**EXPERIMENT – 01**

**IMPLEMENTATION OF SEARCH ALGORITHM**

**SUDOKU SOLVER USING DIFFERENT SEARCH ALGORITHMS**

Depth-First Search (DFS) is a fundamental algorithm used in computer science for traversing or searching tree or graph data structures. It explores as far down a branch as possible before backtracking. It explores each branch of the tree or graph to its deepest level before backtracking to explore other branches. Once it reaches a node with no unvisited adjacent nodes, it backtracks to the previous node to continue the search.

Breadth-First Search (BFS) is an algorithm used for traversing or searching tree or graph data structures. It starts at the root node (or an arbitrary node in the case of a graph) and explores all of the neighbour nodes at the present depth level before moving on to nodes at the next depth level.

The A\* (A-star) algorithm is a widely used pathfinding and graph traversal algorithm that is used to find the shortest path between two points, typically in a grid or graph. An estimate of the cost to reach the goal from the current node is made as heuristic h(n). It guides the algorithm to explore paths that seem promising. The actual cost from the start node to the current node is calculated as g(n). Then based on the output of f(n) = g(n) + h(n) function the next node to be explored is decided upon.

**1) DEPTH FIRST SEARCH ALGORITHM:**

def is\_valid(grid, row, col, num):

    # Check row

    for x in range(9):

        if grid[row][x] == num:

            return False

    # Check column

    for x in range(9):

        if grid[x][col] == num:

            return False

    # Check 3x3 subgrid

    start\_row = row - row % 3

    start\_col = col - col % 3

    for i in range(3):

        for j in range(3):

            if grid[i + start\_row][j + start\_col] == num:

                return False

    return True

def find\_empty\_location(grid):

    for row in range(9):

        for col in range(9):

            if grid[row][col] == 0:

                return row, col

    return None

print("0 represents empty locations")

grid = [

    [3, 0, 6, 5, 0, 8, 4, 0, 0],

    [5, 2, 0, 0, 0, 0, 0, 0, 0],

    [0, 8, 7, 0, 0, 0, 0, 3, 1],

    [0, 0, 3, 0, 1, 0, 0, 8, 0],

    [9, 0, 0, 8, 6, 3, 0, 0, 5],

    [0, 5, 0, 0, 9, 0, 6, 0, 0],

    [1, 3, 0, 0, 0, 0, 2, 5, 0],

    [0, 0, 0, 0, 0, 0, 0, 7, 4],

    [0, 0, 5, 2, 0, 6, 3, 0, 0]

]

# Print the original Sudoku grid

print("Original Sudoku Grid:")

for i in range(0,9):

    for j in range(0,9):

        print(grid[i][j], end=" "),

    print()

def solve\_sudoku\_dfs(grid):

    find = find\_empty\_location(grid)

    if not find:

        return True

    row, col = find

    for num in range(1, 10):

        if is\_valid(grid, row, col, num):

            grid[row][col] = num

            if solve\_sudoku\_dfs(grid):

                return True

            grid[row][col] = 0  # Backtrack

    return False

if solve\_sudoku\_dfs(grid):

    print("Sudoku solved using DFS:")

    for i in range(0,9):

        for j in range(0,9):

            print(grid[i][j], end=" "),

        print()

else:

    print("No solution exists using DFS.")

**2) BREADTH FIRST SEARCH ALGORITHM:**

from collections import deque

def solve\_sudoku\_bfs(grid):

    queue = deque([grid])

    while queue:

        current\_grid = queue.popleft()

        find = find\_empty\_location(current\_grid)

        if not find:

            return current\_grid

        row, col = find

        for num in range(1, 10):

            if is\_valid(current\_grid, row, col, num):

                new\_grid = [row[:] for row in current\_grid]

                new\_grid[row][col] = num

                queue.append(new\_grid)

    return None

solved\_grid\_bfs = solve\_sudoku\_bfs(grid)

if solved\_grid\_bfs:

    print("Sudoku solved using BFS:")

    for i in range(0,9):

        for j in range(0,9):

           print(solved\_grid\_bfs[i][j], end=" "),

        print()

else:

    print("No solution exists using BFS.")

**3) A\* SEARCH ALGORITHM**

from collections import deque

def is\_valid(grid, row, col, num):

    # Check row

    for x in range(9):

        if grid[row][x] == num:

            return False

    # Check column

    for x in range(9):

        if grid[x][col] == num:

            return False

    # Check 3x3 subgrid

    start\_row = row - row % 3

    start\_col = col - col % 3

    for i in range(3):

        for j in range(3):

            if grid[i + start\_row][j + start\_col] == num:

                return False

    return True

def find\_empty\_location(grid):

    for row in range(9):

        for col in range(9):

            if grid[row][col] == 0:

                return row, col

    return None

#Heuristic Values

def count\_remaining\_values(grid):

    count = 0

    for row in range(9):

        for col in range(9):

            if grid[row][col] == 0:

                for num in range(1, 10):

                    if is\_valid(grid, row, col, num):

                        count += 1

    return count

def solve\_sudoku\_astar(grid):

    queue = deque([(count\_remaining\_values(grid), grid)])

    while queue:

        \_, current\_grid = queue.popleft()

        find = find\_empty\_location(current\_grid)

        if not find:

            return current\_grid

        row, col = find

        for num in range(1, 10):

            if is\_valid(current\_grid, row, col, num):

                new\_grid = [row[:] for row in current\_grid]

                new\_grid[row][col] = num

                remaining\_values = count\_remaining\_values(new\_grid)

                queue.append((remaining\_values, new\_grid))

                queue = deque(sorted(queue, key=lambda x: x[0]))  # Sort by remaining values

    return None

solved\_grid\_astar = solve\_sudoku\_astar(grid)

if solved\_grid\_astar:

    print("Sudoku solved using A\*:")

    for i in range(0,9):

        for j in range(0,9):

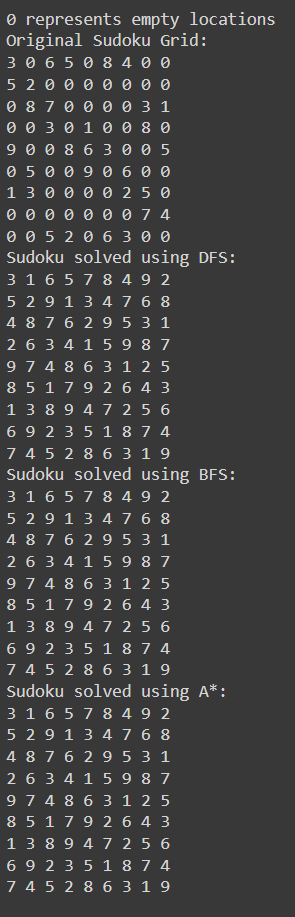
           print(solved\_grid\_astar[i][j], end=" "),

        print()

else:

    print("No solution exists using A\*.")

**RESULTS:**

****

**CONCLUSION:**

Using this experiment, we learned some basic blind search algorithms and implemented them for a simple real-life algorithm. We can conclude from this that A\* is comparatively little difficult to be implemented. BFS explores all possibilities level by level but can be slow and memory-intensive due to the vast number of possibilities in many cases. DFS is more memory-efficient than BFS but can get stuck exploring deep, incorrect paths, leading to potentially longer solution times. *A*\* is the most efficient of the three for solving some real-life problems, as it uses a heuristic to prioritize promising paths, significantly reducing the number of possibilities to explore and often finding the solution faster.

**REFERENCES:**

1. [**https://www.geeksforgeeks.org/search-algorithms-in-ai/**](https://www.geeksforgeeks.org/search-algorithms-in-ai/)
2. [**www.geeksforgeeks.org/sudoku-backtracking-7/**](https://www.geeksforgeeks.org/sudoku-backtracking-7/)
3. [**www.reddit.com/r/learnpython/comments/13mkb4j/sudoku\_solver/**](https://www.reddit.com/r/learnpython/comments/13mkb4j/sudoku_solver/)