Nhập Môn Trí Tuệ Nhân Tạo – Lớp 0201

Đoàn Nhật Khôi - 2374802010242

LAB 2

Câu 3:

# Import defaultdict để tạo dictionary với giá trị mặc định là list

from collections import defaultdict

# Import thư viện NetworkX để xây dựng và vẽ đồ thị

import networkx as nx

# Import matplotlib để hiển thị hình ảnh đồ thị

import matplotlib.pyplot as plt

# Khai báo lớp Graph để tạo và thao tác với đồ thị

class Graph:

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

# Sử dụng defaultdict để lưu danh sách kề

self.graph = defaultdict(list)

def addEdge(self, u, v):

# Thêm cạnh từ đỉnh u đến v (vì là đồ thị vô hướng nên cần thêm cả hai chiều)

self.graph[u].append(v)

# Hàm đệ quy hỗ trợ cho thuật toán DFS

def DFSUtil(self, v, visited):

visited[v] = True # Đánh dấu đỉnh v là đã thăm

print(v, end=' ') # In đỉnh v ra màn hình

# Duyệt qua tất cả các đỉnh kề của v

for i in self.graph[v]:

if visited[i] == False:

self.DFSUtil(i, visited) # Gọi đệ quy tiếp tục DFS với đỉnh chưa thăm

def DFS(self, v):

# Khởi tạo mảng visited với kích thước đủ lớn, đánh dấu tất cả các đỉnh là chưa thăm

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

# Bắt đầu DFS từ đỉnh v

self.DFSUtil(v, visited)

# Chạy chương trình chính

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

g = Graph() # Tạo một đối tượng đồ thị

# Thêm các cạnh vào đồ thị (thêm hai chiều vì là đồ thị vô hướng)

g.addEdge(0, 1)

g.addEdge(1, 0)

g.addEdge(0, 2)

g.addEdge(2, 0)

g.addEdge(2, 4)

g.addEdge(4, 2)

g.addEdge(1, 4)

g.addEdge(4, 1)

g.addEdge(1, 3)

g.addEdge(3, 1)

g.addEdge(3, 4)

g.addEdge(4, 3)

g.addEdge(3, 5)

g.addEdge(5, 3)

g.addEdge(5, 4)

g.addEdge(4, 5)

# In ra màn hình thứ tự duyệt DFS bắt đầu từ đỉnh 0

print("DFS - duyệt tìm kiếm theo chiều sâu bắt đầu từ đỉnh 0")

g.DFS(0)

# --- VẼ ĐỒ THỊ ---

# Tạo đồ thị vô hướng bằng NetworkX

G = nx.Graph()

# Thêm các cạnh vào NetworkX từ danh sách kề của đồ thị g

for u in g.graph:

for v in g.graph[u]:

G.add\_edge(u, v)

# Xác định vị trí hiển thị các đỉnh bằng spring layout (bố cục lò xo)

pos = nx.spring\_layout(G)

# Vẽ đồ thị:

# - with\_labels=True: hiển thị tên các đỉnh

# - node\_color: màu đỉnh

# - edge\_color: màu cạnh

# - node\_size: kích thước đỉnh

# - font\_size: cỡ chữ

nx.draw(G, pos, with\_labels=True, node\_color='lightblue',

edge\_color='gray', node\_size=700, font\_size=16)

# Hiển thị tiêu đề và đồ thị

plt.title("Đồ thị minh họa DFS từ đỉnh 0")

plt.show()

Đồ thị:

A screenshot of a computer

Description automatically generated

Mô tả DFS:

B1: Bắt đầu từ một đỉnh gốc. Đánh dấu đỉnh này là đã duyệt.

B2: Thăm một đỉnh kề chưa được duyệt của đỉnh hiện tại. Đánh dấu đỉnh mới này là đã duyệt và đặt nó làm đỉnh hiện tại.

B3: Lặp lại B2 cho đến khi không còn đỉnh kề nào chưa được duyệt từ đỉnh hiện tại.

B4: Quay lui về đỉnh trước đó.

B5: Nếu còn đỉnh kề khác chưa được duyệt từ đỉnh này, lặp lại từ B2.

B6: Nếu không còn đỉnh kề nào chưa được duyệt và không thể quay lui nữa.

Mô tả duyệt tìm kiếm:

Từ 0 sang 1

Từ 1 sang 4

Từ 4 sang 2

Từ 2 quay lui (vì không còn đỉnh mới)

Tiếp tục từ 4 sang 3

Từ 3 sang 5

Câu 4:

# Import defaultdict để tạo dictionary với giá trị mặc định là list

from collections import defaultdict

# Import NetworkX để vẽ đồ thị

import networkx as nx

# Import matplotlib để hiển thị đồ thị

import matplotlib.pyplot as plt

# Khai báo class đồ thị

class Graph:

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

# Khởi tạo một dictionary mặc định dạng danh sách kề

self.graph = defaultdict(list)

def addEdge(self, u, v):

# Thêm cạnh từ đỉnh u đến v (đồ thị vô hướng nên cần thêm cả hai chiều)

self.graph[u].append(v)

self.graph[v].append(u) # Thêm cạnh ngược lại để đảm bảo đồ thị vô hướng

# Hàm đệ quy để thực hiện DFS từ đỉnh v

def DFSUtil(self, v, visited):

visited[v] = True # Đánh dấu đỉnh v đã được thăm

print(v, end=' ') # In ra đỉnh hiện tại

# Duyệt qua các đỉnh kề chưa được thăm

for i in self.graph[v]:

if not visited[i]: # Nếu đỉnh chưa được thăm

self.DFSUtil(i, visited) # Gọi đệ quy DFS cho đỉnh chưa thăm

def DFS(self, v):

# Tạo danh sách visited với số phần tử bằng số đỉnh lớn nhất + 1

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

# Bắt đầu DFS từ đỉnh v

self.DFSUtil(v, visited)

# Chạy chương trình

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

g = Graph() # Tạo đối tượng đồ thị

# Thêm các cạnh (đồ thị vô hướng nên thêm cả hai chiều cho mỗi cạnh)

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(0, 3)

g.addEdge(1, 4)

g.addEdge(1, 5)

g.addEdge(4, 6)

g.addEdge(5, 7)

# In kết quả DFS từ đỉnh 0

print("DFS - duyệt tìm kiếm theo chiều sâu bắt đầu từ đỉnh 0:")

g.DFS(0)

# Vẽ đồ thị bằng thư viện networkx

G = nx.Graph() # Đồ thị vô hướng

for u in g.graph:

for v in g.graph[u]:

G.add\_edge(u, v) # Thêm cạnh vào đồ thị

pos = nx.spring\_layout(G) # Bố trí vị trí các đỉnh

# Vẽ đồ thị:

# - with\_labels=True: hiển thị tên các đỉnh

# - node\_color: màu đỉnh

# - edge\_color: màu cạnh

# - node\_size: kích thước đỉnh

# - font\_size: cỡ chữ

nx.draw(G, pos, with\_labels=True, node\_color='lightblue',

edge\_color='gray', node\_size=700, font\_size=16)

# Hiển thị tiêu đề và đồ thị

plt.title("Đồ thị minh họa DFS từ đỉnh 0")

plt.show()

Mô tả DFS:

B1: Bắt đầu từ một đỉnh gốc. Đánh dấu đỉnh này là đã duyệt.

B2: Thăm một đỉnh kề chưa được duyệt của đỉnh hiện tại. Đánh dấu đỉnh mới này là đã duyệt và đặt nó làm đỉnh hiện tại.

B3: Lặp lại bước 2 cho đến khi không còn đỉnh kề nào chưa được duyệt.

B4: Quay lui về đỉnh trước đó.

B5: Nếu còn đỉnh kề chưa được duyệt, tiếp tục duyệt.

B6: Nếu không còn đỉnh nào chưa duyệt và không thể quay lui, thuật toán kết thúc.

Mô tả duyệt tìm kiếm:

Bắt đầu từ đỉnh 0

Duyệt sang đỉnh 1

Từ 1 sang 4

Từ 4 sang 6

6 không còn đỉnh kề chưa thăm → quay lui về 4

Từ 4 tiếp tục sang 5

Từ 5 sang 7

7 không còn đỉnh chưa thăm → quay lui

Trở lại 1, rồi 0, sang 2

Cuối cùng sang 3

Câu 5:

from collections import deque # Nhập khẩu deque, một cấu trúc dữ liệu hàng đợi hai đầu, nhưng không được sử dụng trực tiếp trong triển khai A\* này.

import networkx as nx # Nhập thư viện networkx để vẽ đồ thị

import matplotlib.pyplot as plt # Nhập thư viện matplotlib để hiển thị đồ thị

class Graph:

def \_\_init\_\_(self, adjac\_lis):

# Khởi tạo đối tượng Graph.

self.adjac\_lis = adjac\_lis

def get\_neighbors(self, v):

# Trả về danh sách các hàng xóm của một đỉnh v cùng với trọng số cạnh.

return self.adjac\_lis.get(v, [])

def h(self, n, h\_values):

# Hàm heuristic (ước lượng) cho đỉnh n.

return h\_values.get(n, float('inf'))

def a\_star\_algorithm(self, start, stop, h\_values):

# Triển khai thuật toán tìm kiếm A\*.

open\_lst = set([start]) # Tập hợp các đỉnh cần được đánh giá.

closed\_lst = set([]) # Tập hợp các đỉnh đã được đánh giá.

g\_score = {start: 0} # Chi phí từ đỉnh bắt đầu đến chính nó là 0.

came\_from = {start: start} # Đỉnh bắt đầu không có đỉnh nào đi trước nó.

while len(open\_lst) > 0:

n = None

# Tìm đỉnh n trong open\_lst có f\_score (g\_score + h\_score) thấp nhất.

for v in open\_lst:

if n is None or (g\_score.get(v, float('inf')) + self.h(v, h\_values)) < \

(g\_score.get(n, float('inf')) + self.h(n, h\_values)):

n = v

if n is None:

print('Đường đi không tồn tại!')

return None

if n == stop:

# Nếu đỉnh hiện tại n là đỉnh đích, tái tạo và trả về đường đi.

reconst\_path = []

while came\_from[n] != n: # Đi ngược từ đích về đến đỉnh bắt đầu.

reconst\_path.append(n)

n = came\_from[n]

reconst\_path.append(start) # Thêm đỉnh bắt đầu vào đường đi.

reconst\_path.reverse() # Đảo ngược đường đi để có thứ tự đúng từ bắt đầu đến đích.

print(f'Đường đi được tìm thấy: {reconst\_path}')

return reconst\_path

# Duyệt qua tất cả các hàng xóm (m) của đỉnh n.

for (m, weight) in self.get\_neighbors(n):

if m not in open\_lst and m not in closed\_lst:

open\_lst.add(m) # Thêm m vào open\_lst.

came\_from[m] = n # Đặt n làm đỉnh liền trước của m.

g\_score[m] = g\_score[n] + weight # Cập nhật g\_score của m.

else:

if g\_score.get(m, float('inf')) > g\_score[n] + weight:

g\_score[m] = g\_score[n] + weight # Cập nhật g\_score của m.

came\_from[m] = n # Cập nhật đỉnh liền trước của m.

if m in closed\_lst:

closed\_lst.remove(m)

open\_lst.add(m)

open\_lst.remove(n) # Xóa n khỏi open\_lst vì nó đã được mở rộng hoàn toàn.

closed\_lst.add(n) # Thêm n vào closed\_lst.

print('Đường đi không tồn tại!')

return None

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

# Định nghĩa danh sách kề của đồ thị.

adjac\_lis = {

'A': [('B', 4), ('C', 2)],

'B': [('A', 4), ('D', 10), ('E', 12)],

'C': [('A', 2), ('E', 7)],

'D': [('B', 10), ('E', 6), ('Z', 15)],

'E': [('B', 12), ('C', 7), ('D', 6), ('Z', 9)],

'Z': [('D', 15), ('E', 9)]

}

# Định nghĩa các giá trị heuristic cho mỗi đỉnh.

h\_values = {

'A': 1,

'B': 1,

'C': 1,

'D': 1,

'E': 1,

'Z': 1

}

graph = Graph(adjac\_lis)

# Định nghĩa đỉnh bắt đầu và đỉnh kết thúc.

start\_node = 'A' # Thay thế bằng đỉnh bắt đầu bạn muốn

stop\_node = 'Z' # Thay thế bằng đỉnh kết thúc bạn muốn

# Gọi thuật toán A\* để tìm đường đi.

path = graph.a\_star\_algorithm(start\_node, stop\_node, h\_values)

# Vẽ đồ thị bằng thư viện networkx

G = nx.Graph() # Đồ thị vô hướng

for u in graph.adjac\_lis: # Duyệt qua từng đỉnh trong danh sách kề

for v, weight in graph.get\_neighbors(u): # Duyệt qua từng hàng xóm

G.add\_edge(u, v, weight=weight) # Thêm cạnh vào đồ thị

pos = nx.spring\_layout(G) # Bố trí vị trí các đỉnh

# Vẽ đồ thị

nx.draw(G, pos, with\_labels=True, node\_color='lightblue',

edge\_color='gray', node\_size=700, font\_size=16)

# Nếu có đường đi, vẽ đường đi

if path:

path\_edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)] # Tạo danh sách các cạnh của đường đi

nx.draw\_networkx\_edges(G, pos, edgelist=path\_edges, edge\_color='red', width=2) # Vẽ đường đi

plt.title("Đồ thị minh họa A\*") # Đặt tiêu đề cho đồ thị

plt.show() # Hiển thị hình vẽ

Mô tả DFS:

B1: Bắt đầu từ đỉnh gốc start, đưa vào tập open.

B2: Chọn đỉnh n trong open có giá trị f(n) = g(n) + h(n) nhỏ nhất.

B3: Nếu n là đỉnh đích → kết thúc, truy vết đường đi.

B4: Đưa n khỏi open, thêm vào closed.

B5: Duyệt các hàng xóm m của n:

Nếu m chưa trong open hoặc closed, thêm vào open, cập nhật g(m) và came\_from[m] = n.

Nếu m đã trong open hoặc closed nhưng có đường đi tốt hơn qua n, cập nhật lại g(m) và came\_from[m] = n. Nếu trong closed, chuyển m lại về open.

B6: Lặp lại từ bước 2 đến khi tìm được đường đi hoặc open rỗng.

Mô tả duyệt tìm kiểm:

A → B (4), A → C (2)

B → D (10), B → E (12)

C → E (7)

D → E (6), D → Z (15)

E → Z (9)