Practical no. 01

Implement depth first search algorithm and Breadth First Search algorithm. Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

**Code:**

from collections import defaultdict, deque

class Graph:

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

# Default dictionary to store graph

self.graph = defaultdict(list)

def add\_edge(self, u, v):

self.graph[u].append(v)

self.graph[v].append(u) # Since it's undirected, add the reverse edge as well.

def dfs\_recursive(self, vertex, visited=None):

if visited is None:

visited = set()

visited.add(vertex)

print(vertex, end=' ')

for neighbor in self.graph[vertex]:

if neighbor not in visited:

self.dfs\_recursive(neighbor, visited)

def bfs(self, start):

visited = set() # To keep track of visited nodes

queue = deque([start]) # Use deque for an efficient queue implementation

visited.add(start)

while queue:

vertex = queue.popleft() # Pop the front of the queue

print(vertex, end=' ')

# Add all unvisited neighbors to the queue

for neighbor in self.graph[vertex]:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

# Example usage:

g = Graph()

g.add\_edge(0, 1)

g.add\_edge(0, 2)

g.add\_edge(1, 2)

g.add\_edge(2, 3)

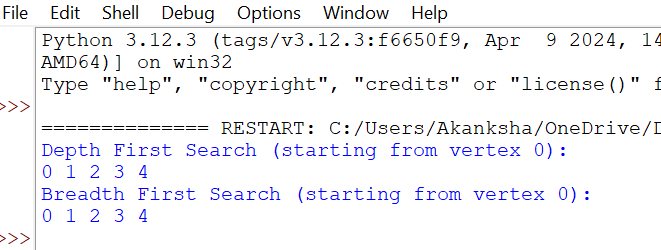
g.add\_edge(3, 4)

print("Depth First Search (starting from vertex 0):")

g.dfs\_recursive(0)

print("\nBreadth First Search (starting from vertex 0):")

g.bfs(0)

**Output:**

Practical no. 02

Implement A star (A\*) Algorithm for any game search problem

**Code:**

import heapq

class AStar:

def \_\_init\_\_(self, grid, start, goal):

self.grid = grid # 2D grid where 0 = walkable, 1 = blocked

self.start = start # Start position (x, y)

self.goal = goal # Goal position (x, y)

self.rows = len(grid)

self.cols = len(grid[0])

def heuristic(self, node):

# Manhattan distance heuristic

return abs(node[0] - self.goal[0]) + abs(node[1] - self.goal[1])

def neighbors(self, node):

# Return valid neighbors (up, down, left, right)

dirs = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Directions: right, down, left, up

result = []

for d in dirs:

neighbor = (node[0] + d[0], node[1] + d[1])

if 0 <= neighbor[0] < self.rows and 0 <= neighbor[1] < self.cols and self.grid[neighbor[0]][neighbor[1]] == 0:

result.append(neighbor)

return result

def a\_star\_search(self):

# Priority queue to store (f\_score, node)

open\_list = []

heapq.heappush(open\_list, (0, self.start))

came\_from = {} # For reconstructing path

g\_score = {self.start: 0} # Cost from start to each node

f\_score = {self.start: self.heuristic(self.start)} # Estimated cost from start to goal

while open\_list:

current = heapq.heappop(open\_list)[1]

# If we reached the goal, reconstruct the path

if current == self.goal:

return self.reconstruct\_path(came\_from, current)

for neighbor in self.neighbors(current):

tentative\_g\_score = g\_score[current] + 1 # Distance from current to neighbor is 1

if neighbor not in g\_score or tentative\_g\_score < g\_score[neighbor]:

# Update the best path to the neighbor

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = tentative\_g\_score + self.heuristic(neighbor)

heapq.heappush(open\_list, (f\_score[neighbor], neighbor))

return [] # Return empty path if no solution

def reconstruct\_path(self, came\_from, current):

# Reconstruct path from came\_from dictionary

total\_path = [current]

while current in came\_from:

current = came\_from[current]

total\_path.append(current)

return total\_path[::-1] # Reverse path to start from the beginning

# 0 = walkable, 1 = blocked

grid = [

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

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

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

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

[0, 0, 0, 1, 0]

]

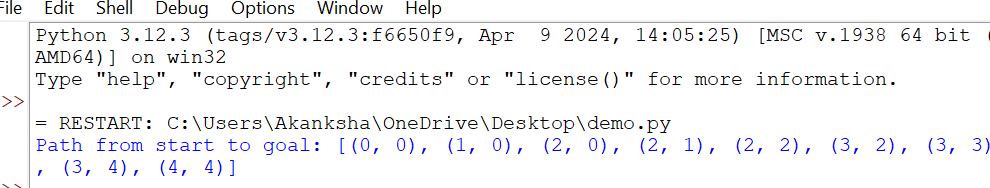
start = (0, 0) # Starting position

goal = (4, 4) # Goal position

a\_star = AStar(grid, start, goal)

path = a\_star.a\_star\_search()

print("Path from start to goal:", path)

**Output:**

**Practical no.03**

Implement Alpha-Beta Tree search for any game search problem.

**Code:**

import math

MAX\_PLAYER = 'X'

MIN\_PLAYER = 'O'

EMPTY = '\_'

class TicTacToe:

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

# Initialize an empty 3x3 board

self.board = [

[EMPTY, EMPTY, EMPTY],

[EMPTY, EMPTY, EMPTY],

[EMPTY, EMPTY, EMPTY]

]

def is\_moves\_left(self, board):

# Check if there are empty cells left on the board

for row in board:

if EMPTY in row:

return True

return False

def evaluate(self, board):

# Check rows, columns, and diagonals for a winner

for row in range(3):

if board[row][0] == board[row][1] == board[row][2]:

if board[row][0] == MAX\_PLAYER:

return 10

elif board[row][0] == MIN\_PLAYER:

return -10

for col in range(3):

if board[0][col] == board[1][col] == board[2][col]:

if board[0][col] == MAX\_PLAYER:

return 10

elif board[0][col] == MIN\_PLAYER:

return -10

if board[0][0] == board[1][1] == board[2][2]:

if board[0][0] == MAX\_PLAYER:

return 10

elif board[0][0] == MIN\_PLAYER:

return -10

if board[0][2] == board[1][1] == board[2][0]:

if board[0][2] == MAX\_PLAYER:

return 10

elif board[0][2] == MIN\_PLAYER:

return -10

return 0

def minimax(self, board, depth, is\_maximizing, alpha, beta):

score = self.evaluate(board)

if score == 10:

return score - depth

if score == -10:

return score + depth

# If no more moves and no winner (tie)

if not self.is\_moves\_left(board):

return 0

if is\_maximizing:

best = -math.inf

for i in range(3):

for j in range(3):

if board[i][j] == EMPTY:

# Make the move

board[i][j] = MAX\_PLAYER

# Recur and choose the maximum value

best = max(best, self.minimax(board, depth + 1, False, alpha, beta))

board[i][j] = EMPTY

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = math.inf

for i in range(3):

for j in range(3):

if board[i][j] == EMPTY:

# Make the move

board[i][j] = MIN\_PLAYER

best = min(best, self.minimax(board, depth + 1, True, alpha, beta))

board[i][j] = EMPTY

# Update beta

beta = min(beta, best)

# Alpha-Beta Pruning

if beta <= alpha:

break

return best

def find\_best\_move(self):

best\_val = -math.inf

best\_move = (-1, -1)

# Traverse all cells, evaluate minimax function for each empty cell, and return the best move

for i in range(3):

for j in range(3):

if self.board[i][j] == EMPTY:

# Make the move

self.board[i][j] = MAX\_PLAYER

# Compute evaluation function for this move

move\_val = self.minimax(self.board, 0, False, -math.inf, math.inf)

# Undo the move

self.board[i][j] = EMPTY

# If the value of the current move is better than the best value, update best move

if move\_val > best\_val:

best\_move = (i, j)

best\_val = move\_val

return best\_move

def print\_board(self):

for row in self.board:

print(" | ".join(row))

print("-" \* 9)

game = TicTacToe()

game.board = [

['X', 'O', 'X'],

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

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

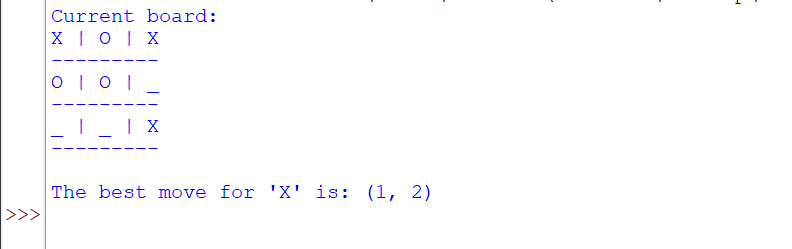
]

print("Current board:")

game.print\_board()

best\_move = game.find\_best\_move()

print(f"\nThe best move for 'X' is: {best\_move}")

**Output:**

**Practical no.04**

Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.

**Code:**

def print\_board(board):

for row in board:

print(" ".join("Q" if col else "." for col in row))

print("\n")

def is\_safe(board, row, col, n):

# Check the current column

for i in range(row):

if board[i][col]:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j]:

return False

for i, j in zip(range(row, -1, -1), range(col, n)):

if board[i][j]:

return False

return True

def solve\_n\_queens\_backtracking(board, row, n):

if row == n:

# If all queens are placed, print the solution

print\_board(board)

return True

found\_solution = False

for col in range(n):

if is\_safe(board, row, col, n):

board[row][col] = True # Place the queen

# Recur to place the rest of the queens

found\_solution = solve\_n\_queens\_backtracking(board, row + 1, n) or found\_solution

# If placing the queen at (row, col) didn't lead to a solution, backtrack

board[row][col] = False

return found\_solution

def n\_queens\_backtracking(n):

board = [[False] \* n for \_ in range(n)] # Initialize n x n board with False (no queens placed)

if not solve\_n\_queens\_backtracking(board, 0, n):

print("No solution exists.")

def is\_safe\_branch\_and\_bound(row, col, cols, diags1, diags2, n):

return not (cols[col] or diags1[row + col] or diags2[row - col + (n - 1)])

def solve\_n\_queens\_branch\_and\_bound(row, n, cols, diags1, diags2, board):

if row == n:

# If all queens are placed, print the solution

print\_board(board)

return True

found\_solution = False

for col in range(n):

if is\_safe\_branch\_and\_bound(row, col, cols, diags1, diags2, n):

# Place the queen

board[row][col] = True

cols[col] = True

diags1[row + col] = True

diags2[row - col + (n - 1)] = True

# Recur to place the rest of the queens

found\_solution = solve\_n\_queens\_branch\_and\_bound(row + 1, n, cols, diags1, diags2, board) or found\_solution

board[row][col] = False

cols[col] = False

diags1[row + col] = False

diags2[row - col + (n - 1)] = False

return found\_solution

def n\_queens\_branch\_and\_bound(n):

board = [[False] \* n for \_ in range(n)] # Initialize n x n board with False (no queens placed)

cols = [False] \* n

diags1 = [False] \* (2 \* n - 1) # For "/" diagonals

diags2 = [False] \* (2 \* n - 1) # For "\" diagonals

if not solve\_n\_queens\_branch\_and\_bound(0, n, cols, diags1, diags2, board):

print("No solution exists.")

**Output:**

Solving n-Queens using Backtracking:

. Q . .

. . . Q

Q . . .

. . Q .

. . Q .

Q . . .

. . . Q

. Q . .

**Practical no. 05**

Implement Greedy search algorithm for any of the following application: • Selection Sort • Minimum Spanning Tree • Single-Source Shortest Path Problem • Job Scheduling Problem • Prim's Minimal Spanning Tree Algorithm • Kruskal's Minimal Spanning Tree Algorithm • Dijkstra's Minimal Spanning Tree Algorithm.

**Code:**

def selection\_sort(arr):

n = len(arr)

# Traverse through all array elements

for i in range(n):

# Find the minimum element in the remaining unsorted array

min\_index = i

for j in range(i+1, n):

if arr[j] < arr[min\_index]:

min\_index = j

# Swap the found minimum element with the first element

arr[i], arr[min\_index] = arr[min\_index], arr[i]

print(f"Step {i+1}: {arr}") # Print the array after each swap

return arr

# Example usage:

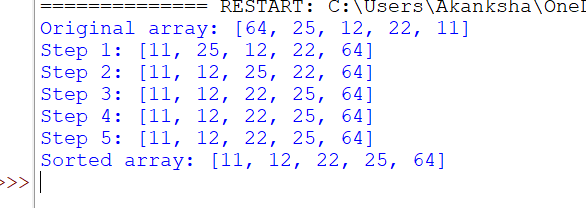
arr = [64, 25, 12, 22, 11]

print("Original array:", arr)

sorted\_arr = selection\_sort(arr)

print("Sorted array:", sorted\_arr)

**Output:**

****