Informe de Tarea 3:

DnaFinder

Asignatura: Estructuras de Datos

Profesor: Christian Vásquez Rebolledo

Fecha: 17 de noviembre de 2025

Integrantes:

- Nicolas Alvarez
- Carlos Bedon
- Matías Espinosa
- Christofer Gutiérrez
- Sebastian Ramirez

Introducción

La Bioinformática enfrenta el desafío constante de procesar y analizar volúmenes masivos de datos genómicos. Una de las tareas más críticas y computacionalmente intensivas es la búsqueda de patrones, la cual es fundamental para la identificación de genes, el alineamiento de secuencias y el análisis de secuencias repetitivas. El problema central es cómo localizar eficientemente todas las instancias de un gen corto (una sub-secuencia de tamaño *m*) dentro de una secuencia S de tamaño *n*, que puede ser inmensa.

Este proyecto implementa una solución de alta eficiencia para este problema. En lugar de utilizar búsquedas de fuerza bruta (cuya complejidad (n \ m) es prohibitiva para genomas reales), nuestro sistema, "DnaFinder", adopta la metodología de un árbol trie 4-ario de profundidad m^3. Esta estructura de datos especializada, optimizada para el alfabeto de cuatro caracteres ('A', 'C', 'G', 'T') del ADN, permite indexar *todos* los n-m genes presentes en la secuencia S. Al hacerlo, transformamos el problema de búsqueda en un problema de acceso directo, permitiendo que las consultas de búsqueda, frecuencia y posición se resuelvan en tiempo óptimo.

El presente informe detalla las decisiones de diseño, la arquitectura modular en C y los desafíos de implementación que se abordaron para construir este sistema de análisis genético.

Objetivos

Los objetivos de esta tarea, definidos en el pliego, fueron:

- Implementar y manipular estructuras de datos abstractas como árboles y listas enlazadas para la gestión de datos.
- Desarrollar habilidades en programación C, centrándose en el manejo de memoria, punteros y eficiencia algorítmica.
- Implementar un sistema de búsqueda de patrones utilizando árboles tries.

Metodología y Estructuras de Datos

El sistema se articula en torno a dos estructuras de datos principales, definidas en los archivos trie.h y list.h, que trabajan en conjunto.

3.1. Árbol Trie 4-ario (TrieNode)

Es la estructura central del proyecto. Cada nodo (TrieNode) representa un prefijo de un gen.

- **Nodos Internos:** Poseen cuatro punteros (hijoA, hijoC, hijoG, hijoT), uno por cada base nucleotídica . Su puntero a la lista de posiciones (positions) se mantiene en NULL. • **Nodos Hoja:** Se encuentran a la profundidad *m*. Sus cuatro punteros de hijos son NULL, pero su puntero positions apunta a una instancia de List .

La estructura definida en trie.h es la siguiente:

Código:
```
// Estructura de un nodo del trie (incs/trie.h)
typedef struct TrieNode {
 struct TrieNode *hijoA;
 struct TrieNode *hijoC;
 struct TrieNode *hijoG;
 struct TrieNode *hijoT;
 List *positions; // se usará en las hojas
} TrieNode;
```

3.2. Listas Enlazadas (List / Node)

Para cada nodo hoja en el trie, se almacena una lista enlazada que registra todas las posiciones de inicio (índices) donde ese gen particular aparece en la secuencia S.

La estructura List (definida en list.h) contiene un puntero head al primer nodo y un contador count, que resultó crucial para implementar eficientemente los comandos max y min.

Código:
```
// Estructura de la lista enlazada (incs/list.h)
typedef struct Node {
 int pos;
 struct Node *next;
} Node;

typedef struct List {
 Node *head;
 int count; // Contador de posiciones
} List;
```

3.3. Proceso de Carga en Dos Fases

El sistema funciona siguiendo estrictamente el proceso de dos fases requerido por la especificación:

1. **Fase 1**: Creación (cmd_start <m>) El comando cmd_start invoca a la función create_trie(m). Esta función genera el "esqueleto" completo del árbol con profundidad m. Como se detalla en la sección de "Desafíos", esta función crea recursivamente todos los nodos. Cuando la recursión alcanza la profundidad 0 (una hoja), se llama a Notes() para inicializar una lista vacía en ese nodo hoja, cumpliendo con el requisito. Los nodos internos, por el contrario, dejan su puntero positions en NULL.
2. **Fase 2**: Poblado (cmd_read <file>) El comando cmd_read lee la secuencia S del archivo (usando read_sequence de file_utils.c). Luego, itera sobre la secuencia S, desde i = 0 hasta n - m. En cada iteración, se extrae el gen de tamaño *m* que comienza en i. Se desciende por el árbol (usando get_leaf_node) y, al llegar al nodo hoja correspondiente, se añade la posición i a su lista (usando add_position).

Diseño y Modularidad del Software

El proyecto se adhirió a las normativas de modularidad, dividiendo el código en unidades lógicas y reutilizables, como se evidencia en el makefile.

- **main.c**: Punto de entrada del programa. Contiene el bucle principal que lee la entrada del usuario y la pasa al procesador de comandos.
- **bio.c / bio.h**: Gestiona el estado general del sistema (BioSystem), conteniendo la raíz del árbol (root) y el largo del gen (gene_length).
- **trie.c / trie.h**: Implementa la lógica fundamental del árbol: create_trie, free_trie, y get_leaf_node.
- **list.c / list.h**: Implementa la lógica de la lista enlazada: Notes, add_position, y free_list.
- **commands.c / commands.h**: Contiene la lógica de aplicación para cada comando (cmd_start, cmd_read, cmd_search, cmd_all, cmd_max, cmd_min, cmd_exit). • file_utils.c / file_utils.h: Abstrae las operaciones de E/S y validación de archivos (read_sequence, validate_sequence).

Desafíos y Decisiones de Implementación

Durante el desarrollo, se abordaron varios desafíos clave los cuales fueron fundamentales para poder tener una mejor comprensión del verdadero funcionamiento de la lógica del proyecto

Desafío 1: Lógica Fundamental de Creación del Árbol

- **Problema**: El readme inicial identificó como desafío el "Manejo en la lógica fundamental de la creación del árbol". El requisito de "Generar el árbol (con las listas vacías)" antes de leer la secuencia S^{20} requería una implementación cuidadosa.
- **Decisión/Solución**: Se resolvió con una implementación recursiva en create_trie(int depth) (en trie.c).
    - **Caso Base** (depth == 0): Es un nodo hoja. Se crea el nodo y se inicializa su lista (leafNode->positions = create_list();).
    - **Caso Recursivo** (depth > 0): Es un nodo interno. Se crea el nodo, positions se deja en NULL, y se llama recursivamente a create_trie(depth - 1) para los 4 hijos (hijoA, hijoC, etc.).

      Esta decisión separó limpiamente la creación de la estructura de la inserción de datos, simplificando el resto del código.

Desafío 2: Recolección de Datos para all, max y min

- **Problema:** Los comandos cmd_all, cmd_max y cmd_min requieren visitar *todos* los genes que *existen* en la secuencia S, no todos los 4^m genes posibles. • **Decisión/Solución:** Se implementó una función auxiliar collect_genes_recursive (en commands.c). Esta función realiza un recorrido en profundidad (DFS) sobre el árbol. Al llegar a un nodo hoja (current_depth == depth), comprueba si el gen existe (node->positions->count > 0). Si existe, almacena el gen, su frecuencia (list->count) y el puntero a su lista en un arreglo temporal GeneInfo.
    - cmd_all imprime este arreglo completo.
    - cmd_max y cmd_min realizan dos pasadas sobre este arreglo: una para encontrar el valor max_frequency/min_frequency y otra para imprimir todos los genes que coincidan con ese valor.

Desafío 3: Orden de Impresión de Posiciones

- **Problema:** La función add_position agrega posiciones al inicio de la lista (LIFO) por eficiencia (O(1)). Sin embargo, el flujo de ejemplo muestra las posiciones en orden ascendente (FIFO), ej: "AA 4 7". Una implementación ingenua imprimiría "AA 7 4".
- **Decisión/Solución:** En las implementaciones de cmd_search, cmd_all, cmd_max y cmd_min, las posiciones se copian de la lista enlazada a un arreglo temporal. Este arreglo luego se ordena (usando *bubble sort*, que es eficiente para las listas cortas esperadas) y finalmente se imprime, asegurando la conformidad con la salida de ejemplo.

Desafío 4: Gestión de Memoria y Robustez

- **Problema:** El proyecto requiere gestión de memoria sin fugas [23]y manejo robusto de errores.
- **Decisión/Solución:**
  - **Memoria:** Se implementó free_trie (en trie.c) usando un recorrido en post orden, que primero libera recursivamente a los hijos, luego libera la lista de posiciones (free_list) si existe, y finalmente libera el nodo actual (free(root)). El comando cmd_exit llama a free_bio_system, que inicia esta cascada de liberación.
  - **Robustez:** process_command (en commands.c) comprueba si sys->root es NULL antes de ejecutar comandos como read o search, evitando segfaults. Además, cmd_read y cmd_search validan la entrada del usuario (validate_sequence y strlen(gene) == sys->gene_length)

Pruebas y Ejemplo de Uso

El sistema se probó utilizando el archivo adn.txt proporcionado (S = "TACTAAGAAGC") con un tamaño de gen m=2. El flujo de ejecución y los resultados coinciden exactamente con los proporcionados en la especificación de la tarea.

Bash:

```
>bio start 2
Tree created with height: 2
>bio read data/adn.txt
Sequence S read from file adn.txt
>bio search CC
-1
>bio search AA
4 7
>bio max
AA 4 7
AG 5 8
TA 0 3
>bio min
AC 1
CT 2
GA 6
GC 9
>bio all
AA 4 7
AC 1
AG 5 8
CT 2
GA 6
GC 9
TA 0 3
>bio exit
Clearing cache and exiting...
```

(Salida generada por la implementación actual, coincidente con el flujo de ejemplo)

Conclusión

El desarrollo del proyecto "DnaFinder" ha culminado con la implementación exitosa de un sistema robusto y eficiente para el análisis de secuencias genéticas, cumpliendo con todos los objetivos propuestos. El sistema no solo es funcional, sino que valida empíricamente la idoneidad de la estructura de datos trie 4-aria como solución superior al problema de búsqueda de patrones en alfabetos fijos.

El resultado técnico más significativo es la eficiencia algorítmica lograda: una vez que el índice es construido por cmd_read (una operación de que se realiza una sola vez), la subsecuente búsqueda de *cualquier* gen (cmd_search) se ejecuta en tiempo. Esta complejidad es independiente del tamaño *n* de la secuencia S, lo cual representa una ventaja exponencial sobre las alternativas de fuerza bruta y es crucial para la escalabilidad en aplicaciones bioinformáticas reales.

Más allá de la funcionalidad, el proyecto sirvió como un ejercicio intensivo en habilidades de programación avanzada en C, particularmente en la gestión precisa de la memoria dinámica. Los principales desafíos no fueron conceptuales, sino de implementación: asegurar la correcta inicialización recursiva del árbol (create_trie), garantizar la ausencia de fugas de memoria (a través de free_trie y free_list), y manejar la recolección y ordenamiento de datos para los comandos max y min.

Como trabajo futuro, el sistema podría optimizarse para manejar valores de *m* extremadamente grandes. Para la creación de hojas en cmd_start se vuelve impracticable. Una futura versión podría implementar un "trie disperso" (sparse trie) que solo cree nodos "bajo demanda" durante
la fase cmd_read, reduciendo drásticamente la huella de memoria a cambio de una lógica de inserción más compleja.