ALGORÍTMICA II
PROYECTO PRIMER PARCIAL
SISTEMA DE ANÁLISIS DE CÓDIGO Y DOCUMENTACIÓN INTELIGENTE

Docente: Ing. Marcelo Bernardo López de La Rosa
Materia: Algorítmica II
Estudiantes: Jassir Mijail Guzman Arnez
Código: 68987

Cochabamba - Bolivia
22/08/25

Índice

1. Introducción

En el campo del desarrollo de software y la ingeniería, la eficiencia en el acceso a la información y la capacidad de analizar código fuente son pilares fundamentales para la productividad y la calidad. La documentación técnica, a menudo densa y extensa, requiere mecanismos de búsqueda que vayan más allá de simples coincidencias textuales, mientras que la evaluación de la originalidad y la reutilización de código demanda herramientas que puedan comprender la lógica subyacente en lugar de solo la sintaxis superficial.

Este proyecto aborda estas dos problemáticas interrelacionadas a través del diseño e implementación de un Sistema de Análisis de Código y Documentación Inteligente. El sistema se enfoca en dos objetivos principales. Primero, optimizar la búsqueda de términos en documentación técnica mediante la construcción de un índice basado en un Árbol de Búsqueda Binaria Óptimo (OBST). A diferencia de las estructuras de búsqueda tradicionales, el OBST utiliza el conocimiento *a priori* de las frecuencias de búsqueda de los términos para minimizar el costo promedio de acceso, una solución ideal para la consulta de vocabularios estáticos y bien definidos.

Segundo, el sistema introduce un mecanismo para la detección de similitud entre fragmentos de código, basado en una versión mejorada del algoritmo de la Secuencia Común Más Larga (LCS) Ponderada. Al tokenizar el código fuente y asignar pesos a los tokens según su importancia sintáctica (palabras clave, operadores, etc.), este enfoque trasciende las diferencias estilísticas y de nomenclatura para ofrecer una medida cuantitativa de la similitud algorítmica y estructural.

A lo largo de este documento, se presenta el modelado matemático, el diseño algorítmico, la implementación y la validación empírica de ambas soluciones. A través de benchmarks con datos reales, se demostrará el compromiso fundamental entre el costo de construcción y la eficiencia de búsqueda del OBST, y se validará la precisión del LCS Ponderado para la comparación inteligente de código.

2. Contexto del Problema

En el contexto de las plataformas de desarrollo colaborativo, como GitHub y GitLab, se observa una creciente necesidad de optimizar el análisis de código y documentación técnica. Estas plataformas albergan grandes cantidades de información y código, lo que hace que las búsquedas y la gestión de recursos sean cada vez más desafiantes. Además, la necesidad de detectar similitudes entre fragmentos de código, así como mejorar la eficiencia en la búsqueda de términos técnicos, se vuelve crítica para mantener la productividad y calidad del software.

2.1. Problemática Principal

Las plataformas mencionadas requieren sistemas inteligentes que permitan realizar dos tareas clave:

2.1.1. Optimización de la Búsqueda de Términos Técnicos en Documentación:

Los usuarios, a menudo, buscan términos técnicos específicos dentro de la documentación. Sin embargo, realizar estas búsquedas puede ser ineficiente si no se cuenta con un sistema adecuado que minimice el número de comparaciones necesarias. Este problema se aborda mediante la construcción de un Árbol de Búsqueda Binaria Óptimo (OBST), el cual está diseñado para organizar los términos de forma que se minimice el costo esperado de búsqueda. Esto se logra a través de la asignación de probabilidades de éxito y fallo a las búsquedas, lo que permite optimizar la estructura del árbol para mejorar el rendimiento de las búsquedas.

2.1.2. Detección de Similitud entre Fragmentos de Código:

La capacidad de detectar similitudes entre fragmentos de código es esencial en plataformas de desarrollo colaborativo, tanto para mejorar la calidad del código como para evitar la duplicación. Este problema se resuelve mediante el algoritmo de Subsecuencia Común Más Larga (LCS), modificado para considerar la ponderación de diferentes tipos de tokens (como palabras clave, identificadores, operadores, etc.). Además, se debe establecer un umbral de similitud para identificar cuándo dos

fragmentos de código son lo suficientemente similares como para ser considerados una coincidencia. Este sistema de LCS ponderada permite identificar similitudes sutiles entre diferentes versiones de un mismo código, incluso si se han realizado cambios en los nombres de las variables o en la estructura.

2.2. Relevancia y Aplicación

Ambos problemas están en el núcleo de la eficiencia en el desarrollo de software y la gestión de documentación. Un Índice de Documentación Óptimo mejora la experiencia del usuario al reducir el tiempo de búsqueda en grandes volúmenes de documentación técnica, lo cual es fundamental en plataformas de desarrollo colaborativo. Por otro lado, un Detector de Similitud de Código optimiza la revisión de código, facilita la refactorización y puede ayudar a prevenir problemas de código duplicado o mal implementado.

El proyecto busca proporcionar soluciones eficientes a estos problemas, utilizando técnicas de programación dinámica como la programación de Árboles de Búsqueda Binaria Óptimos y el cálculo de la Subsecuencia Común Más Larga ponderada. Estas técnicas permiten reducir los costos de operación y mejorar la precisión de las búsquedas y comparaciones en sistemas de desarrollo de software.

3. Modelado Matemático y Algoritmos Propuestos

3.1. Índice de Documentación Óptimo (OBST)

3.1.1. Modelo Matemático

El objetivo de este problema es construir un Árbol de Búsqueda Binaria Óptimo (OBST) para términos técnicos en la documentación, minimizando el costo esperado de búsqueda, el cual está determinado por las probabilidades de búsqueda exitosa p_i y las probabilidades de búsqueda fallida q_i entre términos consecutivos. Dado un conjunto de n términos ordenados lexicográficamente, y considerando las probabilidades de búsqueda de cada término, el objetivo es construir un árbol que minimice el número total de comparaciones al realizar una búsqueda.

El costo esperado de búsqueda C se calcula a partir de los costos de las subárboles y el costo de las comparaciones para cada término. El cálculo de los costos de los subárboles se realiza de manera recursiva, y el costo mínimo se obtiene eligiendo la raíz de cada subárbol que minimice el costo esperado.

3.1.2. Recurrencia

Sea $E[i, j]$ el costo mínimo de un subárbol que contiene las claves de k_i a k_j (donde $i \leq j$). La fórmula recursiva para este costo es:

$$E[i, j] = r = iminj(E[i, r - 1] + E[r + 1, j] + W[i, j])$$

Donde $W[i, j]$ es el peso total de las probabilidades en el subárbol definido por k_i a k_j:

$$W[i, j] = t = i\sum jpt + t = i - 1\sum jqt$$

3.1.3. Pseudocódigo

El siguiente pseudocódigo detalla cómo calcular el árbol de búsqueda binaria óptimo:

```
function OPTIMAL_BST(keys[1..n], p[1..n], q[0..n]):
    for i = 1..n+1:
        E[i][i-1] = q[i-1]
        W[i][i-1] = q[i-1]
    for i = 1..n:
        W[i][i] = q[i-1] + p[i] + q[i]

    for len = 1..n:
        for i = 1..n-len+1:
            j = i + len - 1
            W[i][j] = W[i][j-1] + p[j] + q[j]     // prefijos para
sumar rápido
            E[i][j] = +INF
            for r = i..j:
                cost = E[i][r-1] + E[r+1][j] + W[i][j]
                if cost < E[i][j]:
                    E[i][j] = cost
                    ROOT[i][j] = r
```

```
    return (E[1][n], ROOT)
```

3.1.4. Complejidad

La complejidad de este algoritmo es $O(n^3)$, ya que hay tres bucles anidados: uno para la longitud de las subcadenas, otro para el inicio de cada subcadena y un tercero para probar todas las posibles raíces dentro de esa subcadena. Esto se puede mejorar a $O(n^2)$ utilizando la monotonía de raíces de Knuth.

3.1.5. Reconstrucción del Árbol Óptimo

El árbol se reconstruye a partir de la tabla $ROOT[i, j]$, que contiene la raíz de cada subárbol óptimo. Esto se puede hacer de forma recursiva:

```
function RECONSTRUIR(ROOT, i, j):
    if i > j: return NIL
    r = ROOT[i][j]
    nodo = Node(keys[r])
    nodo.left  = RECONSTRUIR(ROOT, i, r-1)
    nodo.right = RECONSTRUIR(ROOT, r+1, j)
    return nodo
```

3.2. Detector de Similitud de Código (LCS ponderada)

3.2.1. Modelo Matemático

El problema de la similitud de código se basa en encontrar la subsecuencia común más larga (LCS) entre dos fragmentos de código. Se ignoran los espacios en blanco y los comentarios, y se asignan diferentes pesos a los diferentes tipos de tokens (por ejemplo, palabras clave, identificadores, operadores, literales). La similitud entre dos fragmentos de código se calcula mediante una versión ponderada de LCS. El cálculo de la LCS ponderada utiliza una matriz de programación dinámica $C[i, j]$, donde i y j representan las posiciones en las secuencias de tokens X y Y, respectivamente. El valor de $C[i, j]$ se actualiza según el siguiente principio:

- Si los tokens coinciden, el valor de $C[i,j]$ se incrementa por el peso asociado a ese tipo de token.
- Si no coinciden, se toma el máximo entre las dos posibilidades anteriores.

3.2.2. Pseudocódigo

El siguiente pseudocódigo implementa el cálculo de la LCS ponderada:

```
function LCS_WEIGHTED(X[1..m], Y[1..n], weight_of):
    // X, Y = tokens; weight_of(token) retorna peso por tipo
    C = matrix (m+1) x (n+1) filled with 0
    P = matrix (m+1) x (n+1)  // para backtracking: '↖', '↑', '←'

    for i = 1..m:
        for j = 1..n:
            if EQUIV(X[i], Y[j]):            // igualdad
léxica/normalizada
                w = weight_of(X[i])
                if C[i-1][j-1] + w >= C[i-1][j] and C[i-1][j-1] + w >=
C[i][j-1]:
                    C[i][j] = C[i-1][j-1] + w ; P[i][j] = '↖'
                else if C[i-1][j] >= C[i][j-1]:
                    C[i][j] = C[i-1][j]       ; P[i][j] = '↑'
                else:
                    C[i][j] = C[i][j-1]       ; P[i][j] = '←'
            else:
                if C[i-1][j] >= C[i][j-1]:    C[i][j] = C[i-1][j] ;
P[i][j]='↑'
                else:                          C[i][j] = C[i][j-1] ;
P[i][j]='←'

    // reconstrucción
    L = empty list
    i=m; j=n
    while i>0 and j>0:
        if P[i][j]=='↖': prepend L, X[i]; i--; j--
        else if P[i][j]=='↑': i--
        else: j--
```

```
    totalX = sum(weight_of(x) for x in X)
    totalY = sum(weight_of(y) for y in Y)
    score  = 2*C[m][n] / (totalX + totalY)
    return (C[m][n], reverse(L), score)
```

3.2.3. Complejidad

La complejidad temporal de este algoritmo es $O(mn)$, donde m y n son las longitudes de las dos secuencias de código. La complejidad espacial es también $O(mn)$ si se requiere reconstruir la subsecuencia, o $O(min(m,n))$ si solo se necesita el valor del score.

3.2.4. Tokenizador de Código

El tokenizador es responsable de convertir el código fuente en una secuencia de tokens clasificados que serán utilizados en el cálculo de la Subsecuencia Común Más Larga (LCS). El proceso se realiza en los siguientes pasos:

1. **Eliminación de Comentarios y Espacios**: Se eliminan los comentarios y los espacios en blanco innecesarios para centrarse solo en los elementos del código que afectan su funcionamiento.
2. **Separación en Tokens**: El código se divide en tokens usando delimitadores comunes como identificadores, palabras clave, operadores, literales y símbolos.
3. **Clasificación de Tokens**: Cada token se clasifica según su tipo (por ejemplo, keyword, identifier, operator, literal, punct).
4. **Normalización**: Los identificadores pueden ser normalizados para mejorar la detección de similitudes entre fragmentos de código que utilizan nombres diferentes pero que realizan la misma tarea.

4. Implementación

4.1. Implementación General del Sistema

4.1.1. Descripción del Sistema:

El sistema está diseñado para gestionar proyectos y permitir la ejecución de análisis basados en dos algoritmos clave: Árbol de Búsqueda Binaria Óptimo (OBST) y Subsecuencia Común Más Larga Ponderada (LCS). Los proyectos contienen archivos de código y documentación, y el sistema realiza análisis sobre estos archivos.

- **Gestión de Proyectos**:
    - Los proyectos se registran, se almacenan en un archivo JSON (`proyectos.json`) y se pueden listar o eliminar en cualquier momento.
    - Un proyecto está compuesto por:
        - **Nombre** del proyecto.
        - **Ruta** al directorio que contiene los archivos de código.
        - **Ruta** al archivo de documentación (en formato PDF).
- **Interfaz de Usuario**:
    - La interfaz del sistema es una **interfaz gráfica de usuario (GUI)** intuitiva, implementada con la biblioteca `tkinter`. A diferencia de una interfaz de línea de comandos, la GUI utiliza botones, menús desplegables y selectores de archivos, lo que simplifica la interacción y reduce la posibilidad de errores por parte del usuario.

4.1.2. Flujo del Sistema:

El flujo de trabajo está diseñado para ser directo y visual, guiando al usuario a través de los siguientes menús:

- **Menú Principal:** La ventana principal presenta cuatro botones que dirigen al usuario a las funciones clave del sistema:

    - **Gestionar Proyectos:** Navega a un submenú para registrar, listar o eliminar proyectos.
    - **Analizar un Proyecto Existente:** Permite seleccionar un proyecto registrado y elegir entre los análisis disponibles.
    - **Herramientas Individuales:** Ofrece acceso directo a las funcionalidades de OBST y LCS sin estar ligadas a un proyecto.
    - **Salir:** Cierra la aplicación.
- **Gestión de Proyectos:** En esta vista, el usuario puede:
    - **Registrar:** Se utilizan cuadros de texto y selectores de archivos para ingresar el nombre y las rutas del proyecto.
    - **Listar:** Los proyectos existentes se muestran en una lista clara.
    - **Eliminar:** Un menú desplegable permite seleccionar un proyecto de la lista para su eliminación.
- **Análisis de Proyecto:** Tras seleccionar un proyecto de una lista, el usuario puede:
    - **Analizar la Documentación:** Ejecuta el algoritmo OBST sobre el archivo de documentación asociado al proyecto.
    - **Comparar Archivos de Código:** Abre un selector de archivos para que el usuario elija los dos archivos de código a comparar, ya sean del proyecto o de cualquier otra ubicación.
- **Herramientas Individuales:** Este menú autónomo ofrece una funcionalidad similar a los análisis de proyecto, pero con la flexibilidad de trabajar con archivos externos.
    - **Análisis OBST:** El usuario puede seleccionar cualquier archivo de documentación en formato PDF para generar su árbol de búsqueda óptimo.
    - **Comparación LCS:** Permite comparar dos archivos de código elegidos de forma independiente.

4.2. Implementación del Algoritmo OBST

4.2.1. Descripción del Algoritmo OBST:

El algoritmo Árbol de Búsqueda Binaria Óptimo (OBST) tiene como objetivo optimizar las búsquedas de términos en un conjunto de claves ordenadas. Dado un conjunto de claves con probabilidades de búsqueda exitosa p_i y probabilidades de búsqueda fallida q_1, el algoritmo construye un árbol binario que minimiza el costo esperado de búsqueda.

4.2.2. Implementación Detallada:

1. **Estructuras de Datos**:
    - **Tabla de Costos** $E[i][j]$: Almacena el costo mínimo para un subárbol que contiene las claves desde k_i hasta k_j.
    - **Tabla de Raíces** $ROOT[i][j]$: Almacena la raíz óptima de cada subárbol.
    - **Tabla de Pesos** $W[i][j]$: Almacena la suma de las probabilidades de éxito y fallo para cada subárbol.
2. **Proceso**:
    - Inicializamos las tablas **E** y **W**.
    - Para cada subárbol de longitud 1, calculamos su costo y asignamos su raíz.
    - Para subárboles de longitud mayor a 1, calculamos el costo mínimo de cada subárbol y elegimos la raíz que minimiza el costo total.
3. **Reconstrucción del Árbol**:
    - Utilizamos la tabla **ROOT** para reconstruir el árbol óptimo, asignando recursivamente las raíces de los subárboles.
4. **Código** (Implementación Completa en `obst.py`):

    La implementación detallada de OBST ya ha sido proporcionada en el archivo `obst.py`, donde se implementan todas las funciones necesarias para calcular los costos y la reconstrucción del árbol.

4.3. Implementación del Algoritmo LCS Ponderada

4.3.1. Descripción del Algoritmo LCS Ponderada:

Para abordar la comparación de código, se implementó una versión avanzada del algoritmo de Subsecuencia Común Más Larga Ponderada (Weighted LCS). La eficacia de este algoritmo depende críticamente de un pre-procesamiento de datos inteligente, por lo que el núcleo de nuestra mejora se centró en un sistema de tokenización sofisticado.

4.3.2. Implementación Detallada:

1. **Tokenizador Híbrido y Granular**: A diferencia de un tokenizador básico que solo distingue entre categorías genéricas (keyword, identifier), nuestro sistema utiliza un enfoque híbrido. Primero, aprovecha el módulo `tokenize` nativo de Python para una segmentación robusta del código fuente, manejando correctamente la sintaxis del lenguaje. Luego, aplica un sistema de clasificación basado en **expresiones regulares (regex)** para sub-dividir estos tokens en categorías mucho más específicas y significativas, como `CONTROL_FLOW` (if, for, while), `DEF_KEYWORD` (def, class), `ARITHMETIC_OP` (+, *) o `COMPARISON_OP` (==, >=). Esto nos permite entender el propósito estructural de cada elemento del código.
2. **Normalización de Tokens**: Para lograr una comparación abstracta y centrada en la lógica, el sistema implementa un proceso de **normalización**. Los identificadores (nombres de variables y funciones) y los literales (números y cadenas de texto) se mapean a placeholders genéricos (ej. `ID_0`, `ID_1`, `LIT_N_0`). De esta manera, dos funciones que realizan la misma operación pero usan diferentes nombres de variables serán vistas como estructuralmente idénticas. Este paso es crucial para eliminar el ruido superficial y evaluar la verdadera similitud algorítmica.
3. **Algoritmo LCS Ponderado**: El algoritmo `lcs_weighted` opera sobre las secuencias de tokens normalizados. Utiliza una matriz de programación dinámica para encontrar la subsecuencia común con la máxima puntuación. La ponderación se realiza utilizando los pesos asignados a las categorías granulares definidas por el tokenizador. Por ejemplo, una coincidencia en un token de `CONTROL_FLOW` aporta significativamente más al score de similitud que una coincidencia en un literal, reflejando su mayor importancia

estructural. El score final se normaliza para obtener un porcentaje de similitud entre 0 y 1.

Esta arquitectura modular, encapsulada dentro de una clase `CodeComparator`, no solo proporciona resultados más precisos, sino que también ofrece una API interna limpia y extensible para el sistema.

5. Experimentación y Benchmarking

5.1. Objetivo

El objetivo de esta fase experimental es validar empíricamente el rendimiento del Árbol de Búsqueda Binaria Óptimo (OBST) y compararlo con dos de las estructuras de datos auto-balanceables más eficientes y utilizadas en la práctica: el **Árbol AVL** y el **Árbol Rojo-Negro**. La comparación se centrará en dos métricas clave: el **tiempo de construcción** de la estructura y el **costo de búsqueda esperado**, para evaluar el compromiso fundamental (*trade-off*) entre el costo de preparación inicial y la eficiencia operativa a largo plazo.

5.2. Diseño del Benchmark

Para asegurar que la comparación fuese representativa de un caso de uso real, se descartó el uso de datos sintéticos con distribuciones de probabilidad uniformes. En su lugar, se diseñó un benchmark que utiliza datos extraídos de un documento real, garantizando una distribución de frecuencias de términos natural y sesgada, que es el escenario donde el OBST teóricamente debe destacar.

La metodología se estructuró en los siguientes pasos:

1. **Fuente de Datos:** Se utilizó un documento académico en formato PDF (*07-ABB.pdf*) como fuente de texto. Esta elección garantiza la presencia de un vocabulario técnico y una distribución de frecuencias de palabras que sigue la Ley de Zipf, donde pocos términos son muy frecuentes y muchos son infrecuentes.

2. **Extracción y Procesamiento de Claves:** Se implementó un script en Python (`probability_calculator.py`) que automatiza la extracción de datos:
    - Se utiliza la biblioteca `PyPDF2` para extraer el texto crudo del documento.
    - Se emplea la biblioteca `NLTK` para tokenizar el texto, eliminar *stopwords* (palabras comunes sin valor semántico como "el", "de", "que") en español y filtrar los tokens para conservar únicamente palabras válidas.
    - Se determina un conjunto de **5000 términos** como objetivo inicial para formar las claves del benchmark. Del procesamiento del documento resultaron **766 claves únicas** que se utilizaron para la prueba.
3. **Cálculo de Probabilidades:** A partir de las claves extraídas, se calcularon las probabilidades de búsqueda de la siguiente manera:
    - **Probabilidades de Éxito (`p`):** Se calculó la frecuencia de aparición de cada una de las 766 claves en el texto. Estas frecuencias se normalizaron para que su suma representara una **probabilidad de éxito total del 90%** (parámetro `prob_exito_total = 0.90`). Esta suposición simula un sistema en el que la gran mayoría de las búsquedas se realizan sobre términos existentes.
    - **Probabilidades de Fallo (`q`):** El 10% restante de la probabilidad total se distribuyó de manera uniforme entre los `n+1` posibles intervalos de búsqueda fallida.
4. **Métricas de Rendimiento:**
    - **Tiempo de Construcción:** Medido como el tiempo de reloj (wall-clock time) necesario para generar la estructura de datos completa. Para el OBST, corresponde al tiempo de ejecución del algoritmo de programación dinámica. Para los árboles AVL y Rojo-Negro, es el tiempo total requerido para realizar las 766 inserciones de claves (en orden aleatorio para simular un caso promedio).
    - **Costo de Búsqueda Esperado:** Se define como la suma ponderada de las profundidades de todas las claves, calculada mediante la fórmula: Costo=∑i=1npi×profundidad(ki). Esta métrica representa el

número promedio de nodos que se deben visitar para encontrar una clave.

6. Resultados

El benchmark se ejecutó en un entorno controlado, arrojando resultados claros y consistentes que validan las propiedades teóricas de las estructuras de datos analizadas. Los resultados cuantitativos se resumen en la siguiente tabla:

Métrica	Árbol de Búsqueda Óptimo (OBST)	Árbol AVL	Árbol Rojo-Negro
Tiempo de Construcción (s)	**32.015**	0.011	**0.002**
Costo de Búsqueda Esperado	**6.798**	8.029	8.074

6.1. Análisis del Tiempo de Construcción

Los resultados muestran una diferencia abrumadora en los tiempos de construcción. El OBST requirió más de 32 segundos para calcular la estructura óptima, mientras que los árboles AVL y Rojo-Negro se construyeron en meros milisegundos.

Esta disparidad es una consecuencia directa de la complejidad algorítmica de cada método. La construcción del OBST mediante programación dinámica tiene una complejidad de O(n3), lo que implica un costo computacional muy elevado que crece polinómicamente con el número de claves. Por el contrario, la inserción de n elementos en un árbol AVL o Rojo-Negro tiene una complejidad total de O(nlogn), lo que los hace extremadamente eficientes para la construcción dinámica.

Se concluye que el costo de construcción del OBST representa una inversión computacional inicial muy significativa, factible únicamente en escenarios donde el índice es estático y se puede generar de forma "offline".

6.2. Análisis del Costo de Búsqueda Esperado

En esta métrica, la inversión del OBST demuestra su valor. El OBST logró un costo de búsqueda esperado de 6.798, lo que representa una mejora de aproximadamente el 15% en comparación con el costo del Árbol AVL (8.029) y el Árbol Rojo-Negro (8.074).

Esta victoria en eficiencia se debe a que el OBST es la única estructura que utiliza el conocimiento *a priori* de las probabilidades de búsqueda. El algoritmo construye un árbol deliberadamente desbalanceado desde una perspectiva puramente estructural, pero perfectamente optimizado desde una perspectiva probabilística. Las claves más frecuentes, como "árbol", "nodo" o "búsqueda", son estratégicamente ubicadas cerca de la raíz, minimizando el camino de búsqueda para las consultas más comunes.

Por otro lado, los árboles AVL y Rojo-Negro, al ser ajenos a las probabilidades, optimizan únicamente la altura del árbol. Su objetivo es mantener un balance estructural, lo que puede resultar en que una clave muy frecuente termine en un nodo profundo si su valor alfabético así lo determina. El rendimiento casi idéntico entre ambos demuestra su eficacia como estructuras de propósito general, pero también su incapacidad para adaptarse a patrones de acceso específicos.

En un sistema a gran escala que gestiona millones de consultas, una reducción del 15% en el costo promedio de búsqueda se traduce en un ahorro sustancial de recursos computacionales, menor latencia y una mejor experiencia de usuario.

6.3. Experimentación: Detección de Similitud de Código

6.3.1. Objetivo

El objetivo de esta segunda fase experimental fue validar la eficacia del algoritmo de LCS Ponderado para la detección de similitud entre fragmentos de código. El

experimento se diseñó para probar la hipótesis de que el sistema puede identificar una equivalencia lógica y estructural fundamental entre dos implementaciones de un mismo algoritmo, a pesar de diferencias superficiales en la sintaxis, estilo de codificación y nomenclatura.

6.3.2. Diseño y Metodología

Para realizar una prueba controlada y significativa, se siguió la siguiente metodología:

1. **Creación de Archivos de Prueba:** Se desarrollaron dos archivos de código fuente en Python, `code1.py` y `code2.py`. Ambos implementan una **función recursiva para calcular la secuencia de Fibonacci**, asegurando que son semánticamente equivalentes. Sin embargo, se introdujeron variaciones textuales de forma deliberada:
    - **Nombres de Identificadores:** Se utilizaron diferentes nombres para las funciones (`fibonacci` vs. `calcular_fib`) y sus parámetros.
    - **Comentarios y Documentación:** Un archivo utilizó comentarios en línea, mientras que el otro empleó una *docstring* formal.
    - **Estructura del Código:** Se alteró ligeramente la lógica de los casos base y el espaciado para simular diferentes estilos de programación.
2. **Proceso de Comparación:** La prueba se ejecutó utilizando la función `comparar_archivos_codigo_api` expuesta en la API del sistema. Esta función encapsula de manera integral todo el flujo de trabajo del análisis de similitud, que incluye:
    - La lectura de ambos archivos de código fuente.
    - La tokenización de cada archivo para descomponer el texto en una secuencia de unidades sintácticas (tokens).
    - La asignación de pesos a cada token según su tipo (palabra clave, operador, identificador, etc.), dando mayor importancia a los elementos que definen la estructura lógica.
    - El cálculo de la Secuencia Común Más Larga (LCS) Ponderada entre las dos secuencias de tokens.

6.4. Análisis de Resultados: Detección de Similitud

6.4.1. Resultados Cuantitativos

Para validar la implementación del comparador de código, se ejecutó un experimento comparando dos funciones que resuelven el mismo problema (ej. Fibonacci) pero utilizando diferentes enfoques (recursivo vs. iterativo) y nombres de variables. La invocación se realizó a través de la API interna del sistema, la cual gestionó la lectura de archivos y la ejecución del análisis.

Los resultados obtenidos fueron los siguientes:

- **Score de Similitude Ponderado:** `0.6899` (o 69.0%)

Este score indica una similitud estructural y lógica considerable, capturando las coincidencias en los elementos algorítmicos clave a pesar de las diferencias en la implementación superficial.

La **Subsecuencia Común Más Larga (LCS)** de tokens normalizados que el algoritmo encontró es particularmente reveladora:

```
[
    "def",
    "ID_0",
    "(",
    "ID_1",
    ")",
    ":",
    "if",
    "ID_1",
    ":",
    "return",
```

```
    "ID_1",
    "ID_0",
    "(",
    "ID_1",
    "-",
    "LIT_N_1",
    ")",
    "+",
    "ID_0",
    "(",
    "ID_1",
    "-",
    "LIT_N_2",
    ")"
]
```

6.4.2. Interpretación de Resultados

La interpretación de esta secuencia demuestra el poder del sistema de normalización:

- Los tokens `def`, `if`, `return`, `(`, `)`, `:`, `-`, `+` son elementos estructurales y operativos directos que fueron encontrados en ambos códigos.
- **`ID_0`**, **`ID_1`**: Representan identificadores (nombres de función y parámetros) que, aunque eran diferentes en los archivos originales, fueron correctamente

mapeados a los mismos placeholders genéricos, indicando que cumplían el mismo rol sintáctico.

- `LIT_N_1`, `LIT_N_2`: Representan los literales numéricos (probablemente `1` y `2` en la lógica de Fibonacci) que fueron abstraídos a un formato genérico.

En conclusión, los resultados validan que la implementación es capaz de "ver a través" de las diferencias superficiales en el código para identificar y cuantificar con éxito la similitud en el núcleo lógico y estructural, cumpliendo el objetivo propuesto.

7. Conclusiones

Tras el desarrollo teórico, la implementación y la validación experimental de las soluciones propuestas, se han alcanzado las siguientes conclusiones fundamentales:

1. **El Árbol de Búsqueda Binaria Óptimo (OBST) representa una solución altamente eficaz para la indexación de vocabularios estáticos y con patrones de acceso predecibles.** Los resultados del benchmark demostraron de manera concluyente que, a pesar de su elevado costo computacional de construcción (O(n3)), el OBST ofrece una mejora significativa en el costo de búsqueda esperado (aproximadamente un 15% en nuestras pruebas) en comparación con estructuras auto-balanceables de propósito general como los árboles AVL y Rojo-Negro. Esto valida el compromiso fundamental del algoritmo: una alta inversión inicial para lograr una eficiencia operativa superior a largo plazo, justificando su uso en aplicaciones donde el rendimiento de las consultas es crítico.
2. **El algoritmo de LCS Ponderado es una herramienta robusta y precisa para la detección de similitud lógica en el código fuente.** El experimento de comparación validó que, al tokenizar el código y asignar pesos a los componentes sintácticos, el sistema es capaz de ignorar diferencias superficiales (nombres de variables, comentarios, espaciado) y cuantificar con éxito el "esqueleto algorítmico" compartido. La capacidad del sistema para identificar la estructura lógica subyacente lo convierte en una herramienta valiosa para aplicaciones académicas, como la detección de

plagio, y para la ingeniería de software, como la identificación de código duplicado o la refactorización.

3. **La integración de técnicas de procesamiento de lenguaje natural (PLN) es crucial para la aplicabilidad de los algoritmos en escenarios reales.** La extracción automática de términos y el cálculo de sus frecuencias a partir de documentos PDF fueron pasos indispensables para alimentar el algoritmo OBST con datos significativos. De manera similar, la tokenización del código fue el prerrequisito para la ejecución del LCS. Esto subraya que la eficacia de algoritmos complejos a menudo depende de un pre-procesamiento de datos inteligente y bien diseñado.

En síntesis, este proyecto ha demostrado con éxito la aplicación práctica de dos algoritmos avanzados de programación dinámica para resolver problemas relevantes en el análisis de software y documentación. Los resultados obtenidos no solo validan las propiedades teóricas de estos algoritmos, sino que también sientan las bases para el desarrollo de herramientas más sofisticadas y conscientes del contexto para la ingeniería de software.

8. Referencias

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). The MIT Press.

9. Anexos

Repositorio en GitHub:
https://github.com/jassir1902/Proyecto_Primer_Parcial_Algoritmica_II.git