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
20/08/25

Índice

1. Introducción

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

5. Resultados

6. Experimentación y Benchmarking

7. Discusión

8. Conclusiones

9. Documentación de Contribuciones

9.1. Plan de Trabajo para el Día 1

Como el trabajo se desarrolla individualmente, las responsabilidades fueron gestionadas de forma integral. Durante el primer día, el enfoque principal fue la modelación teórica y el diseño de algoritmos para ambos problemas:

- **Índice de Documentación Óptimo (OBST):** El primer paso fue modelar el problema matemáticamente, formular la recurrencia, y proponer el pseudocódigo correspondiente. También se analizó la complejidad y se sugirió una mejora para reducirla de$O(n^3)$ a $O(n^2)$.
- **Detector de Similitud de Código (LCS ponderada):** Se modeló el cálculo de la LCS con pesos y se definieron los pseudocódigos correspondientes. Además, se discutió la complejidad y la optimización de espacio utilizando técnicas de programación dinámica.

10. Referencias

11. Anexos