CHANDIGARH UNIVERSITY, GHARUAN (MOHALI)

# Department of Computer Science & Engineering

# LAB FILE

**SUBJECT: Artificial intelligence and machine learning with lab (21CSH-316)**

## B.E. III Year – V Semester

## (CSE/IT)


**Submitted By**
Nishant Kumar Mehta
21BCS-3402
21BCS-IOT-602/B

**Submitted To**
Er. Parvez Rahi

**NAME:** **UID:**

# INDEX

| Exp, No | Experiments | Conduct (12) | Viva (10) | Record (8) | Total (30) | Signature |
|---|---|---|---|---|---|---|
| 1. | Evaluate the performance and effectiveness of the A° algorithm Implementation In Python. | | | | | |
| 2. | Implement the DFS algorithm and analyze Its performance and characteristics | | | | | |
| 3. | Implement the BFS algorithm and analyze its performance and characteristics | | | | | |
| 4. | Implementation of Python Libraries for ML application such as Pandas and Matplotlib | | | | | |
| 5. | Implementation of Python basic Libraries such as Math, Numpy and Scipy | | | | | |
| 6. | Implementing Linear Regression and Logistic Regression models | | | | | |
| 7. | To determine the optimal number of clusters (K) using K-means clustering algorithm | | | | | |
| 8. | Write a python program to compute Mean, Median, Mode, Variance and Standard Deviation using Datasets | | | | | |
| 9. | Implement Naive Bayes theorem to classify the English text | | | | | |
| 10. | Implement Exploratory Data Analysis on any data set. | | | | | |

# Experiment: 1.1

**Student Name: Nishant Kumar Mehta** **UID:** 21BCS-*3402*
**Branch:** CSE **Section/Group:** IOT-602/B
**Semester:** 5th **Date:** 16/08/2023
**Subject Name**: AIML Lab **Subject Code:** 21CSH-316

## 1. AIM: Implement A* algorithm in python.

## 2. Tools/Resource Used:

- Python programming language.
- A* algorithm implementation in Python.
- Relevant data or problem scenario for testing the algorithm.

## 3. Algorithm:

- Define the problem scenario or task for which the A* algorithm will be used.
- Implement the A* algorithm in Python, taking into accounts the specific problem requirements and constraints.
- Provide necessary data structures, such as graphs or grids, to represent the problem space.
- Write code to initialize the start and goal states or nodes.
- Implement the A* algorithm, including the heuristic function and the necessary data structures, such as priority queues or heaps.
- Run the algorithm on the given problem scenario and record the execution time.
- Monitor and log the nodes expanded, the path generated, and any other relevant information during the algorithm's execution.
- Repeat steps 4-7 for multiple problem scenarios or test cases, if applicabl

## 4. Program Code:

```
import heapq

class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0 # Cost from start node to current node
        self.h = 0 # Heuristic (estimated cost) from current node to goal node
        self.f = 0 # Total cost (g + h)

    def __lt__(self, other):
```

```
        return self.f < other.f

def heuristic(node, goal):
    # Manhattan distance heuristic (can be changed to Euclidean distance or
    others) return abs(node.position[0] - goal[0]) + abs(node.position[1] - goal[1])


                    def astar(grid, start, goal):
    open_list = []
    closed_set = set()

    start_node = Node(start)
    goal_node = Node(goal)

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.position == goal_node.position:
            path = []
            while current_node is not None:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(current_node.position)

        for next_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]: # Possible adjacent
positions
            node_position = (current_node.position[0] +
next_position[0], current_node.position[1] + next_position[1])

            if node_position[0] < 0 or node_position[0] >= len(grid)
or node_position[1] < 0 or node_position[1] >= len(grid[0]):
                continue

            if grid[node_position[0]][node_position[1]] == 1:
                continue

            if node_position in closed_set:
                continue

            new_node = Node(node_position, current_node)
            new_node.g = current_node.g + 1
```

```
                new_node.h = heuristic(new_node, goal_node.position)
            new_node.f = new_node.g + new_node.h

            for node in open_list:
                if new_node.position == node.position and new_node.f >= node.f:
                    break
            else:
                heapq.heappush(open_list, new_node)
    return None # No path found

# Example
usage: grid = [
    [0, 0, 0, 0],
    [0, 1, 1, 0],
    [0, 0, 0, 0],
    [0, 0, 1, 0]
]

start_point = (0, 0)
goal_point = (3, 3)

path = astar(grid, start_point, goal_point)
print(path)
```

## 5. Output/Result:

```
[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3, 3)]
```

## 6. Learning Outcomes:

- What is the A* algorithm, and how does it work?
- How is the heuristic function used in the A* algorithm?
- What are the advantages and disadvantages of the A* algorithm compared to other search algorithms?
- How does the choice of heuristic function impact the performance of the A* algorithm?