AD3311 ARTIFICIAL INTELLIGENCE LABORATORY L T P C

0 0 3 1.5

COURSE OBJECTIVES:

- To design and implement search strategies
- To implement game playing techniques
- To implement CSP techniques
- To develop systems with logical reasoning
- To develop systems with probabilistic reasoning

LIST OF EXPERIMENTS:

1. Implement basic search strategies – 8-Puzzle, 8 - Queens problem, Cryptarithmetic.
2. Implement A* and memory bounded A* algorithms
3. Implement Minimax algorithm for game playing (Alpha-Beta pruning)
4. Solve constraint satisfaction problems
5. Implement propositional model checking algorithms
6. Implement forward chaining, backward chaining, and resolution strategies
7. Build naïve Bayes models
8. Implement Bayesian networks and perform inferences
9. Mini-Project

TOTAL: 45 PERIODS

COURSE OUTCOMES:

At the end of this course, the students will be able to:

CO1: Design and implement search strategies
CO2: Implement game playing and CSP techniques
CO3: Develop logical reasoning systems
CO4: Develop probabilistic reasoning systems

EX.No. 1a IMPLEMENT BASIC SEARCH STRATEGIES 8-PUZZLE PROBLEM
DATE:

AIM

To write a python program to implement basic search strategies – 8-Puzzle Problem.

ALGORITHM

1. The code starts by creating a Solution class and then defining the method solve.
2. The function takes in a board as an argument, which is a list of tuples representing the positions on the board.
3. It iterates through each position in the list and creates a dictionary with that position's value set to 0.
4. Then it iterates through all possible moves for that position and returns their number of occurrences in dict.
5. After this, it loops over all nodes on the board until it finds one where there are no more moves left to make (i.e., when len(current_nodes) == 0).
6. This node will be returned as -1 if found or else its index will be stored into pos_0 so that we can find out what move was made at that point later on.
7. The next step is finding out what move was made at every node by looping over all possible moves for each node using self's find_next function, which takes in a single node as an argument and returns any other nodes connected to it via path-finding algorithms like DFS or BFS (see below).
8. For example, if pos_0 = 1 then self would call: moves = { 0: [1], 1:
9. The code will print the number of paths in a solution.

PROGRAM

```
class Solution:
  def solve(self, board):
    dict = {}
    flatten = []
    for i in range(len(board)):
      flatten += board[i]
    flatten = tuple(flatten)
    dict[flatten] = 0
    if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):
      return 0
    return self.get_paths(dict)
  def get_paths(self, dict):
    cnt = 0
    while True:
      current_nodes = [x for x in dict if dict[x] == cnt]
      if len(current_nodes) == 0:
        return -1
      for node in current_nodes:
        next_moves = self.find_next(node)
        for move in next_moves:
          if move not in dict:
            dict[move] = cnt + 1
          if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):
            return cnt + 1
      cnt += 1
  def find_next(self, node):
        moves = {
          0: [1, 3],
          1: [0, 2, 4],
          2: [1, 5],
          3: [0, 4, 6],
          4: [1, 3, 5, 7],
          5: [2, 4, 8],
          6: [3, 7],
          7: [4, 6, 8],
          8: [5, 7],
        }
        results = []
        pos_0 = node.index(0)
        for move in moves[pos_0]:
          new_node = list(node)
          new_node[move], new_node[pos_0] = new_node[pos_0], new_node[move]
          results.append(tuple(new_node))
        return results
ob = Solution()
matrix = [
  [3, 1, 2],
  [4, 7, 5],
  [6, 8, 0]
]
print("Number of Moves=", ob.solve(matrix))
```

OUTPUT:

```
Number of Moves= 4
```

RESULT:

Thus the program to implement 8 puzzles search strategy is implemented and executed successfully.

EX.No. 1b IMPLEMENT BASIC SEARCH STRATEGIES – 8-QUEENS PROBLEM
DATE:

AIM

To write a python program to implement basic search strategies – 8-Queens Problem.

ALGORITHM

1. The code starts by asking the user to enter a number.
2. It then creates an NxN matrix with all elements set to 0.
3. The code then defines two functions: attack and N_queens.
4. The function attack checks vertically and horizontally, while the function N_queens checks diagonally.
5. If either of these functions return true, it means that there is a queen in that position on the board.
6. The code is a function that will check if there are enough queens on the chessboard.
7. The code starts by defining a function, N_queens (n), which will return true if there are enough queens and False otherwise.
8. The variable n is used to define how many queens need to be placed on the board for it to be considered complete.

PROGRAM

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())
# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queens(N)
```

```
for i in board:
    print (i)
```

OUTPUT

```
Enter the number of queens
8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

RESULT

Thus the program to implement 8 queens search strategy is implemented and executed successfully.

EX.No.1c IMPLEMENT BASIC SEARCH STRATEGIES – CRYPT ARITHMETIC
DATE:

AIM

To write a python program to implement basic search strategies – Crypt arithmetic.

ALGORITHM

1. The code starts by declaring a variable called num.
2. It is initialized to 0 and then it starts looping through the word, assigning a value of 10 for each letter in the word.
3. The assignment statement assigns an integer value to every character in the string.
4. The code continues by checking if any of the first letters are zero or if any of the last letters are zero.
5. If either one is true, then that means there was no valid assignment and so this function returns false which will cause our program to stop running at that point.
6. The code is written to find the value of a word in an assignment.
7. The code begins by declaring two variables, num and assigned.
8. The first variable, num, will store the number of times that letter appears in the word.
9. The second variable, assigned, stores the letters in the word with their corresponding values.
10. Next, a for loop is created which iterates through each character in the word.
11. In this loop we check if any letter has been set to zero or not assigned at all.
12. If either condition is true then it returns false and stops iterating through characters within that iteration of the loop.
13. Otherwise it increments num by 10 and assigns its value to assigned [char].
14. This process repeats until all characters have been checked for
15. The code is a function that takes in three arguments: word1, word2, and result.
16. The function is called solve and it returns the solutions to the equation of word1 + word2 = result.
17. The first line of code assigns values to letters so that they are sorted alphabetically by their value.
18. This is done with set () because each letter has two possible values (A-Z or a-z).
19. The next line creates an empty list named assigned which will hold all the possible assignments for words 1 and 2.
20. Next, we iterate through 10 numbers from 0-9 using range ().
21. We then check if any number not in assigned is found by checking if it's not in the list of letters (assigned) .
22. If there isn't one yet, we create a new letter at index cur letter with pop (), assign num to it with assignment operator (=), then call _solve (word1, word2, result) on our solution list created earlier
23. Then we remove cur letter from assigned so that only A-Z remain as valid characters for assigning values to words 1 and 2 respectively.
24. The code is the solution to a word puzzle.
25. It will take in two words and assign them to a result.
26. The assignment can be done by finding values in the words, or by assigning letters to numbers.
27. The first line of code assigns the value of 1 to letter "a" and 2 to letter "b".
28. This is done with find value ().
29. The next line assigns 3 to letter "c", 4 to letter "d", 5 to letter "e", 6 to letter
30. The code is trying to find the number of solutions.
31. The code starts by checking if len (result) > max (len (word1), len (word2)) + 1 or len (letters) > 10, which means that it will stop searching when either one of those conditions is met.
32. Then, the code creates a list called solutions and iterates through each solution in order to print out its contents.
33. The first line says "CRYPTARITHMETIC PUZZLE SOLVER".

34. This is just an example of what you might see on your screen after running this program.
35. The code will print 0 Solutions!
36. If the input is not a word or number.
37. The code above will then iterate through the list of solutions and print them out.

PROGRAM

```
def find_value(word, assigned):
    num = 0
    for char in word:
        num = num * 10
        num += assigned[char]
    return num
def is_valid_assignment(word1, word2, result, assigned):
    # First letter of any word cannot be zero.
    if assigned[word1[0]] == 0 or assigned[word2[0]] == 0 or assigned[result[0]] == 0:
        return False
    return True
def _solve(word1, word2, result, letters, assigned, solutions):
    if not letters:
        if is_valid_assignment(word1, word2, result, assigned):
            num1 = find_value(word1, assigned)
            num2 = find_value(word2, assigned)
            num_result = find_value(result, assigned)
            if num1 + num2 == num_result:
                solutions.append((f'{num1} + {num2} = {num_result}', assigned.copy()))
        return
    for num in range(10):
        if num not in assigned.values():
            cur_letter = letters.pop()
            assigned[cur_letter] = num
            _solve(word1, word2, result, letters, assigned, solutions)
            assigned.pop(cur_letter)
            letters.append(cur_letter)
def solve(word1, word2, result):
    letters = sorted(set(word1) | set(word2) | set(result))
    if len(result) > max(len(word1), len(word2)) + 1 or len(letters) > 10:
        print('0 Solutions!')
        return
    solutions = []
    _solve(word1, word2, result, letters, {}, solutions)
    if solutions:
        print('\nSolutions:')
        for soln in solutions:
            print(f'{soln[0]}\t{soln[1]}')
if __name__ == '__main__':
    print('CRYPTARITHMETIC PUZZLE SOLVER')
    print('WORD1 + WORD2 = RESULT')
    word1 = input('Enter WORD1: ').upper()
    word2 = input('Enter WORD2: ').upper()
    result = input('Enter RESULT: ').upper()
    if not word1.isalpha() or not word2.isalpha() or not result.isalpha():
        raise TypeError('Inputs should ony consists of alphabets.')
    solve(word1, word2, result)
```

OUTPUT

```
CRYPTARITHMETIC PUZZLE SOLVER
WORD1 + WORD2 = RESULT
Enter WORD1: SEND
Enter WORD2: MORE
Enter RESULT: MONEY
Solutions:
9567 + 1085 = 10652 {'Y': 2, 'S': 9, 'R': 8, 'O': 0, 'N': 6, 'M': 1, 'E':
5, 'D': 7}
```

RESULT

Thus the program to implement crypt arithmetic search strategy is implemented and executed successfully

Ex.No. 2 **Implement A* Algorithm**
DATE:

AIM

To write a python program to implement A* Algorithm.

ALGORITHM

1. The code is a class that is subclassed from the State class.
2. The code creates an instance of the State_String class with three arguments: value, parent, and start.
3. The goal is set to 0 because it will be used as a starting point for the path.
4. The __init__ method initializes all three variables in this new state object.
5. It also sets up two methods: GetDistance and CreateChildren.
6. The code is an example of a State class that has a parent and two children.
7. The parent will have the start and goal values, while the children will be initialized with 0 as their value.
8. The state_string class is derived from the state class, but it has one additional function: GetDistance().
9. This function calculates how far away from the current state's goal value this particular state is.
10. The code is a class that holds the final magic.
11. It has two methods: Solve and Analyze.
12. The Solve method starts by creating an empty list of nodes, which is called path.
13. Then it creates a PriorityQueue object with qsize() set to 0, then puts all the values in the queue into this new node at index 2.
14. This process repeats until there are no more nodes left in the queue or there is only one node left in the queue, which means that we have found our solution!
15. The Analyze method loops through each value from priorityQueue and checks if it's not already on visitedQueue or if its distance from goal is less than 1 (the length of visitedQueue).
16. If so, they add their children to visitedQueue and create their own children using CreateChildren().
17. The code attempts to create a class that will hold the final magic of solving a puzzle.
18. The class is called A_Star_Solver and it has two methods, __init__() and Solve().
19. The __init__() method initializes the state of the object by setting its start point, goal point, and priority queue.
20. The Solve() method visits each element in the priority queue until either there are no more elements or it finds an element with a path that matches its own.
21. When this happens, it sets itself as being visited on that path and moves on to visit all children of that node.
22. After visiting all children nodes, if there are still no paths found then it returns back to the beginning state (the start point).

PROGRAM

```
from queue import PriorityQueue
#Creating Base Class
class State(object):
    def __init__(self, value, parent, start = 0, goal = 0):
        self.children = []
        self.parent = parent
        self.value = value
        self.dist = 0
        if parent:
            self.start = parent.start
            self.goal = parent.goal
            self.path = parent.path[:]
            self.path.append(value)
        else:
            self.path = [value]
            self.start = start
            self.goal = goal
    def GetDistance(self):
        pass
    def CreateChildren(self):
        pass
# Creating subclass
class State_String(State):
    def __init__(self, value, parent, start = 0, goal = 0 ):
        super(State_String, self).__init__(value, parent, start, goal)
        self.dist = self.GetDistance()
    def GetDistance(self):
       if self.value == self.goal:
           return 0
       dist = 0
       for i in range(len(self.goal)):
           letter = self.goal[i]
           dist += abs(i - self.value.index(letter))
       return dist
    def CreateChildren(self):
        if not self.children:
            for i in range(len(self.goal)-1):
                val = self.value
                val = val[:i] + val[i+1] + val[i] + val[i+2:]
                child = State_String(val, self)
                self.children.append(child)
# Creating a class that hold the final magic
class A_Star_Solver:
    def __init__(self, start, goal):
        self.path = []
        self.vistedQueue =[]
        self.priorityQueue = PriorityQueue()
        self.start = start
        self.goal = goal
    def Solve(self):
        startState = State_String(self.start,0,self.start,self.goal)
        count = 0
        self.priorityQueue.put((0,count, startState))
        while(not self.path and self.priorityQueue.qsize()):
            closesetChild = self.priorityQueue.get()[2]
            closesetChild.CreateChildren()
            self.vistedQueue.append(closesetChild.value)
            for child in closesetChild.children:
                if child.value not in self.vistedQueue:
                    count += 1
            if not child.dist:
                self.path = child.path
                break
            self.priorityQueue.put((child.dist,count,child))
        if not self.path:
            print("Goal Of is not possible !" + self.goal )
        return self.path
# Calling all the existing stuffs
if __name__ == "__main__":
    start1 = "secure"
    goal1 = "rescue"
    print("Starting....")
    a = A_Star_Solver(start1,goal1)
    a.Solve()
    for i in range(len(a.path)):
        print("{0}){1}".format(i,a.path[i]))
```

OUTPUT

```
Starting....
0)secure
1)secrue
2)sercue
3)srecue
4)rsecue
5)rescue
```

RESULT

Thus the program to implement A* algorithm is implemented and executed successfully.

EX.No. 3　　IMPLEMENT MINI-MAX ALGORITHM FOR GAME PLAYING
DATE:　　　　　　　　(ALPHA-BETA PRUNING)

AIM

To write a python program to implement Mini-max algorithm for game playing.

ALGORITHM

1. The code first creates an array of values.
2. These values represent the possible outcomes of a mini-max calculation.
3. Next, the code calls the minimax () function to find the optimal value for the current player.
4. This function takes five arguments: depth, nodeIndex, maxTurn, scores, and targetDepth
5. The first two arguments specify how deep into the tree the minimax calculation should occur, and nodeIndex specifies which child node in that depth should be used as the starting point for this calculation.
6. The third argument is a Boolean flag that tells minimax () whether or not maximizingPlayer should be used during this calculation.
7. The code then sets up some variables to track which value in values represents best for each of MIN and MAX.
8. The code then loops through all of values' children nodes, calling minimax () on each one with different combinations of True and False as its arguments.
9. It keeps track of these results using best and val variables respectively.
10. If any of these calculations results in a better overall value than either MIN or MAX, then that new best value becomes the new minimum
11. The code will return the optimal value for the current player

PROGRAM

```
import math
def minimax (curDepth, nodeIndex,
          maxTurn, scores,
          targetDepth):
   # base case : targetDepth reached
   if (curDepth == targetDepth):
      return scores[nodeIndex]
   if (maxTurn):
      return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
              minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth))
   else:
      return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),
              minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth))
# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

OUTPUT

```
The optimal value is : 12
```

RESULT

Thus the program to implement Minimax algorithm for game playing is implemented and executed successfully

EX.No.4 **SOLVE CONSTRAINT SATISFACTION PROBLEMS**
DATE:

AIM

To write a python program to solve constraint satisfaction problems.

ALGORITHM

1. The code starts by defining some variables.
2. The first variable, assignment, is a list of strings.
3. The second variable, VARIABLES, is a list of five strings.
4. Next, the code defines some constraints.
5. The first constraint is that the two lists have the same length (len(assignment) == len(VARIABLES)).
6. The next constraint is that each string in VARIABLES must be in one of the three domains (Monday, Tuesday, and Wednesday).
7. Finally, the last constraint is that each string in DOMAIN must be associated with at least one variable in VARIABLES.
8. The code then starts to search for an assignment that meets all the constraints.
9. First it checks if assignment is complete (if len (assignment) == len (VARIABLES)) and then it selects a variable from VARIABLES based on its domain and value.
10. If there are no more variables to select, then the code returns none.
11. Otherwise it backtracks through the assignment looking for a consistent combination of values for each selected variable.
12. If successful, it assigns those values to corresponding variables in result and returns this as solution.
13. The code first looks for a variable not yet assigned in the VARIABLES list.
14. If no variable is found, it returns none.
15. If an assignment is found, the code checks to see if the variables are consistent with each other.
16. If they are not, the code backtracks and tries again with a different variable.
17. If all variables are consistent with each other, then the code assigns the value of day to var3 and returns False.

PROGRAM

```
VARIABLES = ["csc", "maths", "phy", "che", "tam", "eng", "bio"]
DOMAIN = ["Monday", "Tuesday", "Wednesday"]
CONSTRAINTS = [
("csc", "maths"),
("csc", "phy"),
("mat", "phy"),
("mat", "che"),
("mat", "tam"),
("phy", "tam"),
("phy", "eng"),
("che", "eng"),
("tam", "eng"),
("tam", "bio"),
("eng", "bio")
]
def backtrack(assignment):
   #"""Runs backtracking search to find an assignment."""
   # Check if assignment is complete
```

```
    if len(assignment) == len(VARIABLES):
        return assignment
    var = select_unassigned_variable(assignment)
    for value in DOMAIN:
        if consistent(var, value, assignment):
            assignment[var] = value
            result = backtrack(assignment)
            if result is not None:
                return result
    return None
def select_unassigned_variable(assignment):
    # """Chooses a variable not yet assigned, in order."""
    for var in VARIABLES:
        if var not in assignment.keys():
            return var
def consistent(var, value, assignment):
    #"""Checks to see if an assignment is consistent."""
    for var1, var2 in CONSTRAINTS:
        if var1 == var or var2 == var:
            for var3, day in assignment.items():
                if (var3 == var2 or var3 == var1) and day == value:
                    return False
    return True
solution = backtrack(dict())
print(solution)
```

OUTPUT

```
{'csc': 'Monday', 'maths': 'Tuesday', 'phy': 'Tuesday', 'che': 'Monday', 'tam': 'MoMonday', 'eng': 'Wednesday',
'bio': 'Tuesday'}
```

RESULT

Thus the program to solve constraint satisfaction problem is implemented and executed successfully.

EX.No. 5 **PROPOSITIONAL MODEL CHECKING ALGORITHMS**
DATE:

AIM

To write a python program to implement Propositional Model checking Algorithm.

ALGORITHM

1. Class Literal, it has attributes name and sign to denote whether the literal is positive or negative in use.
2. The __neg__ function returns a new literal with the same name but the opposite sign of its parent literal.
3. The __repr__ function returns the string of the literal name,(or the string with a negative sign) each time the instance of the literal is called.
4. The CNFConvert function converts the KiB from a list of sets to a list of list for easier computing
5. The VariableSet function finds all the used literals in the KB, and in order to assist with running the DPLL.
6. The Negativeofx function is for holding the negative form of the literal, for use in the DPLL algorithm
7. The pickX function picks a literal from the variable set and works with it as a node in the tree.
8. Now define the functions splitfalseliterals() and splitTrueLiteral().
9. Create the function dpll() that performs the dpll algorithm recursively.
10. Finally call the function to execute the code.

PROGRAM

```
import re
class Literal:
# Class Literal, it has attributes name and sign to denote whether the literal is positive or negative in use
    def __init__(self, name, sign=True):
        self.name = str(name)
        self.sign = sign
    def __neg__(self): # returns a new literal with the same name but the opposite sign of its parent literal
        return Literal(self.name, False)
    def __str__(self):
        return str(self.name)
def __repr__(self):
    # returns the string of the literal name,( or the string with a negative sign) each time the instance of the
literal is called
    if self.sign:
        return '%r' % str(self.__str__())
    else:
        return '%r' % str("-" + self.__str__())
def CNFconvert(KB):
    # This function converts the Kb from a list of sets to a list of list for easier computing
    storage=[]
    for i in KB:
        i=list(i)
        for j in i:
            j=str(j)
        storage.append(i)
    return storage
def VariableSet(KB):
    # This function finds all the used literals in the KB, and in order to assist with running the DPLL
    KB=eval((CNFconvert(KB).__str__()))
    storage=[]
    for obj in KB:
        for item in obj:
            if item[0] == '-' and item[1:] not in storage:
                storage.append(str(item[1:]))
            elif item not in storage and item[0] != '-':
                storage.append(str(item))
    return storage
def Negativeofx(x):
    # This function is for holding the negative form of the literal, for use in the DPLL algorithm
    check=re.match("-", str(x))
    if (check):
        return str(x[1:])
    else:
        return "-"+str(x)
def pickX(literals, varList):
    # This function picks a literal from the variable set and works with it as a node in the tree
    for x in varList:
        if x not in literals:
            break
        return x
def splitFalseLiterals(cnf, x):
    holder=[]
    for item in cnf:
        if x in item:
            item.remove(x)
        holder.append(item)
    return holder
def splitTrueLiteral(cnf, x):
    holder=[]
    for item in cnf:
        if x in item:
            continue
        else:
            holder.append(item)
        return holder
def unitResolution(clauses):
    literalholder={} # dictionary for holding the literal holder and their
    bool
    i=0
    # This part of the code goes through each and every clause until the all literals in the KB are resolved
    while i < len(clauses): # for each clause
        newClauses=[]
        clause=clauses[i]
        # picks a clause to work on
        if (len(clause) == 1):
            literal=str(clause[0])
            pattern=re.match("-", literal)
            # Populates the dictionary
        if (pattern):
            nx=literal[1:]
            literalholder[nx]=False
        else:
            nx="-"+literal
            literalholder[literal]=True
        # Checks for all other appearances o the literal or its opposite int he KB
        for item in clauses:
            if item != clauses[i]:
                if (nx in item):
                    item.remove(nx)
                    newClauses.append(item)
                i=0
                clauses=newClauses
            # no unit clause
            else:
                i += 1
    return literalholder, clauses
def dpll(clauses, varList):
    # recursively performs the dpll algorithm
    literals, cnf=unitResolution(clauses)
    if (cnf == []):
        return literals
    elif ([] in cnf):
        return "notsatisfiable"
    else:
        # pick a literal which isn't set yet but has an impact on the Kb, and then work on it recursively
        while True:
            x=pickX(literals, varList)
            x=str(x)
            nx=Negativeofx(x)
            ncnf=splitTrueLiteral(cnf, x)
            ncnf=splitFalseLiterals(ncnf, nx)
            if ncnf == cnf:
                varList.remove(x)
            else:
                break
        # does the same dpll recursively, but follows the true path for that variable
        case1=dpll(ncnf, varList)
        if (case1 != "notsatisfiable"):
            copy=case1.copy()
            copy.update(literals)
            copy.update({x: True})
            return copy
        # does the dpll recursively, but follows the false path for that variable
        case1=dpll(ncnf, varList)
        if not case1:
            copy=case1.copy()
            copy.update(literals)
            copy.update({x: False})
            return copy
        else:
            return "notsatisfiable"
def DPLL(KB):
# Finally restructures the output to fit the required output by the assignment description
    KB=eval((CNFconvert(KB).__str__()))
    varList=VariableSet(KB)
    result=dpll(KB, varList)
    if result == 'notsatisfiable':
        False
    else:
        for i in varList:
            if i in result and result[i] == True:
                result[i]='true'
            elif i in result and result[i] == False:
                result[i]='false'
            else:
                result[i]='free'
        return [True, result]
    A=Literal('A')
    B=Literal('B')
    C=Literal('C')
    D=Literal('D')
    KB=[{A, B}, {A, -C}, {-A, B, D}]
    print(DPLL(KB))
```

OUTPUT

```
[True, {'B': 'true', 'A': True, 'C': 'free', 'D': 'free'}]
```

RESULT

Thus the program to implement Propositional Model checking Algorithm is implemented and executed successfully.

EX.No. 6a **IMPLEMENT FORWARD CHAINING ALGORITHM**
DATE:

AIM

To write a python program toiImplement Forward Chaining Algorithm.

ALGORITHM

1. The code starts by declaring a variable called "x".
2. This is the number of items in the database.
3. The code then declares two variables, "database" and "knowbase".
4. The first one is an array that stores all the words in the database.
5. The second one is an array that stores all the colors of objects in our world.
6. Next, it prints out a message saying *-----Forward--Chaining-----* to let us know what's going on.
7. Then it displays some text with instructions for how to use this program before displaying any output from it (i.e., telling us what we're supposed to do).
8. It also tells us which input option will be valid if you enter 1 or 2 as your input value (in this case, frog0 and green1).
9. If you enter 3 or 4 instead, then they'll tell you about canary1 and yellow3 respectively.
10. Else it prints invalid knowledge database

PROGRAM

```
database = ["Croaks", "Eat Flies", "Shrimps", "Sings"]
knowbase = ["Frog", "Canary", "Green", "Yellow"]
def display():
    print("\n X is \n1..Croaks \n2.Eat Flies \n3.shrimps \n4.Sings ", end='')
    print("\n Select One ", end='')
def main():
    print("*-----Forward--Chaining-----*", end='')
    display()
    x = int(input())
    print(" \n", end='')
    if x == 1 or x == 2:
        print(" Chance Of Frog ", end='')
    elif x == 3 or x == 4:
        print(" Chance of Canary ", end='')
    else:
        print("\n-------In Valid Option Select --------", end='')
    if x >= 1 and x <= 4:
        print("\n X is ", end='')
        print(database[x-1], end='')
        print("\n Color Is 1.Green 2.Yellow", end='')
        print("\n Select Option ", end='')
        k = int(input())
        if k == 1 and (x == 1 or x == 2): # frog0 and green1
            print(" yes it is ", end='')
            print(knowbase[0], end='')
            print(" And Color Is ", end='')
            print(knowbase[2], end='')
        elif k == 2 and (x == 3 or x == 4): # canary1 and yellow3
            print(" yes it is ", end='')
            print(knowbase[1], end='')
            print(" And Color Is ", end='')
            print(knowbase[3], end='')
        else:
            print("\n---InValid Knowledge Database", end='')
if __name__ == "__main__":
    main()
```

OUTPUT

```
*-----Forward--Chaining-----*
X is
1.Croaks
2.Eat Flies
3.shrimps
4.Sings
Select One 1
Chance Of Frog
X is Croaks
Color Is
1.Green
2.Yellow
Select Option 1
yes it is Frog And Color Is Green
```

RESULT

Thus the program to implement Forward Chaining Algorithm is implemented and executed successfully

EX.No. 6b **IMPLEMENT BACKWARD CHAINING ALGORITHM**
DATE:

AIM

To write a python program to implement backward chaining algorithm

ALGORITHM

1. The code starts with a function called display () which prints out the text "X is 1.frog 2.canary"
2. And then asks for input from the user, asking them to select one of two options: Chance of eating flies or Chance of shrimping.
3. If they choose either option, it will print out what that option would be in green or yellow color respectively.
4. The next line is x = int (input ()) which takes the value entered by the user and assigns it to variable x.
5. The if statement checks whether x equals 1 or 2
6. So if they enter 1 as their answer, it will print out "Chance of eating flies"
7. Otherwise it will print "Chance of shrimping".

PROGRAM

```
database = ["Croaks", "Eat Flies", "Shrimps", "Sings"]
knowbase = ["Frog", "Canary"]
color = ["Green", "Yellow"]
def display():
    print("\n X is \n1.frog \n2.canary ", end='')
    print("\n Select One ", end='')
def main():
    print("*-----Backward--Chaining-----*", end='')
    display()
    x = int(input())
    print(" \n", end='')
    if x == 1:
        print(" Chance Of eating flies ", end='')
    elif x == 2:
        print(" Chance of shrimping ", end='')
    else:
        print("\n-------In Valid Option Select --------", end='')
    if x >= 1 and x <= 2:
        print("\n X is ", end='')
        print(knowbase[x-1], end='')
        print("\n1.green \n2.yellow")
        k = int(input())
        if k == 1 and x == 1: # frog0 and green1
            print(" yes it is in ", end='')
            print(color[0], end='')
            print(" colour and will ", end='')
            print(database[0])
        elif k == 2 and x == 2: # canary1 and yellow3
            print(" yes it is in", end='')
            print(color[1], end='')
            print(" Colour and will ", end='')
            print(database[1])
        else:
            print("\n---InValid Knowledge Database", end='')
if __name__ == "__main__":
    main()
```

OUTPUT

```
*-----Backward--Chaining-----*
X is
1.frog
2.canary
Select One 1
Chance Of eating flies
X is Frog
1.green
2.yellow
1
yes it is in Green colour and will Croak
```

RESULT

Thus the program to implement backward chaining algorithm is implemented and executed successfully.

EX.No. 7 **IMPLEMENT NAÏVE BAYES MODELS**
DATE:

AIM

To write a python program to implement Naïve Bayes Models.

ALGORITHM

1. The code starts by loading the iris dataset.
2. The data is then split into a training and test set of equal size.
3. Next, a Gaussian Naive Bayes classifier is trained using the training set.
4. Then predictions are made on the test set with accuracy scores calculated for each prediction.
5. Finally, a confusion matrix is created to show how well each prediction was classified as correct or incorrect
6. The code is used to train a Gaussian Naive Bayes classifier and then use it to make predictions.
7. The code prints the model's predictions, as well as the test set's output for comparison.

PROGRAM

```
from sklearn import datasets
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from pandas import DataFrame
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
# The dataset
iris = datasets.load_iris()
X = iris.data
Y = iris.target
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=1/3)
# Training a Gaussian Naive Bayes classifier
model = GaussianNB()
model.fit(X_train, Y_train)
# Predictions
model_predictions = model.predict(X_test)
print("\n",model_predictions)
print("\n",Y_test)
# Accuracy of prediction
accuracyScore = accuracy_score(Y_test, model_predictions)
print("\naccuracyScore is",accuracyScore )
# Creating a confusion matrix
cm=confusion_matrix(Y_test,model_predictions)
print("\nconfusion matrix",cm)
```

OUTPUT

```
[0 2 1 1 0 0 2 2 1 2 1 0 1 2 0 2 2 0 2 1 1 2 2 0 1 0 0 2 2 0 0 2 0 1 1 1 1
1 2 1 1 0 0 2 1 0 1 0 2 2]
[0 2 1 1 0 0 2 2 1 2 1 0 1 2 0 2 2 0 2 1 1 2 2 0 1 0 0 2 2 0 0 2 0 1 1 1 1
1 2 1 1 0 0 2 1 0 1 0 2 2]
accuracyScore is 1.0
confusion matrix [[16 0 0]
[ 0 17 0]
[ 0 0 17]]
```

RESULT

Thus the program to implement Naïve Bayes Model is implemented and executed successfully.

EX.No. 8 IMPLEMENT BAYESIAN NETWORKS AND PERFORM INFERENCES
DATE:

AIM:

To write a python program to implement Bayesian Networks and perform inferences.

ALGORITHM:

1. The code starts by importing the datasets module.
2. This is used to load the iris dataset from sklearn, which is then passed into a variable called data.
3. The target and data_tensor are then created with torch's numpy library.
4. The code continues by creating an instance of a sequential model that has three layers:
1. bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=4), nn.ReLU(), and bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=100).
5. The cross entropy loss function is then created as well as an instance of it being assigned to the cross entropy loss variable named crossEntropyLoss().
6. The code will create a model that takes in 100 features and outputs 3.
7. The code starts by importing the necessary libraries.
8. Then it creates a model that will be used to predict the target variable, which is the number of people who are likely to buy a product in one month.
9. The code then calculates how much money it would cost for this prediction and compares it with what's actually spent on advertising.
10. The code starts by creating an optimizer object that uses Adam as its learning algorithm and has a parameter of 0.01 for its learning rate.
11. It then loops through 3000 steps, each step taking about 1 second, where models are created from data_tensor using torch's max function and predicted is calculated using torch's max function on models' data tensor (which is just another way of saying "the most recent value").
12. Cross entropy loss is calculated between predictions and targets using cross_entropy_loss
13. Finally, kloss , which stands for Kullback-Leibler divergence, is calculated with klloss .
14. This loss measures how different two probability distributions are from each other; in this case we're measuring how different our predictions are from their actual values (i.e., whether they were correct).
15. Total cost is then calculated by adding cross entropy plus klweight*kl .
16. Finally, optim
17. The code is used to predict the output of a model given input data.
18. The code starts by initializing the variables that will be needed for later calculations.
19. Next, it calculates the cross entropy loss between target and predicted values.
20. Then, it calculates the cost function which is then minimized using Adam optimizer.
21. Finally, it prints out the predicted value and total cost after every iteration of optimization process.
22. The code starts by defining a function called draw_graph that takes in a predicted value.
23. The code then creates two subplots on the same figure, one for each of the predictions.
24. The first subplot is created with fig_1 and has an index of 1, which means it's the second plot in this figure.
25. This plot will have two axes: x-axis and y-axis.
26. The x-axis represents time, while the y-axis represents accuracy percentage (0% to 100%).
27. The second subplot is created with fig_2 and has an index of 2, which means it's the third plot in this figure.
28. This plot will also have two axes: x-axis and y-axis but they represent different values than those on fig_1 .
29. The code is a function that takes in an input of predicted and returns the accuracy, cross entropy, and KL values.

30. The first line of code calculates the size of the tensor using target_tensor.size(0) which will be equal to 1 because it is a one-dimensional tensor.
31. Next, we have the function draw_graph which draws a graph with two subplots; one for accuracy and one for cross entropy.
32. The last line prints out some statistics on how accurate this prediction was.

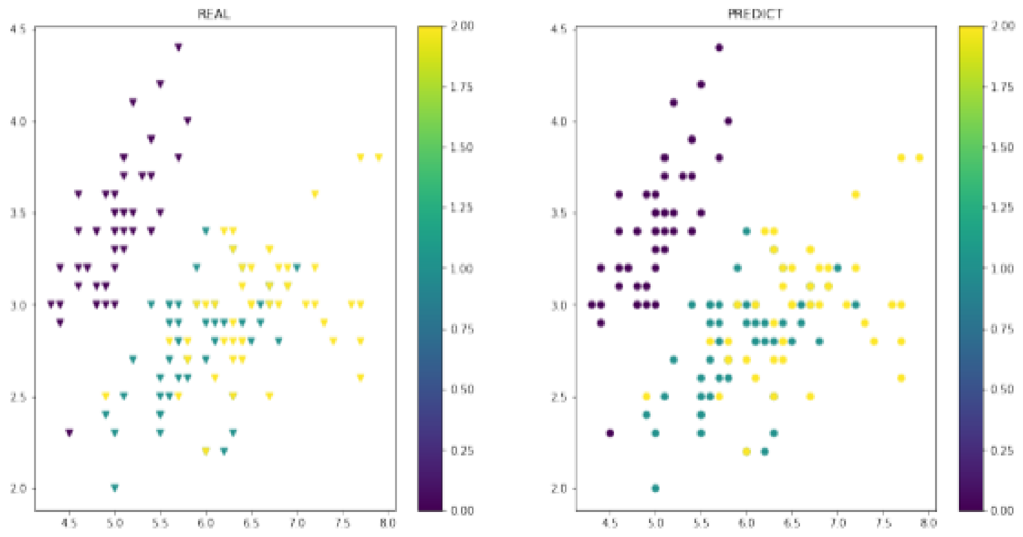
PROGRAM

```
import numpy as np
from sklearn import datasets
import torch
import torch.nn as nn
import torch.optim as optim
import torchbnn as bnn
import matplotlib.pyplot as plt
dataset = datasets.load_iris()
data = dataset.data
target = dataset.target
data_tensor = torch.from_numpy(data).float()
target_tensor = torch.from_numpy(target).long()
model = nn.Sequential(bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=4, out_features=100),
nn.ReLU(), bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=100, out_features=3), )
cross_entropy_loss = nn.CrossEntropyLoss()
klloss = bnn.BKLLoss(reduction='mean', last_layer_only=False)
klweight = 0.01
optimizer = optim.Adam(model.parameters(), lr=0.01)
for step in range(3000):
        models = model(data_tensor)
        cross_entropy = cross_entropy_loss(models, target_tensor)
        kl = klloss(model)
        total_cost = cross_entropy + klweight*kl
        optimizer.zero_grad()
        total_cost.backward()
        optimizer.step()
_, predicted = torch.max(models.data, 1)
final = target_tensor.size(0)
correct = (predicted == target_tensor).sum()
print('- Accuracy: %f %%' % (100 * float(correct) / final))
print('- CE : %2.2f, KL : %2.2f' % (cross_entropy.item(), kl.item()))
def draw_graph(predicted):
        fig = plt.figure(figsize=(16, 8))
        fig_1 = fig.add_subplot(1, 2, 1)
        fig_2 = fig.add_subplot(1, 2, 2)
        z1_plot = fig_1.scatter(data[:, 0], data[:, 1], c=target, marker='v')
        z2_plot = fig_2.scatter(data[:, 0], data[:, 1], c=predicted)
        plt.colorbar(z1_plot, ax=fig_1)
        plt.colorbar(z2_plot, ax=fig_2)
        fig_1.set_title("REAL")
        fig_2.set_title("PREDICT")
        plt.show()
models = model(data_tensor)
```

```
_, predicted = torch.max(models.data, 1)
draw_graph(predicted)
```

OUTPUT

```
- Accuracy: 96.000000 %
- CE : 0.08, KL : 3.20
```



RESULT

Thus, the program to implement Bayesian Networks and perform inferences is implemented and executed successfully.