Câu 1:

Quick sort

import random

import time

import threading

import heapq

def generate\_numbers\_file(filename="numbers.txt", num\_lines=25\_000\_000, max\_value=25\_000\_000):

    """Tạo file chứa 25 triệu số ngẫu nhiên."""

    with open(filename, "w") as f:

        for \_ in range(num\_lines):

            f.write(f"{random.randint(1, max\_value)}\n")

def quicksort(arr):

    """Quicksort tuần tự."""

    if len(arr) <= 1:

        return arr

    pivot = arr[len(arr) // 2]

    left = [x for x in arr if x < pivot]

    middle = [x for x in arr if x == pivot]

    right = [x for x in arr if x > pivot]

    return quicksort(left) + middle + quicksort(right)

def parallel\_quicksort(arr, num\_threads=4):

    """Quicksort song song sử dụng đa luồng."""

    if len(arr) <= 1:

        return arr

    pivot = arr[len(arr) // 2]

    left = [x for x in arr if x < pivot]

    middle = [x for x in arr if x == pivot]

    right = [x for x in arr if x > pivot]

    sorted\_left = []

    sorted\_right = []

    left\_thread = threading.Thread(target=lambda: sorted\_left.extend(quicksort(left)))

    right\_thread = threading.Thread(target=lambda: sorted\_right.extend(quicksort(right)))

    left\_thread.start()

    right\_thread.start()

    left\_thread.join()

    right\_thread.join()

    return sorted\_left + middle + sorted\_right

def merge\_sorted\_chunks(sorted\_chunks):

    """Hợp nhất các khối dữ liệu đã sắp xếp với dữ liệu động."""

    return list(heapq.merge(\*sorted\_chunks))

def read\_and\_sort(filename="numbers.txt", chunk\_size=1\_000\_000):

    """Đọc file theo từng khối và sắp xếp song song."""

    sorted\_chunks = []

    with open(filename, "r") as f:

        while chunk := f.readlines(chunk\_size):

            numbers = list(map(int, chunk))

            sorted\_chunks.append(parallel\_quicksort(numbers))

    sorted\_numbers = merge\_sorted\_chunks(sorted\_chunks)

    with open("sorted\_numbers.txt", "w") as f:

        for num in sorted\_numbers:

            f.write(f"{num}\n")

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

    start\_time = time.time()

    print("Đang tạo file...")

    generate\_thread = threading.Thread(target=generate\_numbers\_file)

    generate\_thread.start()

    generate\_thread.join()

    print("Đã tạo xong file numbers.txt")

    end\_time = time.time()

    print(f"Tổng thời gian tạo file: {end\_time - start\_time:.2f} giây")

    print("Bắt đầu sắp xếp...")

    sort\_start\_time = time.time()

    read\_and\_sort()

    sort\_end\_time = time.time()

    print(f"Tổng thời gian sắp xếp: {sort\_end\_time - sort\_start\_time:.2f} giây")

A computer screen with white text

AI-generated content may be incorrect.

Merge Sort and Bubble Sort:  
import random

import time

import threading

import heapq

from concurrent.futures import ThreadPoolExecutor

def generate\_numbers\_file(filename="numbers.txt", num\_lines=25\_000\_000, max\_value=25\_000\_000):

    with open(filename, "w") as f:

        f.writelines(f"{random.randint(1, max\_value)}\n" for \_ in range(num\_lines))

def merge\_sort(arr):

    if len(arr) <= 1:

        return arr

    mid = len(arr) // 2

    left = merge\_sort(arr[:mid])

    right = merge\_sort(arr[mid:])

    return merge(left, right)

def quick\_sort(arr):

    if len(arr) <= 1:

        return arr

    pivot = arr[len(arr) // 2]

    left = [x for x in arr if x < pivot]

    middle = [x for x in arr if x == pivot]

    right = [x for x in arr if x > pivot]

    return quick\_sort(left) + middle + quick\_sort(right)

def bubble\_sort(arr):

    n = len(arr)

    for i in range(n):

        swapped = False

        for j in range(0, n - i - 1):

            if arr[j] > arr[j + 1]:

                arr[j], arr[j + 1] = arr[j + 1], arr[j]

                swapped = True

        if not swapped:

            break

    return arr

def heap\_sort(arr):

    heapq.heapify(arr)

    return [heapq.heappop(arr) for \_ in range(len(arr))]

def merge(left, right):

    return list(heapq.merge(left, right))

def parallel\_merge\_sort(arr, num\_threads=4):

    if len(arr) <= 1:

        return arr

    chunk\_size = len(arr) // num\_threads

    chunks = [arr[i \* chunk\_size:(i + 1) \* chunk\_size] for i in range(num\_threads)]

    with ThreadPoolExecutor(max\_workers=num\_threads) as executor:

        sorted\_chunks = list(executor.map(merge\_sort, chunks))

    while len(sorted\_chunks) > 1:

        sorted\_chunks.append(merge(sorted\_chunks.pop(0), sorted\_chunks.pop(0)))

    return sorted\_chunks[0]

def parallel\_quick\_sort(arr, num\_threads=4):

    chunk\_size = len(arr) // num\_threads

    chunks = [arr[i \* chunk\_size:(i + 1) \* chunk\_size] for i in range(num\_threads)]

    with ThreadPoolExecutor(max\_workers=num\_threads) as executor:

        sorted\_chunks = list(executor.map(quick\_sort, chunks))

    while len(sorted\_chunks) > 1:

        sorted\_chunks.append(merge(sorted\_chunks.pop(0), sorted\_chunks.pop(0)))

    return sorted\_chunks[0]

def parallel\_bubble\_sort(arr, num\_threads=4):

    chunk\_size = len(arr) // num\_threads

    chunks = [arr[i \* chunk\_size:(i + 1) \* chunk\_size] for i in range(num\_threads)]

    with ThreadPoolExecutor(max\_workers=num\_threads) as executor:

        sorted\_chunks = list(executor.map(bubble\_sort, chunks))

    while len(sorted\_chunks) > 1:

        sorted\_chunks.append(merge(sorted\_chunks.pop(0), sorted\_chunks.pop(0)))

    return sorted\_chunks[0]

def parallel\_heap\_sort(arr, num\_threads=4):

    chunk\_size = len(arr) // num\_threads

    chunks = [arr[i \* chunk\_size:(i + 1) \* chunk\_size] for i in range(num\_threads)]

    with ThreadPoolExecutor(max\_workers=num\_threads) as executor:

        sorted\_chunks = list(executor.map(heap\_sort, chunks))

    while len(sorted\_chunks) > 1:

        sorted\_chunks.append(merge(sorted\_chunks.pop(0), sorted\_chunks.pop(0)))

    return sorted\_chunks[0]

def read\_and\_sort(filename="numbers.txt", chunk\_size=1\_000\_000, sort\_method="merge"):

    sorted\_chunks = []

    with open(filename, "r") as f:

        while True:

            chunk = f.readlines(chunk\_size)

            if not chunk:

                break

            numbers = list(map(int, chunk))

            if sort\_method == "merge":

                sorted\_chunks.append(parallel\_merge\_sort(numbers))

            elif sort\_method == "quick":

                sorted\_chunks.append(parallel\_quick\_sort(numbers))

            elif sort\_method == "bubble":

                sorted\_chunks.append(parallel\_bubble\_sort(numbers))

            elif sort\_method == "heap":

                sorted\_chunks.append(parallel\_heap\_sort(numbers))

            else:

                raise ValueError("Phương thức sắp xếp không hợp lệ")

    sorted\_numbers = list(heapq.merge(\*sorted\_chunks))

    output\_filename = f"sorted\_numbers\_{sort\_method}.txt"

    with open(output\_filename, "w") as f:

        f.writelines(f"{num}\n" for num in sorted\_numbers)

    print(f"Kết quả được lưu vào {output\_filename}")

def main():

    start\_time = time.time()

    print("Đang tạo file...")

    generate\_thread = threading.Thread(target=generate\_numbers\_file)

    generate\_thread.start()

    generate\_thread.join()

    print("Đã tạo xong file numbers.txt")

    print(f"Tổng thời gian tạo file: {time.time() - start\_time:.2f} giây")

    for method in ["merge", "quick", "bubble", "heap"]:

        print(f"Bắt đầu sắp xếp bằng {method}...")

        sort\_start\_time = time.time()

        read\_and\_sort(sort\_method=method)

        print(f"Tổng thời gian sắp xếp ({method}): {time.time() - sort\_start\_time:.2f} giây")

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

    main()

A computer screen with white text

AI-generated content may be incorrect.

Câu 2:

import concurrent.futures

import threading

from collections import defaultdict

lock = threading.Lock()  # Để bảo vệ việc cập nhật ma trận C

# Hàm nhân ma trận CSR bằng threading

def csr\_matrix\_multiply(A\_values, A\_columns, A\_row\_ptr, B\_values, B\_columns, B\_row\_ptr, A\_shape, B\_shape):

    C\_values = []

    C\_columns = []

    C\_row\_ptr = [0]

    with concurrent.futures.ThreadPoolExecutor() as executor:

        futures = []

        for i in range(A\_shape[0]):

            futures.append(executor.submit(multiply\_row, i, A\_values, A\_columns, A\_row\_ptr, B\_values, B\_columns, B\_row\_ptr, B\_shape))

        for future in concurrent.futures.as\_completed(futures):

            row\_values, row\_columns = future.result()

            with lock:  # Bảo vệ khi cập nhật C\_values và C\_columns

                C\_values.extend(row\_values)

                C\_columns.extend(row\_columns)

                C\_row\_ptr.append(len(C\_values))

    return C\_values, C\_columns, C\_row\_ptr

# Hàm nhân một hàng của A với toàn bộ ma trận B

def multiply\_row(i, A\_values, A\_columns, A\_row\_ptr, B\_values, B\_columns, B\_row\_ptr, B\_shape):

    row\_result = defaultdict(float)

    row\_start = A\_row\_ptr[i]

    row\_end = A\_row\_ptr[i+1]

    for j in range(row\_start, row\_end):

        A\_val = A\_values[j]

        A\_col = A\_columns[j]

        with lock:  # Đảm bảo an toàn khi đọc B\_values động

            B\_range\_start = B\_row\_ptr[A\_col]

            B\_range\_end = B\_row\_ptr[A\_col+1]

        for k in range(B\_range\_start, B\_range\_end):

            with lock:

                B\_val = B\_values[k]

                B\_col = B\_columns[k]

            row\_result[B\_col] += A\_val \* B\_val

    row\_values = list(row\_result.values())

    row\_columns = list(row\_result.keys())

    return row\_values, row\_columns

# Hàm kiểm tra với ma trận ví dụ

def test\_matrix\_multiplication():

    A\_values = [1, 3, 4]

    A\_columns = [0, 2, 0]

    A\_row\_ptr = [0, 1, 2, 3]

    A\_shape = (3, 3)

    B\_values = [1, 2, 5]

    B\_columns = [0, 1, 2]

    B\_row\_ptr = [0, 1, 2, 3]

    B\_shape = (3, 3)

    C\_values, C\_columns, C\_row\_ptr = csr\_matrix\_multiply(A\_values, A\_columns, A\_row\_ptr, B\_values, B\_columns, B\_row\_ptr, A\_shape, B\_shape)

    print("C\_values:", C\_values)

    print("C\_columns:", C\_columns)

    print("C\_row\_ptr:", C\_row\_ptr)

test\_matrix\_multiplication()

A screenshot of a computer

AI-generated content may be incorrect.

Câu 3:

import numpy as np

from PIL import Image

from concurrent.futures import ThreadPoolExecutor

# Chia ảnh thành các khối nhỏ đồng nhất

def split\_image(image, block\_size=(100, 100)):

    image\_array = np.array(image)

    blocks = []

    h, w = image\_array.shape[:2]

    # Chia ảnh thành các khối 100x100 (hoặc kích thước khác)

    for i in range(0, h, block\_size[0]):

        for j in range(0, w, block\_size[1]):

            block = image\_array[i:i+block\_size[0], j:j+block\_size[1]]

            # Thêm padding nếu khối nhỏ hơn kích thước yêu cầu

            if block.shape[0] < block\_size[0] or block.shape[1] < block\_size[1]:

                padded\_block = np.zeros((block\_size[0], block\_size[1], 3), dtype=block.dtype)

                padded\_block[:block.shape[0], :block.shape[1]] = block

                blocks.append(padded\_block)

            else:

                blocks.append(block)

    return blocks

# Áp dụng bộ lọc cho từng khối ảnh

def apply\_filter\_to\_block(block, filter\_params):

    # Ví dụ áp dụng bộ lọc thay đổi độ sáng

    brightness = filter\_params['brightness']

    return block \* brightness

# Xử lý ảnh song song

def process\_image\_parallel(image, filter\_params):

    blocks = split\_image(image)

    # Sử dụng ThreadPoolExecutor để xử lý các khối ảnh song song

    with ThreadPoolExecutor() as executor:

        processed\_blocks = list(executor.map(lambda block: apply\_filter\_to\_block(block, filter\_params), blocks))

    # Kết hợp các khối lại với nhau

    block\_size = processed\_blocks[0].shape[0]  # Lấy kích thước của khối (đảm bảo tất cả các khối có cùng kích thước)

    # Tính số lượng khối trên mỗi hàng (số cột)

    num\_blocks\_per\_row = image.width // block\_size

    rows = []

    for i in range(0, len(processed\_blocks), num\_blocks\_per\_row):

        row\_blocks = processed\_blocks[i:i + num\_blocks\_per\_row]

        # Kiểm tra nếu số lượng khối trong hàng cuối không đủ, thêm padding cho đủ

        if len(row\_blocks) < num\_blocks\_per\_row:

            row\_blocks += [np.zeros\_like(row\_blocks[0])] \* (num\_blocks\_per\_row - len(row\_blocks))

        row = np.concatenate(row\_blocks, axis=1)

        rows.append(row)

    # Kết hợp các hàng lại với nhau

    return np.concatenate(rows, axis=0)

# Đọc ảnh và áp dụng bộ lọc

image = Image.open('image.jpg')

filter\_params = {'brightness': 1.2}  # Ví dụ về bộ lọc

result\_image = process\_image\_parallel(image, filter\_params)

# Chuyển đổi lại thành ảnh để hiển thị

result\_image\_pil = Image.fromarray(result\_image.astype(np.uint8))

result\_image\_pil.show()

Câu 4:

# Câu 4:

import heapq

import random

from concurrent.futures import ThreadPoolExecutor, as\_completed

import threading

# Hàm Dijkstra

def dijkstra\_parallel(graph, start\_vertex):

    # Số đỉnh trong đồ thị

    num\_vertices = len(graph)

    # Khoảng cách ban đầu từ start\_vertex

    distances = {vertex: float('inf') for vertex in range(num\_vertices)}

    distances[start\_vertex] = 0

    # Priority Queue (Min-heap)

    min\_heap = [(0, start\_vertex)]  # (distance, vertex)

    heapq.heapify(min\_heap)

    # Đảm bảo tính đồng bộ khi truy cập và thay đổi các phần tử

    lock = threading.Lock()

    # Hàm xử lý mỗi đỉnh trong đồ thị

    def process\_vertex(vertex):

        with lock:

            # Lấy đỉnh có khoảng cách tối thiểu

            distance, current\_vertex = heapq.heappop(min\_heap)

            if distance > distances[current\_vertex]:

                return

            # Cập nhật các đỉnh kề

            for neighbor, weight in graph[current\_vertex]:

                new\_distance = distances[current\_vertex] + weight

                if new\_distance < distances[neighbor]:

                    distances[neighbor] = new\_distance

                    heapq.heappush(min\_heap, (new\_distance, neighbor))

    # Sử dụng ThreadPoolExecutor để xử lý song song

    with ThreadPoolExecutor() as executor:

        futures = []

        for vertex in range(num\_vertices):

            futures.append(executor.submit(process\_vertex, vertex))

        # Đảm bảo tất cả các thread hoàn thành

        for future in as\_completed(futures):

            future.result()

    return distances

# Hàm thêm cạnh vào đồ thị

def add\_edge(graph, u, v, weight):

    graph[u].append((v, weight))

    graph[v].append((u, weight))

# Hàm xóa cạnh khỏi đồ thị

def remove\_edge(graph, u, v):

    graph[u] = [edge for edge in graph[u] if edge[0] != v]

    graph[v] = [edge for edge in graph[v] if edge[0] != u]

# Tạo đồ thị mẫu

def create\_graph(num\_vertices):

    graph = {i: [] for i in range(num\_vertices)}

    for \_ in range(100):

        u, v = random.sample(range(num\_vertices), 2)

        weight = random.randint(1, 10)

        add\_edge(graph, u, v, weight)

    return graph

# Sử dụng thuật toán Dijkstra song song

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

    num\_vertices = 10  # Số đỉnh trong đồ thị

    graph = create\_graph(num\_vertices)

    print("Khoảng cách ban đầu:")

    print(dijkstra\_parallel(graph, 0))

    # Thêm và xóa các cạnh

    print("\nSau khi thêm cạnh:")

    add\_edge(graph, 0, 4, 5)

    print(dijkstra\_parallel(graph, 0))

    print("\nSau khi xóa cạnh:")

    remove\_edge(graph, 0, 4)

    print(dijkstra\_parallel(graph, 0))

A black screen with white text

AI-generated content may be incorrect.

Câu 5:

#CÂU 5:

import random

import math

from concurrent.futures import ThreadPoolExecutor, as\_completed

# Hàm tính số điểm trong hình tròn

def monte\_carlo\_simulation(num\_samples):

    inside\_circle = 0

    # Thực hiện mô phỏng cho từng điểm ngẫu nhiên

    for \_ in range(num\_samples):

        x = random.uniform(-1, 1)

        y = random.uniform(-1, 1)

        # Kiểm tra xem điểm có nằm trong hình tròn không

        if x\*\*2 + y\*\*2 <= 1:

            inside\_circle += 1

    return inside\_circle

# Hàm chính để tính π song song với phản hồi thời gian thực

def estimate\_pi\_parallel(total\_samples, feedback\_interval=1000, max\_threads=4):

    samples\_per\_thread = total\_samples // max\_threads

    results = []

    # Sử dụng ThreadPoolExecutor để thực hiện song song

    with ThreadPoolExecutor(max\_workers=max\_threads) as executor:

        futures = []

        # Chia công việc cho các thread

        for i in range(max\_threads):

            futures.append(executor.submit(monte\_carlo\_simulation, samples\_per\_thread))

        inside\_circle\_total = 0

        for idx, future in enumerate(as\_completed(futures)):

            inside\_circle\_total += future.result()

            # Cập nhật giá trị π và điều chỉnh chiến lược lấy mẫu sau mỗi feedback\_interval

            if (idx + 1) \* samples\_per\_thread >= feedback\_interval:

                pi\_estimate = 4 \* inside\_circle\_total / ((idx + 1) \* samples\_per\_thread)

                print(f"Feedback: ước tính giá trị của π sau {idx + 1} phần công việc là: {pi\_estimate}")

                # Cải thiện tốc độ hội tụ (có thể thay đổi chiến lược lấy mẫu ở đây nếu cần)

    pi\_final = 4 \* inside\_circle\_total / total\_samples

    print(f"Giá trị cuối cùng của π: {pi\_final}")

    return pi\_final

# Ví dụ sử dụng

total\_samples = 10000000  # Tổng số mẫu muốn sử dụng

estimate\_pi\_parallel(total\_samples)

A black screen with white text

AI-generated content may be incorrect.

Câu 6:  
import zlib

import threading

from concurrent.futures import ThreadPoolExecutor

import random

import os

# Hàm nén dữ liệu sử dụng RLE (Run-Length Encoding)

def run\_length\_encode(data):

    encoded = []

    i = 0

    while i < len(data):

        count = 1

        while i + 1 < len(data) and data[i] == data[i + 1]:

            i += 1

            count += 1

        encoded.append((data[i], count))

        i += 1

    return encoded

# Hàm kiểm tra mức độ dư thừa trong dữ liệu

def is\_data\_redundant(data):

    freq = {}

    for byte in data:

        if byte in freq:

            freq[byte] += 1

        else:

            freq[byte] = 1

    # Nếu một ký tự chiếm hơn 50% dữ liệu, ta cho rằng có dư thừa

    max\_freq = max(freq.values())

    return max\_freq > len(data) / 2

# Hàm xử lý một khối dữ liệu và nén nó

def process\_chunk(data):

    if is\_data\_redundant(data):

        # Nén dữ liệu bằng RLE nếu dữ liệu có dư thừa

        return run\_length\_encode(data)

    else:

        # Nếu dữ liệu không dư thừa, sử dụng nén gzip (DEFLATE)

        return zlib.compress(data)

# Hàm nén tệp song song

def parallel\_file\_compression(input\_file, output\_file, chunk\_size=1024, max\_threads=4):

    with open(input\_file, 'rb') as infile, open(output\_file, 'wb') as outfile:

        data = infile.read(chunk\_size)

        with ThreadPoolExecutor(max\_workers=max\_threads) as executor:

            futures = []

            while data:

                futures.append(executor.submit(process\_chunk, data))

                data = infile.read(chunk\_size)

            for future in futures:

                compressed\_chunk = future.result()

                if isinstance(compressed\_chunk, bytes):

                    outfile.write(compressed\_chunk)

                else:

                    # Nếu kết quả là dạng dữ liệu (ví dụ: RLE) ta cần phải chuyển thành byte stream

                    outfile.write(bytes(str(compressed\_chunk), 'utf-8'))

# Ví dụ sử dụng

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

    input\_file = 'input.txt'  # Đảm bảo rằng tệp này tồn tại

    output\_file = 'compressed\_output.bin'

    parallel\_file\_compression(input\_file, output\_file)

    print(f"Tệp đã được nén và lưu vào {output\_file}")

Câu 7:

#Cau 7

import random

import time

# Hàm tính tổng khoảng cách của một hành trình

def calculate\_distance(tour, distance\_matrix):

    total\_distance = 0

    for i in range(len(tour) - 1):

        total\_distance += distance\_matrix[tour[i]][tour[i + 1]]

    total\_distance += distance\_matrix[tour[-1]][tour[0]]  # Quay lại điểm xuất phát

    return total\_distance

# Hàm tạo quần thể ban đầu

def create\_initial\_population(num\_cities, population\_size):

    population = []

    for \_ in range(population\_size):

        tour = list(range(num\_cities))

        random.shuffle(tour)

        population.append(tour)

    return population

# Hàm lai ghép (crossover)

def crossover(parent1, parent2):

    child = [-1] \* len(parent1)

    start, end = sorted(random.sample(range(len(parent1)), 2))

    child[start:end] = parent1[start:end]

    for i in range(len(parent2)):

        if parent2[i] not in child:

            for j in range(len(child)):

                if child[j] == -1:

                    child[j] = parent2[i]

                    break

    return child

# Hàm đột biến (mutation)

def mutate(tour, mutation\_rate):

    if random.random() < mutation\_rate:

        idx1, idx2 = random.sample(range(len(tour)), 2)

        tour[idx1], tour[idx2] = tour[idx2], tour[idx1]

    return tour

# Thuật toán di truyền song song đơn giản

def parallel\_genetic\_algorithm(distance\_matrix, population\_size=100, generations=100, mutation\_rate=0.01):

    num\_cities = len(distance\_matrix)

    population = create\_initial\_population(num\_cities, population\_size)

    for generation in range(generations):

        # Đánh giá độ thích nghi của từng cá thể

        fitness\_scores = [calculate\_distance(tour, distance\_matrix) for tour in population]

        # Chọn lọc cá thể tốt nhất

        best\_tour = population[fitness\_scores.index(min(fitness\_scores))]

        # Tạo quần thể mới

        new\_population = [best\_tour]  # Giữ lại cá thể tốt nhất

        while len(new\_population) < population\_size:

            parent1, parent2 = random.choices(population, k=2)

            child = crossover(parent1, parent2)

            child = mutate(child, mutation\_rate)

            new\_population.append(child)

        population = new\_population

    return best\_tour, calculate\_distance(best\_tour, distance\_matrix)

# Ví dụ sử dụng

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

    # Ma trận khoảng cách giữa các thành phố

    distance\_matrix = [

        [0, 10, 15, 20],

        [10, 0, 35, 25],

        [15, 35, 0, 30],

        [20, 25, 30, 0]

    ]

    start\_time = time.time()

    best\_tour, best\_distance = parallel\_genetic\_algorithm(distance\_matrix)

    end\_time = time.time()

    print(f"Best Tour: {best\_tour}")

    print(f"Best Distance: {best\_distance}")

    print(f"Time: {end\_time - start\_time} seconds")

A black screen with white text

AI-generated content may be incorrect.

Câu 8:

import numpy as np

import concurrent.futures

import time

import random

# Thông số mô phỏng

GRID\_SIZE = 100  # Lưới 100x100

TIME\_STEPS = 50  # Số bước thời gian

VISCOSITY = 0.1  # Độ nhớt của chất lỏng

# Khởi tạo lưới vận tốc (u, v) và áp suất p

velocity\_u = np.zeros((GRID\_SIZE, GRID\_SIZE))

velocity\_v = np.zeros((GRID\_SIZE, GRID\_SIZE))

pressure = np.zeros((GRID\_SIZE, GRID\_SIZE))

# Danh sách vật cản động (sẽ thay đổi theo thời gian)

obstacles = set()

# Hàm cập nhật vận tốc theo phương trình Navier-Stokes (đơn giản hóa)

def update\_velocity(x, y):

    if (x, y) in obstacles:

        return 0, 0  # Nếu là vật cản, vận tốc = 0

    u\_new = velocity\_u[x, y] + VISCOSITY \* (velocity\_u[(x-1) % GRID\_SIZE, y] + velocity\_u[(x+1) % GRID\_SIZE, y] +

                                            velocity\_u[x, (y-1) % GRID\_SIZE] + velocity\_u[x, (y+1) % GRID\_SIZE] -

                                            4 \* velocity\_u[x, y])

    v\_new = velocity\_v[x, y] + VISCOSITY \* (velocity\_v[(x-1) % GRID\_SIZE, y] + velocity\_v[(x+1) % GRID\_SIZE, y] +

                                            velocity\_v[x, (y-1) % GRID\_SIZE] + velocity\_v[x, (y+1) % GRID\_SIZE] -

                                            4 \* velocity\_v[x, y])

    return u\_new, v\_new

# Hàm cập nhật áp suất

def update\_pressure(x, y):

    if (x, y) in obstacles:

        return pressure[x, y]

    return (pressure[(x-1) % GRID\_SIZE, y] + pressure[(x+1) % GRID\_SIZE, y] +

            pressure[x, (y-1) % GRID\_SIZE] + pressure[x, (y+1) % GRID\_SIZE]) / 4

# Cập nhật vật cản động (có thể xuất hiện/ngẫu nhiên thay đổi)

def update\_obstacles():

    if random.random() < 0.1:  # Xác suất thay đổi vật cản 10%

        new\_x = random.randint(10, GRID\_SIZE - 10)

        new\_y = random.randint(10, GRID\_SIZE - 10)

        if (new\_x, new\_y) in obstacles:

            obstacles.remove((new\_x, new\_y))  # Loại bỏ vật cản cũ

        else:

            obstacles.add((new\_x, new\_y))  # Thêm vật cản mới

# Hàm chạy mô phỏng song song

def run\_simulation():

    global velocity\_u, velocity\_v, pressure

    for step in range(TIME\_STEPS):

        update\_obstacles()  # Cập nhật vật cản động

        # Cập nhật vận tốc song song

        with concurrent.futures.ThreadPoolExecutor() as executor:

            future\_velocities = {(x, y): executor.submit(update\_velocity, x, y)

                                 for x in range(GRID\_SIZE) for y in range(GRID\_SIZE)}

        for (x, y), future in future\_velocities.items():

            velocity\_u[x, y], velocity\_v[x, y] = future.result()

        # Cập nhật áp suất song song

        with concurrent.futures.ThreadPoolExecutor() as executor:

            future\_pressures = {(x, y): executor.submit(update\_pressure, x, y)

        for x in range(GRID\_SIZE) for y in range(GRID\_SIZE)}

        for (x, y), future in future\_pressures.items():

            pressure[x, y] = future.result()

        print(f"Step {step + 1}/{TIME\_STEPS} completed.")

# Chạy mô phỏng

start\_time = time.time()

run\_simulation()

end\_time = time.time()

print(f"Mô phỏng hoàn tất sau {end\_time - start\_time:.2f} giây.")

A screenshot of a computer

AI-generated content may be incorrect.

Cau 9:

#Cau 9

import random

import time

# Hàm tìm các tập phổ biến

def find\_frequent\_itemsets(transactions, min\_support):

    item\_counts = {}

    for transaction in transactions:

        for item in transaction:

            item\_counts[item] = item\_counts.get(item, 0) + 1

    frequent\_itemsets = {frozenset([item]): count for item, count in item\_counts.items() if count >= min\_support}

    return frequent\_itemsets

# Hàm sinh các tập ứng viên mới

def generate\_candidates(prev\_itemsets, k):

    candidates = set()

    for itemset1 in prev\_itemsets:

        for itemset2 in prev\_itemsets:

            union = itemset1.union(itemset2)

            if len(union) == k:

                candidates.add(union)

    return candidates

# Thuật toán Apriori song song đơn giản

def parallel\_apriori(transactions, min\_support=2):

    frequent\_itemsets = find\_frequent\_itemsets(transactions, min\_support)

    k = 2

    while True:

        candidates = generate\_candidates(frequent\_itemsets.keys(), k)

        if not candidates:

            break

        candidate\_counts = {candidate: 0 for candidate in candidates}

        for transaction in transactions:

            for candidate in candidates:

                if candidate.issubset(transaction):

                    candidate\_counts[candidate] += 1

        new\_frequent\_itemsets = {itemset: count for itemset, count in candidate\_counts.items() if count >= min\_support}

        if not new\_frequent\_itemsets:

            break

        frequent\_itemsets.update(new\_frequent\_itemsets)

        k += 1

    return frequent\_itemsets

# Ví dụ sử dụng

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

    transactions = [

        {'A', 'B', 'C'},

        {'A', 'B'},

        {'A', 'C'},

        {'A'},

        {'B', 'C'},

        {'B'},

        {'C'}

    ]

    start\_time = time.time()

    frequent\_itemsets = parallel\_apriori(transactions, min\_support=2)

    end\_time = time.time()

    print("Frequent Itemsets:")

    for itemset, count in frequent\_itemsets.items():

        print(f"{itemset}: {count}")

    print(f"Time: {end\_time - start\_time} seconds")

A screen shot of a computer program

AI-generated content may be incorrect.

Cau 10:

#Cau 10

import random

import time

# Hàm tính đầu ra của mạng neural

def neural\_network(input\_data, weights):

    return sum(x \* w for x, w in zip(input\_data, weights))

# Hàm cập nhật trọng số

def update\_weights(weights, input\_data, target, learning\_rate=0.01):

    prediction = neural\_network(input\_data, weights)

    error = target - prediction

    for i in range(len(weights)):

        weights[i] += learning\_rate \* error \* input\_data[i]

    return weights

# Thuật toán huấn luyện mạng neural song song đơn giản

def parallel\_neural\_network(training\_data, weights, epochs=100, learning\_rate=0.01):

    for epoch in range(epochs):

        for input\_data, target in training\_data:

            weights = update\_weights(weights, input\_data, target, learning\_rate)

    return weights

# Ví dụ sử dụng

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

    training\_data = [

        ([0, 0], 0),

        ([0, 1], 1),

        ([1, 0], 1),

        ([1, 1], 0)

    ]

    weights = [random.random() for \_ in range(2)]

    start\_time = time.time()

    trained\_weights = parallel\_neural\_network(training\_data, weights)

    end\_time = time.time()

    print(f"Trained Weights: {trained\_weights}")

    print(f"Time: {end\_time - start\_time} seconds")

A black screen with white text

AI-generated content may be incorrect.