Name	**Hatim Yusuf Sawai**
UID no.	**2021300108**
Experiment No.	**3**

AIM:	**Implement A* Searching Algorithm**
PROBLEM DEFINITION:	**Implement a Sudoku problem using the Informed searching technique using A*. Analyze the algorithm with respect to Completeness, Optimality, time and space Complexity**

THEORY:

Introduction:

Sudoku is a popular number-placement puzzle requiring logical reasoning and deduction. Implementing an efficient Sudoku solver is a classic problem in artificial intelligence and machine learning. In this experiment, we utilize the informed searching technique, specifically the A* algorithm, to solve Sudoku puzzles. A* combines elements of both uniform cost search and heuristic search to efficiently explore the puzzle space and find a solution.

2			
	1		2
		3	
			4

A* Search Algorithm:

A* is a widely used informed search algorithm that combines the advantages of both uniform cost search and heuristic search. It's commonly applied to solve pathfinding and optimization problems.

Here's how A* works:

1. **Initialization:**
    - Start with an initial node (the start state) and a goal node (the target state).
    - Create an open list to store nodes to be evaluated and a closed list for nodes that have been evaluated.
2. **Evaluation Function:**
    - A* uses an evaluation function, denoted as "f(n)," for each node "n." The evaluation function combines two components:
        - "g(n)": The cost of the path from the start node to node "n."
        - "h(n)": A heuristic estimate of the cost from node "n" to the goal.
    - The total cost "f(n)" is the sum of "g(n)" and "h(n)."
3. **Add Start Node to Open List:**
    - Add the start node to the open list with "g(start) = 0" and compute "h(start)," typically using a heuristic that estimates the remaining cost to reach the goal.
4. **Main Loop:**
    - While the open list is not empty:
        - Select the node "n" with the lowest "f(n)" from the open list.
        - If "n" is the goal node, the algorithm terminates, and the solution is found.
        - Otherwise, move "n" from the open list to the closed list to mark it as evaluated.
5. **Expand Node "n":**
    - Generate the neighbors of node "n" (possible successor states).
    - For each neighbor "m," calculate:
        - "g(m)": The cost of the path from the start to "m" through "n" (i.e., "g(n) + cost(n, m)").
        - "h(m)": The heuristic estimate of the cost from "m" to the goal.

	- "f(m)": The total estimated cost ("g(m) + h(m)").
	6. **Update Nodes:**
	  - For each neighbor "m":
	    - If "m" is already in the open list and the new "g(m)" is lower than the previous "g(m)," update "g(m)" and "f(m)."
	    - If "m" is not in the open list, add it with the computed "g(m)" and "h(m)."
	7. **Repeat:**
	  - Repeat steps 4 to 6 until the goal node is reached or the open list becomes empty.
	8. **Path Reconstruction:**
	  - If a solution is found, reconstruct the path from the goal node to the start node using the "parent" pointers stored in each node.
	9. **Termination:**
	  - The algorithm terminates when either a solution is found or the open list is empty, indicating that no path to the goal exists.
	A* is complete (finds a solution if one exists) and optimal (finds the shortest path) under certain conditions:
	- "h(n)" is admissible (never overestimates the true cost to the goal).
	- The branching factor is finite.
	- The cost of each step is non-negative.

CODE:	import heapq
	import time
	
	puzzle = [
	    [2, 0, 0, 0],
	    [0, 1, 0, 2],
	    [0, 0, 3, 0],
	    [0, 0, 0, 4]
	]

```
goal_state = [
    [2, 4, 1, 3],
    [3, 1, 4, 2],
    [4, 2, 3, 1],
    [1, 3, 2, 4]
]

# heuristic function to calculate misplaced numbers.
def heuristic(node):
    misplaced = 0
    for i in range(4):
        for j in range(4):
            if node[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced

def generate_neighbors(node):
    neighbors = []
    for i in range(4):
        for j in range(4):
            if node[i][j] == 0:
                for num in range(1, 5):
                    neighbor = [row[:] for row in node]
                    neighbor[i][j] = num
                    neighbors.append(neighbor)
                return neighbors

# A* search algorithm.
def astar_search(start):
    open_list = [(heuristic(start), start)]
    closed_set = set()
    print("Initial state:")
    for row in puzzle:
        print(row)
    print()
    while open_list:
```

```
        _, current_node = heapq.heappop(open_list)

        if current_node == goal_state:
            return current_node

        closed_set.add(tuple(map(tuple, current_node)))

        for neighbor in generate_neighbors(current_node):
            if tuple(map(tuple, neighbor)) not in closed_set:
                g_value = 1
                f_value = g_value + heuristic(neighbor)
                print(f"cost = {f_value}")
                for row in neighbor:
                    print(row)
                print()
                heapq.heappush(open_list, (f_value, neighbor))

    return None

def main():
    t = time.time()
    solution = astar_search(puzzle)
    d = time.time() - t
    print("Final State:")
    if solution:
        for row in solution:
            print(row)
        print("\nDuration for Solving: %d ms" % (d * 1000))
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

OUTPUT:	

```
  Hitstar53 at …\AIML-Practicals on  main (⚠☑)
 ╰→ python -u "d:\SEM_5\AIML-Practicals\Exp3\sudoku.py"
Initial state:
[2, 0, 0, 0]
[0, 1, 0, 2]
[0, 0, 3, 0]
[0, 0, 0, 4]

cost = 12
[2, 1, 0, 0]
[0, 1, 0, 2]
[0, 0, 3, 0]
[0, 0, 0, 4]

cost = 12
[2, 2, 0, 0]
[0, 1, 0, 2]
[0, 0, 3, 0]
[0, 0, 0, 4]

cost = 2
[2, 4, 1, 3]
[3, 1, 4, 2]
[4, 2, 3, 1]
[1, 3, 3, 4]

cost = 2
[2, 4, 1, 3]
[3, 1, 4, 2]
[4, 2, 3, 1]
[1, 3, 4, 4]

Final State:
[2, 4, 1, 3]
[3, 1, 4, 2]
[4, 2, 3, 1]
[1, 3, 2, 4]

Duration for Solving: 74 ms
```

CONCLUSION:	In this experiment, we successfully applied the A* algorithm to solve Sudoku puzzles. A* demonstrated completeness, ensuring it can solve Sudoku instances with valid solutions. The optimality of A* depends on the chosen heuristic, with accurate heuristics improving solution quality.