TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA Công Nghệ Thông Tin
BỘ MÔN: Công Nghệ Phần Mềm

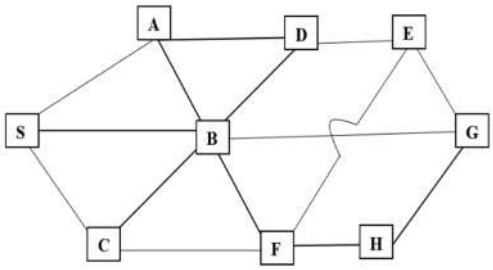
ĐỀ THI VÀ BÀI LÀM

Tên học phần: Trí tuệ nhân tạo
Mã học phần: Hình thức thi: *Tự luận có giám sát*
Đề số: **02** Thời gian làm bài: 75 phút *(không kể thời gian chép/phát đề)*
Được sử dụng tài liệu khi làm bài.

Họ tên: Trương Quang Lộc **Lớp**: 21TCLC_DT3……**MSSV**: 102210214……

Sinh viên làm bài trực tiếp trên tệp này, lưu tệp với định dạng MSSV_HọTên.pdf và nộp bài thông qua MSTeam:

Câu 1 (5 *điểm*): Cho đồ thị vô hướng G = (V,E) như hình vẽ với V là tập đỉnh và E là tập cạnh.



a) *(1 điểm)* Hãy viết đoạn code biểu diễn đồ thị trên bằng cách khởi tạo tập đỉnh V và tập cạnh E. (Ví dụ: V = [“S”, “A”, “B”], E =[(“S”, “A”), (“S”, “B”)])

 # Trả lời: Dán code vào bên dưới

Cách biểu diễn 1 (theo như ví dụ)

```
V = ["S", "A", "B", "C", "D", "F", "E", "H", "G"]
E = [("S", "A"), ("S", "B"), ("S", "C"),
     ("A", "D"), ("A", "B"),
     ("B", "D"), ("B", "G"), ("B", "F"),
     ("C", "B"), ("C", "F"),
     ("D", "E"),
     ("F", "E"), ("F", "H"),
     ("E", "G"), ("H", "G")]
```

Cách biểu diễn 2 (theo code bài làm)

```
# Create a graph
    graph = Graph()

    # Add edges to the graph
```

```
    graph.addEdge('S', 'A')
    graph.addEdge('S', 'B')
    graph.addEdge('S', 'C')

    graph.addEdge('A', 'D')
    graph.addEdge('A', 'B')

    graph.addEdge('B', 'D')
    graph.addEdge('B', 'G')
    graph.addEdge('B', 'F')

    graph.addEdge('C', 'B')
    graph.addEdge('C', 'F')

    graph.addEdge('D', 'E')

    graph.addEdge('F', 'E')
    graph.addEdge('F', 'H')

    graph.addEdge('E', 'G')

    graph.addEdge('H', 'G')
```

b) *(3 điểm)* Hãy viết chương trình sử dụng thuật toán **tìm kiếm theo chiều sâu (DFS)** để tìm đường đi từ đỉnh "S" đến đỉnh "G" trong đồ thị được biểu diễn ở câu a). Trong chương trình, hãy in ra thứ tự đỉnh khám phá trong quá trình tìm kiếm. Nếu không tìm thấy thì in "*Khong tim thay duong di*"

Trả lời: Dán code vào bên dưới

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.adjList = defaultdict(list)

    def addEdge(self, u, v):
        self.adjList[u].append(v)

    def dfs(self, currentNode, endNode, visited, parent):
        visited.append(currentNode)

        if currentNode == endNode:
            path = [endNode]
            while currentNode != start_vertex:
                path.append(parent[currentNode])
                currentNode = parent[currentNode]
            path.reverse()
            return path

        for neighbor in self.adjList[currentNode]:
```

```
            if neighbor not in visited:
                parent[neighbor] = currentNode
                path = self.dfs(neighbor, endNode, visited, parent)
                if path:
                    return path

        return None
def main():
    # Create a graph
    graph = Graph()

    # Add edges to the graph
    graph.addEdge('S', 'A')
    graph.addEdge('S', 'B')
    graph.addEdge('S', 'C')

    graph.addEdge('A', 'D')
    graph.addEdge('A', 'B')

    graph.addEdge('B', 'D')
    graph.addEdge('B', 'G')
    graph.addEdge('B', 'F')

    graph.addEdge('C', 'B')
    graph.addEdge('C', 'F')

    graph.addEdge('D', 'E')

    graph.addEdge('F', 'E')
    graph.addEdge('F', 'H')

    graph.addEdge('E', 'G')

    graph.addEdge('H', 'G')

    start_vertex = 'S'
    end_vertex = 'G'
    visited = []
    parent = {}
    path = graph.dfs(start_vertex, end_vertex, visited, parent)

    print("DFS found path from ", start_vertex, "to ", end_vertex, ":", path)

if __name__ == "__main__":
    main()
```

Trả lời: Dán kết quả thực thi vào bên dưới:

```
DFS found path from S to G : ['S', 'A', 'D', 'E', 'G']
```

c) *(1 điểm)* Hãy trực quan hóa kết quả tìm kiếm đường đi từ đỉnh "S" đến đỉnh "G"

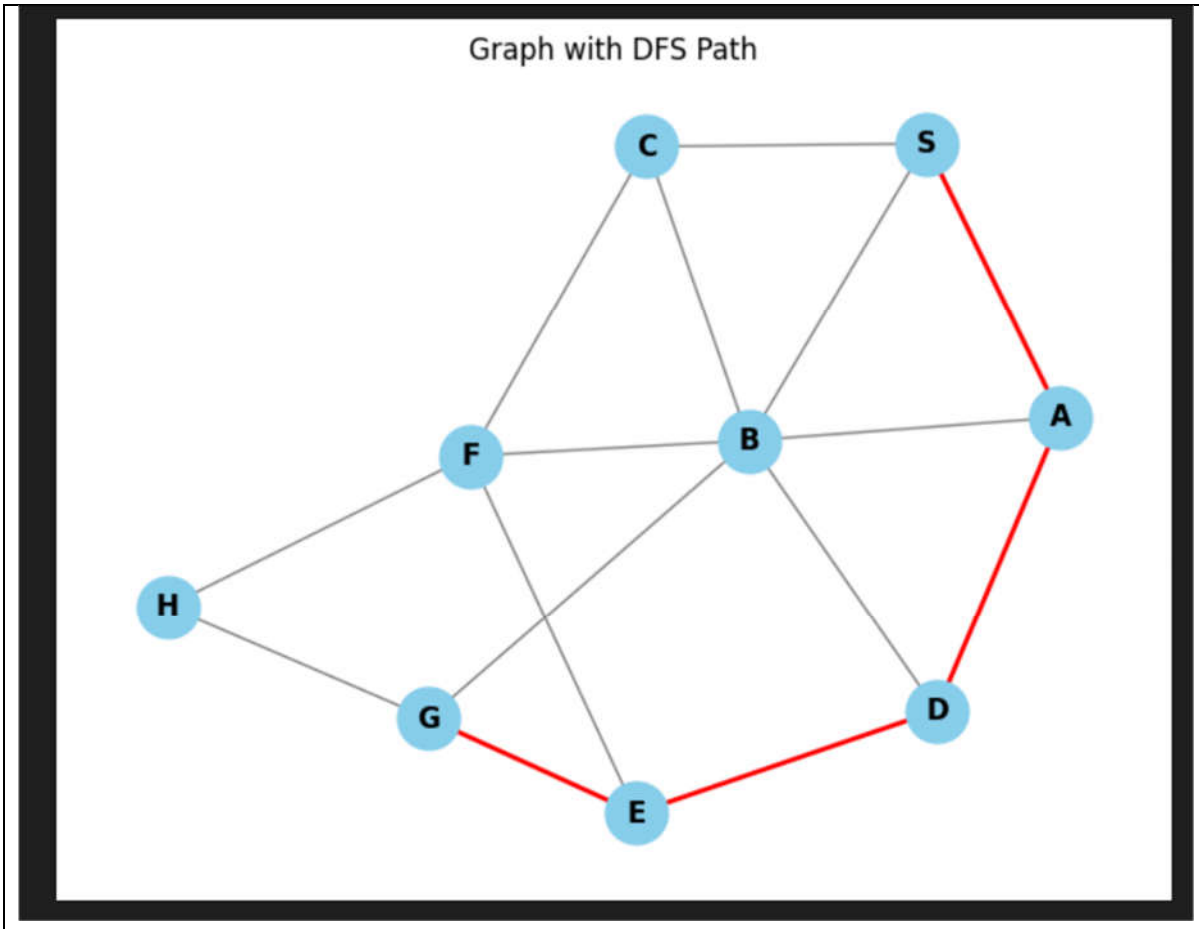
Trả lời: Dán code vào bên dưới

- Trực quan hóa sử dụng thư viện "networkx" và "matplotlib"

```
import networkx as nx # For visualizing the graph
import matplotlib.pyplot as plt
from collections import defaultdict

class Graph:
    def __init__(self):
        self.adjList = defaultdict(list)

    def addEdge(self, u, v):
        self.adjList[u].append(v)

    def dfs(self, currentNode, endNode, visited, parent):
        visited.append(currentNode)

        if currentNode == endNode:
            path = [endNode]
            while currentNode != start_vertex:
                path.append(parent[currentNode])
                currentNode = parent[currentNode]
            path.reverse()
            return path

        for neighbor in self.adjList[currentNode]:
            if neighbor not in visited:
                parent[neighbor] = currentNode
                path = self.dfs(neighbor, endNode, visited, parent)
                if path:
                    return path

        return None

# Create a graph
graph = Graph()

# Add edges to the graph
graph.addEdge('S', 'A')
graph.addEdge('S', 'B')
graph.addEdge('S', 'C')

graph.addEdge('A', 'D')
graph.addEdge('A', 'B')

graph.addEdge('B', 'D')
```

```
graph.addEdge('B', 'G')
graph.addEdge('B', 'F')

graph.addEdge('C', 'B')
graph.addEdge('C', 'F')

graph.addEdge('D', 'E')

graph.addEdge('F', 'E')
graph.addEdge('F', 'H')

graph.addEdge('E', 'G')

graph.addEdge('H', 'G')

start_vertex = 'S'
end_vertex = 'G'
visited = []
parent = {}
path = graph.dfs(start_vertex, end_vertex, visited, parent)

# Create a networkx graph
G = nx.Graph()
for u in graph.adjList:
    for v in graph.adjList[u]:
        G.add_edge(u, v)

# Draw the graph
pos = nx.spring_layout(G, seed=42)  # Position nodes using Fruchterman-Reingold force-directed algorithm
nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=700, font_size=12, font_weight='bold', edge_color='gray')

# Highlight the path
if path:
    path_edges = [(path[i], path[i+1]) for i in range(len(path)-1)]
    nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='red', width=2)

# Show the plot
plt.title("Graph with DFS Path")
plt.show()
```

Trả lời: Dán kết quả thực thi vào bên dưới:

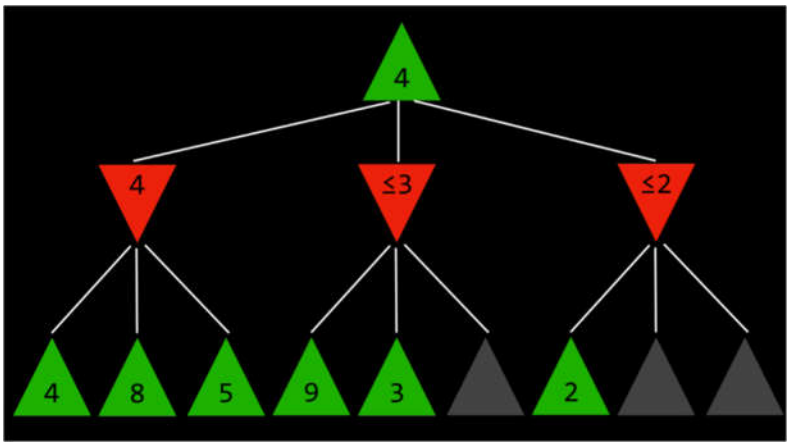


Câu 2 (3 *điểm*):

a) *(2 điểm)* Mô tả thuật toán hoặc hàm thực thi chiến lược cắt tỉa alpha- beta

 # Trả lời: viết mô tả thuật toán hoặc dán code vào bên dưới

- Cắt tỉa alpha-beta giúp cải thiện độ tối ưu của thuật toán Minimax bằng loại bỏ đi các nhánh không cần thiết trong quá trình tính toán trên cây đồ thị.
- Trong quá trình duyệt, nếu giá trị beta của nút Min nhỏ hơn hoặc bằng giá trị alpha của nút Max trước đó, hoặc ngược lại, ta có thể loại bỏ các nhánh không cần thiết, giúp cắt tỉa cây tìm kiếm và tăng tốc độ thực hiện.
- Ảnh ví dụ:



- Hàm MinValue:

```
def MinValue(node, alpha, beta):
    # nhận ba tham số:
    # node là nút hiện tại trong cây trò chơi,
    # alpha là giá trị tốt nhất mà nút cha có thể đạt được tại nút cha gốc,
    # beta là giá trị tốt nhất mà nút cha có thể đạt được khi tiến hành cắt tỉa.
    if len(node.children) == 0:
        return node
        # Nếu nút hiện tại không có con nào,
        # nghĩa là đây là nút lá của cây, chúng ta trả về giá trị của nút đó.

    # Khởi tạo giá trị của nút hiện tại với một giá trị lớn, để sau này có thể so sánh
và cập nhật giá trị của nút này.
    node.value = 100000

    # Duyệt qua các con của nút hiện tại.
    for child in node.children:
        # Gọi hàm MaxValue để tìm giá trị tốt nhất mà nút con có thể đạt được.
        temp = MaxValue(child, alpha, beta)

        # So sánh giá trị tìm được từ nút con với giá trị hiện tại của nút và cập nhật
giá trị của nút hiện tại.
        if temp.value < node.value:
            node.value = temp.value

        # Nếu giá trị của nút con nhỏ hơn hoặc bằng alpha, thực hiện cắt tỉa và trả về
nút con.
        if child.value <= alpha:
            return child

        # Nếu giá trị của nút con nhỏ hơn beta, cập nhật beta với giá trị của nút con.
        if child.value < beta:
            beta = child.value

    # Trả về None nếu không có nút con nào phù hợp với các điều kiện cắt tỉa
    return None
return node
```

- Hàm MaxValue tương tự MinValue nhưng thực hiện ngược lại.

b) *(1 điểm)* trình bày ứng dụng của chiến lược cắt tỉa alpha -beta

Trả lời: viết câu trả lời vào bên dưới
- Ứng dụng: Ví dụ trong trò chơi tic_tac_toe 3x3 có tới 362880 trường hợp khả thi, đối với cờ vua trong 40 nước đầu tiên có tới 30^80 nước đi, dẫn đến khả năng đáp ứng của bộ nhớ và tốc độ xử lý phần cứng không thể đáp ứng được yêu cầu. Thuật toán alpha-beta sẽ giúp loại bỏ các bước không cần thiết, giúp tiết kiệm bộ nhớ và tốc độ xử lý bài toán.

- Đoạn code thực hiện cắt tỉa alpha-beta với trò chơi tic_tac_toe:

```
def minimax(self, maximizing_player, depth, alpha = float('-inf'), beta=float('inf')):
        self.steps_computed += 1  # Increment the step count
        if self.is_winner('O'):
            # Nếu máy tính thắng trò chơi, trả về điểm 10 - depth
            return 10 - depth
        if self.is_winner('X'):
            # Nếu người chơi thắng trò chơi, trả về điểm depth - 10
            return depth - 10
        if ' ' not in self.board:
            # Trò chơi hòa, trả về 0
            return 0
        if maximizing_player:
            max_eval = float('-inf')
            for move in self.available_moves():
                self.board[move] = 'O'
                # eval = self.minimax(False, depth + 1)
                eval = self.minimax(False, depth + 1, alpha, beta)

                self.board[move] = ' '
                max_eval = max(max_eval, eval)

                # alpha-beta pruning
                alpha = max(alpha, eval)  # Update alpha
                if(self.steps_computed < 50):
                    print("maximizing_player")
                    print(f"eval: {eval}, max_eval: {max_eval}, alpha: {alpha}, beta:
{beta}")

                if eval >= beta:  # Prune
                    # print("Prune beta")
                    return eval
                #
            return max_eval
        else:
            min_eval = float('inf')
            for move in self.available_moves():
                self.board[move] = 'X'
                # eval = self.minimax(True, depth + 1)
                eval = self.minimax(True, depth + 1, alpha, beta)
                self.board[move] = ' '
                min_eval = min(min_eval, eval)
                # alpha-beta pruning
                beta = min(beta, eval)  # Update alpha
```

```
                if(self.steps_computed < 50):
                    print("minimizing_player")
                    print(f"eval: {eval}, min_eval: {min_eval}, alpha: {alpha}, beta:
{beta}")

                if eval <= alpha:  # Prune
                    # print("Prune alpha")
                    return eval
                #
            return min_eval
```

Câu 3(2 *điểm*): Cho hàm f(x) = $\left(e^{-x} - \frac{4}{e^{x}}\right)^{2}$

a) *(2 điểm)* Viết chương trình tính giá trị lớn nhất của hàm f(x) sử dụng thuật toán Gradient Descent with Momentum

Trả lời: viết câu trả lời vào bên dưới

```
import numpy as np
def f(x):
    return (np.exp(-x) - 4/np.exp(x))**2
def df(x):
    # return 6 * (np.exp(-2*x) - 4/np.exp(x))
    return -18/np.exp(2*x)

def gradient_descent_with_momentum(initial_x, learning_rate=0.01, momentum=0.9,
N_loops=1000, epsilon=1e-6):
    x = initial_x
    x_history = [x]
    velocity=0 # Tốc độ ban đầu Vo

    for i in range(N_loops):
        gradient=df(x)
        # velocity = momentum * velocity -learning_rate * gradient
        velocity=momentum*velocity+learning_rate*gradient
        x_old=x
        x=x-velocity
        if np.abs(x-x_old) < epsilon:
            print(f"Tìm được x min sau {i+1}vòng lặp.")
            break
        x_history.append(x)

    return x, f(x), x_history

def main():
    initial_x=-1.0# điểm bắt đầu
    learning_rate=0.001
    momentum=0.5
```

```
    N_loops=1000
    epsilon=1e-5# sai số
    min_x, min_value, x_history = gradient_descent_with_momentum(initial_x,
learning_rate, momentum, N_loops, epsilon)
    print("Minimum x:", min_x)
    print("Minimum value of f(x):", min_value)


main()
```

Trả lời: Dán kết quả thực thi vào bên dưới

```
Minimum x: 2.14381770182096
Minimum value of f(x): 0.12363633232480441
```

Đà Nẵng, ngày 27 tháng 3 năm 2024

GIẢNG VIÊN BIÊN SOẠN ĐỀ THI

TRƯỞNG BỘ MÔN
(đã duyệt)