Solving Sudoku

Team #: 34

Team Members:

Andy Corliss (7th grade)

Max Corliss (8th grade)

Phillip Ionkov (7th grade)

Ming Lo (6th grade)

Team Mentor:

Li-Ta Lo

Latchesar Ionkov

School: Los Alamos Middle School/Aspen Elementary School

Area of Science: Game Theory

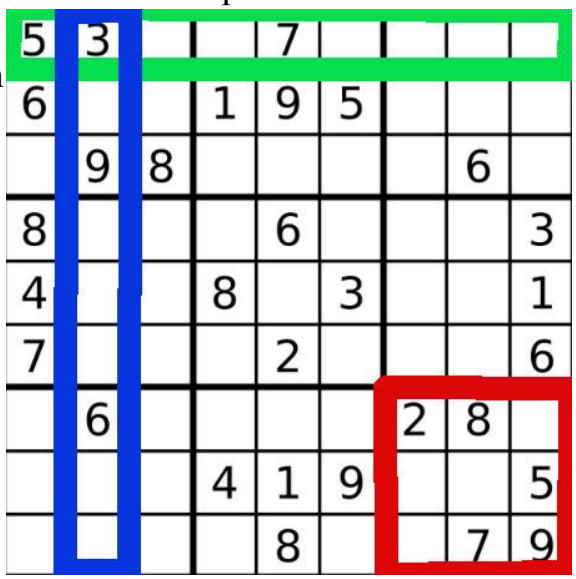
Computer Language: Python

Executive Summary:

The purpose of this project was to create a program that could solve any given Sudoku puzzle. Our solver can solve and graphically present any 4x4 or 9x9 Sudoku puzzle, and could theoretically solve larger puzzles, such as 16x16 or 25x25 puzzles, however, these puzzles were too large to complete within a reasonable timeframe with our code. Sudoku is a logic puzzle where each row, column, and box is filled with non-repeating numbers. It is set on a board with NxN squares where N is a number with a whole-number square root (4x4, 9x9, etc.) “Well-formed” Sudoku puzzles can only have one solution. Our solver uses a backtracking method in which possible solutions for each square are tested and removed if they are found to be incorrect later in the puzzle. This year, we have created a program that can solve puzzles manually entered or read from a text file, hence achieving our goal of creating a Sudoku solver that can solve any Sudoku puzzle.

Introduction:

Sudoku is a logic based puzzle in which you have to fill each square with numbers so that there are no numbers repeating in each box, row and column.

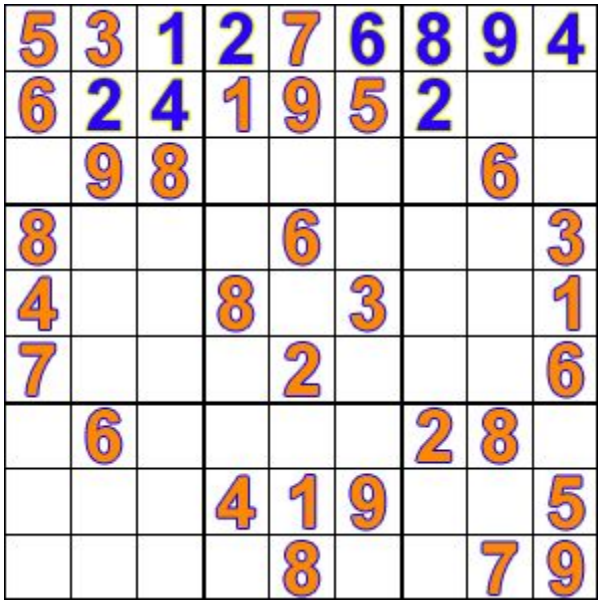


Row
Column
Box

Sudoku is normally played on a 9x9 board, but 4x4, 16x16, 25x25, and much larger boards exist. The puzzle is set up with several numbers already entered into the puzzle, and then you solve from there. The first variants of these types of puzzles appeared in 1895 in French newspapers, and the first form of Sudoku we solve today was introduced in a Dell magazine in the 1970s.

Description:

The first step in order to solve the puzzle is to make a board. Our board is a 2D array, and each element in the list is either an empty space or a number. We wrote a function, `printboard`, that



creates the board using lines and dashes, producing a graphical representation. This function helped us visualize the problem and how to solve it. We then used several other functions (`notOnRow`, `notOnCol`, and `notInSquare`) that check to see what numbers could be placed in a certain square and are able to fit in the applicable rows, columns and boxes. A function called `findEmpty` searches each row and checks to see if any squares are vacant.

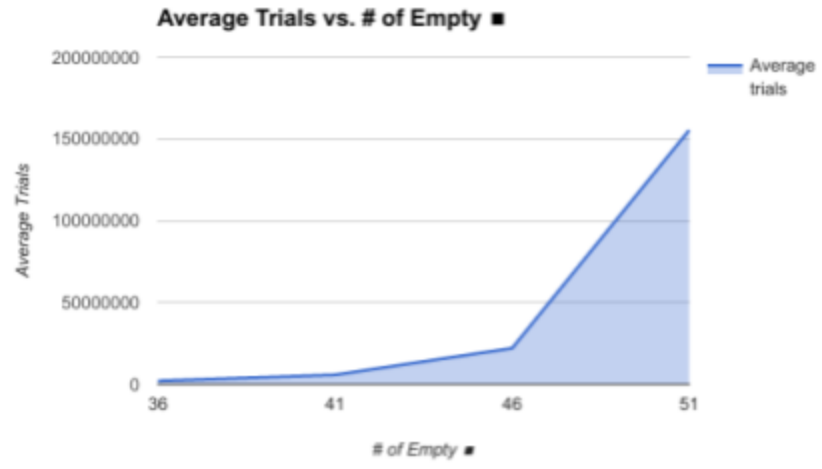
We then used a method called backtracking to solve the Sudoku puzzles. Backtracking algorithms iterate through possible solutions. For each empty square found by `findEmpty`, solutions are eliminated if they do not fit. Once the function finds a solution for the current square, it will move on to the next square and try other possibilities. It will repeat this until there is no solution for the next empty square. If this happens, the program will go back and remove previous squares to attempt a new potential solution. This will continue until the puzzle is solved, or when no empty squares remain.

Now that we had a method to solve the puzzle, we could start using downloaded text files to test the code and analyze the results. To do this, we had to make the program read each line of the text file that we chose. We ‘‘decoded’’ the line into the board for the program to solve.

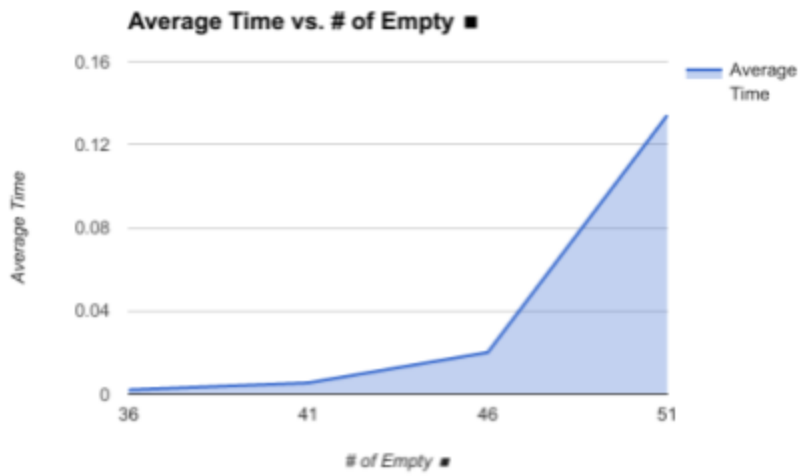
We wanted to see how long it would take for our program to solve the puzzles, so we imported the time module and timed the computer. We then compared the times by finding the averages and steps of backtracking.

Results:

We downloaded the Sudoku puzzles as text files from: http://www.printable-sudoku-puzzles.com/wfiles/. Each text file contains 10,000 puzzles. There are 36 empty spaces in the first text file, 41 in the second, 46 in the third, and 51 in the fourth. These puzzles were tested on a MacBook Air in Python 3, and results could vary depending on the type of computer and conditions. The graphs below are very similar, showing that the amount of blank squares strongly with the difficulty of the puzzles as shown by average trials, time, and fails. The graphs also show that the amount of time to solve a puzzle with differ exponentially with the number of empty squares.



The amount of trials is the number of times the computer recursed through the solve function.



The amount of time is the number of seconds it took the computer to solve a problem.

The number of fails is the amount of times the computer needs to remove a number and go back.

Conclusion:

This year, we have created a program that will solve a Sudoku puzzle. We can now manually input a puzzle into our program and solve it with the click of a button. We completed a major step that enables us to take numbers from text files that we downloaded from the internet, form Sudoku puzzles from them, and solve them. We have created a program that will show how the computer solves the Sudoku puzzle by using backtracking, and have made a counter for how many times it uses backtracking. We have learned much more about recursion, backtracking and the Python programming language. We have also learned more about Sudoku, game theory, and how to solve complex puzzles.

Acknowledgements:

We have had so much support during this Supercomputing season. First, we have to thank our mentor, Li-Ta Lo, for frequently helping us with all of our work, especially when we had obstacles to overcome and problems to solve. We also thank our parents for investing their time to bring us to meetings and encouraging us to improve our project. We also would like to express gratitude towards our Interim Report judges, who gave us expert and professional insight to help us to continue and improve our project to reach our goals.

Appendix:

Timer Code:

```
import math
import random
import time
import matplotlib.pyplot as plt
import numpy as np

d = 9
sqrd = int(math.sqrt(d))

board = [[' ' for i in range(d)] for i in range(d)]

def filedecode():
    global sudokuFile
    currentnum = 0
    for i in range(d):
        for j in range(d):
            if sudokuFile[currentnum] == '0':
                board[i][j] = ' '
            elif sudokuFile[currentnum].isdigit():
                board[i][j] = int(sudokuFile[currentnum])
            currentnum = currentnum + 1

def notOnRow(row, num):
    for i in range(d):
        if board[row][i] == num:
            return False
    return True


def notOnCol(col, num):
    for i in range(d):
        if board[i][col] == num:
            return False
    return True


def notInSquare(row, col, num):
    row = row - (row % sqrd)
    col = col - (col % sqrd)
```

```
    for i in range(sqrd):
        for j in range(sqrd):
            if board[row + i][col + j] == num:
                return False
    return True


def isPossible(row, col, num):
    if notOnRow(row, num) and notOnCol(col, num) and notInSquare(row, col, num):
        return True
    else:
        return False


def findEmpty():
    for row in range(d):
        for col in range(d):
            if board[row][col] == " ":
                return (row, col)
    return (-1, -1)


ncalls = 0
ntrials = 0
nfails = 0

total_ncalls = []
total_ntrials = []
total_nfails = []

def solve():
    global ncalls, ntrials, nfails
    ncalls = ncalls + 1
    (row, col) = findEmpty()
    if (row, col) == (-1, -1):
        return True
    for i in range(1, d + 1):
        ntrials = ntrials + 1
        if isPossible(row, col, i):
            board[row][col] = i
            if solve():
                return True
            else:
```

```
                board[row][col] = " "
                nfails = nfails + 1
    return False


times = []

def timeit(num):
    global times, ntrials, total_ntrials, ncalls, total_ncalls, nfails, total_nfails

    times = []
    total_ntrials = []
    total_nfails = []
    total_ncalls = []
    ntrials = 0
    nfails = 0
    ncalls = 0

    global sudokuFile

    for i in open('/Users/ming/Downloads/' + str(num) + '.txt', 'r'):
        sudokuFile = i
        filedecode()
        #printboard()
        start = time.time()
        solve()
        end = time.time()
        #printboard()
        time1 = end - start
        #print(time1)

        times.append(time1)
        total_ntrials.append(ntrials)
        total_nfails.append(nfails)
        total_ncalls.append(ncalls)

    print("The total time was : " + str(sum(times)) + " seconds")
    print("The max time for each puzzle was %f" % max(times))
    print("The average time for each puzzle was " + str(sum(times ) /len(times)) + " seconds")
    print("Average number of trials " + str(np.mean(total_ntrials)))
    print("Average number of fails " + str(np.mean(total_nfails)))
    print("Average number of calls " + str(np.mean(total_ncalls)))
    #plt.hist(times, bins=10)
```

```
    #plt.show()

    #times.sort(reverse=True)
    #print(times[0:10])

if __name__ == '__main__':
    #timeit(1)
    for i in [0, 1, 2, 3, 5]:
        timeit(i)
```

Sudoku Visual Representation:

```
import math
import turtle
t = turtle.Turtle()
t.speed(0)
t.penup()
t.goto(-200, -200)
t.pendown()
y = 0
recursions = 0
d = 9
sqrd = int(math.sqrt(d))

def write_square(row, col, num):
    t.goto(-200 + (400 / 9) * col + (400 / 30), 200 - (400 / 230) - (400 / 9) * (row + 1))
    t.color("black")
    t.write( num , align = "left", font = ("Arial", 25, "normal"))

def erase_square(row, col, num):
    t.goto(-200 + (400 / 9) * col + (400 / 30), 200 - (400 / 230) - (400 / 9) * (row + 1))
    t.color("white")
    t.write(num , align = "left", font = ("Arial", 25, "bold"))

def notOnRow(row, num):
  for i in range(d):
    if board[row][i] == num:
      return False
  return True

def notOnCol(col, num):
  for i in range(d):
    if board[i][col] == num:
      return False
```

```
  return True

def notInSquare(row, col, num):
  row = row - (row % sqrd)
  col = col - (col % sqrd)
  for i in range(sqrd):
    for j in range(sqrd):
      if board[row + i][col + j] == num:
        return False
  return True

def findEmpty():
  for row in range(d):
    for col in range(d):
      if board[row][col] == " ":
        return (row, col)
  return (-1, -1)

def isPossible(row, col, num):
  if notOnRow(row, num) and notOnCol(col, num) and notInSquare(row, col, num):
    return True
  else:
    return False

def solve():
  global y
  y = y + 1
  (row, col) = findEmpty()
  if (row, col) == (-1, -1):
    return True
  for i in range(1, d + 1):
    global recursions
    recursions = recursions + 1
    if isPossible(row, col, i):
      board[row][col] = i
      write_square(row, col, i)
      if solve():
        return True
      else:
        board[row][col] = " "
        erase_square(row, col, i)
  return False
```

```
board = [[' ' for i in range(d)] for i in range(d)]
def customsetup():
    board[1][0] = 7
    board[3][0] = 1
    board[4][0] = 5
    board[7][0] = 6
    board[8][0] = 8

    board[0][1] = 5
    board[2][1] = 8
    board[5][1] = 7

    board[1][2] = 2
    board[4][2] = 4
    board[8][2] = 3

    board[7][3] = 1
    board[8][3] = 9

    board[1][4] = 3
    board[2][4] = 6
    board[5][4] = 1
    board[6][4] = 4
    board[7][4] = 7

    board[0][5] = 7
    board[3][5] = 5
    board[4][5] = 9
    board[7][5] = 8

    board[1][6] = 4
    board[2][6] = 5

    board[1][7] = 9
    board[3][7] = 3
    board[4][7] = 2
    board[5][7] = 8
    board[8][7] = 1

    board[6][8] = 2
    board[7][8] = 9
    board[8][8] = 6
```

```
def drawboard(size, dim):
    for i in range(4):
        t.forward(size)
        t.left(90)
    for i in range(int(dim / 2)):
        t.forward((size / dim))
        t.left(90)
        t.forward(size)
        t.right(90)
        t.forward(size/dim)
        t.right(90)
        t.forward(size)
        t.left(90)
    t.penup()
    t.goto(-200 + size, -200)
    t.pendown()
    t.left(90)
    for i in range(int(dim / 2)):
        t.forward((size / dim))
        t.left(90)
        t.forward(size)
        t.right(90)
        t.forward(size/dim)
        t.right(90)
        t.forward(size)
        t.left(90)

drawboard(400, 9)
customsetup()
t.pencolor("blue")
t.penup()
t.goto(-200, 200)
t.seth(0)
for i in range(9):
    t.penup()
    t.goto(-200 + 400 / 30, 200 - (400 / 9) * (i + 1) - 400 / 230)
    for j in range(9):
        t.write( board[i][j] , align = "left", font = ("Arial", 25, "normal"))
        t.forward(400 / 9)
solve()
t.pencolor("black")
t.seth(0)
```

References:

1. https://en.wikipedia.org/wiki/Sudoku
2. https://en.wikipedia.org/wiki/Sudoku_solving_algorithms
3. https://en.wikipedia.org/wiki/Recursion
4. https://en.wikipedia.org/wiki/Backtracking
5. https://web.stanford.edu/class/cs209/lectures/CS209-Lecture-02-Recursion.pdf
6. https://www.python.org/
7. https://repl.it/
8. http://www.printable-sudoku-puzzles.com/wfiles/