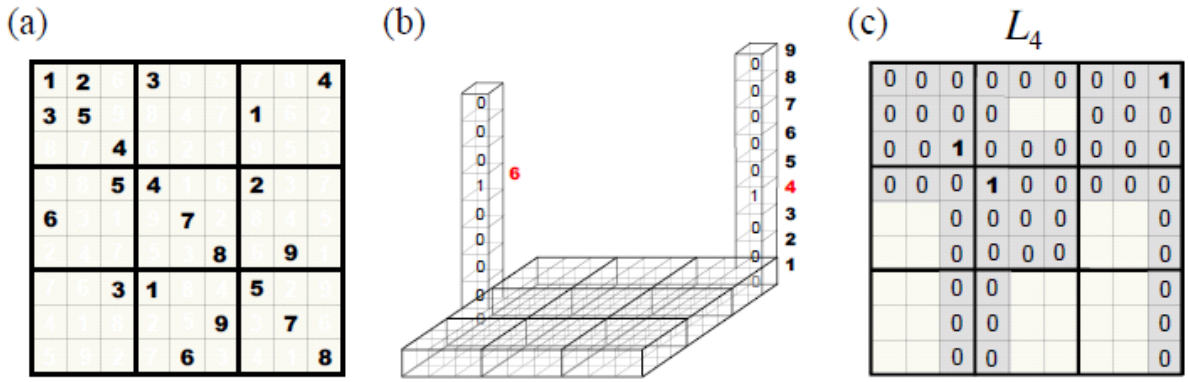

Fig. 2. Boolean representation. (a) a typical puzzle with bold digits as clues ; (b) Setup of the Boolean representation in a 9 x 9 x9 grid; (c) Layer L4 of the puzzle (the one containing the digit 4) with 1-s in the location of the clues and the regions blocked out for digit 4 by the presence of the clues (shaded area).

The third dimension from this array is used to hold information about the digits that can be placed in the location until the reaching of a certain stage from the solving chain (according to the restrictions imposed by the already occupied locations over lines, columns and zones and, if applicable, by other additional restrictions, such as the main and secondary diagonals from the grid).

reads the initial clues;
operates TIRs according to the line, column and zone related restrictions imposed by the initial clues ;
while (unsolved cells)
  for each unsolved cell (i,j)
    if $\left(\sum_{k=1}^{9} c(i,j,k)=1\right)$ then
      find k for which c(i,j,k)=1;
      c(i,j,10)= k ;
      /*(exclusivity of the tested digit in cell(i,j)
      cell (i,j) is solved)*/
      c(i,j,1...9)=0;
      break;
    end if;
  end for;
  for each digit in {1,2..,9}
    operates newly derived line related *TIR*s;
    operates newly derived column related *TIR*s;
    operates newly derived zone related *TIR*s;
    for each cell (i, j)
      if $\left(\sum_{i=1}^{9} c(i,:,tested\_digit)=1\right)$ then
        find j for which c(i,j,tested_digit)=1;
        c(i,j,10)=tested_digit;/*(exclusivity of the tested digit on line i was detected, cell (i,j) is solved)*/
        c(i,j,1...9)=0;
        break;
      end if;
      if $\left(\sum_{i=1}^{9} c(:,j,tested\_digit)=1\right)$ then
        find i for which c(i,j,tested_digit)=0;
        c(i,j,10)=tested_digit;/*(exclusivity of the tested digit on column j was detected, cell (i,j) is solved)*/
        c(i,j,1...9)=0;
        break;
      end if;
      if $\left(\sum_{\substack{i1\in zone(i,j)\\ j1\in zone(i,j)}} c(i_1,j_1,tested\_digit)=1\right)$ then
        find the pair (i_1,j_1) for which $c(i_1,j_1,tested\_digit)=0$ within zone(i,j);
        $c(i_1,j_1,10)$=tested_digit; /*(exclusivity of the tested digit on zone including the cell (i_1,j_1) was detected, cell (i_1,j_1) is solved)*/
        $c(i_1,j_1,1...9)=0$;
      end if;
    end for;
  end for;
  /*accomplish special tests if no new solution found*/
end while;

Fig. 3. Main steps of the algorithm.

For example if c(2,4,3)=1 and c(2,4,6)=1 whilst the rest of components following the pattern c(2,4,*) are zero, it means that the cell from the second line and fourth column can host only the digits 3 and 6 and the solution for this place is not known yet. On the other hand, a set of values c(i,j,k)=0 for k=1...9 and c(i,j,10)=7 means "either the place from the line *i* and column *j* was given as an initial clue and contains the digit 7, or the algorithm deduced that the digit from this place is 7".

Except for the indices following the pattern c(*,*,10), reserved for the storing of places' solutions, all the values for the third dimension of c are initially set to 1. The locations of the first 0-s overwriting them are provided by the initial clues of the grid to be solved. When the algorithm advances, other 0-s are imposed by restrictions until the final solution is derived (all c(*,*,1...9) are zero and c(*,*,10) contain the solution).

The overwriting of 0-s from the third dimension will be addressed from this point forward as "transitions imposed by restrictions" and will be denoted by *TIR*s.

Additional vectors, with significant names ("line", "column" and "zone") are used to store flags used by the algorithm any time new restrictions can be imposed, as follows: line(3,7)=1 if the digit 7 is present on the third line and line(3,7)=0 otherwise.

3.2. Algorithm Description

The main steps of the algorithm are presented by Fig. 3. When no new *TIR*s can be deduced within the cycle "*for each digit*", special tests are accomplished.

A first special test focuses on the identification and exploiting of restrictions involved by the condition known as "Pointing pairs, pointing triples " (Fig. 4 , Sudokuwiki, 2011) .

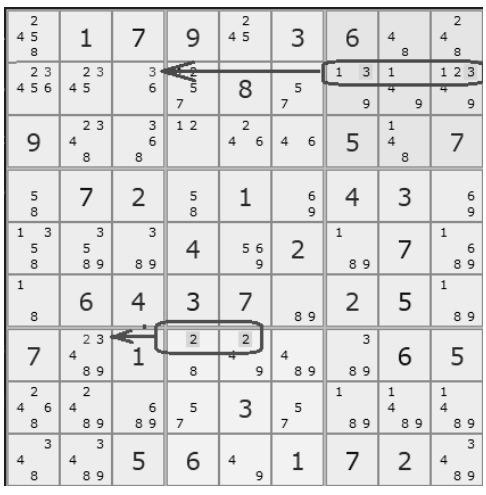

Fig. 4. Example of grid where two conditions of type "Pointing pairs, pointing triples" are accomplished

The rule from this context is "If a pair or triple of digits "*x*" can reside only on a certain line/column from a certain zone *z*, it forbits the presence of "*x*" in the places from the same line/column in the grid outside *z*". This type of restrictions is implemented as follows:

for each zone z(m)

if $\sum_{i1,j1\in zone(i)} c(i_1,j_1,tested\_digit) == 2(or\ 3)$ then

find the set of i_1 and j_1 for which c(i_1,j_1,tested_digit)=1;

if i_1-s are identic then

/* impose restrictions over line i_1*/

c(i_1,*,tested_digit)=0 excepting the columns belonging to z(m)

else

if j_1-s are identic then

/* impose restrictions over column j_1*/

c(*,j_1,tested_digit)=0 excepting the lines belonging to z(m)

end if; end if; end if; end for;

The "line-to-box" and respectively "column-to-box" restrictions are depicted by Fig. 5 (Sudokuwiki, 2011). In this example, the digit "2" can appear on the first line only within the zone 2, and therefore the cells from zone 2 not belonging to the first line cannot host 2.

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

Fig. 5. Example for "line-to-box" restriction.

The restrictions of type "line (or column) to box" are implemented as follows:

for each digit in {1,2,...9}

for each zone z(m)

for each line i from z(m)

if $\sum_{j\in z(m)} c(i,j,tested\_digit) = \sum c(i,:,tested\_digit)$ then

/*impose restrictions over zone:*/

c(i_1,j,tested_digit)=0 for (all $j\in z(m)$ and $i_1 \in z(m), i_1 \neq i$)

end if; end for;

for each column j from z(m)

if $\sum_{i\in z(m)} c(i,j,tested\_digit) = \sum c(:,j,tested\_digit)$ then

/*impose restrictions over zone: */

c(i,j_1,tested_digit)=0 for (all $i\in z(m)$ and $j_1 \in z(m), j_1 \neq j$)

end if; end for;

end for; end for;

Other restrictions that prooved to be useful are the so called "hiden/naked pairs, hiden/naked triples". An example for their application is presented in Fig. 6 (Sudokuwiki, 2011). In this example, the pair (1,6), appearing for the first zone only on the first line (providing that neither 1 nor 6 appear in any other place

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

Fig. 6. Example for "naked pairs" restriction.

from this zone), restricts all the remaining cells from this line such as they can not host none of the digits from the pair.

The implementation of this restriction for lines is:

for each digit1 in {1,2,...9}

for each digit2 in {1,2,...9}

for each line i

cond1=$\left(\sum_{j=1:9} c(i,:,digit1) == \sum_{j=1:9} c(i,:,digit2) == 2\right)$;

cond2 is true if all j-s for which c(i,j,digit1)=1 match all j-s for which c(i,j,digit2)=1;

if (cond1 and cond2) then */the combination (digit1,digit2) forms a hiden/naked pair along line *i* and imposes restrictions over line *i* except the specific combinations (i,jnaked) where pairs appear:*/

c(i,:,digit1)=0;c(i,:,digit2)=0;

restore the old values of cells involved in pair;

end if; end for; end for; end for;

Similar restrictions are imposed by hiden/naked pairs along columns (in this case restrictions of type c(*,j,digit1) and c(*,j,digit2) are imposed).

The presence of hiden/naked pairs within a certain zone z_k is detected with conditions *cond1* which can be written as follows:

$$\sum_{i_z,j_z\in z_k} c(i_z,j_z,digit1) == \sum_{i_z,j_z\in z_k} c(i_z,j_z,digit2) == 2 \quad (1)$$

The second condition here requires that all jz-s for which c(iz,jz,digit1)=1 match all jz-s for which c(iz,jz,digit2)=1. In this case the restrictions which can be imposed are: c(iz,jz,digit1)=0 and c(iz,jz,digit2)=0 for all {iz,jz} belonging to z_k.

A special combination of coordinates of places and their content is known in gammers' world as "x-wing", an example being given in Fig. 7 (Sudokuwiki, 2011).

A x-wing is delimited by places forming a rectangle, each of them hosting a certain digit *x* (in this example *x*=2). On two of the diagram's "units" of the same type (columns or lines) which overlap a pair of similar units forming the rectangle's sides, *x* appears only twice (in this example 2

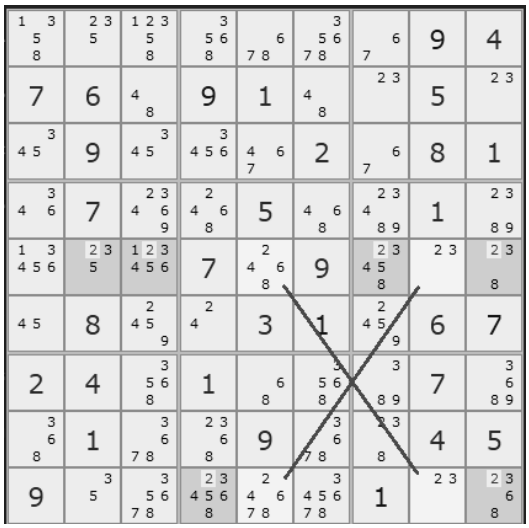

Fig. 7. Example for x-wing forming and restrictions.

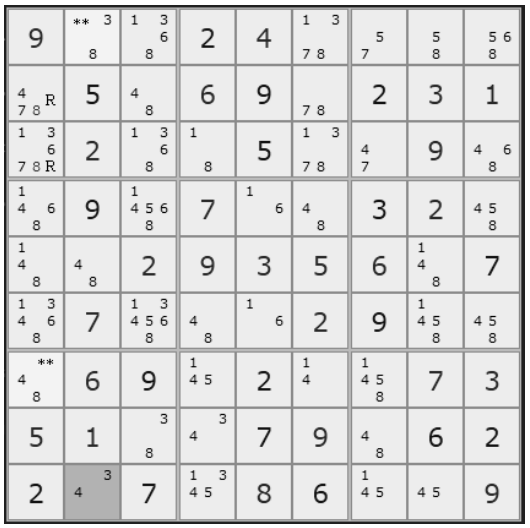

Fig. 8. Example for y-wing forming and restrictions.

appears only twice along the columns 5 and 8 respectively, which overlap the x-wing's vertical sides) and both apparitions are in x-wing corners. The restrictions imposed by x-wing follow the rule (Sudokuwiki, 2011) "when there are only two candidates for a value, in each of 2 different units of the same kind, and these candidates lie also on 2 other units of the same kind, then all other candidates for that value can be eliminated from the latter two units". In this example, 2-s are forbitten in the shadded places, which follow the upper and lower sides of the x-wing.

The implementation of x-wing conducted by lines is as follows:

```
for each digit in {1,2,...9}
    for each pair of lines (i1,i2) , i1=1...8, i2=i1+1...9
```

$$c1 = \left(\sum c(i1,:,tested\_digit) == \sum c(i2,:,tested\_digit) == 2 \right)$$

```
     if c1 then
      find the pairs  (j1_i1,j2_i1) and  (j1_i2,j2_i2)
        corresponding to lines i1 and i2 where the flags for
         tested_digit were 1;
       if maching pairs of js (corners formed) then
        /*x-wing with corners (i1,j1), (i1,j2), (i2,j1), (i2,j2)
          impose restrictions over columns */
         c(:,j1,tested_digit)=0 ; c(:,j2,tested_digit)=0 ;
        //recover corners
         c(i1,j1,tested_digit)=1; c(i1,j2,tested_digit)=1 etc.
     end if; end if;
    end for; end for;
```

Finally, the last special test implemented by our algorithm is related to the so-called "y-wings", an example being given in Fig. 8 (Sudokuwiki , 2011).

The principle behind the forming of this special structure is as follows: a certain place p (the "pivot") can host exactly 2 digits ($p1$ and $p2$). The other two places $y1$ and $y2$, also hosting only a pair of digits, can "see" p and do not "see" each other. In this context "seeing" means to share a certain unit (line, column or zone). $y1$ contains exactly the digits ($p1$, r) and $y2$ contains exactly the digits ($p2$, r).

Then r is restricted to appear in any place pr which can see both places $y1$ and $y2$.

In the given example, the place identified by the coordinates (9,2) is the pivot, the places with coordinates (1,2) and respectively (7,1) are the y-wing's branches and the restricted digit is 6, restrictions appearing in the places with coordinates (2,1) and (2,3).

The implementation of y-wing was made as follows:

```
identify all places p(l) which can host exactly 2 digits
  for all possible triples of places (p(l1), p(l2) and p(l3))
     identify their coordinates from the current position
     following the rule:  i(l)=(int) (l/9); j(l)=l-(i-1)*9;
      if  the triple (p(l1), p(l2) , p(l3)) forms a y-wing with
                 p(l1) as pivot then
        determine the restricted digit r and all p(lr) playing
        the role of pr ;
        for each p(lr)
          determine i(lr)=(int) (lr/9); j(lr)=lr-(i-1)*9;
           if p(lr) sees both p(l2) and p(l3)
            //impose restrictions according to y-wing
              c(ir,jr,r)=0;
          end if;
        end for;
     end if;
  end for;
```

4. TESTS

4.1. Solving Easy Classic Diagrams

Our first tests were done considering the grids from Figs. 9-11, rated as easy because their solving does not require strategies for the imposing of indirect restrictions. The clues are represented with bold fonts.

A natural approach is to evaluate firstly the number of occurances for every distinct digit in the set of input data (clues). Considering the grid from Fig. 9, this means: 5 occurances of digit 1, 3 occurances of digit 2 and so on.

If the analysis within the *while* cycle from Fig. 3 is made

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

Fig. 9. First example of easy grid (22 clues).

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

Fig. 10. Second example of easy grid (18 clues).

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

Fig. 11. The third example of easy grid (17 clues).

such as to consider firstly as "tested digit" the digits found as having the greatest number of occurancies, the conditions of exclusivity which result into cells' solutioning will be derived faster and easier, therefore providing an additional guarantee for the algorithm's possibility to find incrementaly new solutions up to the solutioning of the entire grid.

For the grid depicted by Fig. 9, separate tests were performed considering that the cycle "*for each digit...*" will consider the following sequence of tested digits : 1, 6, 2, 3, 8, 7,9,4,5.

The algorithm was tested for many easy classic grids with at least 17 clues (for which a single solution exists) and no failures were recorded.

The runtimes required to solve the grids from Figs. 8...10, whose number of clues varies from 17 to 24 are gathered in Table 1. In order to diminish as much as possible the

Table 1. Runtimes for standard classic easy grids

Diagram identification	Type of test sequence	Mean runtime [s]
Fig. 9.	Default	0.070
	Calculated	0.073
Fig. 10.	Default	0.116
	Calculated	0.109
Fig. 11.	Default	0.100
	Calculated	0.108

influence of the programming environment's multi-tasking features over the real performances of the algorithm, the runtimes were evaluated as mean values over 10 distinct runnings. The analysis of data from Table 1 proves the algorithm's efficiency: very small runtimes (around 0.1 s) were required to solve grids with the minumum number of clues required by grids with a single solution. Obviously smaller runtimes were recorded when the 1st grid was solved, as more clues were provided. No rule can be derived relative to the relation between the runtimes corresponding to the cases relying on the default test sequence (1,2,3...,9) and respectively to the cases using test sequences imposed by the distribution of clues' values (refered from this point forward as "calculated test sequences"). However small differences are recorded between both cathegories of runtimes. The explanation consists in the unpredictable relation between two runtimes T1 and T2: T1 represents the runtime saved during the execution of the algorithm's main cycle due to easier and faster derivings of TIRs whilst T2 represents the runtime wasted with the imposed test sequence evaluation. The unpredictibility is generated by the random selection of both characteristics of clues: values and positions respectively.

4.2. Solving Classic Diagrams of Medium Difficulty

Many tests were made on classic grids with a medium degree of dificulty, where strategies as those implemented by our algorithm must be used to deduce indirect restrictions over the content of (sets of) places. For a significant percent of tests, the special strategies were sufficient to find the solution without the use of the backtracking branches from our algorithm. A first example for this kind of diagrams is depicted by Fig. 12.

For this case, the efficiency of branches dealing with x-wing scenarios implemented by our algorithm is depicted

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

Fig. 12. First example of grid with medium difficulty.

4	258	9	7	1	6	2358	58	2358
6	1	25	3	8	9	257	4	257
38	7	38	2	4	5	1	6	9
3578	58	3578	9	6	4	578	2	1
258	258	4	1	7	3	6	9	58
1	9	6	8	5	2	47	3	47
9	6	58	4	2	1	358	7	358
257	3	1257	6	9	8	245	15	245
28	4	128	5	3	7	9	18	6

Fig. 13. Example from execution: indirect elimination through x-wing.

4	25	9	7	1	6	2358	58	2358
6	1	25	3	8	9	257	4	257
38	7	38	2	4	5	1	6	9
37	58	37	9	6	4	58	2	1
258	258	4	1	7	3	6	9	58
1	9	6	8	5	2	47	3	47
9	6	58	4	2	1	358	7	358
57	3	157	6	9	8	24	15	24
28	4	128	5	3	7	9	18	6

Fig. 14. Example from execution : indirect elimination through y-wing.

by Fig. 13, where a x-wing determined by the digit 7 along the lines 2 and 6 and bordered by the columns 7 and 9 imposes the restriction "7 cannot appear in the place with coordinates (4,7)" .

The efficiency of branches dealing with y-wings scenarios is depicted by Fig. 14, where the pivot is surounded by an ellipse, the y-wing branches are marked by rectangles and the cell marked by a rhomb is restricted – it is not allowed to contain the digit 2.

Still for certain grids (as that depicted by Fig. 15 – classified as "tough"), the use of the backtracking branch is compulsory.

We considered usefull to provide in detail the evolution of execution (Table 2). To denote the restrictions, we used the following codes: lc = derived from single possible occurance along line or column; us=unique solution (single possibility in place after restrictions' applying); z= unique solution in zone; B – solution selected by backtracking.

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

Fig. 15. Second example of grid with medium difficulty.

Beginning with the 16-th step (marked with *), the use of special strategies was compulsory, and beginning with the 26-th step (marked with **), the backtracking branch was activated, as the scenario from Fig. 16 was achieved.

3	16	9	26	18	258	4	7	25
2	146	5	7	14	9	8	16	3
14	8	7	246	3	25	9	16	25
7	5	4	8	6	1	2	3	9
6	13	13	9	2	4	7	5	8
9	2	8	3	5	7	6	4	1
14	1347	13	24	478	28	5	9	6
5	**49***	2	1	**49**	6	3	8	8
8	**79**	6	5	**79**	3	1	2	4

Fig. 16. Partial solution for the grid from Fig. 15, when backtracking becomes operational.

The presence of a special rectangular construction (marked with bolded fonts in Fig. 16) makes the algorithm to choose between two alternatives for the place positioned at the coordinates (8, 2): 4 and respectively 9. Only the first option generates a valid solution.

An analysis of the runtimes required for the solutioning of both analysed medium difficulty grids revealed a mean runtime of 0.0936 s corresponding to the grid from Fig. 14 and respectively of 0.376 s for the grid from Fig. 15, revealing the additional effort in the case when the backtracking branch was used.

Table 2. Places solutions along with the restrictions which yielded them for grid from Fig. 15.

Detection order	Index line	Index column	Solution	Restriction code	Detection order	Index line	Index column	Solution	Restriction code	Detection order	Index line	Index column	Solution	Restriction code
1	4	3	4	lc	18	9	8	2	us	35	7	2	7	us
2	4	4	8	us	19	8	8	8	us	36	9	2	9	us
3	5	7	7	us	20	9	4	5	us	37	5	2	3	us
4	6	1	9	us	21	9	6	3	us	38	7	5	8	us
5	4	6	1	us	22	8	3	2	lc	39	1	5	1	us
6	5	8	5	us	23	2	3	5	lc	40	7	6	2	us
7	6	4	3	us	24	8	1	5	lc	41	1	2	6	us
8	4	9	9	us	25	2	9	3	us	42	3	6	5	us
9	5	5	2	us	26**	3	5	3	lc	43	7	4	4	us
10	6	6	7	us	27	8	2	4	B	44	1	4	2	us
11	6	7	6	us	28	2	5	4	lc	45	2	2	1	us
12	7	9	6	lc	29	7	1	1	us	46	3	9	2	us
13	1	8	7	lc	30	8	5	9	us	47	1	6	8	us
14	3	7	9	lc	31	3	1	4	us	48	2	8	6	us
15	8	7	3	z	32	7	3	3	us	49	3	4	6	us
16*	2	7	8	lc	33	9	5	7	us	50	1	9	5	us
17	9	1	8	us	34	5	3	1	us	51	3	8	1	us

As both runtimes were very small (under 0.4 s), one can conclude that even for medium difficulty grids the algorithm is very efficient from the execution point of view. The moderate additional memory required by the auxiliary data structures used to improve the runtime efficiency and to implement the special strategies that help avoiding the backtracking up to a point when it should be used to select between a significantly reduced set of options (or should not be used at all), let us conclude that this algorithm represents a reliable option for runtime applications where the time is critical and the grids to be solved are known as having a medium degree of difficulty.

CONCLUSIONS

The logic, implementation, testing and evaluation of an efficient Sudoku solver, successfully tested for classic grids of small and medium difficulty, were presented.

An efficient 3-d array containing information on grid's content and on possible candidates for its unsolved cells at every computation step is used. Auxiliary matrices containing flags related to restrictions associated with lines, columns and zones are used, along with execution branches that implement special strategies for indirect applying of restrictions (known as „x-wing", „y-wing", „hiden/naked pairs", „box-to-line restrictions" etc.). These techniques result into the reducing (usually up to zero) of the necessity to use the segment of backtracking code. This is revealed by the obtaining of very good runtimes (not exceeding 0.4 s for classic grids of medium difficulty and respectively 0.15 s for easy puzzles) as compared to usually runtimes of seconds reported by literature.

The solver can be easily adapted to solve grids with additional restrictions (e.g. restrictions corresponding to diagonals, symmetric grids etc.).

The moderate additional memory required by the auxiliary data structures used to improve the runtime efficiency and to implement the special indirect restrictioning-related strategies let us conclude that this algorithm represents a reliable option for runtime applications where the time is critical and the grids to be solved are known as having a medium degree of difficulty.

REFERENCES

Ercsey-Ravasz, M. and Toroczkai, Z. (2012), The Chaos Within Sudoku, Nature Oct. 2012, article no. 725, available on line at http://arxiv.org/pdf /1208.0370v1.pdf.

University of Missouri (2012), "Sudoku in Economics", available at web.missouri.edu/~dls6w4/ Word/EconSudoku.docx

King, M. (2010) "Mervyn King: Sudoku for Economists", available at http://www.keyscorner.com/archives /2010/01/21/mervyn-king-sudoku-for-economists/

Rosenhause, J. and Taalman, L. (2011), *Taking Sudoku Seriously: The Math Behind the World's most Popular Pencil Puzzle* , Oxford University Press, New York.

Sudoku Players Forum (2009), *The Hardest Sudokus,* available at http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539.html.

Wiki.answers (2005) http://wiki.answers.com/Q/ How_many_possible_solutions_are_there_for_a_single_ sudoku_game.

Lynce, I. and Ouaknine, J. (2006), Sudoku as a SAT problem, *Proceed. of AIMATH 06*, pp. 121-130.

Simonis, H. (2005), Sudoku as a Constraint Problem, *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems* , pp. 13-27.

Shan, G. and Yap, R. (2008), Solving puzzles (eg. Sudoku) and other combinatorial problems with SAT, *Nurop 2008*, available at http://www.nus.edu.sg/ nurop/ 2009/ SoC/GaoShan_Solving_Puzzles__Eg._Sudoku_and _ other_Combinatorial_Problems_with_SAT.pdf.

Eppstein, D. (2005), Nonrepetitive paths and cycles in graphs with application to Sudoku, *ACM Computing Research Repository* .

Sudokuwiki (2011), http://www.sudokuwiki.org/ Intersection_Removal

Stanford University (2013), *Lecture 11 - Programming Abstractions*, available at http://www.youtube.com/ watch? v=p-gpaIGRCQI

Armstrong Atlantic State University (2013), http://www.cs.armstrong.edu/liang/intro7e/ book/ Sudoku.java.

Python Fiddle (2012), *Shortest Sudoku Solver in Python*, available at http://pythonfiddle.com/shortest-sudoku-solver-in-python/.

Detar, C. (2012), *Automatic Sudoku Solver,* available at tirl.org.software.sudoku

Eferanto, E. (2008), *Sudoku creator/solver with PHP*, available at http://www.emanueleferonato.com/ 2008/12/09/sudoku-creatorsolver-with-php.