Experiment:-1

Objective:- Implement Basic Search Strategies 8-Puzzle Problem.

Theory:- The 8-puzzle problem is a classic problem in artificial intelligence and search theory. It involves a 3x3 grid with 8 numbered tiles and one blank space. The objective is to move the tiles by sliding them into the blank space to reach a goal configuration from a given initial configuration.

Algorithm:-

Using BFS:- We can perform a Breadth-first search on the state space tree. This always finds a goal state nearest to the root. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

- Breadth-first search on the state-space tree.
- Always finds the nearest goal state.
- Same sequence of moves irrespective of initial state.

Step by step approach:

- Start from the root node.
- Explore all neighboring nodes at the present depth.
- Move to the next depth level and repeat the process.
- If a goal state is reached, return the solution.

Program:-

```
from collections import deque
def bfs(start_state):
 target = [1, 2, 3, 4, 5, 6, 7, 8 , 0]
 dq = deque([start_state])
 visited = {tuple(start_state): None}
 while dq:
 state = dq.popleft()
 if state == target:
 path = []
 while state:
 path.append(state)
 state = visited[tuple(state)]
return path[::-1]
 zero = state.index(0)
row, col = divmod(zero, 3)
 for move in (-3, 3, -1, 1):
```

```
 new_row, new_col = divmod(zero + move, 3)
 if 0 <= new_row < 3 and 0 <= new_col < 3 and abs(row - new_row) + abs(col - new_col)
== 1:  neighbor = state[:]
 neighbor[zero], neighbor[zero + move] = neighbor[zero + move],
neighbor[zero] if tuple(neighbor) not in visited:
 visited[tuple(neighbor)] = state
 dq.append(neighbor)
def printSolution(path):
 for state in path:
 print("\n".join(' '.join(map(str, state[i:i+3])) for i in range(0, 9, 3)), end="\n-----\n")
```

 # Example Usage

```
startState = [1, 3, 0 , 6, 8, 4, 7, 5, 2]
solution = bfs(startState)
if solution:
 printSolution(solution)
 print(f"Solved in {len(solution) - 1} moves.")
else:
 print("No solution found.")
```

Output:-

```
1 3 0
6 8 4
7 5 2
-----
1 3 4
6 8 0
7 5 2
-----
1 3 4
6 8 2
7 5 0
-----
1 3 4
6 8 2
7 0 5
-----
.
.
-----
4 5 0
7 8 6
-----
1 2 3
4 5 6
```

```
7 8 0
-----
```
Solved in 20 moves.

Using A*:-

A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. The A* algorithm is a heuristic search that combines aspects of both BFS and DFS. The A* algorithm uses heuristics to efficiently explore the state space and find an optimal solution faster than BFS or DFS.

Program:-

```
 def a_star(start_state):
 open_list = []
 closed_list = set()
 heapq.heappush(open_list, PuzzleState(start_state, None, None, 0,
heuristic(start_state)))  while open_list:
 current_state = heapq.heappop(open_list)

 if current_state.board == goal_state:
 return current_state
 closed_list.add(tuple(current_state.board))
 blank_pos = current_state.board.index(0)
 for move in moves:
 if move == 'U' and blank_pos < 3: # Invalid move up
 continue
 if move == 'D' and blank_pos > 5: # Invalid move down
 continue
 if move == 'L' and blank_pos % 3 == 0: # Invalid move left
 continue
 if move == 'R' and blank_pos % 3 == 2: # Invalid move right
 continue


 new_board = move_tile(current_state.board, move, blank_pos)
 if tuple(new_board) in closed_list:
 continue
 new_state = PuzzleState(new_board, current_state, move, current_state.depth
+ 1, current_state.depth + 1 + heuristic(new_board))
heapq.heappush(open_list, new_state)
 return None
# Function to print the solution path
 def print_solution(solution):
 path = []
```

```
 current = solution
 while current:
 path.append(current)
 current = current.parent
 path.reverse()
 for step in path:
 print(f"Move: {step.move}")
 print_board(step.board)
# Initial state of the puzzle
initial_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]
# Solve the puzzle using A* algorithm
 solution = a_star(initial_state)
# Print the solutionI
 f solution:
 print(colored("Solution found:", "green"))
 print_solution(solution)
else:
 print(colored("No solution exists.", "red"))
```

Output:-

Solution found:

Move: None

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
```

Move: R

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
.
.
```

Move: R

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
```

```
+---+---+---+
| 7 | | 8 |
+---+---+---+
```

Move: R

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 | | 
+---+---+---+
```