Eliahou Mayor

Sudoku Puzzle

To run the program, just run:

python3 Sudoku.py

from the directory that the program is in, or open it up in your favorite IDE and hit run.

The program will ask you for the path to the input file, enter it and then hit enter.

The program will then produce the solution output file in the directory it is running in, with the title output.txt. If you wish to keep this file, make sure you rename it before running the program again.

# Sudoku CSP

def readProblem(board,domain,assignment):

print("Enter the path for the file that contains the problem:")

filepath = input()

f = open(filepath, "r") # open file

# read the board

i = 0 # the row we are up to

for line in f:

values = line.split() # split into 9 numbers

for j in range(9):

board[i][j] = int(values[j])

if int(values[j]) != 0:

domain[i][j] = [int(values[j])] # mark the domain as just that value

assignment[i][j] = int(values[j]) # mark that value as assigned

i += 1

f.close() # close file

def forwardChecking(board,domain):

queue = [(x,y) for x in range(9) for y in range(9) if board[x][y] != 0] # everything assigned already

for valueAssigned in queue:

for position in getRowColumnBlock(valueAssigned[0],valueAssigned[1]): # check all of its row,column,block

x = position[0]

y = position[1]

if board[valueAssigned[0]][valueAssigned[1]] in domain[x][y]: # remove the value assigned

domain[x][y].remove(board[valueAssigned[0]][valueAssigned[1]])

# if that position has no more options left, return false

if len(domain[x][y]) == 0:

return False

# if it has one value left, let's also check it's neighbors by adding to the queue

if len(domain[x][y]) == 1:

queue.extend(getRowColumnBlock(x,y))

return True

# gets all tiles in the same row,column and block

def getRowColumnBlock(x,y):

positions = []

# get the tiles in the row of the x,y tile

for j in range(9):

if j != y:

positions.append((x,j))

# get the tiles in the column of the x,y tile

for i in range(9):

if i != x:

positions.append((i,y))

# get the tiles in the block of the x,y tile

if x < 3:

if y < 3:

for i in range(3):

for j in range(3):

if not(i == x and j == y):

positions.append((i,j))

elif y < 6:

for i in range(3):

for j in range(3,6):

if not(i == x and j == y):

positions.append((i,j))

else:

for i in range(3):

for j in range(6,9):

if not(i == x and j == y):

positions.append((i,j))

elif x < 6:

if y < 3:

for i in range(3,6):

for j in range(3):

if not(i == x and j == y):

positions.append((i,j))

elif y < 6:

for i in range(3,6):

for j in range(3,6):

if not(i == x and j == y):

positions.append((i,j))

else:

for i in range(3,6):

for j in range(6,9):

if not(i == x and j == y):

positions.append((i,j))

elif x < 9:

if y < 3:

for i in range(6,9):

for j in range(3):

if not(i == x and j == y):

positions.append((i,j))

elif y < 6:

for i in range(6,9):

for j in range(3,6):

if not(i == x and j == y):

positions.append((i,j))

else:

for i in range(6,9):

for j in range(6,9):

if not(i == x and j == y):

positions.append((i,j))

return positions

def backtracingAlgo(board,domain,assignment):

if assignmentComplete(assignment): # if assignment is done and is consistent

return assignment # we are done

unassignedTile = getUnassignedVar(board,domain,assignment) # get next unassigned tile

for value in domain[unassignedTile[0]][unassignedTile[1]]: # try every value in domain

if isValidAssignment(assignment,unassignedTile,value): #check if valid move, if yes do the rest:

assignment[unassignedTile[0]][unassignedTile[1]] = value # assign it

result = backtracingAlgo(board,domain,assignment) # explore further

if result != None: # path leads to a solution

return result

assignment[unassignedTile[0]][unassignedTile[1]] = None # path does not lead to solution

return None # failed to find solution, return None

# checks if an assignment is valid

def isValidAssignment(assignment,unassignedTile,value):

# get tiles that the assignment needs to be checked against

tiles = getRowColumnBlock(unassignedTile[0],unassignedTile[1])

for tile in tiles:

if assignment[tile[0]][tile[1]] == value:

return False

return True

# get next unassigned variable

def getUnassignedVar(board,domain,assignment):

# get the size of the smallest domain

smallestDomainSize = 9

for i in range(9):

for j in range(9):

if assignment[i][j] is None:

smallestDomainSize = min(len(domain[i][j]),smallestDomainSize)

# get those tiles that are unassigned with smallest domain

unassigned = [(x,y) for x in range(9) for y in range(9) if assignment[x][y] is None and len(domain[x][y]) == smallestDomainSize]

if len(unassigned) == 1: # if there's only 1 variable with smallest domain, return it

return unassigned[0]

else: # if there's a tie, use degree heuristic

largestDegree = -1 # something really small

largestDegreeTile = unassigned[0]

for tile in unassigned:

if largestDegree < getNumberOfUnassignedNeighbors(assignment,tile): # get largest, if there's a tie, get the first one thats largest

largestDegree = getNumberOfUnassignedNeighbors(assignment,tile)

largestDegreeTile = tile

return largestDegreeTile

# gets the number of unassigned neighbors

def getNumberOfUnassignedNeighbors(assignment,tile):

count = 0

i = tile[0]

j = tile[1]

# check tile directly on top of you

if i > 0:

if(assignment[i - 1][j] is None):

count += 1

# check tile directly beneath you

if i < 8:

if(assignment[i+1][j] is None):

count += 1

# check tile directly to the left of you

if j > 0:

if(assignment[i][j-1] is None):

count += 1

# check tile directly to the right of you

if j < 8:

if(assignment[i][j-1] is None):

count += 1

# check tile that is diagonal upper left of you

if i > 0 and j > 0:

if(assignment[i-1][j-1] is None):

count += 1

# check tile that is diagonal upper right of you

if i > 0 and j < 8:

if(assignment[i-1][j+1] is None):

count += 1

# check tile that is diagonal lower left of you

if i < 8 and j > 0:

if(assignment[i+1][j-1] is None):

count += 1

# check tile that is diagonal lower right of you

if i < 8 and j < 8:

if(assignment[i+1][j+1] is None):

count += 1

return count

# checks if assignment is complete and is consistent

def assignmentComplete(assignment):

for row in assignment:

for column in row:

if column == None:

return False

return True

# writes the solution

def writeSolution(assignment):

filepath = "output.txt"

f = open(filepath, "w") # open file

for i in range(9):

for j in range(9):

f.write(str(assignment[i][j]) + " ")

f.write("\n")

f.close() # close file

# put everything together

def solvePuzzle():

board = [[0 for x in range(9)] for y in range(9)]

domain = [[[1,2,3,4,5,6,7,8,9] for x in range(9)] for y in range(9)]

assignment= [[None for x in range(9)] for y in range(9)] # contains variables that are assigned

readProblem(board,domain,assignment)

if(forwardChecking(board,domain)):

result = backtracingAlgo(board,domain,assignment)

if result is not None:

writeSolution(assignment)

if \_\_name\_\_ == "\_\_main\_\_":

solvePuzzle()

Output File 1:

4 3 5 2 6 9 7 8 1

6 8 2 5 7 1 4 9 3

1 9 7 8 3 4 5 6 2

8 2 6 1 9 5 3 4 7

3 7 4 6 8 2 9 1 5

9 5 1 7 4 3 6 2 8

5 1 9 3 2 6 8 7 4

2 4 8 9 5 7 1 3 6

7 6 3 4 1 8 2 5 9

Output File 2:

1 2 3 6 7 8 9 4 5

5 8 4 2 3 9 7 6 1

9 6 7 1 4 5 3 2 8

3 7 2 4 6 1 5 8 9

6 9 1 5 8 3 2 7 4

4 5 8 7 9 2 6 1 3

8 3 6 9 2 4 1 5 7

2 1 9 8 5 7 4 3 6

7 4 5 3 1 6 8 9 2

Output File 3:

2 7 6 3 1 4 9 5 8

8 5 4 9 6 2 7 1 3

9 1 3 8 7 5 2 6 4

4 6 8 1 2 7 3 9 5

5 9 7 4 3 8 6 2 1

1 3 2 5 9 6 4 8 7

3 2 5 7 8 9 1 4 6

6 4 1 2 5 3 8 7 9

7 8 9 6 4 1 5 3 2