Sudoku Puzzle Solver

Swetha Vijaya Raju
Joshua Wang

Contents

Sudoku Puzzle Rules

- Given a partially filled 9x9 grid
- Fill all missing squares with numbers 1-9
- Each column must not have a duplicate number
- Each row must not have a duplicate number
- Each 3x3 square must not have a duplicate number

Solutions

- Sometimes more than one solution
- Sometimes no solution

Goal

- Build a Python Algorithm to recursively solve any given Sudoku puzzle using backtracking else output “No Solution”
- Optimize our code

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

Ways to solve...

Naive Vs. Backtracking

Naive - Trying all possibilities, we have a massive complexity of n^{n^2} which is 9^{81} combinations which looks like:

24,617,052,109,557,166,370,343,440,484,714,402,816,
062,905,910,473,854,503,264,650,483,846,634,
633,746,074,437,503,994,802,721,729

V.S

Backtracking

1. Pick empty square
2. Try all numbers
3. Insert numbers 1-9 until one works
4. Repeat with next empty square
5. Backtrack when no solution works

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

Backtracking

1. Pick empty square
2. Try all numbers
3. Insert numbers 1-9 until one works
4. Repeat with next empty square
5. When no solution works, backtrack and try the next solution in the previous step
6. Iterate until complete or “No Solution”

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

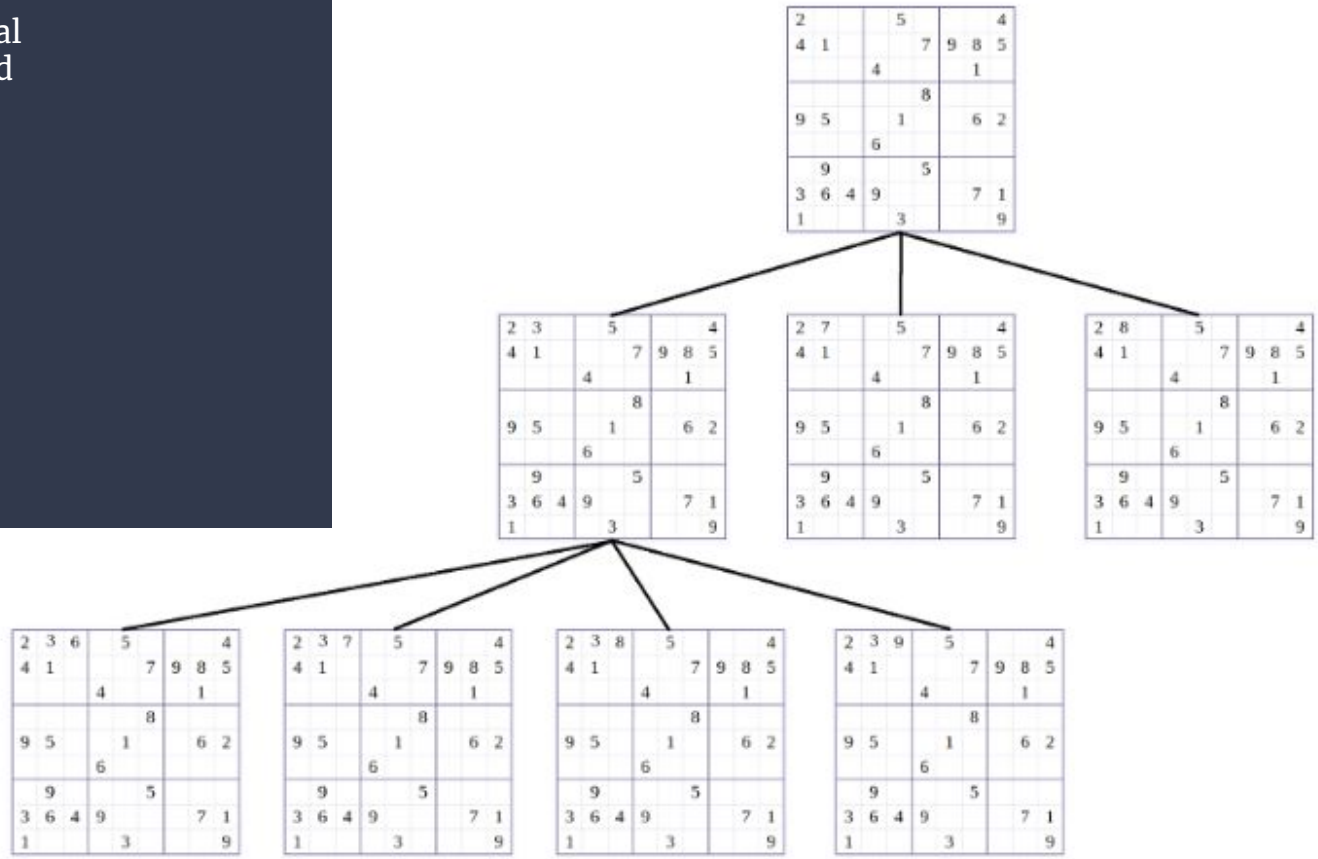
Backtracking

1. Pick empty square
2. Try all numbers
3. Insert numbers 1-9 until one works
4. Repeat with next empty square
5. When no solution works, backtrack and try the next solution in the previous step
6. Iterate until complete or “No Solution”

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

Recursion tree Depth First Search

1. First we map out every potential solution for the first empty grid
2. Next map out the next set of possible answers in the next level
3. Backtrack when no solutions work for that branch
4. With N empty cells we will traverse N rows deep before finding a solution



Ref : https://1729.org.uk/posts/sudoku/

Sudoku Puzzle Solver Algorithm Explained

- Start with a new grid

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

Sudoku Puzzle Solver Algorithm Explained

- Fill all empty slots with Zeros
- Transpose the new grid into a list of list named “puzzle”

```
puzzle = [
    [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],
```

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	4	0
0	9	8	4	3	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

Sudoku Puzzle Solver Algorithm Methods

- **def find_empty_cell(puzzle):**
  - Define a method that finds the next empty cell or “0” in the puzzle.
  - Args:
    - puzzle (list): a 9x9 matrix representing the Sudoku puzzle, where 0 denotes an empty cell.
  - Returns
    - Tuple- a tuple of integers (row, col) representing the indices of the next empty cell, or (-1, 1) if there are no more empty cells

```
puzzle = [
    [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],
```

```
def find_empty_cell(puzzle):
    # iterate through the board
    # to find an empty slot
    for row in range(9):
        for col in range(9):
            if puzzle[row][col] == 0:
                return (row, col)
    return (-1, -1)
```

Code:
https://drive.google.com/file/d/1Il61YlRXQPcW9Am-e9LX8dClP13MuSie/view?usp=sharing

Sudoku Puzzle Solver Algorithm Methods

- **def is_valid(puzzle, row, col, num):**
  - Define a method that checks if assigning num to the cell at (row, col) in the puzzle would violate any of the Sudoku rules
  - Args:
    - puzzle (list): a 9x9 matrix representing the Sudoku puzzle, where 0 denotes an empty cell.
    - row (int): an integer representing the row index of the cell
    - col (int): an integer representing the column index of the cell
    - Num (int): an integer representing the value to be assigned to the cell
  - Returns
    - **bool**: True of the assignment is valid, False otherwise

```
puzzle = [
    [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],
```

```
def find_empty_cell(puzzle):
    # iterate through the board
    # to find an empty slot
    for row in range(9):
        for col in range(9):
            if puzzle[row][col] == 0:
                return (row, col)
    return (-1, -1)
```

```
def is_valid(puzzle, row, col, num):
    # check row for numbers matching num
    for i in range(9):
        if puzzle[row][i] == num:
            return False
    # check column for numbers matching num
    for i in range(9):
        if puzzle[i][col] == num:
            return False
    # check square for numbers matching num
    square_row = (row // 3) * 3
    square_col = (col // 3) * 3
    for i in range(square_row, square_row + 3):
        for j in range(square_col, square_col + 3):
            if puzzle[i][j] == num:
                return False
```

Sudoku Puzzle Solver Algorithm Methods

- **def solve_sudoku(puzzle):**
  - This function uses previous methods and inserts numbers into empty cells
  - Checks if given Sudoku puzzle is solved
  - If not, use backtracking algorithm by trying a new number, recursively calling itself within the method.
  - Args:
    - puzzle (list): a 9x9 matrix representing the Sudoku puzzle, where 0 denotes an empty cell.
  - Method calls:
    - def find_empty_cell(puzzle):
    - def is_valid(puzzle, row, col, num):
  - Returns:
    - bool: True if the puzzle is solvable, False otherwise

```
puzzle = [
    [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],
```

```
def find_empty_cell(puzzle):
    # iterate through the board
    # to find an empty slot
    for row in range(9):
        for col in range(9):
            if puzzle[row][col] == 0:
                return (row, col)
    return (-1, -1)
```

```
def is_valid(puzzle, row, col, num):
    # check row for numbers matching num
    for i in range(9):
        if puzzle[row][i] == num:
            return False
    # check column for numbers matching num
    for i in range(9):
        if puzzle[i][col] == num:
            return False
    # check square for numbers matching num
    square_row = (row // 3) * 3
    square_col = (col // 3) * 3
    for i in range(square_row, square_row + 3):
        for j in range(square_col, square_col + 3):
            if puzzle[i][j] == num:
                return False
```

```
def solve_sudoku(puzzle):
    # find an empty cell with find_empty_cell() method given puzzle
    row, col = find_empty_cell(puzzle)
    # if there are no empty cells, then the puzzle is solved
    if row == -1:
        return True
    # try all possible values for the empty cell using is_valid method
    for num in range(1, 10):
        if is_valid(puzzle, row, col, num):
            # if num is_valid, append the value into puzzle
            puzzle[row][col] = num
            # check if puzzle is solved
            if solve_sudoku(puzzle):
                return True
            # if the recursive call failed,
            # backtrack by setting previous assignment back to 0
            puzzle[row][col] = 0
    # if none of the possible values for the empty cell led to a solution,
    # then the puzzle is unsolvable
    return False
```

Sudoku Puzzle Solver Algorithm Methods

- **def print_answer(puzzle):**
  - This function prints the solved Sudoku puzzle grid if there is an answer.
  - Args:
    - puzzle (list): a 9x9 matrix representing the Sudoku puzzle, where 0 denotes an empty cell.
  - Method calls:
    - def solve_sudoku(puzzle):
  - Returns: None

```
puzzle = [
    [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],
```

```
def find_empty_cell(puzzle):
    # iterate through the board
    # to find an empty slot
    for row in range(9):
        for col in range(9):
            if puzzle[row][col] == 0:
                return (row, col)
    return (-1, -1)
```

```
    # check row for numbers matching num
    for i in range(9):
        if puzzle[row][i] == num:
            return False
    # check column for numbers matching num
    for i in range(9):
        if puzzle[i][col] == num:
            return False
    # check square for numbers matching num
    square_row = (row // 3) * 3
    square_col = (col // 3) * 3
    for i in range(square_row, square_row + 3):
        for j in range(square_col, square_col + 3):
            if puzzle[i][j] == num:
                return False
```

```
def solve_sudoku(puzzle):
    # find an empty cell with find_empty_cell() method given puzzle
    row, col = find_empty_cell(puzzle)
    # if there are no empty cells, then the puzzle is solved
    if row == -1:
        return True
    # try all possible values for the empty cell using is_valid method
    for num in range(1, 10):
        if is_valid(puzzle, row, col, num):
            # if num is_valid, append the value into puzzle
            puzzle[row][col] = num
            # check if puzzle is solved
            if solve_sudoku(puzzle):
                return True
            # if the recursive call failed,
            # backtrack by setting previous assignment back to 0
            puzzle[row][col] = 0
    # if none of the possible values for the empty cell led to a solution,
    # then the puzzle is unsolvable
    return False
```

```
def print_answer(puzzle):
    # if solve_sudoku returns True, puzzle is solved and print Solution
    if solve_sudoku(puzzle):
        print("Solution:")
        for i in range(9):
            for j in range(9):
                print(puzzle[i][j], end=" ")
            print()
    else:
        print("No solution found.")
```

Sudoku Puzzle Solver Algorithm Methods

- **def print_answer(puzzle):**
  - This function prints the solved Sudoku puzzle grid if there is an answer.
  - Args:
    - puzzle (list): a 9x9 matrix representing the Sudoku puzzle, where 0 denotes an empty cell.
  - Method calls:
    - def solve_sudoku(puzzle):
  - Returns: None

```
puzzle = [
  puzzle = [
      [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],
```

```
# Test Case
puzzle = [
    [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],
]

print_answer(puzzle)
```

```
Solution:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

```
def print_answer(puzzle):

    # if solve_sudoku returns True, puzzle is solved and print Solution
    if solve_sudoku(puzzle):
        print("Solution:")
        for i in range(9):
            for j in range(9):
                print(puzzle[i][j], end=" ")
            print()
    else:
        print("No solution found.")
```

Can we do better?

Optimize ‘find empty cell’

- The find_empty_cell() method does a full sweep everytime it is called.
- This is very inefficient

```
def find_empty_cell(puzzle):
    # iterate through the board
    # to find an empty slot
    for row in range(9):
        for col in range(9):
            if puzzle[row][col] == 0:
                return (row, col)
    return (-1, -1)
```

Optimize - find empty cell

- Create a list of empty cells
- Pop an element from that list (
- Push element back to that list when backtracking

```
def __init__(self, puzzle):
    self.puzzle = puzzle
    self.empty_cells = [(row, col) for row in range(9)
                        for col in range(9)
                        if self.puzzle[row][col] == 0]
```

```
def solve(self):
    if not self.empty_cells:
        return True

    row, col = self.empty_cells.pop()

    for num in range(1, 10):
        if self.is_valid(row, col, num):
            self.puzzle[row][col] = num

            if self.solve():
                return True

            self.puzzle[row][col] = 0

    self.empty_cells.append((row, col))
    return False
```

Can we do better?

Optimize ‘validity check’

- The is_valid() method does a full sweep everytime it is called.
- This is very inefficient

```
def is_valid(self, row, col, num):
    for i in range(9):
        if self.puzzle[row][i] == num:
            return False

    for i in range(9):
        if self.puzzle[i][col] == num:
            return False

    square_row = (row // 3) * 3
    square_col = (col // 3) * 3

    for i in range(square_row, square_row + 3):
        for j in range(square_col, square_col + 3):
            if self.puzzle[i][j] == num:
                return False
    return True
```

Optimize - validity check

- get_valid_numbers(row, col) creates 3 sets that holds all the numbers present in the particular row, column and subgrid.
- Union these 3 to get a set of used numbers.
- Get a set of unused numbers from the union set.

```
def get_valid_numbers(self, row, col):
    row_set = set(self.puzzle[row])
    col_set = set(self.puzzle[i][col] for i in range(9))

    square_row = (row // 3) * 3
    square_col = (col // 3) * 3
    subgrid_set = set(self.puzzle[i][j]
                      for i in range(square_row, square_row + 3)
                      for j in range(square_col, square_col + 3))

    used_numbers = row_set | col_set | subgrid_set
    unused_numbers = set(range(1, 10)) - used_numbers

    return unused_numbers
```

Optimize - validity check

- get_valid_numbers(row, col) creates 3 sets that holds all the numbers present in the particular row, column and subgrid.
- Union these 3 to get a set of used numbers.
- Get a set of unused numbers from the union set.

```
def solve(self):

    if not self.empty_cells:
        return True

    row, col = self.empty_cells.pop()

    for num in self.get_valid_numbers(row, col):
        self.puzzle[row][col] = num

        if self.solve():
            return True

        self.puzzle[row][col] = 0

    self.empty_cells.append((row, col))
    return False
```

Can we do better?

Optimize empty_cells with priority queue

- The empty_cells is a list that works like a stack and pops an element that was added last.
- This is very inefficient.
- The ordering should be based on a better parameter that reduces the number of recursive calls.

```
def __init__(self, puzzle):
    self.puzzle = puzzle
    self.empty_cells = [(row, col) for row in range(9)
                        for col in range(9)
                        if self.puzzle[row][col] == 0]
```

```
def solve(self):
    if not self.empty_cells:
        return True

    row, col = self.empty_cells.pop()

    for num in range(1, 10):
        if self.is_valid(row, col, num):
            self.puzzle[row][col] = num

            if self.solve():
                return True

            self.puzzle[row][col] = 0

    self.empty_cells.append((row, col))
    return False
```

Optimize - Priority Queue

- Create a heap-based priority queue to keep track of empty cells.
- Implement a min heap data structure here.
- This data structure will pop the item with the least number of domains at any particular time.
- Push element back to the min heap when backtracking.

```python
def __init__(self, puzzle):
    self.puzzle = puzzle
    self.empty_cells = []
    for row in range(9):
        for col in range(9):
            if self.puzzle[row][col] == 0:
                valid_numbers = self.get_valid_numbers(row, col)
                heapq.heappush(self.empty_cells, (len(valid_numbers), row, col))
```

```python
def solve(self):
    if not self.empty_cells:
        return True

    num_choices, row, col = heapq.heappop(self.empty_cells)

    for num in self.get_valid_numbers(row, col):
        self.puzzle[row][col] = num

        if self.solve():
            return True

        self.puzzle[row][col] = 0

    heapq.heappush(self.empty_cells, (len(self.get_valid_numbers(row, col)), row, col))
    return False
```

Optimize - Priority Queue

```
Choice: 1, Row: 4, Column: 4
Choice: 1, Row: 6, Column: 5
Choice: 1, Row: 6, Column: 8
Choice: 1, Row: 7, Column: 7
Choice: 2, Row: 0, Column: 3
Choice: 2, Row: 2, Column: 0
Choice: 2, Row: 2, Column: 3
Choice: 2, Row: 2, Column: 4
Choice: 2, Row: 2, Column: 5
```

	0	1	2	3	4	5	6	7	8
0	5	3	0	0	7	0	0	0	0
1	6	0	0	1	9	5	0	0	0
2	0	9	8	0	0	0	0	6	0
3	8	0	0	0	6	0	0	0	3
4	4	0	0	8	0	3	0	0	1
5	7	0	0	0	2	0	0	0	6
6	0	6	0	0	0	0	2	8	0
7	0	0	0	4	1	9	0	0	5
8	0	0	0	0	8	0	0	7	9

Can we do better?

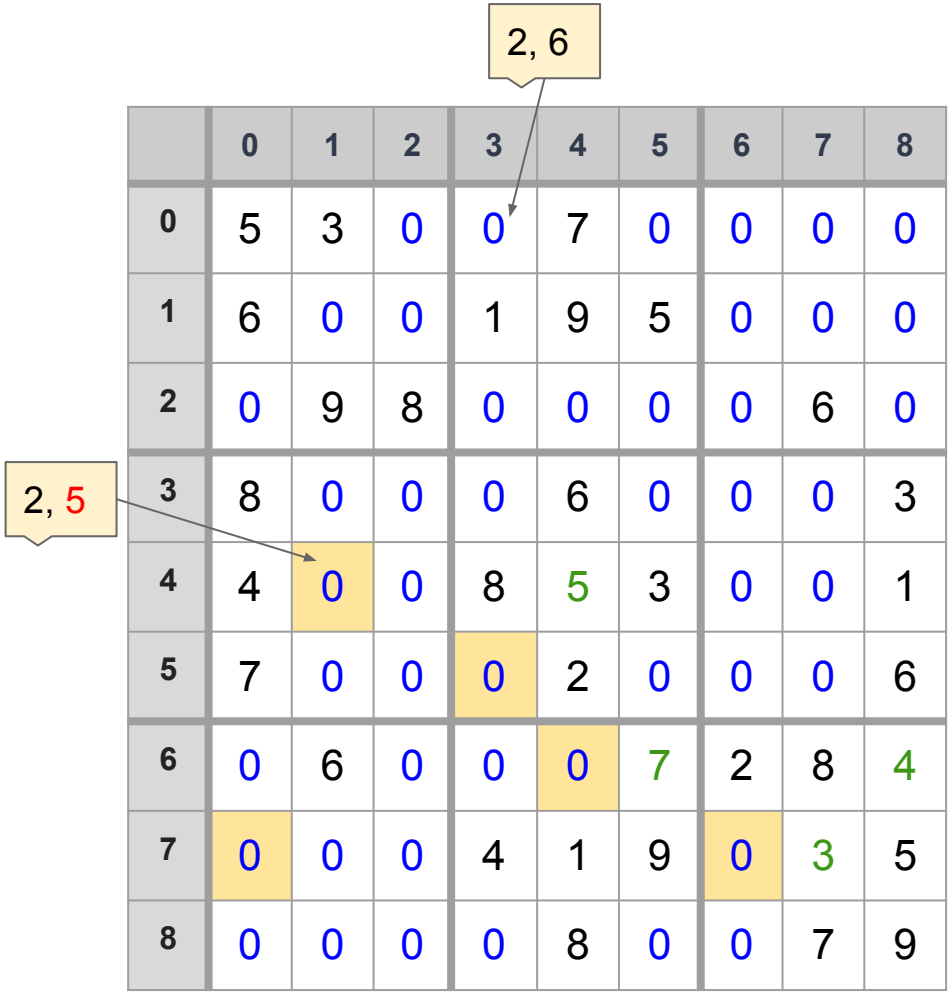
Optimize solve()

- get_valid_numbers(row, col) won't yield an updated set of unused numbers after a change was made in the previous stack.
- This is inconsistent and causes more redundant operations.
- We need to reduce the search space.

```
def solve(self):
    if not self.empty_cells:
        return True

    num_choices, row, col = heapq.heappop(self.empty_cells)

    for num in self.get_valid_numbers(row, col):
        self.puzzle[row][col] = num

        if self.solve():
            return True

        self.puzzle[row][col] = 0

    heapq.heappush(self.empty_cells,
                   (len(self.get_valid_numbers(row, col)), row, col))

    return False
```

Can we do better?

```
Choice: 1, Row: 4, Column: 4
Choice: 1, Row: 6, Column: 5
Choice: 1, Row: 6, Column: 8
Choice: 1, Row: 7, Column: 7
Choice: 2, Row: 0, Column: 3
Choice: 2, Row: 2, Column: 0
Choice: 2, Row: 2, Column: 3
Choice: 2, Row: 2, Column: 4
Choice: 2, Row: 2, Column: 5
```



	0	1	2	3	4	5	6	7	8
0	5	3	0	0	7	0	0	0	0
1	6	0	0	1	9	5	0	0	0
2	0	9	8	0	0	0	0	6	0
3	8	0	0	0	6	0	0	0	3
4	4	0	0	8	5	3	0	0	1
5	7	0	0	0	2	0	0	0	6
6	0	6	0	0	0	7	2	8	4
7	0	0	0	4	1	9	0	3	5
8	0	0	0	0	8	0	0	7	9

Forward Checking

- Constraint propagation technique
- Used to reduce the size of the search space
- Eliminate variables from consideration at the earliest by assigning a value to a variable and then propagating that value through the constraints to eliminate inconsistent values from the domains of the remaining variables.

Forward Checking - Sudoku

- Update the domains of the variables (empty cells) based on,
  - values assigned to the current cell
  - constraints imposed by the Sudoku rules
- For example, when a value is assigned to a cell,
  - domains of the variables in the same row, column, and subgrid are updated to eliminate the assigned value from their domains.
- Reduces the number of possible values for each variable and can speed up the search for a solution

Arc Consistency - AC3 Algorithm

- AC-3 algorithm checks and enforces arc consistency in a Constraint Satisfaction Problems.
- Maintains a queue of arcs (variable-value pairs) that need to be checked for consistency.
- Checks each arc in the queue by removing any inconsistent values from the domain of the variables involved in that arc.
- The AC-3 algorithm continues to check and enforce arc consistency until
    - all the arcs in the queue have been processed
    - it finds an inconsistency

Optimize – Forward Checking & Arc Consistency

- self.domains maps each empty cell in the Sudoku grid to a set of integers representing the valid numbers that can be placed in that cell.
- self.constraints maps each empty cell to a list of tuples representing the coordinates of the cells that share a row, column, or subgrid with that cell
- self.discarded keeps track of the numbers that have been removed from the domains of each empty cell during the forward checking process.

```
def __init__(self, puzzle):
    self.puzzle = puzzle
    self.empty_cells = []
    self.domains = dict()
    self.constraints = dict()
    self.discarded = dict()
    for row in range(9):
        for col in range(9):
            if self.puzzle[row][col] == 0:
                valid_numbers = self.get_valid_numbers(row, col)
                heapq.heappush(self.empty_cells, (len(valid_numbers), row, col))
                self.domains[(row, col)] = valid_numbers
                self.constraints[(row, col)] = self.get_constraints(row, col)
```

Optimize – Forward Checking & Arc Consistency

get_constraints(row, col)

- Finds the list of all the constraints for a current cell.
  - Row
  - Column
  - Sub grid

```
def __init__(self, puzzle):
    self.puzzle = puzzle
    self.empty_cells = []
    self.domains = dict()
    self.constraints = dict()
    self.discarded = dict()
    for row in range(9):
        for col in range(9):
            if self.puzzle[row][col] == 0:
                valid_numbers = self.get_valid_numbers(row, col)
                heapq.heappush(self.empty_cells, (len(valid_numbers), row, col))
                self.domains[(row, col)] = valid_numbers
                self.constraints[(row, col)] = self.get_constraints(row, col)
```

```
def get_constraints(self, row, col):
    row_constraints = [(row, c) for c in range(9) if c != col]
    col_constraints = [(r, col) for r in range(9) if r != row]
    subgrid_row = (row // 3) * 3
    subgrid_col = (col // 3) * 3
    subgrid_constraints = [(r, c) for r in range(subgrid_row, subgrid_row + 3)
                           for c in range(subgrid_col, subgrid_col + 3)
                           if (r, c) != (row, col)]
    return row_constraints + col_constraints + subgrid_constraints
```

Optimize – Forward Checking & Arc Consistency

- Iterates over the domains of the current cell.
- Once the value is assigned to the cell, performs forward checking to discard the value from the domains of the current cell's constraints.
- Maintain arc consistency from the latest domains of the constraints by updating the number of choices in empty_cells heap.

```
def solve(self):
    if not self.empty_cells:
        return True

    num_choices, row, col = heapq.heappop(self.empty_cells)

    for num in self.domains[(row, col)]:
        self.puzzle[row][col] = num
        self.forward_checking(row, col, num)

        self.maintain_arc_consistency()
        if self.solve():
            return True

        self.puzzle[row][col] = 0
        self.restore_domains(row, col, num)
        self.maintain_arc_consistency()

    heapq.heappush(self.empty_cells, (len(self.domains[(row, col)]), row, col))
    return False
```

Optimize – Forward Checking & Arc Consistency

- When backtracking, restore the domains modified earlier.
- Once again, maintain arc consistency from the latest domains of the constraints by updating the number of choices in empty_cells heap.

```
def solve(self):
    if not self.empty_cells:
        return True

    num_choices, row, col = heapq.heappop(self.empty_cells)

    for num in self.domains[(row, col)]:
        self.puzzle[row][col] = num
        self.forward_checking(row, col, num)

        self.maintain_arc_consistency()
        if self.solve():
            return True

        self.puzzle[row][col] = 0
        self.restore_domains(row, col, num)
        self.maintain_arc_consistency()

    heapq.heappush(self.empty_cells, (len(self.domains[(row, col)]), row, col))
    return False
```

Optimize – Forward Checking & Arc Consistency

forward_checking(row, col, num)

- Iterates over the constraints of the current cell.
- Discard the value from the empty constraints that has the same value in it's domain.
- Update the self.discarded dictionary

```
def forward_checking(self, row, col, num):
    for r, c in self.constraints[(row, col)]:
        if self.puzzle[r][c] == 0 and num in self.domains[(r, c)]:
            self.domains[(r, c)].discard(num)
            if (r, c) not in self.discarded:
                self.discarded[(r, c)] = {num}
            else:
                self.discarded[(r, c)].add(num)
```

```
def maintain_arc_consistency(self):
    temp = []
    for _, r, c in self.empty_cells:
        valid_numbers = self.domains[(r, c)]
        heapq.heappush(temp, (len(valid_numbers), r, c))
    self.empty_cells = temp
```

Optimize – Forward Checking & Arc Consistency

maintain_arc_consistency()

- Iterates over the empty cells to find the latest domains.
- Update the number of choices for the constraint in the empty_cells heap.

```
def forward_checking(self, row, col, num):
    for r, c in self.constraints[(row, col)]:
        if self.puzzle[r][c] == 0 and num in self.domains[(r, c)]:
            self.domains[(r, c)].discard(num)
            if (r, c) not in self.discarded:
                self.discarded[(r, c)] = {num}
            else:
                self.discarded[(r, c)].add(num)
```

```
def maintain_arc_consistency(self):
    temp = []
    for _, r, c in self.empty_cells:
        valid_numbers = self.domains[(r, c)]
        heapq.heappush(temp, (len(valid_numbers), r, c))
    self.empty_cells = temp
```

Optimize – Forward Checking & Arc Consistency

restore_domains()

- Iterates over the constraints of the current cell.
- Get valid numbers for each empty constraint of the current cell with the current state of the puzzle.
- If the current value is in the valid_numbers set, add it to the domain.
- Discard the value from self.discarded

```
def restore_domains(self, row, col, num):
    for r, c in self.constraints[(row, col)]:
        valid_numbers = self.get_valid_numbers(r, c)
        if self.puzzle[r][c] == 0 and num in valid_numbers:
            self.domains[(r, c)].add(num)
            self.discarded[(r, c)].discard(num)
```

```
def maintain_arc_consistency(self):
    temp = []
    for _, r, c in self.empty_cells:
        valid_numbers = self.domains[(r, c)]
        heapq.heappush(temp, (len(valid_numbers), r, c))
    self.empty_cells = temp
```

Optimize – Forward Checking & Arc Consistency

Result

- Consistency maintained
- Result order is seen to be generated with the most updated information.
- This prunes the search tree and makes it more efficient

```
Choice: 1, Row: 4, Column: 4
Choice: 1, Row: 6, Column: 5
Choice: 1, Row: 6, Column: 8
Choice: 1, Row: 7, Column: 7
Choice: 2, Row: 0, Column: 3
Choice: 2, Row: 2, Column: 0
Choice: 2, Row: 2, Column: 3
Choice: 2, Row: 2, Column: 4
Choice: 2, Row: 2, Column: 5
```

```
Choices: 1, Row: 4, Column: 4
Choices: 1, Row: 4, Column: 1
Choices: 1, Row: 4, Column: 7
Choices: 1, Row: 4, Column: 2
Choices: 1, Row: 4, Column: 6
Choices: 1, Row: 5, Column: 3
Choices: 1, Row: 3, Column: 3
Choices: 1, Row: 6, Column: 4
Choices: 1, Row: 2, Column: 4
Choices: 1, Row: 2, Column: 5
Choices: 1, Row: 0, Column: 3
Choices: 1, Row: 0, Column: 5
Choices: 1, Row: 2, Column: 0
Choices: 1, Row: 2, Column: 3
Choices: 1, Row: 2, Column: 6
Choices: 1, Row: 2, Column: 8
Choices: 1, Row: 3, Column: 6
Choices: 1, Row: 3, Column: 5
Choices: 1, Row: 3, Column: 1
Choices: 1, Row: 3, Column: 2
Choices: 1, Row: 3, Column: 7
```

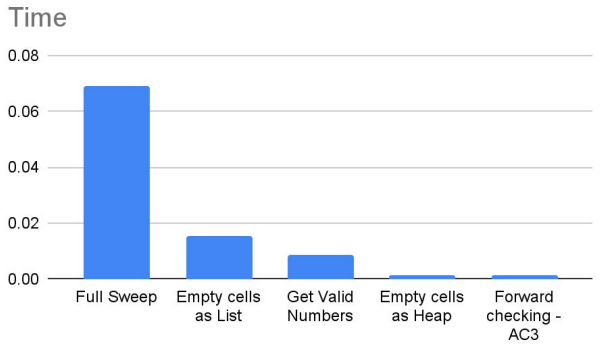
Performance

Optimization	Time Complexity	Explaination	Why
Naive	O(M^(N^2))	M=9, N = 9	Inefficient
Full Sweep	O((N^2)logN)	O(logN(N^2 + N^2)); N = 9	Backtracking to redundant combinations
Empty cells as List	O((N^2)logN)	O(logN(N^2)); N = 9	Does full sweep for empty cells only once and stores it as a list
Get Valid Numbers	O(MNlogN)	0<M<=9, N = 9	Iterates over a set of valid numbers istead of checking if a number is valid. Reduces the number of computations
Empty cells as Heap	O(MNlogN)	0<M<=9 where M is mostly in the lower end; N = 9	Does full sweep for empty cells only once and stores it as a min heap. This reduces the number of recursive calls.
Forward checking - AC3	O(MVlogN)	0<M<=9 where M is mostly in the lower end; N = 10; 0<=V<=81 where V is the number of empty cells	Maintains consistency by updating the current state for domains of constraints. Significantly reduces the number of recursive calls. For easy and medium puzzles, it complely gets rid off backtracking.

Performance - Easy Puzzle

```
puzzle = [
    [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],
]
```

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

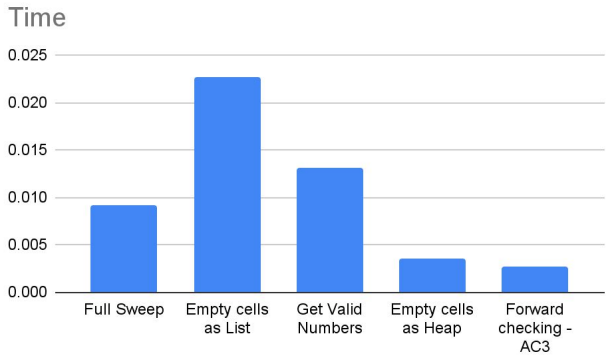


Optimization	Count	Depth	Time
Full Sweep	4209	51	0.06932902336
Empty cells as List	1006	51	0.01519703865
Get Valid Numbers	999	51	0.008784770966
Empty cells as Heap	128	51	0.001593112946
Forward checking - AC3	52	51	0.001189231873

Performance - No Solution

```
puzzle = [
    [5, 3, 0, 0, 7, 0, 0, 0, 3],
    [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],
]
```

```
5 3 0 0 7 0 0 0 3
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
```

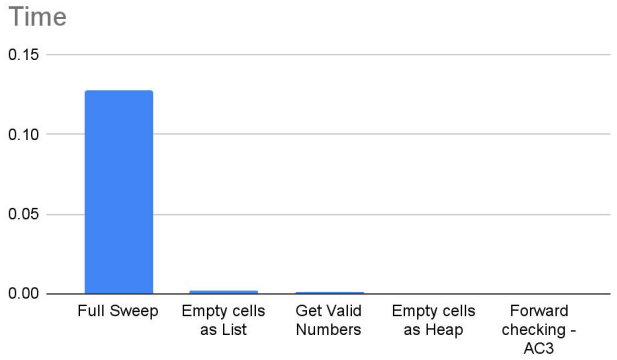


Optimization	Count	Depth	Time
Full Sweep	587	9	0.009208917618
Empty cells as List	1489	42	0.02269220352
Get Valid Numbers	1489	42	0.01306867599
Empty cells as Heap	240	37	0.003496170044
Forward checking - AC3	19	18	0.002764940262

Performance - No Solution

```
puzzle = [
    [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, 3],
]
```

```
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 3
```

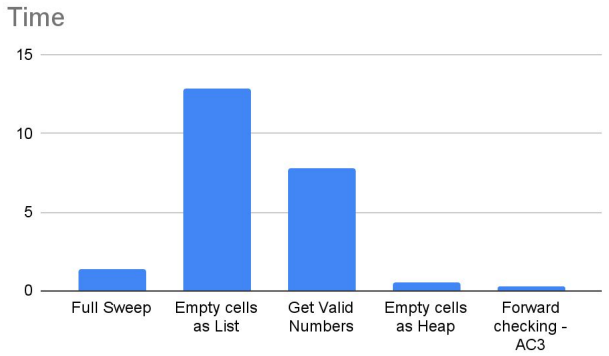


Optimization	Count	Depth	Time
Full Sweep	7402	44	0.1273310184
Empty cells as List	139	6	0.0020570755
Get Valid Numbers	139	6	0.00118803978
Empty cells as Heap	1	0	0.00007104873657
Forward checking - AC3	1	0	0.0004081726074

Performance - Hard Puzzle

```
puzzle = [
    [0, 2, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 6, 0, 0, 0, 0, 3],
    [0, 7, 4, 0, 8, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 3, 0, 0, 2],
    [0, 8, 0, 0, 4, 0, 0, 1, 0],
    [6, 0, 0, 5, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 7, 8, 0],
    [5, 0, 0, 0, 0, 9, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 4, 0]
]
```

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



Optimization	Count	Depth	Time
Full Sweep	78431	62	1.386306047
Empty cells as List	888456	62	12.83464813
Get Valid Numbers	903341	62	7.830549955
Empty cells as Heap	41439	62	0.5627481937
Forward checking - AC3	1759	62	0.2632148266

Future Work

- Optimize get_valid_numbers() method by calculating unused numbers for each row, column and subgrid instead of each grid.
- Explore other data structures to maintain a priority queue.
- Further, optimize the consistency methods to avoid any redundant operations.
- Add a new functionality to compute all possible solutions for a given puzzle.
- Scale for any grid size.
- Parallelize the solve() method to take advantage of multiple cores.

Demo