SOLVING SUDOKU

BERNDT E. SCHWERDTFEGER

For Carmen

ABSTRACT. This paper contains the description of a `C` program for solving *sudoku* puzzles. It includes programming notes explaining the algorithms.
Dedicated to my wife for her interest in the subject.

PREFACE

This paper contains the description of a program for solving *sudoku* puzzles. There are several programs available in the Internet, see the references [1, 2, 4–7]. Some look mysterious to me, without much of a justification. I got interested in understanding the process of resolution and started to write my own routines. The article [8] sketches one approach and has given me some early orientation.

The *Programming Notes* explain the structure and development of my `C` program `sudoku` and give sufficient justification for understanding the algorithms. As I developed the program I made use of basic information structures, that Knuth describes in [3, vol. I] (fundamental algorithms). I thank Carmen for firing up my inspiration in this task.

Berlin, July 19, 2007 B. E. Schwerdtfeger

PROGRAMMING NOTES

Representing *sudoku* puzzles. A *sudoku* puzzle typically looks like this

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

The *sudoku condition* is the rule that the entries of the matrix have to be numbers $1 \leq n \leq 9$ subject to the restriction that there are no repetitions in *rows*, *columns* and *boxes*. To *solve the puzzle* is the task to fill in the empty entries in the matrix with valid numbers such that the *sudoku* condition remains satisfied.

2010 *Mathematics Subject Classification.* Primary 05-04; Secondary 15-04.
Key words and phrases. solving sudoku, program algorithm.
 version 1.0, rev. 514, March 4, 2015.

The given matrix $a = (a_{ik})_{0\le i\le 8, 0\le k\le 8}$ is fed into the program via an `ASCII` text file, stored like this

$$\begin{array}{ccc|ccc|ccc}
0 & 8 & 0 & 0 & 3 & 0 & 0 & 5 & 0 \\
3 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 5 & 0 & 2 & 0 & 3 & 0 & 0 \\
\hline
0 & 0 & 0 & 4 & 0 & 1 & 0 & 6 & 0 \\
4 & 0 & 9 & 0 & 0 & 0 & 1 & 0 & 3 \\
0 & 7 & 0 & 6 & 0 & 3 & 0 & 0 & 0 \\
\hline
0 & 0 & 6 & 0 & 4 & 0 & 7 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 4 \\
0 & 4 & 0 & 0 & 5 & 0 & 0 & 2 & 0
\end{array}$$

Its internal representation is thus given by the condition that $0 \le a_{ik} \le 9$, where the $a_{ik} = 0$ represent the *empty* entries. In the `sudoku` program the data structure is `int a[9][9]`. The routine for reading the matrix is

```
void read(FILE *fp)
{
   int i,k;
   for(i=0;i<9;i++)
      for(k=0; k<9; k++)
         fscanf(fp,"%d",&a[i][k]);
}
```

and printing the matrix is done by

```
void print(FILE *fp)
{
   int i,k;
   for(i=0;i<9;i++){
      if(i%3==0)                // every third line extra newline
         fprintf(fp,"\n");
      for(k=0; k<9; k++){
         if(k%3==0)             // every third cell extra space
            fprintf(fp," ");
         fprintf(fp,"%i ",a[i][k]);
      }
      fprintf(fp,"\n");
   };
}
```

The *sudoku* condition. I start with the task to implement the *sudoku* condition in the computer program. The *sudoku* condition is a property of the whole matrix $a = (a_{ik})$: if the condition is fulfilled, a is *valid*, otherwise a is *invalid*. To be *invalid* it suffices that one of the 27 different vectors (9 *row*, 9 *column* and 9 *box* vectors) has a repetition. Such a violation of the *sudoku* condition will be called a *foul* and is a property of a vector v; we have either `foul(v)=true` or `foul(v)=false`. The routine *invalid* loops thru all 27 vectors, testing for a *foul*. The idea for *foul* is simple: we will go thru the vector v and count the values v_n. If any count is > 1 there is a *foul*. This simple idea will again serve us later.

```
int foul(int v[])
{
   int n,t[10]={0};

   for(n=0;n<9;n++)
```

```
      t[v[n]]+=1;                // count the value v[n]
   for(n=1;n<10;n++)
      if (t[n]>1)                // if any count is >1 ...
         return 1;               // ... step out
   return 0;
}
```

Neighbours. Let us call two matrix elements *neighbours* if they are contained in the *same row*, the *same column* or the *same box*. Each element a_{ik} has 20 neighbours: 8 neighbours in the i^{th} row and 8 in the k^{th} column, plus 4 more elements in the ambient box. Here is an example, where I highlighted the neighbours of element a_{54}, whose position is marked by a cross ($\times$):

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

We need a formula for running thru a box. Let us quickly reflect on the *indexing* of the matrix elements. The element a_{ik} can also uniquely be described by the *box* number b and *cell* number c inside the *box* in which it is located:

	0	1	2	3	4	5	6	7	8
0									
1		0			1			2	
2									
3									
4		3			4			5	
5									
6									
7		6			7			8	
8									

cells in boxes

0	1	2
3	4	5
6	7	8

The transformation formulas back and forth between (i,k) and (b,c) are

$$b = 3\cdot[i/3]+[k/3] \qquad c = 3\cdot i \bmod 3 + k \bmod 3$$
$$i = 3\cdot[b/3]+[c/3] \qquad k = 3\cdot b \bmod 3 + c \bmod 3$$

In the `sudoku` program these formulas will be tacitly used, as well as

$$[b/3] = [i/3] \qquad b \bmod 3 = [k/3]$$
$$[c/3] = i \bmod 3 \qquad c \bmod 3 = k \bmod 3$$

Thus, (i,k) being fixed, the 9 matrix elements

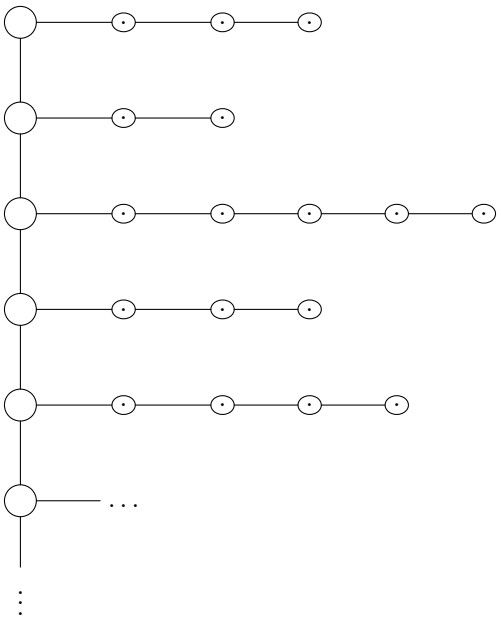
$$a_{3[i/3]+[c/3],3[k/3]+c \bmod 3} \qquad \text{for } 0 \le c \le 8$$

exhaust all elements in the box in which a_{ik} is located.

Candidates. We will focus on the empty entries in the matrix $a = (a_{ik})$. During the process of solving a *sudoku* puzzle we subsequently fill one empty entry after another with a number such that the modified matrix remains *valid*. Those numbers are commonly known as *candidates*.

If $a_{ik} = 0$ and c is a candidate, the assignment $a_{ik} = c$ will be called an *elementary step* and *solving* the *sudoku* puzzle is a sequence of elementary steps, each reducing the number of empty entries by one, until we either have no more empty entries (we have the solution), or there are no more candidates for remaining empty entries. In the latter case we have to backtrack, as we must have picked the wrong candidate in a previous step. It is clear that the whole process eventually comes to an end.

The situation is visualized in the following diagram



where the big circles symbolize the empty entries in the matrix and the smaller (dotted) circles stand for the candidates.

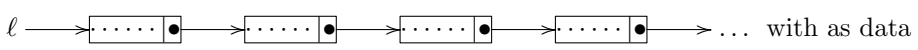
To determine the possible candidates we use the same strategy as in *foul*: go thru the neighbours and count the occurring values. The values with count $= 0$ are the candidates.

Let me illustrate this point with our example puzzle: go thru the neighbours of a_{54} (see previous page) and count the occurring values:

1	2	3	4	5	6	7	8	9
1	1	2	2	1	1	1	0	0

so the candidate list at a_{54} is $\{8, 9\}$. To set $a_{54} = 8$ is an elementary step, as is $a_{54} = 9$, but only one of them will lead to the solution.

Structured Cells of empty elements. To implement the structure in the diagram I have chosen a linked list [3, 2.2.3] of *Cells*



with as data

- a pointer into the matrix $a = (a_{ik})$: `int *b`,
- a candidate array: `int c[10]`.

This structure is defined in the header file as

```
struct Cell{
int *b;                     // pointer into a[][]
int c[10];                  // candidate array
struct Cell *x;             // link to next Cell
};
```

We create the linked list by stepping thru the matrix $a = (a_{ik})$ and for each $a_{ik} = 0$ allocate a `Cell`. We start at the end and go backwards, link the Cells with the next pointer and put the new cell in front of the linked list.

```
void build(void)
{
   int *b=&a[8][8]+1;           // start pointing beyond a[][]
   struct Cell *p;              // pointer to allocated cell

   while(b-->&a[0][0]){         // while b points inside matrix a[][]
      if(*b==0){                // if cell is empty
         p=malloc(sizeof(struct Cell)); // allocate a Cell
         p->b=b;                // fill the pointer to matrix
         p->x=l;                // next pointer
         l=p;                   // insert new cell
      }
   }
   return;
}
```

The following routine `cand` accepts a pointer to `Cell`, determines all candidates for a_{ik} pointed to by the `Cell` and returns the number of candidates (here I make use of the transformation formulas from the *neighbours* section):

```
int cand(struct Cell *p)
{
   int i,k,c,t[10]={0},*b;

   b=p->b;                      // point to empty cell
   c=b-a[0];                    // index into a[][]
   i = c/9;                     // row number
   k = c%9;                     // column number
   for(c=0;c<9;c++){
      t[a[i][c]]+=1;            // count in row i
      t[a[c][k]]+=1;            // count in column k
      t[a[i/3*3+c/3][k/3*3+c%3]]+=1;    // count in box i,k
   };
   b=p->c;                      // point to candidate array
   for(c=1;c<10;c++)            // check for ...
      if (t[c]==0)              // ... candidate and
         *b++=c;                // ... store it
   *b=0;                        // set last entry (null termination)
   return b-p->c;               // return number of candidates
}
```

The recurring solution scheme. The basic idea for the solution is to apply elementary steps. After each step we are basically in the same situation as before, with a modified valid matrix and we can restart our procedure again: thus we start a *recursion* to make the next elementary step. As the number of empty entries

decreases during this process, it comes to an end, and we only need to keep track of successful and flawed candidates.

To manage this housekeeping we will maintain a stack of *modified* Cells: those that are no longer empty but tested by the candidates. A routine `next` will give us the pointer to the *next* Cell with a minimal number of candidates and remove this Cell from the linked list and put it onto the stack. The backtracking consists of taking the Cell back from the stack and insert it into the linked list.

The following *recurring* scheme to the *solution* will do our job:

```
int solution()
{
   int *b,*c;                  // empty, candidate pointer
   struct Cell *p;             // next Cell pointer

   while(p=next()){            // while there are emtpy cells
      b=p->b;                  // point into matrix
      c=p->c;                  // point to candidate
      while(*b=*c++){          // go thru all candidates
         if(solution())        // recursion, if successful ...
            return 1;          // ... we are done
      };                       // ... otherwise, backtracking ...
      m=p->x;                  // remove Cell from solution stack
      p->x=l;                  // .. push it into linked list
      l=p;                     // .. set anchor to new top
      return 0;                // .. signal failure
   };
   return 1;                   // no empty cell left: success
}
```

This *magical* routine `solution` has two possible outcomes: *success* or *failure*. Thus, the main program is quite simple:

```
int  main(void)
   read(stdin);
   if(invalid()){              // if invalid: signal error
      printf("The input file is invalid !\n");
      return 1;
   };
   build();                    // build empty chain
   if (solution())
      print(stdout);           // print the solution
   else
      printf("There is no solution !\n");
   return 0;
}
```

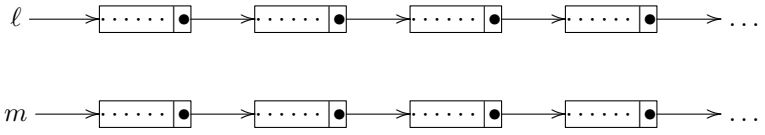
The complete program logic and most tasks are now fully described. We only need to complete the details for the remaining routines

- `invalid` – check if matrix is invalid
- `next` – get the next empty Cell and calculate candidates

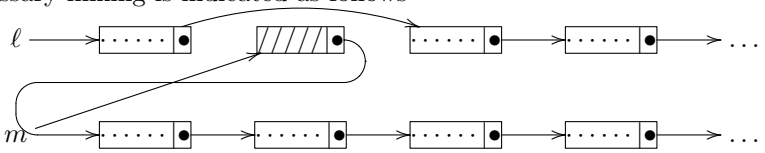
The routine `invalid` is straight forward and needs no explanation. We will now see, who is *next* ...

Who is next ? To determine the next Cell with the required conditions we will go thru the linked list and calculate for each Cell the candidates, keeping track of the (first) minimal one and finally do the necessary linking (remove from linked list, put it on stack).

Here is a diagram



Assume the second Cell in the linked list has minimal number of candidates then the necessary linking is indicated as follows



We learn from this picture that when walking down the linked list we need to maintain a current pointer and a *preceding* pointer. The pair (p, q) will be the pointer pair running thru the linked list and the pair (r, s) will point to the minimal one that we have hit so far. The routine `next` is therefore

```
struct Cell *next(void)
{
   int c,d=9;                   // number of candidates, minimal
   struct Cell *p=NULL;         // preceding Cell
   struct Cell *q=l;            // current Cell
   struct Cell *r=NULL;         // preceding minimal Cell
   struct Cell *s=l;            // minimal Cell

   if(l){
      while(q){                 // while there is a Cell ...
         if((c=cand(q))<d){     // fill candidates and determine ...
            d=c;                // ... minimal number and ...
            r=p;                // ... preceding and ...
            s=q;                // ... minimal Cell
         }
         p=q;                   // current Cell (becomes preceding)
         q=q->x;                // next Cell    (becomes current)
      }
      if(r)                     // s is not first cell
         r->x=s->x;             // remove Cell s from linked list
      else                      // s is first cell
         l=s->x;                // remove Cell s from linked list
      s->x=m;                   // add it to modified Cells
      m=s;                      // set anchor
   }
   return s;                    // return minimal cell
}
```

SUDOKU HEADER FILE AND PROGRAM

/* This is the header file:

```
** -------------------------------------------------------------------
**
```

```
** Module:  sudoku.h
**
** Description:
**
**      Header file for sudoku.c
**      see more details in the programming notes
**
** $Date: 2011-04-05 22:19:50 +0200 (Di, 05 Apr 2011) $
** $Revision: 110 $
**
** Copyright (C) 2007 Berndt E. Schwerdtfeger
**
** ----------------------------------------------------------------*/

// structure of empty cells

   struct Cell{
   int *b;                      // pointer into a[][]
   int c[10];                   // candidate array
   struct Cell *x;              // link to next Cell
   };

// prototype statements for functions

   void read(FILE *);
   void print(FILE *);
   int foul(int[]);
   int invalid(void);
   void build(void);
   int cand(struct Cell *);
   struct Cell *next(void);
   int solution(void);

/*

*/  /* This is the program source:

** ----------------------------------------------------------------
**
** Module:  sudoku.c
**
** Description:
**
**      This program solves sudoku puzzles given in matrix format
**      from standard input (or redirected from an ASCII file) and
**      prints the solution to standard output.
**
**      Input:  sudoku puzzle (incomplete matrix)
**      Output: sudoku solution (completed matrix)
**
** Subroutines: read, print, foul, invalid, build, cand, next, solution
**
** $Date: 2011-04-05 22:19:50 +0200 (Di, 05 Apr 2011) $
** $Revision: 110 $
**
** Copyright (C) 2007 Berndt E. Schwerdtfeger
**
**  This program is free software: you can redistribute it and/or modify
```

```
**  it under the terms of the GNU General Public License as published by
**  the Free Software Foundation, either version 3 of the License, or
**  (at your option) any later version.
**
**  This program is distributed in the hope that it will be useful,
**  but WITHOUT ANY WARRANTY; without even the implied warranty of
**  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
**  GNU General Public License for more details.
**
** ------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "sudoku.h"

// variables

   int a[9][9];                 // sudoku puzzle matrix
   struct Cell *l=NULL;         // anchor to linked list of empty cells
   struct Cell *m=NULL;         // anchor to modified cells

int main(void)
{
// start of main processing

   read(stdin);
   if(invalid()){               // if invalid: signal error
      printf("The input file is invalid !\n");
      return 1;
   };
   build();                     // build empty chain
   if (solution())
      print(stdout);            // print the solution
   else
      printf("There is no solution !\n");
   return 0;
}

/* ------------------------------------------------------------------
**
** Subroutine   read
**
** Description:
**      reads the input file and populates the matrix a[9][9]
**      in sudoku format (see programming notes)
**
**      Input:  FILE *fp file pointer for input file (default stdin)
**      Output: void
**
** ------------------------------------------------------------------*/

void read(FILE *fp)
{
   int i,k;
   for(i=0;i<9;i++)
      for(k=0; k<9; k++)
         fscanf(fp,"%d",&a[i][k]);
```

```
}

/* ------------------------------------------------------------------
**
** Subroutine   print
**
** Description:
**      prints the matrix a[9][9] in sudoku format
**
**      Input:  FILE *fp file pointer for output file (default stdout)
**      Output: void
**
** ------------------------------------------------------------------*/

void print(FILE *fp)
{
   int i,k;
   for(i=0;i<9;i++){
      if(i%3==0)                // every third line extra newline
         fprintf(fp,"\n");
      for(k=0; k<9; k++){
         if(k%3==0)             // every third cell extra space
            fprintf(fp," ");
         fprintf(fp,"%i ",a[i][k]);
      }
      fprintf(fp,"\n");
   };
}

/* ------------------------------------------------------------------
**
** Subroutine   foul
**
** Description:
**      detects violation of sudoku conditions in vector v[]
**
**      Input:  void
**      Output: 1 (sudoku condition is violated)
**              0 (no foul)
**
** ------------------------------------------------------------------*/

int foul(int v[])
{
   int n,t[10]={0};

   for(n=0;n<9;n++)
      t[v[n]]+=1;               // count the value v[n]
   for(n=1;n<10;n++)
      if (t[n]>1)               // if any count is >1 ...
         return 1;              // ... step out
   return 0;
}

/* ------------------------------------------------------------------
**
** Subroutine   invalid
```

```
**
** Description:
**      validates current matrix a[9][9] for sudoku conditions
**
**      Input:  void
**      Output: 1 (invalid)
**              0 (valid)
**
** --------------------------------------------------------------------*/

int invalid()
{
   int i,k,n, v[9];
   for(i=0;i<9;i++)
      if (foul(a[i]))           // test row vector i
         return 1;
   for(k=0;k<9;k++){
      for (i=0;i<9;i++)
         v[i]=a[i][k];          // column vector
      if (foul(v))              // test column vector k
         return 1;
   };
   for(k=0;k<9;k++){
      for(n=0;n<9;n++)
         v[n]=a[k/3*3+n/3][k%3*3+n%3];  //  box vector
      if (foul(v))              // test box vector k
         return 1;
   };
   return 0;                    // all vectors are ok
}

/* --------------------------------------------------------------------
**
** Subroutine   build
**
** Description:
**      builds linked list of emtpy cells
**
**      Input:  void
**      Output: void
**
** --------------------------------------------------------------------*/

void build(void)
{
   int *b=&a[8][8]+1;           // start pointing beyond a[][]
   struct Cell *p;              // pointer to allocated cell

   while(b-->&a[0][0]){         // while b points inside matrix a[][]
      if(*b==0){                // if cell is empty
         p=malloc(sizeof(struct Cell)); // allocate a Cell
         p->b=b;                // fill the pointer to matrix
         p->x=l;                // next pointer
         l=p;                   // insert new cell
      }
   }
   return;
```

```
}

/* ------------------------------------------------------------------
**
** Subroutine   cand
**
** Description:
**      fills the candidates into the Cell
**
**      Input:  void
**      Output: int number of candidates
**
** ------------------------------------------------------------------*/

int cand(struct Cell *p)
{
   int i,k,c,t[10]={0},*b;

   b=p->b;                      // point to empty cell
   c=b-a[0];                    // index into a[][]
   i = c/9;                     // row number
   k = c%9;                     // column number
   for(c=0;c<9;c++){
      t[a[i][c]]+=1;            // count in row i
      t[a[c][k]]+=1;            // count in column k
      t[a[i/3*3+c/3][k/3*3+c%3]]+=1;    // count in box i,k
   };
   b=p->c;                      // point to candidate array
   for(c=1;c<10;c++)            // check for ...
      if (t[c]==0)              // ... candidate and
         *b++=c;                // ... store it
   *b=0;                        // set last entry (null termination)
   return b-p->c;               // return number of candidates
}

/* ------------------------------------------------------------------
**
** Subroutine   next
**
** Description:
**      points to the next minimal empty cell and (if it exists)
**      calculates its candidates
**
**      Input:  void
**      Output: struct Cell pointer to a minimal empty cell
**
**      boundary condition: if l=NULL returns NULL
**
** ------------------------------------------------------------------*/

struct Cell *next(void)
{
   int c,d=9;                   // number of candidates, minimal
   struct Cell *p=NULL;         // preceding Cell
   struct Cell *q=l;            // current Cell
   struct Cell *r=NULL;         // preceding minimal Cell
   struct Cell *s=l;            // minimal Cell
```

```
   if(l){
      while(q){                  // while there is a Cell ...
         if((c=cand(q))<d){      // fill candidates and determine ...
            d=c;                 // ... minimal number and ...
            r=p;                 // ... preceding and ...
            s=q;                 // ... minimal Cell
         }
         p=q;                    // current Cell (becomes preceding)
         q=q->x;                 // next Cell    (becomes current)
      }
      if(r)                      // s is not first cell
         r->x=s->x;              // remove Cell s from linked list
      else                       // s is first cell
         l=s->x;                 // remove Cell s from linked list
      s->x=m;                    // add it to modified Cells
      m=s;                       // set anchor
   }
   return s;                     // return minimal cell
}

/* ---------------------------------------------------------------
** 
** Subroutine   solution
** 
** Description:
**      solves a sudoku matrix
**
**      Input:  void
**      Output: 1 (success)
**              0 (failure)
**
** ---------------------------------------------------------------*/

int solution()
{
   int *b,*c;                    // empty, candidate pointer
   struct Cell *p;               // next Cell pointer

   while(p=next()){              // while there are emtpy cells
      b=p->b;                    // point into matrix
      c=p->c;                    // point to candidate
      while(*b=*c++){            // go thru all candidates
         if(solution())          // recursion, if successful ...
            return 1;            // ... we are done
      };                         // ... otherwise, backtracking ...
      m=p->x;                    // remove Cell from solution stack
      p->x=l;                    // .. push it into linked list
      l=p;                       // .. set anchor to new top
      return 0;                  // .. signal failure
   };
   return 1;                     // no empty cell left: success
}

/*

*/
```

References

[1] Wayne Gould, *Wayne Gould Puzzles*, http://waynegouldpuzzles.com. Accessed December 27, 2015.
[2] Angus Johnson, *Simple Sudoku*, http://angusj.com/sudoku/. Accessed December 27, 2015.
[3] Donald E. Knuth, *The Art of Computer Programming*, Addison–Wesley, 1997/1998.
[4] Sourceforge, *Su Doku Solver*, http://sudoku.sourceforge.net/. Accessed December 27, 2015.
[5] Sudokuessentials, *Sudoku Essentials for Sudoku Fans*, http://sudokuessentials.com/. Accessed December 27, 2015.
[6] Andrew Stuart, *Sudoku Solver*, http://www.scanraid.com/sudoku.htm. Accessed December 27, 2015.
[7] Sudoku Solver, *Sudoku Solver by Logic*, http://www.sudokusolver.co.uk/. Accessed December 27, 2015.
[8] Wikipedia, *Algorithms of sudoku*, http://en.wikipedia.org/wiki/Algorithmics_of_Sudoku.