A* FINAL ALGORITHM
SUDOKU SOLVER

Submitted by,
Gopika Mahadevan (A05244809)

Under the guidance of
Dr. Moonis Ali

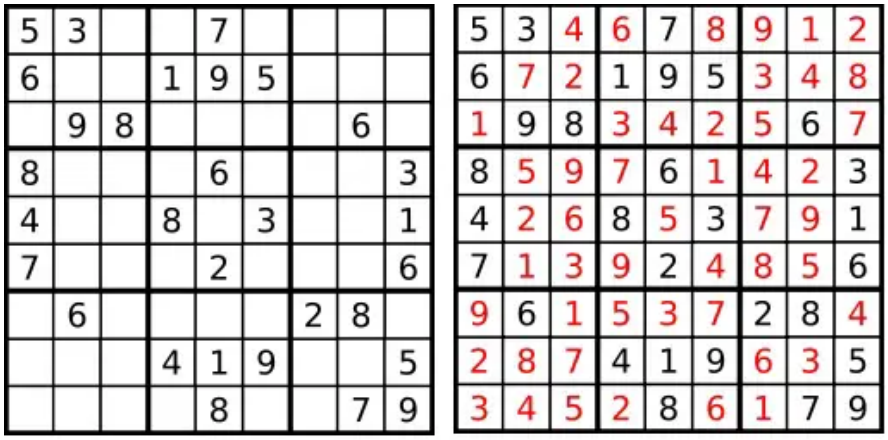
Contents

1.PROBLEM DESCRIPTION

1.1 DESCRIPTION OF THE PROBLEM

A Sudoku puzzle is a popular logic-based number-placement game. The puzzle consists of a 9x9 grid, divided into nine 3x3 sub grids called regions or blocks. The objective of the game is to fill the grid with digits from 1 to 9, ensuring that each row, each column, and each of the nine 3x3 sub grids contains all the digits from 1 to 9 without repetition. The sudoku board I have used for the project has 9 rows, 9 columns and 9 subgrids.

Given below is an example which shows a starting state of sudoku board. The player fills the values following the rule that 1-9 can occur only once in each row,column and subgrid. Finally the player wins the game when he fills up the sudoku board.The final state of the sudoku board is also shown in the image below.



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

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

Figure 1

The primary aim in this project is design and implement a program to solve a sudoku puzzle by following the game rules. A * final algorithm has been used to solve this problem efficiently and optimally. The program takes in the initial configuration of the sudoku board as input processes it using A* final algorithm and then prints the final path leading to the solution and some parameters for measuring performance. These steps have been explained in detail in this report.

1.2 OBJECTIVE OF THE PROJECT

Objective of this project is to take an initial sudoku configuration as input, fill each square in the sudoku ensuring that the game rule is satisfied, using A* final algorithm. The rule is that each row column and sub grid should have numbers from 1-9 without overlap.

The execution time (ET), the number of nodes generated (NG), the number of nodes expanded (NE), the depth of the tree (D), and effective branching factor b* (NG/D) and Total path (TP) for each run should be observed. Additionally, a comparative analysis of results with programs that utilize a different heuristic is required. This analysis aims to provide insights into the efficacy of data structures, code logic, and heuristic functions used in the implementation.

2.DOMAIN

2.1 OVERVIEW OF CONCEPTS USED

Applying A* final algorithm for solving sudoku: In traditional Sudoku puzzles, the puzzle has a single unique solution, and that solution remains consistent regardless of the sequence in which the

cells are filled, as long as the Sudoku rules are followed. This property allows algorithms like A* to effectively solve Sudoku puzzles, ensuring a valid solution is always attainable if one exists.

However, it's important to recognize that while the solution remains the same, the specific path taken to achieve it may vary depending on factors such as the algorithm utilized, applied heuristics, and the order of decision-making regarding which cells to fill next.

Project specifications: Python and NumPy library has been utilized in this project. The sudoku board as a 2D NumPy array(In this 2D array zeroes represent unfilled values in the board)is taken as an input. The program fills up the first found zero and proceeds by scanning the sudoku 2D array starting from 1st row 1st column. It continues to scan row by row.

Candidate Reduction Heuristic: A heuristic is a problem-solving approach or technique that uses practical, intuitive, and rule-of-thumb strategies to find approximate solutions when exact solutions are impractical or unknown. Heuristics are often used in situations where the search space is large or the problem is complex, allowing for quick decision-making based on limited information.

Heuristic used in this implementation is called a candidate reduction heuristic. The candidate reduction heuristic refers to a strategy used to narrow down the possible values for each cell based on the values already present in the board. For example, when trying to determine the possible values for an empty cell, instead of considering all numbers from 1 to 9 as potential candidates, the candidate reduction heuristic looks at the numbers already present in the same row, column, and 3x3 sub grid of the cell. It then eliminates these numbers from the list of candidates for the empty cell, leaving only the remaining numbers as possible candidates.

By reducing the number of candidate values for each empty cell based on the constraints imposed by the existing values in the board, the candidate reduction heuristic helps to narrow down the search space and simplify the process of solving the Sudoku puzzle.

Factor that affects A*final algorithm most is the heuristic chosen. Hence a comparative study with a program that utilizes a different heuristic has also been done.

3.METHODOLOGIES

3.1 A* FINAL ALGORITHM

1. Input: The program prompts the user to enter a 9x9 Sudoku grid as a 2D array represented as a string. It then converts the input into a NumPy array of integers, representing the initial Sudoku board.
2. Initialization: Lists open list and closed list is defined. The algorithm initializes the first node with the initial Sudoku board configuration and adds it to the open list to initiate the A* final algorithm.
   Node is a basic unit in this algorithm where each node has a particular configuration, gn, fn, parent and children. These parameters are specific to a node. This has been explained in successor generation part below.
   Furthermore, the performance parameters such as the number of nodes generated and expanded are also initialized.
3. A* Final Algorithm Loop: The main loop of the algorithm runs until the heuristic value of the best node (lowest estimated cost) becomes 0, indicating that the goal state has been reached.
4. Expansion: In each iteration of the loop, if heuristic is not zero(goal state is not reached), the algorithm expands the best node. Best node is the node with the smallest cost function value, which is the first node from open_list.

It then generates successor nodes by applying valid moves (filling empty cells) to the current Sudoku board configuration. The generate_successor_configuration function in the Sudoku solver algorithm generates successor configurations by identifying the first empty cell (zero value) in the given Sudoku board. It then determines the possible values that can be placed in this cell without violating the Sudoku rules (no repetition in rows, columns, or sub grids). By examining the values already present in the same row, column, and sub grid as the empty cell, the function constructs a set of combined values to represent the numbers already in the row, column, or sub grid of the empty cell. Subtracting these combined values from the set of possible values yields the remaining values that can be placed in the empty cell. For each remaining value, the function creates a new Sudoku board configuration by copying the original board and replacing the empty cell with the remaining value. These new configurations are added to a list of successors, which needs to be added to the open_list to continue the algorithm.

5. Successor Generation: Based on the generated configurations, the algorithm generates successor configuration nodes. The nodes represent the sudoku board's current state. It is a basic unit in this algorithm and each node has the following parameters:
   - Configuration: It's the sudoku board's configuration (values it contains in each cell at a particular point in the algorithm).
   - gn: Distance from the initial configuration.
   - fn: This is the cost function. At each level of this algorithm the node with smallest fn is chosen for expansion.
   - Parent: Each node points to a parent node.
   - Children: List pointing to a node's successors.

   Hence each successor node is assigned a cost function (fn) based on its estimated cost from the initial state and heuristic value. fn = hn+gn where hn is the value of the heuristic (a candidate reduction heuristic) of current state, gn is distance from root node.

6. Updating Open and Closed Lists: This part of the algorithm checks whether each successor node generated above is already present in the open or closed list and makes certain adjustments as explained below:

   - If in Open List: If a successor node is in the open list, its cost function is updated if its new path is more efficient. That is, if OLD node (node in open list, which has already been visited) is alone retained and successor is discarded. Now if this OLD node has a gn value greater than the successor we discarded, then we give OLD node successor's gn (as both nodes have same configuration, configuration of OLD need not be changed) and point the OLD node's parent to best node and update the cost fn too.

   - If successor is in closed list: If a successor node is in the closed list, its cost function is updated, and the cost function of its children is recursively updated if necessary. That is, in this step OLD node is node in closed list. Like open list case explained above , if this OLD node has a gn value greater than the successor we discarded, then we give OLD node successor's gn (as both nodes have same configuration, configuration remains the same) and point the OLD node's parent to best node and update the cost fn too. Now we need to recursively iterate through children of this node. This implementation does that by using Depth First Search* technique.

7. Sorting and Printing: The algorithm sorts the open list based on the cost function fn. The next loop we will be evaluating first element in open list, which is the node with the lowest fn value.
8. Termination: The algorithm terminates when the heuristic value of the best node becomes 0, indicating that the goal state (a complete and valid Sudoku solution) has been reached.

Overall, this implementation uses the A* final algorithm to efficiently searches for a solution to the Sudoku puzzle by generating successor nodes, evaluating their cost functions, and iteratively expanding the most promising nodes until the goal state is reached.

4. SOURCE CODE IMPLEMENTATION

This section explains the implementation details of the Sudoku solving program. The program is structured around various functions and data structures to efficiently solve Sudoku puzzles.

4.1 Representing nodes using Sudoku State class

The Sudoku State class is designed to represent the current state/node of the Sudoku puzzle. Each node has a configuration ,which consists of a 9x9 grid 2D NumPy array, where each cell holds a number from 1 to 9. The Sudoku State class provides a constructor for initializing a node in the puzzle with a given configuration. It is also used for storing, and updating node values, and retrieving information about the puzzle state. In short, it is the basic unit in A* final algorithm which helps to keep track of our current state (for example, how many cells have been filled? , which parent does this node have?) The code snippet with the SudokuState class, which helps us define nodes is given below:

```
class SudokuState:
    def __init__(self, configuration, gn, fn, parent):
        #instance of the sudoku board during the game
        self.configuration=configuration
        # g(n) value of the sudoku board instance
        self.gn=gn
        # f(n) value of the sudoku board instance
        self.fn=fn
        # parent of the sudoku board instance
        self.parent=parent
        #children of the node
        self.children=[]
```

This class encapsulates the puzzle's stater, allowing for easy manipulation and inspection of the puzzle throughout the solving process. By abstracting the puzzle into a class, the code becomes more modular and easier to understand.

4.2 Generate successor configuration function

The generate_successor_configuration() function is a crucial component of the puzzle-solving algorithm. Given a current Sudoku state, this function explores possible moves and generates successor configurations by placing numbers in empty cells. It does so while ensuring that the generated configurations adhere to the rules of Sudoku, such as no duplicate numbers in rows, columns, or 3x3 sub grids.

Generating the successors for the node shown below, considering the first zero position at 7,5 highlighted in yellow:

[[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 0 0 0 5]

[3 4 5 2 8 6 1 7 9]

possible_values = set([1, 2, 3, 4, 5, 6, 7, 8, 9])

combined_values=set([1,2,3,4,5,6,7,8]) .Taking the values from row 7 column 5 and sub grid containing 7,5 position element. The values taken are from the red highlighted values

remaining_values = possible_values - combined_values=([9])

The successor is generated by replacing the zero with the possible values generated. In this case there is only 1 possibility, which is 9, so only one successor:

[[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 0 0 5]

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

This function typically employs techniques like constraint propagation to efficiently explore the search space and generate valid successor configurations. It plays a fundamental role in the search algorithm by expanding the search tree and guiding the exploration towards potential solutions.

4.3 Candidate Reduction heuristic function

The heuristic function heuristic() evaluates the desirability of different Sudoku puzzle configurations based on certain criteria. These criteria may include the number of unfilled cells, the degree of constraint satisfaction, or other domain-specific metrics. The heuristic function provides a measure of how close a configuration is to a solution, guiding the search algorithm towards promising areas of the search space. By incorporating domain knowledge into the search process, the heuristic function helps prioritize certain configurations over others, potentially leading to faster convergence towards a solution.

The heuristic implemented in this implementation isa Candidate reduction heuristic. Where the sum of number of possible candidates for unfilled cells is taken. We decide the number of possible candidates by first finding out the possible values for an unfilled position, by following the same logic as successor generation explained above. Similarly for all unfilled positions we calculate the number of possible values. Sum of the possible values at each unfilled cell is the heuristic value that is returned by this function.

4.4 Main() function specific data structures and loops used

The main function serves as the entry point to the Sudoku solving program. It orchestrates the solving process by initializing the puzzle, managing the search algorithm, and printing performance metrics. This section outlines specific data structures and loops used within the main function to facilitate the solving process:

1. NumPy arrays and functions: The NumPy library is utilized for efficient handling and manipulation of multi-dimensional arrays representing Sudoku grids. They boast memory efficiency by storing data in contiguous blocks, which is advantageous for handling large datasets. This especially useful since sudoku solving requires storing many configurations. NumPy's extensive library of functions and methods simplifies array manipulation and mathematical operations. I have used the following NumPy functions:

   - np.array(array_2d, dtype=int):
   - This function creates a NumPy array from the input array_2d, which could be a Python list, tuple, or another array-like object.
   - The dtype=int argument specifies the data type of the elements in the resulting NumPy array. In this case, it ensures that the elements are converted to integers. We have received an string input from the user, for the integer conversion this is used.

   - np.array_equal(node.configuration, successor.configuration):
   - This function compares two NumPy arrays, node.configuration and successor.configuration, for equality.
   - It returns True if the arrays have the same shape and elements, and False otherwise.
   - This function is particularly useful for checking whether two arrays, representing different states in a problem-solving algorithm like Sudoku, are identical. This is used for checking if a successor exists in open list or closed list.

   - np.where(sudoku_board == 0):
   - This function returns the indices of elements in the NumPy array sudoku_board where the condition sudoku_board == 0 is True.
   - In the context of a Sudoku puzzle, where 0 represents empty cells, this function can be used to find the indices of all empty cells in the puzzle.
   - The returned indices are then used for various purposes, such as identifying possible moves in the Sudoku solving algorithm.

2. Two lists for open and closed states: These lists are maintained to track the states that have been visited (closed) and those that are yet to be explored (open). This helps prevent revisiting previously explored states and ensures that the search algorithm progresses efficiently.
3. While loop: The while loop is employed to iterate through the search process until a solution is found or all possibilities are exhausted. It controls the flow of the solving algorithm, repeatedly generating successor configurations and evaluating heuristic values until a satisfactory solution is reached.
4. Stack for Depth-First Search (DFS): A stack-based approach is adopted for Depth-First Search (DFS), a commonly used search strategy in Sudoku solving algorithms. This is when a node in closed lost is being explored again and gn value changes needs to be reflected to children of the node in consideration. The stack facilitates the exploration of the search space

in a depth-first manner, prioritizing deeper levels of the search tree before backtracking to explore other branches.

4.5 Performance parameters

During the execution of the Sudoku solving program, various performance parameters are computed and printed to provide insights into the efficiency and effectiveness of the solving algorithm. These parameters include:

- The execution time (ET)
- The number of nodes generated (NG)
- The number of nodes expanded (NE)
- The depth of the tree (D)
- Effective branching factor b* (NG/D)
- The total path leading to the solution
- Memory used

5.SOURCE CODE

Copy of the source code is given below:

```
import ast
import numpy as np
import time
from memory_profiler import memory_usage

# this class represents each node (which is a sudoku configuration) during the
program run
class SudokuState:
    def __init__(self, configuration, gn, fn, parent):
        #instance of the sudoku board during the game
        self.configuration=configuration
        # g(n) value of the sudoku board instance
        self.gn=gn
        # f(n) value of the sudoku board instance
        self.fn=fn
        # parent of the sudoku board instance
        self.parent=parent
        #children of the node
        self.children=[]

class Sudoku:
    #find first unfilled position and the number of possible values that can
occupy that cell to generate successors
    def generate_successor_configuration(self,sudoku_board):
        # Define the possible values
        possible_values = set([1, 2, 3, 4, 5, 6, 7, 8, 9])
        successors=[]

        #where is a numpy function which returns position where values are
equal to given value(0 in this case)
```

```
        first_zero_position = np.where(sudoku_board == 0)
        zero_row = first_zero_position[0][0]
        zero_column = first_zero_position[1][0]

        row_values = set(sudoku_board[zero_row, :])
        col_values = set(sudoku_board[:, zero_column])
        subgrid_values = set(sudoku_board[(zero_row//3)*3:(zero_row//3)*3+3,
(zero_column//3)*3:(zero_column//3)*3+3].flatten())

        # Combine the values in the same row, column, and subgrid
        combined_values = row_values.union(col_values, subgrid_values)

        # Subtract the combined values from the possible values- works like
set theory
        remaining_values = possible_values - combined_values

        for value in remaining_values:
            new_sudoku=sudoku_board.copy()
            new_sudoku[zero_row][zero_column]=value
            # DEBUG STATEMENT
print(new_sudoku[zero_row][zero_column],"replaced value at",zero_row,"
",zero_column)
            # DEBUG STATEMENT print(new_sudoku)
            successors.append(np.array(new_sudoku))



        return successors
    #count of zeroes in board returned
    def heuristic1(self,sudoku_board):
        first_zero_position = np.where(sudoku_board == 0)
        return len(first_zero_position[0])


    #the heuristic used is candidate reduction
    def heuristic(self,sudoku_board):
        sum_remaining = 0

        # Define the possible values
        possible_values = set([1, 2, 3, 4, 5, 6, 7, 8, 9])

        # Iterate over each cell in the matrix
        for i in range(9):
            for j in range(9):
                # Skip if the cell already has a value
                if sudoku_board[i][j] == 0:

                    # Find values in the same row
                    row_values = set(sudoku_board[i, :])
```

```
                    # Find values in the same column
                    col_values = set(sudoku_board[:, j])

                    # Find values in the same subgrid
                    subgrid_values = set(sudoku_board[(i//3)*3:(i//3)*3+3,
(j//3)*3:(j//3)*3+3].flatten())

                    # Combine the values in the same row, column, and subgrid
                    combined_values = row_values.union(col_values,
subgrid_values)

                    # Subtract the combined values from the possible values
                    remaining_values = possible_values - combined_values

                    # Add the size of the remaining values to the sum,
                    # That is, these many values can still be tried. All
possibilities for each unfilled cell is filled up
                    sum_remaining += len(remaining_values)


        return sum_remaining

    def main(self):
        # DEBUG STATEMENT initilaizing a sudoku board with 9*9 size all
zeroes,datatype : integer
        # DEBUG STATEMENT initial_sudoku_board = np.zeros((9,9),dtype=int)

        # Prompt the user to enter the 2D array as a string
        array_str = input("Enter the 2D array representing sudoku board: \n")
        array_2d = eval(array_str)

        # Converting to a numpy array
        initial_sudoku_board = np.array(array_2d,dtype=int)
        print("Entered sudoku board is:\n")
        print(initial_sudoku_board)



        # DEBUG STATEMENT
print("heuritstic",self.heuristic(initial_sudoku_board))

        #initializing the first node and adding to open list to initiate
A*final algorithm
        gn=0
        parent_node=None
        child=None
        node=SudokuState(initial_sudoku_board,gn,self.heuristic(initial_sudoku
_board)+gn,parent_node)
```

```
        open_list=[]
        closed_list=[]
        open_list.append(node)

        best_node=node
        best_path=[]
        successors=[]

        # performance parameters
        no_of_nodes_generated=0
        no_of_nodes_expanded=0
        start_time = time.time()

        # A* final algorithm
        while(self.heuristic(best_node.configuration)!=0):
            if open_list:
                # sort open list and get the first node from open list with
the best f(n) value to be expanded
                open_list.sort(key=lambda x:x.fn)
                # DEBUG STATEMENT print("Open List:", [node.fn for node in
open_list])
                # DEBUG STATEMENT print("Open List:", [node.fn for node in
open_list])
                best_node=open_list[0]
                # DEBUG STATEMENT print("This node
expanded",best_node.configuration)
                # DEBUG STATEMENT print("This node
heuristic\n",self.heuristic(best_node.configuration))
                best_path.append(best_node)
                closed_list.append(open_list.pop(0))

                if(self.heuristic(best_node.configuration)==0):

                    break
                else:
                    #each time we generate successors gn for successors will
be incremented by one and successor list is cleared
                    # DEBUG STATEMENT print(" successors!")
                    no_of_nodes_expanded+=1
                    successors.clear()
                    gn=gn+1

                    # generating successors of the node if its not the final
configuration node
                    successor_configuration=self.generate_successor_configurat
ion((best_node.configuration))
                    for successor_config in successor_configuration:
```

```
                        successor=SudokuState(successor_config,gn,self.heurist
ic(successor_config)+gn,best_node)
                        successors.append(successor)
                        no_of_nodes_generated+=1
                    best_node.children.extend(successors)

                    for successor in successors:
                        #check in open if this successor is already present
but not processed
                        in_open_list=False
                        for node in open_list:
                            if(np.array_equal(node.configuration,successor.con
figuration)):
                                in_open_list=True
                                OLD=node
                                successors.remove(successor)
                                successors.append(OLD)
                                if(successor.gn<OLD.gn):
                                    OLD.parent=best_node
                                    OLD.gn=successor.gn
                                    OLD.fn=OLD.gn+
self.heuritic(OLD.configuration)
                        #checking if successor is present in closed list
                        if(not(in_open_list)):
                            for node in closed_list:
                                if(np.array_equal(node.configuration,successor
.configuration)):
                                    OLD=node
                                    # DEBUG STATEMENT
successors.remove(successor)
                                    successors.append(OLD)
                                    if(successor.gn<OLD.gn):
                                        OLD.parent=best_node
                                        OLD.gn=successor.gn
                                        OLD.fn=OLD.gn+
self.heuritic(OLD.configuration)
                                    stack = [OLD]
                                    # propogating g(n) value to children
                                    while stack:
                                        current_node = stack.pop()
                                        for child in current_node.children:
                                            if child.gn > current_node.gn + 1:
                                                child.gn = current_node.gn + 1
                                                child.fn = child.gn +
self.heuristic(child.configuration)
                                                stack.append(child)

                    open_list.extend(successors)
```

```
                    open_list.sort(key=lambda x:x.fn)

            else:
                print("Failure, not a valid input")
                break

        end_time = time.time()

        # displaying results
        print("\n Path to last node:")
        for config in best_path:
            print(config.configuration)

        print("\n Last node:")
        print(best_node.configuration)

        execution_time = (end_time - start_time)


        total_memory_usage = memory_usage(-1, interval=1)

        if(gn!=0):
            effective_branching_factor=no_of_nodes_generated/gn
        else:
            effective_branching_factor=0

        # Create the table
        parameters = [
        ["Execution time (ET)", f"{execution_time} second"],
        ["No of nodes generated (NG)", f"{no_of_nodes_generated}"],
        ["No of nodes expanded (NE)", f"{no_of_nodes_expanded}"],
        ["Depth of the tree (D)", f"{gn}"],
        ["Effective branching factor (b*)", f"{effective_branching_factor}"],
        ["Memory usage in MB:",f"{total_memory_usage[0]}"]
        ]

        # Print the table
        for row in parameters:
            print("{:<40} {:<30}".format(*row))

if __name__=="__main__":
    sudoku=Sudoku()
    sudoku.main()
```

6.COPY OF PROGRAM RUNS

7.ANALYSIS OF THE PROGRAM

1. Use of NumPy:

- Time Complexity: NumPy arrays offer efficient data manipulation and indexing capabilities, which can lead to faster operations compared to traditional Python lists. Hence the Sudoku board inputted has been treated as a NumPy array. Solving sudoku puzzle using A* final algorithm involves checking and comparing sudoku configurations and searching for zeroes and filling up positions in a 2D array. All these operations have been handled efficiently by using the powerful NumPy library.
- Loop Reduction Optimization: For example, to access the 1st position where zero occurs where() function is used. The complexity would have been O(n^2) if we had used normal loops to access an element in a 2D array while using NumPy reduces that to O(n).

2. SudokuState Class:
- Time Complexity: Instantiating a SudokuState object involves minimal computation, typically O(1) time complexity. This is because the time taken to create an object in Python is constant and does not depend on the size of the input or any other external factors. Therefore, regardless of the size of the Sudoku board or any other parameters, the time complexity of creating a SudokuState object remains constant, making it O(1).
- Loop Reduction Optimization: Defining a class by encapsulating node properties within a class reduces the need for repetitive data structures and improves code readability.
- OOP Principles Used: Encapsulation is a fundamental OOP principle employed here. The class encapsulates properties and behaviours related to each Sudoku board configuration, promoting code organization and modularity.

3. generate_successor_configuration Method:
- Time Complexity: The usage of NumPy functions such as np.where() and array slicing significantly reduces the time complexity of finding the first unfilled position and extracting row, column, and sub grid values. These operations have time complexities of O(n) or less, where n is the size of the Sudoku board.
- Loop Reduction Optimization: By leveraging NumPy functions, the method eliminates the need for explicit loop iterations over the Sudoku board to find the first unfilled position and extract values from rows, columns, and sub grids, leading to reduced computational overhead.
- OOP Principles Used: While NumPy functions primarily optimize computational efficiency, the method's encapsulation within the class promotes code modularity and adheres to the principle of encapsulation.

4. heuristic Method:
- Time Complexity: Again, the usage of NumPy functions for extracting values from rows, columns, and sub grids contributes to reducing the time complexity of the method to O(n^2), where n is the size of the Sudoku board.
- Loop Reduction Optimization: By leveraging NumPy functions, the method efficiently calculates the heuristic value by avoiding explicit loop iterations over the Sudoku board, leading to improved computational efficiency.
- OOP Principles Used: The method encapsulates the logic for calculating a heuristic measure within the class, adhering to the principle of encapsulation, and promoting code modularity.

5. A* final algorithm in main()
   The most important part of this implementation is the algorithm , which:
   The A* algorithm is considered a good algorithm for pathfinding because it effectively balances completeness, optimality, and efficiency. Here's how A* achieves these qualities:

- Completeness: A* ensures that if a solution exists, it will eventually find it. This property is essential for pathfinding algorithms as it guarantees that the algorithm won't get stuck in an

infinite loop or fail to find a solution when one exists. A* achieves completeness by systematically exploring the search space until it finds a goal state.

- Optimality: A* guarantees finding the shortest path from the start to the goal if certain conditions are met. Specifically, A* is optimal when it uses an admissible heuristic and a consistent heuristic. An admissible heuristic never overestimates the cost to reach the goal, while a consistent heuristic (also known as monotonicity) satisfies the triangle inequality property. By utilizing these heuristics effectively, A* ensures that it explores the most promising paths first, leading to the discovery of the optimal solution.

- Efficiency: A* is generally efficient in practice, especially when the search space is relatively small like the sudoku puzzle or when an effective heuristic is used. The efficiency of A* comes from its ability to prioritize paths that are likely to lead to the goal by considering both the cost incurred so far (the 'g' value) and the estimated cost to the goal (the 'h' value). By intelligently balancing these factors, A* avoids exploring unnecessary paths and focuses its efforts on the most promising ones.

  In the context of A* final (A* algorithm applied to solving Sudoku puzzles), ensuring that the path to the goal is saved and that children nodes are not discarded is crucial for its effectiveness:

- Saving the Path to the Goal: A* final saves the path to the goal by keeping track of each node's parent during the search process. This allows it to reconstruct the optimal solution path once the goal state is reached. Saving the path ensures that A* final not only finds a solution but also provides a way to navigate from the start state to the goal state efficiently.

- Not Discarding Children Nodes: A* final explores successor nodes generated from each parent node systematically. It ensures that all possible paths are considered and not prematurely discarded. By maintaining a proper exploration strategy and retaining all potential paths, A* final increases the likelihood of finding the optimal solution while avoiding overlooking promising paths.

  To summarize, A* final's effectiveness as a pathfinding algorithm lies in its ability to systematically explore the search space, prioritize promising paths, ensure completeness, optimality, and efficiency, and maintain the path to the goal while considering all potential successor nodes.
  This algorithm has been implemented using a while loop which runs till the goal node is reached. The complexity will be O(n) where n is number of nodes in the path leading from source node to final node. Hence the implementation of this algorithm is efficient optimal and complete.

In summary, the usage of NumPy functions significantly contributes to reducing time complexity, optimizing loop iterations. And promoting OOP principles such as encapsulation and modularity within the Sudoku solver program makes the code neat understandable and efficient. Furthermore, the implementation of A* final algorithm makes the code efficient.

8.TABULATION OF RESULTS

9.ANALYSIS OF RESULTS

For a program that implements number of zeroes heuristics function (Counting number of unfilled spaces in the sudoku board) and coded in C++ results are as shown below for the 8 test cases discussed in the previous section:

TEST CASE	NODE USED	Execution Time(ET) in second	No of nodes generated (NG)	No of nodes expanded (NE)	Depth of the tree (D)	Effective branching factor (b*)
1	Node1	0.078	0	0	0	0
2	Node1	0.266	7	7	6	1.0
3	Node2	0.521	13	13	13	1.0
4	Node2	0.458	11	11	11	1.0
5	Node2	0.565	11	11	11	1.0
6	Node2	66.75	4631	4631	51	90
7	Node2	23.87	141	141	32	4.40625
8	Node2	16.009	451	399	49	22.5306

For Candidate reduction heuristic used in my implementation:

TEST CASE	NODE USED	Execution Time(ET) in second	No of nodes generated (NG)	No of nodes expanded (NE)	Depth of the tree (D)	Effective branching factor (b*)	Memory usage in MB:
1	Node1	0.0	0	0	0	0	34.90234375
2	Node1	0.0	7	7	7	1.0	34.8515625
3	Node2	0.0	13	13	13	1.0	34.91015625
4	Node2	0.00546050071	11	11	11	1.0	35.109375
5	Node2	0.0	11	11	11	1.0	34.82421875
6	Node2	29.146344900	4631	4631	4631	1.0	38.52734375
7	Node2	0.07406830787	108	106	106	1.018	34.90625
8	Node2	0.53877210617	451	399	399	1.130	35.26953125

Comparing the heuristics , we can see that candidate reduction heuristics performs better than a zero-counting heuristic. The reasons could be the following

Efficiency of Candidate Reduction Heuristic:

- Reduction in Search Space: The candidate reduction heuristic effectively reduces the search space by considering the remaining possible values for each unfilled cell. This reduction in the search space helps in focusing the search on more promising paths, leading to a quicker convergence towards the solution.
- Better Pruning Mechanism: The candidate reduction heuristic enables better pruning of the search tree by discarding unpromising branches early in the search process. It doesn't consider possibilities unless it leads to a valid move. This helps in avoiding unnecessary exploration of large portions of the search space, leading to faster execution times. We can see as the number of nodes generated increases for a much complex scenario, especially in case 7 and 8.In these cases, Candidate reduction performs better than number of zeroes heuristics significantly.

Observations from results:

- Execution Time: The execution time for the candidate reduction heuristic is consistently lower across all test cases compared to the number of zeroes heuristic. This indicates that the candidate reduction heuristic leads to a more efficient search process and quicker convergence towards the solution.
- Number of Nodes Generated and Expanded: In most test cases, the number of nodes generated and expanded is similar for both heuristics, indicating that they explore a similar portion of the search space. However, the candidate reduction heuristic may lead to more effective pruning of unpromising branches, resulting in a more focused exploration of the search space.
- Minor programmatic differences and loops could also have led to more time being taken by zero counting heuristics. But this overhead is significantly less compared to the contribution of the heuristic function, since the algorithm used by both the programs is A* final. Hence we can conclude that candidate reduction heuristic performs better.

10.CONCLUSION

The Sudoku solver project benefits significantly from the implementation of the candidate reduction heuristic. It not only enhances the efficiency of the solver but also demonstrates the importance of choosing effective heuristics in optimizing search algorithms. The candidate reduction heuristic demonstrates superior efficiency, leading to quicker execution times and more effective pruning of the search space. Based on the analysis of the Sudoku solver project, it's evident that the implementation leveraging the candidate reduction heuristic outperforms the one using a zero-counting heuristic.

Furthermore, use of NumPy library has significantly reduced the time taken, by reducing number of loops required. Following object-oriented principles like representation of nodes as a Sudoku State object, modularizing functions in A* final algorithm etc. has made the code even more efficient , compact, and understandable. More importantly a complete, optimal algorithm like A* final, brings in significant changes in exploring search space and producing an apt solution.

To summarize, by leveraging efficient algorithms and heuristics, the solver can efficiently tackle Sudoku puzzles of varying complexities, providing a robust and reliable solution.

11.REFERENCES

[1] sudoku.com, “Sudoku Rules for Complete Beginners,” [Online]. Available: https://sudoku.com/how-to-play/sudoku-rules-for-complete-beginners/.

[2] David Carmel, “Solving Sudoku by Heuristic Search,” [Online]. Available: https://medium.com/@davidcarmel/solving-sudoku-by-heuristic-search-b0c2b2c5346e.

[3] numpy.org, “Arrays ndarray,” [Online]. Available: https://numpy.org/doc/stable/reference/arrays.ndarray.html.

[4] smucs, “A* Search Algorithm,” [Online]. Available: https://medium.com/smucs/a-search-algorithm-e22721d38b87.

[5] OpenAI ChatGPT, “OpenAI ChatGPT Playground,” [Online]. Available: https://chat.openai.com/ (Generated by ChatGPT).