SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY
Kuniamuthur, Coimbatore. Tamil Nadu 641008
(An Autonomous Institution Affiliated to Anna University, Chennai)

SRI KRISHNA INSTITUTIONS COIMBATORE

DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

ACADEMIC YEAR 2023-2024

21CB603 - ARTIFICIAL INTELLIGENCE LABORATORY

Submitted by

Name :

Register No. :

Degree & Branch : B.Tech. Computer Science and Business Systems

Class : III CSBS

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY
Kuniamuthur, Coimbatore. Tamil Nadu 641008
(An Autonomous Institution Affiliated to Anna University, Chennai)

DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

21CB603 - ARTIFICIAL INTELLIGENCE

Continuous Assessment Record

Submitted by

Name : **Register No. :**

Class : III CSBS **Degree & Branch: B.Tech CSBS**

BONAFIDE CERTIFICATE

This is to certify that this is a bonafide record of work done by ____________________________________ (Register No.:_________________________) during the academic year 2023 – 2024.

Faculty In-charge
[Dr.G.Ignisha Rajathi]

HEAD OF THE DEPARTMENT

Submitted for the End Semester practical examination held on ______________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

21CB603 - ARTIFICIAL INTELLIGENCE

Name of the lab In-charge: Dr. G. Ignisha Rajathi

CONTINUOUS EVALUATION SHEET

RUBRICS TABLE

Criteria (100)	Range of Marks			
	Excellent	Good	Average	Below Average
Objective, Algorithm description with sample data (20)	18-20	14-17	10-13	0-9
Coding (30)	28-30	21-27	15-20	0-14
Compilation and Debugging (15)	13-15	10-12	7-9	0-6
Generalization, Execution and Result (15)	13-15	10-12	7-9	0-6
Documentation (10)	9-10	7-8	5-6	0-4
Viva (10)	9-10	7-8	5-6	0-4

INDEX

S.No	Date	Name of the Program	Marks (100)	Signature
Implementation of Uninformed Search Strategy				
1	02.12.2023	Implementation of Breadth First Search		
2	08.12.2023	Implementation of Depth First Search		
3	14.12.2023	Implementation of Uniform Cost Search		
Implementation of Informed Search Strategy				
4	20.12.2023	Implementation of 8-puzzle problem using A* search		
5	19.01.2024	Implementation of Travelling Salesperson Problem Using Hill climbing search		
Advanced Topics				
6	02.02.2024	Study of Prolog		
7	09.02.2024	Implement the Constraint satisfaction problem		
8	15.02.2024	Implement the k-nearest neighbors		
9	22.02.2024	Simple Expert System Using Decision Tree		

Ex No: 01	Implementation of Uninformed Search Strategy Implementation of Breadth First Search
Date:12.12.2023	

Objective:

To implement 8-puzzle problem using the two uninformed search strategies namely breadth first search (BFS)

Algorithm:

BFS:

```
BFS search
function BREADTH-FIRST-SEARCH(initialState, goal Test)
    returns SUCCESS or FAILURE :
    frontier = Queue.new(initialState)
    explored = Set.new
    while not frontier.isEmpty0:
        state = frontier.dequeue()
        explored.add(state)
        if goal Test (state):
            return SUCCESS(state)
        for neighbor in state.neighbors():
            if neighbor not in frontier U explored:
                frontier.enqueue(neighbor)
    return FAILURE
```

Description with sample data:

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

Initial configuration

1	2	3
5	6	
7	8	4

Final Configuration

1	2	3
5	8	6
	7	4

BFS Program:

```
from queue import Queue
import copy
import time
```

```
def printNode(node):
    print(node[0],node[1],node[2])
    print(node[3],node[4],node[5])
    print(node[6],node[7],node[8])
    global nodeNumber
    print('Node:', nodeNumber)
    print('Depth:', len(node[9:]))
    print('Moves:', node[9:])
    print('------')
    nodeNumber += 1

def checkFinal(node):
    if node[:9]==finalNode:
        printNode(node)
        return True
    if node[:9] not in visitedList:
        printNode(node)
        queue.put(node)
        visitedList.append(node[:9])
    return False
if __name__ == '__main__':
    startNode = [1,2,3, 5,6,0, 7,8,4]
    finalNode = [1,2,3, 5,8,6, 0,7,4]

    found = False
    nodeNumber = 0
    visitedList = []
    queue = Queue()
    queue.put(startNode)
    visitedList.append(startNode)
    printNode(startNode)
    t0 = time.time()
    while (not found and not queue.empty()):
        currentNode = queue.get()
        blankIndex = currentNode.index(0)
        if blankIndex!=0 and blankIndex!=1 and blankIndex!=2:
            upNode = copy.deepcopy(currentNode)
            upNode[blankIndex] = upNode[blankIndex-3]
            upNode[blankIndex-3] = 0
            upNode.append('up')
            found = checkFinal(upNode)
        if blankIndex!=0 and blankIndex!=3 and blankIndex!=6 and found==False:
            leftNode = copy.deepcopy(currentNode)
            leftNode[blankIndex] = leftNode[blankIndex-1]
            leftNode[blankIndex-1] = 0
            leftNode.append('left')
```

```
            found = checkFinal(leftNode)
        if blankIndex!=6 and blankIndex!=7 and blankIndex!=8 and found==False:
            downNode = copy.deepcopy(currentNode)
            downNode[blankIndex] = downNode[blankIndex+3]
            downNode[blankIndex+3] = 0
            downNode.append('down')
            found = checkFinal(downNode)
        if blankIndex!=2 and blankIndex!=5 and blankIndex!=8 and found==False:
            rightNode = copy.deepcopy(currentNode)
            rightNode[blankIndex] = rightNode[blankIndex+1]
            rightNode[blankIndex+1] = 0
            rightNode.append('right')
            found = checkFinal(rightNode)
    t1 = time.time()
    print('Time:', t1-t0)
    print('------')
```

Output:

```
1 2 3
5 6 0
7 8 4
Node: 0
Depth: 0
Moves: []
------
1 2 0
5 6 3
7 8 4
Node: 1
Depth: 1
Moves: ['up']
------
1 2 3
5 0 6
7 8 4
Node: 2
Depth: 1
Moves: ['left']
------
1 2 3
5 6 4
7 8 0
Node: 3
Depth: 1
Moves: ['down']
------
1 0 2
```

5 6 3
7 8 4
Node: 4
Depth: 2
Moves: ['up', 'left']
------
1 0 3
5 2 6
7 8 4
Node: 5
Depth: 2
Moves: ['left', 'up']
------
1 2 3
0 5 6
7 8 4
Node: 6
Depth: 2
Moves: ['left', 'left']
------
1 2 3
5 8 6
7 0 4
Node: 7
Depth: 2
Moves: ['left', 'down']
------
1 2 3
5 6 4
7 0 8
Node: 8
Depth: 2
Moves: ['down', 'left']
------
0 1 2
5 6 3
7 8 4
Node: 9
Depth: 3
Moves: ['up', 'left', 'left']
------
1 6 2
5 0 3
7 8 4
Node: 10
Depth: 3

Moves: ['up', 'left', 'down']
------
0 1 3
5 2 6
7 8 4
Node: 11
Depth: 3
Moves: ['left', 'up', 'left']
------
1 3 0
5 2 6
7 8 4
Node: 12
Depth: 3
Moves: ['left', 'up', 'right']
------
0 2 3
1 5 6
7 8 4
Node: 13
Depth: 3
Moves: ['left', 'left', 'up']
------
1 2 3
7 5 6
0 8 4
Node: 14
Depth: 3
Moves: ['left', 'left', 'down']
------
1 2 3
5 8 6
0 7 4
Node: 15
Depth: 3
Moves: ['left', 'down', 'left']

Time: 0.011989152908325195

Result:
The uninformed search using BFS is solved successfully.

Ex: No: 02	Implementation of Depth First Search
Date:08.12.2023	

Objective:

To implement 8-puzzle problem using the two uninformed search strategies namely depth first search (DFS)

Algorithm:

DFS:

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
	returns SUCCESS or FAILURE :
	frontier = Stack.new(initialState)
	explored = Set.new()
	while not frontier.isEmpty(:
		state = frontier.pop()
		explored.add(state)
		if goal Test(state):
			return SUCCESS(state)
		for neighbor in state.neighbors():
			if neighbor not in frontier U explored:
				frontier.push(neighbor)
	return FAILURE
```

Description with sample data:

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

Initial configuration

1	**2**	**5**
3	**4**	**8**
6	**7**	

Final Configuration

	1	**2**
3	**4**	**5**
6	**7**	**8**

Program:
DFS Program:

```
import copy
import time
def printNode(node):
    print(node[0],node[1],node[2])
    print(node[3],node[4],node[5])
    print(node[6],node[7],node[8])
    global nodeNumber
    print('Node:', nodeNumber)
    print('Depth:', len(node[9:]))
    print('Moves:', node[9:])
    print('------')
    nodeNumber += 1
def checkFinal(node):
    if node[:9]==finalNode:
        printNode(node)
        return True
    global insertIndex
    if node[:9] not in visitedList:
        printNode(node)
        stack.insert(insertIndex, node)
        insertIndex += 1
        visitedList.append(node[:9])
    return False
if __name__ == '__main__':
    startNode = [1,2,5, 3,4,8, 6,7,0]
    finalNode = [0,1,2, 3,4,5, 6,7,8]

    found = False
    nodeNumber = 0
    visitedList = []
    stack = []
    stack.append(startNode)
    visitedList.append(startNode)
    printNode(startNode)
    t0 = time.time()
    while (not found and not len(stack)==0):
        currentNode = stack.pop(0)
        blankIndex = currentNode.index(0)
        insertIndex = 0
        if blankIndex!=0 and blankIndex!=1 and blankIndex!=2:
            upNode = copy.deepcopy(currentNode)
            upNode[blankIndex] = upNode[blankIndex-3]
            upNode[blankIndex-3] = 0
```

```
            upNode.append('up')
            found = checkFinal(upNode)
        if blankIndex!=0 and blankIndex!=3 and blankIndex!=6 and found==False:
            leftNode = copy.deepcopy(currentNode)
            leftNode[blankIndex] = leftNode[blankIndex-1]
            leftNode[blankIndex-1] = 0
            leftNode.append('left')
            found = checkFinal(leftNode)
        if blankIndex!=6 and blankIndex!=7 and blankIndex!=8 and found==False:
            downNode = copy.deepcopy(currentNode)
            downNode[blankIndex] = downNode[blankIndex+3]
            downNode[blankIndex+3] = 0
            downNode.append('down')
            found = checkFinal(downNode)
        if blankIndex!=2 and blankIndex!=5 and blankIndex!=8 and found==False:
            rightNode = copy.deepcopy(currentNode)
            rightNode[blankIndex] = rightNode[blankIndex+1]
            rightNode[blankIndex+1] = 0
            rightNode.append('right')
            found = checkFinal(rightNode)
    t1 = time.time()
    print('Time:', t1-t0)
    print('------')
```

Output:

```
1 2 5
3 4 8
6 7 0
Node: 0
Depth: 0
Moves: []
------
1 2 5
3 4 0
6 7 8
Node: 1
Depth: 1
Moves: ['up']
------
1 2 5
3 4 8
6 0 7
Node: 2
Depth: 1
Moves: ['left']
------
1 2 0
```

3 4 5
6 7 8
Node: 3
Depth: 2
Moves: ['up', 'up']
------
1 2 5
3 0 4
6 7 8
Node: 4
Depth: 2
Moves: ['up', 'left']
------
1 0 2
3 4 5
6 7 8
Node: 5
Depth: 3
Moves: ['up', 'up', 'left']
------
0 1 2
3 4 5
6 7 8
Node: 6
Depth: 4
Moves: ['up', 'up', 'left', 'left']

Time: 0.00890064239501953

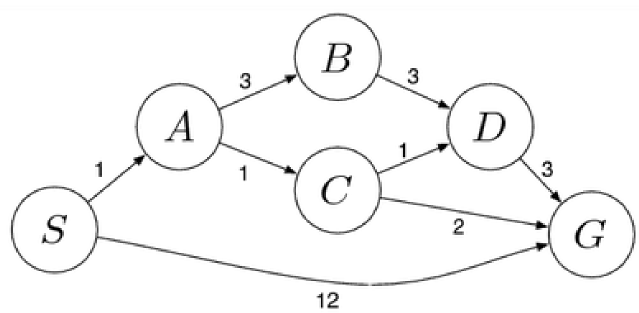
Result:

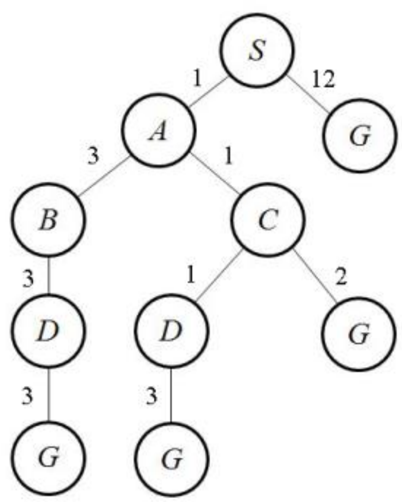
The uninformed search using DFS is solved successfully.

Ex No: 03	Implementation of Uniform Cost Search
14.12.2023	

OBJECTIVE:
To implement and get the desirable output of Uniform Cost Search (UCS).

ALGORITHM:





Uniform Cost Search gives the minimum cumulative cost the maximum priority. The algorithm using this priority queue is the following:

Insert the root into the queue
While the queue is not empty
    Dequeue the maximum priority element from the queue
    (If priorities are same, alphabetically smaller path is chosen)
    If the path is ending in the goal state, print the path and exit
    Else
        Insert all the children of the dequeued element, with the cumulative costs as priority

DESCRIPTION:

Each element of the priority queue is written as [path,cumulative cost].
Initialization: { [S , 0] }
Iteration1: { [S->A , 1] , [S->G , 12] }
Iteration2: { [S->A->C , 2] , [S->A->B , 4] , [S->G , 12] }
Iteration3: { [S->A->C->D , 3] , [S->A->B , 4] , [S->A->C->G , 4] , [S->G , 12] }
Iteration4: { [S->A->B , 4] , [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->G , 12] }
Iteration5: { [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->A->B->D , 7] , [S->G , 12] }
Iteration6 gives the final output as S->A->C->G.

PROGRAM:

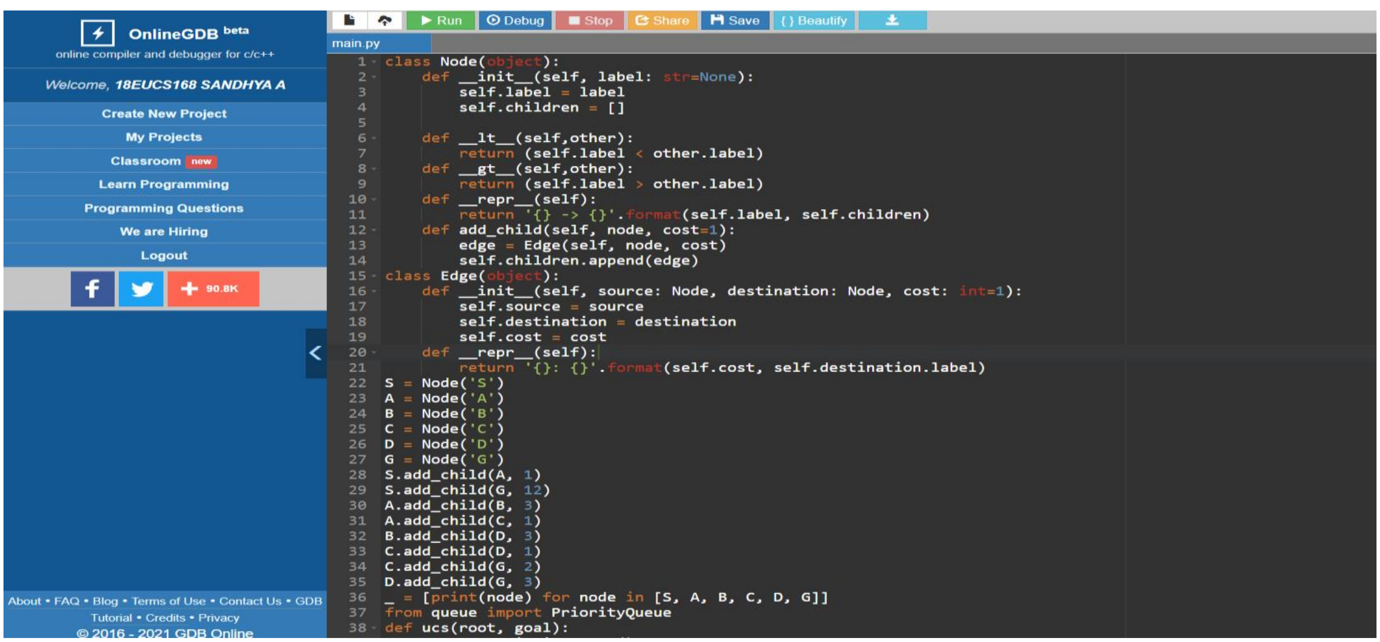
```
class Node(object):
    def __init__(self, label: str=None):
        self.label = label
        self.children = []
    def __lt__(self,other):
        return (self.label < other.label)
    def __gt__(self,other):
        return (self.label > other.label)
    def __repr__(self):
        return '{} -> {}'.format(self.label, self.children)
    def add_child(self, node, cost=1):
        edge = Edge(self, node, cost)
        self.children.append(edge)
class Edge(object):
    def __init__(self, source: Node, destination: Node, cost: int=1):
        self.source = source
        self.destination = destination
        self.cost = cost
    def __repr__(self):
        return '{}: {}'.format(self.cost, self.destination.label)
S = Node('S')
A = Node('A')
B = Node('B')
C = Node('C')
D = Node('D')
G = Node('G')
S.add_child(A, 1)
S.add_child(G, 12)
A.add_child(B, 3)
A.add_child(C, 1)
B.add_child(D, 3)
C.add_child(D, 1)
C.add_child(G, 2)
```

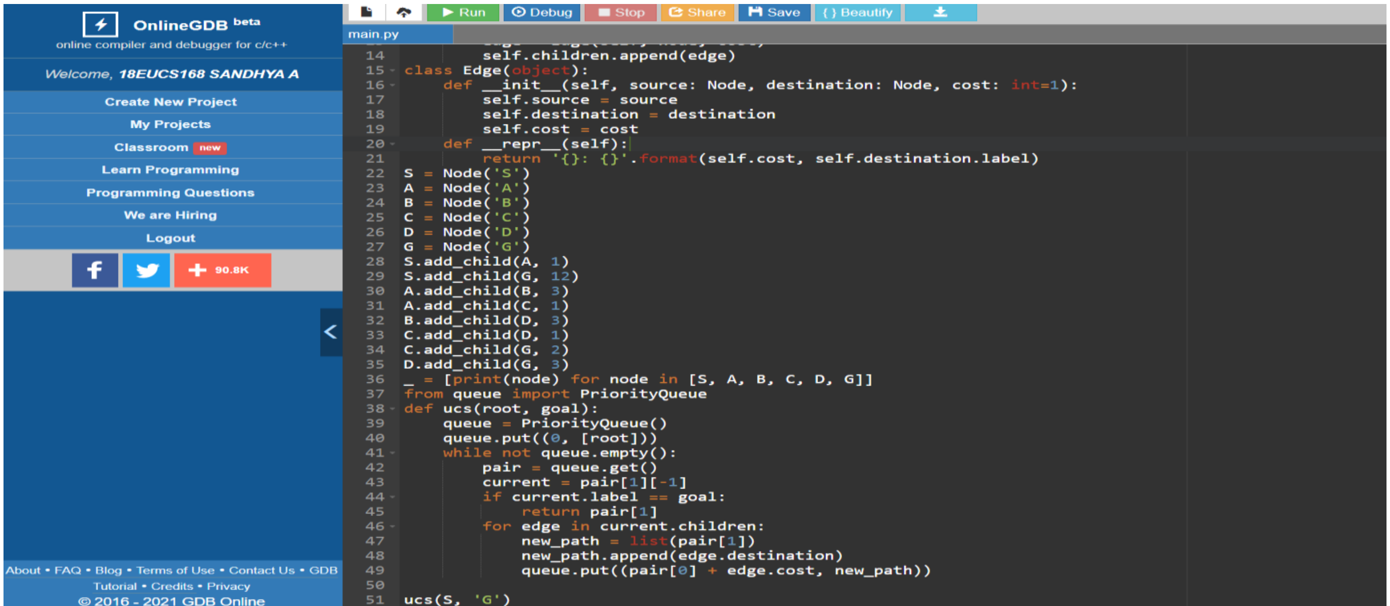
```
D.add_child(G, 3)
_ = [print(node) for node in [S, A, B, C, D, G]]
from queue import PriorityQueue
def ucs(root, goal):
    queue = PriorityQueue()
    queue.put((0, [root]))
    while not queue.empty():
        pair = queue.get()
        current = pair[1][-1]
        if current.label == goal:
            return pair[1]
        for edge in current.children:
            new_path = list(pair[1])
            new_path.append(edge.destination)
            queue.put((pair[0] + edge.cost, new_path))

ucs(S, 'G')
```

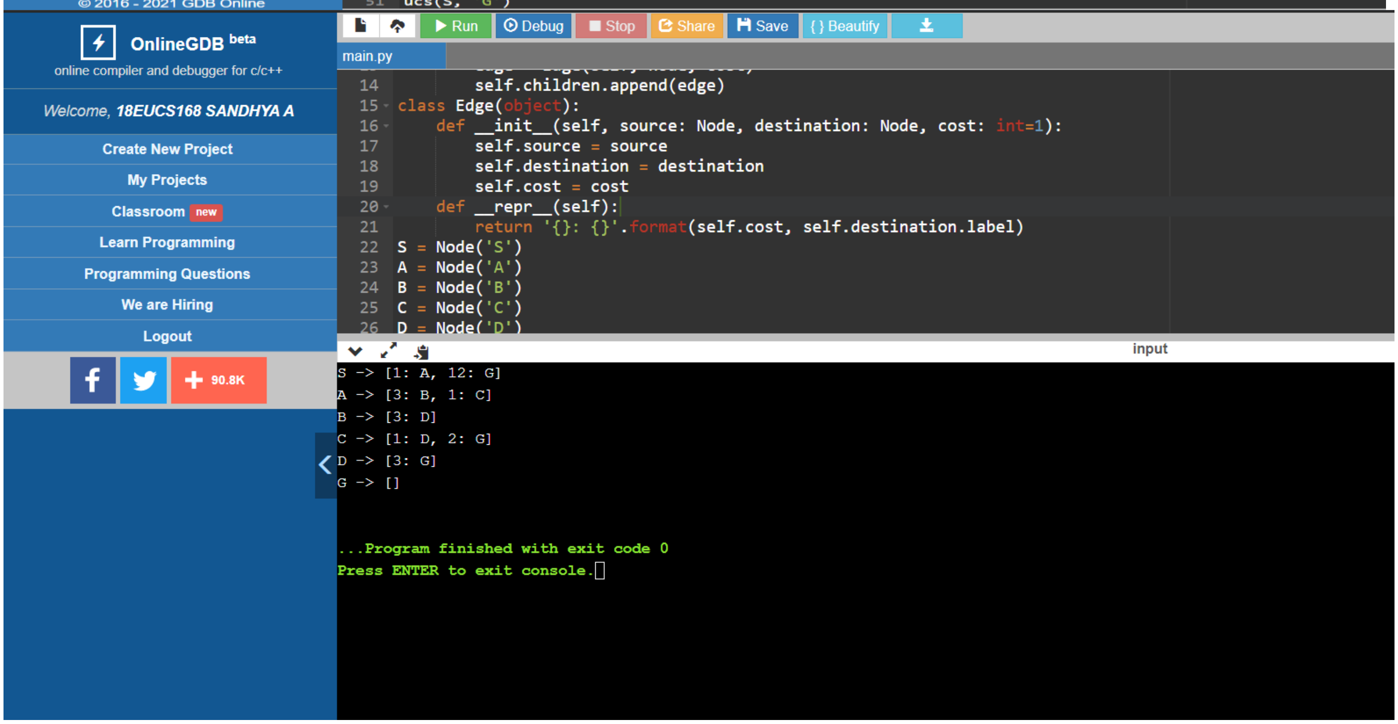
OUTPUT:

```
S -> [1: A, 12: G]
A -> [3: B, 1: C]
B -> [3: D]
C -> [1: D, 2: G]
D -> [3: G]
G-> []
```

SCREENSHOTS:







RESULT

The program is successfully executed and the output is verified

Ex No: 04	**Implementation of Informed Search Strategy** **Implementation of 8-puzzle problem using A* search**
Date:20.12.2023	

Objective:

To solve 8 puzzle problem using A* algorithm.

Algorithm:

- The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.
- OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.

The algorithm is as follows-

Step-01:

- Define a list OPEN.
- Initially, OPEN consists solely of a single node, the start node S.

Step-02:

If the list is empty, return failure and exit.

Step-03:

- Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
- If node n is a goal state, return success and exit.

Step-04:

Expand node n

Step-05:

- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
- Otherwise, go to Step-06.

Step-06:

For each successor node,

- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

Step-07:

Go back to Step-02.

Description with sample data:

A* Algorithm works as-

It maintains a tree of paths originating at the start node.

It extends those paths one edge at a time.

It continues until its termination criterion is satisfied.

A* Algorithm extends the path that minimizes the following function-

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

Here,
'n' is the last node on the path
g(n) is the cost of the path from start node to node 'n'
h(n) is a heuristic function that estimates cost of the cheapest path from node 'n' to the goal node.

Sample Data:
Given an initial state of a 8-puzzle problem and final state to be reached-

2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State

Find the most cost-effective path to reach the final state from initial state using A* Algorithm.
Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

Program:

```
class Node:    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval
    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children
    def shuffle(self,puz,x1,y1,x2,y2):
        """ Move the blank space in the given direction and if the position value are out
```

```
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None
def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j
class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []
    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

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

    def h(self,start,goal):
        """ Calculates the different between the given puzzles """
```

```
        temp = 0
        for i in range(0,self.n):
            for j in range(0,self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp

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

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

            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)
puz = Puzzle(3)
puz.process()
```

Output:

```
Enter the start state matrix

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

1 2 3
4 5 6
7 8 _



   |
   |
  \'/

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

```
  \'/

1 2 3
4 _ 6
7 5 8

   |
   |
  \'/

1 2 3
4 5 6
7 _ 8

   |
   |
  \'/

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

Result:

Thus 8 puzzle problem was solved successfully using A* algorithm.

Ex No: 05	Implementation of Travelling Salesperson Problem Using Hill climbing search
Date:19.01.2024	

Objective:

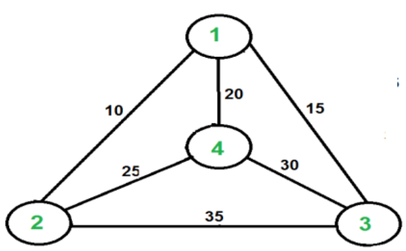
To solve Travelling Salesman Problem (TSP) using Hill climbing Algorithm.

Algorithm:

Step 1 : Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.
Step 2 : Loop until the solution state is found or there are no new operators present which can be applied to the current state.
a) Select a state that has not been yet applied to the current state and apply it to produce a new state.
b) Perform these to evaluate new state
   i. If the current state is a goal state, then stop and return success.
   ii. If it is better than the current state, then make it current state and proceed further.
   iii. If it is not better than the current state, then continue in the loop until a solution is found.
Step 3 : Exit.

Description with sample data:

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
For example, consider the following graph



A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

Program:

```
# Import libraries
import random
import copy
# This class represent a state
class State:
    # Create a new state
    def __init__(self, route:[], distance:int=0):
        self.route = route
        self.distance = distance
    # Compare states
```

```
    def __eq__(self, other):
        for i in range(len(self.route)):
            if(self.route[i] != other.route[i]):
                return False
        return True
    # Sort states
    def __lt__(self, other):
         return self.distance < other.distance
    # Print a state
    def __repr__(self):
        return ('({0},{1})\n'.format(self.route, self.distance))
    # Create a shallow copy
    def copy(self):
        return State(self.route, self.distance)
    # Create a deep copy
    def deepcopy(self):
        return State(copy.deepcopy(self.route), copy.deepcopy(self.distance))
    # Update distance
    def update_distance(self, matrix, home):

        # Reset distance
        self.distance = 0
        # Keep track of departing city
        from_index = home
        # Loop all cities in the current route
        for i in range(len(self.route)):
            self.distance += matrix[from_index][self.route[i]]
            from_index = self.route[i]
        # Add the distance back to home
        self.distance += matrix[from_index][home]
# This class represent a city (used when we need to delete cities)
class City:
    # Create a new city
    def __init__(self, index:int, distance:int):
        self.index = index
        self.distance = distance
    # Sort cities
    def __lt__(self, other):
         return self.distance < other.distance
# Get the best random solution from a population
def get_random_solution(matrix:[], home:int, city_indexes:[], size:int, use_weights=False):
    # Create a list with city indexes
    cities = city_indexes.copy()
    # Remove the home city
    cities.pop(home)
    # Create a population
```

```
    population = []
    for i in range(size):
        if(use_weights == True):
            state = get_random_solution_with_weights(matrix, home)
        else:
            # Shuffle cities at random
            random.shuffle(cities)
            # Create a state
            state = State(cities[:])
            state.update_distance(matrix, home)
        # Add an individual to the population
        population.append(state)
    # Sort population
    population.sort()
    # Return the best solution
    return population[0]

# Get a random solution by using weights
def get_random_solution_with_weights(matrix:[], home:int):

    # Variables
    route = []
    from_index = home
    length = len(matrix) - 1
    # Loop until route is complete
    while len(route) < length:
         # Get a matrix row
        row = matrix[from_index]
        # Create a list with cities
        cities = {}
        for i in range(len(row)):
            cities[i] = City(i, row[i])
        # Remove cities that already is assigned to the route
        del cities[home]
        for i in route:
            del cities[i]
        # Get the total weight
        total_weight = 0
        for key, city in cities.items():
            total_weight += city.distance
        # Add weights
        weights = []
        for key, city in cities.items():
            weights.append(total_weight / city.distance)
        # Add a city at random
        from_index = random.choices(list(cities.keys()), weights=weights)[0]
```

```
        route.append(from_index)
    # Create a new state and update the distance
    state = State(route)
    state.update_distance(matrix, home)
    # Return a state
    return state
# Mutate a solution
def mutate(matrix:[], home:int, state:State, mutation_rate:float=0.01):

    # Create a copy of the state
    mutated_state = state.deepcopy()
    # Loop all the states in a route
    for i in range(len(mutated_state.route)):
        # Check if we should do a mutation
        if(random.random() < mutation_rate):
            # Swap two cities
            j = int(random.random() * len(state.route))
            city_1 = mutated_state.route[i]
            city_2 = mutated_state.route[j]
            mutated_state.route[i] = city_2
            mutated_state.route[j] = city_1
    # Update the distance
    mutated_state.update_distance(matrix, home)
    # Return a mutated state
    return mutated_state
# Hill climbing algorithm
def hill_climbing(matrix:[], home:int, initial_state:State, max_iterations:int, mutation_rate:float=0.01):
    # Keep track of the best state
    best_state = initial_state
    # An iterator can be used to give the algorithm more time to find a solution
    iterator = 0
    # Create an infinite loop
    while True:
        # Mutate the best state
        neighbor = mutate(matrix, home, best_state, mutation_rate)
        # Check if the distance is less than in the best state
        if(neighbor.distance >= best_state.distance):
            iterator += 1
            if (iterator > max_iterations):
                break
        if(neighbor.distance < best_state.distance):
            best_state = neighbor
    # Return the best state
    return best_state
# The main entry point for this module
def main():
```

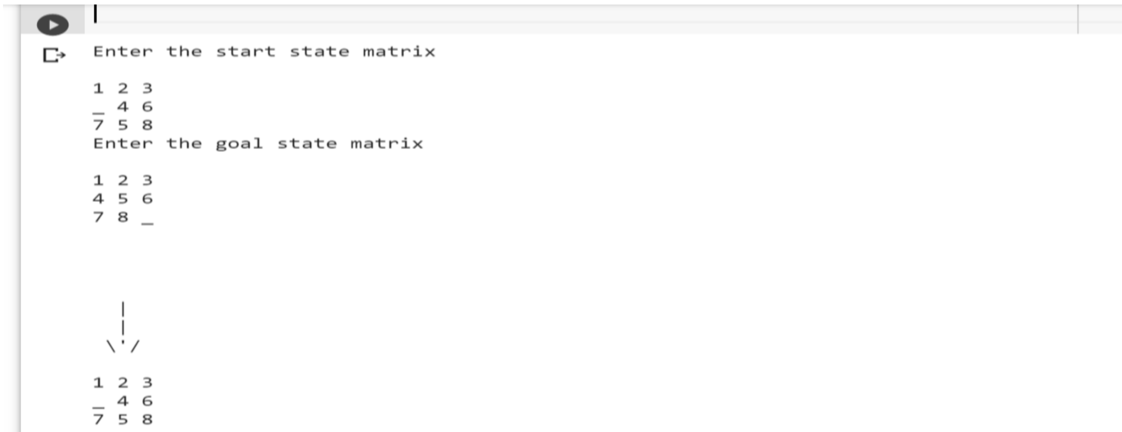
```
    # Cities to travel
    cities = ['New York', 'Los Angeles', 'Chicago', 'Minneapolis', 'Denver', 'Dallas', 'Seattle', 'Boston', 'San Fra
ncisco', 'St. Louis', 'Houston', 'Phoenix', 'Salt Lake City']
    city_indexes = [0,1,2,3,4,5,6,7,8,9,10,11,12]
    # Index of start location
    home = 2 # Chicago
    # Max iterations
    max_iterations = 1000
    # Distances in miles between cities, same indexes (i, j) as in the cities array
    matrix = [[0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972],
            [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579],
            [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260],
            [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987],
            [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
            [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999],
            [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1119, 701],
            [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099],
            [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600],
            [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162],
            [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200],
            [2145, 357, 1453, 1280, 586, 887, 1119, 2300, 653, 1272, 1017, 0, 504],
            [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0]]

    # Run hill climbing to find a better solution
    state = get_best_solution_by_distance(matrix, home)
    state = hill_climbing(matrix, home, state, 1000, 0.1)
    print('-- Hill climbing solution --')
    print(cities[home], end='')
    for i in range(0, len(state.route)):
       print(' -> ' + cities[state.route[i]], end='')
    print(' -> ' + cities[home], end='')
    print('\n\nTotal distance: {0} miles'.format(state.distance))
    print()
# Tell python to run main method
if __name__ == "__main__": main()
```

Output:

-- Hill climbing solution --
Chicago -> St. Louis -> Minneapolis -> Denver -> Salt Lake City -> Seattle -> San Francisco -> Los Angeles -> Phoenix -> Dallas -> Houston -> New York -> Boston -> Chicago

Total distance: 7534 miles

Screenshot:



Result

Thus the Travelling Salesman Problem (TSP) was solved successfully using Hill Climbing algorithm.

Ex No: 06	Study of Prolog
Date:02.02.2024	

Objective:
To design a simple kinship domain system using prolog.

Algorithm:
- List out the various proof statements and their relations
- Write down the rules and add predict clauses if necessary
- Write down the askable statements
- Declare a query and run
- If user input yes for all the questions, then the predicate is proved as true
- Else it is proved as false.

Description:
Prolog stands for "Programming in Logic".
It is the most common logic program language.
Prolog programs consist of clauses
Rules, facts, queries
Clauses consist of literals separated by logical connectors.
Head literal
Zero or more body literals
isGrandfatherOf(G,C) :-
isFatherOf(G,X), (isFatherOf(X,C) ; isMotherOf(X,C)).
Logical connectors are
implication (:-), conjunction (,) and disjunction (;)
Literals consist of a predicate symbol, punctuation symbols and arguments
Punctuation symbols are the comma "," and the round braces "(" and ")"
isGrandfatherOf(G,C) isFatherOf(peter,hans)
fieldHasType(FieldName, type(basic,TypeName,5))
Knowledge representation and reasoning (KR) is the field of artificial intelligence (AI) dedicated to representing information about the world in a form that a computer system can utilize to solve complex tasks such as diagnosing a medical condition or having a dialog in a natural language. Knowledge representation incorporates findings from psychology about how humans solve problems and represent knowledge in order to design formalisms that will make complex systems easier to design and build. Knowledge representation and reasoning also incorporates findings from logic to automate various kinds of reasoning, such as the application of rules or the relations of sets and subsets.

<u>Getting started with prolog:</u>
http://www.swi-prolog.org/pldoc/man?section=quickstart

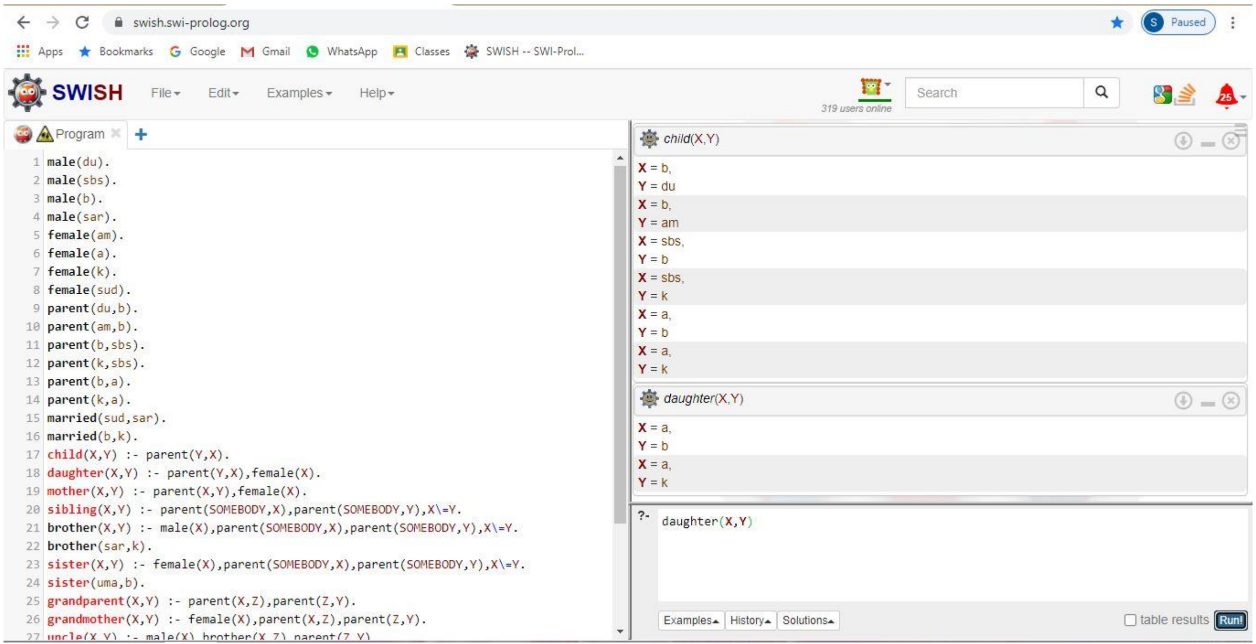
<u>Working with prolog:</u>

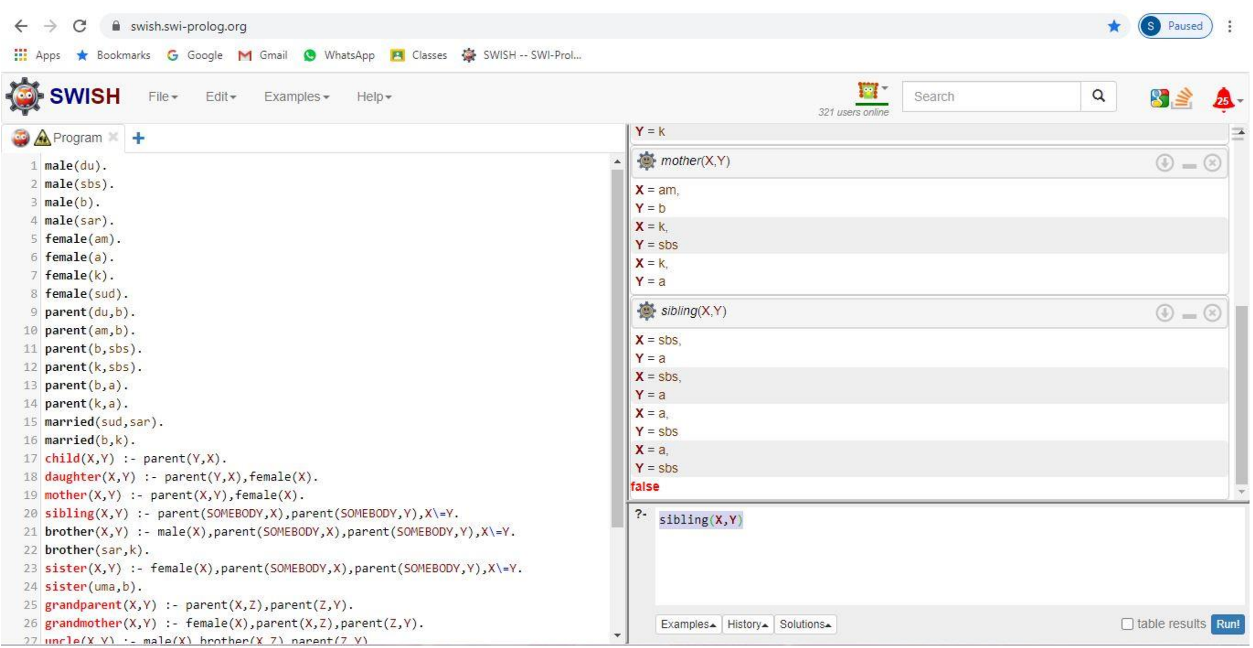
https://swish.swi-prolog.org/

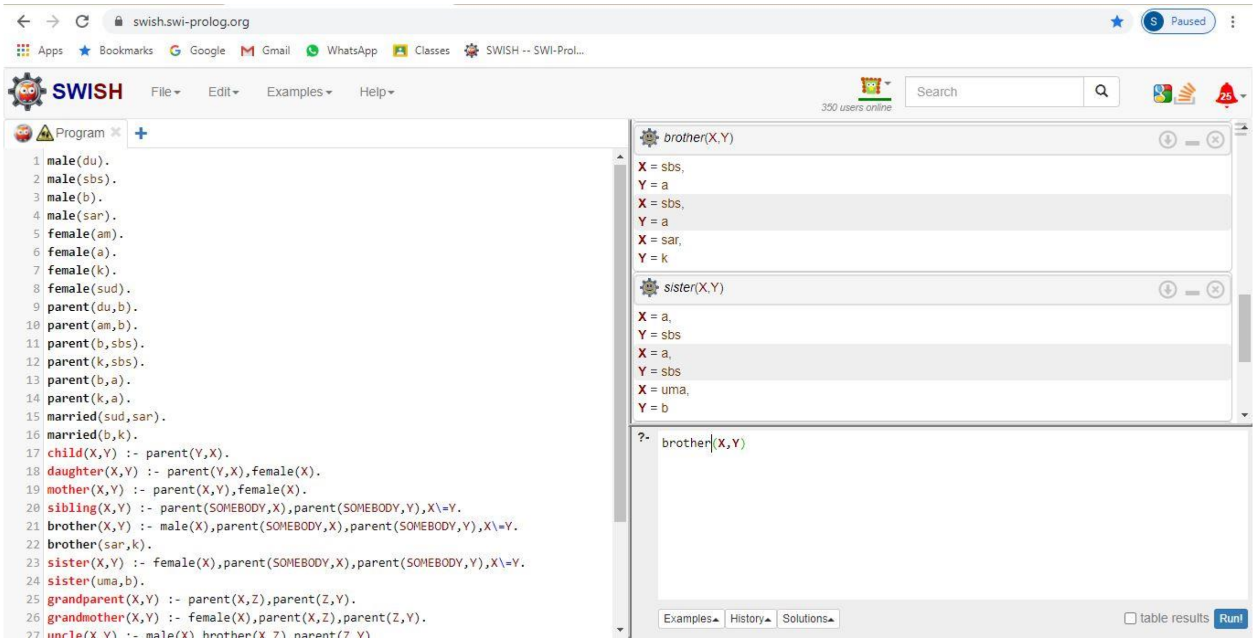
Program:
male(du).
male(sbs).

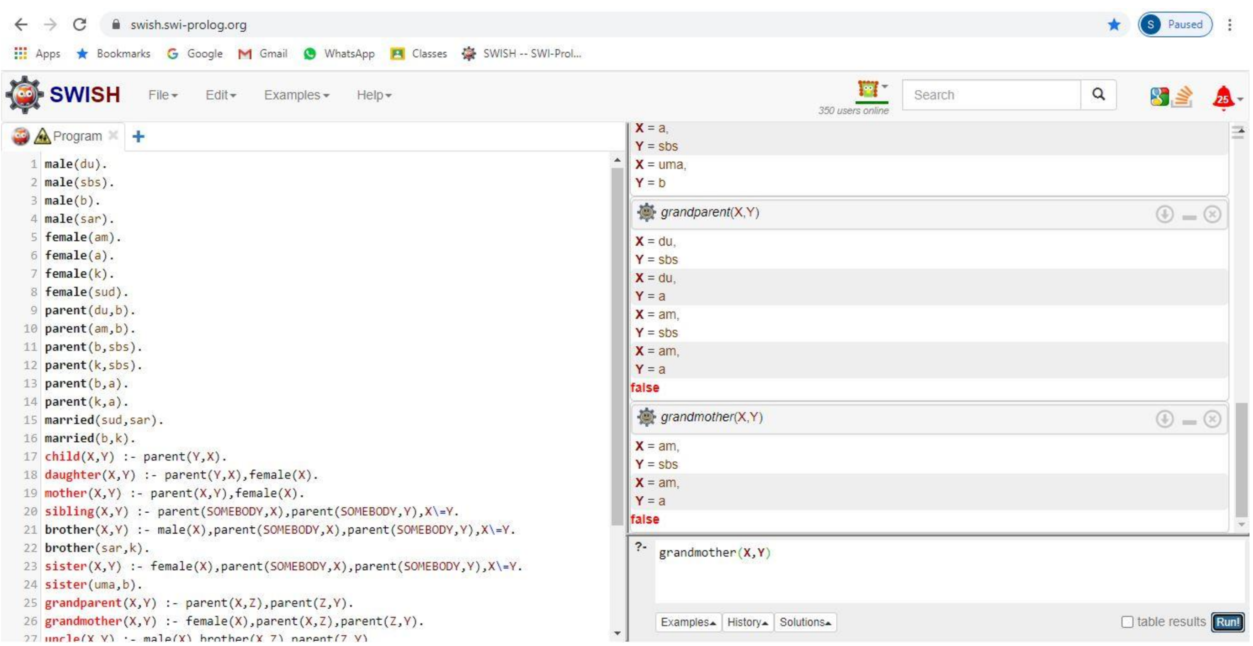
```
male(b).
male(sar).
female(am).
female(a).
female(k).
female(sud).
parent(du,b).
parent(am,b).
parent(b,sbs).
parent(k,sbs).
parent(b,a).
parent(k,a).
married(sud,sar).
married(b,k).
child(X,Y) :- parent(Y,X).
daughter(X,Y) :- parent(Y,X),female(X).
mother(X,Y) :- parent(X,Y),female(X).
sibling(X,Y) :- parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(X,Y) :- male(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(sar,k).
sister(X,Y) :- female(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
sister(uma,b).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
grandmother(X,Y) :- female(X),parent(X,Z),parent(Z,Y).
uncle(X,Y) :- male(X),brother(X,Z),parent(Z,Y).
sister_in_law(X,Y) :- female(X),married(X,Z),brother(Z,Y).
mother_in_law(X,Y) :- female(X),parent(X,Z),married(Z,Y).
wife(X,Y) :- female(X),married(X,Y).
ancestor(X,Y) :- parent(X,Y).
descendant(X,Y) :- ancestor(Y,X).
relative_by_blood(X,Y) :- ancestor(Z,X),ancestor(Z,Y).
```

Output:


swish.swi-prolog.org
Apps
Bookmarks
Google
Gmail
WhatsApp
Classes
SWISH -- SWI-Prol...
Paused
SWISH
File
Edit
Examples
Help
319 users online
Search
Program
male(du).
male(sbs).
male(b).
male(sar).
female(am).
female(a).
female(k).
female(sud).
parent(du,b).
parent(am,b).
parent(b,sbs).
parent(k,sbs).
parent(b,a).
parent(k,a).
married(sud,sar).
married(b,k).
child(X,Y) :- parent(Y,X).
daughter(X,Y) :- parent(Y,X),female(X).
mother(X,Y) :- parent(X,Y),female(X).
sibling(X,Y) :- parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(X,Y) :- male(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(sar,k).
sister(X,Y) :- female(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
sister(uma,b).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
grandmother(X,Y) :- female(X),parent(X,Z),parent(Z,Y).
child(X,Y)
X = b,
Y = du
X = b,
Y = am
X = sbs,
Y = b
X = sbs,
Y = k
X = a,
Y = b
X = a,
Y = k
daughter(X,Y)
X = a,
Y = b
X = a,
Y = k
?- daughter(X,Y)
Examples
History
Solutions
table results
Run!


swish.swi-prolog.org
Apps
Bookmarks
Google
Gmail
WhatsApp
Classes
SWISH -- SWI-Prol...
Paused
SWISH
File
Edit
Examples
Help
321 users online
Search
Program
male(du).
male(sbs).
male(b).
male(sar).
female(am).
female(a).
female(k).
female(sud).
parent(du,b).
parent(am,b).
parent(b,sbs).
parent(k,sbs).
parent(b,a).
parent(k,a).
married(sud,sar).
married(b,k).
child(X,Y) :- parent(Y,X).
daughter(X,Y) :- parent(Y,X),female(X).
mother(X,Y) :- parent(X,Y),female(X).
sibling(X,Y) :- parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(X,Y) :- male(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(sar,k).
sister(X,Y) :- female(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
sister(uma,b).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
grandmother(X,Y) :- female(X),parent(X,Z),parent(Z,Y).
Y = k
mother(X,Y)
X = am,
Y = b
X = k,
Y = sbs
X = k,
Y = a
sibling(X,Y)
X = sbs,
Y = a
X = sbs,
Y = a
X = a,
Y = sbs
X = a,
Y = sbs
false
?- sibling(X,Y)
Examples
History
Solutions
table results
Run!


swish.swi-prolog.org
Apps Bookmarks Google Gmail WhatsApp Classes SWISH -- SWI-Prol...
SWISH File Edit Examples Help
350 users online
Search
Program
male(du).
male(sbs).
male(b).
male(sar).
female(am).
female(a).
female(k).
female(sud).
parent(du,b).
parent(am,b).
parent(b,sbs).
parent(k,sbs).
parent(b,a).
parent(k,a).
married(sud,sar).
married(b,k).
child(X,Y) :- parent(Y,X).
daughter(X,Y) :- parent(Y,X),female(X).
mother(X,Y) :- parent(X,Y),female(X).
sibling(X,Y) :- parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(X,Y) :- male(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(sar,k).
sister(X,Y) :- female(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
sister(uma,b).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
grandmother(X,Y) :- female(X),parent(X,Z),parent(Z,Y).
brother(X,Y)
X = sbs,
Y = a
X = sbs,
Y = a
X = sar,
Y = k
sister(X,Y)
X = a,
Y = sbs
X = a,
Y = sbs
X = uma,
Y = b
?- brother(X,Y)
Examples History Solutions
table results Run!


swish.swi-prolog.org
Apps Bookmarks Google Gmail WhatsApp Classes SWISH -- SWI-Prol...
SWISH File Edit Examples Help
350 users online
Search
Program
male(du).
male(sbs).
male(b).
male(sar).
female(am).
female(a).
female(k).
female(sud).
parent(du,b).
parent(am,b).
parent(b,sbs).
parent(k,sbs).
parent(b,a).
parent(k,a).
married(sud,sar).
married(b,k).
child(X,Y) :- parent(Y,X).
daughter(X,Y) :- parent(Y,X),female(X).
mother(X,Y) :- parent(X,Y),female(X).
sibling(X,Y) :- parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(X,Y) :- male(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(sar,k).
sister(X,Y) :- female(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
sister(uma,b).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
grandmother(X,Y) :- female(X),parent(X,Z),parent(Z,Y).
X = a,
Y = sbs
X = uma,
Y = b
grandparent(X,Y)
X = du,
Y = sbs
X = du,
Y = a
X = am,
Y = sbs
X = am,
Y = a
false
grandmother(X,Y)
X = am,
Y = sbs
X = am,
Y = a
false
?- grandmother(X,Y)
Examples History Solutions
table results Run!

swish.swi-prolog.org

SWISH File Edit Examples Help

351 users online

Program

```
female(sud).
parent(du,b).
parent(am,b).
parent(b,sbs).
parent(k,sbs).
parent(b,a).
parent(k,a).
married(sud,sar).
married(b,k).
child(X,Y) :- parent(Y,X).
daughter(X,Y) :- parent(Y,X),female(X).
mother(X,Y) :- parent(X,Y),female(X).
sibling(X,Y) :- parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(X,Y) :- male(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(sar,k).
sister(X,Y) :- female(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
sister(uma,b).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
grandmother(X,Y) :- female(X),parent(X,Z),parent(Z,Y).
uncle(X,Y) :- male(X),brother(X,Z),parent(Z,Y).
sister_in_law(X,Y) :- female(X),married(X,Z),brother(Z,Y).
mother_in_law(X,Y) :- female(X),parent(X,Z),married(Z,Y).
wife(X,Y) :- female(X),married(X,Y).
ancestor(X,Y) :- parent(X,Y).
descendant(X,Y) :- ancestor(Y,X).
relative_by_blood(X,Y) :- ancestor(Z,X),ancestor(Z,Y).
```

Y = a
false

uncle(X,Y)

X = sar,
Y = sbs
X = sar,
Y = a

sister_in_law(X,Y)

X = sud,
Y = k

mother_in_law(X,Y)

X = am,
Y = k
false

wife(X,Y)

X = sud,
Y = sar

?- wife(X,Y)

Examples History Solutions table results Run!

swish.swi-prolog.org

SWISH File Edit Examples Help

347 users online

Program


```
female(sud).
parent(du,b).
parent(am,b).
parent(b,sbs).
parent(k,sbs).
parent(b,a).
parent(k,a).
married(sud,sar).
married(b,k).
child(X,Y) :- parent(Y,X).
daughter(X,Y) :- parent(Y,X),female(X).
mother(X,Y) :- parent(X,Y),female(X).
sibling(X,Y) :- parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(X,Y) :- male(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
brother(sar,k).
sister(X,Y) :- female(X),parent(SOMEBODY,X),parent(SOMEBODY,Y),X\=Y.
sister(uma,b).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
grandmother(X,Y) :- female(X),parent(X,Z),parent(Z,Y).
uncle(X,Y) :- male(X),brother(X,Z),parent(Z,Y).
sister_in_law(X,Y) :- female(X),married(X,Z),brother(Z,Y).
mother_in_law(X,Y) :- female(X),parent(X,Z),married(Z,Y).
wife(X,Y) :- female(X),married(X,Y).
ancestor(X,Y) :- parent(X,Y).
descendant(X,Y) :- ancestor(Y,X).
relative_by_blood(X,Y) :- ancestor(Z,X),ancestor(Z,Y).
```


false

wife(X,Y)

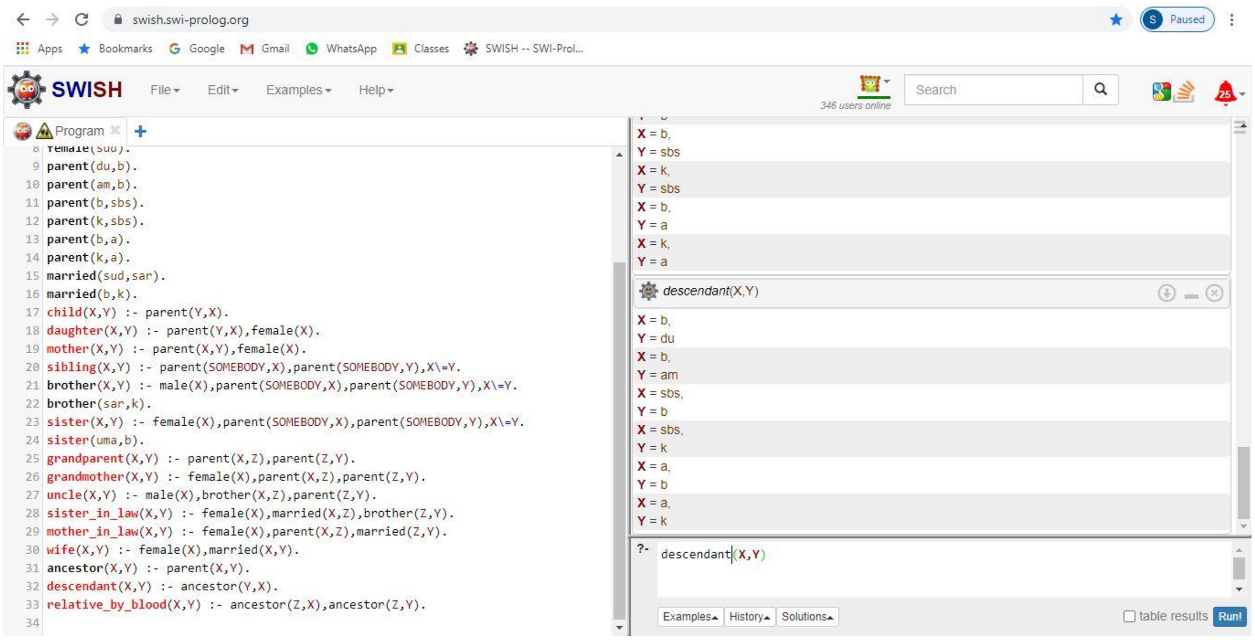
X = sud,
Y = sar

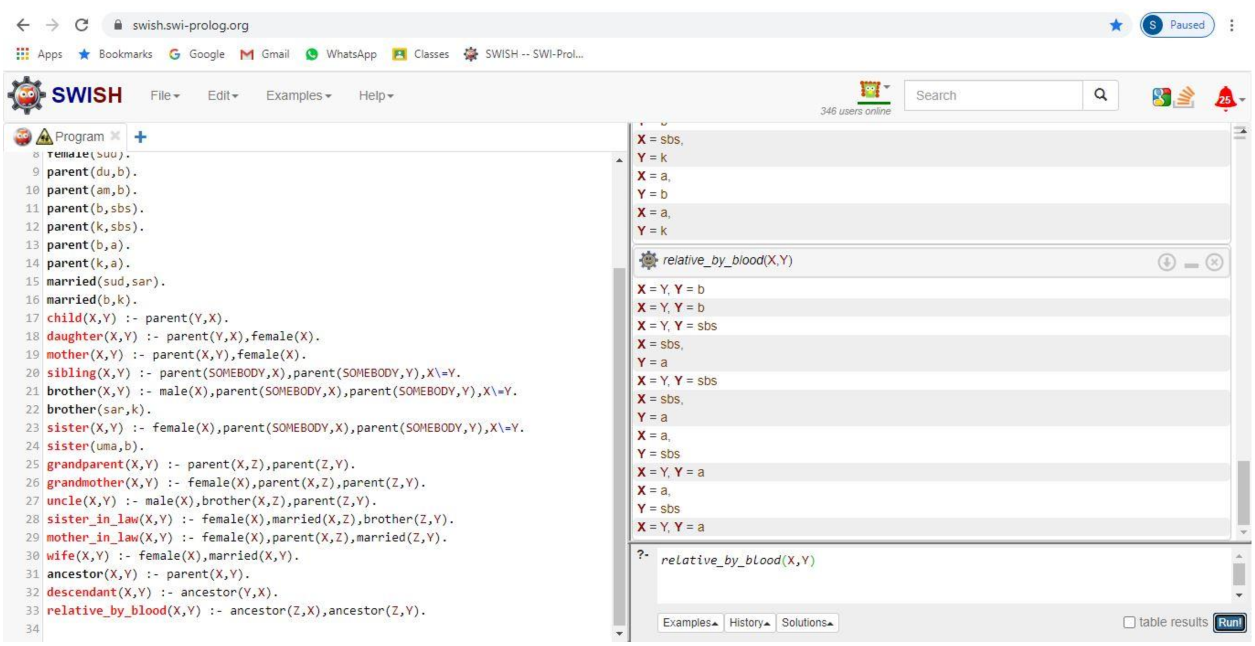
ancestor(X,Y)

X = du,
Y = b
X = am,
Y = b
X = b,
Y = sbs
X = k,
Y = sbs
X = b,
Y = a
X = k,
Y = a

?- ancestor(X,Y)

Examples History Solutions table results Run!





Result:

The program is solved and the output is executed successfully.

Ex No: 07	Implement the Constraint Satisfaction problem
Date: 09.02.2024	

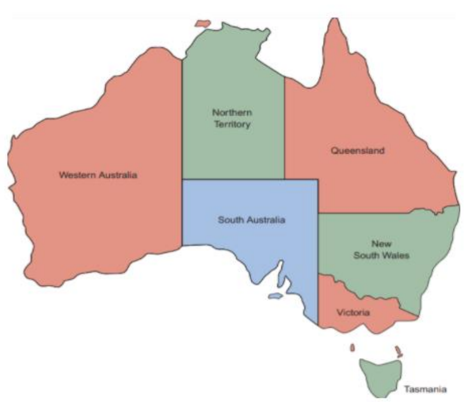
Objective:

To implement graph coloring algorithm using Constraint Satisfaction problem.

Algorithm:

- A constraint satisfaction problem consists of variables of type V that have ranges of values known as domains of type D and constraints that determine whether a particular variable's domain selection is valid.
- The __init__() initializer creates the constraints dict. The add_constraint() method goes through all of the variables touched by a given constraint and adds itself to the constraints mapping for each of them.
- Check if the value assignment is consistent by checking all constraints f or the given variable against it.
- Assignment is complete if every variable is assigned (our base case) and get all variables in the CSP but not in the assignment.
- If either place is not in the assignment, then it is not yet possible for their colors to be conflicting
- Then check if the color assigned to place1 is not the same as the color assigned to place2
- Finally, backtracking_search() is called to find a solution. A correct solution includes an assigned color for every region.

Description with Sample data:



In a solution to the Australian map-coloring problem, no two adjacent parts of Australia can be colored with the same color.

To model the problem as a CSP, we need to define the variables, domains, and constraints.

- The variables are the seven regions of Australia (at least the seven that we'll restrict ourselves to): Western Australia; Northern Territory; South Australia; Queensland; New South Wales; Victoria; and Tasmania. In our CSP, they can be modeled with strings.
- The domain of each variable is the three different colors that can possibly be assigned (we'll use red, green, and blue).
- The constraints are, No two adjacent regions can be colored with the same color, and our constraints are dependent on which regions border one another.
- We can use binary constraints (constraints between two variables).
- Every two regions that share a border also share a binary constraint indicating they can't be assigned the same color.
- To implement these binary constraints in code, we need to subclass the Constraint class.
- The MapColoringConstraint subclass takes two variables in its constructor (therefore being a binary constraint): the two regions that share a border.
- Its overridden satisfied() method check whether the two regions both have a domain value (color) assigned to them—if either doesn't, the constraint's trivially satisfied until they do (there can't be a conflict when one doesn't yet have a color).
- Then it checks whether the two regions are assigned the same color (obviously there's a conflict, meaning the constraint isn't satisfied, when they're the same).

Program:

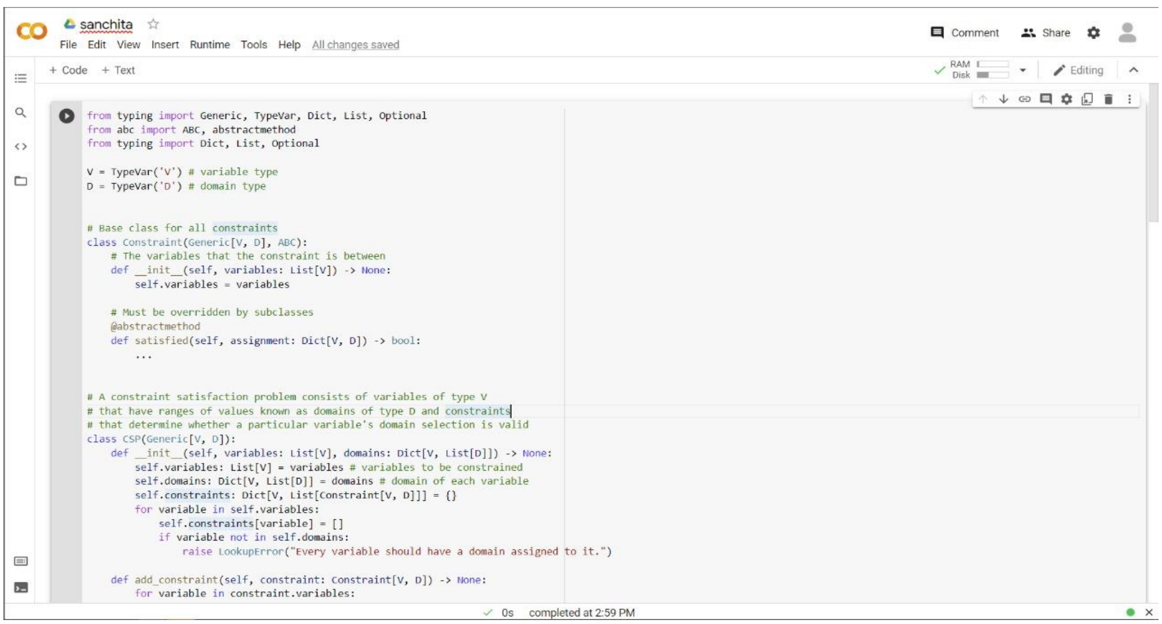
```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod
from typing import Dict, List, Optional

V = TypeVar('V') # variable type
D = TypeVar('D') # domain type

class Constraint(Generic[V, D], ABC):
        def __init__(self, variables: List[V]) -> None:
        self.variables = variables
       @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        ...

class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
        self.variables: List[V] = variables # variables to be constrained
```

```
        self.domains: Dict[V, List[D]] = domains # domain of each variable
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        for variable in self.variables:
            self.constraints[variable] = []
            if variable not in self.domains:
                raise LookupError("Every variable should have a domain assigned to it.")
    def add_constraint(self, constraint: Constraint[V, D]) -> None:
        for variable in constraint.variables:
            if variable not in self.variables:
                raise LookupError("Variable in constraint not in CSP")
            else:
                self.constraints[variable].append(constraint)
     def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
        for constraint in self.constraints[variable]:
            if not constraint.satisfied(assignment):
                return False
        return True

    def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:
               if len(assignment) == len(self.variables):
            return assignment
               unassigned: List[V] = [v for v in self.variables if v not in assignment]

        first: V = unassigned[0]
        for value in self.domains[first]:
            local_assignment = assignment.copy()
            local_assignment[first] = value

            if self.consistent(first, local_assignment):
                result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)

                if result is not None:
                    return result
        return None

class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
        self.place2: str = place2
```

```python
    def satisfied(self, assignment: Dict[str, str]) -> bool:
        if self.place1 not in assignment or self.place2 not in assignment:
            return True
        return assignment[self.place1] != assignment[self.place2]
if __name__ == "__main__":
    variables: List[str] = ["Western Australia", "Northern Territory", "South Australia",
                            "Queensland", "New South Wales", "Victoria", "Tasmania"]
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        domains[variable] = ["red", "green", "blue"]
    csp: CSP[str, str] = CSP(variables, domains)
    csp.add_constraint(MapColoringConstraint("Western Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Western Australia", "South Australia"))
    csp.add_constraint(MapColoringConstraint("South Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Queensland", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("New South Wales", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

Screenshot:

sanchita
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
RAM Disk
Editing
def add_constraint(self, constraint: Constraint[V, D]) -> None:
for variable in constraint.variables:
if variable not in self.variables:
raise LookupError("Variable in constraint not in CSP")
else:
self.constraints[variable].append(constraint)
# Check if the value assignment is consistent by checking all constraints
# for the given variable against it
def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
for constraint in self.constraints[variable]:
if not constraint.satisfied(assignment):
return False
return True
def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:
# assignment is complete if every variable is assigned (our base case)
if len(assignment) == len(self.variables):
return assignment
get all variables in the CSP but not in the assignment
unassigned: List[V] = [v for v in self.variables if v not in assignment]
get the every possible domain value of the first unassigned variable
first: V = unassigned[0]
for value in self.domains[first]:
local_assignment = assignment.copy()
local_assignment[first] = value
# if we're still consistent, we recurse (continue)
if self.consistent(first, local_assignment):
result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)
# if we didn't find the result, we will end up backtracking
if result is not None:
return result
return None
0s completed at 2:59 PM



sanchita
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
RAM Disk
Editing
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod
from typing import Dict, List, Optional
V = TypeVar('V') # variable type
D = TypeVar('D') # domain type
# Base class for all constraints
class Constraint(Generic[V, D], ABC):
# The variables that the constraint is between
def __init__(self, variables: List[V]) -> None:
self.variables = variables
# Must be overridden by subclasses
@abstractmethod
def satisfied(self, assignment: Dict[V, D]) -> bool:
...
# A constraint satisfaction problem consists of variables of type V
# that have ranges of values known as domains of type D and constraints
# that determine whether a particular variable's domain selection is valid
class CSP(Generic[V, D]):
def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
self.variables: List[V] = variables # variables to be constrained
self.domains: Dict[V, List[D]] = domains # domain of each variable
self.constraints: Dict[V, List[Constraint[V, D]]] = {}
for variable in self.variables:
self.constraints[variable] = []
if variable not in self.domains:
raise LookupError("Every variable should have a domain assigned to it.")
def add_constraint(self, constraint: Constraint[V, D]) -> None:
for variable in constraint.variables:
0s completed at 2:59 PM

```python
class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
        self.place2: str = place2

    def satisfied(self, assignment: Dict[str, str]) -> bool:
        # If either place is not in the assignment then it is not
        # yet possible for their colors to be conflicting
        if self.place1 not in assignment or self.place2 not in assignment:
            return True
        # check the color assigned to place1 is not the same as the
        # color assigned to place2
        return assignment[self.place1] != assignment[self.place2]


if __name__ == "__main__":
    variables: List[str] = ["Western Australia", "Northern Territory", "South Australia",
                            "Queensland", "New South Wales", "Victoria", "Tasmania"]
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        domains[variable] = ["red", "green", "blue"]
    csp: CSP[str, str] = CSP(variables, domains)
    csp.add_constraint(MapColoringConstraint("Western Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Western Australia", "South Australia"))
    csp.add_constraint(MapColoringConstraint("South Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Queensland", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("New South Wales", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
```

```python
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        domains[variable] = ["red", "green", "blue"]
    csp: CSP[str, str] = CSP(variables, domains)
    csp.add_constraint(MapColoringConstraint("Western Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Western Australia", "South Australia"))
    csp.add_constraint(MapColoringConstraint("South Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Queensland", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("New South Wales", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

```python
        print("No solution found!")
    else:
        print(solution)
```

```
{'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia': 'blue', 'Queensland': 'red', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'green'}
```

Result:

Thus, the Map Coloring problem was successfully designed using Constraint Satisfaction Algorithm.

Ex No: 08	**Implement the k-nearest neighbors**
Date:15.02.2024	

Objective:

To implement the k-nearest neighbours' algorithm.

Algorithm:

- Pick a value for K from the available data points.
- Take the K nearest neighbors of the new data point according to their Euclidean distance.
- Among these neighbors, count the number of data points in each category and assign the new data point to the category where you counted the most neighbors.

Description:

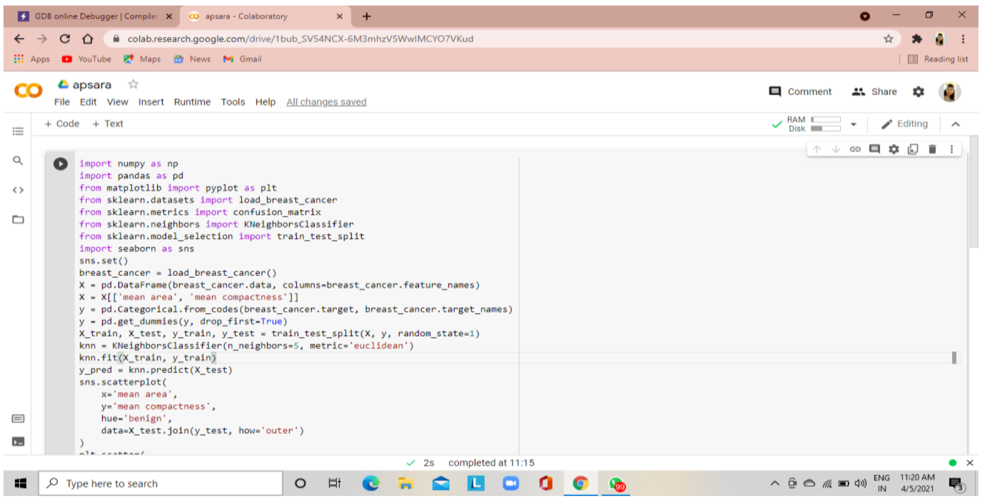
- K-Nearest Neighbors, or KNN for short, is one of the simplest machine learning algorithms and is used in a wide array of institutions.
- KNN is a non-parametric, lazy learning algorithm. When we say a technique is non-parametric, it means that it does not make any assumptions about the underlying data. In other words, it makes its selection based off of the proximity to other data points regardless of what feature the numerical values represent.
- Being a lazy learning algorithm implies that there is little to no training phase. Therefore, we can immediately classify new data points as they present themselves.

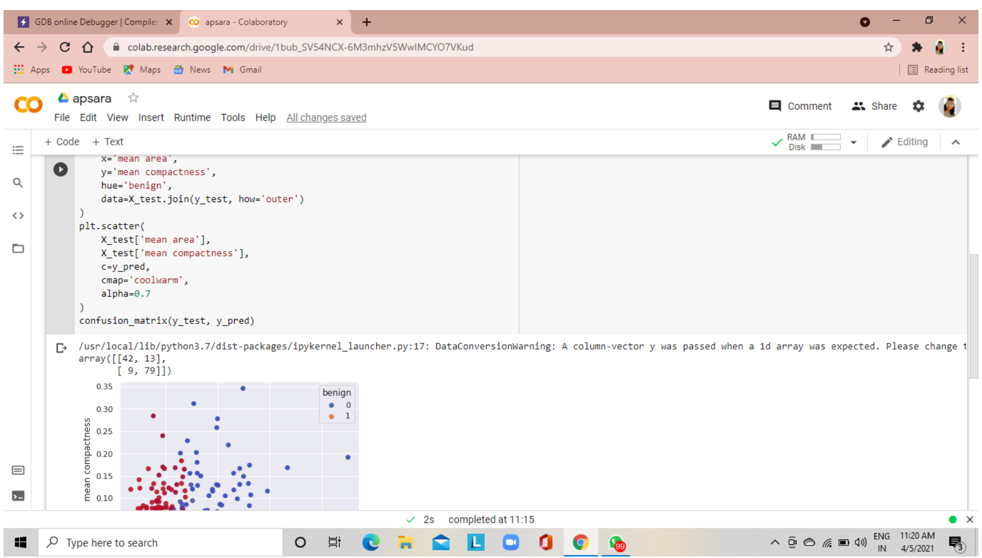
Program:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import seaborn as sns
sns.set()
breast_cancer = load_breast_cancer()
X = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
X = X[['mean area', 'mean compactness']]
y = pd.Categorical.from_codes(breast_cancer.target, breast_cancer.target_names)
y = pd.get_dummies(y, drop_first=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

```
knn = KNeighborsClassifier(n_neighbors=5, metric='euclidean')
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
sns.scatterplot(
    x='mean area',    y='mean compactness',
    hue='benign',    data=X_test.join(y_test, how='outer')
)
plt.scatter(
    X_test['mean area'],     X_test['mean compactness'],
    c=y_pred,     cmap='coolwarm',     alpha=0.7
)
confusion_matrix(y_test, y_pred)
```

Output:





Result:

The program is solved and the output is executed successfully.

Ex No: 9	Simple Expert System Using Decision Tree
Date:22.02.2024	

Objective:

To design a simple expert system using decision trees.

Algorithm:

- List out the various proof statements and their relations
- Write down the rules and add predict clauses if necessary
- Write down the askable statements
- Declare a query and run
- If user input yes for all the questions, then the predicate is proved as true
- Else it is proved as false.

Description with sample data:

An expert system emulates the decision-making ability of a human expert.
Prolog is very well suited for implementing expert systems due to several reasons:

- Prolog itself can be regarded as a simple inference engine or theorem prover that derives conclusions from known rules. Very simple expert systems can be implemented by relying on Prolog’s built- in search and backtracking mechanisms.
- Prolog data structures let us flexibly and conveniently represent rule-based systems that need additional functionality such as probabilistic reasoning.
- We can easily write meta-interpreters in Prolog to implement custom evaluation strategies of rules.

1. Animal Identification

Our aim is to write an expert system that helps us identify animals.

Suppose we have already obtained the following knowledge about animals, which are rules of inference:

- If it has a fur and says woof, then the animal is a dog.
- If it has a fur and says meow, then the animal is a cat.
- If it has feathers and says quack, then the animal is a duck.

Program:

```
/*Identify the animal?*/
prove(true) :- !.
prove((B, Bs)) :- !, prove(B), prove(Bs).
prove(H) :-
clause(H, B),
prove(B).
```

prove(H) :- askable(H), writeln(H), read(Answer),
Answer == yes.
/*Rules*/
animal(dog) :- has_fur(X),helpsForGaurding(X),say_woof(X). animal(cat):-
has_smallfur(X),lovesMilk(X),say_meow(X). animal(duck) :- has_feather(X),
anAmphinian(X),say_quack(X).
/*Askable*/ askable(has_fur(_)). askable(helpsForGaurding(_)). askable(say_woof(_)).
askable(has_smallfur(_)). askable(lovesMilk(_)). askable(say_meow(_)). askable(has_feather(_)).
askable(anAmphinian(_)). askable(say_quack(_)).

Code:

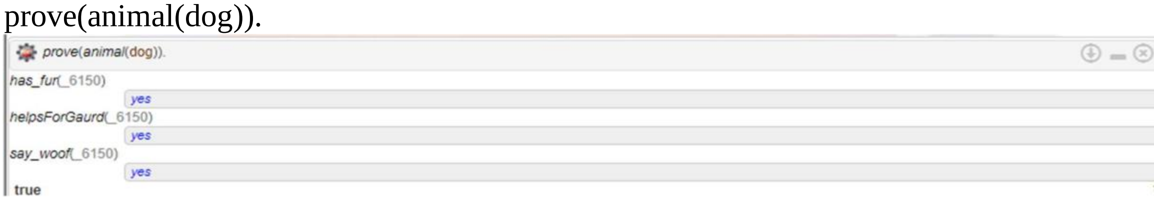
```
prove(true) :- !.

prove((B, Bs)) :- !,
    prove(B),
    prove(Bs).

prove(H) :-
    clause(H, B),
    prove(B).

prove(H) :-
    askable(H),
    writeln(H),
    read(Answer),
    Answer == yes.


animal(dog) :-
    has_fur(X),
    helpsForGaurding(X),
    say_woof(X).

animal(cat):-
    has_smallfur(X),
    lovesMilk(X),
    say_meow(X).

animal(duck) :-
    has_feather(X),
    anAmphinian(X),
    say_quack(X).


askable(has_fur(_)).
askable(helpsForGaurding(_)).
askable(say_woof(_)).
askable(has_smallfur(_)).
askable(lovesMilk(_)).
askable(say_meow(_)).
askable(has_feather(_)).
askable(anAmphinian(_)).
askable(say_quack(_)).
```

Output: Queries:

prove(animal(A)).

prove(animal(A)).
has_fur(_6202)
yes
helpsForGaurd(_6202)
yes
say_woof(_6202)
yes
A = dog

prove(animal(dog)).

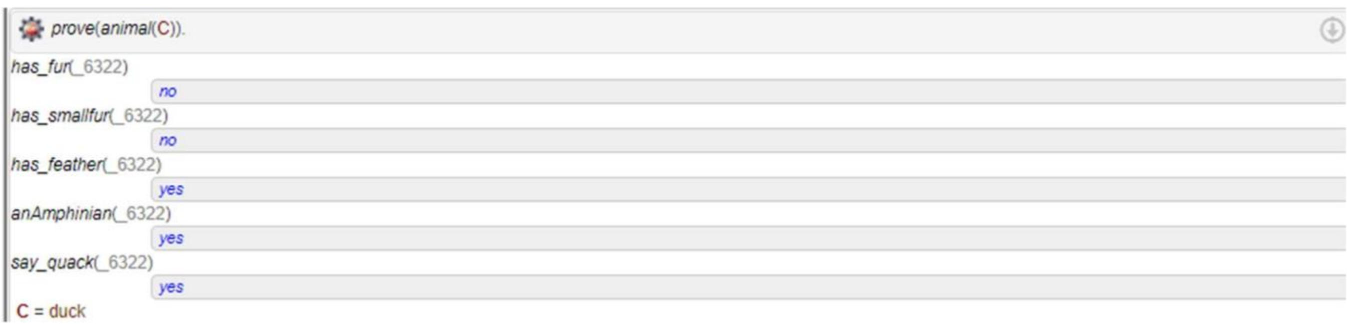


prove(animal(dog)).
has_fur(_6150)
yes
helpsForGaurd(_6150)
yes
say_woof(_6150)
yes
true

prove(animal(B)).

prove(animal(cat)).

prove(animal(C)).

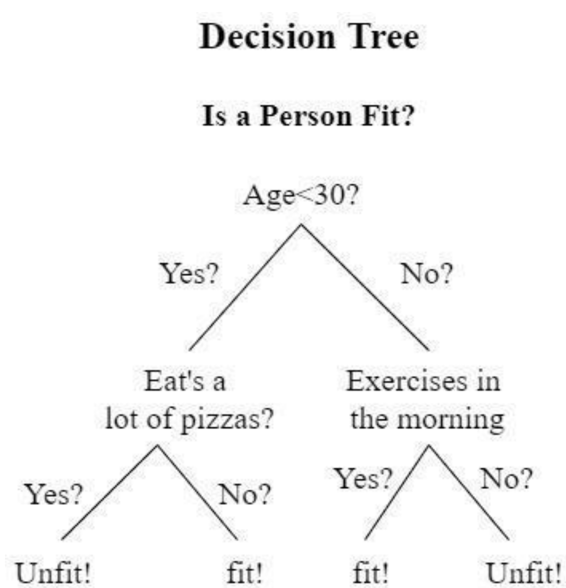


prove(animal(duck)).

TRACE:

```
trace,(prove(animal(dog))).
   Call: prove(animal(dog))
    Call: clause(animal(dog), _6388)
    Exit: clause(animal(dog), (has_fur(_6402),helpsForGaurding(_6402),say_woof(_6402)))
    Call: prove((has_fur(_6402),helpsForGaurding(_6402),say_woof(_6402)))
     Call: prove(has_fur(_6402))
      Call: clause(has_fur(_6402), _6424)
      Fail: clause(has_fur(_6402), _6424)
     Redo: prove(has_fur(_6402))
      Call: askable(has_fur(_6402))
      Exit: askable(has_fur(_6402))
      Call: writeln(has_fur(_6402))
has_fur(_6402)
      Exit: writeln(has_fur(_6402))
      Call: read(_7918)
         yes
      Exit: read(yes)
      Call: yes==yes
      Exit: yes==yes
     Exit: prove(has_fur(_6402))
     Call: prove((helpsForGaurding(_6402),say_woof(_6402)))
      Call: prove(helpsForGaurding(_6402))
       Call: clause(helpsForGaurding(_6402), _8172)
       Fail: clause(helpsForGaurding(_6402), _8172)
      Redo: prove(helpsForGaurding(_6402))
       Call: askable(helpsForGaurding(_6402))
       Exit: askable(helpsForGaurding(_6402))
       Call: writeln(helpsForGaurding(_6402))
helpsForGaurding(_6402)
       Exit: writeln(helpsForGaurding(_6402))
       Call: read(_9668)
         yes
       Exit: read(yes)
       Call: yes==yes
       Exit: yes==yes
      Exit: prove(helpsForGaurding(_6402))
      Call: prove(say_woof(_6402))
       Call: clause(say_woof(_6402), _9922)
       Fail: clause(say_woof(_6402), _9922)
      Redo: prove(say_woof(_6402))
       Call: askable(say_woof(_6402))
       Exit: askable(say_woof(_6402))
       Call: writeln(say_woof(_6402))
say_woof(_6402)
       Exit: writeln(say_woof(_6402))
       Call: read(_11416)
         yes
       Exit: read(yes)
       Call: yes==yes
       Exit: yes==yes
      Exit: prove(say_woof(_6402))
     Exit: prove((helpsForGaurding(_6402),say_woof(_6402)))
    Exit: prove((has_fur(_6402),helpsForGaurding(_6402),say_woof(_6402)))
   Exit: prove(animal(dog))
```

2. **Decision Making:**



Program:

/*Is a person fit?*/
prove(true) :- !
prove((B, Bs)) :- !, prove(B), prove(Bs).
prove(H) :-
clause(H, B),
prove(B).
prove(H) :- askable(H), writeln(H), read(Answer), Answer == yes

/*Rules*/
fit(X):-age_less_than_30(X),exercise_in_morning(X). unfit(X):-eat_lot_pizza(X),age_less_than_30(X).

/*Askable*/
askable(age_less_than_30(_)). askable(eat_lot_pizza(_)). askable(exercise_in_morning(_)).

Code:

```
/*is a person fit*/

prove(true) :- !.

prove((B, Bs)) :- !,
    prove(B),
    prove(Bs).
prove(H) :-
    clause(H, B),
    prove(B).

prove(H) :-
    askable(H),
    writeln(H),
    read(Answer),
    Answer == yes.

fit(X):-age_less_than_30(X),exercise_in_morning(X).
unfit(X):-eat_lot_pizza(X),age_less_than_30(X).


askable(age_less_than_30(_)).
askable(eat_lot_pizza(_)).
askable(exercise_in_morning(_)).
```

Output:

Queries:

prove(fit(saravanan)).

prove(unfit(saravanan))

```
prove(unfit(saravanan)).
eat_lot_pizza(saravanan)
        no
false
```

TRACE:

trace,(prove(fit(mala))).

```
trace, (prove(fit(saravanan))).
   Call: prove(fit(saravanan))
    Call: clause(fit(saravanan), _6320)
    Exit: clause(fit(saravanan), (age_less_than_30(saravanan),exercise_in_morning(saravanan)))
    Call: prove((age_less_than_30(saravanan),exercise_in_morning(saravanan)))
     Call: prove(age_less_than_30(saravanan))
      Call: clause(age_less_than_30(saravanan), _6346)
      Fail: clause(age_less_than_30(saravanan), _6346)
     Redo: prove(age_less_than_30(saravanan))
      Call: askable(age_less_than_30(saravanan))
      Exit: askable(age_less_than_30(saravanan))
      Call: writeln(age_less_than_30(saravanan))
age_less_than_30(saravanan)
      Exit: writeln(age_less_than_30(saravanan))
      Call: read(_7820)
          yes
      Exit: read(yes)
      Call: yes==yes
      Exit: yes==yes
     Exit: prove(age_less_than_30(saravanan))
     Call: prove(exercise_in_morning(saravanan))
      Call: clause(exercise_in_morning(saravanan), _8074)
      Fail: clause(exercise_in_morning(saravanan), _8074)
     Redo: prove(exercise_in_morning(saravanan))
      Call: askable(exercise_in_morning(saravanan))
      Exit: askable(exercise_in_morning(saravanan))
      Call: writeln(exercise_in_morning(saravanan))
exercise_in_morning(saravanan)
      Exit: writeln(exercise_in_morning(saravanan))
      Call: read(_9550)
          yes
      Exit: read(yes)
      Call: yes==yes
      Exit: yes==yes
     Exit: prove(exercise_in_morning(saravanan))
    Exit: prove((age_less_than_30(saravanan),exercise_in_morning(saravanan)))
   Exit: prove(fit(saravanan))
```

Result:

Thus a simple expert system using decision trees was designed successfully.