ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LAB

NAME: G NIKHIL

ROLL: 160122733104

BRANCH: CSE

SECTION: 2

WEEK: 2

AIM: Implementation of A* algorithm using Parent Child.

DESCRIPTION: The A* algorithm is an informed search algorithm used for finding the shortest path from a start node to a goal node. It combines the advantages of Dijkstra's Algorithm and Greedy Best-First Search by using the function:

$$f(n)=g(n)+h(n)$$

In the A* algorithm, each node has a parent-child relationship that helps in reconstructing the optimal path once the goal is reached.

When the goal node is reached, we trace back from the goal to the start node using parent pointers to extract the shortest path

where:

- **g(n)**: Cost from the start node to the current node.
- **h(n)**: Heuristic (estimated) cost from the current node to the goal.
- **Parent Node:** The node from which we arrived at the current node.
- **Child Node:** The next node that can be explored from the current node.

CODE:

```
def astaralgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()

    g = {}
    parents = {}

    g[start_node] = 0
    parents[start_node] = start_node
```

```
    while len(open_set) > 0:
        n = None

        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or Graph_nodes.get(n) is None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)

        if n is None:
            print("Path does not exist!")
            return None

        if n == stop_node:
            path = []
            total_path_length = 0  # Variable to track the total path length
            while parents[n] != n:
                path.append(n)
                total_path_length += next(weight for m, weight in
get_neighbors(parents[n]) if m == n)
                n = parents[n]
```

```
            path.append(start_node)
            path.reverse()

            print("Path found: {}".format(path))
            print("Total path length: {}".format(total_path_length))
            return path

        open_set.remove(n)
        closed_set.add(n)

    print("Path does not exist!")
    return None

def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
    H_dist = {
        'S': 21,
        'B': 14,
        'C': 18,
        'D': 18,
        'E': 5,
        'F': 8,
        'G': 0,
    }
    return H_dist.get(n, 0)

Graph_nodes = {
    'S': [('B', 9), ('C', 4), ('D', 7)],
    'B': [('E', 11)],
    'C': [('E', 17), ('F', 12)],
    'E': [('G', 5)],
```

```
    'D': [('F', 14)],
    'F': [('G', 9)],
}

# Run the algorithm
astaralgo('S', 'G')
```

OUTPUT:

```
Path found: ['S', 'C', 'F', 'G']
Total path length: 25
['S', 'C', 'F', 'G']
```

AIM: Implementation of A* algorithm using the Grid.

DESCRIPTION: In grid-based pathfinding, the A* algorithm is commonly used to navigate through a 2D environment, such as a maze or a map with obstacles.

Steps in A for a Grid*

1. Define the grid (matrix where 0 = walkable, 1 = obstacle).
2. Set start and goal positions.
3. Expand nodes in four directions (up, down, left, right).
4. Calculate g(n), h(n), and f(n) for each node.
5. Use a priority queue (heap) to always expand the node with the lowest f(n).
6. Continue until the goal is reached.
7. Backtrack using the parent pointers to reconstruct the path.

CODE:

```
import math
import heapq

# Define the Cell class
```

```
class Cell:
    def __init__(self):
        self.parent_i = 0  # Parent cell's row index
        self.parent_j = 0  # Parent cell's column index
        self.f = float('inf')  # Total cost of the cell (g + h)
        self.g = float('inf')  # Cost from start to this cell
        self.h = 0  # Heuristic cost from this cell to destination

# Define the size of the grid
ROW = 9
COL = 10

# Check if a cell is valid (within the grid)
def is_valid(row, col):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

# Check if a cell is unblocked
def is_unblocked(grid, row, col):
    return grid[row][col] == 1

# Check if a cell is the destination
def is_destination(row, col, dest):
    return row == dest[0] and col == dest[1]

# Calculate the heuristic value of a cell (Euclidean distance to destination)
def calculate_h_value(row, col, dest):
    return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5

# Trace the path from source to destination
def trace_path(cell_details, dest):
    print("The Path is ")
    path = []
    row = dest[0]
    col = dest[1]

    # Trace the path from destination to source using parent cells
```

```
    while not (cell_details[row][col].parent_i == row and
cell_details[row][col].parent_j == col):
        path.append((row, col))
        temp_row = cell_details[row][col].parent_i
        temp_col = cell_details[row][col].parent_j
        row = temp_row
        col = temp_col

    # Add the source cell to the path
    path.append((row, col))
    # Reverse the path to get the path from source to destination
    path.reverse()

    # Print the path
    for i in path:
        print("->", i, end=" ")
    print()

# Implement the A* search algorithm
def a_star_search(grid, src, dest):
    # Check if the source and destination are valid
    if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
        print("Source or destination is invalid")
        return

    # Check if the source and destination are unblocked
    if not is_unblocked(grid, src[0], src[1]) or not is_unblocked(grid, dest[0],
dest[1]):
        print("Source or the destination is blocked")
        return

    # Check if we are already at the destination
    if is_destination(src[0], src[1], dest):
        print("We are already at the destination")
        return

    # Initialize the closed list (visited cells)
```

```
closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
# Initialize the details of each cell
cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]

# Initialize the start cell details
i = src[0]
j = src[1]
cell_details[i][j].f = 0
cell_details[i][j].g = 0
cell_details[i][j].h = 0
cell_details[i][j].parent_i = i
cell_details[i][j].parent_j = j

# Initialize the open list (cells to be visited) with the start cell
open_list = []
heapq.heappush(open_list, (0.0, i, j))

# Initialize the flag for whether destination is found
found_dest = False

# Main loop of A* search algorithm
while len(open_list) > 0:
    # Pop the cell with the smallest f value from the open list
    p = heapq.heappop(open_list)

    # Mark the cell as visited
    i = p[1]
    j = p[2]
    closed_list[i][j] = True

    # For each direction, check the successors
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]
    for dir in directions:
        new_i = i + dir[0]
        new_j = j + dir[1]

        # If the successor is valid, unblocked, and not visited
```

```
            if is_valid(new_i, new_j) and is_unblocked(grid, new_i, new_j) and not
closed_list[new_i][new_j]:
                # If the successor is the destination
                if is_destination(new_i, new_j, dest):
                    # Set the parent of the destination cell
                    cell_details[new_i][new_j].parent_i = i
                    cell_details[new_i][new_j].parent_j = j
                    print("The destination cell is found")
                    # Trace and print the path from source to destination
                    trace_path(cell_details, dest)
                    found_dest = True
                    return
                else:
                    # Calculate the new f, g, and h values
                    g_new = cell_details[i][j].g + 1.0
                    h_new = calculate_h_value(new_i, new_j, dest)
                    f_new = g_new + h_new

                    # If the cell is not in the open list or the new f value is smaller
                    if cell_details[new_i][new_j].f == float('inf') or
cell_details[new_i][new_j].f > f_new:
                        # Add the cell to the open list
                        heapq.heappush(open_list, (f_new, new_i, new_j))
                        # Update the cell details
                        cell_details[new_i][new_j].f = f_new
                        cell_details[new_i][new_j].g = g_new
                        cell_details[new_i][new_j].h = h_new
                        cell_details[new_i][new_j].parent_i = i
                        cell_details[new_i][new_j].parent_j = j

    # If the destination is not found after visiting all cells
    if not found_dest:
        print("Failed to find the destination cell")

def main():
    # Define the grid (1 for unblocked, 0 for blocked)
    grid = [
```

```
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
        [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
    ]

    # Define the source and destination
    src = [8, 0]
    dest = [0, 0]

    # Run the A* search algorithm
    a_star_search(grid, src, dest)

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

OUTPUT:

```
The destination cell is found
The Path is
-> (8, 0) -> (7, 0) -> (6, 0) -> (5, 0) -> (4, 1) -> (3, 2) -> (2, 1) -> (1, 0) -> (0, 0)
```

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LAB

NAME: G NIKHIL

ROLL: 160122733104

BRANCH: CSE

SECTION: 2

WEEK: 3

Signature of the Faculty………………………...

AIM: To Implement an 8-puzzle solver using Heuristic Euclidian technique.

DESCRIPTION: The Euclidean Distance heuristic is based on the straight-line distance between a tile's current position and its goal position. It is used in the *A algorithm** to guide the search towards the goal state efficiently.

CODE:

```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated
fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
```

```
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
            return children

    def shuffle(self,puz,x1,y1,x2,y2):
        """ Move the blank space in the given direction and if the position value
are out
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

    def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
```

```
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j

class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to
empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz
```

```
def f(self,start,goal):
    """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
    return self.h(start.data,goal)+start.level

def h(self,start,goal):
    """ Calculates the different between the given puzzles """
    temp = 0
    for i in range(0,self.n):
        for j in range(0,self.n):
            if start[i][j] != goal[i][j] and start[i][j] != '_':
                temp += 1
    return temp

def process(self):
    """ Accept Start and Goal Puzzle state"""
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix \n")
    goal = self.accept()

    start = Node(start,0,0)
    start.fval = self.f(start,goal)
    """ Put the start node in the open list"""
```

```
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have reached
the goal node"""
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)
```

puz = Puzzle(3)

puz.process()

OUTPUT:

```
Enter the start state matrix

1 2 3
4 _ 5
6 7 8
Enter the goal state matrix

1 2 3
4 5 _
6 7 8




  |
  |
 \'/

1 2 3
4 _ 5
6 7 8

  |
  |
 \'/

1 2 3
4 5 _
6 7 8
```

AIM: To Implement an 8-puzzle solver using Heuristic Manhattan technique.

DESCRIPTION: The Manhattan Distance heuristic is one of the most effective heuristics for grid-based pathfinding problems, including the 8-puzzle problem. It calculates the sum of the horizontal and vertical moves required to place each tile in its correct position.

CODE:

```
class Node:
    def __init__(self, data, level, fval):
        """ Initialize the node with the data, level of the node and the calculated
fvalue """
        self.data = data
        self.level = level
        self.fval = fval
    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up, down, left, right} """
        x, y = self.find(self.data, '_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up, down, left, right] respectively. """
        val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
```

```
                child_node = Node(child, self.level+1, 0)
                children.append(child_node)
            return children
        def shuffle(self, puz, x1, y1, x2, y2):
            """ Move the blank space in the given direction and if the position value
    are out
                of limits the return None """
            if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
                temp_puz = self.copy(puz)
                temp = temp_puz[x2][y2]
                temp_puz[x2][y2] = temp_puz[x1][y1]
                temp_puz[x1][y1] = temp
                return temp_puz
            else:
                return None
        def copy(self, root):
            """ Copy function to create a similar matrix of the given node """
            temp = []
            for i in root:
                t = []
                for j in i:
                    t.append(j)
                temp.append(t)
            return temp
        def find(self, puz, x):
```

```
        """ Specifically used to find the position of the blank space """
        for i in range(len(self.data)):
            for j in range(len(self.data)):
                if puz[i][j] == x:
                    return i, j
class Puzzle:
    def __init__(self, size):
        """ Initialize the puzzle size by the specified size, open and closed lists to
empty """
        self.n = size
        self.open = []
        self.closed = []
    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self, start, goal):
        """ Heuristic Function to calculate f(x) = h(x) + g(x) """
        return self.h(start.data, goal) + start.level
    def h(self, start, goal):
```

```
        """ Calculates the Manhattan distance between the current and goal puzzle
states """
        temp = 0
        for i in range(self.n):
            for j in range(self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':  # Ignore the blank space
                    # Find the goal position of the tile
                    goal_x, goal_y = self.find(goal, start[i][j])
                    # Add Manhattan distance to temp
                    temp += abs(i - goal_x) + abs(j - goal_y)
        return temp
    def find(self, puz, x):
        """ Specifically used to find the position of a tile (not just the blank space)
"""
        for i in range(len(puz)):
            for j in range(len(puz)):
                if puz[i][j] == x:
                    return i, j
    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix (use space-separated values and '_' for
blank):\n")
        start = self.accept()
        print("Enter the goal state matrix (use space-separated values and '_' for
blank):\n")
        goal = self.accept()
```

```
        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("Current state:")
            for i in cur.data:
                print(" ".join(i))
            print("\n")
            """ If the difference between current and goal node is 0, we have
reached the goal node """
            if self.h(cur.data, goal) == 0:
                print("Goal reached!")
                break
            for i in cur.generate_child():
                i.fval = self.f(i, goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]
            """ Sort the open list based on f value """
            self.open.sort(key=lambda x: x.fval, reverse=False)
# Run the puzzle solver
puz = Puzzle(3)
```

puz.process()

OUTPUT:

```
Enter the start state matrix (use space-separated values and '_' for blank):

1 2 3
4 5 6
7 _ 8
Enter the goal state matrix (use space-separated values and '_' for blank):

1 2 3
4 5 6
7 8 _



Current state:
1 2 3
4 5 6
7 _ 8


Current state:
1 2 3
4 5 6
7 8 _


Goal reached!
```

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LAB

NAME: G NIKHIL

ROLL: 160122733104

BRANCH: CSE

SECTION: 2

WEEK: 1

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:

```python
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
```