EXPERIMENT – 1

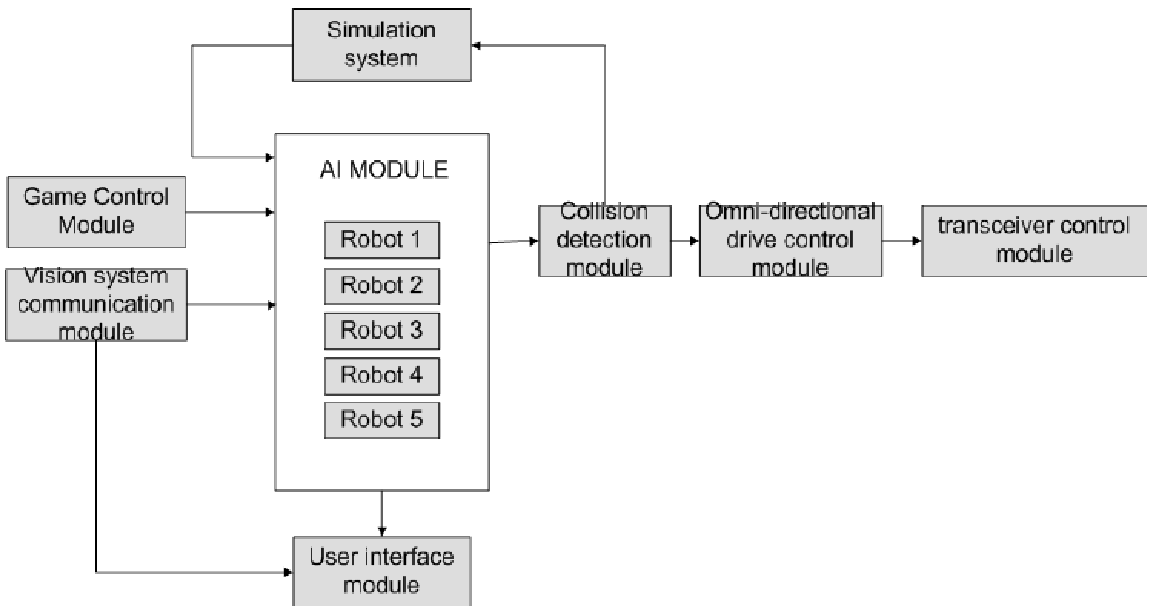
AIM: Explain the workflow of general AI Project.

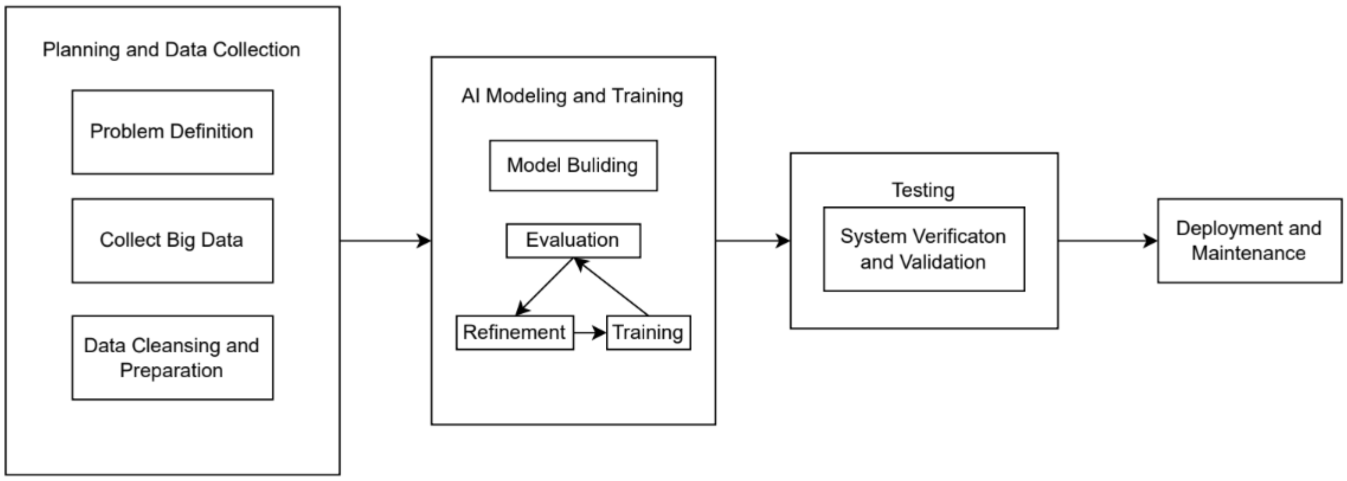
DESCRIPTION: The workflow of a general AI project typically involves the following stages:

1. Problem identification: The first step is to identify the problem or the task that the AI system needs to accomplish. This involves understanding the requirements of the project and defining the goals and objectives that the AI system needs to achieve.
2. Data collection and preparation: The next step is to collect and prepare the data that will be used to train the AI system. This may involve gathering data from various sources, cleaning and organizing the data, and preparing it for use in the training process.
3. Model selection and training: The third step is to select an appropriate model for the AI system and train it using the prepared data. This involves using various algorithms and techniques to train the model to recognize patterns and make predictions based on the data.
4. Model evaluation: Once the model has been trained, it is important to evaluate its performance to determine how well it is able to accomplish the task at hand. This may involve testing the model on new data and comparing its performance to that of other models.
5. Model optimization: Based on the results of the evaluation, the model may need to be optimized to improve its performance. This may involve adjusting the model's parameters or using different algorithms or techniques to improve its accuracy and efficiency.
6. Deployment: Once the model has been optimized, it can be deployed in the production environment where it can be used to accomplish the task at hand. This may involve integrating the AI system with other systems or applications and ensuring that it is functioning properly.
7. Monitoring and maintenance: After deployment, it is important to monitor the performance of the AI system and make any necessary adjustments to ensure that it continues to function properly. This may involve performing regular maintenance and updates, as well as monitoring the system for any issues or errors that may arise.

EXPERIMENTAL RESULTS:

Artificial Intelligence System Architecture:


Simulation system
AI MODULE
Game Control Module
Vision system communication module
Robot 1
Robot 2
Robot 3
Robot 4
Robot 5
Collision detection module
Omni-directional drive control module
transceiver control module
User interface module


Planning and Data Collection
Problem Definition
Collect Big Data
Data Cleansing and Preparation
AI Modeling and Training
Model Buliding
Evaluation
Refinement
Training
Testing
System Verificaton and Validation
Deployment and Maintenance

INFERENCES AND CONCLUSIONS:

In summary, a general AI project's workflow includes the following stages: problem identification, data preparation and preparation, model selection and training, model evaluation, model optimisation, deployment, monitoring, and maintenance. For an AI system to be successfully developed and put into use and be able to carry out the required task or address the recognised issue, each of these stages is essential. Organizations may use the potential of AI to drive innovation and accomplish their business objectives by adhering to this procedure and regularly monitoring and optimising the AI system.

REFERENCES:

- https://towardsdatascience.com/how-to-design-an-artificial-intelligent-system-part-1-concept-development-cdbc8aee30d8
- https://www.edn.com/four-basic-steps-in-implementing-an-ai-driven-design-workflow/
- https://www.kdnuggets.com/2020/11/mathworks-ai-four-steps-workflow.html
- https://labelyourdata.com/articles/lifecycle-of-an-ai-project-stages-breakdown

EXPERIMENT – 2

AIM: Write a program to demonstrate Water Jug Problem

DESCRIPTION:

The water jug problem is a classic puzzle where we are given two jugs of different capacities and an unlimited supply of water. The goal is to use these jugs to obtain a certain quantity of water. At any point, we can either fill a jug with water, empty a jug, or transfer water from one jug to the other until one of the jugs is completely full or empty.

To solve this problem using BFS, we can represent each state of the jugs as a node in a graph. The edges between nodes represent the possible actions we can take to move from one state to another.

For example, if we have two jugs A and B with capacities 4 and 3 respectively, and we want to obtain 2 units of water, we can represent the initial state as (0, 0), where both jugs are empty. The goal state is (2, any value), where A has 2 units of water.

We can generate all possible states by applying the following actions:

1. Fill a jug: we can fill jug A to capacity (4, y) or jug B to capacity (x, 3).
2. Empty a jug: we can empty jug A (0, y) or jug B (x, 0).
3. Transfer water: we can transfer water from jug A to jug B until B is full (x+y, 3), or transfer water from jug B to jug A until A is full (4, x+y).

Using BFS, we start at the initial state and explore all possible states that can be reached by applying the above actions. We keep track of the visited states to avoid revisiting them. When we reach the goal state, we stop the search and return the sequence of actions that led to the goal state.

Using DFS, It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

EXPERIMENTAL RESULTS:

Program:

Using BFS:

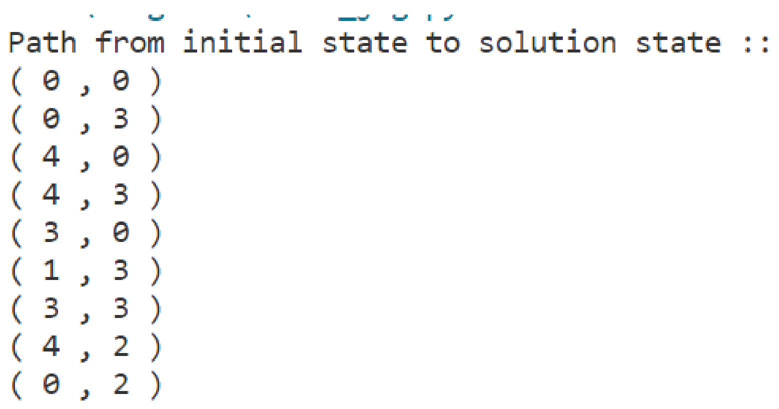
```
from collections import deque
def Solution(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    #Initializing with jugs being empty
    q.append((0, 0))
    while (len(q) > 0):
        # Current state
        u = q.popleft()
        if ((u[0], u[1]) in m):
            continue
        if ((u[0] > a or u[1] > b or
            u[0] < 0 or u[1] < 0)):
            continue
        path.append([u[0], u[1]])

        m[(u[0], u[1])] = 1

        if (u[0] == target or u[1] == target):
            isSolvable = True
            if (u[0] == target):
                if (u[1] != 0):
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):
                    path.append([0, u[1]])
            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                    path[i][1], ")")
            break

        q.append([u[0], b]) # Fill Jug2
        q.append([a, u[1]]) # Fill Jug1

        for ap in range(max(a, b) + 1):
            c = u[0] + ap
            d = u[1] - ap
            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])
            c = u[0] - ap
            d = u[1] + ap
            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])
        q.append([a, 0])
        q.append([0, b])

    if (not isSolvable):
        print("Solution not possible")

Jug1, Jug2, target = 4, 3, 2
print("Path from initial state to solution state ::")
Solution(Jug1, Jug2, target)
```

Using DFS:

```
class Node:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent

    def get_child_nodes(self, capacities):
        a, b = self.state
        max_a, max_b = capacities
        children = []
        children.append(Node((max_a, b), self))
        children.append(Node((a, max_b), self))
        children.append(Node((0, b), self))
        children.append(Node((a, 0), self))
        if a + b >= max_b:
            children.append(Node((a - (max_b - b), max_b), self))
        else:
            children.append(Node((0, a + b), self))
        if a + b >= max_a:
            children.append(Node((max_a, b - (max_a - a)), self))
        else:
            children.append(Node((a + b, 0), self))
        return children
def dfs(start_state, goal_state, capacities):
    start_node = Node(start_state, None)
    visited = set()
    stack = [start_node]

    while stack:
        node = stack.pop()
        if node.state == goal_state:
            path = []
            while node.parent:
                path.append(node.state)
                node = node.parent
            path.append(start_state)
            path.reverse()
            return path
        if node.state not in visited:
            visited.add(node.state)
            for child in node.get_child_nodes(capacities):
                stack.append(child)

    return None
start_state = (0, 0)
a,b=map(int, input("Enter the capacities of jugs: ").split())
c,d=map(int, input("Enter the capacities of goal state: ").split())
goal_state = (c, d)
capacities = (a, b)
path = dfs(start_state, goal_state, capacities)
print(path)
```

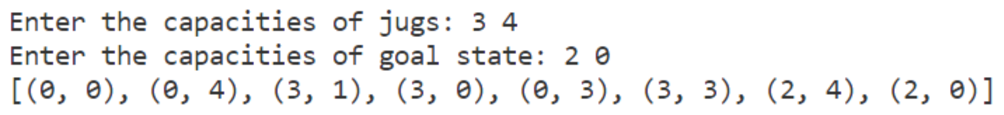
Output:

Using BFS:



```
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

Using DFS:



```
Enter the capacities of jugs: 3 4
Enter the capacities of goal state: 2 0
[(0, 0), (0, 4), (3, 1), (3, 0), (0, 3), (3, 3), (2, 4), (2, 0)]
```

INFERENCES AND CONCLUSION:

ANALYSIS:

Time complexity:

The time complexity of BFS for the water jug problem is O(ab), where a and b are the capacities of the two jugs. This is because there are a*b possible states and each state is visited once. However, in practice, the number of possible states is much smaller due to the constraints of the problem.

Space complexity:

The space complexity of BFS is also O(ab) because we need to store all possible states in memory.

CONCLUSION:

In conclusion, the water jug problem is a classic puzzle that can be solved using the BFS algorithm. By representing each state of the jugs as a node in a graph and using the possible actions as edges between nodes, we can explore all possible states and find the sequence of actions that leads to the goal state.

Although the time complexity of BFS for this problem is O(ab), where a and b are the capacities of the two jugs, the actual number of possible states is much smaller due to the constraints of the problem. Therefore, BFS is a practical approach for solving this problem.

References:

- https://www.futurelearn.com/info/courses/recreational-math/0/steps/43519#:~:text=The%20water%20jug%20problem%20can,diagrams%20that%20solve%20this%20problem.
- https://www.geeksforgeeks.org/two-water-jug-puzzle/

EXPERIMENT – 3

AIM: To implement A* Search algorithm

DESCRIPTION:

A* search algorithm is a heuristic search algorithm that is commonly used for pathfinding and graph traversal problems. It is an extension of Dijkstra's algorithm that uses heuristics to guide the search towards the goal state.

1. Implement the A* search algorithm: Use a priority queue to store the states to be explored, ordered by their priority (the sum of the cost of the path to the state and the estimated cost to reach the goal state). At each step, select the state with the lowest priority from the queue, expand it by generating its successors, and add them to the queue if they have not been visited before or if their cost estimates have been improved.
2. Terminate the search: The search terminates when the goal state is reached or when the priority queue becomes empty.
3. Extract the solution: If the search succeeds in reaching the goal state, extract the path from the initial state to the goal state from the information stored in the nodes of the search space.

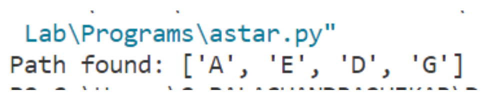
Overall, the A* search algorithm is a powerful heuristic search algorithm that can be used to solve a wide range of pathfinding and graph traversal problems.

EXPERIMENTAL RESULTS:

Program:

```
def aStarAlgo(start_node, stop_node):

        open_set = set(start_node)
        closed_set = set()
        g = {} #store distance from starting node
        parents = {}# parents contains an adjacency map of all nodes

        #ditance of starting node from itself is zero
        g[start_node] = 0
        #start_node is root node i.e it has no parent nodes
        #so start_node is set to its own parent node
        parents[start_node] = start_node
        while len(open_set) > 0:
            n = None
            #node with lowest f() is found
            for v in open_set:
                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                    n = v
            if n == stop_node or Graph_nodes[n] == None:
                pass
            else:
                for (m, weight) in get_neighbors(n):
                    #nodes 'm' not in first and last set are added to first
                    #n is set its parent
                    if m not in open_set and m not in closed_set:
                        open_set.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight
                    #for each node m,compare its distance from start i.e g(m) to the
                    #from start through n node
                    else:
                        if g[m] > g[n] + weight:
                            #update g(m)
                            g[m] = g[n] + weight
                            #change parent of m to n
                            parents[m] = n
                            #if m in closed set,remove and add to open
                            if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)

            if n == None:
                print('Path does not exist!')
                return None
            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                path = []
                while parents[n] != n:
                    path.append(n)
                    n = parents[n]
                path.append(start_node)
                path.reverse()
                print('Path found: {}'.format(path))
                return path
            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_set.remove(n)
            closed_set.add(n)

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

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,
        }
        return H_dist[n]
#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

Output:



```
  Lab\Programs\astar.py"
 Path found: ['A', 'E', 'D', 'G']
```

INFERENCES AND CONCLUSION:

ANALYSIS:

Time complexity

The time complexity of A* search is proportional to the number of states explored, which is influenced by the quality of the heuristic function. A* search is guaranteed to find the optimal path if the heuristic function is admissible and consistent. In the worst-case scenario, the time complexity of A* search is O(b^d), where b is the branching factor of the graph and d is the depth of the solution.

In the given implementation, the time complexity of the algorithm is affected by the following factors:

- The size of the open set and closed set, which can grow quite large in the worst-case scenario
- The number of neighbors for each node, which can impact the time required to explore each possible path
- The efficiency of the heuristic function, which determines how quickly the algorithm converges on the optimal solution.

Therefore, it is difficult to provide a precise time complexity for this implementation without additional information about the graph being searched and the heuristic function being used.

Space complexity

The space complexity of A* search is proportional to the number of states visited and stored in memory during the search. The space complexity of the A* algorithm depends on the number of nodes that are stored in the open set and the closed set, as well as the size of the parents and g dictionaries.

In the worst-case scenario, where all nodes are in the open set, the space complexity would be O(b^d), where b is the branching factor and d is the depth of the solution. However, in practice, the space complexity is much lower due to the use of heuristics and the pruning of nodes that are not likely to lead to a solution.

Therefore, the space complexity of the A* algorithm is typically considered to be O(|V|), where |V| is the number of nodes in the graph.

CONCLUSION:

In conclusion, the above code implements the A* search algorithm to find the shortest path from a given start node to a goal node in a graph. The algorithm uses a heuristic function to estimate the distance to the goal node and uses this estimate to determine the order in which nodes are explored. The code has a time complexity of O(b^d), where b is the branching factor of the graph and d is the depth of the goal node from the start node. The space complexity of the code is also O(b^d), as it stores information about the explored nodes and their parents in memory. Overall, the A* search algorithm is a powerful and efficient algorithm for solving pathfinding problems in a variety of applications.

REFERENCES:

- https://www.geeksforgeeks.org/a-search-algorithm/
- https://www.redblobgames.com/pathfinding/a-star/introduction.html

EXPERIMENT – 4

AIM: To implement 8-puzzle solver using Heuristic search technique (A* Search algorithm)

DESCRIPTION:

This code is an implementation of the A* search algorithm for solving the 8-puzzle problem. The 8-puzzle problem is a sliding puzzle consisting of a 3x3 grid with eight numbered tiles and one blank tile. The goal of the puzzle is to move the tiles into the goal state, which is usually represented as a specific arrangement of the numbers in the grid.

The program consists of two classes: Node and Puzzle.

The Node class represents a single node in the search tree. It contains three properties: data, level, and fval. The data property is a 2D array representing the current state of the puzzle. The level property represents the depth of the node in the search tree. The fval property is the sum of the heuristic value and the level of the node.

The Node class also contains several methods. The generate_child() method generates child nodes by moving the blank tile in the four directions (up, down, left, right) and returns them in a list. The shuffle() method moves the blank tile in a given direction and returns the resulting state of the puzzle. The copy() method creates a copy of the puzzle state. The find() method finds the position of the blank tile in the puzzle state.

The Puzzle class represents the puzzle itself. It contains three properties: n, open, and closed. The n property is the size of the puzzle grid (3 in this case). The open property is a list of nodes that have been generated but not yet expanded. The closed property is a list of nodes that have been expanded.

The Puzzle class also contains several methods. The accept() method accepts the puzzle state from the user. The f() method calculates the heuristic value of a node. The h() method calculates the difference between two puzzle states. The process() method is the main method that implements the A* search algorithm. It accepts the start and goal states of the puzzle, initializes the start node, and adds it to the open list. It then repeatedly selects the node with the lowest fval value from the open list, generates its child nodes, calculates their fval values, adds them to the open list, and removes the selected node from the open list and adds it to the closed list. The process continues until the goal state is reached.

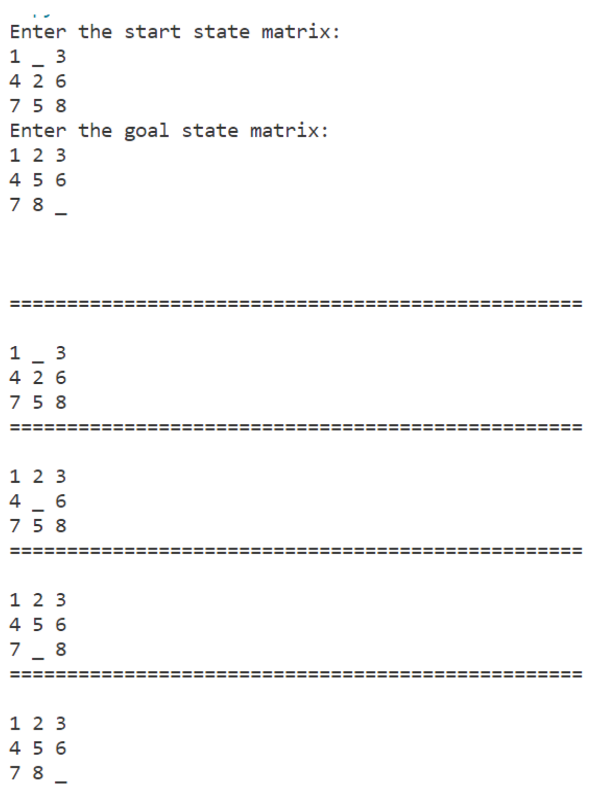
Finally, the main part of the code creates an instance of the Puzzle class and calls its process() method to start the search algorithm. The user is prompted to enter the size of the puzzle grid and the start and goal states of the puzzle. The program then prints the intermediate puzzle states and finally prints the solution when the goal state is reached.

EXPERIMENTAL RESULTS:

Program:

```python
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 hild nodes from the given node by moving the blank space
        # either in the four direction {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 direction [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 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 Heuristic value f(x) = h(x) + g(x)
        return self.h(start.data, goal) + start.level
    def h(self, start, goal):
        # Calculates the difference 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("==================================================\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 open 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 _ 3
4 2 6
7 5 8
Enter the goal state matrix:
1 2 3
4 5 6
7 8 _



==================================================

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

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

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

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

INFERENCES AND CONCLUSION:

ANALYSIS:

Time complexity

The time complexity of the above code is dependent on the size of the puzzle (n) and the number of states explored in the search algorithm.

In terms of the puzzle size (n), the time complexity of generating the child nodes is O(1) as it only involves constant time operations. The time complexity of the shuffle function used in generating the child nodes is O(1) as well. The time complexity of the copy function used in generating the child nodes is O(n^2) as it involves copying a 2D array of size n x n.

The time complexity of the h function used in the A* search algorithm is O(n^2) as it involves iterating over a 2D array of size n x n.

Therefore, the overall time complexity of the code is highly dependent on the size of the puzzle and the number of states explored during the search algorithm. It is difficult to determine the exact time complexity without analyzing the specific puzzle and its solution.

Space complexity

The space complexity of the above code depends on the size of the puzzle and the number of nodes generated during the search process.

The open and closed lists in the **Puzzle** class store the nodes that have been explored and those that are yet to be explored. The maximum size of the open list at any point in time can be at most the total number of nodes in the search tree, which is given by the branching factor raised to the depth of the shallowest solution. In this case, the branching factor is 4 (since each node can have up to 4 possible children), and the depth of the shallowest solution is unknown. Therefore, the space complexity of the search algorithm is exponential in the worst case.

Additionally, the **generate_child** method in the **Node** class creates a list of all possible children of a node. The maximum number of children for a given node is 4 (since each node can move in up to 4 directions), so the space complexity of the **generate_child** method is constant.

Therefore, the overall space complexity of the algorithm is exponential in the worst case.

CONCLUSION:

In conclusion, the above code is an implementation of the A* search algorithm to solve the 8-puzzle problem. It uses heuristics to estimate the distance from the current state to the goal state and sorts the open list based on f value to select the next node to expand. The time complexity of the algorithm is O(b^d), where b is the branching factor and d is the depth of the optimal solution, while the space complexity is O(b^d) due to the need to store all nodes in memory. Overall, the code provides a good example of how the A* algorithm can be used to solve search problems efficiently.

REFERENCES:

- https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288
- https://github.com/topics/n-puzzle

EXPERIMENT – 5

AIM: To implement Constraint Satisfaction problem using backtracking

DESCRIPTION:

Constraint Satisfaction Problem (CSP) is a problem in which the task is to assign values to variables in such a way that all the given constraints are satisfied. In other words, a CSP is a problem in which we have a set of variables, each with a domain of possible values, and a set of constraints that restrict the values that can be assigned to the variables.

Backtracking is an algorithmic technique that is often used to solve CSPs. It works by recursively exploring the search space of possible variable assignments and backtracking when a partial assignment violates one of the constraints.

EXPERIMENTAL RESULTS:

Program:

```
def solve_n_queens(n):
    board = [-1] * n
    return backtrack(board, 0)
def backtrack(board, row):
    if row == len(board):
        return board
    for col in range(len(board)):
        if is_valid(board, row, col):
            board[row] = col
            result = backtrack(board, row + 1)
            if result is not None:
                return result
            board[row] = -1
    return None
def is_valid(board, row, col):
    for r in range(row):
        if board[r] == col or abs(board[r] - col) == row - r:
            return False
    return True
def print_board(board):
    for row in board:
        line = ''
        for col in row:
            if col == -1:

                line += 'Q '
            else:
                line += '. '
        print(line)
    print()
n = 4
solution = solve_n_queens(n)
if solution is not None:
    board = [[-1 for i in range(n)] for j in range(n)]
    for i in range(n):
        board[i][solution[i]] = 1
    print_board(board)
else:
    print(f"No solution found for {n}-Queens problem.")
```

Output:

```
Q . Q Q
Q Q Q .
. Q Q Q
Q Q . Q
```

INFERENCES AND CONCLUSION:

ANALYSIS:

Time complexity

The time complexity of the **solve_n_queens** function is **O(N!)**, where **N** is the size of the board. This is because the function generates all possible permutations of **N** queens on the **N x N** board, which results in **N!** possible configurations. The **backtrack** function, which is called by **solve_n_queens**, has a worst-case time complexity of **O(N^N)**, but this worst case is never reached because the function exits early as soon as it finds a solution. On average, the **backtrack** function will only explore a fraction of the total number of possible configurations, which is closer to **O(N!)**.

Space complexity

The space complexity of the above code is O(n^2), where n is the size of the chessboard. This is because the solution is represented using a two-dimensional board of size n x n, which takes up O(n^2) space.

CONCLUSION:

We have successfully implemented the CSP problem using the backtracking.

EXPERIMENT – 6

AIM: To implement a program for Game search using Minimax algorithm

DESCRIPTION:

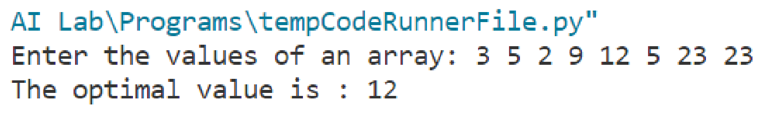
Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.
In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.
are unique for every type of game. Now, the below program is a Python program that implements the minimax algorithm for finding the maximum score a maximizing player can get in a two-player game. The program takes an array of scores as input and uses the minimax function to recursively evaluate the maximum score that can be obtained by the maximizing player. The minimax function works by exploring the game tree to find the optimal sequence of moves that maximizes the score for the maximizing player. The program then outputs the optimal score.

EXPERIMENTAL RESULTS:

Program:

```python
# A simple Python3 program to find
# maximum score that
# maximizing player can get
import math

def minimax (curDepth, nodeIndex,
            maxTurn, scores,
            targetDepth):

    # base case : targetDepth reached
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                    True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    True, scores, targetDepth))

scores = list(map(int,input("Enter the values of an array: ").split()))

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

Output:



```
AI Lab\Programs\tempCodeRunnerFile.py"
Enter the values of an array: 3 5 2 9 12 5 23 23
The optimal value is : 12
```

INFERENCES AND CONCLUSION:

ANALYSIS:

Time complexity

The time complexity of the above code is O(2^(n+1)), where n is the base-2 logarithm of the length of the input array. This is because the program performs a complete binary tree traversal with 2^(n+1) nodes, where each node is visited once. Since the program evaluates each node in the tree once, the time complexity of the algorithm is proportional to the total number of nodes in the tree. Therefore, the time complexity of the algorithm is exponential in the length of the input array, which makes it computationally expensive for large inputs.

Space complexity

The space complexity of the above code is O(n), where n is the base-2 logarithm of the length of the input array. This is because the program uses a recursive approach to evaluate the game tree, and each recursive call adds a new layer to the call stack. Since the depth of the game tree is n, the maximum number of function calls on the stack at any point in time is n. Therefore, the space complexity of the algorithm is proportional to the depth of the game tree, which is logarithmic in the length of the input array.

CONCLUSION:

The above Python code implements the minimax algorithm to find the maximum score that a maximizing player can achieve in a zero-sum two-player game, represented as a complete binary tree of scores. The code defines a recursive function, minimax(), that traverses the game tree, evaluating the scores of the maximizing and minimizing players at each node. The function returns the maximum score of the maximizing player when it reaches the maximum depth of the tree. The program takes an input array of scores and calculates the base-2 logarithm of its length to determine the maximum depth of the game tree. The output of the program is the optimal value that the maximizing player can achieve in the game.

EXPERIMENT – 7

AIM: To implement a program for Game search using Alpha Beta pruning algorithm

DESCRIPTION:

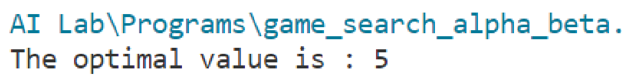
Alpha-beta pruning is an optimization technique used in game trees to reduce the number of nodes evaluated by the minimax algorithm. It works by maintaining two values, alpha and beta, that represent the minimum score the maximizing player is guaranteed and the maximum score the minimizing player is guaranteed, respectively. As the minimax algorithm traverses the game tree, it updates alpha and beta based on the best scores found so far. When a node's alpha value is greater than or equal to its parent's beta value, or vice versa, the algorithm prunes the subtree rooted at that node because it cannot affect the final outcome of the game. This results in a significant reduction in the number of nodes evaluated by the minimax algorithm, making it more efficient for larger game trees. This is a Python code for the implementation of the minimax algorithm with alpha-beta pruning. It finds the optimal value for the current player in a two-player game. The algorithm uses recursion to evaluate the possible moves and prunes the tree by cutting off branches that cannot affect the final decision. The input values represent the values of nodes in a tree. The code outputs the optimal value for the first player in the game.

EXPERIMENTAL RESULTS:

Program:

```python
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                            True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best



values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:



```
AI Lab\Programs\game_search_alpha_beta.
The optimal value is : 5
```

CONCLUSION:

The above code implements the minimax algorithm with alpha-beta pruning to find the optimal value for a two-player game represented by a binary tree. It takes in an input list of values representing the nodes of the tree and returns the maximum score that the maximizing player can get. The alpha-beta pruning technique is used to optimize the search process by eliminating the evaluation of irrelevant nodes.

EXPERIMENT – 8

AIM: To implement a Bayesian network from a given data and infer the data from that Bayesian network.

DESCRIPTION:

Bayesian Networks are used to model uncertainties by using Directed Acyclic Graphs (DAG). A Directed Acyclic Graph is used to represent a Bayesian Network and like any other statistical graph, a DAG contains a set of nodes and links, where the links denote the relationship between the nodes. A DAG models the uncertainty of an event occurring based on the Conditional Probability Distribution (CDP) of each random variable. A Conditional Probability Table (CPT) is used to represent the CPD of each variable in the network

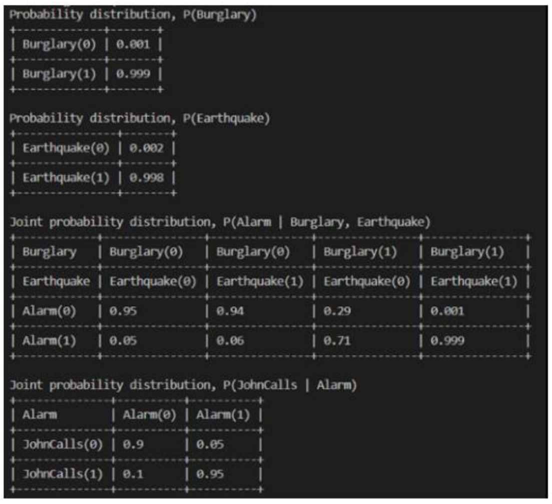
EXPERIMENTAL RESULTS:

Program:

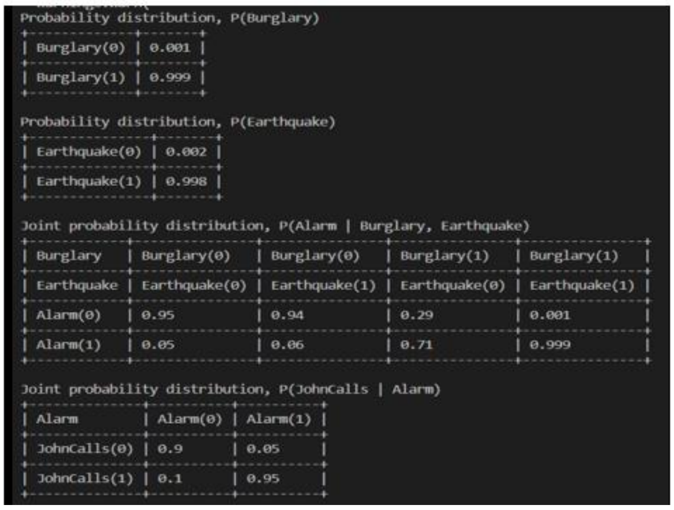
```
import pgmpy.models
import pgmpy.inference
import networkx as nx
import pylab as plt

# Create a bayesian network
model = pgmpy.models.BayesianModel([('Burglary', 'Alarm'),
                                    ('Earthquake', 'Alarm'),
                                    ('Alarm', 'JohnCalls'),
                                    ('Alarm', 'MaryCalls')])

cpd_burglary = pgmpy.factors.discrete.TabularCPD('Burglary', 2, [[0.001], [0.999]])
cpd_earthquake = pgmpy.factors.discrete.TabularCPD('Earthquake', 2, [[0.002], [0.998]])
cpd_alarm = pgmpy.factors.discrete.TabularCPD('Alarm', 2, [[0.95, 0.94, 0.29, 0.001],
[0.05, 0.06, 0.71, 0.999]], evidence=['Burglary',Earthquake'], evidence_card=[2, 2])
cpd_john = pgmpy.factors.discrete.TabularCPD('JohnCalls', 2, [[0.90, 0.05],
                                                           [0.10, 0.95]],
                                              evidence=['Alarm'],
                                              evidence_card=[2])
cpd_mary = pgmpy.factors.discrete.TabularCPD('MaryCalls', 2, [[0.70, 0.01],
                                                           [0.30, 0.99]],
                                              evidence=['Alarm'],
                                              evidence_card=[2])
model.add_cpds(cpd_burglary, cpd_earthquake, cpd_alarm, cpd_john, cpd_mary)
model.check_model()
# Print probability distributions
print('Probability distribution, P(Burglary)')
print(cpd_burglary)
print()
print('Probability distribution, P(Earthquake)')
print(cpd_earthquake)
print()
print('Joint probability distribution, P(Alarm | Burglary, Earthquake)')
print(cpd_alarm)
print()
print('Joint probability distribution, P(JohnCalls | Alarm)')
print(cpd_john)
```

```python
print()
print('Joint probability distribution, P(MaryCalls | Alarm)')
print(cpd_mary)
print()
infer = pgmpy.inference.VariableElimination(model)
# Calculate the probability of a burglary if John and Mary calls (0: True, 1: False)
posterior_probability = infer.query(['Burglary'], evidence={'JohnCalls': 0, 'MaryCalls':
0})
# Print posterior probability
print('Posterior probability of Burglary if JohnCalls(True) and MaryCalls(True)')
print(posterior_probability)
print()


# Calculate the probability of alarm starting if there is a burglary and an earthquake (0:
True, 1: False)
posterior_probability = infer.query(['Alarm'], evidence={'Burglary': 0, 'Earthquake': 0})
# Print posterior probability
print('Posterior probability of Alarm sounding if Burglary(True) and Earthquake(True)')
print(posterior_probability)
```

Output:



```
Probability distribution, P(Burglary)
+-------------+-------+
| Burglary(0) | 0.001 |
+-------------+-------+
| Burglary(1) | 0.999 |
+-------------+-------+

Probability distribution, P(Earthquake)
+---------------+-------+
| Earthquake(0) | 0.002 |
+---------------+-------+
| Earthquake(1) | 0.998 |
+---------------+-------+

Joint probability distribution, P(Alarm | Burglary, Earthquake)
+------------+---------------+---------------+---------------+---------------+
| Burglary   | Burglary(0)   | Burglary(0)   | Burglary(1)   | Burglary(1)   |
+------------+---------------+---------------+---------------+---------------+
| Earthquake | Earthquake(0) | Earthquake(1) | Earthquake(0) | Earthquake(1) |
+------------+---------------+---------------+---------------+---------------+
| Alarm(0)   | 0.95          | 0.94          | 0.29          | 0.001         |
+------------+---------------+---------------+---------------+---------------+
| Alarm(1)   | 0.05          | 0.06          | 0.71          | 0.999         |
+------------+---------------+---------------+---------------+---------------+

Joint probability distribution, P(JohnCalls | Alarm)
+--------------+----------+----------+
| Alarm        | Alarm(0) | Alarm(1) |
+--------------+----------+----------+
| JohnCalls(0) | 0.9      | 0.05     |
+--------------+----------+----------+
| JohnCalls(1) | 0.1      | 0.95     |
+--------------+----------+----------+
```



```
Probability distribution, P(Burglary)
+-------------+-------+
| Burglary(0) | 0.001 |
+-------------+-------+
| Burglary(1) | 0.999 |
+-------------+-------+

Probability distribution, P(Earthquake)
+---------------+-------+
| Earthquake(0) | 0.002 |
+---------------+-------+
| Earthquake(1) | 0.998 |
+---------------+-------+

Joint probability distribution, P(Alarm | Burglary, Earthquake)
+------------+---------------+---------------+---------------+---------------+
| Burglary   | Burglary(0)   | Burglary(0)   | Burglary(1)   | Burglary(1)   |
+------------+---------------+---------------+---------------+---------------+
| Earthquake | Earthquake(0) | Earthquake(1) | Earthquake(0) | Earthquake(1) |
+------------+---------------+---------------+---------------+---------------+
| Alarm(0)   | 0.95          | 0.94          | 0.29          | 0.001         |
+------------+---------------+---------------+---------------+---------------+
| Alarm(1)   | 0.05          | 0.06          | 0.71          | 0.999         |
+------------+---------------+---------------+---------------+---------------+

Joint probability distribution, P(JohnCalls | Alarm)
+--------------+----------+----------+
| Alarm        | Alarm(0) | Alarm(1) |
+--------------+----------+----------+
| JohnCalls(0) | 0.9      | 0.05     |
+--------------+----------+----------+
| JohnCalls(1) | 0.1      | 0.95     |
+--------------+----------+----------+
```

CONCLUSION:

The above code demonstrates how to create a Bayesian network using the pgmpy library in Python. It defines a Bayesian network with 5 variables: Burglary, Earthquake, Alarm, JohnCalls, and MaryCalls, and adds Conditional Probability Distributions (CPDs) to each of them. The code then checks if the model is valid and prints the probability distributions of each variable. The code also uses Variable Elimination to calculate the posterior probability of Burglary if John and Mary both call and the posterior probability of the Alarm sounding if there is a Burglary and an Earthquake. Finally, the code prints the posterior probabilities for both cases.

EXPERIMENT – 9

AIM: To implement a MDP to run value iteration

DESCRIPTION:

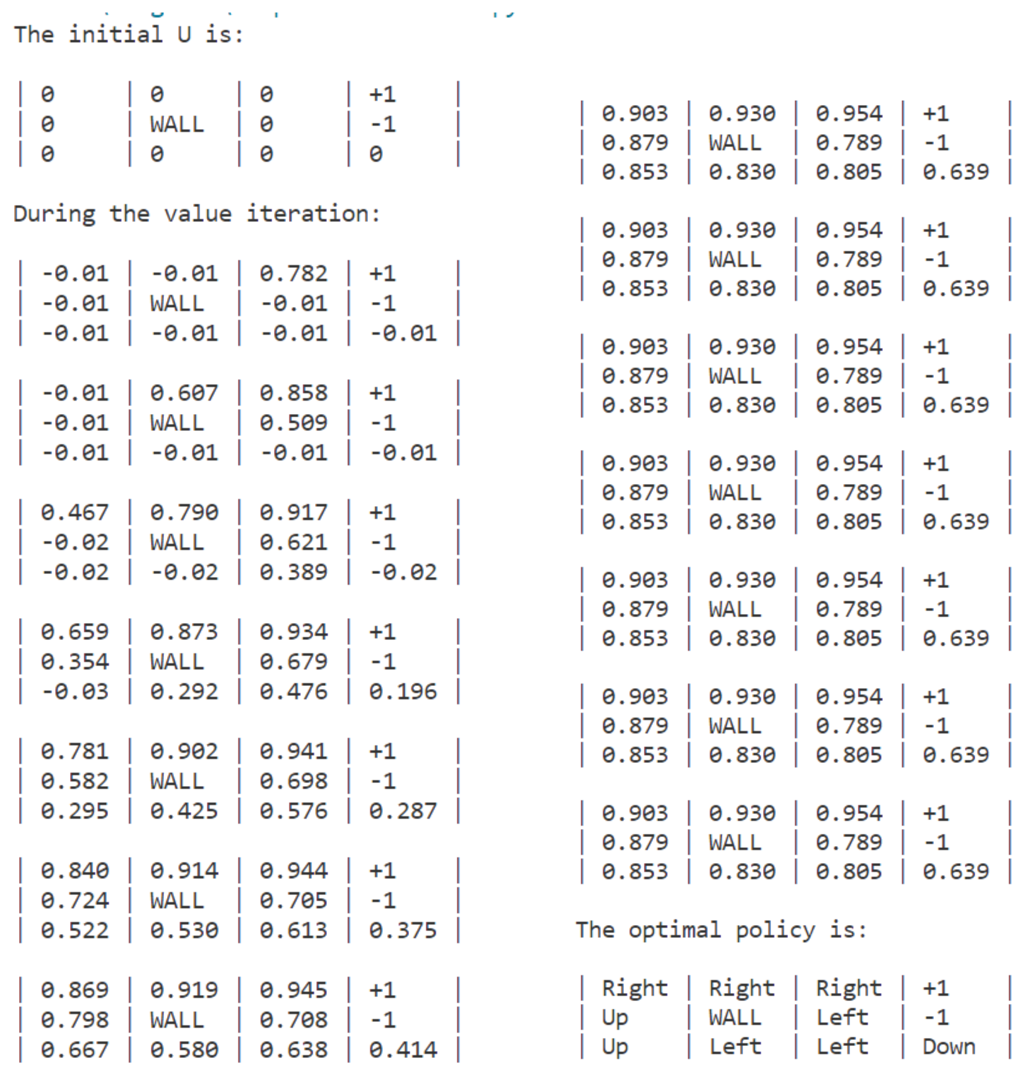
A Markov Decision Process (MDP) is a mathematical framework used to model decision-making problems in situations where outcomes are partly random and partly under the control of a decision-maker. The goal of MDP is to find the optimal policy for a given environment, which is a mapping from states to actions that maximizes the expected total reward received by the agent. Value iteration is an algorithm used to solve MDPs by iteratively computing the optimal values of each state. The algorithm starts with an initial guess for the value of each state and repeatedly updates them until convergence. At each iteration, the algorithm applies the Bellman update equation, which expresses the value of a state as the sum of the immediate reward and the discounted value of the next state. The discount factor is a parameter that determines the relative importance of immediate and future rewards.

EXPERIMENTAL RESULTS:

Program:

```
REWARD = -0.01
DISCOUNT = 0.99
MAX_ERROR = 10**(-3)
NUM_ACTIONS = 4
ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)]
NUM_ROW = 3
NUM_COL = 4
U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
def printEnvironment(arr, policy=False):
    res = ""
    for r in range(NUM_ROW):
        res += "|"
        for c in range(NUM_COL):
            if r == c == 1:
                val = "WALL"
            elif r <= 1 and c == 3:
                val = "+1" if r == 0 else "-1"
            else:
                if policy:
                    val = ["Down", "Left", "Up", "Right"][arr[r][c]]
                else:
                    val = str(arr[r][c])
            res += " " + val[:5].ljust(5) + " |"
        res += "\n"
    print(res)
def getU(U, r, c, action):
    dr, dc = ACTIONS[action]
    newR, newC = r+dr, c+dc
    if newR < 0 or newC < 0 or newR >= NUM_ROW or newC >= NUM_COL or (newR == newC == 1):
        return U[r][c]
    else:
        return U[newR][newC]
def calculateU(U, r, c, action):
    u = REWARD
    u += 0.1 * DISCOUNT * getU(U, r, c, (action-1)%4)
    u += 0.8 * DISCOUNT * getU(U, r, c, action)
    u += 0.1 * DISCOUNT * getU(U, r, c, (action+1)%4)
    return u
def valueIteration(U):
    print("During the value iteration:\n")
    while True:
        nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
        error = 0
        for r in range(NUM_ROW):
            for c in range(NUM_COL):
                if (r <= 1 and c == 3) or (r == c == 1):
                    continue
                nextU[r][c] = max([calculateU(U, r, c, action) for action in 
range(NUM_ACTIONS)])
                error = max(error, abs(nextU[r][c]-U[r][c]))
        U = nextU
        printEnvironment(U)
        if error < MAX_ERROR * (1-DISCOUNT) / DISCOUNT:
            break
    return U
def getOptimalPolicy(U):
    policy = [[-1, -1, -1, -1] for i in range(NUM_ROW)]
    for r in range(NUM_ROW):
        for c in range(NUM_COL):
            if (r <= 1 and c == 3) or (r == c == 1):
                continue
            maxAction, maxU = None, -float("inf")
            for action in range(NUM_ACTIONS):
                u = calculateU(U, r, c, action)
                if u > maxU:
                    maxAction, maxU = action, u
            policy[r][c] = maxAction
    return policy
print("The initial U is:\n")
printEnvironment(U)
U = valueIteration(U)
policy = getOptimalPolicy(U)
print("The optimal policy is:\n")
printEnvironment(policy, True)
```

Output:



```
The initial U is:

| 0     | 0     | 0     | +1    |
| 0     | WALL  | 0     | -1    |
| 0     | 0     | 0     | 0     |

During the value iteration:

| -0.01 | -0.01 | 0.782 | +1    |
| -0.01 | WALL  | -0.01 | -1    |
| -0.01 | -0.01 | -0.01 | -0.01 |

| -0.01 | 0.607 | 0.858 | +1    |
| -0.01 | WALL  | 0.509 | -1    |
| -0.01 | -0.01 | -0.01 | -0.01 |

| 0.467 | 0.790 | 0.917 | +1    |
| -0.02 | WALL  | 0.621 | -1    |
| -0.02 | -0.02 | 0.389 | -0.02 |

| 0.659 | 0.873 | 0.934 | +1    |
| 0.354 | WALL  | 0.679 | -1    |
| -0.03 | 0.292 | 0.476 | 0.196 |

| 0.781 | 0.902 | 0.941 | +1    |
| 0.582 | WALL  | 0.698 | -1    |
| 0.295 | 0.425 | 0.576 | 0.287 |

| 0.840 | 0.914 | 0.944 | +1    |
| 0.724 | WALL  | 0.705 | -1    |
| 0.522 | 0.530 | 0.613 | 0.375 |

| 0.869 | 0.919 | 0.945 | +1    |
| 0.798 | WALL  | 0.708 | -1    |
| 0.667 | 0.580 | 0.638 | 0.414 |

| 0.903 | 0.930 | 0.954 | +1    |
| 0.879 | WALL  | 0.789 | -1    |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1    |
| 0.879 | WALL  | 0.789 | -1    |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1    |
| 0.879 | WALL  | 0.789 | -1    |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1    |
| 0.879 | WALL  | 0.789 | -1    |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1    |
| 0.879 | WALL  | 0.789 | -1    |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1    |
| 0.879 | WALL  | 0.789 | -1    |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1    |
| 0.879 | WALL  | 0.789 | -1    |
| 0.853 | 0.830 | 0.805 | 0.639 |

The optimal policy is:

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |
```

CONCLUSION:

The above code implements the Value Iteration algorithm to solve a simple grid world problem. The grid world has 3 rows and 4 columns with two special states where the agent receives a reward of +1 or -1 respectively. There is also a wall in the middle of the grid that the agent cannot pass through. The objective of the agent is to find the optimal policy that maximizes the expected cumulative reward over time. The Value Iteration algorithm is implemented by first initializing the utility values for each state to zero. Then the algorithm repeatedly updates the utility values for each state based on the Bellman equation until the changes in the utility values become sufficiently small. Finally, the optimal policy is obtained by selecting the action with the highest expected utility value for each state.

EXPERIMENT – 10

AIM: To implement a MDP to run policy iteration

DESCRIPTION:

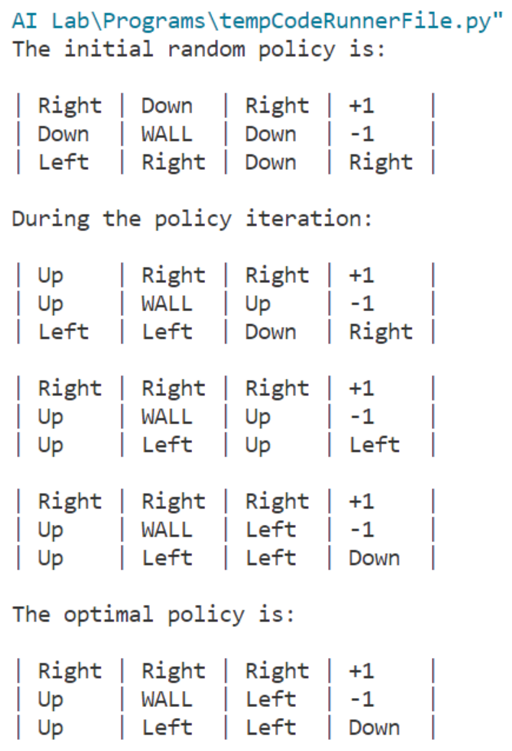
A Markov Decision Process (MDP) is a mathematical framework used to model decision-making problems in situations where outcomes are partly random and partly under the control of a decision-maker. The goal of MDP is to find the optimal policy for a given environment, which is a mapping from states to actions that maximizes the expected total reward received by the agent. Policy iteration is a reinforcement learning algorithm that iteratively improves a policy until it converges to the optimal policy. It involves two main steps: policy evaluation and policy improvement. In policy evaluation, the algorithm iteratively updates the state-value function of the policy until it converges. The state-value function is the expected total reward that can be obtained starting from a given state and following the policy. This step involves solving a system of linear equations, and it can be done using techniques such as value iteration or dynamic programming.

EXPERIMENTAL RESULTS:

Program:

```
import random
REWARD = -0.01
DISCOUNT = 0.99
MAX_ERROR = 10**(-3)
NUM_ACTIONS = 4
ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)]
NUM_ROW = 3
NUM_COL = 4
U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
policy = [[random.randint(0, 3) for j in range(NUM_COL)] for i in range(NUM_ROW)]
def printEnvironment(arr, policy=False):
    res = ""
    for r in range(NUM_ROW):
        res += "|"
        for c in range(NUM_COL):
            if r == c == 1:
                val = "WALL"
            elif r <= 1 and c == 3:
                val = "+1" if r == 0 else "-1"
            else:
                val = ["Down", "Left", "Up", "Right"][arr[r][c]]
            res += " " + val[:5].ljust(5) + " |"
        res += "\n"
    print(res)
def getU(U, r, c, action):
    dr, dc = ACTIONS[action]
    newR, newC = r+dr, c+dc
    if newR < 0 or newC < 0 or newR >= NUM_ROW or newC >= NUM_COL or (newR == newC == 1):
        return U[r][c]
    else:
        return U[newR][newC]
def calculateU(U, r, c, action):
    u = REWARD
    u += 0.1 * DISCOUNT * getU(U, r, c, (action-1)%4)
    u += 0.8 * DISCOUNT * getU(U, r, c, action)
    u += 0.1 * DISCOUNT * getU(U, r, c, (action+1)%4)
    return u
def policyEvaluation(policy, U):
    while True:
        nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
        error = 0
        for r in range(NUM_ROW):
            for c in range(NUM_COL):
                if (r <= 1 and c == 3) or (r == c == 1):
                    continue
                nextU[r][c] = calculateU(U, r, c, policy[r][c])
                error = max(error, abs(nextU[r][c]-U[r][c]))
        U = nextU
        if error < MAX_ERROR * (1-DISCOUNT) / DISCOUNT:
            break
    return U
def policyIteration(policy, U):
    print("During the policy iteration:\n")
    while True:
        U = policyEvaluation(policy, U)
        unchanged = True
        for r in range(NUM_ROW):
            for c in range(NUM_COL):
                if (r <= 1 and c == 3) or (r == c == 1):
                    continue
                maxAction, maxU = None, -float("inf")
                for action in range(NUM_ACTIONS):
                    u = calculateU(U, r, c, action)
                    if u > maxU:
                        maxAction, maxU = action, u
                if maxU > calculateU(U, r, c, policy[r][c]):
                    policy[r][c] = maxAction
                    unchanged = False
        if unchanged:
            break
        printEnvironment(policy)
    return policy
print("The initial random policy is:\n")
printEnvironment(policy)
policy = policyIteration(policy, U)
print("The optimal policy is:\n")
printEnvironment(policy)
```

Output:



```
AI Lab\Programs\tempCodeRunnerFile.py"
The initial random policy is:

| Right | Down  | Right | +1    |
| Down  | WALL  | Down  | -1    |
| Left  | Right | Down  | Right |

During the policy iteration:

| Up    | Right | Right | +1    |
| Up    | WALL  | Up    | -1    |
| Left  | Left  | Down  | Right |

| Right | Right | Right | +1    |
| Up    | WALL  | Up    | -1    |
| Up    | Left  | Up    | Left  |

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |

The optimal policy is:

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |
```

CONCLUSION:

This code implements the policy iteration algorithm to find the optimal policy for a simple grid-world problem. The grid-world problem is defined as a 3x4 grid, with a start state at the top-left corner and a goal state and a penalty state at the top-right and bottom-right corners, respectively. The objective is to find the optimal policy that maximizes the expected reward, while avoiding the penalty state. The code initializes the utility matrix U and a random policy. It then defines functions to calculate the state value for a given action, to perform policy evaluation, and to perform policy improvement. It then runs the policy iteration algorithm until convergence and outputs the optimal policy. Overall, this code demonstrates a simple implementation of the policy iteration algorithm for solving a basic grid-world problem.