Solving Sudoku with MATLAB

Raluca Marinescu Andrea Garcia Ivan Castro
Eduard Paul Enoiu
Mälardalen University, Västerås, Sweden
{rmu09001, aga09001, ico09002, eeu09001}@student.mdh.se

March 25, 2011

Abstract

Sudoku puzzles appear in magazines, newspapers, web pages and even books on a daily basis. In fact, millions of people around the world know how to play Sudoku. However, the science behind this game is much more complex than it looks. That is why many researchers have put a notable amount of effort to generate efficient algorithms to solve these puzzles. Some researchers might even suggest that an algorithm to solve Sudoku games without trying a large amount of permutations does not exist.

This paper describes the development and implementation of a Sudoku solver using MATLAB. Furthermore, this document includes detailed directions about the creation of a graphical user interface and the implementation of a constraint-propagation algorithm to complete Sudoku puzzles interactively.

Contents

1 Introduction

Games and puzzles have been a platform for application of mathematics, artificial intelligence and other fields techniques. The past years have brought into attention a very popular puzzle called Sudoku. The typical Sudoku puzzle grid is a 9-by-9 cells into nine 3-by-3 squares. The rules of the game are: Every row, column, and square (of 3-by-3) must be filled with each of the numbers 1 till 9 and that number cannot appear more than once in any of the row, column, or square. For more general information on Sudoku the reader can access this website[1].

Sudoku has led other researchers to some advances in algorithm design and implementation. This work was largely motivated by the interesting mathematical concepts behind it.

This paper describes the development of a Sudoku solver using MATLAB. The remainder of this paper will cover some research work concerning different solutions for solving Sudoku puzzles, our algorithm solution and GUI implementation will be covered in Section 3. Before concluding the paper in Section 5, some experimental results will be shown in Section 4.

2 Related Work

Sudoku puzzles can be viewed as an interesting problem for different fields of mathematics, computer science, artificial intelligence, physics, and others. There are many researchers from these fields who have proposed algorithms for solving Sudoku puzzles.

In [5] the authors are proposing a search based solution by using some heuristic factors in a modified steepest hill ascent. Some researchers, as found in [6], are suggesting the design of a genetic algorithm by representing the puzzle as a block of chromosomes, more precise as an array of 81 integers. Any crossover appears between the 3x3 grids and any mutations occur only inside the 3x3 grids. From the experiments done in [6] the algorithm based on genetic algorithms performs well though is does not solve all cases. Geem in [3] is proposing a Sudoku solver model based on harmony search that mimics the characteristics of a musician. As mentioned in [4] the performance of the algorithm is not that good and it solves puzzles in less than 40 seconds and 300 iterations. Santos-Garcia and Palomino are suggesting in [9] a method

[1]http://goo.gl/IIf0r

for solving Sudoku puzzles using simple logic with rewriting rules to mimic human intelligence. This method works but, as the author admits, it is not performing well when compared with other techniques. Others like in [10] are suggesting neural networks by modelling an energy driven quantum (Q'tron) neural network to solve the Sudoku puzzles.

In [1], Barlett and Langville are proposing a solution based on binary integer linear programming (BILP). To formulate in a simple way the method, we can say that it uses binary variables to pick a digit for any cell in a Sudoku puzzle. The model in [1] is implemented also in MATLAB.

In [8], Russell Norvig is suggesting a solution based on backtracking. The algorithm is a combination of constraint propagation and direct search by assigning a value to a square on the grid and then propagating these constraints to other squares.

We have decided to solve the puzzles using a constraint-propagation algorithm as in [8] because of the algorithm performance and simplicity, as demonstrated by Norvig in his paper. We will present in the next section our own version of the algorithm implemented in Matlab using constraint propagation.

3 Implementation

3.1 Graphical User Interface

A graphical user interface (GUI) is a collection of visual components that allows the user to interact with a computer program. The usage of buttons or similar components, facilitate the interaction with an application without knowing the underlying program. In contrast, in a command-line based application the user needs to type instructions to perform a task [7]

The implementation of the GUI for this project was done using MATLAB's Graphical User Interface Environment (GUIDE). This environment allows the creation of different layouts by means of drag and drop components. Each component has one or more *callback functions* whose purpose is to execute a set of instructions based on the user's input. An example of this type of functions can be a key press or a mouse click.

Additionally, each component comprises a list of properties that can be edited in order to modify its appearance. For example, one can edit a component's color, size and position on the layout. In fact, one of the main purposes of the callback functions is to update the component's properties.

3.1.1 Sudoku GUI

There are various components that can be used, such as textboxes, push buttons and menus among others. However, for the development of this project only a few different components were used in order to create the GUI of the Sudoku solver.

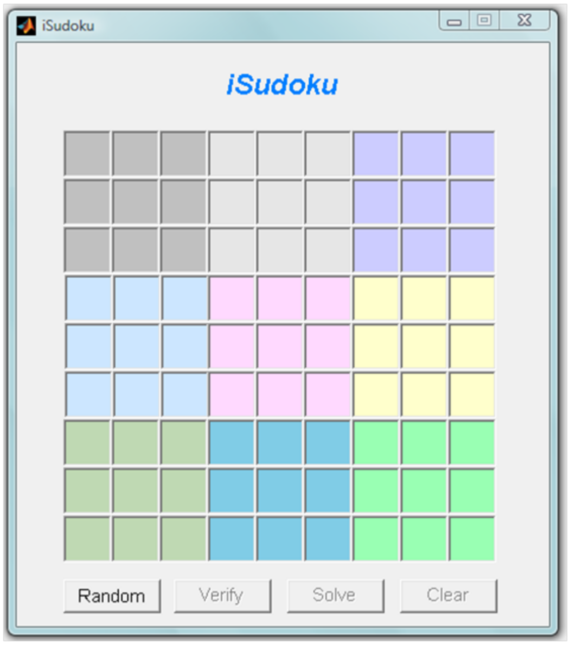
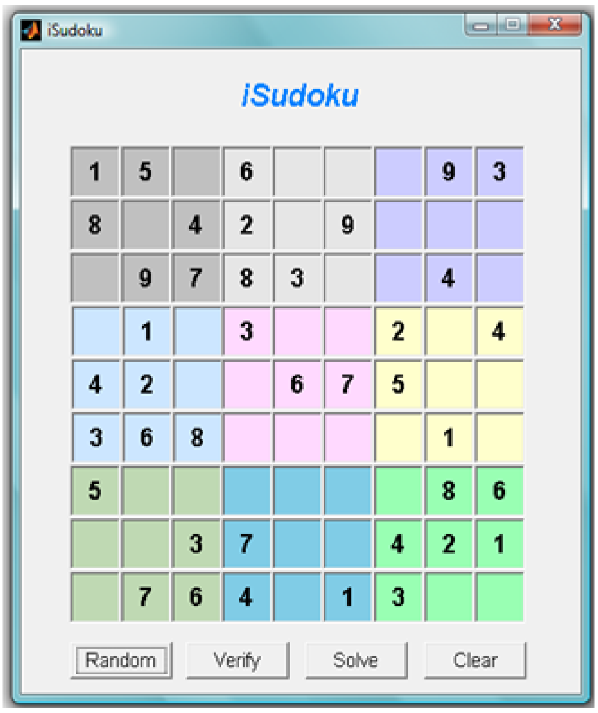


Figure 1: Sudoku GUI at initial state

The Sudoku board is represented by 81 textboxes, where each 3 by 3 sub square uses a different background color in order to be distinguished from each other. Moreover, four push buttons are used to achieve the program's functionality. These buttons are called random, solve, verify and clear. Figure 1 illustrates the Sudoku's GUI at its initial state.

The random button creates and displays a random game on the board. This is done by selecting a game from a database of several games. This button is the only one enabled when the application is started. Once a game has been displayed, the remaining buttons get enabled and only the empty cells can be edited. This means, that the hint cells cannot be modified by the user. An example of a random game is depicted at Figure 2, where its hint cells are displayed in black.



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

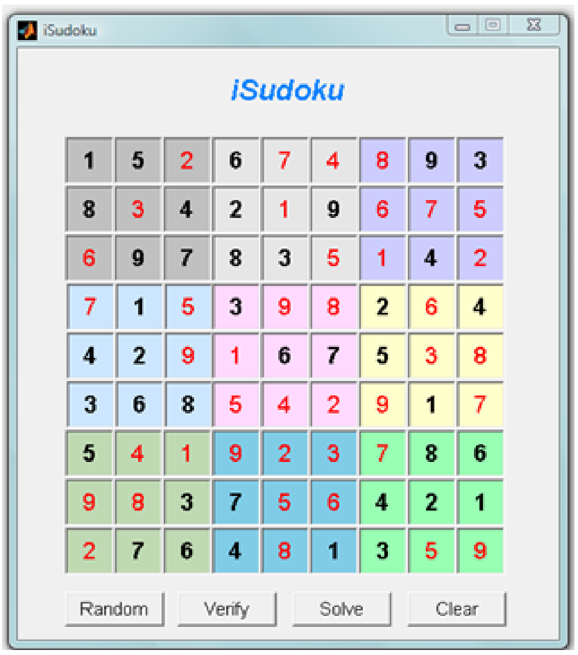
Figure 2: A random game and its hint cells

The solve button solves the current game and displays the solution on the board. This is achieved in three main steps:

1. Read the current game from the board and generate a numerical matrix of 81 elements, where the empty cells are substituted by zeros. A validation of the input data is performed in order to avoid sending invalid values to the algorithm. Only integer values from one to nine can be inserted in the cells.

2. Execute the Sudoku solver function, *iSudokuALG*, using the numerical matrix as an input.

3. Retrieve the solution provided by the Sudoku solver function and populate the board. The hint cells remain in black while the solution cells are highlighted in red. Figure 3 depicts a solution of a game.

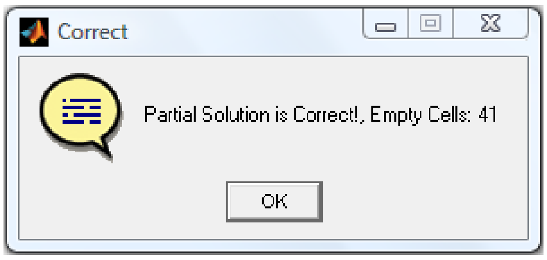


iSudoku

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

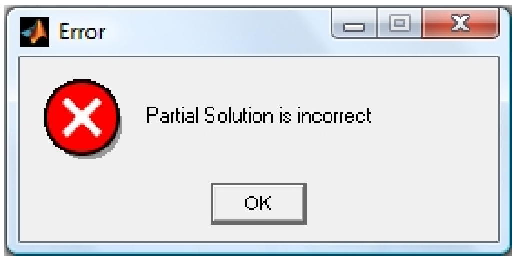
Figure 3: A solved game

The verify button examines the correctness of either a partial game or a complete game. If a partial or a complete game is detected to be incorrect, the program will display a pop-up window with an error message. Otherwise, a different pop-up window will display a message stating that the game is correct and in case of a partial game, the message will also state the number of remaining empty cells. An example of these type of messages is illustrated at Figure 4(a) and Figure 4(b)



Partial Solution is Correct!, Empty Cells: 41

(a) Correct Game



Partial Solution is incorrect

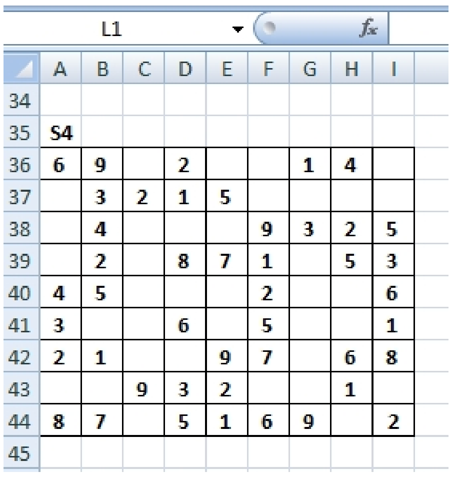
(b) Incorrect Game

Figure 4: GUI Message Boxes

Finally, the clear button implements the simplest functionality as it only clears the board and disables all the buttons except the random one, returning the program to its initial state.

3.1.2 Game Database

In order to have different games to be solved by the program an Excel database was created. This database includes games of different levels of difficulty, which means that the number of hints varies from game to game. All the games are contained within an Excel worksheet that is read when the GUI is initialized. An extract of the game database can be seen at Figure 5.



	A	B	C	D	E	F	G	H	I
34									
35	S4								
36	6	9		2			1	4	
37		3	2	1	5				
38		4				9	3	2	5
39		2		8	7	1		5	3
40	4	5				2			6
41	3			6		5			1
42	2	1			9	7		6	8
43			9	3	2			1	
44	8	7		5	1	6	9		2
45									

Figure 5: Extract of the Excel game database

The reading process is performed by means of MATLAB's built-in function *xlsread*. This function receives as input the name of the Excel file and the worksheet to be read. The output of this function is stored into a cell matrix that later on is used by the random button to select a game. This function is represented by the following:

```
% Read predefined games from the input spreadsheet
[num, cellMat]= xlsread('sudoku.xls', 'Games');
```

3.2 An Algorithm for Solving Sudoku Puzzles

The algorithm was implemented in MATLAB using direct puzzle search and constraint propagation. The constraint propagation was implemented in such a manner that:

- When a value is assigned to a square that same value cannot be used as a possible assignment in all related cells;
- If a cell has only one single value for possible assignment, that value is immediately assigned.

Our MATLAB program has only 4 steps:

1. Find all the possible values for all empty cells;
2. If there is a single possible value, we assign that value to the cell;
3. Propagate constraints to other cells;
4. If all the cells have more than one possible value we fill in a tentative value for that cell.

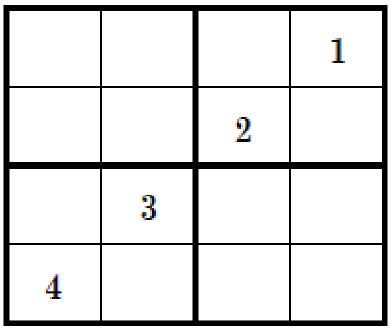
This algorithm, though it is simplistic, it should perform fairly well because of the algorithm's nature, as described in [8]. The process is performed by *iSudokuALG* function. This function receives as an input parameter the cell matrix A and gives as output a solved puzzle in the form of a matrix. This is represented by

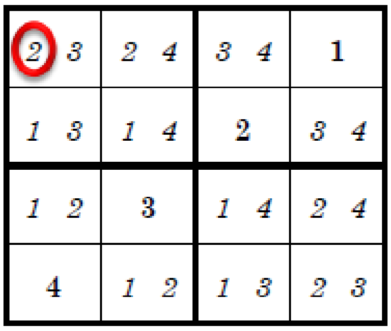
```
% Read predefined games and outputs the solved puzzle
function [A]= iSudokuALG(A)
```

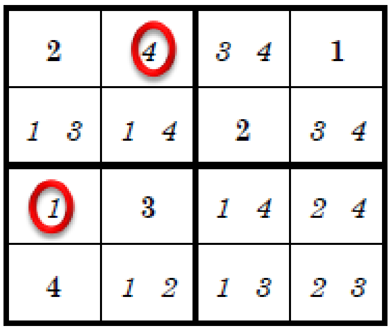
Also we have implemented a function in order to perform a correctness verification function. This function has as input the cell matrix A which contains the current puzzle. The output of the function is stored in variable *val* which can have two values: 0 if the puzzle is correct and 1 otherwise. We represented the verification function as

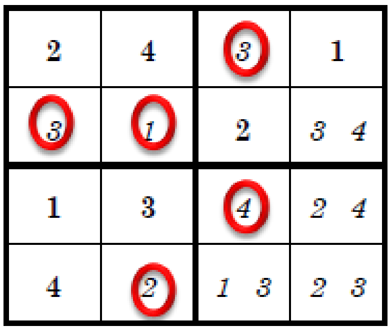
```
% Puzzle verification function
function [val]=verific(A)
```

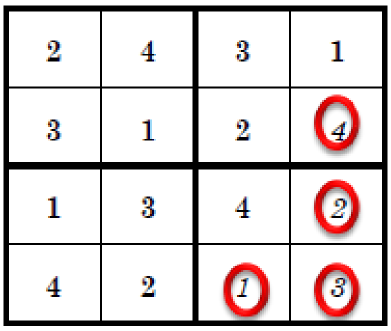
To see how our program works, we will use a simpler 4-by-4 grid with 2-by-2 blocks. As mentioned in [2] these kinds of puzzles are called Shidoku ("Shi" means "four" in Japanese). Figure 6(a) shows a Shidoku puzzle. Figure 6(b) through Figure 6(f) shows how we get to a solution by using our algorithm.


(a) A Shidoku puzzle


(b) The possible candidates for all squares


(c) Inserting value 2 and constraint propagation


(d) Constraint propagation


(e) Insert the remaining values to complete the puzzle

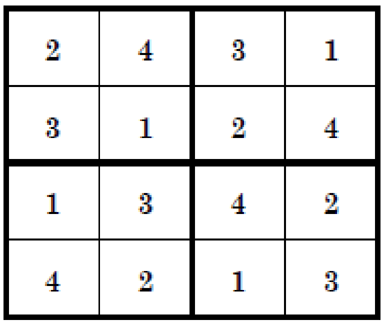

(f) The puzzle completed

Figure 6: The algorithm for solving a simple Shidoku puzzle

In Figure 6(b), the possible assignment, are represented as smaller digits.

Also for example in Figure 6(b) line 1, row 1, we show that we have two possible values to choose from. If a cell contains only one candidate it is filled immediately.

4 Experimental Results

In this section we are presenting some experimental results using different input puzzles and we will describe how our MATLAB implementation behaves to various situations.

4.1 User Interface Testing

After developing the user interface, the functionality associated to each GUI component was tested. More specifically, it was verified the absence of runtime errors shown in MATLAB's command window when the user executes an action. For example, when launching the application or when clicking any of the available buttons, no errors are displayed. Additionally, the program flow was tested to ensure that multiple Sudoku games can be solved without closing and opening the application when a game is finished.

In order to verify the correct operation of the *random* and *solve* buttons, it was checked that all of the Sudoku games in the database could be randomly selected and solved correctly. Similarly, the *verify* button was tested by manually entering incorrect numbers and characters in the Sudoku board. The purpose of this test was to verify that the pop-up error messages are displayed when the proposed solution is incorrect. Finally, the *clear* button was verified by checking that all of the cells in the board are cleared when the button is pushed.

The only issue found during the verification phase for the GUI was related to MATLAB's xlsread function when using Microsoft Excel 2010. Apparently this is a well documented problem [2] regarding how the latest version of Excel handles COM objects and activeX commands which are used by this function . Nevertheless, the xlsread function works correctly when using previous versions of Excel, like version 2007.

[2] http://www.mathworks.com/matlabcentral/newsreader/view_thread/287717#766161

sq_{empty}	s_i	t_e $(seconds)$
37	4	0.064
42	4	0.096
36	2	0.174
41	9	0.622
37	3	0.156
36	3	0.188
41	5	0.276
42	4	0.422
41	4	0.294
37	3	0.344

Table 1: Experimental results for different Sudoku puzzles

4.2 Algorithm Testing

All the algorithm source code was compiled and build using Matlab R2007a and is available online[3]. The test system was a HP G7035EA laptop with 1.86 GHz Celeron Processor, 1 GB RAM and Windows XP as operating system.

In order to verify and test our algorithm we have defined some test parameters. This parameters are t_e (algorithm's execution time in seconds), sq_{empty} (number of empty cells in a puzzle) and s_i (steps or iterations done by the algorithm, meaning that one iteration starts when the solver begins the puzzle from the first cell and finishes when the last cell is computed). Table 1 shows the results of our experiments for different sq_{empty} in relation to t_e and s_i. We can say that t_e is fluctuating in the interval $[0.06, 0.62]$ and sq_{empty} for every puzzle is not a predictable measure of performance related to t_e. Also our algorithm is solving Sudoku puzzles with 40 empty cells in average and s_i is varying between 2 and 9. After analysing all our results we can say that for relatively easy Sudoku puzzles our algorithm is performing quite well.

[3]http://goo.gl/96pCS

5 Conclusions

Implementing a Sudoku solver in MATLAB allowed us to use many of the tools and built-in functions presented during the Numerical Methods course. For example, the utilization of loops, if-else statements and the manipulation of matrices and vectors. Nonetheless, we also learnt new functions by ourselves in order to enhance the application. For instance, we learnt to design GUIs, use pop-up windows to display messages and also importing data from Excel.

Nowadays, there are extensive studies regarding the mathematics of Sudoku as well as many different algorithms to solve the puzzles. Some algorithms are designed to solve the puzzles as quick as possible while some others are designed to solve them as efficiently in terms of computational power and memory. However, finding a suitable algorithm to solve any particular Sudoku game proved to be difficult.

The combination of a simple, yet effective algorithm with a graphical user interface allowed us to generate games, solve them and verify the given solutions in a simple and quick way. Additionally, the good communication and coordination among the team members made possible the completion of the project before the established deadline.

A GUI Source Code

```
% File: iSudokuGUI.m
% Description: This file implements the graphical user interface for the
% iSudoku solver. The user interface consists of 81 text boxes to represent
% the 9 by 9 sudoku board, 1 label fot the GUI title (iSudoku) and 4
% buttons: Random, Verify, Solve and Clear.
% Author(s): Andrea Garcia, Ivan Castro
% Mail(s): aga09001@student.mdh.se, ico09002@student.mdh.se
% Group number: A-3

function varargout = iSudokuGUI(varargin)
% Last Modified by GUIDE v2.5 23-Feb-2011 23:01:41

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @iSudokuGUI_OpeningFcn, ...
                   'gui_OutputFcn',  @iSudokuGUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before iSudokuGUI is made visible.
function iSudokuGUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to iSudokuGUI (see VARARGIN)

% Choose default command line output for iSudokuGUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes iSudokuGUI wait for user response (see UIRESUME)
% Read predefined games from the input spreadsheet
[num, cellMat]= xlsread('sudoku.xls', 'Games');
set(handles.RandomBtn,'UserData', cellMat);

%----------------------------------------------------------------------
% --- Outputs from this function are returned to the command line.
function varargout = iSudokuGUI_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

%-----------------------------------------------------------------------
% Executes on button press in RandomBtn.
% This button is in charge of generating a random dudoku game and display
% it in the user interface. This button also enables the "verify", "solve"
% and "clear" button.
%-----------------------------------------------------------------------
function RandomBtn_Callback(hObject, eventdata, handles)
% hObject    handle to RandomBtn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get a sudoku game using a random number
cellMat = get(handles.RandomBtn,'UserData');
randNum = ceil(20.*rand(1));
srtTarg = ['S' num2str(randNum)];
[cRowSize cColSize] = size(cellMat);

% Find the random game in from the database
rowOffset = 1;
for cRowInd = 1: cRowSize
    if (strcmp(srtTarg, cellMat(cRowInd,1)) == 1)
        rowOffset = cRowInd;
        break;
    end
end

%Fill-up the board using the game matrix
for rowInd = 1:9
    for colInd = 1:9
        cName = ['c' num2str(rowInd) num2str(colInd)];
        cValue = strtrim(cellMat{rowOffset+rowInd, colInd});
        % Set a number in the appropriate position in the board
        if strcmp(cValue, '')
            expr = ['set(handles.' cName ', ''String'', ''' cValue ''',...
            ''FontWeight'', ''normal'', ''Enable'', ''on'',...
            ''ForegroundColor'', [' num2str([0 0 0]) '])'];
        else
            expr = ['set(handles.' cName ', ''String'', ''' cValue ''',...
            ''FontWeight'', ''bold'', ''Enable'', ''inactive'',...
            ''ForegroundColor'', [' num2str([0 0 0]) '])'];
        end
        eval(expr);
    end
end

% Enable the solve, verify and clear buttons
set(handles.SolveBtn, 'Enable', 'on');
set(handles.ClearBtn, 'Enable', 'on');
set(handles.verifyBtn, 'Enable', 'on');


%-----------------------------------------------------------------------
% Executes on button press in SolveBtn.
% This button is in charge of solving the sudoku game displayed on the user
% interface using the iSudokuAlg function.
%-----------------------------------------------------------------------
function SolveBtn_Callback(hObject, eventdata, handles)
% hObject    handle to SolveBtn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% First Read the board:
[validMatrix toSolveMatrix hintMatrix] = readBoard(handles);

% At this point "toSolveMatrix" holds the Sudoku game that needs to be
% solved automatically.

if (validMatrix == 1) % % The board has only valid characters (1-9)
    % Verify that the solution / partial solution is correct
    incorrectGame = verific(toSolveMatrix);
    if (incorrectGame == 0)
        % Solve the Game using the iSudokuAlg function:
        solvedMatrix = iSudokuALG(toSolveMatrix);
        % Populate the GUI with the solution obtained from
        % the solving algorithm:
        for rowInd = 1:9
            for colInd = 1:9
                cName = ['c' num2str(rowInd) num2str(colInd)];
                cValue = num2str(solvedMatrix(rowInd, colInd));
                if (hintMatrix(rowInd,colInd) == 1)
                    % This position corresponds to a hint
                    expr = ['set(handles.' cName ', ''String'',...
                    ''' cValue ''')'];
                else
                    % This position is part of the solution
                    expr = ['set(handles.' cName ', ''String'', ''' ...
                    cValue ''', ''ForegroundColor'', [' num2str([1 0 0])...
                    '], ''Enable'', ''inactive'')'];
                end
                eval(expr);
            end
        end
    else
        % The current game is incorrect, display an error message.
        msg = 'Invalid Game!';
        h = msgbox(msg,'Error','error', 'replace');
    end
end

%--------------------------------------------------------------------
% Executes on button press in ClearBtn.
% This button just clears the board and resets the solve button and the
% verify button to their default state, disabled.
%--------------------------------------------------------------------
function ClearBtn_Callback(hObject, eventdata, handles)
% hObject    handle to ClearBtn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%Clear the board
for rowInd = 1:9
    for colInd = 1:9
        cName = ['c' num2str(rowInd) num2str(colInd)];
        cValue = '';
        expr = ['set(handles.' cName ', ''String'', ''' cValue ...
        ''',''FontWeight'', ''normal'', ''Enable'', ''on'')'];
        eval(expr);
    end
end
% Disable the Solve and Verify buttons:
set(handles.SolveBtn, 'Enable', 'off');
set(handles.verifyBtn, 'Enable', 'off');

%--------------------------------------------------------------------
% Executes on button press in verifyBtn.
% This button verifies that the solution (partial or complete) displayed on
% the board is correct. If it is correct, a pop-up window will display that
% the solution is correct. Otherwise it will display an error message.
%--------------------------------------------------------------------
function verifyBtn_Callback(hObject, eventdata, handles)
% hObject    handle to verifyBtn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% First Read the board:
[validMatrix toSolveMatrix hintMatrix] = readBoard(handles);
numberOfHints = sum(sum(hintMatrix));

if (validMatrix == 1)
    % Verify that the solution / partial solution is correct
    incorrectGame = verific(toSolveMatrix);

    if (all(all(hintMatrix)) == 1 ) % Complete Game
        if (incorrectGame == 1)
            msg = 'Solution is incorrect';
            h = msgbox(msg,'Error','error', 'replace');
        else
            msg = 'Solution is Correct!, Game Completed';
            h = msgbox(msg,'Correct','help', 'replace');
        end
    else % Partial Game
        if (incorrectGame == 1)
            msg = 'Partial Solution is incorrect';
            h = msgbox(msg,'Error','error', 'replace');
        else
            toGo= 81 - numberOfHints;
            msg = ['Partial Solution is Correct!, Empty Cells: '...
            num2str(toGo)];
            h = msgbox(msg,'Correct','help', 'replace');
        end
    end
end

%-----------------------------------------------------------------------
% This function reads the board and generates a numeric matrix with the
% game and a hint matrix. This function is used by both the solve and the
% verify buttons
%-----------------------------------------------------------------------
function [valid toSolveMatrix hintMatrix] = readBoard(handles)

valid = 1;
toSolveMatrix = zeros(9,9);
hintMatrix = zeros(9,9); % This matrix holds the position of the hints
                         % or given numbers before solving the game
for rowInd = 1:9
    for colInd = 1:9
        cName = ['c' num2str(rowInd) num2str(colInd)];
        expr = ['cValue = get(handles.' cName ', ''String'');'];
        eval(expr);

        %Validate data correctness prior solving the game
        %Only values from 1-9 should be entered, an empty cell is valid
        if (isempty(cValue) == 0)
            if( isempty(str2num(cValue)) == 1)
                msg = ['Non numeric value found at row: ' num2str(rowInd)...
                ', column: ' num2str(colInd)];
                h = msgbox(msg,'Error','error', 'replace');
                valid = 0;
                break;
            elseif (length(cValue) ~= 1 ||...
                str2num(cValue) < 1 ||...
                str2num(cValue) > 9 )
                msg = ['Invalid input found at row: ' num2str(rowInd) ...
                ', column: ' num2str(colInd)];
                h = msgbox(msg,'Error','error', 'replace');
                valid = 0;
                break;
            end

            %Cell data is valid, add it to the matrix
            toSolveMatrix(rowInd,colInd) = str2num(cValue);
            hintMatrix(rowInd,colInd) = 1;
        else
            %Change the empty cell to zero and add it to the matrix
            toSolveMatrix(rowInd,colInd) = 0;
        end

    end

    if (valid == 0)
        break;
    end
end
```

B Puzzle Verification Function Source Code

```
% File: verific.m
% Description: This file implements the corectness verification
% function. It has as input the cell matrix A which contains
% the current puzzle. The output of the function is stored in
% variable val which can have two values: 0 if the puzzle is
% correct and 1 otherwise.
% Author(s): Raluca Marinescu, Eduard Enoiu
% Mail(s): rmu09001@student.mdh.se, eeu09001@student.mdh.se
% Group number: A-3

function [val]=verific(A)

flag=0;
for i=1:9
    for nr=1:9
        count=0;
        for j=1:9
            if A(i,j)==nr count=count+1;
            end
        end
        if count>1 flag=1;
        end
    end
end

% if flag==1
%     fprintf('Same number on a row.\n')
%     flag=0;
% end

for i=1:9
    for nr=1:9
        count=0;
        for j=1:9
            if A(j,i)==nr count=count+1;
            end
        end
        if count>1 flag=1;
        end
    end
end

% if flag==1
%     fprintf('Same number on a column.\n')
%     flag=0;
% end
```

```
for n=1:3
   for nr=1:9
       count=0;
       for i=((n-1)*3+1):(n*3)
           for j=((n-1)*3+1):(n*3)
                if A(i,j)==nr count=count+1;
                end
            end
       end
       if count>1 flag=1;
       end
    end
end

if flag==0 val=0;
else val=1;
end
```

C Puzzle Solver Alghorithm Source Code

```
% File: iSudokuALG.m
% Description: This file impements the algorithm for ISudoku Matlab
% app. The process is performed by SudokuALG function. This
% function receives as an input parameter the cell
% matrix A and gives as output a solved puzzle in the form of a
% matrix.
% Author(s): Raluca Marinescu, Eduard Enoiu
% Mail(s): rmu09001@student.mdh.se, eeu09001@student.mdh.se
% Group number: A-3

function [A]= iSudokuALG(A)
clc;
tic;
[flag1]=verific(A);
if flag1==0 fprintf('The Sudoku Puzzle is correct.\n')

    err_flag=0;
    count_zeros=81;

    for i=1:9
        for j=1:9
            B(i,j)=0;
        end
    end

    step=1;
    while ((err_flag==0)&&(count_zeros>0))
        fprintf('\n Step %d\n', step);
        step=step+1;

        C=B;
        %I look for the postions where there is a single possible value
        %and I put that value in the matrix
        for i=1:9
            for j=1:9
                if (A(i,j)+B(i,j))==0
                    possible=[];
                    for k=1:9
                        B(i,j)=k;
                        flag2=verific(A+B);
                        if flag2==0
                            possible=[possible k];
                        end
                    end
                    B(i,j)=0;
                    if length(possible)==1
               B(i,j)=possible;
```

```
           fprintf('Value %d in A(%d,%d)--singleton.\n', B(i,j), i, j);
                end
            end
        end
    end

    %If we don't find one position with a single possible value
    %I put a number (correct) in an empty cell
    if C==B
        count2=0;
        x=1;
        y=1;
        while (count2==0)
            if (A(x,y)+B(x,y))==0
                for k=1:9
                    B(x,y)=k;
                    flag3=verific(A+B);
                    if flag3==0
                        count2=1;
                        fprintf('Value %d in A(%d,%d).\n', B(x,y), x, y);
                        break
                    else
                    B(x,y)=0;
                    end
                end
            end

            if(x<9) x=x+1;
            elseif (y<9)x=1; y=y+1;
            else break
            end

        end
    end

    %if the matrix is the same then I have an error
    if (B==C) err_flag=1;
    end

    %I count the empty cells to know when the puzzle is solved
    count_zeros=0;
    for i=1:9
        for j=1:9
            if ((A(i,j)+B(i,j))==0) count_zeros=count_zeros+1;
            end
        end
    end

end
```

```
    %We save the result; check if it is correct
    A=A+B
    [flag3]=verific(A);
    if flag3==0 fprintf('The Sudoku Puzzle is correct.\n')
    else fprintf('The Sudoku Puzzle is incorrect.\n')
    end
    else fprintf('The Sudoku Puzzle is incorrect.\n')
end
time=toc;
fprintf('\n\n Execution Time: %f \n', time);
```

References

[1] A.C. Bartlett and A.N. Langville. An integer programming model for the Sudoku problem. *Preprint, available at http://www. cofc. edu/˜ langvillea/Sudoku/sudoku2. pdf*, 2006.

[2] JF Crook. A pencil-and-paper algorithm for solving Sudoku puzzles. *Notices of the AMS*, 56(4):460–468, 2009.

[3] Z. Geem. Harmony search applications in industry. *Soft Computing Applications in Industry*, pages 117–134, 2008.

[4] R.C. Green II. Survey of the Applications of Artificial Intelligence Techniques to the Sudoku Puzzle. 2009.

[5] SK Jones, PA Roach, and S. Perkins. Construction of heuristics for a search-based approach to solving Sudoku. *Research and Development in Intelligent Systems XXIV*, pages 37–49, 2008.

[6] T. Mantere and J. Koljonen. Solving, rating and generating Sudoku puzzles with GA. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 1382–1389. Ieee.

[7] Mathworks. Creating graphical user interfaces http://www.mathworks.com/help/techdoc/creating_guis/bqz79mu.html, March 2011.

[8] P. Norvig. Solving every sudoku puzzle. *Preprint*.

[9] G. Santos-Garcia and M. Palomino. Solving Sudoku puzzles with rewriting rules. *Electronic Notes in Theoretical Computer Science*, 176(4):79–93, 2007.

[10] T.W. Yue and Z.C. Lee. Sudoku Solver by Q'tron Neural Networks. *Intelligent Computing*, pages 943–952, 2006.