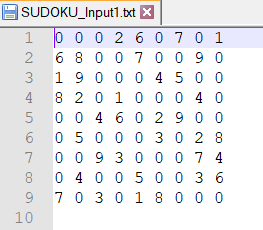
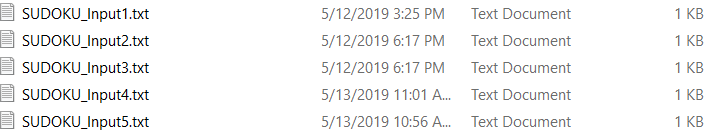
Michael Li (ml5803)

AI Project SUDOKU

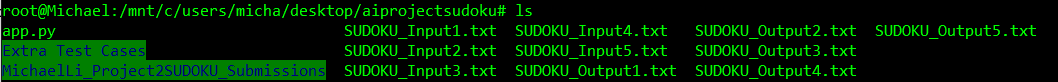
**Instructions:**

Prior to beginning, make sure that MichaelLi\_Project2SUDOKU\_Submissions.zip downloaded and extracted.

1. Open command prompt (or your OS equivalent) and navigate to the project folder.
2. Copy and paste the input files into the project folder. They should be formatted as follows:



1. Check that all input files and app.py is in the project folder.



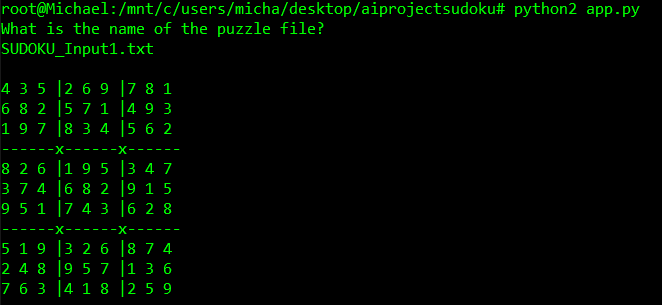
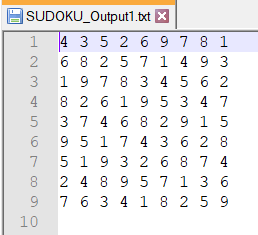
1. Once set up, run the app.py file using python2. To do so, type:

python2 app.py

1. The program will prompt you for the input file. Type in the input file’s name exactly – case sensitive.



1. Press enter and the program will run. It will print the solution on screen and generate a corresponding output file.

**Output files:**

SUDOKU\_Output1.txt:

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

SUDOKU\_Output2.txt:

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

SUDOKU\_Output3.txt:

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

**Source code:**

#Michael Li

#CS 4613

#Sudoku

def format\_input(file):

"""

Read file, input content as a string

"""

raw\_grid=""

for line in open(file, "r").readlines():

for ch in line:

if ch in "0123456789":

raw\_grid += ch

return raw\_grid

def cross(A, B):

"""

Cross product of elements in A and elements in B

"""

return [a+b for a in A for b in B]

class Sudoku():

def \_\_init\_\_(self, initial\_grid):

self.rows = 'ABCDEFGHI'

self.cols = '123456789'

self.setup()

self.grid = initial\_grid

self.values = self.grid\_to\_values()

def setup(self):

"""

Set up units:

boxes = each cell of puzzle

row,col

square = sub 3x3 grids

units = row,col,squares

peers = neighbors, essentially all cells but itself

"""

self.boxes = [r + c for r in self.rows for c in self.cols]

self.row = [cross(r, self.cols) for r in self.rows]

self.col = [cross(self.rows, c) for c in self.cols]

self.square = [cross(rs, cs) for rs in ('ABC','DEF','GHI') for cs in ('123','456','789')]

self.unitlist = self.row + self.col + self.square

self.units = dict((s, [u for u in self.unitlist if s in u]) for s in self.boxes) #row,column,squares

self.peers = dict((s, set(sum(self.units[s],[]))-set([s])) for s in self.boxes) #neighbors

def grid\_to\_values(self):

"""

Convert grid into a dict of {square: char} with '123456789' for empties. ex: A1: "1357"

"""

grid = {}

for val, key in zip(self.grid, self.boxes):

if val == '0':

grid[key] = '123456789' #all values at start if blank cell

else:

grid[key] = val

return grid

def values\_to\_grid(self):

"""

Convert the dictionary board representation to a string.

"""

str = []

for r in self.rows:

for c in self.cols:

v = self.values[r + c]

str.append(v if len(v) == 1 else '0')

return ''.join(str)

def remove\_digit(self, values, box, digit):

"""

Remove a digit from domain of a box.

"""

values[box] = values[box].replace(digit, '')

return values

def remove(self,values):

"""

If box has only 1 number, peers may not have this number.

"""

for box, value in values.items():

if len(value) == 1:

for peer in self.peers[box]:

values = self.remove\_digit(values, peer, value)

return values

def forward\_check(self, values):

"""

After a variable gets 1 value, update peers and check. Terminates if a box has no legal states left.

Either returns modified values or returns false if invalid domain exists

"""

made\_changes = True

while made\_changes:

pre\_solved\_vals = len([box for box in values.keys() if len(values[box]) == 1])

values = self.remove(values)

# Check how many boxes have a determined value, to compare

post\_solved\_vals = len([box for box in values.keys() if len(values[box]) == 1])

# If no new values were added, stop the loop.

made\_changes = pre\_solved\_vals != post\_solved\_vals

# If any box have an invalid domain, puzzle has no solution.

if len([box for box in values.keys() if len(values[box]) == 0]):

return False

return values

def MRV\_and\_degree(self, values):

"""

Picks next box to assign using:

MRV - minimum remaining values -> box with smallest domain

degree - if tied, pick box with highest number of unassigned neighbors

"""

min\_numoptions = min((len(values[s])) for s in self.boxes if len(values[s]) > 1)

#print(min\_numoptions)

tieslist = [(box,self.degree(box,values)) for box in values if len(values[box]) == min\_numoptions]

#print("min"+ min(tieslist,key = lambda deg: deg[1])[0])

return min(tieslist,key = lambda deg: deg[1])[0]

def degree(self,box,values):

"""

Returns how many unassigned neighbors for a box : ex - A1 has 4 unassigned neighbors, return 4

"""

#print(len([neighbor for neighbor in self.peers[box] if len(values[neighbor]) > 1]))

return len([neighbor for neighbor in self.peers[box] if len(values[neighbor]) > 1])

def backtrack\_search(self, values):

"""

Does the backtrack algorithm to "guess" next unassigned variable

MRV and degree heuristic

ORDER-DOMAIN-VALUES already ordered from smallest to largest

INFERENCE => used forward\_check

"""

values = self.forward\_check(values) #forward checking to reduce again

if values is False:

return False # Invalid domain through forward checking

if all(len(values[s]) == 1 for s in self.boxes):

return values #All boxes have 1 number => solved

# SELECT-UNASSIGNED-VARIABLE -> used MRV and highest degree

box = self.MRV\_and\_degree(values)

for value in values[box]: #already ordered from smallest to largest, ex: A1: 1257 -> ORDER-DOMAIN-VALUES

#print(values[s])

new\_sudoku = values.copy()

new\_sudoku[box] = value

guess = self.backtrack\_search(new\_sudoku)

if guess:

return guess

def display(self):

"""

Display the values as a 2-D grid.

"""

width = 1 + max(len(self.values[s]) for s in self.boxes)

line = 'x'.join(['-'\*(width\*3)]\*3)

for r in self.rows:

print(''.join(self.values[r+c].center(width)+('|' if c in '36' else '')

for c in self.cols))

if r in 'CF': print(line)

print

def check\_solved(self, values):

"""

If all domains only have 1 possible value, puzzle is solved, else false.

"""

if values == None: #Forward\_checking determines that values state is invalid -> set false, check if false here.

return False

for box in values.keys():

if len(values[box]) != 1:

return False

return True

def make\_outputfile(self, solved\_status, filename):

"""

Make output file based on what input user specified. Case sensitive on Windows... not sure about other systems.

"""

filename = filename.split(".")

filename[0] = filename[0].replace("Input","Output")

str\_filename = "."

str\_filename = str\_filename.join(filename)

# print(str\_filename)

f = open(str\_filename,"w+")

if(solved\_status):

string\_rep = self.values\_to\_grid()

ptr = 0

for row in range(0,9):

for col in range(0,9):

f.write(string\_rep[ptr]+ " ")

ptr += 1

f.write("\r\n") #windows compatiable formatting...

else:

f.write("Unable to solve this puzzle.")

f.close()

def solve(self):

"""

Solve puzzle:

1) Forward checking

2) Backtrack Algo using MRV and degree, forward check again

3) Check if solved

4) Print and make output files if solved

"""

#first step: forward check, if invalid state occurs, terminate

if(self.forward\_check(self.values) == False):

print("Unable to solve this puzzle.")

return

#if forward check solved puzzle, we can stop there

if self.check\_solved(self.values):

#print("I'm done checking!")

self.display()

return

#if not solved, let's backtrack search

puzzle.values = self.backtrack\_search(self.values)

#after backtrack\_searching, puzzle may or may not be solved depending on if there is a solution

#we check:

if self.check\_solved(self.values):

self.display()

else:

print("Unable to solve this puzzle.")

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

"""

Take user input for puzzle to solve

Solve puzzle

Display on screen

Make outputfile

"""

user\_input = raw\_input("What is the name of the puzzle file?\n")

puzzle = Sudoku(format\_input(user\_input))

print

# print("Grid of possible vals: \n")

# puzzle.display()

# print("After a forward check: \n")

# puzzle.forward\_check(puzzle.values)

# puzzle.display()

# print("After backtrack\_search: \n")

# puzzle.values = puzzle.backtrack\_search(puzzle.values)

# puzzle.display()

# print(puzzle.check\_solved(puzzle.values))

puzzle.solve()

if puzzle.check\_solved(puzzle.values):

puzzle.make\_outputfile(True, user\_input)

else:

puzzle.make\_outputfile(False, user\_input)