Sudoku Puzzles — Creation and Heuristic Calculation

University of Minnesota: MATH 4428
Zhaomeng Chen and Omar Elamri

Introduction

Sudoku puzzles were invented in the 1700's by the Swiss mathematician, Leonhard Euler, and have become widely popular in recent times. The format of these puzzles is very simple: given an n by n grid, with some squares filled in with a number between 1 and n, the player tries to find an arrangement for the grid such that no number repeats twice in the column or row. If such arrangement exists, it is said to be a *solution* of the puzzle. Furthermore, if there is only one solution, then the solution is *unique*.

Often times, there are additional constraints that are introduced to increase difficulty, such as cutting up the grid into subgrids of n boxes, which must also contain no repeats.

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

Figure 1: Example of a Sudoku puzzle

We are tasked to develop an algorithm which constructs Sudoku puzzles with unique solutions of 4 difficulty levels and to minimize the complexity of the algorithm. In order to do this, we propose the following research plan:

Research Plan

Our research plan is divided into five phases:

1. **Creating Completed Grids**: We will develop an algorithm that yields a basic 9 x 9 Sudoku grid. Numbers will be randomized using the Unix *urandom* binary to ensure a reasonable degree of randomness.

2. **Generating Seed Puzzles**: Using the "dig-in" algorithm, Sudoku puzzles with varying degrees of difficulty will be created. One of the limitations of Zhang's algorithm is that the highest difficulty left 22 numbers on the grid. However, the theoretical minimum number of numbers in a 9 x 9 grid that has a unique solution is 17 (Zhang, p. 4). Our goal for this project is for our algorithm to utilize only 17 numbers at the maximum difficulty. In this step, we will generate 4 puzzles of varying difficulty, which will be determined in phase 3.

3. **Designating Heuristics to Assess Difficulty**: To assess difficulty of our puzzles, it is necessary to develop some sort of heuristic that ranks them. (Possibly on a scale from 0-100). Factors that could rank a puzzle based on difficulty will be assessed.

4. **Randomizing Seed Puzzles**: We will develop a randomizing algorithm that takes a seed puzzle and scrambles it while maintaining an unique solution, efffectively creating a new puzzle.

5. **Creating an Interactive Version**: Finally, a long term goal for our project is to create a playable version online alongside **a research paper that highlights our findings**.

Phase 1: Creating Completed Grids

We developed an algorithm that creates a 9 x 9, solved Sudoku grid. This is integral to creating an unsolved puzzle—which we will develop in phase 3. A proof of concept in Python is available in the appendix. Our algorithm is fairly simple:

1. Initialize a 9 x 9 matrix
2. Traverse through each entry in the matrix
3. Get the numbers already used in the entry's row, column and neighborhood (3 x 3 group)
4. From the numbers not already used, randomly fill in the entry
5. If all numbers 1-9 are already used, this particular puzzle is impossible to solve
6. Start at step 1 again until step 5 is no longer applicable

We find that, on average $(n = 76)$, it takes 257 reattempts (step 6) to create a solved puzzle using our algorithm. If time allows, we will revisit this algorithm to reduce its reliance on brute-force randomness.

Phases 2 and 3: Generating Seed Puzzles and Assessing Difficulty

Now that we have a solved Sudoku grid, we now need to convert it into a puzzle. We did this by digging holes into the puzzle at random, replacing the number with a 0 until there are a given number of nonzero numbers left in the grid. However, the algorithm is selective when determining whether or not to dig an entry. The algorithm is as follows:

1. Find a random cell
2. Replace the cell's entry with 0
3. Get the numbers already used in the entry's row, column and neighborhood (3 x 3 group)
4. The only number that is not in the list of numbers already used should be the original entry. However, if this is not true (there are more possibilities), then this entry cannot be dug in a way that guarantees unique solutions.
5. If the previous step is true, return to step 1. If not, replace the dug entry with its original value. Repeat step 1 until a given number of entries are dug.

We now have an unsolved Sudoku puzzle. To assess its difficulty, we designed a simple algorithm to analyze the puzzle.

1. Solve the puzzle with the brute force algorithm designed in phase 1
2. Collect data on how many reruns were used by the brute force algorithm to solve the puzzle
3. Determine if the "solution" matches with our original solved Sudoku grid
4. Repeat $n = 50$ times

By collecting data $(n = 50)$ on the number of cycles used by our algorithm to solve the puzzle, we can determine the true mean μ of the number of cycles needed to solve the puzzle by brute force. From our sample mean $\overline{x}$ ($\frac{\sum_i^n x_i}{n}$) and sample standard deviation σ, we can use the Central Limit Theorem to determine the confidence interval of where the true mean lies. (See Appendix for the MATLAB codes used.)

$$\mu - \overline{x} \leq |\frac{t\sigma}{\sqrt{n}}|$$

$$-\frac{t\sigma}{\sqrt{n}} + \overline{x} \leq \mu \leq \frac{t\sigma}{\sqrt{n}} + \overline{x}$$

Note that for a 95% confidence interval, $t = 2$. If function I is the normal distribution density function,

$$\int_{-t}^{t} I(x)dx = 0.95$$

While not perfect, we used this true mean to assess the overall difficulty of the puzzle. By the end, we had the following 4 seed puzzles with difficulty based off this chart:

Easy	less than 40 cycles
Medium	$40 - 200$ cycles
Hard	$200 - 500$ cycles
Extreme	more than 500 cycles

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

Easy, Mean $\in [0.35, 1.01]$

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

Medium, Mean $\in [38.40, 67.95]$

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

Hard, Mean $\in [219.17, 442.23]$

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

Extreme, Mean $\in [605.33, 998.31]$

*Note that these calculations were done on an Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz
*All confidence intervals were computed at 95% confidence

Phase 4: Randomizing Seed Puzzles

From now on, we will refer to a group of 3 columns containing 3 neighborhoods as a **band** and a group of 3 rows containing 3 neighborhoods as a **stack**. We developed an algorithm that applied transformations to seed puzzles to randomize it, essentially creating a new puzzle, while preserving uniqueness. The transformations are as follows:

- Counterclockwise rotations by 90 degrees
- Reflections
- Transposes
- Permutation of bands
- Permutation of stacks
- Column permutations within a band
- Row permutations within a stack
- Relabeling numbers

In the end, Burnside's lemma tells us that one seed puzzle has $5,472,730,538$ essentially different puzzles. Therefore, given that we have seed puzzles, which we completed in phase 2, we can generate new puzzles of similar difficulty at the run-time complexity of our randomization algorithm—a very optimal O(1). (Each transformation applies once to each cell in the Sudoku matrix.)

Here are some examples of transformed seed puzzles:

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

Transformed Easy

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

Transformed Medium

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

Transformed Hard

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

Transformed Extreme

Conclusion

The methodology described above succeeds in multiple ways. 9 x 9 Sudoku grids can be generated at 4 different difficulty levels: Easy, Medium, Hard, and Extreme. Each level's difficulty is verified by how many cycles it takes a computer to solve it. Generating puzzles from these difficulty levels is incredibly fast and efficient—originating from seed puzzles, essentially different puzzles are calculated with minimal overhead.

However, the initial generation of seed puzzles has a lot left to be desired. Since the development of these puzzles is solely verified based on brute-force cycles, this method is not very efficient. Fortunately, this process only has to be run once for each difficulty level, but adding a feature for dynamic difficulty levels would prove to be costly. In addition, our heuristics for determining difficulty may not be indicative of human difficulty. Our Extreme puzzle only has half the puzzle left—Sudoku puzzles are known to have at the very least 17 entries filled in.

Nonetheless, it is clear that our methodology successfully creates Sudoku puzzles at minimal recurring computational cost. While phase 5 was not reached, the algorithm can trivially be exported as a JavaScript web app.

References

Wikipedia. (n.d.). Sudoku puzzle. Sudoku solving algorithms. Retrieved March 22, 2022, from https://en.wikipedia.org/wiki/Sudoku_solving_algorithms.

Wikipedia. (n.d.). Mathematics of Sudoku. Essentially different solutions. Retrieved April 30, 2022, from https://en.wikipedia.org/wiki/Mathematics_of_Sudoku#Essentially_different_solutions

Zhang, X.-S. (n.d.). Sudoku puzzles generating: From easy to evil - APORC. Retrieved March 23, 2022, from http://zhangroup.aporc.org/images/files/Paper_3485.pdf

Appendix

All the code below is also available on GitHub: https://github.com/eado/sudokool

Phase 1 Python Code

main.py

```
from random import randint, sample

# sudoku puzzles can be separated into groups of 3 x 3 squares
GROUPS = 3
NTH = GROUPS * 3

# initialize puzzle matrix with all zeros
def genZeros():
    return [[0 for _ in range(0, NTH)] for _ in range(0, NTH)]

puzzle = genZeros()
savedPuzzle = genZeros()
initialPuzzle = genZeros()

# transfer puzzle data from one matrix reference to another
def transferPuzzle(initial, new):
    for row in range(0, NTH):
        for column in range(0, NTH):
            new[row][column] = initial[row][column]

# reset puzzle reference to all zeros
def setPuzzle():
    for row in range(0, NTH):
        for column in range(0, NTH):
            puzzle[row][column] = 0

# save digged puzzle
def savePuzzle(): transferPuzzle(puzzle, savedPuzzle)
```

```
# restore digged puzzle
def restorePuzzle(): transferPuzzle(savedPuzzle, puzzle)

# save initially solved puzzle
def saveInitialPuzzle(): transferPuzzle(puzzle, initialPuzzle)

# check if computer-solved solution is the same as the initial solution
def sameSolution():
    for row in range(0, NTH):
        for column in range(0, NTH):
            if puzzle[row][column] != initialPuzzle[row][column]:
                return False
    return True

# pretty print puzzle
def printPuzzle():
    zeros = 0
    for (r, row) in enumerate(puzzle):
        print("-" * (NTH * 2 + 7)) if r % 3 == 0 else None
        for (c, entry) in enumerate(row):
            print("| ", end="") if c % 3 == 0 else None
            num = "_" if entry == 0 else str(entry)
            zeros += 1 if entry == 0 else 0
            print(num + " ", end="")
        print("|")
    print("-" * (NTH * 2 + 7))
    print("Blanks: {}, Entries: {}\n\n".format(zeros, 81 - zeros))

def getNeighborhood(r, c):
    rx = int(r / GROUPS)
    cx = int(c / GROUPS)
    neigh = []
    for rn in range(rx * GROUPS, (rx * GROUPS) + GROUPS):
```

```
        for cn in range(cx * GROUPS, (cx * GROUPS) + GROUPS):
            neigh.append(puzzle[rn][cn])
    return neigh

# for a certain position in the matrix,
# get the possible valid numbers to fill
def getAppNumbers(r, c):
    arr = [x for x in range(1, NTH + 1)]
    # get row
    row = puzzle[r]
    arr = list(filter(lambda x: x not in row, arr))
    # get columns
    col = list(map(lambda row: row[c], puzzle))
    arr = list(filter(lambda x: x not in col, arr))
    # get neighborhood
    neigh = getNeighborhood(r, c)

    arr = list(filter(lambda x: x not in neigh, arr))

    return arr

# for each entry in the matrix, get the applicable numbers
# if no applicable numbers exist, restart the process
def createSolvable(restore = False):
    fails = 0
    failed = False
    while True:
        for row in range(0, NTH): # for each neighborhood
            for column in range(0, NTH):
                if puzzle[row][column] != 0:
                    # if cell is already filled, don't retry
                    continue
                nums = getAppNumbers(row, column) # get applicable numbers
                # no available numbers
```

```
                if (len(nums) == 0):
                    restorePuzzle() if restore else setPuzzle()
                    # revert back to initial puzzle
                    fails += 1 # count number of cycles
                    failed = True # break out of loop
                    break
                else:
                    index = randint(0, len(nums) - 1)
                    # choose a random number to fill
                    puzzle[row][column] = nums[index]
            if failed:
                break
        if failed:
            failed = False
        else:
            break
    return fails


def dig():
    dug = 0
    it = 0
    while dug < 45: # assert that the number of cells dug
        # is at least 45 (this number changes based on difficulty)
        it += 1
        if it > 100: # try to randomly dig 100 times.
            # if we still haven't reached the optimum number, restart
            transferPuzzle(initialPuzzle, puzzle)
            it = 0
            dug = 0
        for row in sample(range(0, GROUPS), GROUPS - 1):
            for col in sample(range(0, GROUPS), GROUPS - 1):
                # choose a random neighborhood
                nums = sample(range(1, NTH), NTH - 1)
                for num in nums:
```

```
                    r = row * GROUPS + (num % GROUPS)
                    # randomize neighborhood order
                    c = col * GROUPS + (num // GROUPS)
                    entry = puzzle[r][c]
                    if entry == 0: # no need to repeat dig
                        continue
                    puzzle[r][c] = 0 # attempt dig
                    appnums = getAppNumbers(r, c)
                    if len(appnums) == 1:
                        # if there are multiple solutions, don't dig
                        dug += 1
                        break
                    else:
                        puzzle[r][c] = entry # place entry back


print("Generating:")
print("Cycles: {}".format(createSolvable()))
saveInitialPuzzle()
printPuzzle()

print("Digging:")
dig()
savePuzzle()
printPuzzle()

cycles = []
for _ in range(0, 50):
    print("Solving: ")
    cycle = createSolvable(True)
    cycles.append(cycle)
    print("Cycles: {}".format(cycle))

    print("Equal: {}".format(sameSolution()))
```

```
        restorePuzzle()

printPuzzle()
print(cycles)
```

Phase 4 Seed Generation

seed.py

```
# sudoku puzzles can be separated into groups of 3 x 3 squares
from random import sample, randint


GROUPS = 3
NTH = GROUPS * 3

puzzle = []

# initialize puzzle matrix with all zeros
def genzeros():
    return [[0 for _ in range(0, NTH)] for _ in range(0, NTH)]

# transfer puzzle data from one matrix reference to another
def transferpuzzle(initial, new):
    for row in range(0, NTH):
        for column in range(0, NTH):
            new[row][column] = initial[row][column]

def printPuzzle():
    zeros = 0
    for (r, row) in enumerate(puzzle):
        print("-" * (NTH * 2 + 7)) if r % 3 == 0 else None
        for (c, entry) in enumerate(row):
            print("| ", end="") if c % 3 == 0 else None
            num = "_" if entry == 0 else str(entry)
            zeros += 1 if entry == 0 else 0
            print(num + " ", end="")
        print("|")
    print("-" * (NTH * 2 + 7))

lines = []
```

```
while True:
    try:
        line = input()
        if line:
            lines.append(line)
        else:
            break
    except EOFError:
        break

for row in lines:
    if row == "-" * (NTH * 2 + 7):
        continue
    stringnums = list(filter(lambda x: x != "␣" and x != "", "".join(row.split("|")).s
    nums = list(map(lambda x: 0 if x == "_" else int(x), stringnums))
    puzzle.append(nums)

# tokenizing
tokens = sample(range(1, NTH + 1), NTH)
print(tokens)

for row in range(0, NTH):
    for column in range(0, NTH):
        val = puzzle[row][column]
        if val == 0:
            continue
        puzzle[row][column] = tokens[val - 1]

# randomize stacks
def swaprows(r1, r2):
    puzzle[r1], puzzle[r2] = puzzle[r2], puzzle[r1]

def swapcols(c1, c2):
    for row in range(0, NTH):
```

```
        puzzle[row][c1], puzzle[row][c2] = puzzle[row][c2], puzzle[row][c1]

if randint(0, 1) == 0:
    swaprows(0, 3)
    swaprows(1, 4)
    swaprows(2, 5)

if randint(0, 1) == 0:
    swaprows(3, 6)
    swaprows(4, 7)
    swaprows(5, 8)

if randint(0, 1) == 0:
    swaprows(6, 0)
    swaprows(7, 1)
    swaprows(8, 2)

# randomize bands
if randint(0, 1) == 0:
    swapcols(0, 3)
    swapcols(1, 4)
    swapcols(2, 5)

if randint(0, 1) == 0:
    swapcols(3, 6)
    swapcols(4, 7)
    swapcols(5, 8)

if randint(0, 1) == 0:
    swapcols(6, 0)
    swapcols(7, 1)
    swapcols(8, 2)

# randomize rows
```

```
for row in range(0, GROUPS):
    if randint(0, 1) == 0:
        swaprows(3 * row + 0, 3 * row + 1)
    if randint(0, 1) == 0:
        swaprows(3 * row + 1, 3 * row + 2)
    if randint(0, 1) == 0:
        swaprows(3 * row + 2, 3 * row + 0)

# randomize cols
for col in range(0, GROUPS):
    if randint(0, 1) == 0:
        swapcols(3 * col + 0, 3 * col + 1)
    if randint(0, 1) == 0:
        swapcols(3 * col + 1, 3 * col + 2)
    if randint(0, 1) == 0:
        swapcols(3 * col + 2, 3 * col + 0)

# randomize transpose
if randint(0, 1) == 0:
    trans = genzeros()
    for row in range(0, NTH):
        for column in range(0, NTH):
            trans[row][column] = puzzle[column][row]

    transferpuzzle(trans, puzzle)

# randomize reflection rows
if randint(0, 1) == 0:
    swaprows(0, 8)
    swaprows(1, 7)
    swaprows(2, 6)
    swaprows(3, 5)

# randomize reflection cols
```

```
if randint(0, 1) == 0:
    swapcols(0, 8)
    swapcols(1, 7)
    swapcols(2, 6)
    swapcols(3, 5)

# randomize rotation
if randint(0, 1) == 0:
    puzzle = [[puzzle[j][i] for j in range(len(puzzle))] for i in range(len(puzzle[0])

printPuzzle()
```

Confidence Interval Calculation

confidenceinterval.m

```
x = [];                                  % Data
SEM = std(x)/sqrt(length(x));            % Standard Error
ts = tinv([0.025  0.975],length(x)-1);   % T-Score
CI = mean(x) + ts*SEM;                   % Confidence Intervals
```