AIM: Identification and Installation of python environment towards the artificial intelligence and machine learning, installing python modules/Packages Import scikitlearn, keras etc.

DESCRIPTION: Artificial Intelligence and Machine Learning require a robust programming environment equipped with specialized libraries and tools. Python is the most widely used language for AI/ML due to its simplicity and extensive library support. This experiment focuses on:

- Installing Python and setting up an environment (Anaconda, Virtual Environment, or Google Colab).
- Installing essential AI/ML packages like numpy, pandas, matplotlib, scikit-learn, keras, and tensorflow.
- Verifying the successful installation of these packages.

PROCEDURE:

Step 1: Checking Python Installation

1. Open the terminal (Command Prompt or Anaconda Prompt).
2. Type the following command to check if Python is installed:
```
python --version
```
3. If Python is not installed, download and install it from Python's official website.

Step 2: Setting Up a Virtual Environment (Optional but Recommended)

1. Create a new virtual environment:
2. Activate the virtual environment:
```
python -m venv aiml_env
```
   - Windows:
```
aiml_env\Scripts\activate
```
   - Mac/Linux:
```
source aiml_env/bin/activate
```

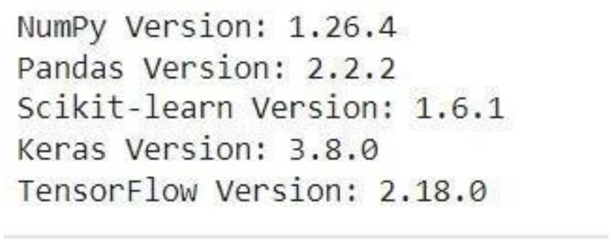
Step 3: Installing Essential Python Modules

Use pip to install AI/ML packages:
```
pip install numpy pandas matplotlib scikit-learn keras tensorflow
```

Step 4: Importing and Verifying Installed Packages

Create a Python script (verify_installation.py) and run the following code:
```
import numpy as np
import pandas as pd
import sklearn
import keras
import tensorflow as tf
print("NumPy Version:", np._version_)
print("Pandas Version:", pd._version_)
print("Scikit-learn Version:", sklearn._version_)
print("Keras Version:", keras._version_)
print("TensorFlow Version:", tf._version_)
```

OUTPUT:



```
NumPy Version: 1.26.4
Pandas Version: 2.2.2
Scikit-learn Version: 1.6.1
Keras Version: 3.8.0
TensorFlow Version: 2.18.0
```

WEEK-2

AIM: Implement A* algorithm on Graph Problem

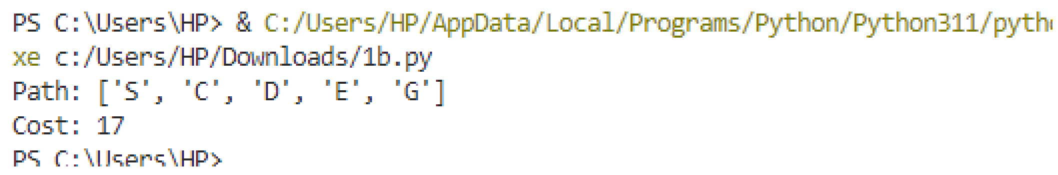
DISCRIPTION: The A* algorithm is a widely used graph traversal and pathfinding algorithm that finds the shortest path from a start node to a goal node. It combines the benefits of Dijkstra's algorithm and the Greedy Best-First Search by using a heuristic function to prioritize nodes. The algorithm maintains an open list of nodes to explore and a closed list of visited nodes, selecting the node with the lowest cost function f(n) = g(n) + h(n), where g(n) is the actual cost from the start node and h(n) is the estimated cost to the goal. A* is optimal and complete when using an admissible heuristic, making it efficient for applications like navigation, AI, and robotics.

CODE:

```
import heapq

def a_star(graph, start, goal, heuristic):

    open_list = []  # Priority queue (min-heap)

    heapq.heappush(open_list, (heuristic[start], 0, start))  # (f, g, node)

    came_from = {}  # To reconstruct path

    g_score = {start: 0}  # Shortest path cost from start

 while open_list:

        _, g, current = heapq.heappop(open_list)

        if current == goal:  # Goal reached, reconstruct path

            path = []

            while current in came_from:

                path.append(current)

                current = came_from[current]

            path.append(start)

            return path[::-1], g  # Reverse path to get correct order

        for neighbor, cost in graph[current]:

            tentative_g = g + cost  # Calculate new cost

            if neighbor not in g_score or tentative_g < g_score[neighbor]:

                g_score[neighbor] = tentative_g

                f_score = tentative_g + heuristic[neighbor]

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

                came_from[neighbor] = current  # Track path
```

```
  return None, None  # No path found

  graph = {
      'S': [('C', 3), ('B', 4)],
      'C': [('E', 10), ('D', 7)],
      'B': [('E', 12), ('F', 5)],
      'E': [('G', 5)],
      'D': [('E', 2)],
      'F': [('G', 16)],
      'G': []
  }

# Define heuristic values for each node
heuristics = {
    'S': 14, 'C': 11, 'B': 12, 'E': 4, 'D': 6, 'F': 11, 'G': 0
}
# Example start and goal nodes
start = 'S'
goal = 'G'

path, total_cost = a_star(graph, start, goal, heuristic)
if path:
    print("Path:", path)
    print("Cost", total_cost)
else:
    print("No path found")
```

OUTPUT:



```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python311/pyth
xe c:/Users/HP/Downloads/1b.py
Path: ['S', 'C', 'D', 'E', 'G']
Cost: 17
PS C:\Users\HP>
```

AIM: Implement A* algorithm on Grid Problems

DISCRIPTION: The A* algorithm is used in grid-based pathfinding problems to find the shortest path from a start position to a goal position. It works on a 2D grid where each cell represents a node and can be traversable or blocked. The algorithm prioritizes nodes using the cost function f(n) = g(n) + h(n), where g(n) is the actual distance from the start node and h(n) is the estimated distance to the goal (often using Manhattan or Euclidean distance). A* efficiently explores the most promising paths first, ensuring optimality when the heuristic is admissible. It is widely used in robotics, AI, and game development for navigation.

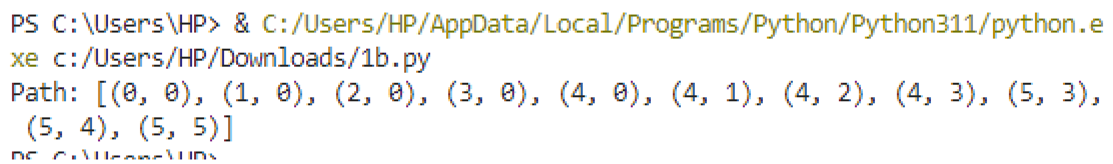
CODE:

```
import heapq

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

    def __lt__(self, other):
        return self.f < other.f  # Compare nodes based on total cost

def heuristic(a, b):
    """ Manhattan Distance Heuristic """
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_algorithm(grid, start, goal):
    open_list = []
    closed_list = set()

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

    heapq.heappush(open_list, start_node)  # Push start node to priority queue

    while open_list:
```

```
current_node = heapq.heappop(open_list)  # Get node with lowest f-score
closed_list.add(current_node.position)

# Goal reached, reconstruct the path
if current_node.position == goal_node.position:
    path = []
    while current_node:
        path.append(current_node.position)
        current_node = current_node.parent
    return path[::-1]  # Return reversed path

# Generate possible movements (up, down, left, right)
neighbors = [(0, -1), (0, 1), (-1, 0), (1, 0)]
for new_position in neighbors:
    node_position = (current_node.position[0] + new_position[0],
                     current_node.position[1] + new_position[1])

    # Check if within grid bounds
    if (node_position[0] < 0 or node_position[0] >= len(grid) or
        node_position[1] < 0 or node_position[1] >= len(grid[0])):
        continue  # Skip invalid positions

    # Check if cell is walkable (0 = walkable, 1 = obstacle)
    if grid[node_position[0]][node_position[1]] != 0:
        continue  # Skip obstacles

    neighbor = Node(node_position, current_node)

    if neighbor.position in closed_list:
        continue  # Ignore nodes already processed

    # Calculate g, h, and f scores
    neighbor.g = current_node.g + 1
    neighbor.h = heuristic(neighbor.position, goal_node.position)
    neighbor.f = neighbor.g + neighbor.h

    # Add neighbor to open list if it's not already explored with a better g-score
```

```
            if add_to_open(open_list, neighbor):
                heapq.heappush(open_list, neighbor)

    return None  # No path found

def add_to_open(open_list, neighbor):
    """ Check if neighbor is already in the open list with a lower g-score """
    for node in open_list:
        if neighbor.position == node.position and neighbor.g > node.g:
            return False
    return True

# Example grid (0 = open path, 1 = obstacle)
grid = [
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 1, 1, 0, 0, 0],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 1, 0, 0, 0]
]

# Define start and goal positions
start = (0, 0)  # Top-left corner
goal = (5, 5)   # Bottom-right corner
# Run A* algorithm
path = a_star_algorithm(grid, start, goal)
# Print results
print("Path:", path)
```

OUTPUT:



```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python311/python.e
xe c:/Users/HP/Downloads/1b.py
Path: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (5, 3),
 (5, 4), (5, 5)]
```

WEEK-3

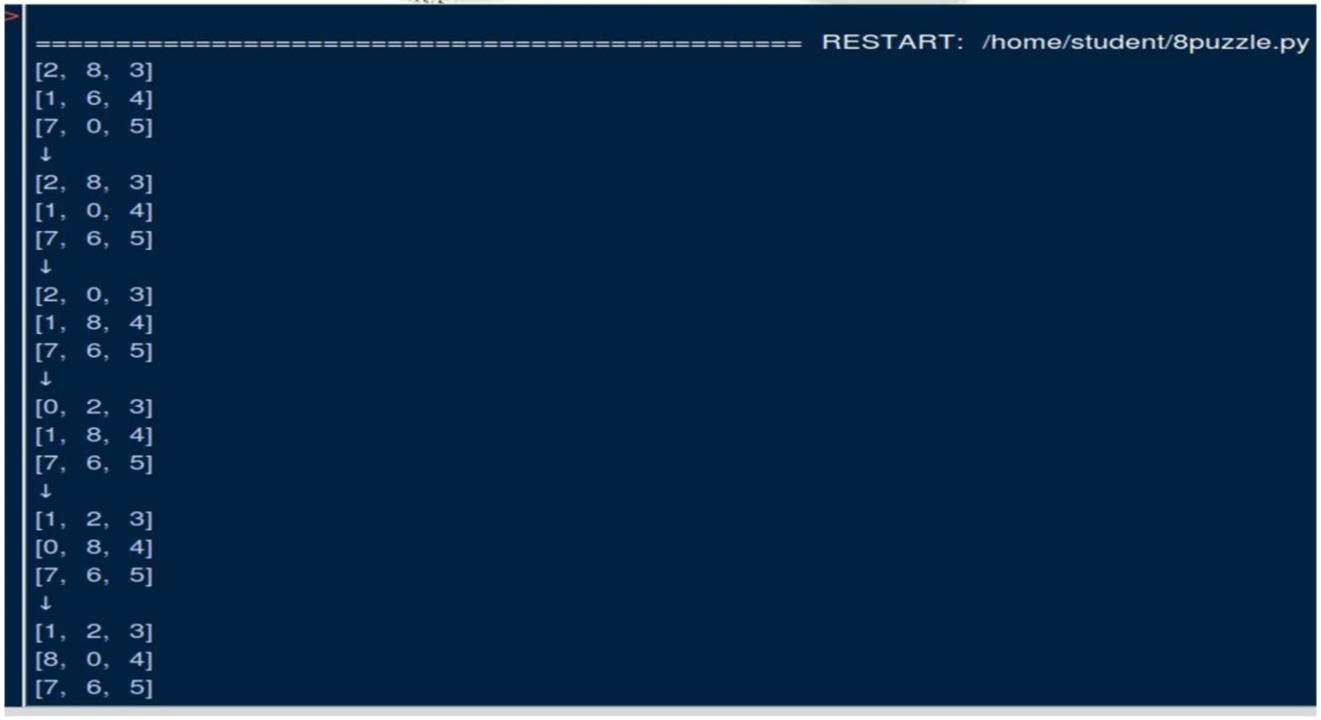
AIM: Implement an 8-puzzle solver using Heuristic search technique (Misplaced Tiles)

DISCRIPTION: An 8-puzzle solver using a heuristic search technique, such as the A* algorithm, finds the optimal sequence of moves to reach the goal state. The puzzle consists of a 3×3 grid with numbered tiles (1-8) and one empty space, where tiles can slide into the empty space. The solver uses heuristics like the **Manhattan Distance** (sum of tile distances from their goal positions) or **Misplaced Tiles** (count of tiles not in place) to guide the search efficiently. The A* algorithm evaluates each state using **f(n) = g(n) + h(n)**, where **g(n)** is the cost so far and **h(n)** is the estimated cost to the goal. This approach ensures an optimal and efficient solution to the problem.

CODE:

```
import heapq
def a_star(graph, start, goal, heuristic):
    open_list = []  # Priority queue (min-heap)
    heapq.heappush(open_list, (heuristic[start], 0, start))  # (f, g, node)
   came_from = {}  # To reconstruct path
    g_score = {start: 0}  # Shortest path cost from start
    while open_list:
        _, g, current = heapq.heappop(open_list)
        if current == goal:  # Goal reached, reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1], g  # Reverse path to get correct order
        for neighbor, cost in graph[current]:
            tentative_g = g + cost  # Calculate new cost
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic[neighbor]
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))
                came_from[neighbor] = current  # Track path
   return None, None  # No path found
graph = {  'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
    'J': [('E', 5), ('I', 3)]}
heuristic = { 'A': 10, 'B': 8, 'C': 5, 'D': 7, 'E': 3, 'F': 6, 'G': 5, 'H': 3, 'I': 1, 'J': 0}
start, goal = 'A', 'J'
path, total_cost = a_star(graph, start, goal, heuristic)
if path:
    print("Path found:", path)
    print("Shortest distance:", total_cost)
else:
    print("No path found")
```

OUTPUT:



```
================================================ RESTART: /home/student/8puzzle.py
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
↓
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
↓
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
↓
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
↓
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
↓
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

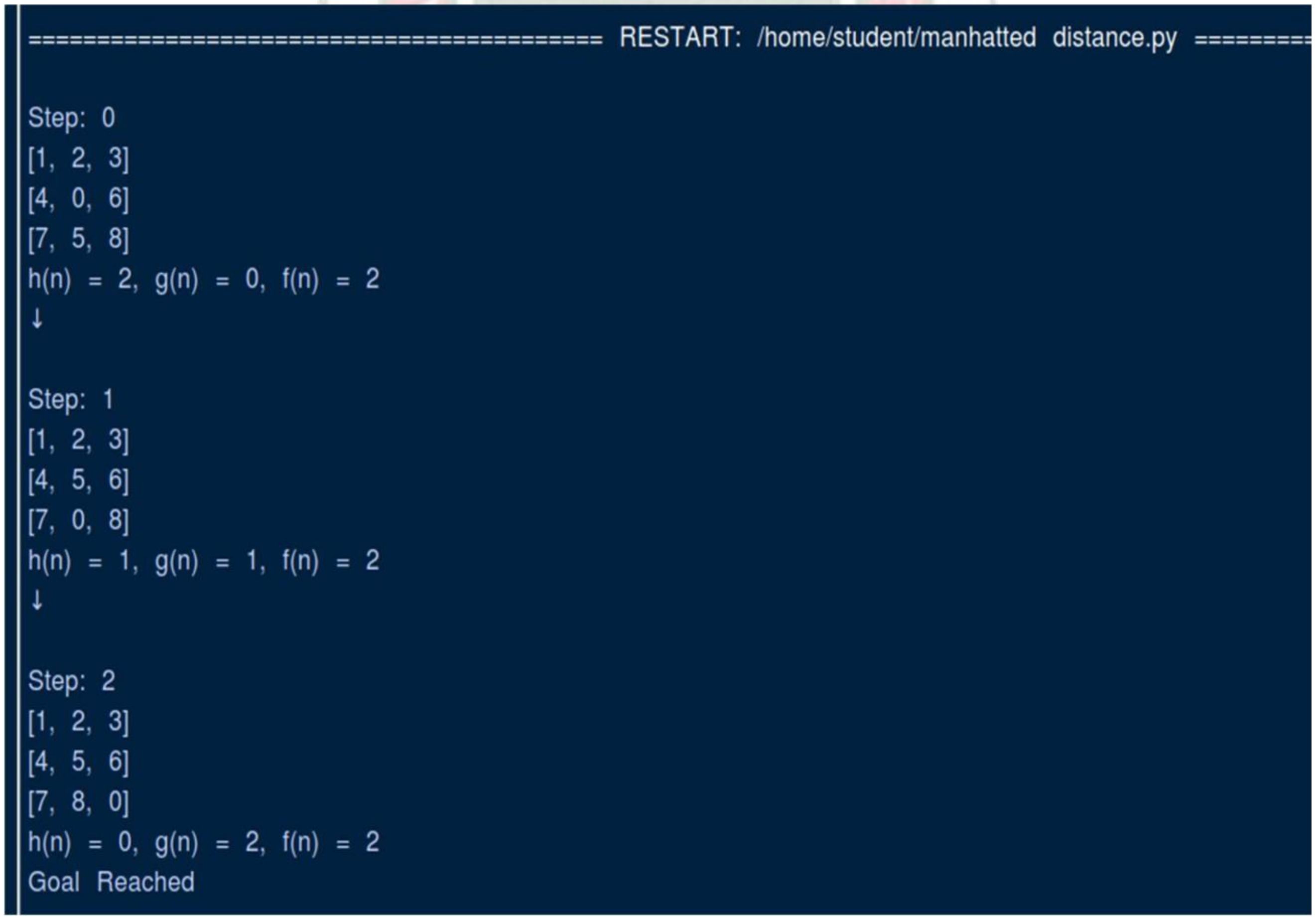
AIM: Implement 8 puzzle problem Using Manhattan Distance

DISCRIPTION: The A* algorithm using the Manhattan Distance heuristic is commonly applied to grid-based pathfinding problems. The Manhattan Distance heuristic h(n) calculates the sum of the absolute differences between the current node's coordinates and the goal node's coordinates (h(n) = |x1 - x2| + |y1 - y2|). The algorithm prioritizes nodes based on the cost function f(n) = g(n) + h(n), where g(n) is the actual cost from the start node. This heuristic is suitable for movement in a 4-directional grid (up, down, left, right) without diagonal movement. A* ensures an optimal path when the heuristic does not overestimate the actual cost. It is widely used in AI, robotics, and game development for pathfinding.

CODE:

```
 import heapq
class Puzzle:
    def _init_(self, board, g, parent=None):
        self.board = board
        self.g = g  # Depth (cost so far)
        self.h = self.manhattan_distance()
        self.f = self.g + self.h  # A* function: f(n) = g(n) + h(n)
        self.parent = parent  # To track the path
  def __lt__(self, other):
        return self.f < other.f  # Priority queue comparison
def manhattan_distance(self):
        goal_pos = {1: (0, 0), 2: (0, 1), 3: (0, 2),
                    4: (1, 0), 5: (1, 1), 6: (1, 2),
                    7: (2, 0), 8: (2, 1), 0: (2, 2)}
distance = 0
        for i in range(3):
            for j in range(3):
                value = self.board[i][j]
                if value != 0:  # Ignore blank tile
                    goal_x, goal_y = goal_pos[value]
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance
def get_blank_pos(self):
        """Find blank space (0)"""
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j
def generate_moves(self):
        """Generate possible moves"""
        x, y = self.get_blank_pos()
        moves = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = [row[:] for row in self.board]
                new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
                moves.append(Puzzle(new_board, self.g + 1, self))
return moves
def solve_puzzle(initial_state):
    """A* Search to solve the 8-puzzle"""
    start = Puzzle(initial_state, 0)
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
open_list = []
    closed_set = set()
    heapq.heappush(open_list, start)
while open_list:
        current = heapq.heappop(open_list)
if current.board == goal:
            path = []
            while current:
                path.append(current)
                current = current.parent
            return path[::-1]  # Reverse to get the correct order
closed_set.add(tuple(map(tuple, current.board)))
for move in current.generate_moves():
            if tuple(map(tuple, move.board)) not in closed_set:
                heapq.heappush(open_list, move)
return None  # No solution found
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
solution = solve_puzzle(initial_state)
if solution:
    for step in solution:
        print("\nStep:", step.g)
        for row in step.board:
            print(row)
        print(f"h(n) = {step.h}, g(n) = {step.g}, f(n) = {step.f}")
        print("↓" if step.board != [[1, 2, 3], [4, 5, 6], [7, 8, 0]] else "Goal Reached
else:
    print("No solution found.")
```

OUTPUT:



```
========================================= RESTART: /home/student/manhatted distance.py =========

Step: 0
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
h(n) = 2, g(n) = 0, f(n) = 2
↓

Step: 1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
h(n) = 1, g(n) = 1, f(n) = 2
↓

Step: 2
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
h(n) = 0, g(n) = 2, f(n) = 2
Goal Reached
```