ASSIGNMENT-2

SUDOKU-PUZZLE (python)

SAI AKA(1910110333) & KONDURU SANDILYA(1910110205)

## Abstract

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic sudoku, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid contain all of the digits from 1 to 9. This sudoku problem can be solved by computers in various methods like Rule based method, backtracking, Boltzmann machines. Here we solved sudoku as a constraint satisfaction problem(CSP) approach.


***Keywords:*** CSP, backtracking.

## ***INTRODUCTION:***

A Sudoku square of order n consists of n 4 variables formed into a n 2 × n 2 grid with values from 1 to n 2 such that the entries in each row, each column and in each of the n 2 major n × n blocks are all different.

Currently, only Sudoku problems of order 3 (9 × 9 grid) are widely used. It was claimed that there are 6,670,903,752,021,072,936,960 valid Sudoku squares of order 3.

## ***CONSTRAINTS IN SUDOKU PROBLEM:***

1. Cell Constraint: All cells S ij ∈ S may contain no more than 1 value AND The value must be between 1 and n.

2. Row Constraint: All values in rows S i ∈ S must be unique, i.e. If ∃S ij = x then ∃S ij 0 s.t. S ij 0 = x.

3. Column Constraint: All values in columns S j ∈ S must be unique, i.e. If ∃S ij = x then ∃S i ‘ j’ s.t. S i 0 j = x.

4. Box Constraint: We define a box as being a sqrt(n) x sqrt(n) sub-grid of S s.t. the top left cell of the sub-grid is always $S_{ij}$ where (i − 1) % sqrt(n) == 0 and (j − 1) % sqrt(n) == 0. We denote a box with the top-left cell $S_{ij}$ as $B_{ij}$ .

## ***RULE BASED APPROACH:***

This algorithm builds on a heuristic for solving Sudoku puzzles. The algorithm consists of testing a puzzle for certain rules that fills in squares or eliminates candidate numbers. Those rules are listed below:

**1.Naked Single:** This means that a square only have one candidate number

**2. Hidden Single:** If a region contains only one square which can hold a specific number then that number must go into that square.

**3. Naked pair/triple:** Two or three cells in the same region have a union of two or three candidates in common.

**4. Hidden pair/triple pair:** Two or three cells in the same region have the last remaining two or three candidates for that in region in error.

## ***Pseudocode:***

```
Puzzle Solve Sudoku Rulebased(puzzle)
while(true){
//Apply the rules and restart the loop if the rule
//was applicable. Meaning that the advanced rules
//are only applied when the simple rules failes.
```

//Note also that applyNakedSingle/Tuple takes a reference

//to the puzzle and therefore changes the puzzle directly

If (applyNakedSingle(puzzle))

continue

If (applyNakedTuple(puzzle))

continue

break

}

//Resort to backtrack as no rules worked

**Screenshots:**

Initially the given csv is being loaded in the program and the entries in the csv file are converted into strings.

```
import pandas as pd
```

```
sample_data = pd.read_csv('puzzle1(2).csv') # csv file reading
```

```
sample_data
```

```
sample_data.iloc[0:8,2]
```

```
notation = []
```

```
for i in range(8):
    for j in range(7):
        if sample_data.iloc[i,j] == 0:
            sample_data.iloc[i,j] = "x"
        notation.append(sample_data.iloc[i,j])
```

```
sample_data
```

```
values = ''.join(str(v) for v in notation) # all the entries in the csv file are being converted in to string
```

```
values
```

```
def cartesian_product(x,y):

    return [a+b for a in x for b in y]
# takes two iterable values and return the cartesian product in a list
```

Display function basically displays the sudoku game board and the function is given in the following image.

```
# displays the game board

def display_game_board(values):

    print('')

    rows = 'ABCDEFGHI'
    cols = '123456789'
    boxes = cartesian_product(rows, cols)

    width = 1+max(len(values[s]) for s in boxes)
    line = '+'.join(['-'*(width*3)]*3)
    for r in rows:
        print(''.join(values[r+c].center(width)+('|' if c in '36' else '')
                      for c in cols))
        if r in 'CF': print(line)
    return
```

Elimination function eliminates the possible values according to the rules of the sudoku game as discussed above to get a simplified version of the puzzle.

```
# elimination function eliminates the possible values according to the rules to get a simplified version of the puzzle.
def eliminate(Grid):
    for k,v in Grid.items():
        if len(v) != 1: #  checks if the box needs elimination
            peers = peer_dict[k]    # takes all the peers
            peer_values = set([Grid[p] for p in peers if len(Grid[p]) == 1])
            Grid[k] = ''.join(set(Grid[k]) - peer_values)
    return Grid
```

Next comes the choice function which checks out possibility of placing the number if it had left with only one choice.

```
# checks the places the number if the there is only one choice to place it
def choice(Grid):
    for unit in unit_list:
        for num in '123456789':
            num_places = [box for box in unit if num in Grid[box]]
            if len(num_places) == 1:
                Grid[num_places[0]] = num
    return Grid
```

As discussed in the beginning about naked pairs or triples the following function considers the situation where two or more values are in common with any other row or column.

```
# Kindly check the documentation to know about naked_pairs

def naked_pairs(Grid):
    for unit in unit_list:

        # slice the Grid to contain only the boxes in the unit
        values = dict([[box, ''.join(sorted(Grid[box]))] for box in unit])

        # find all the items with 2-digit values
        double_digits = dict([[box, values[box]] for box in values if len(values[box])==2])

        # check if any of those 2-digit values occur exactly twice
        double_digits_occuring_twice = dict([[box, val] for box, val in double_digits.items() if list(double_digits.value

        if len(double_digits_occuring_twice.items()) != 0:
            # reverse the dictionary to get the key-pairs easily
            reverse_dict = {}
            for k, v in double_digits_occuring_twice.items():
                reverse_dict.setdefault(v, []).append(k)

            # it is a list of 2 items(keys | boxes) only
            naked_pairs = list(reverse_dict.items())[0][1]

            # remove the naked_pairs digits from other boxes in the unit
            for k,v in values.items():
                if (k not in naked_pairs) and (len(v) > 1):
                    values[k] = ''.join(set(values[k]) - set(values[naked_pairs[0]]))

        # replace the values in Grid with the updated values
        for k,v in values.items():
            Grid[k] = v

    return Grid
```

The run function finds out when the program gets stuck. This can be found out by Just checking the board values before and after eliminating and making the only choice. If the values didn’t change then it means we got stuck.

```
def run(Grid):
    stuck = False
    while not stuck:
        # Check how many boxes have a fixed value
        previous_solved = len([box for box in Grid.keys() if len(Grid[box]) == 1])


        Grid = eliminate(Grid)


        Grid = choice(Grid)


        Grid = naked_pairs(Grid)

        # Check how many boxes have a value, to compare
        post_solved_values = len([box for box in Grid.keys() if len(Grid[box]) == 1])

        # If no new values were added, stop the loop.
        stuck = previous_solved == post_solved_values


        # if the current sudoku board is cannot be solved then return False
        if len([box for box in Grid.keys() if len(Grid[box]) == 0]):
            return False
    return Grid
```

After getting stuck we can use any search algorithms for checking the possible moves. Here, we use DFS.

```
def search(Grid):
    Grid = run(Grid)

    if Grid is False:
        return False

    if all(len(v) == 1 for k,v in Grid.items()):
        return Grid

    # Choose one of the unfilled squares with the fewest possibilities
    length,k = min((len(val), key) for key,val in Grid.items() if len(val) > 1)
    # print(k, length)

    # Now use recurrence to solve each one of the resulting sudoku
    for digit in Grid[k]:
        new_sudoku = dict(list(Grid.items()))
        new_sudoku[k] = digit
        attempt = search(new_sudoku)
        if attempt:
            return attempt
```

The following is the main function:

```python
if __name__ == '__main__':



    rows = 'ABCDEFGHI'
    cols = '123456789'
    boxes = cartesian_product(rows, cols)

    row_units = [cartesian_product(r, cols) for r in rows]
    col_units = [cartesian_product(rows, c) for c in cols]
    box_units = [cartesian_product(r,c)
                 for r in ['ABC', 'DEF', 'GHI']
                 for c in ['123','456','789']]

    unit_list = row_units + col_units + box_units

    # each box(key) with its units(value)
    unit_dict = dict((box, [unit for unit in unit_list if box in unit]) for box in boxes)

    # each box with its peers
    peer_dict = dict((box, set(sum(unit_dict[box], [])) - set([box])) for box in boxes)

    # start string converted to dictionary
    assert len(start) == 81
    Grid = dict(zip(boxes, start))

    # replacing the x with '123456789' (possible values in the box)
    for k,v in Grid.items():
        if v == 'x':
            Grid[k] = '123456789'

    solved_grid = search(Grid)

    display_game_board(solved_grid)
```

***OUTPUT SCREENSHOT:***

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

***References:***

1. Study of Brute Force and Heuristic Approach to Solve Sudoku by Taruna Kumari ,Preeti yadav , Lavina (International Journal of Emerging Trends & Technology in Computer Science (IJETTCS) Web Site: www.ijettcs.org Email: editor@ijettcs.org Volume 4, Issue 5(2), September - October 2015

2. Methods for Solving Sudoku Puzzles by Anshul Kanakia (kanakia@colorado.edu) and John Klingner(john.klingner@colorado.edu) CSCI - 5454, CU Boulder.

3. Sudoku as a Constraint Problem by Helmut Simonis IC-Parc Imperial College London .

4. A study of Sudoku solving algorithms by PATRIK BERGGREN PABERGG@KTH.SE GOTLANDSGATAN 46 LGH 1104116 65 STOCKHOLM—DAVID NILSSON DAVNILS@KTH.SE KUNGSHAMRA 48 LGH 1010170 70 SOLNA.