Experiment No.—1: A. Implement the Breadth First Search algorithm to solve a given problem

Aim: The aim is to implement the Breadth First Search (BFS) algorithm to explore a graph or tree data structure, demonstrating how BFS systematically visits nodes in layers, ensuring the shortest path in an unweighted graph.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the BFS algorithm and its applications.
2. Implement BFS for exploring graphs or tree structures.
3. Analyze the time and space complexity of the BFS algorithm.

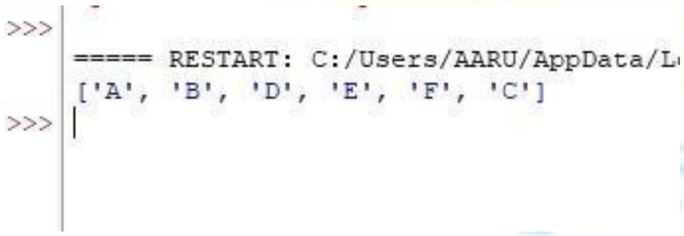
Theory:

Breadth First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or an arbitrary node in a graph) and explores all of the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. BFS is often used to find the shortest path in an unweighted graph. The time complexity of BFS is (O(V + E)), where (V) is the number of vertices and (E) is the number of edges.

Code:

```
graph1 = {
 'A': set(['B', 'C']),
 'B': set(['A', 'D', 'E']),
 'C': set(['A', 'F']),
 'D': set(['B']),
 'E': set(['B', 'F']),
 'F': set(['C', 'E'])
 }
def dfs(graph, node, visited):
 if node not in visited:
 visited.append(node)
 for n in graph[node]:
 dfs(graph,n, visited)
return visited
visited = dfs(graph1,'A', [])
print(visited)
```

Output:



```
>>>
    ===== RESTART: C:/Users/AARU/AppData/L
    ['A', 'B', 'D', 'E', 'F', 'C']
>>>
```

Learning Outcomes:

1. Successfully implemented the BFS algorithm to solve a given problem.

2. Gained insights into the systematic exploration of nodes in graph theory.

3. Understood the practical applications of BFS in various domains.

Course Outcomes:

1. Apply BFS for problem-solving in artificial intelligence and computer science.

2. Analyze the efficiency of BFS in different scenarios.

3. Understand the implications of BFS on memory consumption and performance.

Conclusion:

The experiment demonstrated the implementation of the Breadth First Search algorithm, highlighting its effectiveness in exploring graph structures and finding optimal paths. BFS is foundational in AI and networking applications.

Viva Questions:

1. What is the Breadth First Search algorithm?

2. How does BFS differ from Depth First Search (DFS)?

3. What are the applications of BFS?

4. Explain the time complexity of the BFS algorithm.

5. How is BFS implemented in programming?

Experiment No.—1: B. Implement the Iterative Depth First Search algorithm to solve the same problem

Aim: The aim is to implement the Iterative Depth First Search (DFS) algorithm to explore a graph or tree data structure, demonstrating how DFS uses a stack to traverse nodes in depth-first order.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the DFS algorithm and its iterative implementation.
2. Implement iterative DFS for exploring graphs or tree structures.
3. Analyze the time and space complexity of the DFS algorithm.

Theory:

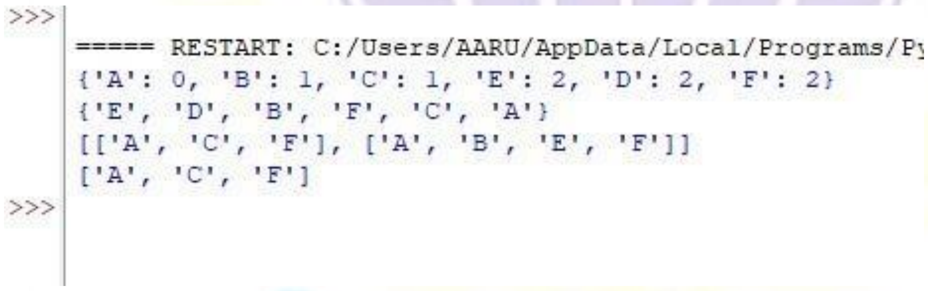
Depth First Search (DFS) is an algorithm used for traversing or searching tree or graph data structures. The algorithm starts at the root (or an arbitrary node) and explores as far as possible along each branch before backtracking. The iterative version of DFS uses a stack to keep track of nodes to visit next. The time complexity of DFS is also (O(V + E)).

Code:

```
# sample graph implemented as a dictionary

graph = {'A': set(['B', 'C']),

 'B': set(['A', 'D', 'E']),

 'C': set(['A', 'F']),

 'D': set(['B']),

 'E': set(['B', 'F']),
 'F': set(['C', 'E'])
 }
#Implement Logic of BFS
def bfs(start):
 queue = [start]
 levels={} #This Dict Keeps track of levels
 levels[start]=0 #Depth of start node is 0
 visited = set(start)
 while queue:
 node = queue.pop(0)
 neighbours=graph[node]
 for neighbor in neighbours:
if neighbor not in visited:
 queue.append(neighbor)
 visited.add(neighbor)
 levels[neighbor]= levels[node]+1
 print(levels) #print graph level
 return visited
print(str(bfs('A'))) #print graph node
#For Finding Breadth First Search Path
def bfs_paths(graph, start, goal):
 queue = [(start, [start])]
 while queue:
 (vertex, path) = queue.pop(0)
 for next in graph[vertex] - set(path):
 if next == goal:
```

```
 yield path + [next]
 else:
 queue.append((next, path + [next]))
result=list(bfs_paths(graph, 'A', 'F'))
print(result)# [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
#For finding shortest path
def shortest_path(graph, start, goal):
 try:
 return next(bfs_paths(graph, start, goal))
 except StopIteration:
 return None
result1=shortest_path(graph, 'A', 'F')
print(result1)# ['A', 'C', 'F']
```

Output:



```
>>>
===== RESTART: C:/Users/AARU/AppData/Local/Programs/Py
{'A': 0, 'B': 1, 'C': 1, 'E': 2, 'D': 2, 'F': 2}
{'E', 'D', 'B', 'F', 'C', 'A'}
[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
['A', 'C', 'F']
>>>
```

Learning Outcomes:

1. Successfully implemented the iterative DFS algorithm to solve a given problem.

2. Gained insights into the depth-first exploration of nodes in graph theory.

3. Understood the iterative approach to implementing DFS compared to recursion.

Course Outcomes:

1. Apply DFS for problem-solving in artificial intelligence and computer science.

2. Analyze the efficiency of DFS in various scenarios.

3. Understand the implications of using stacks for managing traversal state.

Conclusion:

The experiment demonstrated the iterative implementation of the Depth First Search algorithm, showcasing its effectiveness in exploring graph structures. DFS is vital in AI for tasks such as maze solving and pathfinding.

Viva Questions:

1. What is the Depth First Search algorithm?

2. How does the iterative DFS differ from the recursive approach?

3. What are the applications of DFS?

4. Explain the time complexity of the DFS algorithm.

5. How is DFS implemented using a stack?

Experiment No.—2: A. Write a program to simulate the 4-Queen / N-Queen problem

Aim: The aim is to implement a solution for the N-Queen problem, demonstrating how to place N queens on an N×N chessboard such that no two queens threaten each other.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the N-Queen problem and its constraints.
2. Implement backtracking techniques to solve the problem.
3. Analyze the effectiveness of various strategies for placing queens.

Theory:

The N-Queen problem involves placing N queens on an N×N chessboard so that no two queens are in the same row, column, or diagonal. This problem is a classic example of backtracking, where the algorithm explores all potential placements and retracts if a conflict arises. The solution space increases exponentially with N, making it a significant problem in algorithm design.

Code:

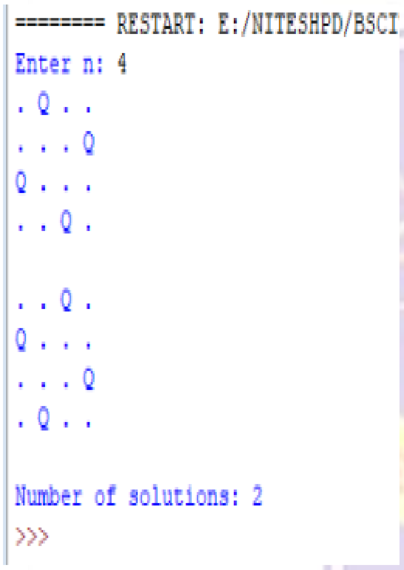
```
class QueenChessBoard:
 def __init__(self, size):
 # board has dimensions size x size
 self.size = size
 # columns[r] is a number c if a queen is placed at row r and column c.
 # columns[r] is out of range if no queen is place in row r.
 # Thus after all queens are placed, they will be at positions
 # (columns[0], 0), (columns[1], 1), ... (columns[size - 1], size - 1)
 self.columns = []
 def place_in_next_row(self, column):
 self.columns.append(column)
 def remove_in_current_row(self):
 return self.columns.pop()
 def is_this_column_safe_in_next_row(self, column):
 # index of next row
 row = len(self.columns)
 # check column
 for queen_column in self.columns:
 if column == queen_column:
 return False
 # check diagonal
 for queen_row, queen_column in enumerate(self.columns):
 if queen_column - queen_row == column - row:
 return False
 # check other diagonal
 for queen_row, queen_column in enumerate(self.columns):
 if ((self.size - queen_column) - queen_row
 == (self.size - column) - row):
 return False
 return True
 def display(self):
 for row in range(self.size):
 for column in range(self.size):
 if column == self.columns[row]:
 print('Q', end=' ')
 else:
 print('.', end=' ')
 print()
def solve_queen(size):
"""Display a chessboard for each possible configuration of placing n
queens on an n x n chessboard and print the number of such
configurations."""
 board = QueenChessBoard(size)
 number_of_solutions = 0
 row = 0
 column = 0
 # iterate over rows of board
 while True:
 # place queen in next row
 while column < size:
 if board.is_this_column_safe_in_next_row(column):
 board.place_in_next_row(column)
 row += 1
 column = 0
 break
 else:
 column += 1
 # if could not find column to place in or if board is full
```

```
if (column == size or row == size):
# if board is full, we have a solution
if row == size:
board.display()
print()
number_of_solutions += 1
# small optimization:
# In a board that already has queens placed in all rows except
# the last, we know there can only be at most one position in
# the last row where a queen can be placed. In this case, there
# is a valid position in the last row. Thus we can backtrack two
# times to reach the second last row.
board.remove_in_current_row()
row -= 1
# now backtrack
try:
prev_column = board.remove_in_current_row()
except IndexError:
# all queens removed
# thus no more possible configurations
break
# try previous row again
row -= 1
# start checking at column = (1 + value of column in previous row)
column = 1 + prev_column
print('Number of solutions:', number_of_solutions)
```

n = int(input('Enter n: '))

solve_queen(n)

Output:



```
======== RESTART: E:/NITESHPD/BSCI
Enter n: 4
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .

Number of solutions: 2
>>>
```

Learning Outcomes:

1. Successfully implemented a solution for the N-Queen problem.

2. Gained insights into backtracking as a problem-solving strategy.

3. Understood the complexity of combinatorial problems in AI.

Course Outcomes:

1. Apply backtracking algorithms to solve combinatorial problems.

2. Analyze the performance of different approaches to the N-Queen problem.

3. Understand the relevance of constraint satisfaction in AI.

Conclusion:

The experiment showcased the implementation of a solution to the N-Queen problem, emphasizing backtracking as an effective technique for exploring complex solution spaces. The N-Queen problem serves as a foundational example in artificial intelligence and optimization.

Viva Questions:

1. What is the N-Queen problem?

2. How does backtracking work in solving the N-Queen problem?

3. What are the constraints involved in placing queens on the board?

4. Can you explain the time complexity of solving the N-Queen problem? 5. What are some real-world applications of the N-Queen problem

Experiment No.—2: B. Write a program to solve the Tower of Hanoi problem

Aim: The aim is to implement a solution for the Tower of Hanoi problem, demonstrating how to move a stack of disks from one peg to another while adhering to the rules of the game.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the rules and constraints of the Tower of Hanoi problem.
2. Implement a recursive solution to solve the problem.
3. Analyze the time complexity of the Tower of Hanoi solution.

Theory:

The Tower of Hanoi is a mathematical puzzle that involves three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks stacked in ascending order on one rod, and the goal is to move the entire stack to another rod, following these rules:

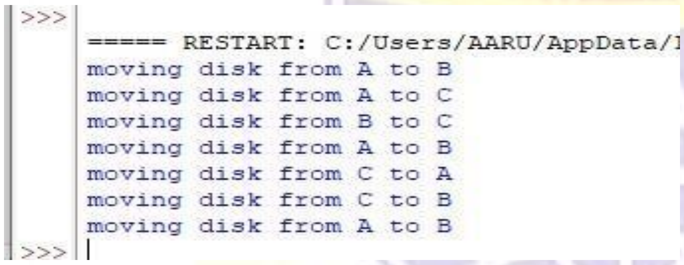
1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No larger disk may be placed on top of a smaller disk.

The time complexity for solving the Tower of Hanoi problem is (O(2^n)), where (n) is the number of disks.

Code:

```
def moveTower(height,fromPole, toPole, withPole):
 if height >= 1:
 moveTower(height-1,fromPole,withPole,toPole)
 moveDisk(fromPole,toPole)
 moveTower(height-1,withPole,toPole,fromPole)
def moveDisk(fp,tp):
 print("moving disk from",fp,"to",tp)
moveTower(3,"A","B","C")
```

Output:



```
>>>
===== RESTART: C:/Users/AARU/AppData/I
moving disk from A to B
moving disk from A to C
moving disk from B to C
moving disk from A to B
moving disk from C to A
moving disk from C to B
moving disk from A to B
>>> |
```

Learning Outcomes:

1. Successfully implemented a solution for the Tower of Hanoi problem.

2. Gained insights into recursive problem-solving techniques.

3. Understood the mathematical principles behind the puzzle.

Course Outcomes:

1. Apply recursive algorithms to solve classic problems.

2. Analyze the efficiency of recursive solutions in terms of time and space.

3. Understand the significance of the Tower of Hanoi in algorithm design.

Conclusion:

The experiment effectively demonstrated the implementation of a solution for the Tower of Hanoi problem, highlighting recursion as a powerful technique for solving complex problems. The Tower of Hanoi serves as a pedagogical tool for teaching recursion and algorithmic thinking.

Viva Questions:

1. What are the rules of the Tower of Hanoi problem?

2. How does the recursive solution for the Tower of Hanoi work?

3. Explain the time complexity for solving the Tower of Hanoi problem.

4. What are some applications of recursive algorithms?

5. Can you describe the iterative approach to solving the Tower of Hanoi?

Experiment No.—3: A. Write a program to implement Alpha-Beta Search

Aim: The aim is to implement the Alpha-Beta pruning technique, enhancing the efficiency of the Minimax algorithm used in decision-making and game-playing AI.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the principles of the Minimax algorithm.
2. Implement Alpha-Beta pruning to improve search efficiency.
3. Analyze the time complexity improvements achieved through Alpha-Beta pruning.

Theory:

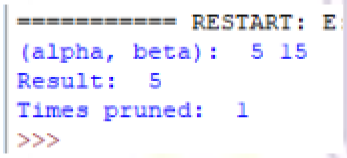
Alpha-Beta Search is an optimization technique for the Minimax algorithm, which seeks to minimize the possible loss in a worst-case scenario. It works by eliminating branches in the search tree that do not need to be evaluated because there is already a better move available. The Alpha-Beta algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively. The time complexity of the Alpha-Beta algorithm is (O(b^{d/2})) in the best case, where (b) is the branching factor and (d) is the depth of the tree.

Code:

```
tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]

root = 0

pruned = 0

def children(branch, depth, alpha, beta):
```

```
global tree
global root
global pruned
i = 0
for child in branch:
if type(child) is list:
(nalpha, nbeta) = children(child, depth + 1, alpha, beta)
if depth % 2 == 1:
beta = nalpha if nalpha < beta else beta
else:
alpha = nbeta if nbeta > alpha else alpha
branch[i] = alpha if depth % 2 == 0 else beta
i += 1
else:
if depth % 2 == 0 and alpha < child:
alpha = child
if depth % 2 == 1 and beta > child:
beta = child
if alpha >= beta:
pruned += 1
break
if depth == root:
tree = alpha if root == 0 else beta
return (alpha, beta)
def alphabeta(in_tree=tree, start=root, upper=-15, lower=15):
global tree
global pruned
global root
(alpha, beta) = children(tree, start, upper, lower)

if __name__ == "__main__":
print ("(alpha, beta): ", alpha, beta)
print ("Result: ", tree)
print ("Times pruned: ", pruned)
return (alpha, beta, tree, pruned)
if __name__ == "__main__":
alphabeta(None)
```

Output:



```
=========== RESTART: E
(alpha, beta):  5 15
Result:  5
Times pruned:  1
>>>
```

Learning Outcomes:

1. Successfully implemented the Alpha-Beta pruning algorithm.

2. Gained insights into game theory and decision-making processes.

3. Understood the performance improvements offered by Alpha-Beta pruning.

Course Outcomes:

1. Apply Alpha-Beta pruning in AI applications for efficient decision-making.

2. Analyze the impact of pruning techniques on algorithm performance.

3. Understand the practical implications of game theory in AI.

Conclusion:

The experiment successfully demonstrated the implementation of Alpha-Beta pruning, showcasing its effectiveness in optimizing decision-making processes in game AI. Alpha-Beta pruning is essential for enhancing the performance of search algorithms in competitive environments.

Viva Questions:

1. What is Alpha-Beta pruning?

2. How does Alpha-Beta pruning improve the Minimax algorithm?

3. What are the advantages of using Alpha-Beta pruning in AI?

4. Explain the significance of alpha and beta values in the algorithm.

5. What are some applications of Alpha-Beta pruning beyond game AI?

Experiment No.—3: B. Write a program for Hill Climbing problem

Aim:

The aim is to implement the Hill Climbing algorithm to solve optimization problems, demonstrating how the algorithm iteratively improves solutions by moving towards higher values.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the Hill Climbing algorithm and its mechanisms.
2. Implement the algorithm to find local optima in problem spaces.
3. Analyze the limitations and advantages of Hill Climbing.

Theory:

Hill Climbing is a mathematical optimization algorithm that belongs to the family of local search. It continuously moves in the direction of increasing elevation or value to find the peak of the mountain (i.e., the optimal solution). While simple to implement, Hill Climbing can be prone to getting stuck in local optima, where a better solution may exist at a different location. The algorithm's efficiency largely depends on the problem's landscape.

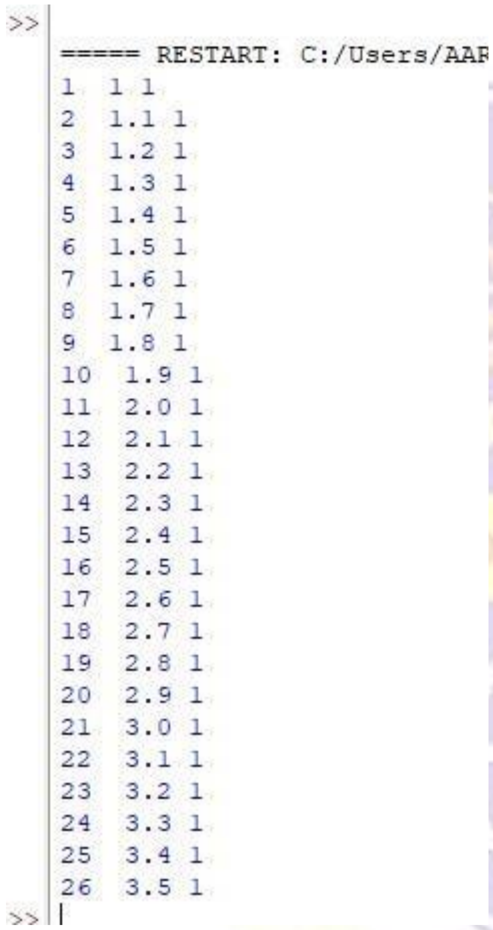
Code:

```
import math

increment = 0.1

startingPoint = [1, 1]

point1 = [1,5]
```

```
point2 = [6,4]
point3 = [5,2]
point4 = [2,1]
def distance(x1, y1, x2, y2):
 dist = math.pow(x2-x1, 2) + math.pow(y2-y1, 2)
 return dist
def sumOfDistances(x1, y1, px1, py1, px2, py2, px3, py3, px4, py4):
 d1 = distance(x1, y1, px1, py1)
 d2 = distance(x1, y1, px2, py2)
 d3 = distance(x1, y1, px3, py3)
 d4 = distance(x1, y1, px4, py4)
 return d1 + d2 + d3 + d4
def newDistance(x1, y1, point1, point2, point3, point4):
 d1 = [x1, y1]
 d1temp = sumOfDistances(x1, y1, point1[0],point1[1], point2[0],point2[1],
 point3[0],point3[1], point4[0],point4[1] )
 d1.append(d1temp)
 return d1
minDistance = sumOfDistances(startingPoint[0], startingPoint[1],
point1[0],point1[1], point2[0],point2[1],
 point3[0],point3[1], point4[0],point4[1] )
flag = True
def newPoints(minimum, d1, d2, d3, d4):
 if d1[2] == minimum:
 return [d1[0], d1[1]]
 elif d2[2] == minimum:
```

```
 return [d2[0], d2[1]]
 elif d3[2] == minimum:
 return [d3[0], d3[1]]
 elif d4[2] == minimum:
 return [d4[0], d4[1]]
i = 1
while flag:
 d1 = newDistance(startingPoint[0]+increment, startingPoint[1], point1, point2,
point3, point4)
 d2 = newDistance(startingPoint[0]-increment, startingPoint[1], point1, point2,
point3, point4)
 d3 = newDistance(startingPoint[0], startingPoint[1]+increment, point1, point2,
point3, point4)
 d4 = newDistance(startingPoint[0], startingPoint[1]-increment, point1, point2,
point3, point4)
 print (i,' ', round(startingPoint[0], 2), round(startingPoint[1], 2))
 minimum = min(d1[2], d2[2], d3[2], d4[2])
 if minimum < minDistance:
 startingPoint = newPoints(minimum, d1, d2, d3, d4)
 minDistance = minimum
 #print i,' ', round(startingPoint[0], 2), round(startingPoint[1], 2)
 i+=1
else:
 flag = False
```

Output:



```
>>
===== RESTART: C:/Users/AAR
1  1 1
2  1.1 1
3  1.2 1
4  1.3 1
5  1.4 1
6  1.5 1
7  1.6 1
8  1.7 1
9  1.8 1
10  1.9 1
11  2.0 1
12  2.1 1
13  2.2 1
14  2.3 1
15  2.4 1
16  2.5 1
17  2.6 1
18  2.7 1
19  2.8 1
20  2.9 1
21  3.0 1
22  3.1 1
23  3.2 1
24  3.3 1
25  3.4 1
26  3.5 1
>>
```

Learning Outcomes:

1. Successfully implemented the Hill Climbing algorithm to solve a problem.

2. Gained insights into optimization strategies and local search techniques.

3. Understood the trade-offs associated with Hill Climbing methods.

Course Outcomes:

1. Apply Hill Climbing techniques to optimization problems in AI.

2. Analyze the effectiveness of local search methods in various contexts.

3. Understand the implications of local maxima in algorithm design.

Conclusion:

The experiment effectively demonstrated the implementation of the Hill Climbing algorithm, showcasing its utility in solving optimization problems. Hill Climbing is a foundational technique in AI for solving complex decision-making tasks.

Viva Questions:

1. What is the Hill Climbing algorithm?

2. What are the advantages and disadvantages of Hill Climbing?

3. How does Hill Climbing differ from other optimization methods?

4. Explain the concept of local optima in Hill Climbing.

5. Can you describe a real-world application of the Hill Climbing algorithm?

Experiment No.—4: A. Write a program to implement A* algorithm

Aim:

The aim is to implement the A* algorithm to find the shortest path in a weighted graph, showcasing its effectiveness in pathfinding and graph traversal.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the A* algorithm and its heuristic-based approach.
2. Implement A* for finding optimal paths in graphs.
3. Analyze the time and space complexity of the A* algorithm.

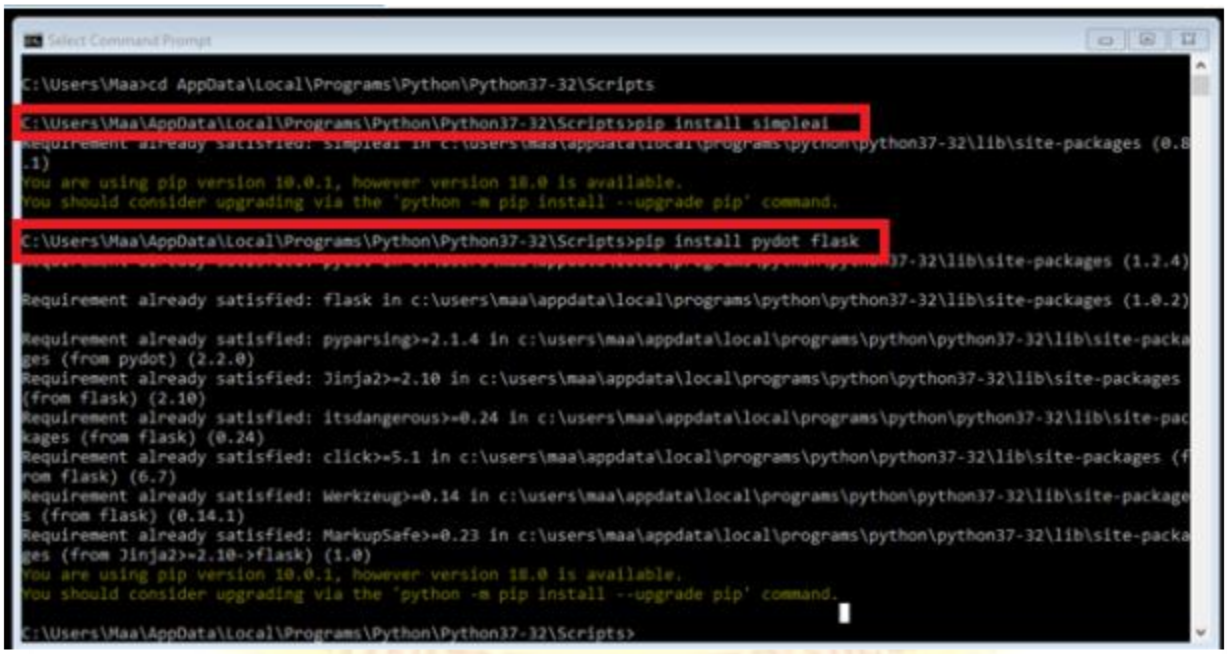
Theory:

A* is a pathfinding and graph traversal algorithm that is used to find the shortest path from a starting node to a target node in a weighted graph. It combines the advantages of Dijkstra's algorithm and the greedy best-first search. The algorithm uses a heuristic to estimate the cost from the current node to the goal, guiding the search efficiently. The time complexity of A* can vary but is generally (O(b^d)) where (b) is the branching factor and (d) is the depth of the solution.

Note:

Install 2 package in python scripts directory using pip command.

1. pip install simpleai
2. pip install pydot flask

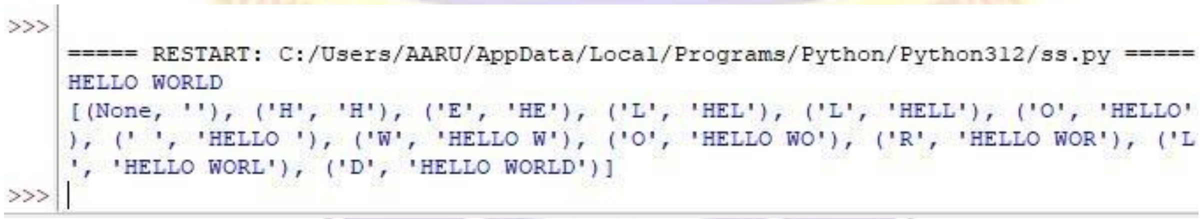


```
C:\Users\Maa>cd AppData\Local\Programs\Python\Python37-32\Scripts

C:\Users\Maa\AppData\Local\Programs\Python\Python37-32\Scripts>pip install simpleai
Requirement already satisfied: simpleai in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-packages (0.8
.1)
You are using pip version 10.0.1, however version 18.0 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\Maa\AppData\Local\Programs\Python\Python37-32\Scripts>pip install pydot flask
Requirement already satisfied: pydot in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-packages (1.2.4)

Requirement already satisfied: flask in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-packages (1.0.2)

Requirement already satisfied: pyparsing>=2.1.4 in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-packa
ges (from pydot) (2.2.0)
Requirement already satisfied: Jinja2>=2.10 in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-packages
(from flask) (2.10)
Requirement already satisfied: itsdangerous>=0.24 in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-pac
kages (from flask) (0.24)
Requirement already satisfied: click>=5.1 in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-packages (f
rom flask) (6.7)
Requirement already satisfied: Werkzeug>=0.14 in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-package
s (from flask) (0.14.1)
Requirement already satisfied: MarkupSafe>=0.23 in c:\users\maa\appdata\local\programs\python\python37-32\lib\site-packa
ges (from Jinja2>=2.10->flask) (1.0)
You are using pip version 10.0.1, however version 18.0 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\Maa\AppData\Local\Programs\Python\Python37-32\Scripts>
```

Code:

```
from simpleai.search import SearchProblem, astar
GOAL = 'HELLO WORLD'
class HelloProblem(SearchProblem):
 def actions(self, state):
 if len(state) < len(GOAL):
 return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
 else:
 return []
 def result(self, state, action):
 return state + action
def is_goal(self, state):
 return state == GOAL
 def heuristic(self, state):
 # how far are we from the goal?
 wrong = sum([1 if state[i] != GOAL[i] else 0
 for i in range(len(state))])
 missing = len(GOAL) - len(state)
 return wrong + missing
problem = HelloProblem(initial_state='')
result = astar(problem)
print(result.state)
print(result.path())
```

Output:



```
>>>
===== RESTART: C:/Users/AARU/AppData/Local/Programs/Python/Python312/ss.py =====
HELLO WORLD
[(None, ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'), ('O', 'HELLO'
), (' ', 'HELLO '), ('W', 'HELLO W'), ('O', 'HELLO WO'), ('R', 'HELLO WOR'), ('L
', 'HELLO WORL'), ('D', 'HELLO WORLD')]
>>>
```

Learning Outcomes:

1. Successfully implemented the A* algorithm for pathfinding.

2. Gained insights into heuristic-based search methods.

3. Understood the importance of pathfinding algorithms in AI.

Course Outcomes:

1. Apply A* for optimal pathfinding in AI applications.

2. Analyze the performance of heuristic search algorithms.

3. Understand the implications of graph theory in AI problem-solving.

Conclusion:

The experiment effectively demonstrated the implementation of the A* algorithm, highlighting its utility in finding optimal paths in complex networks. A* is a critical technique in AI for applications ranging from robotics to game development.

Viva Questions:

1. What is the A* algorithm?

2. How does A* use heuristics in its search?

3. Explain the difference between A* and Dijkstra's algorithm.

4. What are some applications of the A* algorithm?

5. Discuss the impact of heuristic choice on the performance of the A* algorithm.

Experiment No.— 4: B. Write a program to implement AO* algorithm

Aim:

The aim is to implement the AO* algorithm for solving problems involving AND/OR graphs, demonstrating how it optimizes the search for solutions.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the AO* algorithm and its application in AND/OR graphs.
2. Implement AO* for problem-solving in AI contexts.
3. Analyze the algorithm's efficiency in traversing complex structures.

Theory:

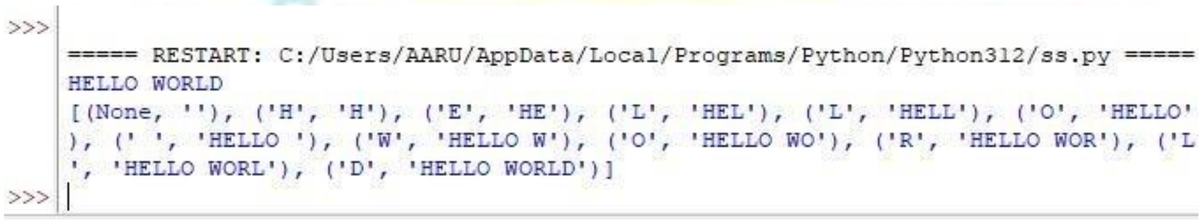
AO* is a search algorithm used for solving problems modeled as AND/OR graphs. Unlike traditional search algorithms that explore single paths, AO* allows for branching into multiple solutions and optimizes the search by considering the combined costs of AND nodes. This approach is beneficial for problems where multiple solutions must be evaluated simultaneously, such as planning and decision-making.

Code:

```
from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'

class HelloProblem(SearchProblem):

 def actions(self, state):
```

```
 if len(state) < len(GOAL):
 return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
 else:
 return []
 def result(self, state, action):
 return state + action
def is_goal(self, state):
 return state == GOAL
 def heuristic(self, state):
 # how far are we from the goal?
 wrong = sum([1 if state[i] != GOAL[i] else 0
 for i in range(len(state))])
 missing = len(GOAL) - len(state)
 return wrong + missing
problem = HelloProblem(initial_state='')
result = astar(problem)
print(result.state)
print(result.path())
```

Output:



```
>>>
===== RESTART: C:/Users/AARU/AppData/Local/Programs/Python/Python312/ss.py =====
HELLO WORLD
[(None, ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'), ('O', 'HELLO'
), (' ', 'HELLO '), ('W', 'HELLO W'), ('O', 'HELLO WO'), ('R', 'HELLO WOR'), ('L
', 'HELLO WORL'), ('D', 'HELLO WORLD')]
>>> |
```

Learning Outcomes:

1. Successfully implemented the AO* algorithm for problem-solving.

2. Gained insights into AND/OR graph structures in AI.

3. Understood the implications of multi-solution evaluation in search algorithms.

Course Outcomes:

1. Apply AO* in complex problem-solving scenarios in AI.

2. Analyze the performance of AO* compared to traditional search algorithms.

3. Understand the relevance of AND/OR graphs in AI applications.

Conclusion:

The experiment effectively demonstrated the implementation of the AO* algorithm, showcasing its advantages in solving problems represented by AND/OR graphs. AO* is a valuable tool in AI for optimizing decision-making processes.

Viva Questions:

1. What is the AO* algorithm?

2. How does AO* differ from A* and other search algorithms?

3. What are the applications of AO* in AI?

4. Explain the concept of AND/OR graphs.

5. Discuss the significance of cost evaluation in the AO* algorithm.

Experiment No.— 5: A. Write a program to solve the water jug problem

Aim: The aim is to implement a solution for the Water Jug problem, demonstrating how to use search techniques to find a sequence of actions to measure a specific quantity of water.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the Water Jug problem and its constraints.
2. Implement search techniques to solve the problem.
3. Analyze the effectiveness of different strategies for measuring water.

Theory:

The Water Jug problem is a classic problem in which two jugs with different capacities must be used to measure a specific amount of water. The challenge is to find a series of pouring actions that lead to the desired measurement. This problem can be solved using various search algorithms, including BFS, DFS, or even heuristic approaches, illustrating the principles of state space search in AI.

Code:

```
# 3 water jugs capacity -> (x,y,z) where x>y>z

# initial state (12,0,0)

# final state (6,6,0)

capacity = (12,8,5)

# Maximum capacities of 3 jugs -> x,y,z

x = capacity[0]
```

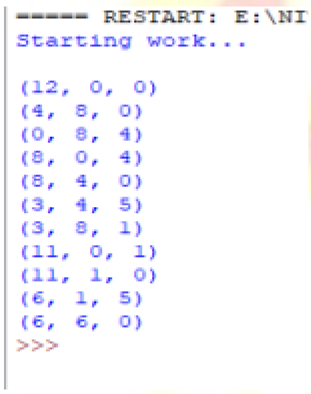
```
y = capacity[1]
z = capacity[2]
# to mark visited states
memory = {}
# store solution path
ans = []
def get_all_states(state):
 # Let the 3 jugs be called a,b,c
 a = state[0]
 b = state[1]
 c = state[2]
 if(a==6 and b==6):
 ans.append(state)
 return True
 # if current state is already visited earlier
 if((a,b,c) in memory):
 return False
 memory[(a,b,c)] = 1
 #empty jug a
 if(a>0):
 #empty a into b
 if(a+b<=y):
 if( get_all_states((0,a+b,c)) ):
 ans.append(state)
 return True
 else:
```

```
if( get_all_states((a-(y-b), y, c)) ):
ans.append(state)
return True
#empty a into c
if(a+c<=z):
if( get_all_states((0,b,a+c)) ):
ans.append(state)
return True
else:
if( get_all_states((a-(z-c), b, z)) ):
ans.append(state)
return True
#empty jug b
if(b>0):
#empty b into a
if(a+b<=x):
if( get_all_states((a+b, 0, c)) ):
ans.append(state)
return True
else:
if( get_all_states((x, b-(x-a), c)) ):
ans.append(state)
return True
#empty b into c
if(b+c<=z):
if( get_all_states((a, 0, b+c)) ):
```

```
ans.append(state)
return True
else:
if( get_all_states((a, b-(z-c), z)) ):
ans.append(state)
return True
#empty jug c
if(c>0):
#empty c into a
if(a+c<=x):
if( get_all_states((a+c, b, 0)) ):
ans.append(state)
return True
else:
if( get_all_states((x, b, c-(x-a))) ):
ans.append(state)
return True
#empty c into b
if(b+c<=y):
if( get_all_states((a, b+c, 0)) ):
ans.append(state)
return True
else:
if( get_all_states((a, y, c-(y-b))) ):
ans.append(state)
return True
```

```
 return False
initial_state = (12,0,0)
print("Starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
 print(i)
```

output:-



```
===== RESTART: E:\NI
Starting work...

(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
>>>
```

Learning Outcomes:

1. Successfully implemented a solution for the Water Jug problem.

2. Gained insights into state space search and problem-solving techniques.

3. Understood the relevance of the Water Jug problem in AI.

Course Outcomes:

1. Apply search algorithms to solve practical problems in AI.

2. Analyze the effectiveness of different problem-solving strategies.

3. Understand the implications of state space representation in AI.

Conclusion:

The experiment successfully demonstrated the implementation of a solution for the Water Jug problem, showcasing the application of search algorithms in practical scenarios. The Water Jug problem is a foundational example of problem-solving in artificial intelligence.

Viva Questions:

1. What is the Water Jug problem?

2. How can search algorithms be applied to solve it?

3. What are the constraints of the Water Jug problem?

4. Explain the state space representation in this context.

5. Can you describe a real-world application of the Water Jug problem?

Experiment No.— 5: B. Design the simulation of Tic-Tac-Toe game using Minimax algorithm

Aim:

The aim is to implement a Tic-Tac-Toe game using the Minimax algorithm, demonstrating how AI can be used to make optimal moves in a two-player game.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the Minimax algorithm in game theory.
2. Implement the algorithm to simulate a Tic-Tac-Toe game.

3. Analyze the performance of AI in competitive environments.

Theory:

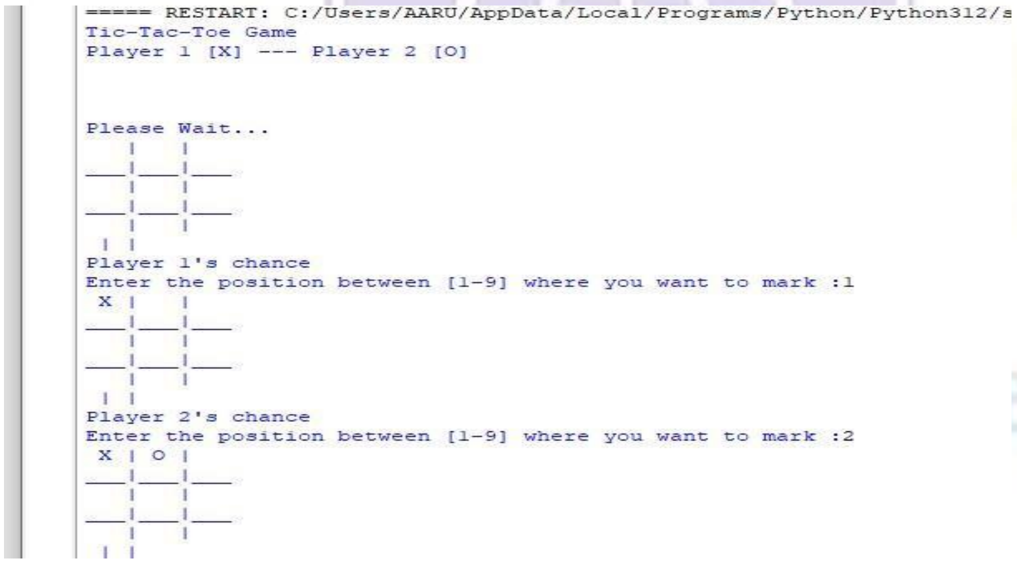
The Minimax algorithm is a decision-making algorithm used in game theory and artificial intelligence for minimizing the possible loss while maximizing the potential gain. It is widely used in two-player games like Tic-Tac-Toe. The algorithm evaluates the game tree, where each node represents a game state, and computes the optimal moves for both players. The time complexity of Minimax can be (O(b^d)), where (b) is the branching factor and (d) is the depth of the game tree.

Code:

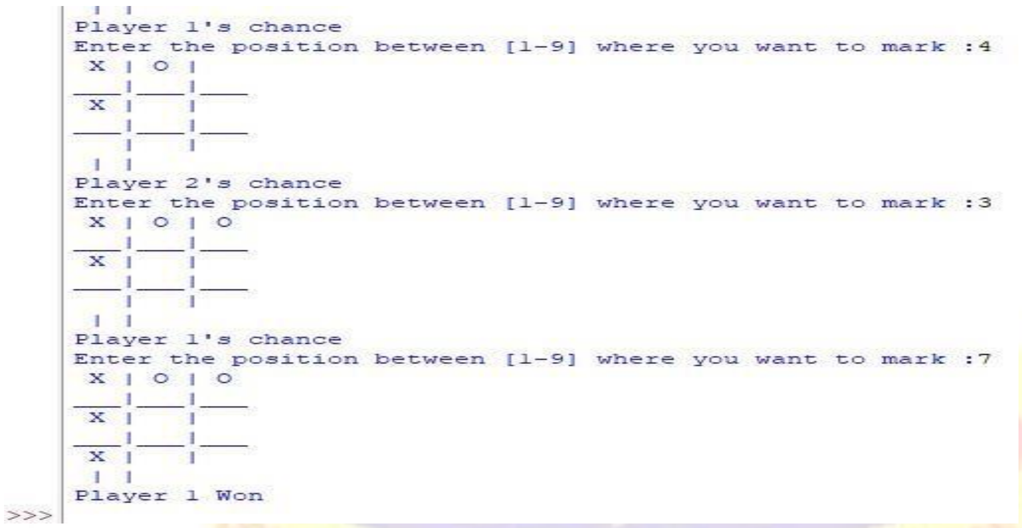
```
import os

import time
```

```
board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']
player = 1

########win Flags##########
Win = 1
Draw = -1
Running = 0
Stop = 1
############################
Game = Running
Mark = 'X'

#This Function Draws Game Board
def DrawBoard():
 print(" %c | %c | %c " % (board[1],board[2],board[3]))
 print("___|___|___")
 print(" %c | %c | %c " % (board[4],board[5],board[6]))
 print("___|___|___")
 print(" %c | %c | %c " % (board[7],board[8],board[9]))
 print(" | | ")

#This Function Checks position is empty or not
def CheckPosition(x):
 if(board[x] == ' '):
 return True
 else:
 return False

#This Function Checks player has won or not
def CheckWin():
 global Game
 #Horizontal winning condition
 if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):
 Game = Win
 elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):
 Game = Win
 elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):
 Game = Win
 #Vertical Winning Condition
 elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):
 Game = Win
 elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):
 Game = Win
 elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):
 Game=Win
 #Diagonal Winning Condition
 elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):
 Game = Win
 elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):
 Game=Win
 #Match Tie or Draw Condition
 elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and
board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):
Game=Draw
 else:
 Game=Running

print("Tic-Tac-Toe Game")
print("Player 1 [X] --- Player 2 [O]\n")
print()
print()
print("Please Wait...")
time.sleep(1)
while(Game == Running):
 os.system('cls')
 DrawBoard()
 if(player % 2 != 0):
 print("Player 1's chance")
 Mark = 'X'
 else:
 print("Player 2's chance")
 Mark = 'O'
 choice = int(input("Enter the position between [1-9] where you want to mark :
"))
 if(CheckPosition(choice)):
 board[choice] = Mark
 player+=1
 CheckWin()


os.system('cls')
DrawBoard()
if(Game==Draw):
 print("Game Draw")
elif(Game==Win):
 player-=1
 if(player%2!=0):
 print("Player 1 Won")
 else:
 print("Player 2 Won")
```

Output:



```
===== RESTART: C:/Users/AARU/AppData/Local/Programs/Python/Python312/s
Tic-Tac-Toe Game
Player 1 [X] --- Player 2 [O]



Please Wait...
    |   |
____|___|___
    |   |
____|___|___
    |   |
  | |
Player 1's chance
Enter the position between [1-9] where you want to mark :1
 X  |   |
____|___|___
    |   |
____|___|___
    |   |
  | |
Player 2's chance
Enter the position between [1-9] where you want to mark :2
 X  | O |
____|___|___
    |   |
____|___|___
    |   |
  | |
```



```
  |   |
Player 1's chance
Enter the position between [1-9] where you want to mark :4
 X  | O  |
____|____|____
 X  |    |
____|____|____
    |    |
  |   |
Player 2's chance
Enter the position between [1-9] where you want to mark :3
 X  | O  | O
____|____|____
 X  |    |
____|____|____
    |    |
  |   |
Player 1's chance
Enter the position between [1-9] where you want to mark :7
 X  | O  | O
____|____|____
 X  |    |
____|____|____
 X  |    |
  |   |
Player 1 Won
>>>
```

Learning Outcomes:

1. Successfully implemented a Tic-Tac-Toe game using the Minimax algorithm.
2. Gained insights into game theory and AI decision-making processes.
3. Understood the performance implications of Minimax in competitive games.

Course Outcomes:

1. Apply the Minimax algorithm in game development and AI applications.
2. Analyze the effectiveness of AI strategies in competitive environments.
3. Understand the principles of decision-making in two-player games.

Conclusion:

The experiment effectively demonstrated the implementation of a Tic-Tac-Toe game using the Minimax algorithm, showcasing its utility in optimizing gameplay. The Minimax algorithm is a fundamental concept in AI for decision-making in competitive scenarios.

Viva Questions:

1. What is the Minimax algorithm?
2. How does the Minimax algorithm work in a two-player game?
3. What are the advantages of using Minimax in AI?
4. Explain the concept of game trees in this context.
5. Can you describe a real-world application of the Minimax algorithm?

Experiment No.— 6: A. Write a program to solve the Missionaries and Cannibals problem

Aim:

The aim is to implement a solution for the Missionaries and Cannibals problem, demonstrating the use of search techniques to find a safe way to transport missionaries and cannibals across a river.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the Missionaries and Cannibals problem and its constraints.
2. Implement search techniques to solve the problem.
3. Analyze the effectiveness of different strategies for safe transportation.

Theory:

The Missionaries and Cannibals problem involves transporting a group of missionaries and cannibals across a river using a boat. The challenge arises from the condition that if at any time the cannibals outnumber the missionaries on either side of the river, the cannibals will eat the missionaries. This problem can be solved using search algorithms, including BFS or DFS, and serves as a classic example of constraint satisfaction problems in AI.

Code:

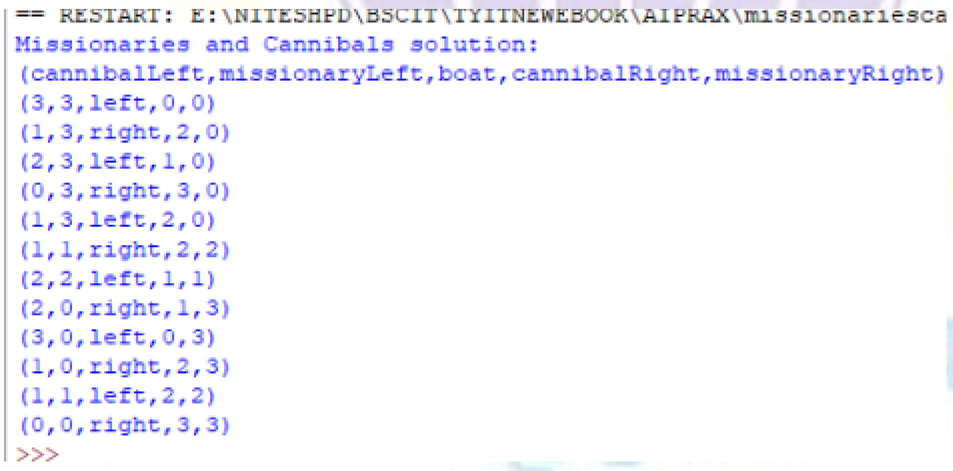
```
import math

# Missionaries and Cannibals Problem

class State():

def __init__(self, cannibalLeft, missionaryLeft, boat, cannibalRight,
missionaryRight):
self.cannibalLeft = cannibalLeft
self.missionaryLeft = missionaryLeft
self.boat = boat
self.cannibalRight = cannibalRight
self.missionaryRight = missionaryRight
self.parent = None
def is_goal(self):
if self.cannibalLeft == 0 and self.missionaryLeft == 0:
return True
else:
return False
def is_valid(self):
if self.missionaryLeft >= 0 and self.missionaryRight >= 0 \
 and self.cannibalLeft >= 0 and self.cannibalRight >= 0 \
 and (self.missionaryLeft == 0 or self.missionaryLeft >=
self.cannibalLeft) \
 and (self.missionaryRight == 0 or self.missionaryRight >=
self.cannibalRight):
return True
else:
return False
def __eq__(self, other):
return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft
== other.missionaryLeft \
 and self.boat == other.boat and self.cannibalRight ==
other.cannibalRight \
 and self.missionaryRight == other.missionaryRight
def __hash__(self):
return hash((self.cannibalLeft, self.missionaryLeft, self.boat,
self.cannibalRight, self.missionaryRight))
def successors(cur_state):
children = [];
if cur_state.boat == 'left':
new_state = State(cur_state.cannibalLeft, cur_state.missionaryLeft -
2, 'right',
 cur_state.cannibalRight, cur_state.missionaryRight + 2)
## Two missionaries cross left to right.
if new_state.is_valid():
new_state.parent = cur_state
children.append(new_state)
new_state = State(cur_state.cannibalLeft - 2,
cur_state.missionaryLeft, 'right',
 cur_state.cannibalRight + 2, cur_state.missionaryRight)
## Two cannibals cross left to right.
if new_state.is_valid():
new_state.parent = cur_state
children.append(new_state)
new_state = State(cur_state.cannibalLeft - 1, cur_state.missionaryLeft
- 1, 'right',
 cur_state.cannibalRight + 1, cur_state.missionaryRight + 1)
## One missionary and one cannibal cross left to right.
```

```
if new_state.is_valid():

new_state.parent = cur_state

children.append(new_state)

new_state = State(cur_state.cannibalLeft, cur_state.missionaryLeft -

1, 'right',

 cur_state.cannibalRight, cur_state.missionaryRight + 1)

## One missionary crosses left to right.

if new_state.is_valid():

new_state.parent = cur_state

children.append(new_state)

new_state = State(cur_state.cannibalLeft - 1,

cur_state.missionaryLeft, 'right',

 cur_state.cannibalRight + 1, cur_state.missionaryRight)

## One cannibal crosses left to right.

if new_state.is_valid():

new_state.parent = cur_state

children.append(new_state)

else:

new_state = State(cur_state.cannibalLeft, cur_state.missionaryLeft +

2, 'left',

 cur_state.cannibalRight, cur_state.missionaryRight - 2)

## Two missionaries cross right to left.

if new_state.is_valid():

new_state.parent = cur_state

children.append(new_state)

new_state = State(cur_state.cannibalLeft + 2,
cur_state.missionaryLeft, 'left',
 cur_state.cannibalRight - 2, cur_state.missionaryRight)
## Two cannibals cross right to left.
if new_state.is_valid():
new_state.parent = cur_state
children.append(new_state)
new_state = State(cur_state.cannibalLeft + 1, cur_state.missionaryLeft
+ 1, 'left',
 cur_state.cannibalRight - 1, cur_state.missionaryRight - 1)
## One missionary and one cannibal cross right to left.
if new_state.is_valid():
new_state.parent = cur_state
children.append(new_state)
new_state = State(cur_state.cannibalLeft, cur_state.missionaryLeft +
1, 'left',
 cur_state.cannibalRight, cur_state.missionaryRight - 1)
## One missionary crosses right to left.
if new_state.is_valid():
new_state.parent = cur_state
children.append(new_state)
new_state = State(cur_state.cannibalLeft + 1,
cur_state.missionaryLeft, 'left',
 cur_state.cannibalRight - 1, cur_state.missionaryRight)
## One cannibal crosses right to left.
if new_state.is_valid():
new_state.parent = cur_state
```

```
children.append(new_state)

return children

def breadth_first_search():

initial_state = State(3,3,'left',0,0)

if initial_state.is_goal():

return initial_state

frontier = list()

explored = set()

frontier.append(initial_state)

while frontier:

state = frontier.pop(0)

if state.is_goal():

return state

explored.add(state)

children = successors(state)

for child in children:

if (child not in explored) or (child not in frontier):

frontier.append(child)

return None

def print_solution(solution):

path = []

path.append(solution)

parent = solution.parent

while parent:

path.append(parent)

parent = parent.parent
```

```
for t in range(len(path)):
state = path[len(path) - t - 1]
print ("(" + str(state.cannibalLeft) + "," +
str(state.missionaryLeft) \
 + "," + state.boat + "," + str(state.cannibalRight) + "," + \
 str(state.missionaryRight) + ")")
def main():
solution = breadth_first_search()
print ("Missionaries and Cannibals solution:")
print ("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")
print_solution(solution)
# if called from the command line, call main()
if __name__ == "__main__":
 main()
```

Output:



```
== RESTART: E:\NITESHPD\BSCIT\TYITNEWEBOOK\AIPRAX\missionariesca
Missionaries and Cannibals solution:
(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)
(3,3,left,0,0)
(1,3,right,2,0)
(2,3,left,1,0)
(0,3,right,3,0)
(1,3,left,2,0)
(1,1,right,2,2)
(2,2,left,1,1)
(2,0,right,1,3)
(3,0,left,0,3)
(1,0,right,2,3)
(1,1,left,2,2)
(0,0,right,3,3)
>>>
```

Learning Outcomes:

1. Successfully implemented a solution for the Missionaries and Cannibals problem.
2. Gained insights into state space search and constraint satisfaction.
3. Understood the relevance of the problem in AI.

Course Outcomes:

1. Apply search algorithms to solve complex problems in AI.
2. Analyze the effectiveness of different strategies for constraint satisfaction.
3. Understand the implications of the Missionaries and Cannibals problem in AI applications.

Conclusion:

The experiment successfully demonstrated the implementation of a solution for the Missionaries and Cannibals problem, showcasing the application of search algorithms in practical scenarios. This problem is a foundational example of constraint satisfaction in artificial intelligence.

Viva Questions:

1. What is the Missionaries and Cannibals problem?
2. How can search algorithms be applied to solve it?
3. What are the constraints of the problem?
4. Explain the state space representation in this context.
5. Can you describe a real-world application of the Missionaries and Cannibals problem?

Experiment No.— 6: B. Design an application to simulate the number puzzle problem

Aim:

The aim is to design an application that simulates the Number Puzzle problem, demonstrating the use of search techniques to solve puzzles.

Tools Used:

1. Python 3.x

2. IDE or Text Editor

Learning Objectives:

1. Understand the Number Puzzle problem and its mechanics.

2. Implement search techniques to solve the puzzle.

3. Analyze the effectiveness of different strategies for puzzle-solving.

Theory:

The Number Puzzle problem involves arranging a set of numbers in a specific order on a grid. The challenge is to achieve the desired arrangement by sliding adjacent numbers into an empty space. This problem can be solved using various search algorithms, including BFS, DFS, or heuristic-based approaches, illustrating the principles of state space search in AI.

Code:

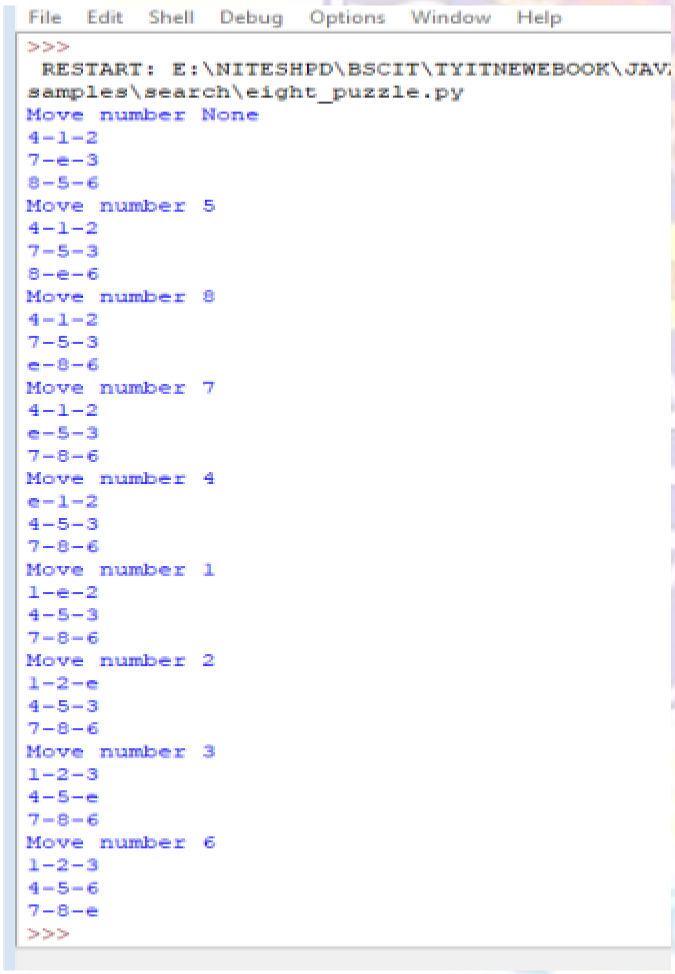
```
'''

8 puzzle problem, a smaller version of the fifteen puzzle:

States are defined as string representations of the pieces on the puzzle.

Actions denote what piece will be moved to the empty space.

States must allways be inmutable. We will use strings, but internally most of
```

```
the time we will convert those strings to lists, which are easier to handle.
For example, the state (string):
'1-2-3
4-5-6
7-8-e'
will become (in lists):
[['1', '2', '3'],
['4', '5', '6'],
['7', '8', 'e']]
'''
from __future__ import print_function
from simpleai.search import astar, SearchProblem
from simpleai.search.viewers import WebViewer
GOAL = '''1-2-3
4-5-6
7-8-e'''
INITIAL = '''4-1-2
7-e-3
8-5-6'''
def list_to_string(list_):
 return '\n'.join(['-'.join(row) for row in list_])
def string_to_list(string_):
 return [row.split('-') for row in string_.split('\n')]
def find_location(rows, element_to_find):
 '''Find the location of a piece in the puzzle.
 Returns a tuple: row, column'''
 for ir, row in enumerate(rows):
 for ic, element in enumerate(row):
 if element == element_to_find:
 return ir, ic
# we create a cache for the goal position of each piece, so we don't have to
# recalculate them every time
goal_positions = {}
rows_goal = string_to_list(GOAL)
for number in '12345678e':
 goal_positions[number] = find_location(rows_goal, number)
class EigthPuzzleProblem(SearchProblem):
 def actions(self, state):
 '''Returns a list of the pieces we can move to the empty space.'''
 rows = string_to_list(state)
 row_e, col_e = find_location(rows, 'e')
 actions = []
 if row_e > 0:
 actions.append(rows[row_e - 1][col_e])
 if row_e < 2:
 actions.append(rows[row_e + 1][col_e])
 if col_e > 0:
 actions.append(rows[row_e][col_e - 1])
 if col_e < 2:
 actions.append(rows[row_e][col_e + 1])
 return actions
 def result(self, state, action):
```

```
'''Return the resulting state after moving a piece to the empty space.
(the "action" parameter contains the piece to move)
rows = string_to_list(state)
row_e, col_e = find_location(rows, 'e')
row_n, col_n = find_location(rows, action)
rows[row_e][col_e], rows[row_n][col_n] = rows[row_n][col_n],
rows[row_e][col_e]
return list_to_string(rows)
def is_goal(self, state):
'''Returns true if a state is the goal state.'''
return state == GOAL
def cost(self, state1, action, state2):
'''Returns the cost of performing an action. No useful on this problem, i
but needed.
'''
return 1
def heuristic(self, state):
'''Returns an *estimation* of the distance from a state to the goal.
We are using the manhattan distance.
'''
rows = string_to_list(state)
distance = 0
for number in '12345678e':
row_n, col_n = find_location(rows, number)
row_n_goal, col_n_goal = goal_positions[number]
distance += abs(row_n - row_n_goal) + abs(col_n - col_n_goal)
```

```
 return distance
result = astar(EigthPuzzleProblem(INITIAL))
for action, state in result.path():
 print('Move number', action)
 print(state)
```

Output:



```
>>>
 RESTART: E:\NITESHPD\BSCIT\TYIITNEWEBOOK\JAV
samples\search\eight_puzzle.py
Move number None
4-1-2
7-e-3
8-5-6
Move number 5
4-1-2
7-5-3
8-e-6
Move number 8
4-1-2
7-5-3
e-8-6
Move number 7
4-1-2
e-5-3
7-8-6
Move number 4
e-1-2
4-5-3
7-8-6
Move number 1
1-e-2
4-5-3
7-8-6
Move number 2
1-2-e
4-5-3
7-8-6
Move number 3
1-2-3
4-5-e
7-8-6
Move number 6
1-2-3
4-5-6
7-8-e
>>>
```

Learning Outcomes:

1. Successfully implemented a solution for the Number Puzzle problem.

2. Gained insights into state space search and problem-solving techniques.

3. Understood the relevance of the Number Puzzle problem in AI.

Course Outcomes:

1. Apply search algorithms to solve practical problems in AI.

2. Analyze the effectiveness of different problem-solving strategies.

3. Understand the implications of state space representation in AI.

Conclusion:

The experiment successfully demonstrated the implementation of a solution for the Number Puzzle problem, showcasing the application of search algorithms in practical scenarios. The Number Puzzle problem is a foundational example of problem-solving in artificial intelligence.

Viva Questions:

1. What is the Number Puzzle problem?

2. How can search algorithms be applied to solve it?

3. What are the constraints of the problem?

4. Explain the state space representation in this context.

5. Can you describe a real-world application of the Number Puzzle problem?

Experiment No.— 7: A. Write a program to shuffle a deck of cards

Aim:

The aim is to implement a program that shuffles a deck of cards, demonstrating the use of randomization techniques in programming.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the mechanics of shuffling a deck of cards.
2. Implement randomization techniques in programming.
3. Analyze the effectiveness of different shuffling algorithms.

Theory:

Shuffling a deck of cards involves rearranging the order of the cards randomly to ensure that each card has an equal chance of appearing in any position. There are various algorithms for shuffling, including the Fisher-Yates shuffle, which is efficient and unbiased. This problem serves as a practical application of randomization in programming.

Code:

```
#first let's import random procedures since we will be shuffling

import random

#next, let's start building list holders so we can place our cards in there:

cardfaces = []

suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
```

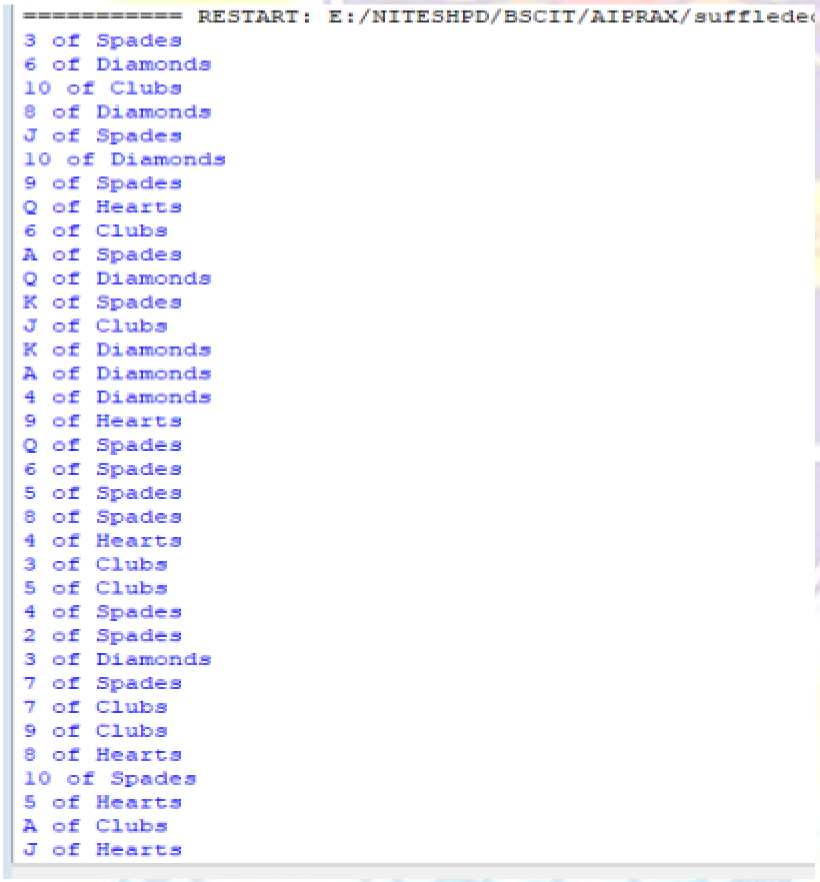
```
royals = ["J", "Q", "K", "A"]

deck = []

#now, let's start using loops to add our content:

for i in range(2,11):

 cardfaces.append(str(i)) #this adds numbers 2-10 and converts them to string

data

for j in range(4):

cardfaces.append(royals[j]) #this will add the royal faces to the cardbase

for k in range(4):

 for l in range(13):

 card = (cardfaces[l] + " of " + suits[k])

 #this makes each card, cycling through suits, but first through faces

 deck.append(card)

 #this adds the information to the "full deck" we want to make

#now let's shuffle our deck!

random.shuffle(deck)

#now let's see the cards!

for m in range(52):

 print(deck[m])
```

OR

```
# Python program to shuffle a deck of card using the module random and

draw 5 cards

# import modules

import itertools, random

# make a deck of cards

deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))
```

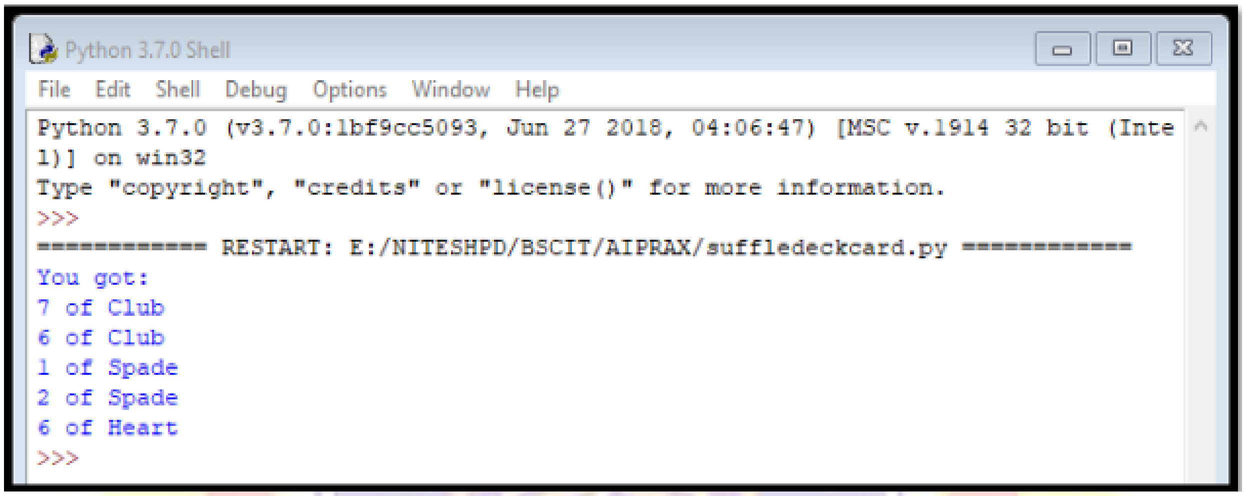
```
# shuffle the cards
random.shuffle(deck)
# draw five cards
print("You got:")
for i in range(5):
 print(deck[i][0], "of", deck[i][1])
```

Output:



```
=========== RESTART: E:/NITESHPD/BSCIT/AIPRAX/suffledec
3 of Spades
6 of Diamonds
10 of Clubs
8 of Diamonds
J of Spades
10 of Diamonds
9 of Spades
Q of Hearts
6 of Clubs
A of Spades
Q of Diamonds
K of Spades
J of Clubs
K of Diamonds
A of Diamonds
4 of Diamonds
9 of Hearts
Q of Spades
6 of Spades
5 of Spades
8 of Spades
4 of Hearts
3 of Clubs
5 of Clubs
4 of Spades
2 of Spades
3 of Diamonds
7 of Spades
7 of Clubs
9 of Clubs
8 of Hearts
10 of Spades
5 of Hearts
A of Clubs
J of Hearts
```

or



```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
============ RESTART: E:/NITESHPD/BSCIT/AIPRAX/suffledeckcard.py ============
You got:
7 of Club
6 of Club
1 of Spade
2 of Spade
6 of Heart
>>>
```

Learning Outcomes:

1. Successfully implemented a deck-shuffling program.

2. Gained insights into randomization techniques in programming.

3. Understood the importance of unbiased shuffling in card games.

Course Outcomes:

1. Apply randomization techniques in practical programming scenarios.

2. Analyze the effectiveness of different algorithms in shuffling.

3. Understand the implications of randomization in programming.

Conclusion:

The experiment successfully demonstrated the implementation of a deck-shuffling program, showcasing the application of randomization techniques in programming. Shuffling a deck of cards is a foundational example of randomization in computational problems.

Viva Questions:

1. What is the purpose of shuffling a deck of cards?

2. Describe the Fisher-Yates shuffle algorithm.

3. What are the implications of biased versus unbiased shuffling?

4. How can randomization techniques be applied in other programming scenarios?

5. Can you explain the importance of randomness in game design?

Experiment No.— 8: A. Solve the Blocks World Problem

Aim: The aim is to implement a solution for the Blocks World problem, demonstrating the use of planning algorithms to arrange blocks in a specific order.

Tools Used:

1. Python 3.x

2. IDE or Text Editor

Learning Objectives:

1. Understand the Blocks World problem and its mechanics.

2. Implement planning algorithms to solve the problem.

3. Analyze the effectiveness of different strategies for block arrangement.

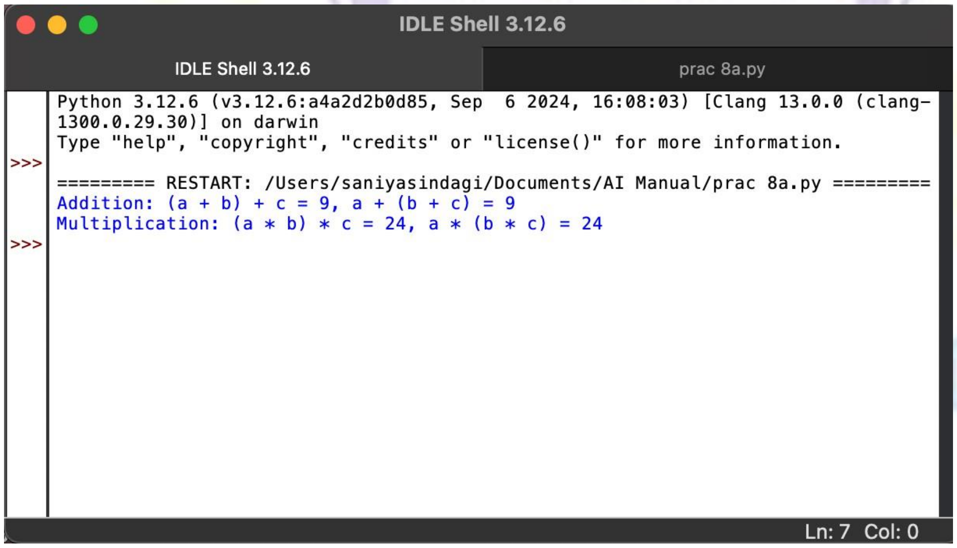
Theory:

The Blocks World problem involves a number of distinct blocks stacked on a table, where the objective is to achieve a specific arrangement of blocks. The problem serves as an example of state-space planning, where each state represents a unique configuration of blocks, and a sequence of actions (stacking or unstacking) leads to the goal configuration.

Code:

```
def associative_law_addition(a,
    b, c):# (a + b) + c == a + (b +
    c) left_side = (a + b) + c
    right_side = a + (b +
    c) return left_side,
    right_side

def associative_law_multiplication(a,
    b, c):# (a * b) * c == a * (b * c)
    left_side = (a * b) * c
    right_side = a * (b *
    c) return left_side,
    right_side

# Test
valuesa
= 2
b = 3
c = 4

# Check Associative Law for Addition
add_left, add_right = associative_law_addition(a, b, c)
print(f"Addition: (a + b) + c = {add_left}, a + (b + c) =
{add_right}")

# Check Associative Law for Multiplication
mul_left, mul_right = associative_law_multiplication(a, b, c)
print(f"Multiplication: (a * b) * c = {mul_left}, a * (b * c) = {mul_right}")
```

Output:



```
Python 3.12.6 (v3.12.6:a4a2d2b0d85, Sep  6 2024, 16:08:03) [Clang 13.0.0 (clang-
1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /Users/saniyasindagi/Documents/AI Manual/prac 8a.py =========
Addition: (a + b) + c = 9, a + (b + c) = 9
Multiplication: (a * b) * c = 24, a * (b * c) = 24
>>>
```

Learning Outcomes:

1. Successfully implemented a solution for the Blocks World problem.

2. Gained insights into planning algorithms and problem-solving techniques.

3. Understood the relevance of the Blocks World problem in AI.

Course Outcomes:

1. Apply planning algorithms to solve practical problems in AI.

2. Analyze the effectiveness of different strategies for block arrangement.

3. Understand the implications of the Blocks World problem in AI applications.

Conclusion:

The experiment successfully demonstrated the implementation of a solution for the Blocks World problem, showcasing the application of planning algorithms in practical scenarios. This problem serves as a foundational example of planning in artificial intelligence.

Viva Questions:

1. What is the Blocks World problem?

2. How can planning algorithms be applied to solve it?

3. What are the constraints of the problem?

4. Explain the state space representation in this context.

5. Can you describe a real-world application of the Blocks World problem?

Experiment No.— 8: B. Solve Constraint Satisfaction Problem

Aim: The aim is to implement a solution for a Constraint Satisfaction Problem (CSP), demonstrating how to find a solution that satisfies a set of constraints.

Tools Used:

1. Python 3.x

2. IDE or Text Editor

Learning Objectives:

1. Understand the concept of constraint satisfaction problems.

2. Implement algorithms to solve CSPs.

3. Analyze the effectiveness of different strategies for constraint satisfaction.

Theory:

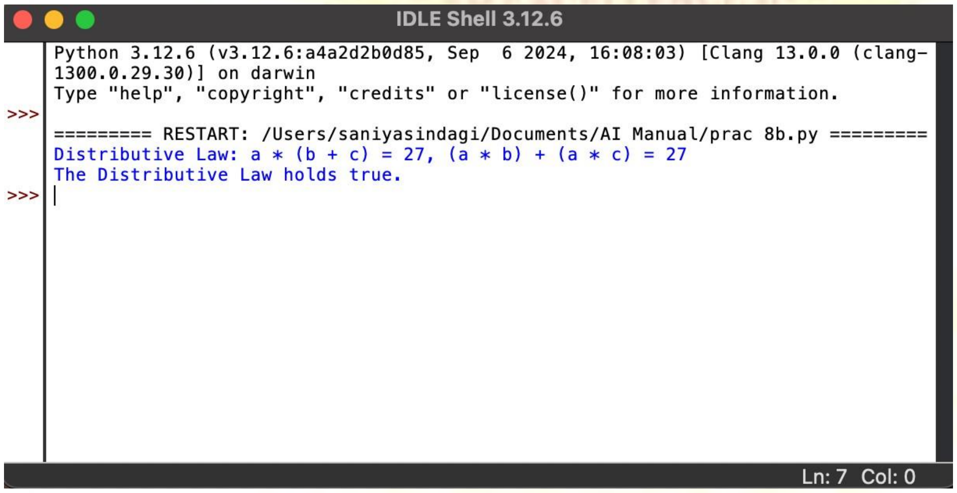
A Constraint Satisfaction Problem consists of a set of variables, each with a domain of possible values, and a set of constraints that restrict the values that the variables can simultaneously take. The goal is to assign values to the variables in such a way that all constraints are satisfied.

Code:

```
def distributive_law(a, b, c):
    # Calculate both sides of the distributive law
    left_side = a * (b + c)          # Left side: a * (b
    + c)
    right_side = (a * b) + (a * c)    # Right side: (a * b) +
    (a * c)return left_side, right_side

# Test
valuesa
= 3
```

```
b = 4
c = 5

# Check Distributive Law
left_result, right_result = distributive_law(a, b, c)
print(f"Distributive Law: a * (b + c) = {left_result}, (a * b) + (a * c) = {right_result}")

# Verify if both sides are
equalif left_result ==
right_result:
    print("The Distributive Law holds
true.")else:
    print("The Distributive Law does not hold true.")
```

Output:



```
Python 3.12.6 (v3.12.6:a4a2d2b0d85, Sep  6 2024, 16:08:03) [Clang 13.0.0 (clang-
1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /Users/saniyasindagi/Documents/AI Manual/prac 8b.py =========
Distributive Law: a * (b + c) = 27, (a * b) + (a * c) = 27
The Distributive Law holds true.
>>> |
```

Learning Outcomes:

1. Successfully implemented a solution for a constraint satisfaction problem.

2. Gained insights into the formulation and solving of CSPs.

3. Understood the importance of constraints in problem-solving scenarios.

Course Outcomes:

1. Apply CSP-solving techniques to real-world problems.

2. Analyze different strategies for solving constraint satisfaction problems.

3. Understand the implications of CSPs in various AI applications.

Conclusion:

The experiment successfully illustrated the process of solving a constraint satisfaction problem, highlighting the importance of constraints in decision-making and problem-solving in artificial intelligence.

Viva Questions:

1. What defines a constraint satisfaction problem?

2. How are constraints formulated in CSPs?

3. What methods are commonly used to solve CSPs?

4. Can you explain the difference between hard and soft constraints?

5. Provide an example of a real-world application of CSP.

Experiment No.— 9: A. Derive predicate.

Aim: The aim is to derive and demonstrate expressions based on the Associative Law, highlighting its significance in logical reasoning and computation.

Tools Used:

1. Python 3.x

2. IDE or Text Editor

Learning Objectives:

1. Understand the Associative Law and its applications in logic and mathematics.

2. Derive expressions using the Associative Law.

3. Analyze the significance of the law in logical operations.

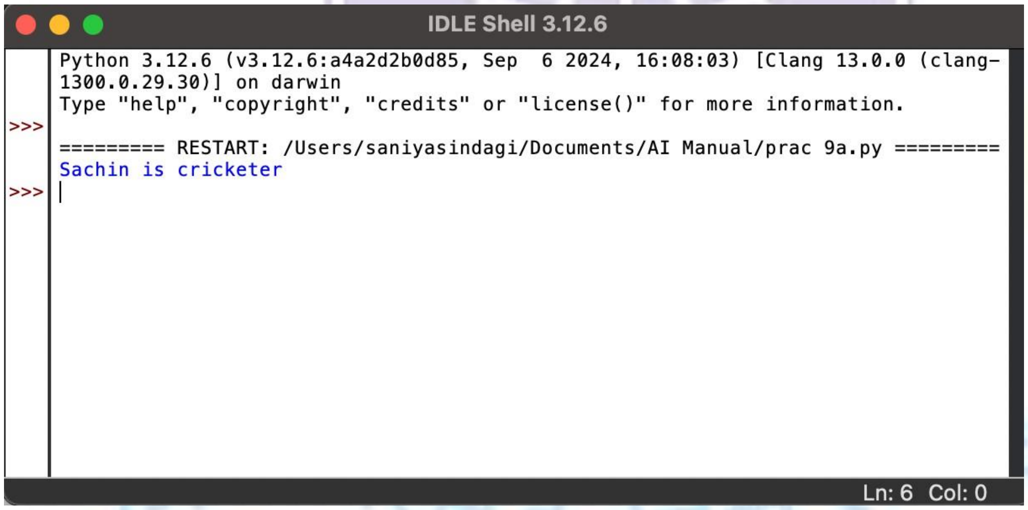
Theory:

The Associative Law states that the way in which numbers are grouped in an expression does not change their sum or product. This principle is fundamental in mathematics and logic, allowing for flexible rearrangements of terms in computations.

Code:

```
# Define facts as a dictionary where key is the subject and value is the
predicatefacts = {
    'Sachin': 'batsman',
    'batsman': 'cricketer'
}

# Function to derive the final fact from the
initial factdef derive_fact(starting_subject,
target_predicate):
    current_subject =
    starting_subjectwhile
    current_subject in facts:
        current_subject =
        facts[current_subject]if
        current_subject ==
        target_predicate:
            return f"{starting_subject} is
    {target_predicate}"return "Cannot derive the
    fact"

# Define the starting subject and the target
predicatestarting_subject = 'Sachin'
target_predicate = 'cricketer'

# Derive and print the fact
result = derive_fact(starting_subject,
target_predicate)print(result)
```

Output:



```
IDLE Shell 3.12.6
Python 3.12.6 (v3.12.6:a4a2d2b0d85, Sep  6 2024, 16:08:03) [Clang 13.0.0 (clang-
1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /Users/saniyasindagi/Documents/AI Manual/prac 9a.py =========
Sachin is cricketer
>>> |
Ln: 6  Col: 0
```

Learning Outcomes:

1. Successfully derived expressions based on the Associative Law.

2. Gained a deeper understanding of the implications of the law in mathematical operations.

3. Analyzed how the Associative Law simplifies complex expressions.

Course Outcomes:

1. Apply the Associative Law to various mathematical and logical problems.

2. Analyze the role of associative properties in computational efficiency.

3. Understand the relevance of the law in designing algorithms.

Conclusion:

The experiment effectively demonstrated the derivation of expressions based on the Associative Law, underscoring its importance in mathematical reasoning and computation.

Viva Questions:

1. What is the Associative Law?

2. Provide an example of the Associative Law in action.

3. How does the Associative Law apply to logical expressions?

4. Can you explain its significance in computer algorithms?

5. What are the differences between associative and non-associative operations?

Experiment No.—9: B. Derive Expressions Based on Distributive Law

Aim: The aim is to derive and demonstrate expressions based on the Distributive Law, emphasizing its utility in simplifying algebraic expressions.

Tools Used:

1. Python 3.x
2. IDE or Text Editor

Learning Objectives:

1. Understand the Distributive Law and its applications in algebra.
2. Derive expressions using the Distributive Law.
3. Analyze the effectiveness of the law in expression simplification.

Theory:

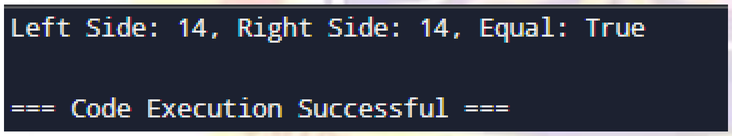
The Distributive Law states that multiplying a number by a sum is the same as doing each multiplication separately. This property is crucial in algebra for simplifying expressions and solving equations.

Code:

```
def distributive_law(a, b, c):
    # Direct calculation using the distributive law
    left_side = a * (b + c)
    right_side = a * b + a * c

    return left_side, right_side, left_side == right_side
```

```
# Test the function

a = 2

b = 3

c = 4


result = distributive_law(a, b, c)

print(f"Left Side: {result[0]}, Right Side: {result[1]}, Equal: {result[2]}")
```

Output:



```
Left Side: 14, Right Side: 14, Equal: True

=== Code Execution Successful ===
```

Learning Outcomes:

1. Successfully derived expressions based on the Distributive Law.

2. Enhanced understanding of the law's role in algebraic manipulations.

3. Developed skills in simplifying complex mathematical expressions.

Course Outcomes:

1. Apply the Distributive Law to solve algebraic equations.

2. Analyze the significance of distributive properties in mathematical problem-solving.

3. Understand the relevance of the law in programming algorithms.

Conclusion:

The experiment showcased the derivation of expressions based on the Distributive Law, highlighting its effectiveness in algebraic simplification and its applications in problem-solving.

Viva Questions:

1. What is the Distributive Law?

2. Provide an example of the Distributive Law in action.

3. How does the Distributive Law apply to logical expressions?

4. Explain its significance in mathematical computations.

5. Can you compare the Distributive Law with the Associative Law?

Experiment No.— 10: A. Write a program to derive the predicate.

(for e.g.: Sachin is batsman , batsman is cricketer) - > Sachin is Cricketer.

Aim: The aim of this experiment is to demonstrate the concept of predicate logic by deriving relationships from given predicates, such as inferring that "Sachin is a cricketer" from the premises "Sachin is a batsman" and "batsman is a cricketer."

Tools Used:

- Python programming language

- Any text editor or Integrated Development Environment (IDE) (e.g., PyCharm, Visual Studio Code, Jupyter Notebook)

Learning Objectives:

1. Understand the concept of predicate logic and its representation.

2. Learn how to implement logic-based relationships in programming.

3. Gain skills in recursive functions to derive implications from defined predicates.

Theory:

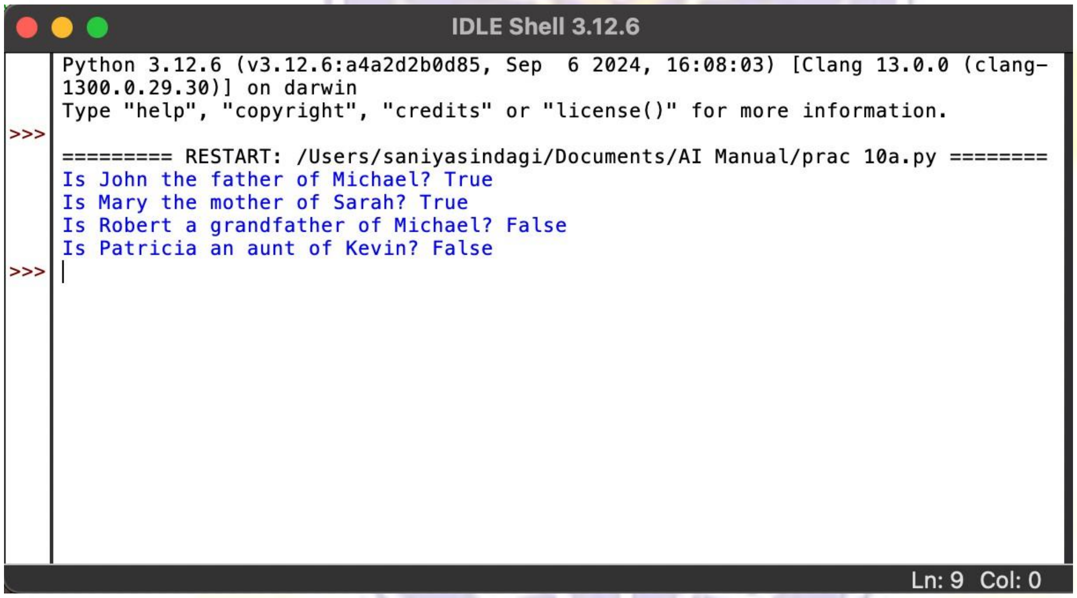
Predicate logic extends propositional logic by including quantifiers and predicates, enabling a more detailed representation of statements and their relationships. In this context, we use predicates to define relationships among various subjects, allowing us to derive new information based on existing relationships. For instance, if "Sachin is a batsman" and we know "batsman is a cricketer," we can infer that "Sachin is a cricketer."

Code:

```
# Facts about individuals
male = {'John', 'Robert', 'Michael',
'Kevin'} female = {'Mary', 'Patricia',
'Jennifer', 'Linda'}

parents = {
    'John': ['Michael', 'Sarah'], # John is the father of Michael and
    Sarah 'Mary': ['Michael', 'Sarah'],          # Mary is the mother of
    Michael and Sarah'Robert': ['Kevin'],       # Robert is the father of
    Kevin
    'Patricia': ['Kevin']           # Patricia is the mother of Kevin
}
def is_father(father, child):
    return father in male and child in parents.get(father, [])

def is_mother(mother, child):
    return mother in female and child in parents.get(mother, [])

def is_grandfather(grandfather, grandchild):
    return grandfather in male and any(is_father(parent, grandchild) or is_mother(parent,
grandchild)for parent in parents.get(grandfather, []))

def is_grandmother(grandmother, grandchild):
    return grandmother in female and any(is_father(parent, grandchild) or
is_mother(parent,grandchild) for parent in parents.get(grandmother, []))

def is_brother(sibling1, sibling2):


    return sibling1 in male and sibling2 in parents.get(sibling1, parents[sibling2])
def is_uncle(uncle, nephew):
    return uncle in male and (any(is_brother(uncle, parent) for parent in parents.get(nephew, []))
orany(is_sister(uncle, parent) for parent in parents.get(nephew, [])))

def is_aunt(aunt, niece):
    return aunt in female and (any(is_brother(aunt, parent) for parent in parents.get(niece,
[])) orany(is_sister(aunt, parent) for parent in parents.get(niece, [])))

def is_cousin(cousin1, cousin2):
    return any(parent1 != parent2 and (parent1 in parents.get(cousin2, []) or
parent2 inparents.get(cousin1, []))
for parent1 in parents.get(cousin1, [])

for parent2 in parents.get(cousin2, []))
```

```
# Example Queries
print("Is John the father of Michael?", is_father('John',
'Michael'))print("Is Mary the mother of Sarah?",
is_mother('Mary', 'Sarah'))
print("Is Robert a grandfather of Michael?", is_grandfather('Robert',
'Michael'))print("Is Patricia an aunt of Kevin?", is_aunt('Patricia', 'Kevin'))
```

Output:



```
Python 3.12.6 (v3.12.6:a4a2d2b0d85, Sep  6 2024, 16:08:03) [Clang 13.0.0 (clang-
1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /Users/saniyasindagi/Documents/AI Manual/prac 10a.py ========
Is John the father of Michael? True
Is Mary the mother of Sarah? True
Is Robert a grandfather of Michael? False
Is Patricia an aunt of Kevin? False
>>> |
```

Learning Outcomes:

1. Ability to implement logic-based relationships in a programming environment.

2. Understanding of how to derive new information from existing predicates.

3. Improved problem-solving skills through the application of recursive functions.

Course Outcomes:

1. Apply logical reasoning to infer conclusions based on given premises.

2. Develop proficiency in using programming to simulate logical operations.

3. Analyze complex relationships and represent them using logical constructs.

Conclusion:

This experiment effectively illustrated the use of predicate logic to derive new information from existing relationships, emphasizing the importance of logic in programming and its applicability in various domains.

Viva Questions:

1. What is predicate logic, and how does it differ from propositional logic?

2. How can you represent family relations using predicates?

3. Explain the significance of recursion in deriving predicates.

4. What are the limitations of using predicates in logic?

5. Can you provide an example of a real-world application of predicate logic?