

BIE-ZUM Semestral Work

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

Sergio Teodoro Castellano Betancor



Index.

Description of the task.	3
My work.....	3
Graphic iterations.....	4
Explanation of the code.....	9
Results.	13
References.....	15



Description of the task.

Sudoku at 100%

Consider the well-known SUDOKU filler, where we have to add digits 1 to 9 to a 9x9 grid so that different conditions for the diversity of digits are met. We will not describe the rules in detail, we assume that they are known. Yours the task is to solve the replenisher using artificial intelligence techniques, but so that it is done as quickly and guaranteed as possible, ie the solving algorithm always ends and gives the correct solution.

It is possible to use the approach from the CSP, ie to model Sudoku as a CSP and then perform a search over the CSP

My work.

I have decided to use backtracking techniques to complete the previous task explained above this section.

In order to solve this problem, I have used the following logic:

- Process sudoku board
- Find empty position
- Find number which could fit in the position and place it in the empty position
 - If there is no number which could fit in the position
 - Then return to the previous empty position and place next valid number.

To exemplify this more graphically I will show you graphically some iterations of how my program will solve this problem.



Graphic iterations.

Let's start with a board with this set of numbers:

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

Now let's follow the pseudocode and let's find an empty position and place the first possible number

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



Now let's advance to the next empty square again and place the next valid number

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


Now let's advance the next empty position and place the first possible number

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



Now let's advance to the next empty square again and place the next valid number

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



As we can see the only number we can place in the same row is 8 but we can't place number 8 due to the column constrain, in this case is where the backtracking part of the code starts working, we will not write number 8 and we will check the next possible number in the previously exanimated position.

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



As we see there is no more possible numbers in the previous position, so we simply repeat this backtracking process with the previous position too

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

Now let's advance the next empty position and place the first possible number.

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



As we can see we can't again place any number due to the game constraints, so let's place the next possible number in the previous position.

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

And this is how the algorithm will work until we find a valid solution



Explanation of the code

First, we need to know how to load sudokus from file which are going to follow the next format:

```
530070000
600195000
098000060
800060003
400803001
700020006
060000280
000419005
000080079
```

To make our work easier have developed the next functions which we will analyse as one whole thing

```
1  def load_sudoku(file_name):
2      f = open(file_name)
3      sudoku = f.read()
4      f.close()
5      return conver_sudoku(sudoku)
6
7  def conver_sudoku(board):
8      conver_sudoku = []
9      lines = board.splitlines()
10     for line in lines:
11         conver_sudoku.append(list(line))
12     #Cast String to int
13     aux = []
14     for i in range(len(conver_sudoku)):
15         aux1 = []
16         for j in range(len(conver_sudoku[0])):
17             aux1.append(int(conver_sudoku[i][j]))
18         aux.append(aux1)
19     return aux
```



With the function “def load_sudoku(file_name)” we basically store the data of the file in a variable, and then we call the function “conver_sudoku(board)” which will create the 2 dimensions array that will store the sudoku, but we also need to cast the data to int type because if we don't do that all our function are not going to be able to operate with strings.

Secondly, the first function we are going to need is find an empty position, that's why we will develop “def find_empty(bo)” which will get the board as a parameter of the function.

```
78 def find_empty(bo):
79     for i in range(len(bo)):
80         for j in range(len(bo[0])):
81             if bo[i][j] == 0:
82                 return (i, j) # row, col
83
84     return None
```

As we see this simple function just goes through our board which is going to be a 2 dimensional array, as soon as it found an empty space (symbolized by 0 in our array) it will return a tuple with the coordinates of the empty position.



Then, we need to find a valid number for a position, for that we will develop the method “def valid(bo, num, pos)” which will get the board, the number we try to check it is valid or not and the position in which this process is being evaluated.

```
40 def valid(bo, num, pos):
41     # Check row
42     for i in range(len(bo[0])):
43         if bo[pos[0]][i] == num and pos[1] != i:
44             return False
45
46     # Check column
47     for i in range(len(bo)):
48         if bo[i][pos[1]] == num and pos[0] != i:
49             return False
50
51     # Check box
52     box_x = pos[1] // 3
53     box_y = pos[0] // 3
54
55     for i in range(box_y*3, box_y*3 + 3):
56         for j in range(box_x * 3, box_x*3 + 3):
57             if bo[i][j] == num and (i,j) != pos:
58                 return False
59
60     return True
```

With first for loop, we check if the variable “num” is similar to any position in the same row, in that case we return false because due to the game constraints we cannot accept it.

With the second for loop, we do the same but, in the row.

Finally in the third part of the function we will check if there is any coincidence in bloc, in case we go trough all this check ups we will return true and consider the analysed number true.



Last but not least, we will develop the function “def solve(bo)” where we will implement the pseudocode explained before.

```
50  def solve(bo):
51      find = find_empty(bo)
52      if not find:
53          return True
54      else:
55          row, col = find
56
57      for i in range(1,10):
58          if valid(bo, i, (row, col)):
59              bo[row][col] = i
60              if solve(bo):
61                  return True
62              bo[row][col] = 0
63
64      return False
```

First, we try to find an empty position, if there is no then it means that the sudoku has been completed otherwise we take the empty position found and start with the recursive process for each empty position.



Results.

Here there are some screenshots of the result of the code once it is runned with different sudokus samples.

```
C:\Users\scastr\AppData\Local\Programs
7 8 0 | 4 0 0 | 1 2 0
6 0 0 | 0 7 5 | 0 0 9
0 0 0 | 6 0 1 | 0 7 8
- - - - -
0 0 7 | 0 4 0 | 2 6 0
0 0 1 | 0 5 0 | 9 3 0
9 0 4 | 0 6 0 | 0 0 5
- - - - -
0 7 0 | 3 0 0 | 0 1 2
1 2 0 | 0 0 7 | 4 0 0
0 4 9 | 2 0 6 | 0 0 7
-----
Solution of the sudoku:
7 8 5 | 4 3 9 | 1 2 6
6 1 2 | 8 7 5 | 3 4 9
4 9 3 | 6 2 1 | 5 7 8
- - - - -
8 5 7 | 9 4 3 | 2 6 1
2 6 1 | 7 5 8 | 9 3 4
9 3 4 | 1 6 2 | 7 8 5
- - - - -
5 7 8 | 3 9 4 | 6 1 2
1 2 6 | 5 8 7 | 4 9 3
3 4 9 | 2 1 6 | 8 5 7
```



C:\Users\scastr\AppData\Local

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

```
- - - - -
0 5 0 | 1 0 7 | 0 9 0
0 0 3 | 9 0 5 | 1 0 0
0 4 0 | 3 0 2 | 0 5 0
```

```
- - - - -
1 0 0 | 0 3 0 | 0 0 2
5 0 0 | 2 0 4 | 0 0 9
0 3 8 | 0 0 0 | 4 6 0
```

Solution of the sudoku:

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

```
- - - - -
8 5 2 | 1 4 7 | 6 9 3
6 7 3 | 9 8 5 | 1 2 4
9 4 1 | 3 6 2 | 7 5 8
```

```
- - - - -
1 9 4 | 8 3 6 | 5 7 2
5 6 7 | 2 1 4 | 3 8 9
2 3 8 | 5 7 9 | 4 6 1
```

C:\Users\scastr\AppData\Local

```
5 3 0 | 0 7 0 | 0 0 0
6 0 0 | 1 9 5 | 0 0 0
0 9 8 | 0 0 0 | 0 6 0
```

```
- - - - -
8 0 0 | 0 6 0 | 0 0 3
4 0 0 | 8 0 3 | 0 0 1
7 0 0 | 0 2 0 | 0 0 6
```

```
- - - - -
0 6 0 | 0 0 0 | 2 8 0
0 0 0 | 4 1 9 | 0 0 5
0 0 0 | 0 8 0 | 0 7 9
```

Solution of the sudoku:

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

```
- - - - -
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
```

```
- - - - -
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
```



References.

- [1] BIE-ZUM Introduction to Artificial Intelligence Lectures
<https://courses.fit.cvut.cz/BI-ZUM/en/lectures/index.html>

