AD3311-ARTIFICIALINTELLIGENCELABORATORY

**(Regulations2021)**

**SET-1&4 SOLVED PROGRAMS**

1. **Consider you have a 3X3 board, where all numbers are in range 0 to 8. The board has no repeating numbers and an empty space. Implement a program to find the minimum number of steps required to arrange the board in sequence using 8 puzzle problem.**

**PROGRAM:**

# Python3 program to print the path from root

# node to destination node for N\*N-1 puzzle

# algorithm using Branch and Bound

# The solution assumes that instance of

# puzzle is solvable

# Importing copy for deepcopy function

import copy

# Importing the heap functions from python

# library for Priority Queue

from heapq import heappush, heappop

# This variable can be changed to change

# the program from 8 puzzle(n=3) to 15

# puzzle(n=4) to 24 puzzle(n=5)...

n = 3

# bottom, left, top, right

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]

# A class for Priority Queue

class priorityQueue:

# Constructor to initialize a

# Priority Queue

def \_\_init\_\_(self):

self.heap = []

# Inserts a new key 'k'

def push(self, k):

heappush(self.heap, k)

# Method to remove minimum element

# from Priority Queue

def pop(self):

return heappop(self.heap)

# Method to know if the Queue is empty

def empty(self):

if not self.heap:

return True

else:

return False

# Node structure

class node:

def \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

cost, level):

# Stores the parent node of the

# current node helps in tracing

# path when the answer is found

self.parent = parent

# Stores the matrix

self.mat = mat

# Stores the position at which the

# empty space tile exists in the matrix

self.empty\_tile\_pos = empty\_tile\_pos

# Stores the number of misplaced tiles

self.cost = cost

# Stores the number of moves so far

self.level = level

# This method is defined so that the

# priority queue is formed based on

# the cost variable of the objects

def \_\_lt\_\_(self, nxt):

return self.cost < nxt.cost

# Function to calculate the number of

# misplaced tiles ie. number of non-blank

# tiles not in their goal position

def calculateCost(mat, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mat[i][j]) and

(mat[i][j] != final[i][j])):

count += 1

return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

level, parent, final) -> node:

# Copy data from parent matrix to current matrix

new\_mat = copy.deepcopy(mat)

# Move tile by 1 position

x1 = empty\_tile\_pos[0]

y1 = empty\_tile\_pos[1]

x2 = new\_empty\_tile\_pos[0]

y2 = new\_empty\_tile\_pos[1]

new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1]

# Set number of misplaced tiles

cost = calculateCost(new\_mat, final)

new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

cost, level)

return new\_node

# Function to print the N x N matrix

def printMatrix(mat):

for i in range(n):

for j in range(n):

print("%d " % (mat[i][j]), end = " ")

print()

# Function to check if (x, y) is a valid

# matrix coordinate

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatrix(root.mat)

print()

# Function to solve N\*N - 1 puzzle algorithm

# using Branch and Bound. empty\_tile\_pos is

# the blank tile position in the initial state.

def solve(initial, empty\_tile\_pos, final):

# Create a priority queue to store live

# nodes of search tree

pq = priorityQueue()

# Create the root node

cost = calculateCost(initial, final)

root = node(None, initial,

empty\_tile\_pos, cost, 0)

# Add root to list of live nodes

pq.push(root)

# Finds a live node with least cost,

# add its children to list of live

# nodes and finally deletes it from

# the list.

while not pq.empty():

# Find a live node with least estimated

# cost and delete it from the list of

# live nodes

minimum = pq.pop()

# If minimum is the answer node

if minimum.cost == 0:

# Print the path from root to

# destination;

printPath(minimum)

return

# Generate all possible children

for i in range(4):

new\_tile\_pos = [

minimum.empty\_tile\_pos[0] + row[i],

minimum.empty\_tile\_pos[1] + col[i], ]

if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

# Create a child node

child = newNode(minimum.mat,

minimum.empty\_tile\_pos,

new\_tile\_pos,

minimum.level + 1,

minimum, final,)

# Add child to list of live nodes

pq.push(child)

# Driver Code

# Initial configuration

# Value 0 is used for empty space

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

# Solvable Final configuration

# Value 0 is used for empty space

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

# Blank tile coordinates in

# initial configuration

empty\_tile\_pos = [ 1, 2 ]

# Function call to solve the puzzle

solve(initial, empty\_tile\_pos, final)

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

1 2 3

5 6 0

7 8 4

1 2 3

5 0 6

7 8 4

1 2 3

5 8 6

7 0 4

1 2 3

5 8 6

0 7 4

Process finished with exit code 0

1. **Implement forward chaining and resolution strategies.**

**PROGRAM:**

global facts

global is\_changed

is\_changed = True

facts = [["vertebrate","duck"],["flying","duck"],["mammal","cat"]]

def assert\_fact(fact):

global facts

global is\_changed

if not fact in facts:

facts += [fact]

is\_changed = True

while is\_changed:

is\_changed = False

for A1 in facts:

if A1[0] == "mammal":

assert\_fact(["vertebrate",A1[1]])

if A1[0] == "vertebrate":

assert\_fact(["animal",A1[1]])

if A1[0] == "vertebrate" and ["flying",A1[1]] in facts:

assert\_fact(["bird",A1[1]])

print(facts)

**OUTPUT::**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

[['vertebrate', 'duck'], ['flying', 'duck'], ['mammal', 'cat'], ['animal', 'duck'], ['bird', 'duck'], ['vertebrate', 'cat'], ['animal', 'cat']]

Process finished with exit code 0

1. **Implementacryptarithmeticpuzzletofinddigitsthatreplaceletterstomakemathematical statements true. Consider the puzzle SEND + MORE = MONEY.**

**PROGRAM:**

import itertools

def solve2():

letters = ('s', 'e', 'n', 'd', 'm', 'o', 'r', 'y')

digits = range(10)

for perm in itertools.permutations(digits, len(letters)):

sol = dict(zip(letters, perm))

if sol['s'] == 0 or sol['m'] == 0:

continue

send = 1000 \* sol['s'] + 100 \* sol['e'] + 10 \* sol['n'] + sol['d']

more = 1000 \* sol['m'] + 100 \* sol['o'] + 10 \* sol['r'] + sol['e']

money = 10000 \* sol['m'] + 1000 \* sol['o'] + 100 \* sol['n'] + 10 \* sol['e'] + sol['y']

if send + more == money:

print(" send"," more"," money")

return send, more, money

print(solve2())

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

send more money

(9567, 1085, 10652)

Process finished with exit code 0

1. **Write a python program to solve tic–tac–toe problem using MINI-MAX algorithm. Find the next optical move for a player.**

**PROGRAM:**

# Python3 program to find the next optimal move for a player

player, opponent = 'x', 'o'

# This function returns true if there are moves

# remaining on the board. It returns false if

# there are no moves left to play.

def isMovesLeft(board) :

for i in range(3) :

for j in range(3) :

if (board[i][j] == '\_') :

return True

return False

# This is the evaluation function as discussed

# in the previous article ( http://goo.gl/sJgv68 )

def evaluate(b) :

# Checking for Rows for X or O victory.

for row in range(3) :

if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :

if (b[row][0] == player) :

return 10

elif (b[row][0] == opponent) :

return -10

# Checking for Columns for X or O victory.

for col in range(3) :

if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

if (b[0][col] == player) :

return 10

elif (b[0][col] == opponent) :

return -10

# Checking for Diagonals for X or O victory.

if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

if (b[0][0] == player) :

return 10

elif (b[0][0] == opponent) :

return -10

if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

if (b[0][2] == player) :

return 10

elif (b[0][2] == opponent) :

return -10

# Else if none of them have won then return 0

return 0

# This is the minimax function. It considers all

# the possible ways the game can go and returns

# the value of the board

def minimax(board, depth, isMax) :

score = evaluate(board)

# If Maximizer has won the game return his/her

# evaluated score

if (score == 10) :

return score

# If Minimizer has won the game return his/her

# evaluated score

if (score == -10) :

return score

# If there are no more moves and no winner then

# it is a tie

if (isMovesLeft(board) == False) :

return 0

# If this maximizer's move

if (isMax) :

best = -1000

# Traverse all cells

for i in range(3) :

for j in range(3) :

# Check if cell is empty

if (board[i][j]=='\_') :

# Make the move

board[i][j] = player

# Call minimax recursively and choose

# the maximum value

best = max( best, minimax(board, depth + 1, not isMax) )

# Undo the move

board[i][j] = '\_'

return best

# If this minimizer's move

else :

best = 1000

# Traverse all cells

for i in range(3) :

for j in range(3) :

# Check if cell is empty

if (board[i][j] == '\_') :

# Make the move

board[i][j] = opponent

# Call minimax recursively and choose

# the minimum value

best = min(best, minimax(board, depth + 1, not isMax))

# Undo the move

board[i][j] = '\_'

return best

# This will return the best possible move for the player

def findBestMove(board) :

bestVal = -1000

bestMove = (-1, -1)

# Traverse all cells, evaluate minimax function for

# all empty cells. And return the cell with optimal

# value.

for i in range(3) :

for j in range(3) :

# Check if cell is empty

if (board[i][j] == '\_') :

# Make the move

board[i][j] = player

# compute evaluation function for this

# move.

moveVal = minimax(board, 0, False)

# Undo the move

board[i][j] = '\_'

# If the value of the current move is

# more than the best value, then update

# best/

if (moveVal > bestVal) :

bestMove = (i, j)

bestVal = moveVal

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

print()

return bestMove

# Driver code

board = [

[ 'x', 'o', 'x' ],

[ 'o', 'o', 'x' ],

[ '\_', '\_', '\_' ]

]

bestMove = findBestMove(board)

print("The Optimal Move is :")

print("ROW:", bestMove[0], " COL:", bestMove[1])

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

The value of the best Move is : 10

The Optimal Move is :

ROW: 2 COL: 2

Process finished with exit code 0

1. **Using heuristic methods write A\* search algorithm to achieve optimality and completeness in graph traversals.**

**PROGRAM:**

import math  
import heapq  
  
# Define the Cell class  
class Cell:  
 def \_\_init\_\_(self):  
 self.parent\_i = 0 # Parent cell's row index  
 self.parent\_j = 0 # Parent cell's column index  
 self.f = float('inf') # Total cost of the cell (g + h)  
 self.g = float('inf') # Cost from start to this cell  
 self.h = 0 # Heuristic cost from this cell to destination  
  
# Define the size of the grid  
ROW = 9  
COL = 10  
  
# Check if a cell is valid (within the grid)  
def is\_valid(row, col):  
 return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)  
  
# Check if a cell is unblocked  
def is\_unblocked(grid, row, col):  
 return grid[row][col] == 1  
  
# Check if a cell is the destination  
def is\_destination(row, col, dest):  
 return row == dest[0] and col == dest[1]  
  
# Calculate the heuristic value of a cell (Euclidean distance to destination)  
def calculate\_h\_value(row, col, dest):  
 return ((row - dest[0]) \*\* 2 + (col - dest[1]) \*\* 2) \*\* 0.5  
  
# Trace the path from source to destination  
def trace\_path(cell\_details, dest):  
 print("The Path is ")  
 path = []  
 row = dest[0]  
 col = dest[1]  
  
 # Trace the path from destination to source using parent cells  
 while not (cell\_details[row][col].parent\_i == row and cell\_details[row][col].parent\_j == col):  
 path.append((row, col))  
 temp\_row = cell\_details[row][col].parent\_i  
 temp\_col = cell\_details[row][col].parent\_j  
 row = temp\_row  
 col = temp\_col  
  
 # Add the source cell to the path  
 path.append((row, col))  
 # Reverse the path to get the path from source to destination  
 path.reverse()  
  
 # Print the path  
 for i in path:  
 print("->", i, end=" ")  
 print()  
  
# Implement the A\* search algorithm  
def a\_star\_search(grid, src, dest):  
 # Check if the source and destination are valid  
 if not is\_valid(src[0], src[1]) or not is\_valid(dest[0], dest[1]):  
 print("Source or destination is invalid")  
 return  
  
 # Check if the source and destination are unblocked  
 if not is\_unblocked(grid, src[0], src[1]) or not is\_unblocked(grid, dest[0], dest[1]):  
 print("Source or the destination is blocked")  
 return  
  
 # Check if we are already at the destination  
 if is\_destination(src[0], src[1], dest):  
 print("We are already at the destination")  
 return  
  
 # Initialize the closed list (visited cells)  
 closed\_list = [[False for \_ in range(COL)] for \_ in range(ROW)]  
 # Initialize the details of each cell  
 cell\_details = [[Cell() for \_ in range(COL)] for \_ in range(ROW)]  
  
 # Initialize the start cell details  
 i = src[0]  
 j = src[1]  
 cell\_details[i][j].f = 0  
 cell\_details[i][j].g = 0  
 cell\_details[i][j].h = 0  
 cell\_details[i][j].parent\_i = i  
 cell\_details[i][j].parent\_j = j  
  
 # Initialize the open list (cells to be visited) with the start cell  
 open\_list = []  
 heapq.heappush(open\_list, (0.0, i, j))  
  
 # Initialize the flag for whether destination is found  
 found\_dest = False  
  
 # Main loop of A\* search algorithm  
 while len(open\_list) > 0:  
 # Pop the cell with the smallest f value from the open list  
 p = heapq.heappop(open\_list)  
  
 # Mark the cell as visited  
 i = p[1]  
 j = p[2]  
 closed\_list[i][j] = True  
  
 # For each direction, check the successors  
 directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]  
 for dir in directions:  
 new\_i = i + dir[0]  
 new\_j = j + dir[1]  
  
 # If the successor is valid, unblocked, and not visited  
 if is\_valid(new\_i, new\_j) and is\_unblocked(grid, new\_i, new\_j) and not closed\_list[new\_i][new\_j]:  
 # If the successor is the destination  
 if is\_destination(new\_i, new\_j, dest):  
 # Set the parent of the destination cell  
 cell\_details[new\_i][new\_j].parent\_i = i  
 cell\_details[new\_i][new\_j].parent\_j = j  
 print("The destination cell is found")  
 # Trace and print the path from source to destination  
 trace\_path(cell\_details, dest)  
 found\_dest = True  
 return  
 else:  
 # Calculate the new f, g, and h values  
 g\_new = cell\_details[i][j].g + 1.0  
 h\_new = calculate\_h\_value(new\_i, new\_j, dest)  
 f\_new = g\_new + h\_new  
  
 # If the cell is not in the open list or the new f value is smaller  
 if cell\_details[new\_i][new\_j].f == float('inf') or cell\_details[new\_i][new\_j].f > f\_new:  
 # Add the cell to the open list  
 heapq.heappush(open\_list, (f\_new, new\_i, new\_j))  
 # Update the cell details  
 cell\_details[new\_i][new\_j].f = f\_new  
 cell\_details[new\_i][new\_j].g = g\_new  
 cell\_details[new\_i][new\_j].h = h\_new  
 cell\_details[new\_i][new\_j].parent\_i = i  
 cell\_details[new\_i][new\_j].parent\_j = j  
  
 # If the destination is not found after visiting all cells  
 if not found\_dest:  
 print("Failed to find the destination cell")  
  
def main():  
 # Define the grid (1 for unblocked, 0 for blocked)  
 grid = [  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],  
 [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],  
 [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],  
 [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],  
 [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],  
 [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],  
 [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],  
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],  
 [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]  
 ]  
  
 # Define the source and destination  
 src = [8, 0]  
 dest = [0, 0]  
  
 # Run the A\* search algorithm  
 a\_star\_search(grid, src, dest)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

The destination cell is found

The Path is

-> (8, 0) -> (7, 0) -> (6, 0) -> (5, 0) -> (4, 1) -> (3, 2) -> (2, 1) -> (1, 0) -> (0, 0)

Process finished with exit code 0

1. **Identify the constraints to find a word in crossword puzzle. Implement a cross word puzzle problem using Constraint Satisfaction Problem.**

**PROGRAM:**

class CSP:

def \_\_init\_\_(self, variables, Domains,constraints):

self.variables = variables

self.domains = Domains

self.constraints = constraints

self.solution = None

def solve(self):

assignment = {}

self.solution = self.backtrack(assignment)

return self.solution

def backtrack(self, assignment):

if len(assignment) == len(self.variables):

return assignment

var = self.select\_unassigned\_variable(assignment)

for value in self.order\_domain\_values(var, assignment):

if self.is\_consistent(var, value, assignment):

assignment[var] = value

result = self.backtrack(assignment)

if result is not None:

return result

del assignment[var]

return None

def select\_unassigned\_variable(self, assignment):

unassigned\_vars = [var for var in self.variables if var not in assignment]

return min(unassigned\_vars, key=lambda var: len(self.domains[var]))

def order\_domain\_values(self, var, assignment):

return self.domains[var]

def is\_consistent(self, var, value, assignment):

for constraint\_var in self.constraints[var]:

if constraint\_var in assignment and assignment[constraint\_var] == value:

return False

return True

puzzle = [[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

[7, 0, 0, 0, 2, 0, 0, 0, 6],

[0, 6, 0, 0, 0, 0, 2, 8, 0],

[0, 0, 0, 4, 1, 9, 0, 0, 5],

[0, 0, 0, 0, 8, 0, 0, 0, 0]

]

def print\_sudoku(puzzle):

for i in range(9):

if i % 3 == 0 and i != 0:

print("- - - - - - - - - - - ")

for j in range(9):

if j % 3 == 0 and j != 0:

print(" | ", end="")

print(puzzle[i][j], end=" ")

print()

print\_sudoku(puzzle)

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

5 3 0 | 0 7 0 | 0 0 0

6 0 0 | 1 9 5 | 0 0 0

0 9 8 | 0 0 0 | 0 6 0

- - - - - - - - - - -

8 0 0 | 0 6 0 | 0 0 3

4 0 0 | 8 0 3 | 0 0 1

7 0 0 | 0 2 0 | 0 0 6

- - - - - - - - - - -

0 6 0 | 0 0 0 | 2 8 0

0 0 0 | 4 1 9 | 0 0 5

0 0 0 | 0 8 0 | 0 0 0

Process finished with exit code 0

1. **Import necessary libraries, load required training and test data, and write a program for text classification using Naive Bayes model.**

**PROGRAM:**

# load the iris dataset

from sklearn.datasets import load\_iris

iris = load\_iris()

# store the feature matrix (X) and response vector (y)

X = iris.data

y = iris.target

# splitting X and y into training and testing sets

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=1)

# training the model on training set

from sklearn.naive\_bayes import GaussianNB

gnb = GaussianNB()

gnb.fit(X\_train, y\_train)

# making predictions on the testing set

y\_pred = gnb.predict(X\_test)

# comparing actual response values (y\_test) with predicted response values (y\_pred)

from sklearn import metrics

print("Gaussian Naive Bayes model accuracy(in %):", metrics.accuracy\_score(y\_test, y\_pred)\*100)

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

Gaussian Naive Bayes model accuracy(in %): 95.0

Process finished with exit code 0

1. **Consider you have 4queens, try to arrange them in chessboard using python program where no queen attack each other.**

**PROGRAM:**

N = 4

# ld is an array where its indices indicate row-col+N-1

ld = [0] \* 30

# rd is an array where its indices indicate row+col

rd = [0] \* 30

# Column array where its indices indicate column

cl = [0] \* 30

# A utility function to print solution

def printSolution(board):

for i in range(N):

for j in range(N):

print(" Q " if board[i][j] == 1 else " . ", end="")

print()

# A recursive utility function to solve N Queen problem

def solveNQUtil(board, col):

# Base case: If all queens are placed, return true

if col >= N:

return True

# Consider this column and try placing this queen in all rows one by one

for i in range(N):

# Check if the queen can be placed on board[i][col]

if (ld[i - col + N - 1] != 1 and rd[i + col] != 1) and cl[i] != 1:

# Place this queen in board[i][col]

board[i][col] = 1

ld[i - col + N - 1] = rd[i + col] = cl[i] = 1

# Recur to place the rest of the queens

if solveNQUtil(board, col + 1):

return True

# If placing the queen in board[i][col] doesn't lead to a solution, backtrack

board[i][col] = 0 # BACKTRACK

ld[i - col + N - 1] = rd[i + col] = cl[i] = 0

# If the queen cannot be placed in any row in this column col, return false

return False

# This function solves the N Queen problem using Backtracking.

# It mainly uses solveNQUtil() to solve the problem.

# It returns false if queens cannot be placed, otherwise,

# returns true and prints placement of queens in the form of 1s.

# Please note that there may be more than one solution;

# this function prints one of the feasible solutions.

def solveNQ():

board = [[0 for \_ in range(N)] for \_ in range(N)]

if not solveNQUtil(board, 0):

print("Solution does not exist")

return False

printSolution(board)

return True

# Driver program to test above function

if \_\_name\_\_ == "\_\_main\_\_":

solveNQ()

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

. . Q .

Q . . .

. . . Q

. Q . .

Process finished with exit code 0

1. **Implement a python program to solve 8 queens problem using basic search strategies.**

**PROGRAM:**

N = 8 # (size of the chessboard)

def solveNQueens(board, col):

if col == N:

print(board)

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQueens(board, col + 1):

return True

board[i][col] = 0

return False

def isSafe(board, row, col):

for x in range(col):

if board[row][x] == 1:

return False

for x, y in zip(range(row, -1, -1), range(col, -1, -1)):

if board[x][y] == 1:

return False

for x, y in zip(range(row, N, 1), range(col, -1, -1)):

if board[x][y] == 1:

return False

return True

board = [[0 for x in range(N)] for y in range(N)]

if not solveNQueens(board, 0):

print("No solution found")

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

[[1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1], [0, 1, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0]]

Process finished with exit code 0

**9(set4): Write a program to implement forward chaining and back ward chaining using python:**

PROGRAM:

class KnowledgeBase:

def \_\_init\_\_(self):

self.facts = set()

self.rules = []

def add\_fact(self, fact):

self.facts.add(fact)

def add\_rule(self, rule):

self.rules.append(rule)

def forward\_chaining(self):

inferred\_facts = set(self.facts) # Start with known facts

agenda = [rule for rule in self.rules if self.\_can\_fire(rule, inferred\_facts)]

while agenda:

rule = agenda.pop(0)

conclusion = rule[0]

if conclusion not in inferred\_facts:

inferred\_facts.add(conclusion)

print(f"Inferred: {conclusion}")

for new\_rule in self.rules:

if self.\_can\_fire(new\_rule, inferred\_facts):

agenda.append(new\_rule)

def \_can\_fire(self, rule, inferred\_facts):

conclusion, premises = rule

return all(premise in inferred\_facts for premise in premises)

# Example usage of Forward Chaining:

kb\_fc = KnowledgeBase()

kb\_fc.add\_fact('A')

kb\_fc.add\_fact('B')

kb\_fc.add\_rule(('C', ['A', 'B']))

kb\_fc.add\_rule(('D', ['C']))

print("Applying forward chaining:")

kb\_fc.forward\_chaining()

class KnowledgeBase:

def \_\_init\_\_(self):

self.facts = set()

self.rules = {}

def add\_fact(self, fact):

self.facts.add(fact)

def add\_rule(self, conclusion, premises):

self.rules[conclusion] = premises

def is\_true(self, fact, visited=None):

if visited is None:

visited = set()

if fact in visited:

return False

visited.add(fact)

if fact in self.facts:

return True

elif fact in self.rules:

premises = self.rules[fact]

return all(self.is\_true(premise, visited) for premise in premises)

else:

return False

# Example usage of Backward Chaining:

kb\_bc = KnowledgeBase()

kb\_bc.add\_fact('A')

kb\_bc.add\_fact('B')

kb\_bc.add\_rule('C', ['A', 'B'])

kb\_bc.add\_rule('D', ['C'])

goal = 'D'

print(f"Checking if {goal} can be inferred using backward chaining:")

if kb\_bc.is\_true(goal):

print(f"Yes, {goal} can be inferred from the knowledge base.")

else:

print(f"No, {goal} cannot be inferred from the knowledge base.")

OUTPUT:

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\FB.py"

Applying forward chaining:

Inferred: C

Inferred: D

Checking if D can be inferred using backward chaining:

Yes, D can be inferred from the knowledge base.

Process finished with exit code 0

1. **Implement alpha beta pruning search.**

**PROGRAM:**

# Python3 program to demonstrate

# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta

MAX, MIN = 1000, -1000

# Returns optimal value for current player

#(Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer,

values, alpha, beta):

# Terminating condition. i.e

# leaf node is reached

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

# Recur for left and right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

# Alpha Beta Pruning

if beta <= alpha:

break

return best

else:

best = MAX

# Recur for left and

# right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

# Alpha Beta Pruning

if beta <= alpha:

break

return best

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

values = [3, 5, 6, 9, 1, 2, 0, -1]

print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

The optimal value is : 5

Process finished with exit code 0

**14. Program to implement breadth first search.**

**PROGRAM**

# Python3 Program to print BFS traversal

# from a given source vertex. BFS(int s)

# traverses vertices reachable from s.

from collections import defaultdict

# This class represents a directed graph

# using adjacency list representation

class Graph:

# Constructor

def \_\_init\_\_(self):

# Default dictionary to store graph

self.graph = defaultdict(list)

# Function to add an edge to graph

def addEdge(self, u, v):

self.graph[u].append(v)

# Function to print a BFS of graph

def BFS(self, s):

# Mark all the vertices as not visited

visited = [False] \* (max(self.graph) + 1)

# Create a queue for BFS

queue = []

# Mark the source node as

# visited and enqueue it

queue.append(s)

visited[s] = True

while queue:

# Dequeue a vertex from

# queue and print it

s = queue.pop(0)

print(s, end=" ")

# Get all adjacent vertices of the

# dequeued vertex s.

# If an adjacent has not been visited,

# then mark it visited and enqueue it

for i in self.graph[s]:

if not visited[i]:

queue.append(i)

visited[i] = True

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

# Create a graph given in

# the above diagram

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print("Following is Breadth First Traversal"

" (starting from vertex 2)")

g.BFS(2)

**OUTPUT:**

"C:\Users\GRT AI&DS\ailab\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\ailab\main.py"

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

Process finished with exit code 0

**18. Implement backward chaining and resolution strategies.**

**PROGRAM:**

class KnowledgeBase:

def \_\_init\_\_(self):

self.facts = {}

self.rules = {}

def add\_fact(self, fact, value=True):

self.facts[fact] = value

def add\_rule(self, conclusion, premises):

self.rules[conclusion] = premises

def is\_true(self, fact):

if fact in self.facts:

return self.facts[fact]

elif fact in self.rules:

premises = self.rules[fact]

for premise in premises:

if not self.is\_true(premise):

return False

return True

else:

return False

# Example usage:

kb = KnowledgeBase()

kb.add\_fact('A')

kb.add\_fact('B')

kb.add\_rule('C', ['A', 'B'])

goal = 'C'

if kb.is\_true(goal):

print(f'{goal} is true based on the knowledge base.')

else:

print(f'{goal} cannot be inferred from the knowledge base.')

class Resolution:

def \_\_init\_\_(self, clauses):

self.clauses = clauses

def resolve(self, clause1, clause2):

new\_clause = []

for literal in clause1:

if literal not in clause2 and f'-{literal}' not in clause2:

new\_clause.append(literal)

for literal in clause2:

if literal not in clause1 and f'-{literal}' not in clause1:

new\_clause.append(literal)

return new\_clause

def apply\_resolution(self):

new\_clauses = self.clauses[:]

while True:

n = len(new\_clauses)

pairs = [(new\_clauses[i], new\_clauses[j]) for i in range(n) for j in range(i + 1, n)]

for (clause1, clause2) in pairs:

resolvent = self.resolve(clause1, clause2)

if [] in resolvent:

return True # Empty clause found, contradiction

if resolvent not in new\_clauses:

new\_clauses.append(resolvent)

if len(new\_clauses) == n:

return False # No new clauses added, no contradiction found

clauses = [

['A', 'B'],

['-A', 'C'],

['-B', '-C'],

['-A', '-B']

]

resolver = Resolution(clauses)

if resolver.apply\_resolution():

print("Contradiction found: clauses are unsatisfiable.")

else:

print("No contradiction found: clauses are satisfiable.")

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\1.py"

C is true based on the knowledge base.

Process finished with exit code 0

**19. BASE+BALL=GAMES is a crypt arithmetic puzzle. Implement a python program to make this statement true.**

**PROGRAM:**

class Main:

#Set 1 when one character is assigned previously

use = [0] \* 10

class Node:

def \_\_init\_\_(self):

self.letter = ''

self.value = 0

def isValid(self, nodeList, count, s1, s2, s3):

val1 = 0

val2 = 0

val3 = 0

m = 1

j = 0

i = 0

#find number for first string

for i in range(len(s1) - 1, -1, -1):

ch = s1[i]

for j in range(count):

#when ch is present, break the loop

if nodeList[j].letter == ch:

break

val1 += m \* nodeList[j].value

m \*= 10

m = 1

#find number for the second string

for i in range(len(s2) - 1, -1, -1):

ch = s2[i]

for j in range(count):

if nodeList[j].letter == ch:

break

val2 += m \* nodeList[j].value

m \*= 10

m = 1

#find number for the third string

for i in range(len(s3) - 1, -1, -1):

ch = s3[i]

for j in range(count):

if nodeList[j].letter == ch:

break

val3 += m \* nodeList[j].value

m \*= 10

#check whether the sum is the same as the 3rd string or not

if val3 == (val1 + val2):

return 1

return 0

def permutation(self, count, nodeList, n, s1, s2, s3):

#when values are assigned

if n == count - 1:

for i in range(10):

#for those numbers, which are not used

if self.use[i] == 0:

#assign value i

nodeList[n].value = i

if self.isValid(nodeList, count, s1, s2, s3) == 1:

print("Solution found:", end='')

#print code, which is assigned

for j in range(count):

print(f" {nodeList[j].letter} = {nodeList[j].value}", end='')

return 1

return 0

for i in range(10):

#for those numbers, which are not used

if self.use[i] == 0:

#assign value i and mark as not available for future use

nodeList[n].value = i

self.use[i] = 1

if self.permutation(count, nodeList, n + 1, s1, s2, s3) == 1:

#go for the next characters

return 1

#when backtracking, make available again

self.use[i] = 0

return 0

def solvePuzzle(self, s1, s2, s3):

#Number of unique characters

uniqueChar = 0

len1 = len(s1)

len2 = len(s2)

len3 = len(s3)

#There are 26 different characters

freq = [0] \* 26

for i in range(len1):

freq[ord(s1[i]) - ord('A')] += 1

for i in range(len2):

freq[ord(s2[i]) - ord('A')] += 1

for i in range(len3):

freq[ord(s3[i]) - ord('A')] += 1

for i in range(26):

#whose frequency is > 0, they are present

if freq[i] > 0:

uniqueChar += 1

#as there are 10 digits in the decimal system

if uniqueChar > 10:

print("Invalid strings")

return 0

nodeList = [self.Node() for \_ in range(uniqueChar)]

j = 0

for i in range(26):

#assign all characters found in three strings

if freq[i] > 0:

nodeList[j].letter = chr(i + ord('A'))

j += 1

return self.permutation(uniqueChar, nodeList, 0, s1, s2, s3)

if \_\_name\_\_ == "\_\_main\_\_":

main = Main()

s1 = "BASE"

s2 = "BALL"

s3 = "GAMES"

if main.solvePuzzle(s1, s2, s3) == 0:

print("No solution")

**OUTPUT:**

"C:\Users\GRT AI&DS\AI\_LAB\venv\Scripts\python.exe" "C:\Users\GRT AI&DS\AI\_LAB\main.py"

Solution found: A = 4 B = 2 E = 1 G = 0 L = 5 M = 9 S = 6

Process finished with exit code 0