Solving Sudoku Puzzles Using Backtracking and Simulated Annealing Algorithms

Aditya Samaroo
College of Engineering
Northeastern University

Abstract

Sudoku is a popular puzzle around the world as well as in the programming community. There simple rules and only a single solution for each unique puzzle which can be handled using deduction. The objective of this paper is to build two very different algorithms to solve three Sudoku puzzles with varying difficulty. The performance of each algorithm will be based on its time, and things specific to each algorithm such as number of times the function is called recursively for the backtracking algorithm and the number of iterations for the simulated annealing algorithm. While one algorithm is tried and true for solving Sudoku puzzles, the other provides a more random approach. The results prove what is already known about the backtracking algorithm but also exposes an issue when it comes to how long the simulated annealing algorithm takes to converge.

Introduction

Sudoku is a logic based number placement puzzle. The objective is to fill a 9×9 grid with digits such that each row, column and 3×3 sub-grid contains all of the digits from 1 to 9 without repeating. The difficulty of the puzzle depends on the numbers in the grid at the beginning of the puzzle and where they are placed. Each Sudoku puzzle contains a single solution. In terms of artificial intelligence Sudoku is a deterministic, static and discrete environment where a single agent places numbers in the correct order to gain a valid solution.

A variety of algorithms can be used to solve Sudoku puzzles but choosing the right one depends on the users limitations such as time and memory. Algorithms such as breadth-first search (BFS), depth-first search (DFS) and backtracking can be used but each have their drawbacks. More complex algorithms such as random hillclimb, simulated annealing can also solve Sudoku puzzles but do introduce the stochastic variable to the environment, which can either benefit or impede the algorithms solution time.

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

Figure 1: Unsolved Sudoku Board

Background

Search algorithms such as BFS and DFS can be used to solve Sudoku puzzles. Each space in the grid is a node with a possible solution of digits 1 to 9. The state space in this case is $O(b^m)$ without pruning which is equal to 9^{81} or approximately $1.966270505 * 10^{77}$ which can pose a problem to users in terms of time and available memory. By pruning each node to only allow valid solutions, the state space can be reduced more and more as the tree is expanded and values are placed in the correct spaces. Algorithm 1 shows the pseudocode that can be used in order to solve a Sudoku puzzle.

Related Work

In a paper by Mulisu and Winter, they used a hybrid approach for solving Sudoku puzzles which involves a combination of constraint programming and local search (Musliu and Winter 2017). In their approach, they reduce the domains for each cell variable until all variables are arc consistent before performing the search. This improves upon the BFS and DFS approaches by pruning the invalid solutions from each cells and reducing the state space. The downside is that problems do arise for their algorithm once the search space increases. The paper compared their algorithm's performance to that of simmulated annealing, and two different constraint programming based solvers. They

Algorithm 1 General Search

```
procedure SEARCH(problem)
    node ← a node with state = problem.Initial
    if problem.GoalTest(node.state) then
        return Solution(node)
        frontier ← a FIFO/LIFO queue
        explored ← an empty set
        while frontier.isEmpty() = FALSE do
            node ← POP(frontier)
            add node.state to explored
            for each action in problem(node.STATE)
            child ← child.node(problem, node, action)
            if child.state is not in frontier then
                if problem.GoalTest(child.State) then
                    return Solution(child)
                    frontier INSERT(child, frontier)
                end if
            end if
        end while
    end if
end procedure
```

found that their approach did well compared to the CP based solvers once the state-space was increased.

Another group used β-hill climbing to solve Sudoku puzzles (Al-Betar et al. 2017). This method is a much more stochastic approach, filling in the grid with random numbers and then changing each space over time to improve the board state. Another paper by Al-Betar mentioned various hill climbing algorithms to choose from, such as Simulated Annealing (SA), Tabu Search (TS), Greedy Randomize Adaptive Search Procedure (GRASP), Variable Neighborhood Search (VNS) and Iterated Local Search (ILS) (Al-Betar 2016). The β approach attempts to improve the board state and if the algorithm fails to do so, it compares β, which is in the domain of 0 to 1, to a random number to determine whether or not the change to the board will go through or not. This algorithm did yield a 100% success rate for a majority of the puzzles while changing the different parameters in the algorithm. One of the parameters $\mathcal{N}$, represents the neighboring operator which determines the probability of a space taking on its neighbors value while β represents the probability of a space generating a completely different value from the range 1 to 9. The algorithm was tested by varying one parameter while keeping the other constant.

Project Description

For my project, I wanted to compare two different algorithms in terms of timing to see how well they can solve a Sudoku puzzle. I chose both the backtracking algorithm as well as the simulated annealing algorithm as the most straightforward ways to do so in the given time period. Backtracking involves building a solution by going from empty space to space and filling in values that are valid and available. The algorithm backtracks when a space does not have a valid solution, replacing its previous solution with a new valid number before continuing. The algorithm is fault proof in the sense that it will only return no solution if the Sudoku board violates its own rules to begin with.

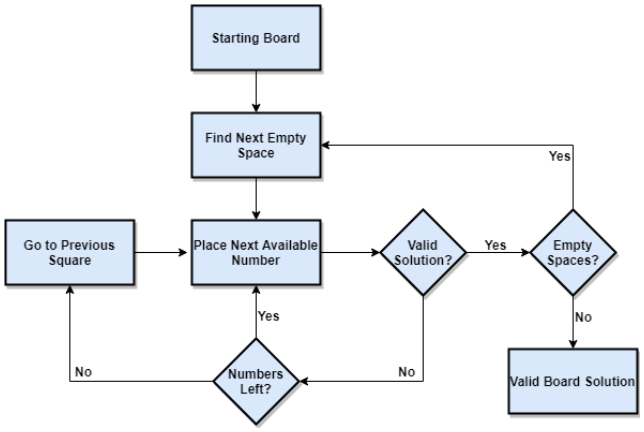

Figure 2: Backtracking Algorithm Flowchart

Algorithm 2 Backtracking Algorithm

```
procedure BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then
        return assignment
    end if
    var ← Select-Unassigned-Variable(csp) for each value in ODV(var, assignment, csp) do
    if value is consistent to assignment then
        add {var = value} to assignment
        if inferences ≠ failure then
            add inferences to assignment
            result ← BACKTRACK(assignment, csp)
            if result ≠ failure then
                return result
            end if
        end if
    end if
    remove {var = value} and inferences from assignment
    return failure
end procedure
```

Note that in the pseudo-code the solve function is called recursively with the previous information which is essentially the backtracking.

The Simulated Annealing algorithm involves filling in the starting board with random integers. It then randomly chooses two neighbors that have been filled in to be candidates for a swap. The algorithm then "scores" the new board by assigning points based on how many unique numbers there are in each column,

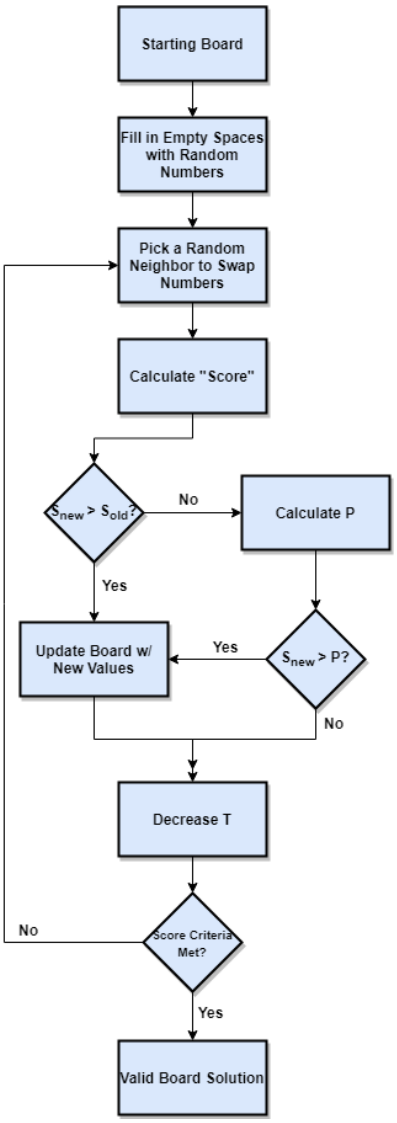

Figure 3: Simulated Annealing Algorithm Flowchart

row and sub-grid, where the lower the score is the better. If the new board score is better than the old board, then the swap goes through, otherwise a probability is calculated, given by:

$$P = e^{(\frac{\delta_s}{T})} \tag{1}$$

δ_s is the difference between the current score and the "candidate" score. If the random number is greater than the probability generated then the change goes through, otherwise the algorithm goes back to the beginning and repeats the same steps. T represents the "temperature" of the system, where the "warmer" the system is, the more volatility and stochasticity there is. As the system "cools" the number of changes that goes through decreases until a global minimum is reached.

I did not choose to include BFS or DFS since the backtracking algorithm incorporates some of the ideas

Algorithm 3 Simulated Annealing Algorithm

```
procedure SIMULATED-ANNEALING(problem)
    returns a state that is a local minimum
    T ← temperature
    while criteria is not met do
        Pick random neighbor
        S_new ← neighbor(s)
        if S_new ≥ S_old then
            S ← S_new
        else if P ≥ random(0, 1) then
            S ← S_new
        else
            No change in S
        end if
        T ← cooler
    end while
    return solution
end procedure
```

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

Figure 4: Easy Sudoku

from the search algorithms. Instead of presenting all possible solutions in a node, the backtracking algorithm already prunes invalid solutions from the node so the state space is that much smaller to begin with.

Experiment

Three different Sudoku puzzles were generated from an online source with three different difficulties: easy, medium, and hard. The easy difficulty puzzle contains 54 empty spaces, the medium puzzle has 59 empty spaces and the hard puzzle has 56 empty spaces. Although the hardest difficulty puzzle has less empty spaces than that of the medium puzzle, the placement of the numbers in the starting grid also plays a role in how difficult a puzzle may be.

To measure performance of each algorithm I recorded the amount of time each code took for each of the puzzles. For the backtracking puzzle specifically, I also recorded the number of times the algorithm backtracked before returning a valid solution. For the simulated annealing algorithm, I recorded the score every 1000 iterations, as well as the best score that it had achieved, capping the number of iterations at 400,000

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

Figure 5: Medium Sudoku

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

Figure 6: Hard Sudoku

to prevent an infinite loop. I also chose to lower the the temperature of the system by .001% each iteration to allow for more data to be collected.

For the easy puzzle, the backtracking algorithm took 0.061 seconds and backtracked 2,155 times. The medium puzzle took 0.22 seconds and backtracked 8,203 times, while the hard puzzle took 3.83 seconds and backtracked 141,793 times. The results are trivial since this algorithm has been proven time and time again to be the optimal way to solve 9×9 Sudoku boards.

Algorithm	Time (sec)	Difficulty	Backtracks
Backtracking	0.061	Easy	2,155
Backtracking	0.22	Medium	8,203
Backtracking	3.83	Hard	141,793

For the Simulated Annealing algorithm, the easy puzzle took 9.75 seconds and 35,000 iterations. The algorithm was then unable to solve the medium and hard puzzles since it reached the break condition implemented in the code.

Algorithm	Time (sec)	Difficulty	Iterations
SA	9.75	Easy	35,000
SA	104.75	Medium	400,000
SA	111.207	Hard	400,000

For the easy puzzle we can see that the algorithm converged on a solution around 35,000 iteration (Figure 7). From the beginning, there was a great improvement and there is an overall trend to a minimum as the iterations increased.

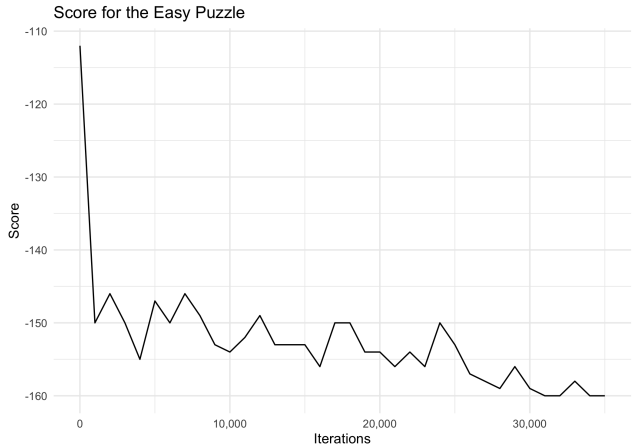

Figure 7: Score of the Easy Sudoku vs. Iterations

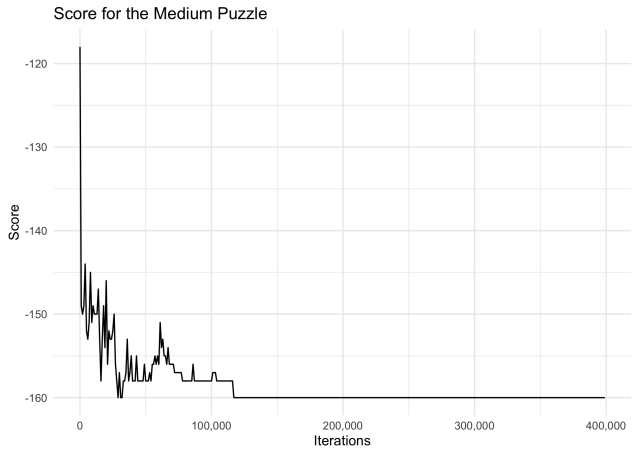

Figure 8: Score of the Medium Sudoku vs. Iterations

For the medium and hard difficulty Sudoku puzzles, the same trend is apparent as in the easy version. At approximately 30,000 to 40,000 iterations for the medium puzzle (Figure 8), it seems that the algorithm was very close to converging to a solution but was unable to. The stochasticity of the puzzle then came into play and brought the board to a state that was worse than what it was previously. This then causes a pitfall at around 55,000 to 60,000 iterations and then it plateaus after 100,000 iterations. The final puzzle after the algorithm timed out was unable to find swap two numbers that were conflicting in two columns. This was consistent after several attempts. The same story also applies for the hard puzzle (Figure 9), at around 10,000 to 15,000 iterations the algorithm was close to

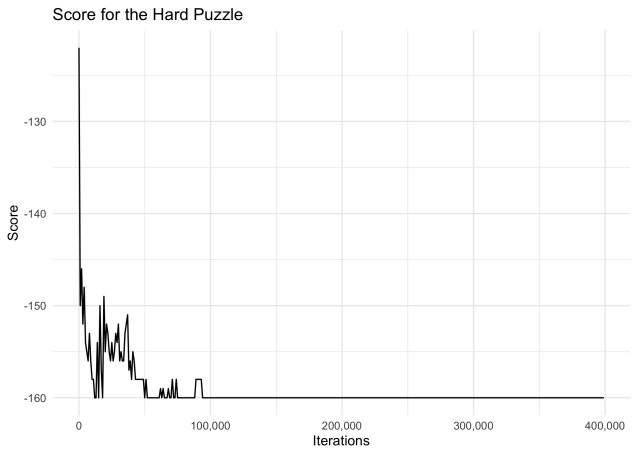

Figure 9: Score of the Hard Sudoku vs. Iterations

converging to a valid solution. It then goes to a worse board state before re-converging at the same score at around 50,000 iterations before timing out.

Conclusion

In this paper, the backtracking and simulated annealing algorithms are both very different ways to arrive at the same singular solution. The backtracking algorithm provides a simple yet effective approach, while the simulated annealing method involves probability, randomness to an environment that is deterministic.

On a 9×9 Sudoku board the differences are apparent as the difficulty increases. The SA algorithm was unable to solve the two more difficult puzzles, it plateaued after 100,000 iterations and was not able to find the last couple of spaces and return a valid solution. The issue may be temperature parameter, probability check or the algorithm did not have enough time to converge to a valid solution. I believe that it is the former, since both algorithms were very close even before the iteration cap was met. As the board moves from a 9×9 grid to an N×N grid, the simulated annealing algorithm may have a better time tackling and solving the board compared to the backtracking algorithm with some parameter tweaking.

Future work on this project would involve testing other algorithms once the simulated annealing algorithm is perfected. Comparing simulated annealing to other stochastic algorithms can also be done with some seeding in order to maintain consistency. Another point to work on for this project includes unifying all of the algorithms into one program, allowing the user to input a Sudoku puzzle of their choice and choose the algorithm(s) of their choice to solve.

References

[Al-Betar et al. 2017] Al-Betar, M. A.; Awadallah, M. A.; Bolaji, A. L.; and Alijla, B. O. 2017. β-hill climbing algorithm for sudoku game. In *2017 Palestinian International Conference on Information and Communication Technology (PICICT)*, 84–88.

[Al-Betar 2016] Al-Betar, M. 2016. β -hill climbing: an exploratory local search. *Neural Computing and Applications* 28.

[Crook 2009] Crook, J. 2009. A pencil-and-paper algorithm for solving sudoku puzzles. *Notices of the American Mathematical Society* 56.

[DELAHAYE 2006] DELAHAYE, J.-P. 2006. The science behind sudoku. *Scientific American* 294(6):80–87.

[Musliu and Winter 2017] Musliu, N., and Winter, F. 2017. A hybrid approach for the sudoku problem: Using constraint programming in iterated local search. *IEEE Intelligent Systems* 32(2):52–62.

[Narayanaswamy, Ma, and Shrivastava 2019] Narayanaswamy, A.; Ma, Y. P.; and Shrivastava, P. 2019. Image detection and digit recognition to solve sudoku as a constraint satisfaction problem. *CoRR* abs/1905.10701.

[Russell and Norvig 2009] Russell, S., and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. USA: Prentice Hall Press, 3rd edition.

Raw Code

Backtracking Algorithm

```
import time
t0 = time.time()
ez_board = [
    [8, 2, 9, 0, 0, 0, 0, 4, 0],
    [0, 0, 0, 7, 0, 0, 3, 8, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 6],
    [0, 0, 8, 1, 0, 0, 6, 3, 9],
    [0, 0, 0, 0, 0, 9, 0, 0, 0],
    [0, 9, 4, 5, 0, 7, 0, 0, 0],
    [5, 0, 0, 0, 0, 0, 0, 0, 4],
    [6, 0, 0, 0, 0, 3, 7, 0, 0],
    [0, 0, 0, 0, 1, 0, 2, 5, 0]
]
t1 = time.time()
medium_board = [
    [0, 2, 0, 5, 0, 7, 0, 4, 0],
    [0, 0, 0, 2, 0, 0, 0, 0, 9],
    [1, 0, 0, 0, 0, 0, 6, 0, 8],
    [0, 1, 0, 3, 0, 0, 0, 0, 0],
    [4, 0, 2, 7, 0, 0, 0, 0, 0],
    [0, 0, 9, 0, 0, 0, 8, 0, 0],
    [0, 4, 0, 0, 0, 0, 7, 3, 0],
    [0, 0, 0, 0, 6, 2, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
]
t2 = time.time()
hard_board = [
    [0, 4, 0, 1, 0, 0, 2, 0, 6],
    [0, 0, 0, 4, 5, 6, 0, 0, 0],
    [7, 0, 0, 0, 8, 0, 0, 0, 0],
    [0, 0, 0, 3, 0, 0, 0, 0, 0],
    [0, 0, 9, 0, 0, 8, 0, 0, 0],
    [0, 0, 3, 6, 0, 0, 9, 1, 0],
    [1, 3, 0, 0, 0, 0, 0, 9, 7],
    [0, 7, 0, 0, 4, 0, 5, 0, 8],
    [9, 0, 0, 0, 0, 0, 0, 0, 0]
]

class Counter(object):
    counts = {}

    @staticmethod
    def count(func):
        def wrapped(*args,**kwargs):
            if func.__name__ in Counter.counts.keys():
                Counter.counts[func.__name__] += 1
```

```
            else:
                Counter.counts[func.__name__] = 1
            return func(*args,**kwargs)
        return wrapped

@Counter.count
def solve(bo):
    find = find_empty(bo)
    if not find:
        return True
    else:
        row, col = find

    for i in range(1, 10):
        if valid(bo, i, (row, col)):
            bo[row][col] = i

            if solve(bo):
                return True
            bo[row][col] = 0
    return False


def valid(bo, num, pos):
    # Check row
    for i in range(len(bo[0])):
        if bo[pos[0]][i] == num and pos[1] != i:
            return False

    # Check column
    for i in range(len(bo)):
        if bo[i][pos[1]] == num and pos[0] != i:
            return False

    # Check box
    box_x = pos[1] // 3
    box_y = pos[0] // 3

    for i in range(box_y * 3, box_y * 3 + 3):
        for j in range(box_x * 3, box_x * 3 + 3):
            if bo[i][j] == num and (i, j) != pos:
                return False

    return True
```

```
def print_board(bo):
    for i in range(len(bo)):
        if i % 3 == 0 and i != 0:
            print("- - - - - - - - - - - - - ")

        for j in range(len(bo[0])):
            if j % 3 == 0 and j != 0:
                print(" | ", end="")

            if j == 8:
                print(bo[i][j])
            else:
                print(str(bo[i][j]) + " ", end="")


def find_empty(bo):
    for i in range(len(bo)):
        for j in range(len(bo[0])):
            if bo[i][j] == 0:
                return i, j  # row, col

    return None


print_board(ez_board)
print("\n")
solve(ez_board)
print("Solved!")
print_board(ez_board)
print(Counter.counts)
t3 = time.time()
t4 = t3-t0
print(t4)


print_board(medium_board)
print("\n")
solve(medium_board)
print("Solved!")
print("\n")
print_board(medium_board)
print(Counter.counts)
t5 = time.time()
t6 = t5-t0
print(t6)
```

```
print_board(hard_board)
print("\n")
solve(hard_board)
print("Solved!")
print("\n")
print_board(hard_board)
print(Counter.counts)
t7 = time.time()
t8 = t7-t0
print(t8)
```

Simulated Annealing Algorithm

```
import sys
from copy import deepcopy
from math import exp
from random import shuffle, random, sample, randint
import numpy as np
import time


def get_column_indices(i, type="data index"):
    """
    Get all indices for the column of ith index
    or for the ith column (depending on type)
    """
    if type == "data index":
        column = i % 9
    elif type == "column index":
        column = i
    indices = [column + 9 * j for j in range(9)]
    return indices


def get_row_indices(i, type="data index"):
    """
    Get all indices for the row of ith index
    or for the ith row (depending on type)
    """
    if type == "data index":
        row = i // 9
    elif type == "row index":
        row = i
    indices = [j + 9 * row for j in range(9)]
    return indices


class SudokuPuzzle(object):

    def __init__(self, data=None, original_entries=None):
        """
                        data - input puzzle as one array, all rows concatenated.
                               (default - incomplete puzzle)

            original_entries - for inheritance of the original entries of one
                               sudoku puzzle's original, immutable entries we don't
                               allow to change between random steps.
```

```
        """
        if data is None:
            self.data = np.array([5, 3, 0, 0, 7, 0, 0, 0, 0,
                                  6, 0, 0, 1, 9, 5, 0, 0, 0,
                                  0, 9, 8, 0, 0, 0, 0, 6, 0,
                                  8, 0, 0, 0, 6, 0, 0, 0, 3,
                                  4, 0, 0, 8, 0, 3, 0, 0, 1,
                                  7, 0, 0, 0, 2, 0, 0, 0, 6,
                                  0, 6, 0, 0, 0, 0, 2, 8, 0,
                                  0, 0, 0, 4, 1, 9, 0, 0, 5,
                                  0, 0, 0, 0, 8, 0, 0, 7, 9])
        else:
            self.data = data

        if original_entries is None:
            self.original_entries = np.arange(81)[self.data > 0]
        else:
            self.original_entries = original_entries

    def randomize_on_zeroes(self):
        """
        Go through entries, replace incomplete entries (zeroes)
        with random numbers.
        """
        for num in range(9):
            block_indices = self.get_block_indices(num)
            block = self.data[block_indices]
            zero_indices = [ind for i, ind in enumerate(block_indices) if block[i] == 0]
            to_fill = [i for i in range(1, 10) if i not in block]
            shuffle(to_fill)
            for ind, value in zip(zero_indices, to_fill):
                self.data[ind] = value

    def get_block_indices(self, k, ignore_originals=False):
        """
        Get data indices for kth block of puzzle.
        """
        row_offset = (k // 3) * 3
        col_offset = (k % 3) * 3
        indices = [col_offset + (j % 3) + 9 * (row_offset + (j // 3)) for j in range(9)]
        if ignore_originals:
            indices = filter(lambda x: x not in self.original_entries, indices)
        return indices

    def view_results(self):
        """
```

```
        Visualize results as a 9 by 9 grid
        (given as a two-dimensional numpy array)
        """

        def notzero(s):
            if s != 0:
                return str(s)
            if s == 0:
                return "X"

        results = np.array([self.data[get_row_indices(j, type="row index")]
            for j in range(9)])
        out_s = ""
        for i, row in enumerate(results):
            if i % 3 == 0:
                out_s += "=" * 25 + '\n'
            out_s += "| " + " | ".join(
                [" ".join(notzero(s) for s in list(row)[3 * (k - 1):3 * k])
                    for k in range(1, 4)]) + " |\n"
        out_s += "=" * 25 + '\n'
        print(out_s)

    def score_board(self):
        """
        Score board by viewing every row and column and giving
        -1 points for each unique entry.
        """
        score = 0
        for row in range(9):
            score -= len(set(self.data[get_row_indices(row, type="row index")]))
        for col in range(9):
            score -= len(set(self.data[get_column_indices(col, type="column index")]))
        return score

    def make_candidate_data(self):
        """
        Generates "neighbor" board by randomly picking
        a square, then swapping two small squares within.
        """
        new_data = deepcopy(self.data)
        block = randint(0, 8)
        num_in_block = len(self.get_block_indices(block, ignore_originals=True))
        random_squares = sample(range(num_in_block), 2)
        square1, square2 = [self.get_block_indices(block, ignore_originals=True)[ind]
            for ind in random_squares]
        new_data[square1], new_data[square2] = new_data[square2], new_data[square1]
```

```
        return new_data


def sudoku_solver(input_data=None):
    """
    Uses a simulated annealing technique to solve a Sudoku puzzle.

    Randomly fills out the sub-squares to be consistent sub-solutions.

    Scores a puzzle by giving a -1 for every unique element
    in each row or each column. Best solution has a score of -162.
    (This is our stopping rule.)

    Candidate for new puzzle is created by randomly selecting
    sub-square, then randomly flipping two of its entries, evaluating
    the new score. The delta_S is the difference between the scores.

    Let T be the global temperature of our system, with a geometric
    schedule for decreasing (perhaps by T <- .999 T).

    If U is drawn uniformly from [0,1], and exp((delta_S/T)) > U,
    then we accept the candidate solution as our new state.
    """

    SP = SudokuPuzzle(input_data)
    print("Original Puzzle:")
    SP.view_results()
    SP.randomize_on_zeroes()
    best_SP = deepcopy(SP)
    current_score = SP.score_board()
    best_score = current_score
    T = .5
    count = 0

    while count < 400000:
        try:
            if count % 1000 == 0:
                print("Iteration %s, \t T = %.5f,
                    \t best_score = %s, \t current_score = %s" %
                    (count, T, best_score, current_score))
            candidate_data = SP.make_candidate_data()
            SP_candidate = SudokuPuzzle(candidate_data, SP.original_entries)
            candidate_score = SP_candidate.score_board()
            delta_S = float(current_score - candidate_score)

            if exp((delta_S / T)) - random() > 0:
```

```
                    SP = SP_candidate
                    current_score = candidate_score

                if current_score < best_score:
                    best_SP = deepcopy(SP)
                    best_score = best_SP.score_board()

                if candidate_score == -162:
                    SP = SP_candidate
                    break

                T = .99999 * T
                count += 1
            except:
                print("Hit an inexplicable numerical error. Try again.")
        if best_score == -162:
            print("\nSOLVED THE PUZZLE.")
        else:
            print("\nDIDN'T SOLVE. (%s/%s points). Try again." % (best_score, -162))
        print("\nFinal Puzzle:")
        SP.view_results()


    if __name__ == "__main__":
        if len(sys.argv) > 1:
            try:
                input_puzzle = np.array([int(s) for s in sys.argv[1]])
            except:
                print("Puzzle must be 81 consecutive integers, 0s for skipped entries.")
            assert len(input_puzzle) == 81, "Puzzle must have 81 entries."
            t0 = time.time()
            sudoku_solver(input_data=input_puzzle)
            t1 = time.time()
            t = t1 - t0
            print(t)

        else:
            sudoku_solver()


    # python sudoku_SA.py
    829000040000700380000001006008100639000009000094507000500000004600003700000010250
    # python sudoku_SA.py
    020507040000200009100000608010300000420700000009000800040000730000062001000000000
    # python sudoku_SA.py
    040100206000456000700080000000300000009008000036009101300000970700405089000000000
```