Elufowoju David

Coding Report

1(lab 4)-

### \*\*\*\*Code provided\*\*\*\*

1b)

#### Input: "TTTTTATTTACGCCA"

#### ****Step-by-Step Execution:****

i = 0: First character is 'T'. Count 5 consecutive 'T'.  
y = "T5", move i to index 5.

i = 5: Character is 'A'. No repetition.  
y = "T5A", move i to index 6.

i = 6: Character is 'T'. Count 4 consecutive 'T'.  
y = "T5AT4", move i to index 10.

i = 10: Character is 'A'. No repetition.  
y = "T5AT4A", move i to index 11.

i = 11: Character is 'C'. Count 2 consecutive 'C'.  
y = "T5AT4AC2", move i to index 13.

i = 13: Character is 'G'. No repetition.  
y = "T5AT4AC2G", move i to index 14.

i = 14: Character is 'C'. No repetition.  
y = "T5AT4AC2GC", move i to index 15.

**Output**: "T5AT4AC2GC"

C)

### ****Modified Function with Only One Loop****

The pseudocode uses two loops: a while loop and a nested for loop. This can be rewritten to use just one loop with similar logic.

\*\*\*\*Code Provided\*\*\*\*

### Testing the Function

#### Input: "TTTTTATTTACGCCA"

Both functions will produce the same output.

**Output**: "T5AT4AC2GC"

The second implementation reduces redundancy and uses only one loop.

2(lab 6)-

Function to find the Minimum Spanning Tree (MST) using Prim's algorithm.

    Parameters:

    graph (dict): The input graph represented as an adjacency list.

                  Each key is a node, and its value is a list of tuples,

                  where each tuple represents (neighbor, weight).

                  Example: {

                      'A': [('B', 1), ('D', 3)],

                      'B': [('A', 1), ('C', 2)],

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

                      'D': [('A', 3), ('C', 4)]

                  }

    Returns:

    list: A list of edges that form the MST. Each edge is represented as

          a tuple (node1, node2, weight).

\*\*\*\*Code Provided\*\*\*\*

3(lab 7)

Applies Beasley's reduction rules to the set covering problem.

Parameters:

universal\_set (list): The universal set U containing all elements.

subsets (list of lists): A list of subsets of the universal set U.

Returns:

tuple: A tuple containing three lists:

- Updated universal set (list)

- Updated list of subsets (list of lists)

- Mandatory subsets (list of lists)

\*\*\*\* Code Provided \*\*\*\*

4)(lab 8)

The Graph Coloring Problem involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. A GRASP (Greedy Randomized Adaptive Search Procedure) approach combines greedy heuristics with randomization to iteratively refine solutions. The algorithm starts by randomly selecting a vertex and assigning colors greedily to minimize conflicts. If conflicts exist, it uses a local search strategy to refine the coloring. The process iterates, aiming to minimize the number of colors used and resolve conflicts until a satisfactory solution is found or a maximum iteration limit is reached.

Solves the graph coloring problem using a GRASP approach. :param graph: A dictionary where keys are vertices and values are lists of adjacent vertices. :param max\_iterations: Maximum number of iterations to refine the solution. :return: A dictionary of vertex-color assignments.

\*\*\*\*Code Provided\*\*\*\*

This implementation demonstrates the GRASP approach with iterative refinement and randomization.

3rd assesed)

\*\*\*\* Code Provided \*\*\*\*

4th assesed)

\*\*\*\* Code Provided \*\*\*\*

4b)

· **Bubble Sort**: O(n2)O(n^2)O(n2)

· **Insertion Sort**: O(n2)O(n^2)O(n2)

· **Selection Sort**: O(n2)O(n^2)O(n2)

· **Merge Sort**: O(nlog⁡ n)O(n \log n)O(nlogn)

5th assesed)

\*\*\*\* Code Provided \*\*\*\*

### ****Explanation of Changes****:

**Threaded Execution for** fun3:

* 1. In fun5, the calls to fun3 are parallelized by using the threading.Thread class. Each call to fun3 is run in its own thread, with the results being stored in a shared result dictionary, where the thread index (i) is used as the key.

· **Threaded Execution for** fun4:

* · Similarly, the calls to fun4 are parallelized in the same manner. Each thread computes the result for a different candidate solution in C\_pool, and the results are stored in the shared result dictionary.

· **Final Evaluation**:

* · After collecting results from fun3 and fun4, the best solution is evaluated in a standard loop, but this part is not parallelized as it is a straightforward comparison step.

5b)

### ****Justification for Choosing**** threading

The threading module was chosen because it offers direct and straightforward control over threads, making it suitable for this problem. Here’s a detailed explanation of why threading is the most appropriate choice and why the other modules were not used:

### ****Advantages of Using**** threading

**Lightweight and Effective for I/O-Bound Tasks**:

* 1. The problem involves tasks like generating and refining candidate solutions (fun3 and fun4), which are not computationally intensive but involve frequent function calls and intermediate operations. Threads are lightweight and well-suited for such I/O-bound or moderately CPU-bound workloads.

**Simplicity and Explicit Control**:

* 1. The threading module gives explicit control over thread creation, lifecycle management, and synchronization.
  2. For a relatively simple task like this, threading enables parallelism without the abstraction layer introduced by higher-level modules like concurrent.futures.

**Shared Memory Access**:

* 1. Threads in Python share memory by default. This eliminates the need for complex inter-process communication mechanisms like data serialization, as required by multiprocessing.
  2. The problem involves multiple threads working on shared data (e.g., C\_pool), making threading more efficient than processes.

**Low Overhead**:

* 1. Threads are less resource-intensive compared to processes because they run within the same memory space, avoiding the overhead of process creation and communication.

**Flexibility for Fine-Tuned Parallelism**:

* 1. threading allows detailed tuning of thread behavior, such as managing locks or semaphores if needed. This level of control is beneficial when designing algorithms that involve iterative improvement of solutions, as in fun4.

### ****Why Not the Other Modules?****

#### ****1.**** concurrent.futures:

* **Abstraction Overhead**:
  + While concurrent.futures simplifies parallel programming, it abstracts away thread management, which may reduce the ability to fine-tune performance for small-scale parallel tasks like this one.
* **Unnecessary Complexity**:
  + For a problem with a small number of threads (e.g., two threads for fun3), managing a thread pool using ThreadPoolExecutor adds unnecessary complexity.

#### ****2.**** multiprocessing:

* **Heavyweight Processes**:
  + The multiprocessing module creates separate processes, which are more resource-intensive than threads.
* **Serialization Overhead**:
  + Data sharing between processes requires serialization (pickling), which introduces overhead and complexity.
* **Unnecessary for I/O-Bound Tasks**:
  + The tasks in this problem are not CPU-bound, and threads are sufficient for improving performance.

#### ****3.**** asyncio:

* **Event-Driven Model**:
  + asyncio is designed for asynchronous I/O tasks like managing thousands of network connections, not for CPU-based or I/O-bound tasks requiring concurrent execution.
* **Complexity**:
  + Using async and await adds complexity to the code, which is unnecessary for this problem where tasks can be executed in parallel using threads.
* **Does Not Leverage Multiple Cores**:
  + Unlike threads or processes, asyncio runs on a single thread and does not leverage multiple CPU cores effectively.

### ****Conclusion****

The threading module was chosen because it provides explicit and efficient control over thread creation and management. It is lightweight, avoids the overhead of processes or unnecessary abstraction layers, and is well-suited for tasks involving shared memory. This makes it the most appropriate choice for parallelizing the given GRASP algorithm implementation.

6th assesed)

\*\*\*\* Code Provided \*\*\*\*

Explanation-

For the Input:

Graph: A 2D adjacency matrix represents the graph. Graph[I][j] = 1 indicates an edge between nodes I and j.

Functions:

Is\_safe function checks if a color can be assigned to a vertex without conflicting with its neighbors.

graph\_coloring\_backtrack function uses backtracking to assign colors to vertices

find\_min\_colors: iteratively tries increasing number of colors until a valid color is found

For the Output:

The function returns the minimum number of colors needed to color the graph

6b)

### ****Key Characteristics of Backtracking****:

**Recursive Exploration of Choices**:

* 1. The algorithm explores assigning colors to vertices one by one. It recursively moves to the next vertex after successfully assigning a color to the current vertex.

**Validation of Constraints**:

* 1. The is\_safe function ensures that the chosen color for a vertex does not violate the constraints of the problem (no two adjacent vertices have the same color).

**Backtracking on Failure**:

* 1. If assigning a color to a vertex leads to a dead end (e.g., no valid colors remain for subsequent vertices), the algorithm removes the color assignment (backtracks) and tries the next possible color.

**Exhaustive Search**:

* 1. The algorithm explores all possible combinations of color assignments systematically. It only stops when it finds a valid coloring or exhausts all possibilities.

### ****How Backtracking is Applied in the Algorithm****:

**Color Assignment**:

* 1. For each vertex, the algorithm tries to assign a color from the available num\_colors.

**Recursive Exploration**:

* 1. After assigning a color to the current vertex, the algorithm recursively moves to the next vertex (node + 1).

**Backtracking**:

* 1. If no valid color can be assigned to a vertex, it removes the color from the current vertex (current\_colors[node] = 0) and tries a different color for the previous vertex.

**Termination**:

* 1. The algorithm terminates when all vertices are successfully colored, indicating a valid solution.