Artificial Intelligence Assignments Codes

Sunday 28th May, 2023

Listings

```python
from collections import deque

visited = deque([])
queue = deque([])

def bredth_first_search(visited, queue, graph, node):
    visited.append(node)
    queue.append(node)

    while queue:         # Creating loop to visit each node
        m = queue.popleft()
        print(m, end = " ")

        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

graph = {
    '5': ['3', '7'],
    '3': ['2', '4'],
    '2': [],
    '4': ['8'],
    '7': ['6'],
    '8': [],
    '6': []
}

visited = deque([])
queue = deque([])

print("Following is the Breadth First Search")
bredth_first_search(visited, queue, graph, '5')
print(end="\n")
```

Listing 1: Breadth first search

```python
graph = {
    '5': ['3', '7'],
    '3': ['2', '4'],
    '2': [],
    '4': ['8'],
    '7': ['6'],
    '8': [],
    '6': []
}

visited = set()     # Set to keep track of visited nodes of graph.

def depth_first_search(visited, graph, node):
    if node not in visited:
        print(node, end =" ")
        visited.add(node)

        for neighbour in graph[node]:
            depth_first_search(visited, graph, neighbour)

# Driver code for the program
print("Following is the Depth First Search")
depth_first_search(visited, graph, '5')
```

Listing 2: Depth first search

```python
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()

    g = {}               # store distance from starting node
    parents = {}         # parents contains an adjacency map of all nodes

    # distance of starting node from itself is zero
    g[start_node] = 0

    # start_node is root node i.e it has no parent nodes
    # so start_node is set to its own parent node
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None
        # node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or Graph_nodes[n] == None:
            pass

        else:
            for (m, weight) in get_neighbors(n):
                # nodes 'm' not in first and last set are added to first
                # n is set its parent

                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # for each node m,compare its distance from start i.e g(m) to the
                # from start through n node

                else:
                    if g[m] > g[n] + weight:
                        # update g(m)
```

```
                        g[m] = g[n] + weight

                        # change parent of m to n
                        parents[m] = n

                        # if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node

        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]

            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path

        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected

        open_set.remove(n)
        closed_set.add(n)

    print('Path does not exist!')
    return None

# define fuction to return neighbor and its distance
# from the passed node

def get_neighbors(v):
```

```
    if v in Graph_nodes:
        return Graph_nodes[v]

    else:
        return None


#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}

aStarAlgo('A', 'J')
```

Listing 3: A* Algorithm in Python

```python
import sys

class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxsize

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    def primMST(self):

        key = [sys.maxsize] * self.V
        parent = [None] * self.V
        key[0] = 0
        mstSet = [False] * self.V

        parent[0] = -1

        for cout in range(self.V):
            u = self.minKey(key, mstSet)

            mstSet[u] = True
            for v in range(self.V):
                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
```

```
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)

g = Graph(5)

g.graph = [
    [0, 2, 0, 6, 0],
    [2, 0, 3, 8, 5],
    [0, 3, 0, 0, 7],
    [6, 8, 0, 0, 9],
    [0, 5, 7, 9, 0]
]

g.primMST();
```

Listing 4: Prims Algorithm in Python

```python
# Python3 program to solve N Queen
# Problem using backtracking global N

N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end = " ")
        print()


# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens

def isSafe(board, row, col):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solveNQUtil(board, col):

```

```
    # base case: If all queens are placed
    # then 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):
        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True

            # If placing queen in board[i][col]
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this column col then 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 return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.

def solveNQ():
    board = [
        [0, 0, 0, 0],
        [0, 0, 0, 0],
```

```
        [0, 0, 0, 0],
        [0, 0, 0, 0]
    ]

    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False

    printSolution(board)
    return True

# Driver Code
solveNQ()
```

Listing 5: N Queen Problem using backtracking global N in Python

```python
'''
Python3 program to solve N Queen Problem
using Branch or Bound
'''

N = 8

# A utility function to print solution
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end = " ")
        print()

'''
A Optimized function to check if
a queen can be placed on board[row][col]
'''
def isSafe(row, col, slashCode, backslashCode,
           rowLookup, slashCodeLookup,
           backslashCodeLookup):
    if (slashCodeLookup[slashCode[row][col]] or
        backslashCodeLookup[backslashCode[row][col]] or
        rowLookup[row]):
        return False
    return True

'''
A recursive utility function
to solve N Queen problem
'''

def solveNQueensUtil(board, col, slashCode, backslashCode,
                     rowLookup, slashCodeLookup,
                     backslashCodeLookup):
    '''
    base case: If all queens are
    placed then return True
    '''
    if(col >= N):
```

```
        return True
    for i in range(N):
        if(isSafe(i, col, slashCode, backslashCode,
                  rowLookup, slashCodeLookup,
                  backslashCodeLookup)):
            '''
            Place this queen in board[i][col]
            '''
            board[i][col] = 1
            rowLookup[i] = True
            slashCodeLookup[slashCode[i][col]] = True
            backslashCodeLookup[backslashCode[i][col]] = True

            '''
            recur to place rest of the queens
            '''
            if(solveNQueensUtil(board, col + 1,
                                slashCode, backslashCode,
                                rowLookup, slashCodeLookup,
                                backslashCodeLookup)):
                return True

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

            # Remove queen from board[i][col]
            board[i][col] = 0
            rowLookup[i] = False
            slashCodeLookup[slashCode[i][col]] = False
            backslashCodeLookup[backslashCode[i][col]] = False

    '''
    If queen can not be place in any row in
    this column col then return False
    '''
    return False

'''
```

```
This function solves the N Queen problem using
Branch or Bound. It mainly uses solveNQueensUtil()to
solve the problem. It returns False if queens
cannot be placed,otherwise return True or
prints placement of queens in the form of 1s.
Please note that there may be more than one
solutions,this function prints one of the
feasible solutions.
'''

def solveNQueens():
    board = [
        [0 for i in range(N)]
        for j in range(N)
    ]

    # helper matrices
    slashCode = [
        [0 for i in range(N)]
        for j in range(N)
    ]

    backslashCode = [
        [0 for i in range(N)]
        for j in range(N)
    ]

    # arrays to tell us which rows are occupied
    rowLookup = [False] * N

    # keep two arrays to tell us
    # which diagonals are occupied
    x = 2 * N - 1

    slashCodeLookup = [False] * x
    backslashCodeLookup = [False] * x

    # initialize helper matrices
    for rr in range(N):
        for cc in range(N):
```

```python
            slashCode[rr][cc] = rr + cc
            backslashCode[rr][cc] = rr - cc + 7

    if(solveNQueensUtil(board, 0, slashCode, backslashCode,
                        rowLookup, slashCodeLookup,
                        backslashCodeLookup) == False):
        print("Solution does not exist")
        return False

    # solution found
    printSolution(board)
    return True

# Driver Code
solveNQueens()
```

Listing 6: N Queen Problem using Branch And Bound in Python

```python
# Python3 program for the above approach

# Number of vertices in the graph
# define 4 X 4

# check if the colored
# graph is safe or not

def isSafe(graph, color):
    # check for every edge
    for i in range(4):
        for j in range(i + 1, 4):
            if (graph[i][j] and color[j] == color[i]):
                return False
    return True

# This function solves the m Coloring
# problem using recursion. It returns
# false if the m colours cannot be assigned,
# otherwise, return true and prints
# assignments of colours to all vertices.
# Please note that there may be more than
# one solutions, this function prints one
# of the feasible solutions.

def graphColoring(graph, m, i, color):

    # if current index reached end
    if (i == 4):
        # if coloring is safe
        if (isSafe(graph, color)):

            # Print the solution
            printSolution(color)
            return True
        return False

    # Assign each color from 1 to m
    for j in range(1, m + 1):
        color[i] = j
```

```python

        # Recur of the rest vertices
        if (graphColoring(graph, m, i + 1, color)):
            return True
        color[i] = 0
    return False

# A utility function to print solution

def printSolution(color):
    print("Solution Exists:" " Following are the assigned colors ")
    for i in range(4):
        print(color[i], end=" ")

    print("\n")

# Driver code

if __name__ == '__main__':

    graph = [
        [0, 1, 1, 1],
        [1, 0, 1, 0],
        [1, 1, 0, 1],
        [1, 0, 1, 0],
    ]

    m = 3       # Number of colors

    # Initialize all color values as 0.
    # This initialization is needed
    # correct functioning of isSafe()
    color = [0 for i in range(4)]

    # Function call
    if (not graphColoring(graph, m, 0, color)):
        print("Solution does not exist")
```

Listing 7: Implementation of Graph Colouring Using BackTracking in Python

```python
def greet(bot_name, birth_year):
    print("Hello! My name is {0}.".format(bot_name))
    print("I was created in {0}.".format(birth_year))

def remind_name():
    print('\nPlease, remind me your name.')
    name = input()
    print("What a great name you have, {0}!".format(name))

def guess_age():
    print('\nLet me guess your age.')
    print('Enter remainders of dividing your age by 3, 5 and 7.')

    rem3 = int(input())
    rem5 = int(input())
    rem7 = int(input())
    age = (rem3 * 70 + rem5 * 21 + rem7 * 15) % 105

    print("Your age is {0}; that's a good time to start programming!".format(age))

def number_guess():
    import random
    import math

    print("\nHey! Here's a number guessing game for you!")

    lower = int(input("\nEnter Lower bound:- "))
    upper = int(input("Enter Upper bound:- "))
    x = random.randint(lower, upper)

    print("\n\tYou've only ",
          round(math.log(upper - lower + 1, 2)),
          " chances to guess the integer!\n")

    count = 0
    while count < math.log(upper - lower + 1, 2):
        count += 1

        guess = int(input("Guess a number:- "))

```

```
        if x == guess:
            print("Congratulations you did it in ",
                  count, " try")
            break

        elif x > guess:
            print("You guessed too small!")

        elif x < guess:
            print("You Guessed too high!")

    if count >= math.log(upper - lower + 1, 2):
        print("\nThe number is %d" % x)
        print("\tBetter Luck Next time!")

def count():
    print('\nNow I will prove to you that I can count to any number you want.')
    num = int(input())

    counter = 0

    while counter <= num:
        print("{0} !".format(counter))
        counter += 1

def test():
    print("\nLet's test your programming knowledge.")
    print("Why do we use methods?")
    print("1. To repeat a statement multiple times.")
    print("2. To decompose a program into several small subroutines.")
    print("3. To determine the execution time of a program.")
    print("4. To interrupt the execution of a program.")

    answer = 2
    guess = int(input())

    while guess != answer:
        print("Please, try again.")
        guess = int(input())

```

```python
    print('Completed, have a nice day!')
    print('...............................')
    print('...............................')
    print('...............................')


def end():
    print('Congratulations, have a nice day!')
    print('...............................')
    print('...............................')
    print('...............................')

    input()

greet('ChatBot-v4', '2023')
remind_name()
guess_age()
number_guess()
count()
test()
end()
```

Listing 8: Develop an elementary catboat for any suitable customer interaction application in Python