Proyecto final.

Manuel Alexander Serna Jaraba - 202259345

Santiago Villa Salazar - 202259527

Universidad del valle – Tuluá

Ingeniería en Sistemas

Programación funcional y concurrente.

Profesor: Carlos Andres Delgado Saavedra

18 de Diciembre de 2023

Justificación de las Funciones:

Función 'generarSubcadenas' y 'ingenua'

La función 'generarSubcadenas' se encarga de producir todas las subcadenas posibles de longitud 'n' a partir de un alfabeto específico. Esta función emplea una estructura de datos que es una secuencia de secuencias de caracteres ('Seq[Seq[Char]]'). La recursión se lleva a cabo a través de la función interna 'subcadenasIter', que acepta dos argumentos: 'subs', que representa las subcadenas generadas hasta ahora, y 'n', que indica la longitud actual de las subcadenas en proceso. Cuando 'n' exceda la longitud deseada, la recursión se detiene y se devuelven las subcadenas generadas hasta ese punto.

En términos de las estructuras de datos utilizadas, se utiliza la función 'flatMap' para combinar las subcadenas existentes con cada carácter del alfabeto, generando así nuevas subcadenas. La función 'map' se utiliza para construir las subcadenas iniciales, cada una compuesta por un solo carácter del alfabeto.

La función 'ingenua' busca la primera subcadena que cumple con un oráculo dado. Se basa en la función 'generarSubcadenas' para obtener todas las posibles subcadenas de longitud 'n-1'. La estructura de datos utilizada es una secuencia de caracteres ('Seq[Char]'). La función 'find' busca la primera subcadena que satisface el oráculo, y la función 'head' devuelve dicha subcadena.

La función 'mejorada' implementa un algoritmo de búsqueda más optimizado. Su objetivo es encontrar subcadenas que satisfacen un oráculo dado. A continuación, se describen las estructuras de datos utilizadas y se analiza por qué la función es correcta.

- **Implementación y Justificación:**

Notación

generarSubcadenas:$C*\times N \rightarrow C*$

ingenua()=find(generarSubcadenas(alfabeto,$n-1$),oraculo)

Descripción de la Función 'mejorada'

mejorada()=encontrarSubcadenasAux(subcadenas,oraculo)[0]

encontrarSubcadenasAux(sck,oraculo)=

$$\begin{cases} \{\}, \text{ si } sck=\{\} \\ c, \text{ si } \exists c \in \text{combinaciones} \mid \text{longitud}(c)=n \\ \text{encontrarSubcadenasAux(combinaciones,oraculo), en otro caso} \end{cases}$$

La función **'encontrarSubcadenasAux'** maneja de manera eficiente la generación y filtrado de subcadenas, utilizando la función 'filter' para seleccionar aquellas que cumplen con el oráculo.

La recursión se detiene cuando la lista de subcadenas (`sck`) está vacía, devolviendo una lista vacía.

Se generan nuevas combinaciones de subcadenas y caracteres del alfabeto, y se filtran según el oráculo.

Si alguna de las combinaciones tiene la longitud requerida (`n`), se devuelve esa combinación.

En caso contrario, la función se llama recursivamente con las nuevas combinaciones.

La función principal llama a la función auxiliar con la lista inicial de subcadenas y devuelve la primera subcadena encontrada que satisface el oráculo.

- **Descripción de la Función 'turbo'**

La función 'turbo' implementa un algoritmo de búsqueda optimizado que encuentra subcadenas que cumplen con un oráculo dado. A continuación, se describen las estructuras de datos utilizadas y se analiza por qué la función es correcta.

- **Implementación y Justificación:**

turbo=

⌈ alfabetoEnForma.filter(oraculo(_)).head, si n=1

⌊ uniones.head, si n es par

⌊ uniones.head,si n es impar

encontrarSubcadenasVelozAux(*sck*,*n*,combinacion_anterior)=

⌈ uniones.filter(oraculo(_)), si *sck*.head.size + 1=n
⌊ *sck*, si *sck*.head.size≥n

⌊ encontrarSubcadenasVelozAux(uniones, n, sck_anterior),*sck*.head.size * 2 > n

⌊ encontrarSubcadenasVelozAux(uniones, n, sck_ anterior), en otros casos

Se utilizan condiciones para determinar el flujo del algoritmo. La lógica se basa en generar y combinar subcadenas de manera estratégica para reducir el espacio de búsqueda.

Si la longitud actual de las subcadenas alcanza `n`, se devuelve la lista de subcadenas. Si la longitud es menor que `n`, se realizan combinaciones y filtrados de subcadenas.

Se utilizan diferentes estrategias de combinación según la longitud de las subcadenas, optimizando así el rendimiento del algoritmo.

La función principal decide cómo proceder según la paridad de `n` y llama a la función auxiliar con las subcadenas correspondientes.

Función filtro_cadenas

- **Descripción:**

Esta función busca determinar si una cadena cumple con ciertos criterios al compararla con una lista de subcadenas de longitud `n`.

- **Lógica:**

La función tiene una función auxiliar interna llamada `incluye` que verifica si una cadena está incluida en una lista de subcadenas.

Si la longitud de la cadena es igual a `n`, se comprueba si la cadena está incluida en la lista de subcadenas.

Si la longitud es diferente, se toma un segmento de longitud `n` de la cadena y se comprueba si está incluido en la lista de subcadenas. Luego, la función se llama recursivamente con el resto de la cadena.

Función turboMejorada

- **Implementación**:

Igual que la función Turbo, solo que con un filtro de verificación de subcadenas, para eliminar las subcadenas que no pertenezcan a la lista anterior, para ‘ahorrar’ intentos del oráculo

- **Descripción:**

Esta función implementa un algoritmo de búsqueda optimizado para encontrar subcadenas que cumplen con un oráculo dado.

- **Lógica:**

La lógica de la función se divide en casos:

Si la longitud de las subcadenas más la unidad es igual a `n`, se combinan y filtran las subcadenas, y se devuelve el resultado.

Si la longitud de las subcadenas es mayor o igual a `n`, se devuelven las subcadenas sin cambios.

Si el doble de la longitud de las subcadenas es mayor que `n`, se combinan y filtran las subcadenas utilizando la lista de subcadenas de la iteración anterior.

En otros casos, se combinan y filtran las subcadenas utilizando la lista de subcadenas actual.

La función principal decide cómo proceder según la cantidad de `n` y llama a la función auxiliar con las subcadenas correspondientes.

Función turboAcelerada

La función turboAcelerada implementa un algoritmo de búsqueda de subcadenas optimizado que aprovecha la paralelización en la reconstrucción de secuencias.

- **Descripción General**

La función turboAcelerada busca subcadenas que cumplen con un oráculo utilizando un enfoque optimizado que involucra la reconstrucción paralela de secuencias.

- **Estructuras de Datos Utilizadas**

La función utiliza la estructura de datos de un árbol de sufijos (Trie) para representar las secuencias de manera eficiente.

Se hace uso de las funciones auxiliares turboAceleradaAux y evaluar_arbol para la manipulación y evaluación del árbol de sufijos.

- **Implementación y Lógica**

La función **turboAcelerada** se implementa de la siguiente manera:

turboAcelerada()=

⌈ head(combinar_dos_listas(sub_cadenas,sub_cadenas).filter(oraculo), si *n* es par

⌊ head(combinar_dos_listas(c1,c2).filter(oraculo)), si *n* es impar

Donde:

arbolDeSufijos:{alfabeto}→Trie es una función que construye el árbol de sufijos inicial a partir de las secuencias de un solo carácter del alfabeto.

cadenas_del_arbol:Trie→{subcadenas} es una función que extrae todas las secuencias almacenadas en el árbol de sufijos.

turboAceleradaAux:Trie×{subcadenas}×{subcadenas}→Trie es una función auxiliar que realiza la reconstrucción de secuencias en paralelo y actualiza el árbol de sufijos.

filtro_cadenas:{subcadenas}×{subcadenas}×Int→Boolean es una función que filtra las cadenas según ciertos criterios.

La función principal turboAcelerada inicia con la creación del árbol de sufijos inicial, generado a partir de las subcadenas de longitud 1 del alfabeto.

La función hace uso de la recursión y patrones de concordancia para actualizar el árbol de sufijos de manera eficiente.

Se emplea la función turboAceleradaAux para reconstruir secuencias de manera paralela, y la función evaluar_arbol para evaluar y actualizar el árbol según ciertos criterios.

La función se divide en casos según la paridad de n y realiza combinaciones estratégicas para optimizar la búsqueda.

a. Función raíz:

Devuelve el carácter de la raíz de un Trie.

Utiliza patrones de concordancia para manejar tanto nodos Nodo como nodos Hoja.

b. Función cabezas:

Devuelve una secuencia de caracteres correspondientes a los hijos directos de un Trie.

Utiliza patrones de concordancia para manejar nodos Nodo y nodos Hoja.

c. Función agregar:

Agrega una cadena al árbol de sufijos de manera eficiente y correcta.

Maneja casos donde la cadena ya existe, se deben crear nuevos nodos o simplemente actualizar la marcación de un nodo.

d. Función agregar_secuencias

Agrega una secuencia de cadenas al árbol de sufijos utilizando la función agregar.

Utiliza recursión para agregar cada secuencia al árbol.

e. Función arbolDeSufijos

Construye un árbol de sufijos a partir de una lista de secuencias de cadenas.

Utiliza la función agregar_secuencias para agregar cada secuencia al árbol inicial.

f. Función pertenece

La función pertenece, no se utiliza en el contexto actual del código.

Su lógica de verificación de pertenencia ha sido incorporada directamente en otras partes del código, específicamente en la función filtro_cadenas de turboMejorada, ya que se tienen todas las secuencias en una variable se aprovecha para realizar el filtro con esta variable.

Análisis:

Tamaño	Pruebas	Ingenua	IngenuaPar	Mejorada	MejoradaPar	Turbo	TurboPar
2	Tiempo	36,3095	9,7482	5,333	4,2944	2,3015	14,4436
3	Tiempo	1,9663	2,6116	0,5622	0,4007	2,0667	1,8918
4	Tiempo	5,0651	2,8457	0,6704	0,8752	2,0349	2,9131
5	Tiempo	10,3678	3,3636	0,327	0,7234	2,47	1,5105
6	Tiempo	9,7728	6,6397	0,4829	0,8078	0,3371	0,4924
7	Tiempo	41,7699	18,5876	0,6304	0,956	4,3085	0,5602
8	Tiempo	70,3383	26,583	0,2731	0,7061	0,2072	0,5209
9	Tiempo	261,5879	101,7015	0,4232	2,8104	4,7028	0,8283
10	Tiempo	1053,1885	446,9003	0,2261	2,5049	0,3599	0,6316
11	Tiempo	4433,1857	1249,0443	0,3037	2,1789	1,1425	0,8499
Totales		5923,5518	1868,0255	9,232	16,2578	19,9311	24,6423

TurboMejorada	TurboMejoradaPar	TurboAcelerada	TurboAceleradaPar
1,3685	2,2822	9,4018	2,0063
2,9629	1,8843	1,9826	4,0739
0,3336	3,8196	0,3144	2,4653
2,3404	2,9009	7,8325	4,527
1,8279	0,7342	0,6468	0,7904
0,9053	1,3739	2,8372	1,3789
0,388	1,8478	1,8231	1,251
4,4896	11,2496	5,1271	3,0191
0,4496	1,1493	0,8282	3,449
2,3966	4,1248	1,7769	2,8878
17,4624	31,3666	32,5706	25,8487

Análisis del Paralelismo en las Funciones:

Análisis de la Función IngenuaPar:

Estructura de la Función:

- La función divide el conjunto de subcadenas (**resultado**) en dos partes, **c1** y **c2**, utilizando la función **parallel**.
- Luego, realiza la búsqueda del oráculo en ambas mitades en paralelo utilizando la función **find**.
- Finalmente, selecciona la primera secuencia encontrada entre ambas mitades.

Razones de la Mejora:

- La mejora observada puede atribuirse a la ejecución en paralelo de la búsqueda del oráculo en dos conjuntos de datos separados.
- En comparación con la versión secuencial (**Ingenua**), esta versión paralela tiene el potencial de reducir el tiempo total de ejecución, especialmente cuando la operación de búsqueda del oráculo es intensiva en cómputo.

Análisis de la Función Mejorada:

Estructura de la Función:

- La función comienza generando una lista de subcadenas de longitud 1 (**alfabetoEnForma**), donde cada subcadena es un solo carácter del alfabeto.
- Luego, se llama a la función auxiliar **encontrarSecuenciasParAux** con esta lista de subcadenas.

La función **encontrarSecuenciasParAux** opera en dos fases:

1. **Fase 1:** Llamada a la función **encontrarSecuenciasMSubs** en paralelo en dos mitades de la lista original. Cada llamada combina las subcadenas con un carácter del alfabeto y filtra aquellas que cumplen con el oráculo.
2. **Fase 2:** Combina los resultados paralelos (**c1** y **c2**) y filtra las secuencias vacías. Si hay alguna secuencia de longitud **n**, devuelve ese conjunto de secuencias. De lo contrario, realiza una llamada recursiva.

Razones de la Mejora:

- La mejora se basa en la paralelización de la búsqueda de secuencias optimizadas en dos conjuntos de datos separados.
- La función trabaja con secuencias más pequeñas y utiliza paralelismo para evaluar diferentes combinaciones simultáneamente, lo que podría conducir a un rendimiento mejorado.

Datos Irregulares:

- Los datos de rendimiento son irregulares y podrían deberse a la naturaleza de las operaciones paralelas, la carga de trabajo, o el tamaño del conjunto de datos en cada ejecución.
- También puede deberse a que el proceso de combinación sea ligero o el proceso de unión es pesado o a una suma de ambos.

Análisis de la Funciones Turbo:

Paralelismo en Turbo:

- La función **Turbo** y **TurboMejorada** utiliza la paralelización mediante el uso de tareas (**task**) en las fases que requieren búsqueda y combinación de subcadenas.

Análisis Comparativo:

- La versión paralela puede incluir complejidades adicionales, y si la tarea es lo suficientemente simple o no tiene suficiente trabajo para justificar la paralelización, podría resultar más eficiente en su versión secuencial.

- La diferencia de turbo no se nota tanto como las demás funciones, ya que si quitamos el primer caso, la función paralela es más constante y mas rapida en la mayoría de casos, sobretodo en los numeros impares, la funcion paralela es perfecta para esos casos, aunque el umbral debería ser mayor la TurboMejorada ya que en este umbral de datos no se a logrado mejorar el rendimiento lo que quiere decir que la versión secuencial sigue siendo muy eficiente y crear un nuevo hilo solo entorpece el buen algoritmo construido.

- La función mejorada también sufre de lo mismo pero alreves, presenta mejoras con las cadenas pares, aunque analizando más a la versión secuencial se puede observar que esta función es bastante más lenta en cadenas impares, por lo que puede parecer que la función turbo ‘intenta’ solventar este problema sin mucho éxito.

- La función turboAceleradaPar al utilizar Parvector, si quitamos el primer caso de la turboacelerada podemos ver que su rendimiento es muy similar lo que sugiere que la implementación le Parvector es eficiente si se hiciera un cambio del código más profundo, ya que si nodo implementace Parvector y no se hicieran transformaciones, la función TurboAceleradaPar sería considerablemente mejor

Conclusiones Generales:

La implementación y análisis de algoritmos para la búsqueda de subcadenas revela aspectos interesantes sobre el rendimiento y la eficacia de diferentes enfoques. A continuación, se resumen las observaciones clave y conclusiones derivadas de las distintas implementaciones y sus evaluaciones.

Eficiencia de la Versión Ingenua:

- La versión ingenua, que busca todas las subcadenas y selecciona la primera que satisface el oráculo, es sencilla, pero puede ser costosa en términos computacionales, especialmente para tamaños de datos más grandes. La paralelización de esta versión no siempre ofrece mejoras significativas debido a posibles cargas ligeras de trabajo y al overhead asociado con la gestión de tareas.

Mejoras mediante la Versión Mejorada:

- La versión mejorada introduce una estrategia más eficiente al generar subcadenas de manera incremental, evitando la generación completa de todas las subcadenas posibles. La paralelización en este caso demuestra ser beneficiosa al dividir la tarea en partes más pequeñas y procesarlas concurrentemente.

Rendimiento de la Versión Turbo:

- La versión turbo, al forzar la longitud de la subcadena a ser impar y realizar combinaciones eficientes, muestra mejoras sustanciales en comparación con las implementaciones anteriores. Y aunque la versión paralela puede usarse sin problemas en los casos impares, la mejor función sería aquella que combine lo mejor de las dos funciones.

Importancia del Saber Cuándo Paralelizar:

- En estos casos es importante paralelizar cuando los datos son grandes, y cuando el algoritmo pueda paralelizar de una manera más 'natural', ¿A qué me refiero?, en el caso de la función turboParalela, cuando se fuerza a ser impar a un número, los tiempos de ejecución son más lentos, por lo tanto, hay que calcular muy bien cuando se empieza a ganar rendimiento y cuando se puede aplicar de manera natural a un algoritmo.

Balance entre Paralelización y Complejidad:

- La paralelización puede no ser siempre beneficiosa, especialmente para tareas más simples o conjuntos de datos pequeños. La complejidad adicional y el overhead introducido pueden superar los beneficios potenciales de la ejecución paralela.