Sudoku Project AI

Cynthia Cheng (cc5469) and Matthew Swartz (mcs871)

How to Run:

The file is sodoku.py and to run is just to run it in any Python IDLE of your choice. The input file it reads is 15th line of the program. Just replace that string and run the program.

Sudoku Formulation:

For x we have 81 values for each square of the board. The domain is from 1-9. For the constraints our first ones are that all rows have unique values, the second is all columns have unique values, and lastly that all grids are unique values.

X: {x1, x2, x3, x4, x5, … , x81}
D: {1, 2, 3, 4, 5, 6, 7, 8, 9}
C: {
AllDiff(x1, x2, x3, x4, x5, x6, x7, x8, x9), AllDiff(x10, x11, x12, x13, x14, x15, x16, x17, x18), … ,
AllDiff(x73, x74, x75, x76, x77, x78, x79, x80, x81),

AllDiff(x1, x10, x19, x28, x37, x46, x55, x64, x73), AllDiff(x2, x11, x20, x29, x38, x47, x56, x65, x74), … , AllDiff(x9, x18, x27, x36, x45, x54, x63, x72, x81)

AllDiff(x1, x2, x3, x10, x11, x12, x19, x20, x21), AllDiff(x4, x5, x6, x13, x14, x15, x22, x23, x24), … ,
AllDiff(x61, x61, x63, x70, x71, x72, x79, x80, x81)
}

Output Files:

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

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

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

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

Source Code:

```
# initial empty game board
board = []
# load in game board from input file and place into board array variable
def loadInputFile(filename):
    # open input file
    with open(filename) as file:
        lines = [line.rstrip() for line in file]
        for i in range(len(lines)):
            board.append([int(x) for x in lines[i].split()])

# LOAD THE INPUT FILE HERE
loadInputFile("Input1.txt")

# import proper libraries
import numpy as np
from functools import reduce

# convert board array to numpy array
sodukuBoard = np.asarray(board)
# this will be our tuples of 3 numbers for a row of a subgrid
slices = [slice(0,3), slice(3,6), slice(6,9)]
s1,s2,s3 = slices
allgrids=[(si,sj) for si in [s1,s2,s3] for sj in [s1,s2,s3]]

# finds the 3x3 grid slice the var coordinates belong in
def varToGrid(var):
    row,col = var
    grid = ( slices[int(row/3)], slices[int(col/3)] )
```

```
    return grid

# value range of 1-9 stored in array
FULLDOMAIN = np.array(range(1,10))


# CONSTRAINTS
# check that the rows of the solution has no repeats, i.e. one 1, one 2, ... , one 9 per row
def unique_rows(sodukuBoard):
    for row in sodukuBoard:
        # if the row does not equal the array 1-9 then there is a repeat in the row, return false
        if not np.array_equal(np.unique(row), np.array(range(1,10))):
            return False
    return True
# check that the colums of the solution have no repeated 1-9 values
def unique_columns(sodukuBoard):
    #transpose soduku to get columns
    for col in sodukuBoard.T:
        # same as above, compare column to array [1, 2, 3, ... , 9] i.e. no duplicates
        if not np.array_equal(np.unique(col),np.array(range(1,10))) :
            return False
    return True
# check that each grid, the 9 3x3 squares, of the board have no repeated values
def unique_grids(sodukuBoard):
    for grid in allgrids:
        # check a 3x3 grid is equivalent to [1, 2, 3, ... , 9] i.e. no duplicates in the subgrid
        if not np.array_equal(np.unique(sodukuBoard[grid]),np.array(range(1,10))) :
            return False
    return True
# check the board for any 0's
def isComplete(sodukuBoard):
    if 0 in sodukuBoard:
        return False
    else:
        return True
# checks all our constraints to see if our board is a valid complete solution
def checkCorrect(sodukuBoard):
    if unique_columns(sodukuBoard):
        if unique_rows(sodukuBoard):
            if unique_grids(sodukuBoard):
                return True
    return False
```

```
# BACKTRACKING FUNCTIONS
# returns an array of the available values between 1-9 avaible for this square to keep it within
the constraints above
def getDomainValues(var, sodukuBoard):
    row,col = var
    # this creates an array of numbers between 1-9 which already exist in this row, column, or
subgrid
    used_d = reduce(np.union1d, (sodukuBoard[row,:], sodukuBoard[:,col],
sodukuBoard[varToGrid(var)]))
    # values left to pick from for this square to make sure it still satisfies thee constraint
    avail_d = np.setdiff1d(FULLDOMAIN, used_d)
    return avail_d

# Minimum Remaining Value Heuristic
def minimumRemainingVal(vars, sodukuBoard):
    # gets our ORDER-DOMAIN-VALUES for the remaining 0's left on our board
    avail_domains = [getDomainValues(var,sodukuBoard) for var in vars]
    # the number of values a remaning zero can be, i.e. the size of one of the domains in
avail_domain
    avail_sizes = [len(avail_d) for avail_d in avail_domains]
    # the index of the domain with the smallest size
    index = np.argmin(avail_sizes)
    # return the tuple of the coordinates of the 0 being replaced, and its possible domains
    return vars[index], avail_domains[index]

# our actual backtracking algorithm
def backtrackingSearch(sodukuBoard):
    # solution is complete return the board
    if isComplete(sodukuBoard):
        return sodukuBoard
    # our tuples of the coordinates of the 0's left in the board
    vars = [tuple(e) for e in np.transpose(np.where(sodukuBoard==0))]
    # find our zero to replace based on the MRV heuristic
    var, avail_d = minimumRemainingVal(vars, sodukuBoard)
    # recursively solve and backtrack an assignment when stuck
    for value in avail_d:
        # set 0 = to new value
        sodukuBoard[var] = value
        # recusive call
        result = backtrackingSearch(sodukuBoard)
        # checks if a result of the value placement works
        if np.any(result):
            return result
        # reset board to zero as placement did not work with that number
```

```
        else:
            sodukuBoard[var] = 0
    # fails to find value
    return False


# Solve the board
backtrackingSearch(sodukuBoard)
checkCorrect(sodukuBoard)

# Write solution to output file
with open('output.txt','w') as f:
    output_str = ""
    # loop through solution array
    for i in range(9):
        for j in range(9):
            # add number with a space
            output_str += (str(sodukuBoard[i][j]) + " ")
        # new line for each row
        output_str += "\n"

    f.write(output_str)
```