Channabasaveshwara Institute of Technology
(Affiliated to VTU, Belagavi & Approved by AICTE, New Delhi)
(NAAC Accredited & ISO 9001:2015 Certified Institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.

Department of Artificial Intelligence and Data Science

ARTIFICIAL INTELLIGENCE
BAD402

(CBCS SCHEME)

B.E - IV Semester

Lab Manual 2025-26

Name: _____

USN: _____

Batch:_____Section: _____

.

QMP 7.1 D/F

Channabasaveshwara Institute of Technology
(Affiliated to VTU, Belagavi & Approved by AICTE, New Delhi)
(**NAAC Accredited & ISO 9001:2015 Certified Institution)**
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.

Department of Artificial Intelligence and Data Science

ARTIFICIAL INTELLIGENCE
BAD402
(PRACTICAL COMPONENT OF IPCC)

Prepared by:
Mrs. Tejaswini S
Assistant Professor
AD Department

HOD
Dr. Gavisiddappa
Prof. & Head
AD Department

.

Department of Artificial Intelligence & Data Science

SYLLABUS

ARTIFICIAL INTELLIGENCE
PRACTICAL COMPONENT OF IPCC
[As per Choice Based Credit System (CBCS) scheme]
SEMESTER – IV (AD)

Subject Code: BAD402 **CIE Marks: 25**
Hours/ Week : 02 (02 Hours Laboratory) **Test Hours: 03**

Using suitable simulation software, demonstrate the operation of the following programs:

Sl.No	Experiments
1.	Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem
2.	Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python
3.	Implement A* Search algorithm
4.	Implement AO* Search algorithm
5.	Solve 8-Queens Problem with suitable assumptions
6.	Implementation of TSP using heuristic approach
7.	Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining
8.	Implement resolution principle on FOPL related problems
9.	Implement Tic-Tac-Toe game using Python
10.	Build a bot which provides all the information related to text in search box
11.	Implement any Game and demonstrate the Game playing strategies

CIE for the practical component of the IPCC

- **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments write-ups are added and scaled down to **15 marks**.
- The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

General Instructions to Students

1. Students should come with thorough preparation for the experiment to be conducted.
2. Students should take prior permission from the concerned faculty before availing the leave.
3. Students should come with formals and to be present on time in the laboratory.
4. Students will not be permitted to attend the laboratory unless they bring the practical record fully completed in all respects pertaining to the experiments conducted in the previous session.
5. Students will be permitted to attend the laboratory unless they bring the observation book fully completed in all respects pertaining to the experiments conducted in the present session.
6. They should obtain the signature of the staff-in –charge in the observation book after completing each experiment.
7. Practical record should be neatly maintained.
8. Ask lab Instructor for assistance for any problem.
9. Do not download or install software without the assistance of laboratory Instructor.
10. Do not alter the configuration of system.
11. Turn off the systems after use.

Program 1
Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

Theory:

There are two jugs of volume A litre and B litre. Neither has any measuring mark on it.There is a pump that can be used to fill the jugs with water. How can you get exactly x litre of water into the A litre jug. Assuming that we have unlimited supply of water.

Note: Let's assume we have A=4 litre and B= 3 litre jugs. And we want exactly 2 Litre water into jug A (i.e. 4 litre jug) how we will do this.

The state space for this problem can be described as the set of ordered pairs of integers (x, y)

Where, x represents the quantity of water in the 4-gallon jug x= 0,1,2,3,4

y represents the quantity of water in 3-gallon jug y=0,1,2,3

Start State: (0,0)

Goal State: (2,0)

Generate production rules for the water jug problem

We basically perform three operations to achieve the goal.

1. Fill water jug.
2. Empty water jug
3. Transfer water jug

Production Rules: **Rule**	**State**	**Process**
1	(X,Y | X<4)	(4,Y) {Fill 4-gallon jug}
2	(X,Y |Y<3)	(X,3) {Fill 3-gallon jug}
3	(X,Y |X>0)	(0,Y) {Empty 4-gallon jug}
4	(X,Y | Y>0)	(X,0) {Empty 3-gallon jug}
5	(X,Y | X+Y>=4 ^ Y>0)	(4,Y-(4-X)) {Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}
6	(X,Y | X+Y>=3 ^X>0)	(X-(3-Y),3) {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}
7	(X,Y | X+Y<=4 ^Y>0)	(X+Y,0) {Pour all water from 3-gallon jug into 4-gallon jug}
8	(X,Y | X+Y <=3^ X>0)	(0,X+Y) {Pour all water from 4-gallon jug into 3-gallon jug}
9	(0,2)	(2,0) {Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

Initialization:

Start State: (0,0)

Apply Rule 2:

(X,Y | Y<3) ->

(X,3)

{Fill 3-gallon jug}

Now the state is (X,3)

Iteration 1:

Current State: (X,3)

Apply Rule 7:

(X,Y | X+Y<=4 ^Y>0)

(X+Y,0)

{Pour all water from 3-gallon jug into 4-gallon jug}

Now the state is (3,0)

Iteration 2:

Current State : (3,0)

Apply Rule 2:

(X,Y | Y<3) ->

(3,3)

{Fill 3-gallon jug}

Now the state is (3,3)

Iteration 3:

Current State:(3,3)

Apply Rule 5:

(X,Y | X+Y>=4 ^ Y>0)

(4,Y-(4-X))

{Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}

Now the state is (4,2)

Iteration 4:

Current State : (4,2)

Apply Rule 3:

(X,Y | X>0)

(0,Y)

{Empty 4-gallon jug}

Now state is (0,2)

Iteration 5:

Current State : (0,2)

Apply Rule 9:

(0,2)

(2,0)

{Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

Now the state is (2,0)

Program:

```
from collections import defaultdict
# jug1 and jug2 contain the value
jug1, jug2, aim = 4, 3, 2
# Initialize dictionary with default value as false.

visited = defaultdict(lambda: False)
def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
            waterJugSolver(amt1, 0) or
            waterJugSolver(jug1, amt2) or
            waterJugSolver(amt1, jug2) or
            waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
            amt2 - min(amt2, (jug1-amt1))) or
            waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
            amt2 + min(amt1, (jug2-amt2))))
    else:
        return False
print("Steps: ")
waterJugSolver(0, 0)
```

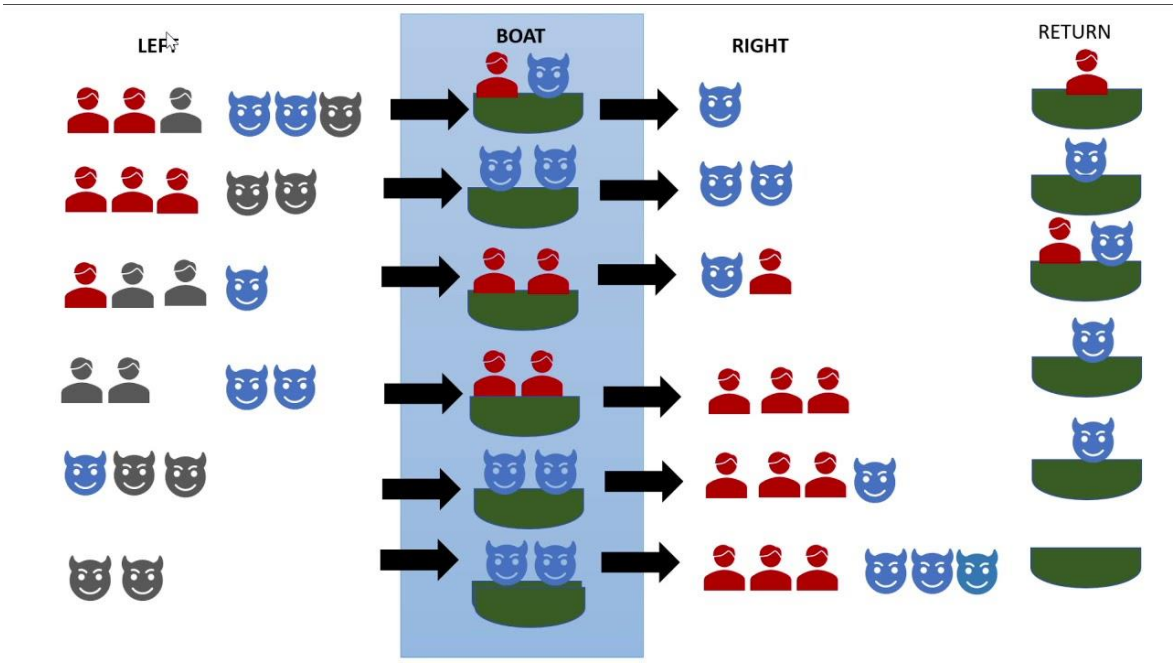
Result:

```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2

Out[1]: True
```

Program 2
Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python.

Theory:

- Three missionaries and three cannibals wish to cross the river.
- They have a small boat that will carry up to two people.
- Everyone can navigate the boat.
- If at any time the cannibals outnumber the missionaries on bank of the river, they will eat the missionaries.
- Goal is to get everyone across the river without the missionaries risking being eaten by cannibals.



Program:

```
#Python program to illustrate Missionaries & cannibals Problem
print("\n")
print("\tGame Start\nNow the task is to move all of them to right side of the river")
print('''rules:\n1. The boat can carry at most two people\n2. If cannibals num greater than missionaries then the cannibals would eat the missionaries\n3. The boat cannot cross the river by itself with no people on board''')
lM = 3          #lM = Left side Missionaries number
lC = 3          #lC = Left side Cannibals number
rM=0            #rM = Right side Missionaries number
rC=0            #rC = Right side cannibals number
userM = 0       #userM = User input for number of missionaries for right to left side travel
userC = 0       #userC = User input for number of cannibals for right to left travel
k = 0
print("\nM M M C C C |    --- | \n")
try:
        while(True):
                while(True):
                        print("Left side -> right side river travel")
                        #uM = user input for number of missionaries for left to right travel
                        #uC = user input for  number of cannibals for left to right travel
                        uM = int(input("Enter number of Missionaries travel => "))
                        uC = int(input("Enter number of Cannibals travel => "))
                        if((uM==0)and(uC==0)):
                                print("Empty travel not possible")
                                print("Re-enter : ")
                        elif(((uM+uC) <= 2)and((lM-uM)>=0)and((lC-uC)>=0)):
                                break
                        else:
                                print("Wrong input re-enter : ")
                lM = (lM-uM)
                lC = (lC-uC)
                rM += uM
                rC += uC
```

```
        print("\n")
        for i in range(0,lM):
                print("M ",end="")
        for i in range(0,lC):
                print("C ",end="")
        print("| --> | ",end="")
        for i in range(0,rM):
                print("M ",end="")
        for i in range(0,rC):
                print("C ",end="")
        print("\n")
        k +=1
    if(((lC==3)and (lM ==1)) or ((lC==3)and(lM==2)) or((lC==2)and(lM==1))
or((rC==3)and (rM == 1)) or ((rC==3)and(rM==2)) or ((rC==2)and(rM==1))):
                print("Cannibals eat missionaries:\nYou lost the game")
                break
    if((rM+rC) == 6):
            print("You won the game : \n\tCongrats")
            print("Total attempt")
            print(k)
            break
    while(True):
                print("Right side -> Left side river travel")
                userM = int(input("Enter number of Missionaries travel => "))
                userC = int(input("Enter number of Cannibals travel => "))
               if((userM==0)and(userC==0)):
                        print("Empty travel not possible")
                        print("Re-enter : ")
                elif(((userM+userC) <= 2)and((rM-userM)>=0)and((rC-userC)>=0)):
                        break
                else:
                        print("Wrong input re-enter : ")
        lM += userM
        lC += userC
        rM -= userM
        rC -= userC
        k +=1
        print("\n")
        for i in range(0,lM):
```

```
                print("M ",end="")
            for i in range(0,lC):
                print("C ",end="")
            print("| <-- | ",end="")
            for i in range(0,rM):
                print("M ",end="")
            for i in range(0,rC):
                print("C ",end="")
            print("\n")

            if(((lC==3)and (lM == 1)) or ((lC==3)and(lM==2)) or((lC==2)and(lM==1))or
        ((rC==3)and (rM == 1)) or ((rC==3)and(rM==2)) or ((rC==2)and(rM==1))):
                print("Cannibals eat missionaries:\nYou lost the game")
                break
except EOFError as e:
            print("\nInvalid input please retry !!")
```

Result:

```
        Game Start
Now the task is to move all of them to right side of the river
rules:
1. The boat can carry at most two people
2. If cannibals num greater than missionaries then the cannibals would eat the missionaries
3. The boat cannot cross the river by itself with no people on board

M M M C C C |     --- |

Left side -> right side river travel
Enter number of Missionaries travel => 1
Enter number of Cannibals travel => 1


M M C C | --> | M C

Right side -> Left side river travel
Enter number of Missionaries travel => 1
Enter number of Cannibals travel => 0


M M M C C | <-- | C

Left side -> right side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 2
```

```
M M M | --> | C C C

Right side -> Left side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 1


M M M C | <-- | C C

Left side -> right side river travel
Enter number of Missionaries travel => 2
Enter number of Cannibals travel => 0


M C | --> | M M C C

Right side -> Left side river travel
Enter number of Missionaries travel => 1
Enter number of Cannibals travel => 1


M M C C | <-- | M C

Left side -> right side river travel
Enter number of Missionaries travel => 2
Enter number of Cannibals travel => 0


  C C | --> | M M M C

  Right side -> Left side river travel
  Enter number of Missionaries travel => 0
  Enter number of Cannibals travel => 1


  C C C | <-- | M M M

  Left side -> right side river travel
  Enter number of Missionaries travel => 0
  Enter number of Cannibals travel => 2


  C | --> | M M M C C

  Right side -> Left side river travel
  Enter number of Missionaries travel => 0
  Enter number of Cannibals travel => 1


  C C | <-- | M M M C

  Left side -> right side river travel
  Enter number of Missionaries travel => 0
  Enter number of Cannibals travel => 2


     | --> | M M M C C C

     You won the game :
             Congrats
     Total attempt
     11
```

Program 3
Implement A* Search algorithm

A* Search Algorithm and Its Basic Concepts

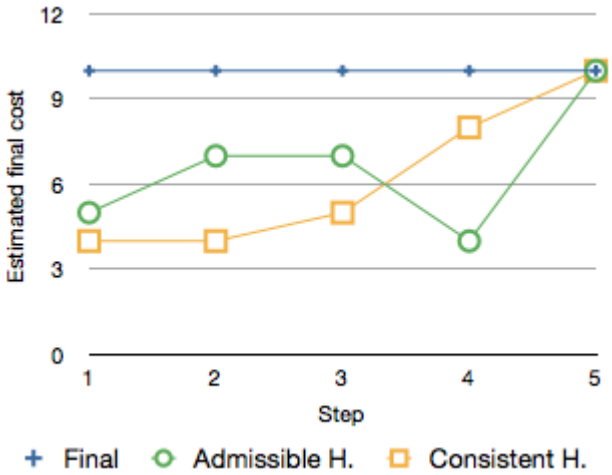
A* algorithm works based on heuristic methods, and this helps achieve optimality. A* is a different form of the best-first algorithm. Optimality empowers an algorithm to find the best possible solution to a problem. Such algorithms also offer completeness; if there is any solution possible to an existing problem, the algorithm will definitely find it.

When A* enters into a problem, firstly, it calculates the cost to travel to the neighboring nodes and chooses the node with the lowest cost. If The f(n) denotes the cost, A* chooses the node with the lowest f(n) value. Here 'n' denotes the neighboring nodes. The calculation of the value can be done as shown below:

f(n)=g(n)+h(n)f(n)=g(n)+h(n)
g(n) = shows the shortest path's value from the starting node to node n
h(n) = The heuristic approximation of the value of the node

The heuristic value has an important role in the efficiency of the A* algorithm. To find the best solution, you might have to use different heuristic functions according to the type of the problem. However, the creation of these functions is a difficult task, and this is the basic problem we face in AI.

What is a Heuristic Function?



A heuristic is simply called a heuristic function that helps rank the alternatives given in a search algorithm at each of its steps. It can either produce a result on its own or work in conjugation with a given algorithm to create a result. Essentially, a heuristic function helps algorithms to make the best decision faster and more efficiently. This ranking is based on the best available information and

helps the algorithm decide the best possible branch to follow. Admissibility and consistency are the two fundamental properties of a heuristic function.

Admissibility of the Heuristic Function

A heuristic function is admissible if it can effectively estimate the real distance between a node 'n' and the end node. It never overestimates; if it ever does, it will be denoted by 'd', which also denotes the accuracy of the solution.

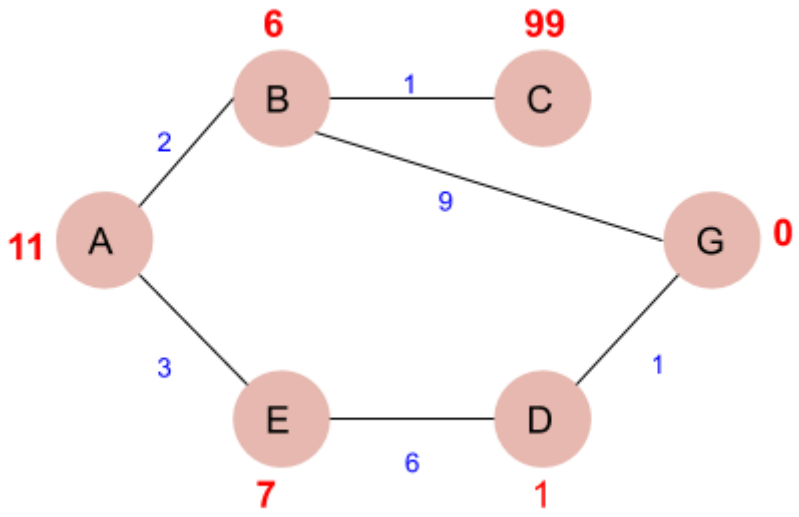
Consistency of the Heuristic Function

A heuristic function is consistent if the estimate of a given heuristic function turns out to be equal to or less than the distance between the goal (n) and a neighbor and the cost calculated to reach that neighbor.

A* is indeed a very powerful algorithm used to increase the performance of artificial intelligence. It is one of the most popular search algorithms in AI. The sky is the limit when it comes to the potential of this algorithm. However, the efficiency of an A* algorithm highly depends on the quality of its heuristic function. Wonder why this algorithm is preferred and used in many software systems? There is no single facet of AI where the A*algorithm has not found its application. From search optimization to games, robotics, and machine learning, the A* algorithm is an inevitable part of a smart program.

Implementation with Python

In this section, we are going to find out how the A* search algorithm can be used to find the most cost-effective path in a graph. Consider the following graph below.



The numbers written on edges represent the distance between the nodes, while the numbers written on nodes represent the heuristic values. Let us find the most cost-effective path to reach from start state A to final state G using the A* Algorithm.

Let's start with node A. Since A is a starting node, therefore, the value of g(x) for A is zero, and from the graph, we get the heuristic value of A is 11, therefore

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

$0+ 11 =11$

Thus for A, we can write

$A=11$

Now from A, we can go to point B or point E, so we compute f(x) for each of them

$A \rightarrow B = 2 + 6 = 8$

$A \rightarrow E = 3 + 6 = 9$

Since the cost for $A \rightarrow B$ is less, we move forward with this path and compute the f(x) for the children nodes of B

Since there is no path between C and G, the heuristic cost is set to infinity or a very high value

$A \rightarrow B \rightarrow C = (2 + 1) + 99= 102$

$A \rightarrow B \rightarrow G = (2 + 9) + 0 = 11$

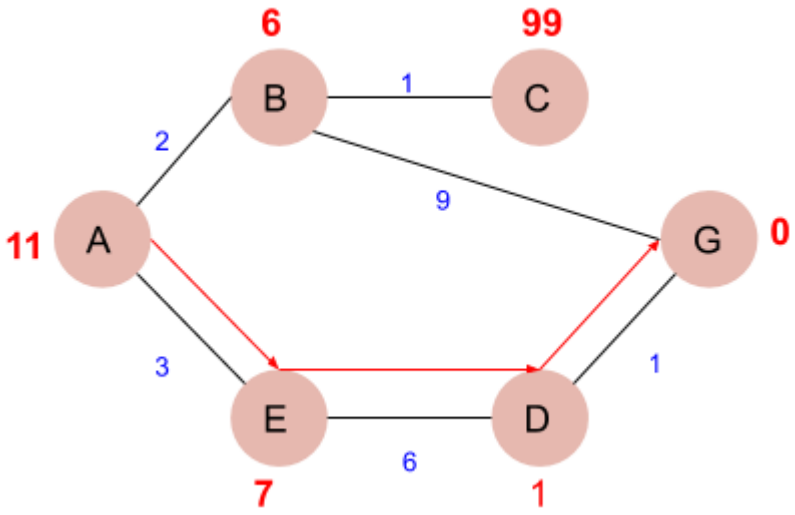
Here the path $A \rightarrow B \rightarrow G$ has the least cost but it is still more than the cost of $A \rightarrow E$, thus we explore this path further

A → E → D = (3 + 6) + 1 = 10

Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path. And compute the f(x) for the children of D

A → E → D → G = (3 + 6 + 1) +0 =10

Now comparing all the paths that lead us to the goal, we conclude that A → E → D → G is the most cost-effective path to get from A to G.



Next, we write a program in Python that can find the most cost-effective path by using the a-star algorithm.

First, we create two sets, viz- open and close. The open contains the nodes that have been visited, but their neighbors are yet to be explored. On the other hand, close contains nodes that, along with their neighbors, have been visited.

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
         #distance 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!')
```

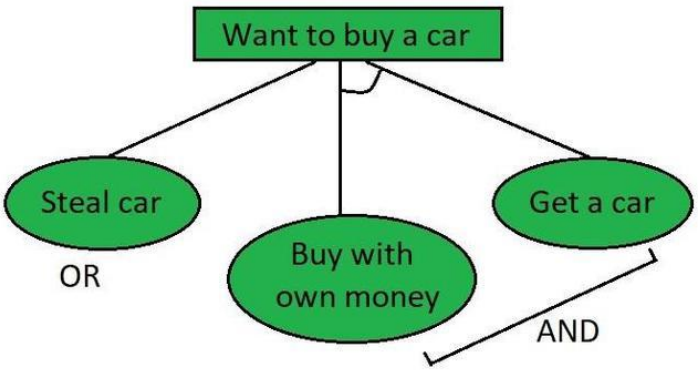
```
            return None
    # if the current node is the stop_node then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            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')
```

Result:

```
Path found: ['A', 'F', 'G', 'I', 'J']
Out[6]: ['A', 'F', 'G', 'I', 'J']
```

Program 4
Implement AO* Search algorithm

The **AO* algorithm**, short for **"Anytime Optimistic"** algorithm.
Best-first search is what the AO* algorithm does. The AO* method **divides** any given difficult **problem into a smaller group** of problems that are then resolved **using the AND-OR** graph concept. AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems. The AND side of the graph represents a set of tasks that must be completed to achieve the main goal, while the OR side of the graph represents different methods for accomplishing the same main goal.



In the above figure, the buying of a car may be broken down into smaller problems or tasks that can be accomplished to achieve the main goal in the above figure, which is an example of a simple AND-OR graph. The other task is to either steal a car that will help us accomplish the main goal or use your own money to purchase a car that will accomplish the main goal. The AND symbol is used to indicate the AND part of the graphs, which refers to the need that all sub problems containing the AND to be resolved before the preceding node or issue may be finished.

The start state and the target state are already known in the knowledge-based search strategy known as the AO* algorithm, and the best path is identified by heuristics. The informed search technique considerably reduces the algorithm's time complexity. The AO* algorithm is far more effective in searching AND-OR trees than the A* algorithm.

Working of AO* algorithm:

The evaluation function in AO* looks like this:

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

f(n) = Actual cost + Estimated cost

here,

f(n) = The actual cost of traversal.
g(n) = the cost from the initial node to the current node.
h(n) = estimated cost from the current node to the goal state.

Difference between the A* Algorithm and AO* algorithm

- A* algorithm and AO* algorithm both works on the **best first search**.
- They are both **informed search** and works on given heuristics values.
- **A*** always **gives** the **optimal solution** but AO* doesn't guarantee to give the optimal solution.
- Once AO* got a solution **doesn't explore** all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses **less memory.**
- opposite to the A* algorithm, the AO* algorithm cannot go into an endless **loop.**

Real-Life Applications of AO* algorithm:

Vehicle Routing Problem:
The vehicle routing problem is determining the shortest routes for a fleet of vehicles to visit a set of customers and return to the depot, while minimizing the total distance traveled and the total time taken. The AO* algorithm can be used to find the optimal routes that satisfy both objectives.

Portfolio Optimization:
Portfolio optimization is choosing a set of investments that maximize returns while minimizing risks. The AO* algorithm can be used to find the optimal portfolio that satisfies both objectives, such as maximizing the expected return and minimizing the standard deviation.

Program:

```
class Graph:
    def __init__(self, graph_dict=None, heuristic=None):
        self.graph = graph_dict or {}
        self.heuristic = heuristic or {}

    def get_neighbors(self, node):
        return self.graph.get(node, [])

    def get_heuristic(self, node):
        return self.heuristic.get(node, float('inf'))

def ao_star(graph, start):
    open_list = set([start])
    closed_list = set()
    solution_graph = {}
```

```
def recur_ao_star(node):
    if node in closed_list:
        return solution_graph[node]

    neighbors = graph.get_neighbors(node)
    if not neighbors:
        solution_graph[node] = (0, None)
        closed_list.add(node)
        return solution_graph[node]

    min_cost = float('inf')
    best_path = None

    for path, cost in neighbors:
        total_cost = cost
        sub_path = []

        for sub_node in path:
            sub_cost, _ = recur_ao_star(sub_node)
            total_cost += sub_cost
            sub_path.append(sub_node)

        if total_cost < min_cost:
            min_cost = total_cost
            best_path = sub_path

    solution_graph[node] = (min_cost, best_path)
    closed_list.add(node)
    return solution_graph[node]

recur_ao_star(start)
return solution_graph
```

```
# Example usage
if __name__ == "__main__":
    graph = {
        'A': [(['B', 'C'], 1), (['D'], 5)],
        'B': [(['E'], 3)],
        'C': [(['E'], 1)],
        'D': [(['G'], 2)],
        'E': [(['G'], 5)],
        'G': []
    }
    heuristic = {
        'A': 6,
        'B': 2,
        'C': 2,
        'D': 4,
        'E': 0,
        'G': 0
    }
    g = Graph(graph, heuristic)
    solution = ao_star(g, 'A')

    print("Solution Graph:")
    for node, (cost, path) in solution.items():
        print(f"Node: {node}, Cost: {cost}, Path: {path}")
```

Result:

```
Solution Graph:
Node: G, Cost: 0, Path: None
Node: E, Cost: 5, Path: ['G']
Node: B, Cost: 8, Path: ['E']
Node: C, Cost: 6, Path: ['E']
Node: D, Cost: 2, Path: ['G']
Node: A, Cost: 7, Path: ['D']
```

Program 5
Solve 8-Queens Problem with suitable assumptions

Program:

```
N = 8
board = [[0] * N for _ in range(N)]

def print_solution(board):
    for row in board:
        print(' '.join('Q' if x else '.' for x in row))
def is_safe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True
```

```
def solve_nq_util(board, col):
    if col >= N:
        return True
    for i in range(N):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_nq_util(board, col + 1) == True:
                return True
            board[i][col] = 0
    return False

def solve_nq():
    if not solve_nq_util(board, 0):
        print("Solution does not exist")
        return False

    print_solution(board)
    return True

solve_nq()
```

Result:

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

Program 6
Implementation of TSP using heuristic approach

Algorithm for Traveling Salesman Problem

We will use the dynamic programming approach to solve the Travelling Salesman Problem (TSP).

Before starting the algorithm, let's get acquainted with some terminologies:

- A graph G=(V, E), which is a set of vertices and edges.
- V is the set of vertices.
- E is the set of edges.
- Vertices are connected through edges.
- Dist(i,j) denotes the non-negative distance between two vertices, i and j.

Let's assume S is the subset of cities and belongs to {1, 2, 3, …, n} where 1, 2, 3…n are the cities and i, j are two cities in that subset. Now cost(i, S, j) is defined in such a way as the length of the shortest path visiting node in S, which is exactly once having the starting and ending point as i and j respectively.

For example, cost (1, {2, 3, 4}, 1) denotes the length of the shortest path where:

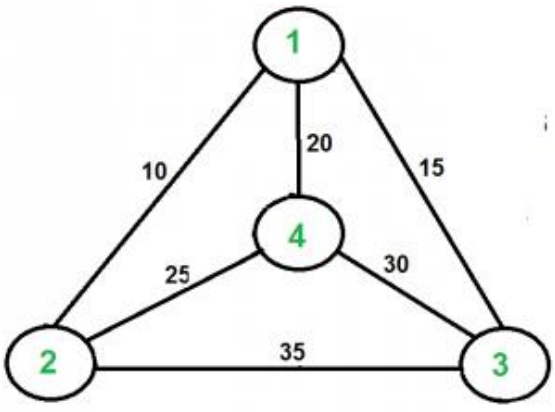
- Starting city is 1
- Cities 2, 3, and 4 are visited only once
- The ending point is 1

The dynamic programming algorithm would be:

- Set cost(i, , i) = 0, which means we start and end at i, and the cost is 0.
- When |S| > 1, we define cost(i, S, 1) = ∝ where i !=1 . Because initially, we do not know the exact cost to reach city i to city 1 through other cities.
- Now, we need to start at 1 and complete the tour. We need to select the next city in such a way-

cost(i, S, j)=min cost (i, S−{i}, j)+dist(i,j) where i∈S and i≠j

For the given figure, the adjacency matrix would be the following:



dist(i,j)	1	2	3	4
1	0	10	15	20
2	10	0	35	25
3	15	35	0	30
4	20	25	30	0

Let's see how our algorithm works:

Step 1) We are considering our journey starting at city 1, visit other cities once and return to city 1.

Step 2) S is the subset of cities. According to our algorithm, for all |S| > 1, we will set the distance cost(i, S, 1) = ∝. Here cost(i, S, j) means we are starting at city i, visiting the cities of S once, and now we are at city j. We set this path cost as infinity because we do not know the distance yet. So the values will be the following:

Cost (2, {3, 4}, 1) = ∝ ; the notation denotes we are starting at city 2, going through cities 3, 4, and reaching 1. And the path cost is infinity. Similarly-

cost(3, {2, 4}, 1) = ∝

cost(4, {2, 3}, 1) = ∝

Step 3) Now, for all subsets of S, we need to find the following:

cost(i, S, j)=min cost (i, S−{i}, j)+dist(i,j), where j∈S and i≠j

That means the minimum cost path for starting at i, going through the subset of cities once, and returning to city j. Considering that the journey starts at city 1, the optimal path cost would be= cost(1, {other cities}, 1).

Let's find out how we could achieve that:

Now S = {1, 2, 3, 4}. There are four elements. Hence the number of subsets will be 2^4 or 16. Those subsets are-

1) |S| = Null:

{Φ}

2) |S| = 1:

{{1}, {2}, {3}, {4}}

3) |S| = 2:

{{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}}

4) |S| = 3:

{{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {1, 3, 4}}

5) |S| = 4:

{{1, 2, 3, 4}}

As we are starting at 1, we could discard the subsets containing city 1.

The algorithm calculation:

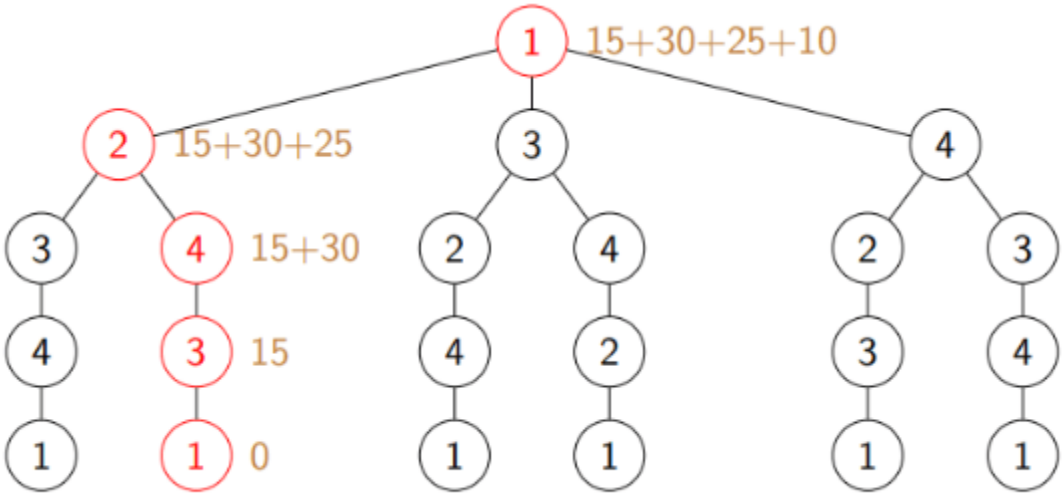
1) |S| = Φ:

cost (2, Φ, 1) = dist(2, 1) = 10
cost (3, Φ, 1) = dist(3, 1) = 15
cost (4, Φ, 1) = dist(4, 1) = 20
2) |S| = 1:

cost (2, {3}, 1) = dist(2, 3) + cost (3, Φ, 1) = 35+15 = 50
cost (2, {4}, 1) = dist(2, 4) + cost (4, Φ, 1) = 25+20 = 45
cost (3, {2}, 1) = dist(3, 2) + cost (2, Φ, 1) = 35+10 = 45
cost (3, {4}, 1) = dist(3, 4) + cost (4, Φ, 1) = 30+20 = 50
cost (4, {2}, 1) = dist(4, 2) + cost (2, Φ, 1) = 25+10 = 35
cost (4, {3}, 1) = dist(4, 3) + cost (3, Φ, 1) = 30+15 = 45
3) |S| = 2:

cost (2, {3, 4}, 1) = min [dist[2,3]+Cost(3,{4},1) = 35+50 = 85,
dist[2,4]+Cost(4,{3},1) = 25+45 = 70] = 70
cost (3, {2, 4}, 1) = min [dist[3,2]+Cost(2,{4},1) = 35+45 = 80,
dist[3,4]+Cost(4,{2},1) = 30+35 = 65] = 65
cost (4, {2, 3}, 1) = min [dist[4,2]+Cost(2,{3},1) = 25+50 = 75
dist[4,3]+Cost(3,{2},1) = 30+45 = 75] = 75
4) |S| = 3:

cost (1, {2, 3, 4}, 1) = min [dist[1,2]+Cost(2,{3,4},1) = 10+70 = 80
dist[1,3]+Cost(3,{2,4},1) = 15+65 = 80
dist[1,4]+Cost(4,{2,3},1) = 20+75 = 95] = 80
So the optimal solution would be **1-2-4-3-1**



```
Output of Given Graph:

Minimum weight Hamiltonian Cycle:

10 + 25 + 30 + 15:= 80
```

Program:

```python
# Python3 program to implement traveling salesman
# problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4
```

```python
# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    vertex = []          # store all vertex apart from source vertex
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        current_pathweight = 0        # store current Path weight(cost)

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path
```

```python
# Driver Code
if __name__ == "__main__":

    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
             [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

Result:

```
80
```

Program 7
Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining

Let’s implement both Forward Chaining and Backward Chaining with a specific example to illustrate how these strategies work.

Let's set up a simple knowledge base containing logical rules and facts related to animal classification. We'll implement both forward chaining and backward chaining to determine whether a given animal is classified as a mammal or bird based on the provided rules.

Knowledge Base

We'll define a set of rules and facts:

- Facts:
  - has_fur(tiger)
  - has_feathers(penguin)
  - lays_eggs(penguin)
  - lays_eggs(sparrow)
  - has_fur(cat)
- Rules:
  - If an animal has fur, it is a mammal.
  - If an animal has feathers and lays eggs, it is a bird.

Forward Chaining

Let's implement forward chaining to classify animals.

```
class ForwardChaining:
    def __init__(self, rules, facts):
        self.rules = rules
        self.facts = set(facts)

    def apply_rules(self):
        while True:
            new_facts = set()
            for antecedent, consequent in self.rules:
                if antecedent.issubset(self.facts):
                    new_facts.add(consequent)
            if not new_facts.difference(self.facts):
                break
            self.facts.update(new_facts)

    def get_facts(self):
        return self.facts
```

```
# Define rules as (antecedent, consequent) pairs
rules = [
    ({"has_fur(tiger)"}, "mammal(tiger)"),
    ({"has_feathers(penguin)", "lays_eggs(penguin)"}, "bird(penguin)"),
    ({"lays_eggs(sparrow)", "has_feathers(sparrow)"}, "bird(sparrow)"),
    ({"has_fur(cat)"}, "mammal(cat)")
]

# Initial facts
initial_facts = {"has_fur(tiger)", "has_feathers(penguin)", "lays_eggs(penguin)", "lays_eggs(sparrow)", "has_fur(cat)"}

fc = ForwardChaining(rules, initial_facts)
fc.apply_rules()

print("Derived Facts:", fc.get_facts())
```

Output:

```
Derived Facts: {'has_fur(cat)', 'mammal(cat)', 'lays_eggs(penguin)', 'lays_eggs(sparrow)', 'bird(penguin)', 'mammal(tiger)', 'h
as_feathers(penguin)', 'has_fur(tiger)'}
```

Backward Chaining

Now, let's implement backward chaining to classify animals.

```
class BackwardChaining:
    def __init__(self, rules, facts):
        self.rules = rules
        self.facts = set(facts)

    def is_fact(self, fact):
        if fact in self.facts:
            return True

        for antecedent, consequent in self.rules:
            if consequent == fact:
                if all(self.is_fact(ant) for ant in antecedent):
                    self.facts.add(fact)
                    return True
        return False
```

```
# Define rules as (antecedent, consequent) pairs
rules = [
    ({"has_fur(tiger)"}, "mammal(tiger)"),
    ({"has_feathers(penguin)", "lays_eggs(penguin)"}, "bird(penguin)"),
    ({"lays_eggs(sparrow)", "has_feathers(sparrow)"}, "bird(sparrow)"),
    ({"has_fur(cat)"}, "mammal(cat)")
]

# Initial facts
initial_facts = {"has_fur(tiger)", "has_feathers(penguin)", "lays_eggs(penguin)", "lays_eggs(sparrow)", "has_fur(cat)"}

# Goals
goals = ["mammal(tiger)", "bird(penguin)", "bird(sparrow)", "mammal(cat)"]

bc = BackwardChaining(rules, initial_facts)

for goal in goals:
    if bc.is_fact(goal):
        print(f"Goal {goal} can be derived from the facts.")
    else:
        print(f"Goal {goal} cannot be derived from the facts.")
```

Output:

```
Goal mammal(tiger) can be derived from the facts.
Goal bird(penguin) can be derived from the facts.
Goal bird(sparrow) cannot be derived from the facts.
Goal mammal(cat) can be derived from the facts.
```

Explanation

- **Forward Chaining**: Starts with initial facts and iteratively applies rules to derive new facts. In the provided example, it will derive that `tiger` and `cat` are mammals and `penguin` and `sparrow` are birds based on the given rules and initial facts.
- **Backward Chaining**: Starts with the goal and checks if it can be derived from the facts. It recursively checks the antecedents of each rule to see if they can be satisfied by the initial facts. In the provided example, it will check if the given goals (`mammal(tiger)`, `bird(penguin)`, `bird(sparrow)`, `mammal(cat)`) can be derived from the initial facts and rules.

These implementations demonstrate how forward and backward chaining can be used to classify animals based on a set of rules and initial facts in Python.

Program 8
Implement resolution principle on FOPL related problems

```
def negation(literal):
    """Returns the negation of a literal."""
    if literal.startswith("~"):
        return literal[1:]
    else:
        return "~" + literal

def resolve(clause1, clause2):
    """Performs resolution on two clauses."""
    resolvents = []
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == negation(literal2):
                resolvent = [lit for lit in (clause1 + clause2) if lit != literal1 and lit != literal2]
                resolvents.append(resolvent)
    return resolvents

def resolve_all(clauses):
    """Performs resolution on all possible pairs of clauses."""
    new_clauses = set()
    for i, clause1 in enumerate(clauses):
        for clause2 in clauses[i+1:]:
            new_clauses.update(resolve(clause1, clause2))
    return new_clauses

def resolution(kb):
    """Performs resolution to determine the satisfiability of the knowledge base."""
    clauses = [clause.split("||") for clause in kb]
    while True:
        new_clauses = resolve_all(clauses)
        if not new_clauses:
            return "Satisfiable"
        if any([] in clause for clause in new_clauses):
            return "Unsatisfiable"
        clauses.extend(new_clauses)
```

```
# Example usage:
knowledge_base = [
    "(P || Q || ~R)",
    "(~P || R)",
    "(~Q || R)",
    "(~R || ~P || Q)"
]
result = resolution(knowledge_base)
print(result)
```

Output:

```
Satisfiable
```

Explanation

1. **Negation Function**: The negation function handles negating a literal by checking if it starts with ~ and appropriately removing or adding it.
2. **Resolve Function**: The resolve function takes two clauses, finds complementary literals, and removes them to produce resolvents.
3. **Resolve All Function**: The resolve_all function generates all possible resolvents from the current set of clauses.
4. **Resolution Function**: The resolution function uses a loop to continuously generate new resolvents until no more new clauses are found or an empty clause is found (indicating unsatisfiability).

This implementation reads the knowledge base, parses it into individual clauses, and then performs resolution to determine if the knowledge base is satisfiable. The example knowledge base provided will be processed, and the result will be printed as either "Satisfiable" or "Unsatisfiable".

Program 9
Implement Tic-Tac-Toe game using Python

Program:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': '
'}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
    printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
    move = input()
    theBoard[move] = turn
    if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
printBoard(theBoard)
```

Output:

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
top-M
 |X|
-+-+-
 | |
-+-+-
 | |
Turn for O. Move on which space?
mid-L
 |X|
-+-+-
O| |
-+-+-
 | |
Turn for X. Move on which space?
low-L
 |X|
-+-+-
O| |
-+-+-
X| |
Turn for O. Move on which space?
mid-R
```

```
 |X|
-+-+-
O| |O
-+-+-
X| |
Turn for X. Move on which space?
mid-M
 |X|
-+-+-
O|X|O
-+-+-
X| |
Turn for O. Move on which space?
low-M
 |X|
-+-+-
O|X|O
-+-+-
X|O|
Turn for X. Move on which space?
top-R
 |X|X
-+-+-
O|X|O
-+-+-
X|O|
Turn for O. Move on which space?
low-R
 |X|X
-+-+-
O|X|O
-+-+-
X|O|O
Turn for X. Move on which space?
top-L
X|X|X
-+-+-
O|X|O
-+-+-
X|O|O
```

Another way:

```
def print_board(board):
    """ Print the current state of the Tic-Tac-Toe board """
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board, player):
    """ Check if the specified player has won the game """
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2-i] == player for i in range(3)):
        return True
    return False
```

```
def is_full(board):
    """ Check if the board is completely filled """
    return all(cell != ' ' for row in board for cell in row)

def tic_tac_toe():
    """ Main function to run the Tic-Tac-Toe game """
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    while True:
        print_board(board)
        print(f"Player {current_player}'s turn.")
        row = int(input("Enter row (1-3): "))
        col = int(input("Enter column (1-3): "))
        row -= 1
        col -= 1
        if board[row][col] == ' ':
            board[row][col] = current_player
        else:
            print("Invalid move! Try again.")
            continue

        # Check if the current player has won
        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break

        # Check if the board is full (tie game)
        if is_full(board):
            print_board(board)
            print("It's a tie!")
            break
        # Switch to the other player
        current_player = 'O' if current_player == 'X' else 'X'

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

Output:

```
  |   |
---------
  |   |
---------
  |   |
---------
Player X's turn.
Enter row (1-3): 1
Enter column (1-3): 3
  |   | X
---------
  |   |
---------
  |   |
---------
Player O's turn.
Enter row (1-3): 2
Enter column (1-3): 1
  |   | X
---------
O |   |
---------
  |   |
---------
Player X's turn.
Enter row (1-3): 2
Enter column (1-3): 2
  |   | X
---------
O | X |
---------
  |   |
```

```
Player O's turn.
Enter row (1-3): 1
Enter column (1-3): 2
  | O | X
---------
O | X |
---------
  |   |
---------
Player X's turn.
Enter row (1-3): 3
Enter column (1-3): 1
  | O | X
---------
O | X |
---------
X |   |
---------
Player X wins!
```

Program 10
Build a bot which provides all the information related to text in search box

Theory:

The bot performs a Google search based on the user's input and retrieves the search results. You can extend this code further to include more advanced features such as processing the search results for specific types of information, implementing natural language processing for understanding user queries better, and integrating with other APIs for additional functionalities.

To see the results of the above program, you can simply run the script in your Python environment and follow the instructions. Here's how you can do it:

- ✓ Copy the provided code into a Python file (e.g., search_bot.py).
- ✓ Save the file.
- ✓ Open a terminal or command prompt.
- ✓ Navigate to the directory where you saved the Python file.
- ✓ Run the Python script by typing python search_bot.py and pressing Enter.
- ✓ The program will prompt you to enter a search query. Type your query and press Enter.
- ✓ The program will then fetch the search results from Google and display them in the terminal.

Program:

```
import requests
from bs4 import BeautifulSoup

class SearchBot:
    def __init__(self):
        pass

    def search(self, query):
        # Perform a search query using Google
        search_url = f"https://www.google.com/search?q={query}"
        response = requests.get(search_url)

        # Parse the HTML content
        soup = BeautifulSoup(response.text, 'html.parser')

        # Extract search results
        search_results = []
        for result in soup.find_all('div', class_='BNeawe vvjwJb AP7Wnd'):
            search_results.append(result.get_text())

        return search_results

# Example usage
if __name__ == "__main__":
    bot = SearchBot()
    query = input("Enter your search query: ")
    results = bot.search(query)
    for idx, result in enumerate(results, 1):
        print(f"{idx}. {result}\n")
```

Output:

```
Enter your search query: Synonym for Chatbot
1. Chatbot Definition, Types, Pros & Cons, Examples - Investopedia

2. 161 Humanlike Conversational AI Synonyms - Chatbots.org

3. Synonyms for Chatbot - Power Thesaurus

4. 100+ Synonyms & Antonyms for CHATBOT - Power Thesaurus

5. {total_synonyms} Humanlike conversational AI synonyms

6. Conversational AI and Chatbots are not synonyms - LinkedIn

7. How could a chat bot learn synonyms? - AI Stack Exchange

8. Synonyms - AI chatbot trained on your data, with Human Agent ...

9. Chatbot synonyms, Chatbot antonyms - FreeThesaurus.com

10. 130 different synonyms are used to refer to 'chat bot', claims ...
```

Program 11
Implement any Game and demonstrate the Game playing strategies

Program:

```
import random
class GuessTheNumberGame:
    def __init__(self, max_attempts=5, min_num=1, max_num=100):
        self.min_num = min_num
        self.max_num = max_num
        self.secret_number = random.randint(min_num, max_num)
        self.max_attempts = max_attempts
        self.attempts = 0

    def play(self):
        print("Welcome to Guess the Number Game!")
        print(f"I'm thinking of a number between {self.min_num} and {self.max_num}.")

        while self.attempts < self.max_attempts:
            guess = self.get_guess()
            if guess == self.secret_number:
                print(f"Congratulations! You guessed the number {self.secret_number} correctly!")
                break
            elif guess < self.secret_number:
                print("Too low! Try again.")
            else:
                print("Too high! Try again.")
            self.attempts += 1
        else:
            print(f"Sorry, you've run out of attempts! The correct number was {self.secret_number}.")

    def get_guess(self):
        while True:
            try:
                guess = int(input(f"Guess the number ({self.min_num} - {self.max_num}): "))
                if self.min_num <= guess <= self.max_num:
                    return guess
                else:
                    print(f"Please enter a number between {self.min_num} and {self.max_num}.")
            except ValueError:
                print("Please enter a valid number.")

# Demonstration
if __name__ == "__main__":
    game = GuessTheNumberGame()
    game.play()
```

Result:

Output 1:

```
Welcome to Guess the Number Game!
I'm thinking of a number between 1 and 100.
Guess the number (1 - 100): 40
Too low! Try again.
Guess the number (1 - 100): 60
Too low! Try again.
Guess the number (1 - 100): 80
Too high! Try again.
Guess the number (1 - 100): 65
Too low! Try again.
Guess the number (1 - 100): 75
Too low! Try again.
Sorry, you've run out of attempts! The correct number was 77.
```

Output 2:

```
Welcome to Guess the Number Game!
I'm thinking of a number between 1 and 100.
Guess the number (1 - 100): 45
Too low! Try again.
Guess the number (1 - 100): 80
Too high! Try again.
Guess the number (1 - 100): 54
Too high! Try again.
Guess the number (1 - 100): 49
Congratulations! You guessed the number 49 correctly!
```