# A PROJECT

# ON

# Sudoku

# USING

# Constraint Satisfaction Problems


## BITS Pilani, Dubai Campus, DIAC

## (Nov 16, 2022)

**By:**

**Megha Manoj (2020A7PS0016U)**

**Shifa Parveen Allaudeen (2020A7PS0040U)**

**Fatima Masood (2020A7PS0103U)**

**Birla Institute of Technology and Science**

**D.I.A.C, Dubai, UAE**

**A Report**

**On**

# Sudoku using

# Constraint Satisfaction Problem

**Prepared for**

**Dr Sujala D. Shetty**

**Instructor in charge**

**By**

**Megha Manoj**

**Shifa Parveen Allaudeen**

**Fatima Masood**

**Approved by**

**Dr Sujala D. Shetty**

**Instructor in charge**

**November 2022**

# ACKNOWLEDGEMENT

We would first like to express our sincere gratitude to Prof. Madapusi, Director, BITS Pilani, Dubai Campus, for giving us an opportunity to apply our knowledge in engineering concepts in a practical atmosphere.

Next, we would like to thank our instructor Dr. Sujala D. Shetty for her constant support and encouragement, and invaluable guidance throughout the completion of the project.

Finally, we would like to thank anyone who directly and indirectly helped us in completing this assignment.

# TABLE OF CONTENTS

# Chapter 1: INTRODUCTION

**Aim**: To solve sudoku puzzles of different difficulty levels using constraint satisfaction problem algorithms.

**Game Playing in AI**

Game playing was one of the first tasks undertaken in Artificial Intelligence. Game theory has its history from 1950, almost from the days when computers became programmable. Game playing is the design of artificial intelligence programs to be able to play more than one game successfully .The very first game that is been tackled in AI is chess. Initiators in the field of game theory in AI were Konard Zuse (the inventor of the first programmable computer and the first programming language), Claude Shannon (the inventor of information theory), Norbert Wiener (the creator of modern control theory), and Alan Turing. Since then, there has been a steady progress in the standard of play, to the point that machines have defeated human champions (although not every time) in chess and backgammon, and are competitive in many other games.

**Types of Game:**

- Perfect Information Game: In which player knows all the possible moves of himself and opponent and their results.

  E.g. Chess.

- Imperfect Information Game: In which player does not know all the possible moves of the opponent. E.g. Bridge since all the cards is not visible to player.
- Definition Game playing is a search problem defined by following components:
  - Initial state: This defines initial configuration of the game and identifies first player to move.
  - Successor function: This identifies which are the possible states that can be achieved from the current state. This function returns a list of (move, state) pairs, each indicating a legal move and the resulting state.
  - Goal test: Which checks whether a given state is a goal state or not. States where the game ends are called as terminal states.
  - Path cost / utility / payoff function: Which gives a numeric value for the terminal states? In chess, the outcome is win, loss or draw, with values +1, -1, or 0. Some games have wider range of possible outcomes.

**Characteristics of game playing**

- Unpredictable Opponent: Generally we cannot predict the behaviour of the opponent. Thus we need to find a solution which is a strategy specifying a move for every possible opponent move or every possible state.
- Time Constraints: Every game has a time constraints. Thus it may be infeasible to find the best move in this time.
- The most common search technique in game playing is Minimax search procedure. It is depth-first depth-limited search procedure. It is used for games like chess and tic-tac-toe.

**Sudoku**

Sudoku is a logic based puzzle game which originated in Japan. The game involves filling out the empty spaces present in a grid of 9x9 cells with numbers from 1 to 9. The 9x9 grid is further divided into 9 squares which are subgrids of 3x3. the rules of sudoku are as follows:

1. Each row, column and diagonal must be filled with numbers within the range 1-9 such that each cell of the 9x9 grid has a unique number.
2. No number must be repeated in each column, row or diagonal.
3. Similarly, the 9 cells of each subgrid of the 9x9 grid must contain a unique number from 1-9.
4. the sum of all the values filled in the cells of each column, row or diagonal must be 45.
5. Similarly, the sum of each subgrid must be 45.

Every well-designed sudoku puzzle is said to have a unique solution which can be achieved by following the game's rules. The values already present within the puzzle serve as clues for logical guidance in reaching the final solution. It is said that a puzzle which has less than or equal to 16 clues will have multiple solutions as it gives the opportunity for the arrangement of the numbers. The minimum number of clues a sudoku puzzle can have which will provide a unique solution is 17.

# Chapter 2: METHODOLOGY

**Constraint Satisfaction Problem**

Constraint Satisfaction Problem is a technique where certain restrictions/constraints are set on the variables present in a problem for solving. The method involves assigning values from a domain of available unique values such that the all set constraints are satisfied. To solve a problem using constraint satisfaction, the following definitions are required:

1. variables (A)- set of all variables present in the problem. A= {a1, a2,...,an}
2. domain (D)- set of all possible values which can be assigned to a variable.
3. constraints- relations that hold true between variables using the values in the domain

Each variable ai has a domain Di consisting of all possible values which ai can be assigned. The constraints eliminate the values which do not satisfy the requirements.

Types of CSP algorithms:

1. Backtracking search

   Backtracking search starts with an empty solution and assigns values based on the constraints one at a time until all the variables are assigned a value satisfying the problem.

   The algorithm chooses variables based on minimum heuristic value(MRV) for quick backtracking by selecting a variable with a smaller domain than the rest. If the assigned variable causes some inconsistency with the already assigned variables in terms of the constraints, the algorithm backtracks to the previously assigned variables and reassigns the next possible value to them.

   In this problem, a recursive backtrack algorithm has been implemented. The recursive algorithm checks the number of conflicts occuring for a variable with its neighbours. In case of zero conflicts, the variable is assigned a value and proceeds to assign values for the next variable recursively. In case of a conflict between the chosen variable and the previous variable, the algorithm changes the value of the previous variable.

2. Backtracking with Forward checking

   Forward checking algorithm prunes the value assigned to a particular variable from its neighbour nodes at the time of assignment. It reduces the domain of the neighbouring variables and hence prevents extended exploration for assigning values to a variable, thereby making the solving the problem faster.

3. Backtracking with Maintaining Arc Consistency (MAC)

   Arcs in constraint satisfaction problem refers to a constraint between 2 variables, say x and y such that (x, y) is assigned a value from satisfying a constraint. a constraint with no values removable is said to be consistent. Arc consistency ensures that every relation (x,y) in a constraint satisfaction problem is consistent.

An algorithm AC-3 has been implemented for maintaining arc consistency. It assigns a value to a variable and subtracts the assigned value from all its neighbouring variables. If the domain between a variable and its neighbours becomes void, it leads to some consistency in the value assigned to the variable. This is resolved by changing the variable's value and proceeding forward.

Applying CSP to sudoku gives the following definitions:

1. Variables V= {CELL0,...CELL80} where CELLi represents a single cell and its position in the sudoku grid of 81 cells.
2. Domain Di of each variable V where Di= {0,1,2,3,4,5,6,7,8,9}
3. Constraints- the following constraints have been applied to the Sudoku puzzle

   i. Each cell in a row must have a unique value

   ii. Each cell in a column must have a unique value

   iii. Each cell in a 3x3 sub-grid must have a unique value.

# Chapter 3: CODE

**Code:**

```
#Main.py
from tkinter import *
from gui import SudokuUI
root = Tk()

SudokuUI(root)

root.title("Sudoku")
root.mainloop()
```

```
#gui.py
from tkinter import *
from timeit import default_timer as timer
from tkinter import messagebox
import threading
import copy

from sudokucsp import SudokuCSP
from csp import backtracking_search, mrv, unordered_domain_values, forward_checking, mac,
no_inference
MARGIN = 20  # Pixels around the board
SIDE = 50  # Width of each board cell
WIDTH_B = HEIGHT_B = MARGIN * 2 + SIDE * 9  # Width and height of the whole board
WIDTH = WIDTH_B + 130  # Width of board, solve and reset


class SudokuUI(Frame):

    def __init__(self, parent):
        self.parent = parent
        self.original_board = [[0 for j in range(9)] for i in range(9)] #empty board
        self.current_board = copy.deepcopy(self.original_board) #board for displaying soln
        Frame.__init__(self, parent)
        self.row, self.col = 0, 0
        self.__initUI()

    def __initUI(self):
        self.pack(fill=BOTH, expand=1)
        self.canvas = Canvas(self, width=WIDTH_B, height=HEIGHT_B)
        self.canvas.pack(fill=BOTH, side=TOP)
```

```
        self.canvas.grid(row=0, column=0, rowspan=30, columnspan=60)

        # level 1 ->easy & level 2 -> hard
        self.level = IntVar(value=1)
        self.which = 0 #select board to be solved from 3 boards each for both levels

        # StringVar shows the time used by an algorithm
        self.time = StringVar()
        self.time.set("Time:                  ")
        # number of backtracks
        self.n_bt = StringVar()
        self.n_bt.set("N. BT:   ")


        # the default: lvl= 1 and which= 1
        self.__change_level()

        #solve and reset
        self.clear_button = Button(self, text="Reset", command=self.__clear_board, width=15,
height=5)
        self.clear_button.grid(row=10, column=61, padx=20, columnspan=3)
        self.solve_button = Button(self, text="Solve", command=self.solve_clicked, width=15,
height=5)
        self.solve_button.grid(row=13, column=61, padx=20, columnspan=3)

        lbltime = Label(self, textvariable=self.time)
        lblBT = Label(self, textvariable=self.n_bt)

        Label(self, text="Level:              ").grid(row=14, column=61)
        lbltime.grid(row=25, column=0)

        #to choose a level to play at
        lblBT.grid(row=27, column=0)
        self.level = IntVar()
        self.lvl = []
        self.lvl.append(Radiobutton(self, text="Easy   ", variable=self.level, value=1,
command=self.__change_level))
        self.lvl[0].grid(row=15, column=62, padx=2)
        self.lvl.append(Radiobutton(self, text="Hard  ", variable=self.level, value=2,
command=self.__change_level))
        self.lvl[1].grid(row=16, column=62)


        Label(self, text="Inference:              ").grid(row=17, column=61)
        lbltime.grid(row=30, column=0)

        #choose algorithm
        lblBT.grid(row=32, column=0)
        self.inference = StringVar()
        self.radio = []
        self.radio.append(Radiobutton(self, text="No Inference", variable=self.inference,
```

```
value="NO_INFERENCE"))
        self.radio[0].grid(row=17, column=62, padx=2)
        self.radio.append(Radiobutton(self, text="FC              ", variable=self.inference, value="FC"))
        self.radio[1].grid(row=18, column=62)
        self.radio.append(Radiobutton(self, text="MAC           ", variable=self.inference,
value="MAC"))
        self.radio[2].grid(row=19, column=62)
        self.inference.set("NO_INFERENCE")

        Label(self, text="Variable to choose:").grid(row=21, column=61)
        lbltime.grid(row=30, column=0)

        lblBT.grid(row=32, column=0)
        #heuristic
        self.var_to_choose = StringVar()
        self.radio.append(Radiobutton(self, text="MRV", variable=self.var_to_choose, value="MRV"))
        self.radio[3].grid(row=21, column=62)

        self.var_to_choose.set("MRV")

        self.__draw_grid()
        self.__draw_puzzle()

    def solve_clicked(self):

        # disable buttons
        for rb in self.radio:
            rb.config(state=DISABLED)

        for l in self.lvl:
            l.config(state=DISABLED)

        self.clear_button.config(state=DISABLED)
        self.solve_button.config(state=DISABLED)

        p = threading.Thread(target=self.solve_sudoku)
        p.start()
        #messagebox.showinfo("Working", "We are looking for a solution, please wait some seconds
...")

    def solve_sudoku(self):

        s = SudokuCSP(self.current_board)
        inf, dv, suv = None, None, None

        if self.inference.get() == "NO_INFERENCE":
            inf = no_inference
        elif self.inference.get() == "FC":
            inf = forward_checking
        elif self.inference.get() == "MAC":
            inf = mac
```

```
        if self.var_to_choose.get() == "MRV":
            suv = mrv

        start = timer()
        a = backtracking_search(s, select_unassigned_variable=suv,
order_domain_values=unordered_domain_values,
                                inference=inf)
        end = timer()
        # if a != null -> soln found
        # if a == null -> alert that initial board breaks constraints
        if a:
            for i in range(9):
                for j in range(9):
                    index = i * 9 + j
                    self.current_board[i][j] = a.get("CELL" + str(index))
        else:
            messagebox.showerror("Error", "Invalid sudoku puzzle, please check the initial state")

        # solution display
        self.__draw_puzzle()
        self.time.set("Time: "+str(round(end-start, 5))+" seconds")
        self.n_bt.set("N. BR: "+str(s.n_bt))

        # re-enabling buttons
        for rb in self.radio:
            rb.config(state=NORMAL)

        for l in self.lvl:
            l.config(state=NORMAL)

        self.clear_button.config(state=NORMAL)
        self.solve_button.config(state=NORMAL)

    def __change_level(self):
        self.which = (self.which+1) % 3
        if self.level.get() == 1:
            if self.which == 0:
                self.original_board[0] = [0, 6, 0, 3, 0, 0, 8, 0, 4]
                self.original_board[1] = [5, 3, 7, 0, 9, 0, 0, 0, 0]
                self.original_board[2] = [0, 4, 0, 0, 0, 6, 0, 0, 7]
                self.original_board[3] = [0, 9, 0, 0, 5, 0, 0, 0, 0]
                self.original_board[4] = [0, 0, 0, 0, 0, 0, 0, 0, 0]
                self.original_board[5] = [7, 1, 3, 0, 2, 0, 0, 4, 0]
                self.original_board[6] = [3, 0, 6, 4, 0, 0, 0, 1, 0]
                self.original_board[7] = [0, 0, 0, 0, 6, 0, 5, 2, 3]
                self.original_board[8] = [1, 0, 2, 0, 0, 9, 0, 8, 0]
            elif self.which == 1:
                self.original_board[0] = [7, 9, 0, 4, 0, 2, 3, 8, 1]
                self.original_board[1] = [5, 0, 3, 0, 0, 0, 9, 0, 0]
                self.original_board[2] = [0, 0, 0, 0, 3, 0, 0, 7, 0]
```

```
            self.original_board[3] = [0, 0, 0, 0, 0, 5, 0, 0, 2]
            self.original_board[4] = [9, 2, 0, 8, 1, 0, 7, 0, 0]
            self.original_board[5] = [4, 6, 0, 0, 0, 0, 5, 1, 9]
            self.original_board[6] = [0, 1, 0, 0, 0, 0, 2, 3, 8]
            self.original_board[7] = [8, 0, 0, 0, 4, 1, 0, 0, 0]
            self.original_board[8] = [0, 0, 9, 0, 8, 0, 1, 0, 4]
        elif self.which == 2:
            self.original_board[0] = [0, 3, 0, 5, 0, 6, 2, 0, 0]
            self.original_board[1] = [8, 2, 0, 0, 0, 1, 0, 0, 4]
            self.original_board[2] = [6, 0, 7, 8, 3, 0, 0, 9, 1]
            self.original_board[3] = [0, 0, 0, 0, 0, 0, 0, 2, 9]
            self.original_board[4] = [5, 0, 0, 6, 0, 7, 0, 0, 3]
            self.original_board[5] = [3, 9, 0, 0, 0, 0, 0, 0, 0]
            self.original_board[6] = [4, 5, 0, 0, 8, 9, 1, 0, 2]
            self.original_board[7] = [9, 0, 0, 1, 0, 0, 0, 4, 6]
            self.original_board[8] = [0, 0, 3, 7, 0, 4, 0, 5, 0]

    elif self.level.get() == 2:
        if self.which == 0:
            self.original_board[0] = [8, 0, 0, 0, 0, 0, 0, 0, 0]
            self.original_board[1] = [0, 0, 3, 6, 0, 0, 0, 0, 0]
            self.original_board[2] = [0, 7, 0, 0, 9, 0, 2, 0, 0]
            self.original_board[3] = [0, 5, 0, 0, 0, 7, 0, 0, 0]
            self.original_board[4] = [0, 0, 0, 0, 4, 5, 7, 0, 0]
            self.original_board[5] = [0, 0, 0, 1, 0, 0, 0, 3, 0]
            self.original_board[6] = [0, 0, 1, 0, 0, 0, 0, 6, 8]
            self.original_board[7] = [0, 0, 8, 5, 0, 0, 0, 1, 0]
            self.original_board[8] = [0, 9, 0, 0, 0, 0, 4, 0, 0]
        elif self.which == 1:
            self.original_board[0] = [2, 0, 0, 0, 0, 0, 0, 4, 3]
            self.original_board[1] = [1, 9, 0, 0, 3, 0, 0, 0, 0]
            self.original_board[2] = [0, 6, 0, 0, 0, 5, 0, 0, 0]
            self.original_board[3] = [0, 5, 0, 2, 6, 0, 0, 0, 8]
            self.original_board[4] = [0, 0, 0, 0, 7, 0, 0, 0, 0]
            self.original_board[5] = [6, 0, 0, 0, 5, 3, 0, 1, 0]
            self.original_board[6] = [0, 0, 0, 6, 0, 0, 0, 2, 0]
            self.original_board[7] = [0, 0, 0, 0, 8, 0, 0, 3, 4]
            self.original_board[8] = [9, 1, 0, 0, 0, 0, 0, 0, 6]
        elif self.which == 2:
            self.original_board[0] = [0, 0, 0, 0, 2, 0, 0, 0, 5]
            self.original_board[1] = [0, 0, 1, 6, 0, 0, 0, 0, 0]
            self.original_board[2] = [0, 6, 0, 7, 0, 0, 0, 8, 1]
            self.original_board[3] = [0, 0, 0, 3, 0, 0, 5, 0, 0]
            self.original_board[4] = [3, 0, 8, 5, 0, 6, 2, 0, 9]
            self.original_board[5] = [0, 0, 4, 0, 0, 7, 0, 0, 0]
            self.original_board[6] = [7, 4, 0, 0, 0, 9, 0, 1, 0]
            self.original_board[7] = [0, 0, 0, 0, 0, 5, 9, 0, 0]
            self.original_board[8] = [8, 0, 0, 0, 7, 0, 0, 0, 0]

    self.current_board = copy.deepcopy(self.original_board)
    self.__draw_puzzle()
```

```
  def __draw_grid(self):

     for i in range(10):
        if i % 3 == 0:
           color = "black"
        else:
           color = "gray"
        x0 = MARGIN + i * SIDE
        y0 = MARGIN
        x1 = MARGIN + i * SIDE
        y1 = HEIGHT_B - MARGIN
        self.canvas.create_line(x0, y0, x1, y1, fill=color)
        x0 = MARGIN
        y0 = MARGIN + i * SIDE
        x1 = WIDTH_B - MARGIN
        y1 = MARGIN + i * SIDE
        self.canvas.create_line(x0, y0, x1, y1, fill=color)

  def __draw_puzzle(self):
     self.canvas.delete("numbers")
     self.time.set("Time:                ")
     self.n_bt.set("N. BT:   ")
     for i in range(9):
        for j in range(9):
           cell = self.current_board[i][j]
           if cell != 0:
              x = MARGIN + j * SIDE + SIDE / 2
              y = MARGIN + i * SIDE + SIDE / 2
              if str(cell) == str(self.original_board[i][j]):
                 self.canvas.create_text(x, y, text=cell, tags="numbers", fill="black")
              else:
                 self.canvas.create_text(x, y, text=cell, tags="numbers", fill="red")

  def __clear_board(self):
     self.current_board = copy.deepcopy(self.original_board)
     self.__draw_puzzle()
```

```
#sudokucsp.py
from csp import *

class SudokuCSP(CSP):

   def __init__(self, board):
```

```
    self.domains = {}
    self.neighbors = {}
    # variable names = "CELL [no.]"
    for v in range(81):
        self.neighbors.update({'CELL' + str(v): {}})
    for i in range(9):
        for j in range(9):
            name = (i * 9 + j)
            var = "CELL"+str(name)
            self.add_neighbor(var, self.get_row(i) | self.get_column(j) | self.get_square(i, j))
            # if the board has a value in cell[i][j], the domain of this variable will be that number
            if board[i][j] != 0:
                self.domains.update({var: str(board[i][j])})
            else:
                self.domains.update({var: '123456789'})

    CSP.__init__(self, None, self.domains, self.neighbors, different_values_constraint)

# returns subgrid given row and column index
def get_square(self, i, j):
    if i < 3:
        if j < 3:
            return self.get_square_box(0)
        elif j < 6:
            return self.get_square_box(3)
        else:
            return self.get_square_box(6)
    elif i < 6:
        if j < 3:
            return self.get_square_box(27)
        elif j < 6:
            return self.get_square_box(30)
        else:
            return self.get_square_box(33)
    else:
        if j < 3:
            return self.get_square_box(54)
        elif j < 6:
            return self.get_square_box(57)
        else:
            return self.get_square_box(60)

# returns the subgrid based on of the index's variabile, it must be 0, 3, 6, 27, 30, 33, 54, 57 or 60
def get_square_box(self, index):
    tmp = set()
    tmp.add("CELL"+str(index))
    tmp.add("CELL"+str(index+1))
    tmp.add("CELL"+str(index+2))
    tmp.add("CELL"+str(index+9))
    tmp.add("CELL"+str(index+10))
```

```
        tmp.add("CELL"+str(index+11))
        tmp.add("CELL"+str(index+18))
        tmp.add("CELL"+str(index+19))
        tmp.add("CELL"+str(index+20))
        return tmp

    def get_column(self, index):
        return {'CELL'+str(j) for j in range(index, index+81, 9)}

    def get_row(self, index):
            return {('CELL' + str(x + index * 9)) for x in range(9)}

    def add_neighbor(self, var, elements):
        # if variable!= itself then add to neighbour
        self.neighbors.update({var: {x for x in elements if x != var}})
```

```
#csp.py
class CSP:
    """
    A CSP is specified by the following inputs:
        variables   A list of atomic variables.
        domains     A dict containing a variable as key and all possible values it can be assigned as its
value
        neighbors   A dict containing key as a variable and its value is all its neighbouring variables
        constraints A function f(A, a, B, b) that returns true if neighbors
                A, B satisfy the constraint when they have values A=a, B=b

    argument 'a' is an assignment
    dict of {var:val} entries:
        assign(var, val, a)     Assign a[var] = val
        unassign(var, a)        unassign value from variable
        nconflicts(var, val, a) Return the number of other variables that has same value as current
variable
        curr_domains[var]       remaining consistent values for var used by constraint propagation
routines.
    The following methods are used only by graph_search and tree_search:
        actions(state)          Return a list of actions
        result(state, action)   Return a successor of state
        goal_test(state)        Return true if all constraints satisfied
    The following are just for debugging purposes:
        nassigns                tracks the number of assignments made
        display(a)              prints puzzle
    """
```

```
    def __init__(self, variables, domains, neighbors, constraints):
        variables = variables or list(domains.keys()) #if variable is empty domain keys becomes its
values
        self.variables = variables
        self.domains = domains
        self.neighbors = neighbors
        self.constraints = constraints
        self.initial = ()
        self.curr_domains = None
        self.no_assigns = 0 #no. of assignments
        self.n_bt = 0 #no. of backtracks

    def assign(self, var, val, assignment):
        assignment[var] = val
        self.no_assigns += 1

    def unassign(self, var, assignment):
        if var in assignment:
            del assignment[var]

    def nconflicts(self, var, val, assignment):
        """Return the number of conflicts var=val has with other variables."""
        count = 0
        for var2 in self.neighbors.get(var):
            val2 = None
            if assignment.__contains__(var2):
                val2 = assignment[var2] #if assigned variables contain var2, value assigned to val2
            if val2 is not None and self.constraints(var, val, var2, val2) is False:
                count += 1 #if val2 is non empty or constrant not satisfied
        return count

    def display(self, assignment):
        print('CSP:', self, 'with assignment:', assignment)

    def goal_test(self, state):
        assignment = dict(state)
        #create dictionary assignment
        #if length of assignment and variables are equal then all variables have been assigned
        #returns true if there are no conflicts
        return (len(assignment) == len(self.variables)
                and all(self.nconflicts(variables, assignment[variables], assignment) == 0
                        for variables in self.variables))


    # These are for constraint propagation

    def support_pruning(self):
        if self.curr_domains is None:
            self.curr_domains = {v: list(self.domains[v]) for v in self.variables}

    def suppose(self, var, value):
```

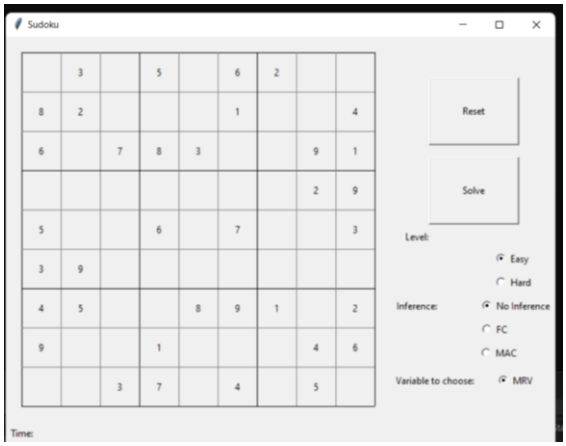
```
        """Assume var=value."""
        self.support_pruning()
        removals = [(var, a) for a in self.curr_domains[var] if a != value]
        self.curr_domains[var] = [value]
        return removals

    def prune(self, var, value, removals):
        """Rule out var=value."""
        self.curr_domains[var].remove(value)
        if removals is not None:
            removals.append((var, value))

    def choices(self, var):
        """Return all values for var that aren't currently ruled out."""
        return (self.curr_domains or self.domains)[var]

    def restore(self, removals):
        """Undo a supposition and all inferences from it."""
        for B, b in removals:
            self.curr_domains[B].append(b)


# _____________a_____________________________________________________________________
# Constraint Propagation with AC-3

def AC3(csp, queue=None, removals=None):
    if queue is None:
        queue = [(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]]
    csp.support_pruning()
    while queue:
        (Xi, Xj) = queue.pop()
        if revise(csp, Xi, Xj, removals):
            if not csp.curr_domains[Xi]:
                return False
            for Xk in csp.neighbors[Xi]:
                if Xk != Xi:
                    queue.append((Xk, Xi))
    return True


def revise(csp, Xi, Xj, removals):
    """Return true if value removed."""
    revised = False
    for x in csp.curr_domains[Xi][:]:
        # If Xi=x conflicts with Xj=y for every possible y, eliminate Xi=x
        if all(not csp.constraints(Xi, x, Xj, y) for y in csp.curr_domains[Xj]):
            csp.prune(Xi, x, removals)
            revised = True
    return revised

# ____________________________________________________________________________
```

```
# CSP Backtracking Search

# Variable ordering

def first_unassigned_variable(assignment, csp):
    """The default variable order."""
    for var in csp.variables:
        if var not in assignment:
            return var

def mrv(assignment, csp):
    """Minimum-remaining-values heuristic."""
    vars_to_check = []
    size = []
    for v in csp.variables:
        if v not in assignment.keys():
            vars_to_check.append(v)
            size.append(num_legal_values(csp, v, assignment))
    return vars_to_check[size.index(min(size))]


def num_legal_values(csp, var, assignment):
    if csp.curr_domains:
        return len(csp.curr_domains[var])
    else:
        count = 0
        for val in csp.domains[var]:
            if csp.nconflicts(var, val, assignment) == 0:
                count += 1
        return count

# Value ordering


def unordered_domain_values(var, assignment, csp):
    """The default value order."""
    return csp.choices(var)


def lcv(var, assignment, csp):
    """Least-constraining-values heuristic."""
    return sorted(csp.choices(var),
                  key=lambda val: csp.nconflicts(var, val, assignment))


# Inference


def no_inference(csp, var, value, assignment, removals):
    return True
```

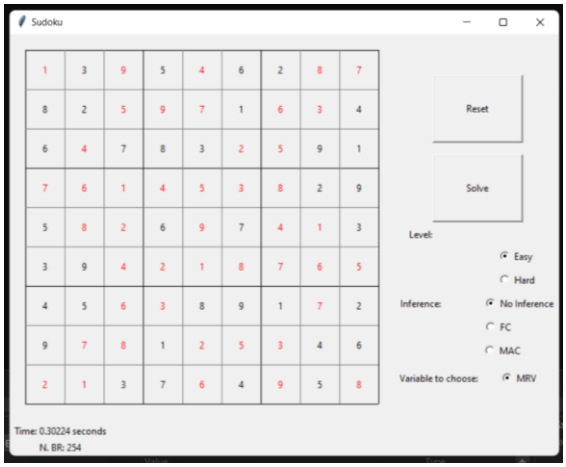
```
def forward_checking(csp, var, value, assignment, removals):
    """Prune neighbor values inconsistent with var=value."""
    for B in csp.neighbors[var]:
        if B not in assignment:
            for b in csp.curr_domains[B][:]:
                if not csp.constraints(var, value, B, b):
                    csp.prune(B, b, removals)
            if not csp.curr_domains[B]:
                return False
    return True


def mac(csp, var, value, assignment, removals):
    """Maintain arc consistency."""
    return AC3(csp, [(X, var) for X in csp.neighbors[var]], removals)



def backtracking_search(csp,
                        select_unassigned_variable,
                        order_domain_values,
                        inference):

    def backtrack(assignment): #recursive backtrack
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if csp.nconflicts(var, value, assignment) == 0:
                csp.assign(var, value, assignment)
                removals = csp.suppose(var, value)
                if inference(csp, var, value, assignment, removals):
                    result = backtrack(assignment)
                    if result is not None:
                        return result
                    else:
                        csp.n_bt += 1
                csp.restore(removals)
        csp.unassign(var, assignment)
        return None

    result = backtrack({})
    assert result is None or csp.goal_test(result)
    return result


def different_values_constraint(A, a, B, b):
    return a != b
```

## Solutions

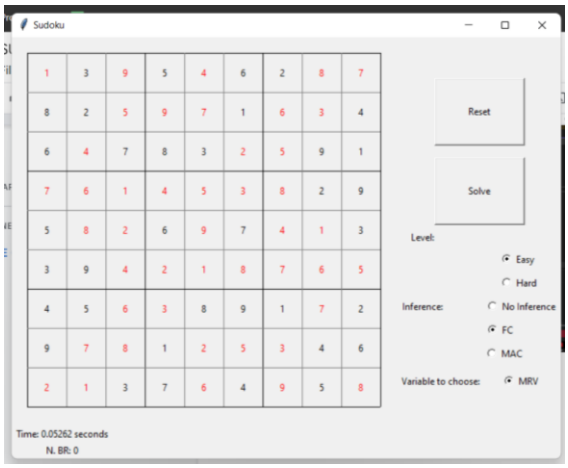
### Initial




### Backtracking



## Backtracking + FC




## Backtracking + Arc consistency

# REFERENCES


- http://vtucs.com/wp-content/uploads/2015/02/Game-Playing-in-Artificial-Intelligence.pdf
- https://www.geeksforgeeks.org/game-playing-in-artificial-intelligence/#:~:text=Game%20Playing%20is%20an%20important,winning%20or%20losing%20the%20game.&text=Generate%20procedure%20so%20that%20only%20good%20moves%20are%20generated.
- https://www.researchgate.net/publication/220835235_Sudoku_and_AI#:~:text=The%20Sudoku%20puzzle%20is%20used,demonstrate%20artificial%20intelligence%20solution%20strategies.