EXPERIMENT – 01
Implementation of Search Algorithm

1. Aim:
To implement and analyse the performance of various search algorithms (e.g., Depth-First Search, Breadth-First Search, A* Algorithm).

2. Objectives:
- Understand the working principles of different search algorithms.
- Compare the efficiency and complexity of each algorithm.
- Apply the search algorithm to solve a given problem.

3. Brief Theory:
Search algorithms are fundamental techniques used in artificial intelligence and computer science for navigating through data structures (such as trees or graphs) to find specific information or solve problems.

- Depth-First Search (DFS) explores as far as possible along a branch before backtracking.
- Breadth-First Search (BFS) explores all the nodes at the present depth level before moving on to nodes at the next depth level.
- *A Algorithm** uses heuristics to improve the efficiency of the search process by estimating the cost to reach the goal.

4. Hints:
- Consider a state space where the start state is number 1 and each state k has two successors: numbers 2k and 2k + 1.

  a) Draw the portion of the state space for states 1 to 20.
  b) Suppose the goal state is 15.
- Start with implementing basic search algorithm (e.g., DFS or BFS) before moving on to more complex ones like A*.
- Test your algorithms on both small and large datasets to understand their behaviour and performance.

5. <u>Implementation of application (Sudoku Solver):</u>

Python Code:

```
#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.")

#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.")

# 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*.")
```

6. **Result:**

```
0 represents empty locations
Original Sudoku 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
Sudoku solved using DFS:
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
Sudoku solved using BFS:
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
Sudoku solved using A*:
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

7. Conclusion:

In conclusion, this experiment successfully implemented and analysed Depth-First Search (DFS), Breadth-First Search (BFS), and A* algorithms. These algorithms were applied to solve a Sudoku puzzle, showcasing their problem-solving capabilities. DFS used a backtracking approach, while BFS explored all levels of the state space. A* introduced heuristics, making the search more efficient by estimating the cost to reach the goal. The experiment demonstrated the unique advantages of each algorithm and their applicability in different scenarios. All methods successfully solved the Sudoku puzzle, highlighting their effectiveness in search problems.

8. **Link to the code uploaded on: https://github.com/Niti0209/Blind-search-algorithms.git**

9. **List of Reference used for implementation.**

- **https://www.geeksforgeeks.org/search-algorithms-in-ai/**
- **www.geeksforgeeks.org/sudoku-backtracking-7/**
- **www.reddit.com/r/learnpython/comments/13mkb4j/sudoku_solver/**