SRM INSTITUTE OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

18CSC305J-ARTIFICIAL INTELLIGENCE

SEMESTER – 6

K2 SECTION BATCH-2

REGISTRATION NUMBER	RA1911031010057
NAME	JOITA MITRA

B.TECH (CSE-IT)

FACULTY INCHARGE:
DR. V.M. GAYATHRI,
ASSISTANT PROFESSOR,
DEPARTMENT OF NETWORKING
AND COMMUNICATION

INDEX

Exercise: 1
Date : 28-01-2022

TOY PROBLEM

Problem Statement:
Given an integer array **nums**, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Algorithm :
Given an array nums[], write a program to find the maximum sum of a subarray among all subarrays,

1. Initialize an empty array, dp[]
2. Append the first element of nums[] in dp[]
3. Set the current_largest_sum to dp[0]
4. Loop through the entirety of array nums[] starting from the second element
5. Calculate the maximum between nums[i] and dp[i-1] + nums[i] and append it to the current index of dp[]re
6. Compare the current_largest_sum with dp[i] and store the maximum in current_largest_sum
7. Repeat steps 5 and 6 till you reach the end of the loop
8. Return the maximum sum stored in current_largest_sum

Optimization Technique :
This problem can be solved by using the optimization technique of Dynamic Programming. Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.

The approach followed in the above algorithm is as follows:

- We look for all positive contiguous segments of the array and keep track of the maximum sum contiguous segment among all positive segments. Each time we get a positive-sum, compare it with the maximum sum and update it accordingly.

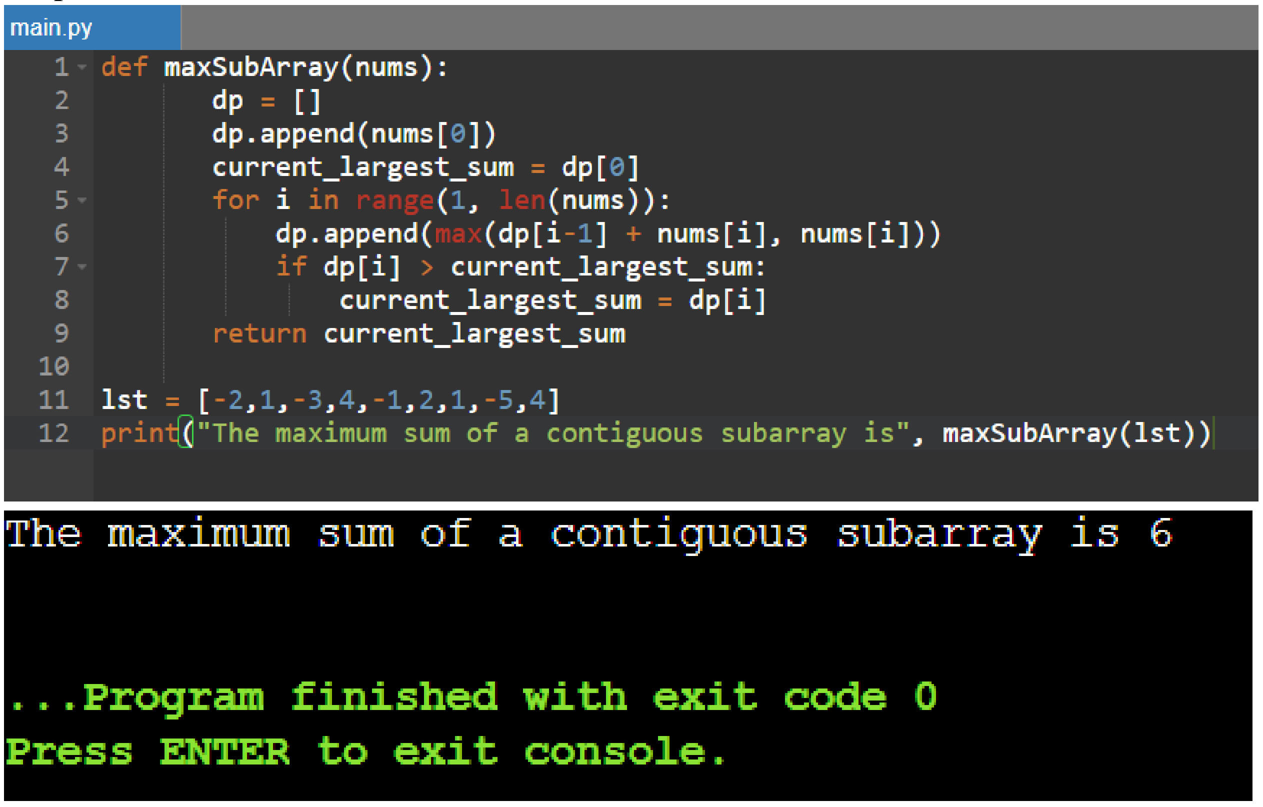
- If the array contains all non-negative numbers, the maximum subarray sum would be the sum of the entire array.

- Several different sub-arrays may have the same max sum but we need to just return the value of the max subarray sum.

Program:

```
class Solution:
  def maxSubArray(nums):
      dp = []
      dp.append(nums[0])
      current_largest_sum = dp[0]
      for i in range(1, len(nums)):
          dp.append(max(dp[i-1] + nums[i], nums[i]))
          if dp[i] > current_largest_sum:
              current_largest_sum = dp[i]
      return current_largest_sum
lst = [-2,1,-3,4,-1,2,1,-5,4]
print(maxSubArray(lst))
```

Output:



```
def maxSubArray(nums):
        dp = []
        dp.append(nums[0])
        current_largest_sum = dp[0]
        for i in range(1, len(nums)):
            dp.append(max(dp[i-1] + nums[i], nums[i]))
            if dp[i] > current_largest_sum:
                current_largest_sum = dp[i]
        return current_largest_sum

lst = [-2,1,-3,4,-1,2,1,-5,4]
print("The maximum sum of a contiguous subarray is", maxSubArray(lst))
```

```
The maximum sum of a contiguous subarray is 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The maximum sum of a contiguous subarray was successfully found, given an array of n elements.

Exercise: 2
Date : 08-02-2022

AGENT PROBLEM

Problem Statement:
Vacuum cleaner problem is a well-known search problem for an agent which works on Artificial Intelligence. In this problem, our vacuum cleaner is our agent. It is a **goal based agent**, and the goal of this agent, which is the vacuum cleaner, is to clean up the whole area.

Problem Explanation:
In the classical vacuum cleaner problem, we have two rooms and one vacuum cleaner. There is dirt in both the rooms and it is to be cleaned. The vacuum cleaner is present in any one of these rooms. There are **eight possible states** possible in our vacuum cleaner problem. We have to reach a state in which both the rooms are clean and are dust free.

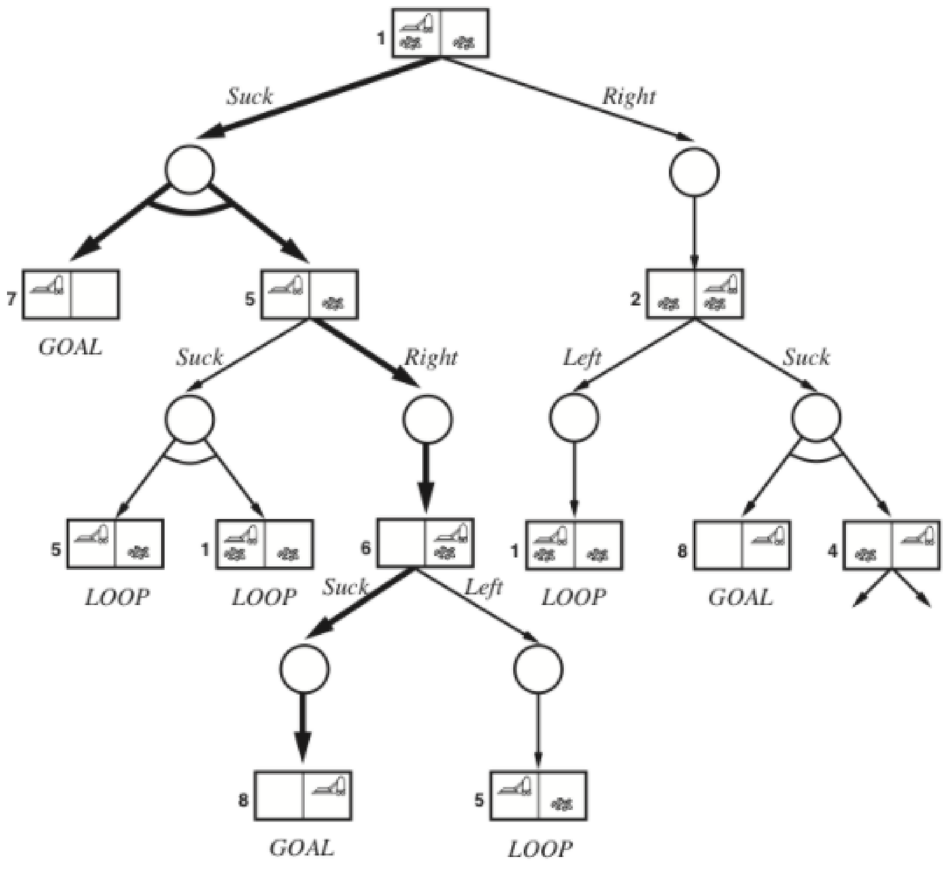
The vacuum cleaner can perform the following functions: move left, move right, move forward, move backward and to suck dust. But as there are only two rooms in our problem, the vacuum cleaner performs only the following functions here: move left, move right and suck.

Problem state space: 2 positions
Dirt or no dirt : 8 problem states
Actions: Left (L), Right (R), or Suck (S)
Goal: no dirt in the rooms
Path costs: one unit per action

Algorithm :
Write a program to find the performance measurement in the vacuum cleaner problem given a number of problem states.

1. Initialize goal_state to {'A':'0','B':'0'} where 0 indicates Clean and 1 indicates Dirty.
2. Initialize cost to 0.
3. Accept the input from the user namely location of vacuum and status of rooms.
4. If the location with vacuum is A, check the status of both A and B, modify the state and cost accordingly.
5. Else the location with vacuum is B, check the status of the rooms, modify the state and cost.
6. Display the goal state and the performance measurement stored in cost.

State Space Tree:



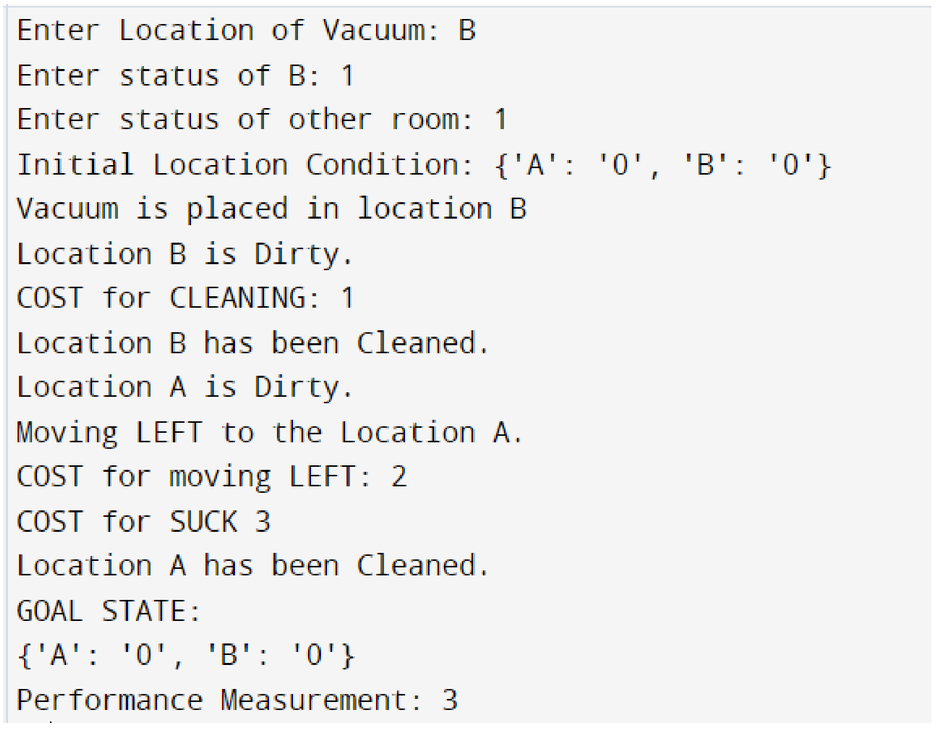
Here, states 1 and 2 are our initial states and state 7 and state 8 are our final states (goal states).

Program:

```
#Enter LOCATION A/B in capital letters
#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum_world():
      # initializing goal_state
      # 0 indicates Clean and 1 indicates Dirty
   goal_state = {'A': '0', 'B': '0'}
   cost = 0

   location_input = input("Enter Location of Vacuum: ") #user_input of location vacuum is placed
   status_input = input("Enter status of " + location_input +": ") #user_input if location is dirty or
clean
   status_input_complement = input("Enter status of other room: ")
   print("Initial Location Condition: " + str(goal_state))

   if location_input == 'A':
    # Location A is Dirty.
    print("Vacuum is placed in Location A")
    if status_input == '1':
        print("Location A is Dirty.")
        # suck the dirt  and mark it as clean
        goal_state['A'] = '0'
        cost += 1                      #cost for suck
        print("Cost for CLEANING A: " + str(cost))
        print("Location A has been Cleaned.")

        if status_input_complement == '1':
            # if B is Dirty
            print("Location B is Dirty.")
            print("Moving right to the Location B. ")
            cost += 1                       #cost for moving right
            print("COST for moving RIGHT: " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                       #cost for suck
            print("COST for SUCK: " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':# if B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                       #cost for moving right
            print("COST for moving RIGHT: " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                       #cost for suck
            print("Cost for SUCK: " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.
```

```
        if status_input == '1':
            print("Location B is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['B'] = '0'
            cost += 1  # cost for suck
            print("COST for CLEANING: " + str(cost))
            print("Location B has been Cleaned.")

            if status_input_complement == '1':
                # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT: " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("COST for SUCK " + str(cost))
                print("Location A has been Cleaned.")

        else:
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

            if status_input_complement == '1':  # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT: " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("Cost for SUCK: " + str(cost))
                print("Location A has been Cleaned. ")
            else:
                print("No action " + str(cost))
                # suck and mark clean
                print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

Output:



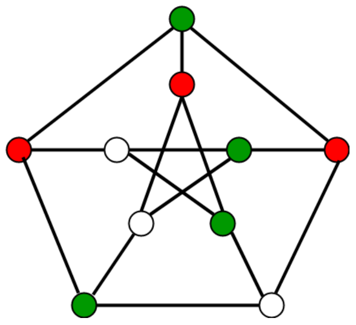
```
Enter Location of Vacuum: B
Enter status of B: 1
Enter status of other room: 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING: 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT: 2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Result:
Hence, using the vacuum cleaner problem solution we have we reached a state in which both the rooms are clean and are dust free.

Exercise: 3
Date : 15-02-2022

CONSTRAINT SATISFACTION PROBLEM

Problem Statement:
Given an undirected graph and a number m, determine if the graph can be coloured with at most m colors such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices. Following is an example of a graph that can be coloured with 3 different colors



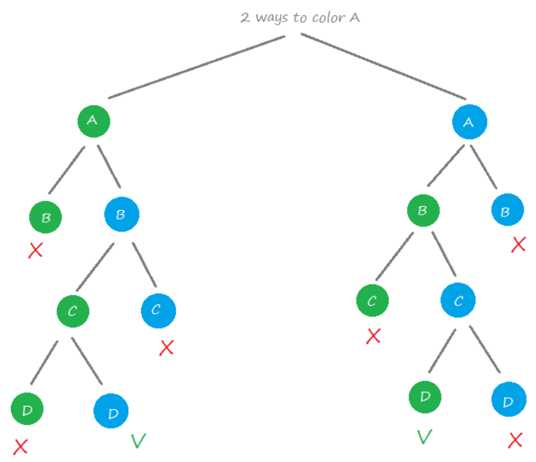
Problem Analysis:

- A graph represented in 2D array format of size V * V where V is the number of vertices in the graph and the 2D array is the adjacency matrix representation and the value of graph[i][j] is 1 if there is a direct edge from i to j, otherwise the value is 0.
- An integer m that denotes the maximum number of colors which can be used in graph coloring.

Algorithm:

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in the output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true, break the loop and return true.
6. If no recursive function returns true then return false.

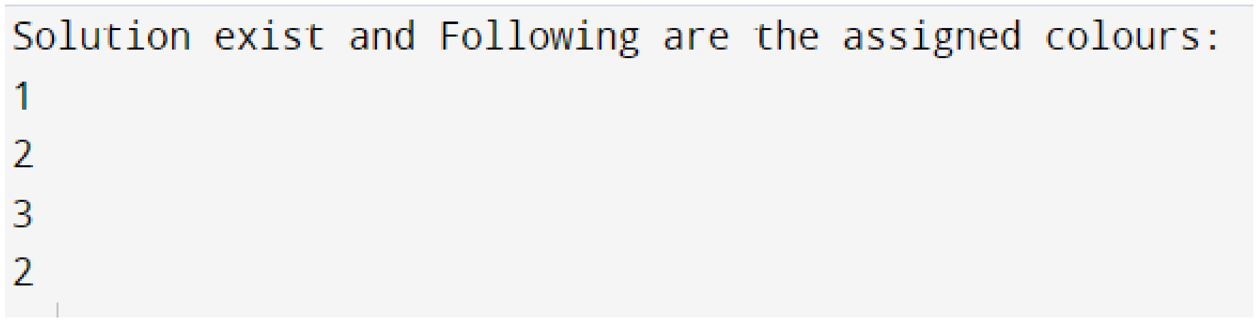
State Space Tree:



Program:

```
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
    def isSafe(self, v, colour, c):
        for i in range(self.V):
            if self.graph[v][i] == 1 and colour[i] == c:
                return False
        return True
def graphColourUtil(self, m, colour, v):
    if v == self.V:
       return True

    for c in range(1, m+1):
       if self.isSafe(v, colour, c) == True:
          colour[v] = c
       if self.graphColourUtil(m,colour,v+1)==True:
          return True
       colour[v] = 0
  def graphColouring(self, m):
        colour = [0] * self.V
        if self.graphColourUtil(m, colour, 0) == False:
            return False
        # Print the solution
        print("Solution exist and Following are the assigned colours:")
        for c in colour:
            print(c)
        return True
g  = Graph(4)
g.graph = [[0,1,1,1], [1,0,1,0], [1,1,0,1], [1,0,1,0]]
m=3
g.graphColouring(m)
```

Output:



```
Solution exist and Following are the assigned colours:
1
2
3
2
```

Complexity:
Time Complexity: O(M^V), in the worst case.
Space Complexity: O(V), as extra space is used for coloring vertices.

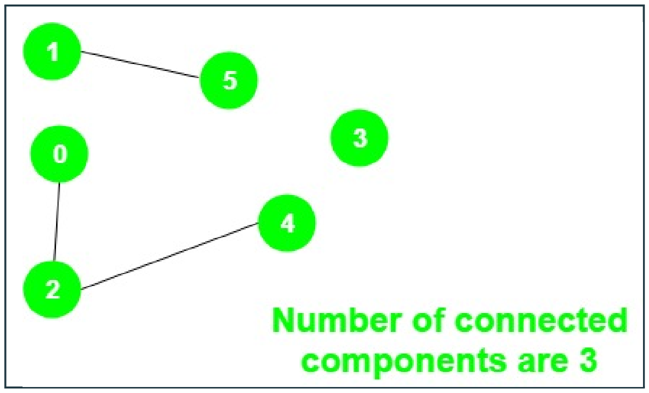
Result:
The program was executed successfully and it returned the colors assigned to the vertices.

Exercise: 4
Date : 22-02-2022

DFS AND BFS

Problem Statement:
Given an undirected graph **g**, the task is to print the number of connected components in the graph using DFS and BFS.

Input:



Output:
3
There are three connected components:
1 – 5, 0 – 2 – 4 and 3

Problem Analysis:

- Both DFS and BFS can be used to solve the given problem i.e. to count the number of connected components in an undirected graph.
- DFS: Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.
- BFS: Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes layerwise. Then, it move towards the next-level neighbour nodes and explores all the unexplored nodes.

Problem Solution Using DFS

Algorithm:

```
1. Initialise all vertices as not visited and variable count as 0.
2. Do the following for every vertex 'v'.
    (a) If 'v' is not visited before, call DFSUtil(v)
    (b) Print new line character

3. In DFSUtil(v), mark 'v' as visited.
4. Do the following for every adjacent vertex 'u' of 'v'.
5. If 'u' is not visited, then recursively call DFSUtil(u)
```

6. Increment variable count by 1.
7. Return the final value of count once all nodes have been visited.

Program:
```
class Graph:
        def __init__(self, V):
                self.V = V
                self.adj = [[] for i in range(self.V)]

        def NumberOfconnectedComponents(self):
                visited = [False for i in range(self.V)]
                count = 0
                for v in range(self.V):
                        if (visited[v] == False):
                                self.DFSUtil(v, visited)
                                count += 1
                return count

        def DFSUtil(self, v, visited):
                visited[v] = True
                for i in self.adj[v]:
                        if (not visited[i]):
                                self.DFSUtil(i, visited)

        def addEdge(self, v, w):
                self.adj[v].append(w)
                self.adj[w].append(v)

# Driver code
if __name__=='__main__':

        g = Graph(5)
        g.addEdge(1, 0)
        g.addEdge(2, 3)
        g.addEdge(3, 4)

        print("No. of connected components:",g.NumberOfconnectedComponents())
```

<u>Problem Solution Using BFS</u>

Algorithm:
Given n nodes labelled from 0 to n - 1 and a list of undirected edges, we execute the following steps:

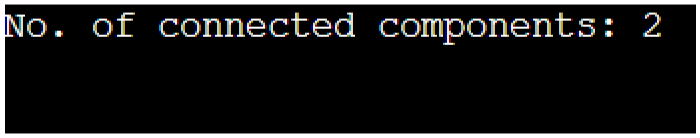
1. Start from index 0 to n.
2. For each index, use BFS to find all it’s related numbers and append them to the visited set.
3. If this index has no more related number then count + 1 and start from the next index.

4. If the index is in the visited set, we skip to the next index.
5. Return the final value of count once all nodes have been visited.

Program:

```
from typing import List
import collections
class Graph:
        def countComponents(self, n: int, edges: List[List[int]]) -> int:
           dist = collections.defaultdict(list)
           for source, target in edges:
              dist[source].append(target)
              dist[target].append(source)
           count = 0
           visited=set()
           queue = collections.deque()
           for x in range(n):
              if x in visited:
                 continue
              queue.append(x)
              while queue:
                 source=queue.popleft()
                 if source in visited:
                    continue
                 visited.add(source)
                 for target in dist[source]:
                    queue.append(target)
              count+=1
           return count

   # Driver code
   if __name__=='__main__':
        edges = [[0, 1],[1, 2],[3, 4]]
        n=5
        g = Graph()
        print("No. of connected components:",g.countComponents(n,edges))
```

Output:



```
No. of connected components: 2
```

Result:
The program was executed successfully and it returned the number of connected components in an undirected graph using DFS and BFS.

Exercise: 5
Date : 08-03-2022

BEST FIRST SEARCH AND A-STAR

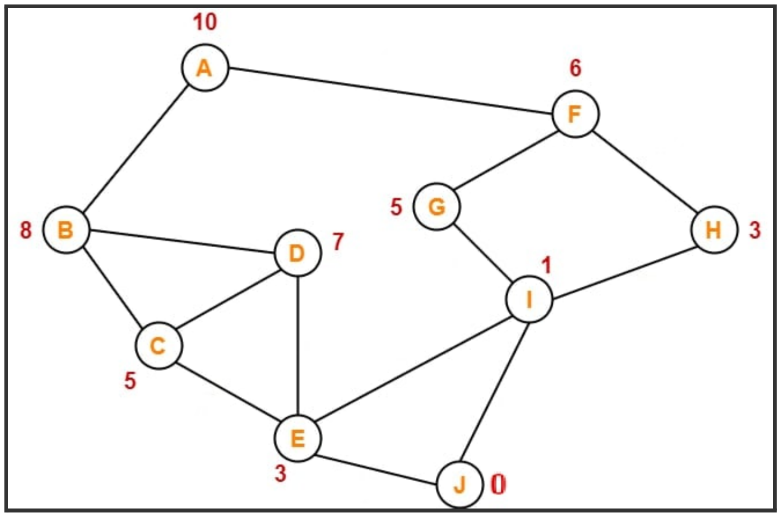
Problem Statement:
Given a graph g, the task is to find the path from source node to goal node using Best First Search and A* algorithms.

Problem Analysis:

- **Best First Search:** Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of DFS and BFS. Here, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e. **f(n)= h(n)**, where, h(n)= estimated cost from node n to the goal. The greedy best first algorithm is implemented by the priority queue.

- **A* Search:** A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search. In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a fitness number, **f(n)= g(n)+h(n)**

Problem Solution Using Best First Search



Node	H(n)
A	10
B	8
C	5

D	7
E	3
F	6
G	5
H	3
I	1
J	0

Calculation:
Expand the nodes of A and put in the CLOSED list

Initialization: Open [B, F], Closed [A]

Iteration 1: Open [B], Closed [A, F]

Iteration 2: Open [G,H,B], Closed [A, F]

: Open [G, B], Closed [A, F, H]

Iteration 3: Open [G, B, I], Closed [A, F, H]

: Open [G, B], Closed [A, F, H, I]

Iteration 4: Open [G, B, J], Closed [A, F, H, I]

: Open [G, B], Closed [A, F, H, I, J]

Hence the final solution path will be: **A----> F----->H---->I---->J**

Program:
```
graph = {
'A':[('B',8), ('F',6)],
'B':[('C',5), ('D',7)],
'C':[('E',3)],
'D':[('E',3)],
'E':[('J',0)],
'F':[('G',5), ('H',3)],
'G':[('I',1)],
'H':[('I',1)],
'I':[('J',0)]
}

def bfs(start, target, graph, queue=[], visited=[]):
    if start not in visited:
        print(start)
        visited.append(start)
    queue=queue+[x for x in graph[start] if x[0][0] not in visited]
    queue.sort(key=lambda x:x[1])
    if queue[0][0]==target:
        print(queue[0][0])
    else:
        processing=queue[0]
        queue.remove(processing)
        bfs(processing[0], target, graph, queue, visited)
bfs('A', 'J', graph)
```

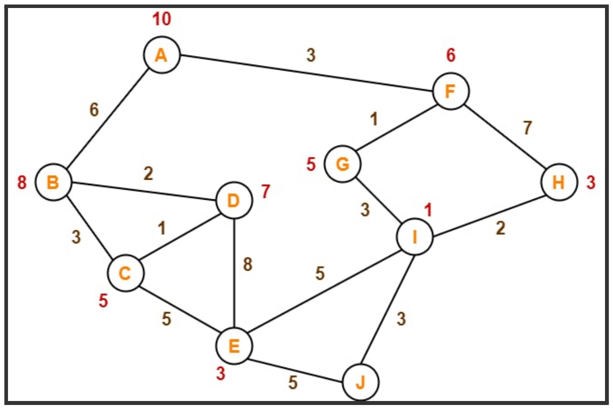
Output:

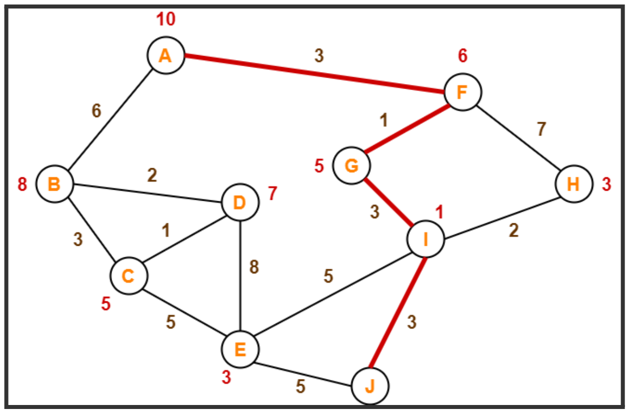
```
A
F
H
I
J
```

Complexity:

- Time Complexity: The worst case time complexity of Greedy best first search is **O(b^m)**.
- Space Complexity: The worst case space complexity of Greedy best first search is **O(b^m)**. Where, m is the maximum depth of the search space.

<u>Problem Solution Using A*</u>





Calculation:

Step-1: We start with node A. Node B and Node F can be reached from node A.

A* Algorithm calculates f(B) and f(F).

- f(B) = 6 + 8 = 14
- f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F.

Path- A → F

Step-2: Node G and Node H can be reached from node F.

A* Algorithm calculates f(G) and f(H).

- f(G) = (3+1) + 5 = 9
- f(H) = (3+7) + 3 = 13

Since f(G) < f(H), so it decides to go to node G.

Path- A → F → G

Step-3: Node I can be reached from node G.

A* Algorithm calculates f(I).

f(I) = (3+1+3) + 1 = 8

It decides to go to node I.

Path- A → F → G → I

Step-04: Node E, Node H and Node J can be reached from node I.

A* Algorithm calculates f(E), f(H) and f(J).

- f(E) = (3+1+3+5) + 3 = 15
- f(H) = (3+1+3+2) + 3 = 12
- f(J) = (3+1+3+3) + 0 = 10

Since f(J) is least, so it decides to go to node J.

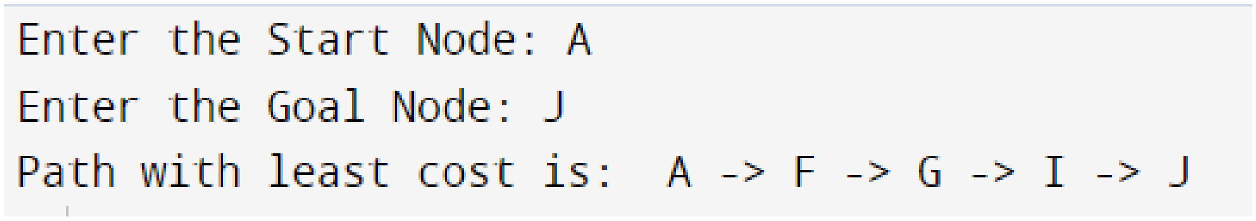
Path- A → F → G → I → J

This is the required shortest path from node A to node J.

Program:

```
graph=[['A','B',6,8],
    ['A','F',3,6],
    ['B','C',3,5],
    ['B','D',2,7],
    ['C','E',5,3],
    ['D','C',1,5],
    ['D','E',8,3],
    ['E','I',5,1],
    ['E','J',5,0],
    ['F','G',1,5],
    ['F','H',7,3],
    ['G','I',3,1],
        ['H','I',2,1],
        ['I','J',3,0]]
temp = []
temp1 = []
for i in graph:
    temp.append(i[0])
    temp1.append(i[1])
nodes = set(temp).union(set(temp1))
def A_star(graph, costs, open, closed, cur_node):
    if cur_node in open:
        open.remove(cur_node)
    closed.add(cur_node)
    for i in graph:
        if(i[0] == cur_node and costs[i[0]]+i[2]+i[3] < costs[i[1]]):
            open.add(i[1])
            costs[i[1]] =  costs[i[0]]+i[2]+i[3]
            path[i[1]] = path[i[0]] + ' -> ' + i[1]
    costs[cur_node] = 999999
    small = min(costs, key=costs.get)
    if small not in closed:
        A_star(graph, costs, open,closed, small)
costs = dict()
temp_cost = dict()
path = dict()
for i in nodes:
    costs[i] = 999999
    path[i] = ' '
open = set()
closed = set()
start_node = input("Enter the Start Node: ")
open.add(start_node)
path[start_node] = start_node
costs[start_node] = 0
A_star(graph, costs, open, closed, start_node)
goal_node = input("Enter the Goal Node: ")
print("Path with least cost is: ",path[goal_node])
```

Output:



```
Enter the Start Node: A
Enter the Goal Node: J
Path with least cost is:  A -> F -> G -> I -> J
```

Complexity:

- Time Complexity: The time complexity of the A* search is **O(b^d)** where b is the branching factor.

- Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

Result:
The program was executed successfully and it returned the path from source node to goal node for a given graph using Best First Search and A* Algorithms.

Exercise: 6
Date : 15-03-2022

MINIMAX ALGORITHM

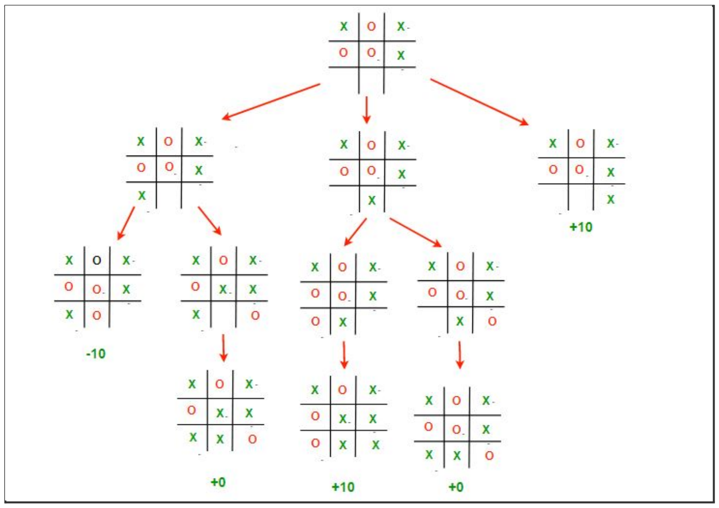
Problem Statement:

We will be implementing the minimax algorithm for Tic-Tac-Toe. The game is to be played between two people. One of the players chooses ‘O’ and the other ‘X’ to mark their respective cells. The game starts with one of the players and the game ends when either of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).

Problem Analysis:
Minimax algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that the opponent is also playing optimally. It uses recursion to search through the game-tree. Here, we will recursively generate the game tree by exploring all possible moves for each board state and upon reaching a terminal state, we will assign a value of :

- 1 for winning,
- -1 for losing
- 0 for draw.

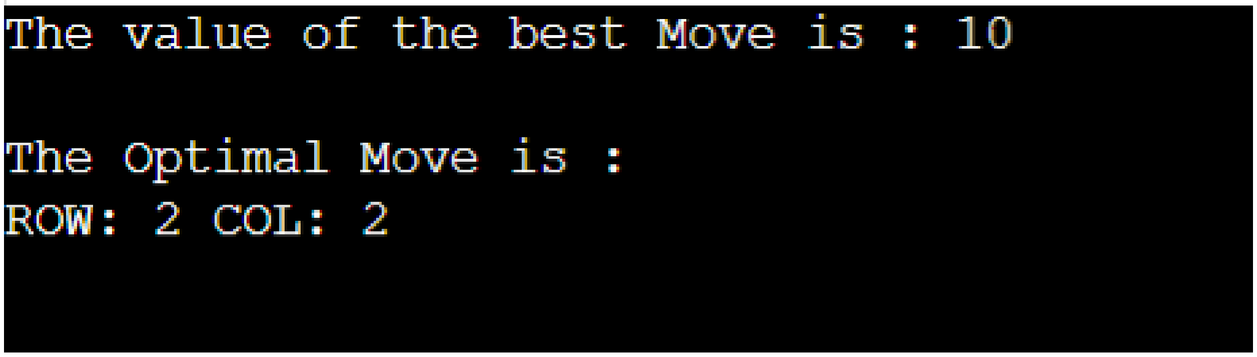
Then based on these terminal states, for each explored turn either maximizer or minimizer will pick the most appropriate move.



Program:

```
player, opponent = 'x', 'o'
def isMovesLeft(board) :
    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False

def evaluate(b) :
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            else if (b[row][0] == opponent) :
                return -10
\
    for col in range(3) :
        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :
            if (b[0][col] == player) :
                return 10
            else if (b[0][col] == opponent) :
                return -10

    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :
        if (b[0][0] == player) :
            return 10
        else if (b[0][0] == opponent) :
            return -10

    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :
        if (b[0][2] == player) :
            return 10
        else if (b[0][2] == opponent) :
            return -10
    return 0

def minimax(board, depth, isMax) :
    score = evaluate(board)

    if (score == 10) :
        return score
    if (score == -10) :
        return score
    if (isMovesLeft(board) == False) :
        return 0
    if (isMax) :
        best = -1000
        for i in range(3) :
            for j in range(3) :
                if (board[i][j]=='_') :
                    board[i][j] = player
                    best = max( best, minimax(board, depth + 1, not isMax) )
                    board[i][j] = '_'
        return best

    else :
        best = 1000
        for i in range(3) :
            for j in range(3) :
                if (board[i][j] == '_') :
                    board[i][j] = opponent
                    best = min(best, minimax(board, depth + 1, not isMax))
                    board[i][j] = '_'
        return best

def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)
    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                board[i][j] = player
                moveVal = minimax(board, 0, False)
                board[i][j] = '_'
                if (moveVal > bestVal) :
                bestMove = (i, j)
                bestVal = moveVal

    print("The value of the best Move is :", bestVal)
    print()
    return bestMove

board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', 'x' ],
    [ '_', '_', '_' ]
]

bestMove = findBestMove(board)
print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

Output:



```
The value of the best Move is : 10

The Optimal Move is :
ROW: 2 COL: 2
```

Complexity:

- **Time complexity**- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is O(b^m), where b is the branching factor of the game-tree, and m is the maximum depth of the tree.

- **Space Complexity**- Space complexity of Minimax algorithm is also similar to DFS which is O(b^m).

Result:
The program was executed successfully and the optimal move was found for the game of tic-tac-toe using Minimax Algorithm.

Exercise: 7
Date : 05-04-2022

UNIFICATION AND RESOLUTION PROBLEMS

Problem Statement:
To make two different logical atomic expressions identical by finding a substitution.

Problem Description:
Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.

- It takes two literals as input and makes them identical using substitution.The basic idea is, can the given terms be made to represent the same structure.
- In other words, by replacing certain sub-expression variables with other expressions, unification tries to identify two symbolic expressions. Unification is used in automated reasoning technology, which remains one of the major application areas of unification.
- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It fails if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

Some basic conditions for unification are:

- Predicate symbols must be the same, atoms or expressions with different predicate symbols can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Algorithm:

Let Ψ1 and Ψ2 be two atomic sentences and σ be a unifier such that, **Ψ1**σ **= Ψ2**σ, then it can be expressed as **UNIFY(Ψ1, Ψ2).**

```
Step. 1: If Ψ1 or Ψ2 is a variable or constant, then:
        a) If Ψ1 or Ψ2 are identical, then return NIL.
        b) Else if Ψ1is a variable,
                a. then if Ψ1 occurs in Ψ2, then return FAILURE
                b. Else return { (Ψ2/ Ψ1)}.
        c) Else if Ψ2 is a variable,
                a. If Ψ2 occurs in Ψ1 then return FAILURE,
                b. Else return {( Ψ1/ Ψ2)}.
        d) Else return FAILURE.
Step.2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return FAILURE.
Step. 3: IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.
Step. 4: Set Substitution set(SUBST) to NIL.
Step. 5: For i=1 to the number of elements in Ψ1.
        a) Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result
        into S.
        b) If S = failure then returns Failure
        c) If S ≠ NIL then do,
                a. Apply S to the remainder of both L1 and L2.
                b. SUBST= APPEND(S, SUBST).
Step.6: Return SUBST.
```

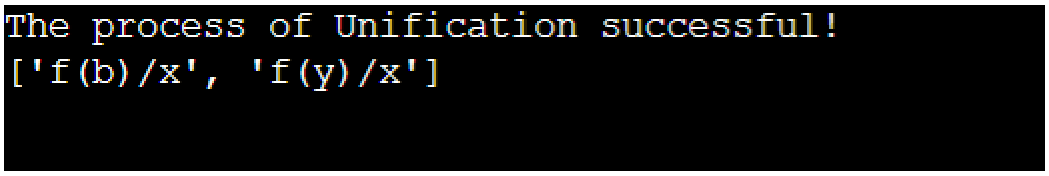
Program:

```
def get_index_comma(string):
  index_list = list()
  par_count = 0
  for i in range(len(string)):
     if string[i] == ',' and par_count == 0:
        index_list.append(i)
     elif string[i] == '(':
        par_count += 1
     elif string[i] == ')':
            par_count -= 1
    return index_list

def is_variable(expr):
    for i in expr:
        if i == '(' or i == ')':
            return False
    return True

def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)
    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])
    return predicate_symbol, arg_list
def get_arg_list(expr):
    _, arg_list = process_expression(expr)
      flag = True
      while flag:
    flag = False
          for i in arg_list:
           if not is_variable(i):
               flag = True
               _, tmp = process_expression(i)
               for j in tmp:
                   if j not in arg_list:
                         arg_list.append(j)
               arg_list.remove(i)
return arg_list

def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
    if var in arg_list:
          return True
    return False
```

```
def unify(expr1, expr2):
    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

        if predicate_symbol_1 != predicate_symbol_2:
            return False
        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:
            sub_list = list()
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])
                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)
            return sub_list
if __name__ == '__main__':
    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
```

```
    print('The process of Unification successful!')
    print(result)
```

Output:



```
The process of Unification successful!
['f(b)/x', 'f(y)/x']
```

Result:
Two different logical atomic expressions were made identical by finding a substitution.

RESOLUTION

Problem Statement:
To resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.

Problem Description:
Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

Example:

- John likes all kind of food.
- Apple and vegetable are food
- Anything anyone eats and not killed is food.
- Anil eats peanuts and still alive
- Harry eats everything that Anil eats.
  Prove by resolution that:
- John likes peanuts.

Algorithm:

- In the first step of resolution graph, **¬likes(John, Peanuts)** , and **likes(John, x)** get resolved(canceled) by substitution of **{Peanuts/x}**, and we are left with **¬ food(Peanuts)**
- In the second step of the resolution graph, **¬ food(Peanuts)** , and **food(z)** get resolved (canceled) by substitution of **{ Peanuts/z}**, and we are left with **¬ eats(y, Peanuts) V killed(y)** .
- In the third step of the resolution graph, **¬ eats(y, Peanuts)** and **eats (Anil, Peanuts)** get

resolved by substitution **{Anil/y}**, and we are left with **Killed(Anil)** .

- In the fourth step of the resolution graph, **Killed(Anil)** and ¬ **killed(k)** get resolve by substitution **{Anil/k}**, and we are left with ¬ **alive(Anil)** .
- In the last step of the resolution graph ¬ **alive(Anil)** and **alive(Anil)** get resolved.

Program:

```
import copy
import time
class Parameter:
  variable_count = 1
  def __init__(self, name=None):
     if name:
        self.type = "Constant"
        self.name = name
     else:
        self.type = "Variable"
        self.name = "v" + str(Parameter.variable_count)
        Parameter.variable_count += 1
  def isConstant(self):
     return self.type == "Constant"
  def unify(self, type_, name):
     self.type = type_
     self.name = name
def resolve(self, queryStack, visited, depth=0):
     if time.time() > self.timeLimit:
        raise Exception
     if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()
        queryPredicateName = negatedQuery.name
        if queryPredicateName not in self.sentence_map:
           return False
        else:
           queryPredicate = negatedQuery
           for kb_sentence in self.sentence_map[queryPredicateName]:
              if not visited[kb_sentence.sentence_index]:
                 for kbPredicate in kb_sentence.findPredicates(queryPredicateName):
                    canUnify, substitution = performUnification(
                       copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))
                    if canUnify:
                       newSentence = copy.deepcopy(kb_sentence)
                       newSentence.removePredicate(kbPredicate)
                       newQueryStack = copy.deepcopy(queryStack)

                       if substitution:
                          for old, new in substitution.items():
                             if old in newSentence.variable_map:
```

```
                                        parameter = newSentence.variable_map[old]
                                        newSentence.variable_map.pop(old)
                                        parameter.unify(
                                            "Variable" if new[0].islower() else "Constant", new)
                                        newSentence.variable_map[new] = parameter
                                    for predicate in newQueryStack:
                                        for index, param in enumerate(predicate.params):
                                            if param.name in substitution:
                                                new = substitution[param.name]
                                                predicate.params[index].unify(
                                                    "Variable" if new[0].islower() else "Constant", new)
                                for predicate in newSentence.predicates:
                                    newQueryStack.append(predicate)


    new_visited = copy.deepcopy(visited)
                                if kb_sentence.containsVariable() and len(kb_sentence.predicates) > 1:
                                    new_visited[kb_sentence.sentence_index] = True
                                if self.resolve(newQueryStack, new_visited, depth + 1):
                                    return True
                  return False
          return True
def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
query.unify("Constant", kb.name)
                else:
                    return False, {}

else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
                else:
                    if kb.name not in substitution:
```

```
                    substitution[kb.name] = query.name
                elif substitution[kb.name] != query.name:
                    return False, {}
   return True, substitution
def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip()
                          for _ in range(noOfSentences)]
        return inputQueries, inputSentences
def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()
if __name__ == '__main__':
   inputQueries_, inputSentences_ = getInput('C:/ai/lab//7-Unification
Resolutiion/Resolution/Input/input_1.txt')
   knowledgeBase = KB(inputSentences_)
   knowledgeBase.prepareKB()
   results_ = knowledgeBase.askQueries(inputQueries_)
   printOutput("output.txt", results_)
```

Input:

```
6
F(Joe)
H(John)
~H(Alice)
~H(John)
G(Joe)
G(Tom)
14
~F(x) | G(x)
~G(x) | H(x)
~H(x) | F(x)
~R(x) | H(x)
~A(x) | H(x)
~D(x,y) | ~H(y)
```

~B(x,y) | ~C(x,y) | A(x)

B(John,Alice)

B(John,Joe)

~D(x,y) | ~Q(y) | C(x,y)

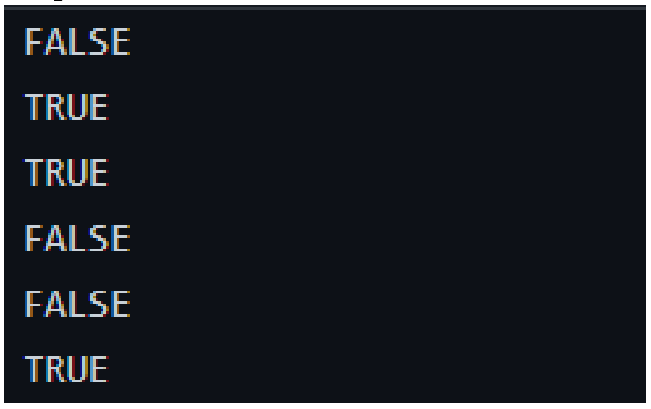
D(John,Alice)

Q(Joe)

D(John,Joe)

R(Tom)

Output:



```
FALSE
TRUE
TRUE
FALSE
FALSE
TRUE
```

Result:
Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements using Resolution in first order logic.

Exercise No: 8
Date : 05-04-2022

MONTY HALL PROBLEM

Problem Statement:
The Monty Hall problem is a famous, seemingly paradoxical problem in conditional probability and reasoning using Bayes' theorem. Information affects your decision that at first glance seems as though it shouldn't.

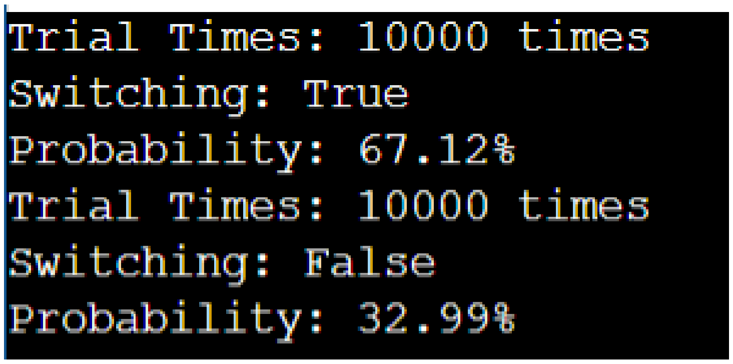
Problem Description:
The Monty Hall problem is a famous, seemingly paradoxical problem in conditional probability and reasoning using Bayes' theorem. Information affects your decision that at first glance seems as though it shouldn't.
In the problem, you are on a game show, being asked to choose between three doors. Behind each door, there is either a car or a goat. You choose a door. The host, Monty Hall, picks one of the other doors, which he knows has a goat behind it, and opens it, showing you the goat. (You know, by the rules of the game, that Monty will always reveal a goat.) Monty then asks whether you would like to switch your choice of door to the other remaining door. Assuming you prefer having a car more than having a goat, do you choose to switch or not to switch?

Algorithm:

1. After you have chosen a door, the door remains closed for the time being.
2. The game show host, Monty Hall, who knows what is behind the doors, now has to open one of the two remaining doors, and the door he opens must have a goat behind it.
3. If both remaining doors have goats behind them, he chooses one randomly.
4. After Monty Hall opens a door with a goat, he will ask you to decide whether you want to stay with your first choice or to switch to the last remaining door.
5. Imagine that you chose Door 1 and the host opens Door 3, which has a goat.
6. He then asks you "Do you want to switch to Door Number 2?

Program:
```
from random import shuffle, choice
door_numbers = [1, 2, 3]
wanted_award = "car"
unwanted_award = "goat"
awards = [wanted_award, unwanted_award, unwanted_award]
def monty_hall_trial(initial_door_number, should_switch):
    shuffle(awards)
    doors = dict(zip(door_numbers, awards))
    remaining_door_numbers = [x for x in door_numbers if x != initial_door_number]
    for door_number in remaining_door_numbers:
        if doors[door_number] == unwanted_award:
            remaining_door_numbers.remove(door_number)
            break
    switched_door_number = remaining_door_numbers[0]
    final_door_number = switched_door_number if should_switch else initial_door_number
    won_car = doors[final_door_number] == wanted_award
    return won_car
def simulate_monty_hall(trial_number, should_switch):
    winning_counts = 0
    for trial_i in range(trial_number):
        initial_pick = choice(door_numbers)
        won_car = monty_hall_trial(initial_pick, should_switch)
        winning_counts += int(won_car)
    winning_prob = winning_counts/trial_number
    print(f"Trial Times: {trial_number} times\n"
          f"Switching: {should_switch}\n"
          f"Probability: {winning_prob:.2%}")

simulate_monty_hall(10000, True)
simulate_monty_hall(10000, False)
```

Output:



```
Trial Times: 10000 times
Switching: True
Probability: 67.12%
Trial Times: 10000 times
Switching: False
Probability: 32.99%
```

Result:

The Monty Hall Problem is solved.