Detección de Similitudes entre Códigos a partir de sus Grafos de Dependencia

Daniel Toledo Martínez, Osvaldo Roberto Moreno Prieto

February 10, 2025

1 Introducción

La detección de plagio en código fuente es un área relevante en el análisis de software. Tradicionalmente, existen tres enfoques principales para este problema: la detección basada en texto, el análisis sintáctico y el análisis semántico. En este trabajo nos centraremos en el análisis semántico, que se considera más robusto frente a algunas técnicas de ofuscación comunes, como cambios en el formato, reordenamiento de bloques de código, o cambios en las estructuras de control y de datos.

El problema de modelado de la estructura sobre la cual se trabajará (PDG) queda fuera del alcance de este trabajo, por lo que nos enfocaremos en los algoritmos que operan sobre dicha estructura.

2 Propuestas de Solución

A continuación, se presentan propuestas para determinar similitudes entre los códigos mediante sus PDG.

2.1 Propuesta 1: Isomorfismo de Grafos

Para determinar si dos códigos son copias exactas, modificadas por alguna de las técnicas de ofuscación antes mencionadas, proponemos el enfoque de isomorfismo de grafos (GI). El objetivo es determinar si los grafos G y H (correspondientes a los PDG de dos códigos) son isomorfos.

2.1.1 Definición del Algoritmo

El algoritmo propuesto para resolver el problema de isomorfismo de grafos se basa en técnicas de refinamiento de clases de equivalencia (también conocido como coloración o etiquetado). Para esto, vamos a establecer un grupo de invariantes sobre los vértices (clase, grado de entrada, grado de salida, conexiones con otras clases) y realizar un proceso de refinamiento hasta que los vértices estén correctamente distribuidos en sus respectivas clases. Dos vértices pertenecen a la misma clase siempre que sus invariantes sean iguales.

Pasos del algoritmo:

1. Inicialización: Para cada vértice de G, determinar sus invariantes y agrupar los vértices en clases iniciales.
2. Refinamiento: Refinar las clases existentes usando la información de los vecinos.
3. Repetir el refinamiento hasta que las clases no puedan ser refinadas más (estabilización).
4. Aplicar el mismo proceso para H.
5. Verificar la coincidencia de las clases y generar posibles biyecciones entre las clases.

2.1.2 Análisis de Complejidad Temporal

Sean n el número de vértices y m el número de aristas:

1. (Inicialización) Calcular invariantes: $O(n + m)$
2. (Refinamiento) Generar etiqueta para todos los vértices: $O(n + m)$
3. (Refinamiento) Número de iteraciones: $O(n)$ en el peor caso (las clases se dividen hasta que cada vértice esté en su propia clase)

Total: $O(n) \times O(n + m) = O(2n + nm) = O(nm)$. Para grafos densos: $O(n^3)$. Para grafos dispersos: $O(n^2)$

2.1.3 Limitaciones

Esta solución encuentra grafos isomorfos, por tanto es débil en los casos en que ocurre cualquier adición a la lógica. Si todos los vértices tienen el mismo grado, el refinamiento no progresa (grafos regulares). Puede que no se distingan clases, requiriendo backtracking adicional (grafos fuertemente simétricos). No es concluyente para todos los grafos, algunos grafos no isomorfos pasan la prueba (falsos positivos), como los grafos de Cayley.

El problema en cuestión es suficientemente complejo en su evaluación temporal como para dar a luz a una nueva categoría de complejidad GI y aunque su complejidad es polinomial, no resuelve el problema de isomorfismo para todos los casos, por lo que se combina con otras técnicas.

2.2 Propuesta 2: Isomorfismo de Subgrafos

El segundo enfoque plantea determinar si un subgrafo de G es isomorfo con H, lo que indicaría que una parte de G implementa la misma lógica que H. Siendo G y H los PDG de A y B respectivamente.

2.2.1 Definición Formal del Problema

Dado un par de grafos $G = (V_g, E_g)$ y $H = (V_h, E_h)$, el problema de isomorfismo de subgrafos (SIP) consiste en determinar si existe un subgrafo $G' = (V'_g, E'_g)$ tal que G' sea isomorfo con H. Es decir, existe una biyección $f : V_h \to V'_g$ que preserva las adyacencias. Se debe tener en cuenta que la definición se aplica tanto para grafos dirigidos como para no dirigidos y, si existen etiquetas en los vértices o aristas, estas deben preservarse.

2.2.2 Demostración de NP-Completitud

1. Demostración de que SIP es NP Dada una biyección $f : V_h \to V'_g$, donde V_h es el conjunto de vértices del grafo H y $V'_g \subseteq V_g$ es un subconjunto de vértices del grafo G, verificamos si H es isomorfo a un subgrafo de G. Esto implica comprobar las siguientes condiciones para cada par de vértices $u, v \in V_h$:

- Si $\{u, v\} \in E_h$, entonces $\{f(u), f(v)\} \in E_g$.
- Si $\{u, v\} \notin E_h$, entonces $\{f(u), f(v)\} \notin E_g$.

La verificación se realiza en tiempo $O(n^2)$, donde $n = |V_h|$, ya que necesitamos comparar todos los pares de vértices en H con sus correspondientes imágenes en G. Dado que esta verificación es polinomial en el tamaño de la entrada, SIP es NP.

2. Reducción de k-Clique a SIP El problema **k-Clique** consiste en determinar si un grafo $G = (V, E)$ contiene un subgrafo completo (clique) de tamaño k.

Transformación:

Dada una entrada del problema k-Clique, construimos una entrada de SIP:

1. **Grafo de entrada G:** Mantenemos $G = (V, E)$ como entrada a SIP.
2. **Grafo objetivo H:** Construimos un grafo completo $H = K_k$.

Propiedades de la reducción:

- Si G contiene un clique de tamaño k, entonces existe un subgrafo de G que es isomorfo a $H = K_k$.
- Si existe un subgrafo de G isomorfo a $H = K_k$, entonces este subgrafo es un clique de tamaño k en G.

Complejidad de la reducción:

La construcción del grafo $H = K_k$ y la transformación de la instancia de k-Clique a SIP pueden realizarse en tiempo $O(k^2)$, que es polinomial en función del tamaño de la entrada del problema k-Clique.

Nos queda demostrar k-Clique.

3. Demostración de que k-Clique es NP-Completo Dada una solución (un conjunto de k vértices), verificamos si estos vértices forman un clique en G. Esto implica comprobar que todos los pares de vértices en el conjunto están conectados por una arista. Dado que hay $\binom{k}{2}$ pares de vértices, la verificación toma tiempo $O(k^2)$, que es polinomial en el tamaño de la entrada, k-Clique es NP.

Reducción desde 3-SAT a k-Clique: El problema **3-SAT** consiste en determinar si tiene solución una fórmula en forma normal conjuntiva, donde cada cláusula tiene 3 literales.

Transformación:: Dada una entrada de 3-SAT con m cláusulas y n variables, construimos un grafo $G = (V, E)$ con $k = m$:

1. **Nodos del grafo:** Cada nodo representa un literal en una cláusula. Específicamente, si tenemos m cláusulas $C_1, C_2, \ldots, C_m$, creamos un nodo por cada literal en cada cláusula.

2. **Aristas del grafo:** Conectamos dos nodos con una arista si cumplen las siguientes condiciones:

   - Los nodos pertenecen a cláusulas distintas.
   - Los literales no son complementarios.

3. **Valor de k:** Establecemos $k = m$.

Propiedades de la reducción:

- Si existe una asignación que soluciona la fórmula 3-SAT, entonces podemos seleccionar un literal verdadero de cada cláusula. Estos literales formarán un clique de tamaño $k = m$ en el grafo.

- Si existe un clique de tamaño $k = m$ en el grafo, entonces podemos usar los literales correspondientes a los nodos del clique para construir una asignación que solucione la fórmula.

Complejidad de la reducción: La construcción del grafo y la determinación de k se pueden realizar en tiempo polinomial en función del tamaño de la fórmula 3-SAT.

Luego k-Clique es NP-Hard.

2.2.3 Limitaciones

Este enfoque resulta computacionalmente costoso y es innecesariamente general para nuestro problema específico de detección de plagio.

2.3 Propuesta 3: Isomorfismo de Árboles

Dado que el PDG en nuestro problema tiene una estructura de árbol, podemos enfocar los algoritmos anteriores a un dominio más restringido.

2.3.1 Definición del Algoritmo

El **Algoritmo de Aho-Hopcroft-Ullman (AHU)** es un método eficiente para determinar si dos árboles enraizados son isomorfos. La idea central es etiquetar progresivamente los vértices de los árboles, de modo que cada vértice obtenga una descripción única basada en la información de sus hijos. Finalmente, se comparan las etiquetas de las raíces de ambos árboles para determinar si son isomorfos.

Pasos del Algoritmo:

1. **Etiquetado inicial:** Asignar una etiqueta preliminar a cada vértice según su tipo.
2. **Recursión sobre los hijos:** Para cada nodo, aplicar recursivamente el algoritmo sobre sus hijos.
3. **Ordenación y concatenación:** Ordenar las etiquetas de los hijos de cada nodo en orden lexicográfico y concatenarlas para formar una nueva etiqueta única para ese nodo.
4. **Repetición para el segundo árbol:** Realizar el mismo procedimiento para el segundo árbol.
5. **Comparación final:** Comparar las etiquetas de las raíces de ambos árboles.

Este algoritmo aprovecha la estructura recursiva de los árboles para garantizar que las etiquetas reflejen la estructura completa del subárbol asociado a cada nodo.

2.3.2 Análisis de Complejidad Temporal

El algoritmo AHU procesa cada nodo ordenando las etiquetas de sus hijos y concatenándolas para formar una nueva etiqueta única, analizaremos la complejidad temporal del algoritmo utilizando inducción para demostrar que su tiempo de ejecución es $O(n \log n)$, donde n es la cardinalidad de nodos en el árbol.

Caso Base ($n = 1$): Cuando el árbol tiene un solo vértice, no hay hijos que procesar ni etiquetas que concatenar. Por lo tanto, la complejidad es constante:

$$T(1) = O(1)$$

Hipótesis de inducción: ($m < n$): Supongamos que para todo árbol de tamaño ($m < n$) se cumple que:

$$T(m) \leq am \log m$$

para alguna constante ($a > 0$)

Lema: Supongamos que el árbol tiene n nodos y que el nodo actual tiene k hijos con tamaños de subárboles $n_1, n_2, \ldots, n_k$, donde:

$$n_1 + n_2 + \cdots + n_k = n - 1$$

La complejidad entonces pasa por:

1. **Procesar los subárboles de los hijos:** $T(n_1) + T(n_2) + \cdots + T(n_k)$.
2. **Ordenar las etiquetas de los hijos:** Dado que hay k hijos, ordenar es $O(k \log k)$.

Procesar este árbol es:

$$T(n) = \sum_{i=1}^{k} T(n_i) + c_1 \cdot k \log k$$

donde $c_1 > 0$ es una constante.

Paso inductivo: Queremos demostrar que $T(N) \leq c \cdot N \log N$. Usando el lema, tenemos que:

$$T(N) = \sum_{i=1}^{k} T(n_i) + c_1 \cdot k \log k$$

Por hipótesis $T(n_i) \leq c_i \cdot n_i \log n_i$. Sustituyendo en la ecuación:

$$T(N) \leq \sum_{i=1}^{k} c_i \cdot n_i \log n_i + c_1 \cdot k \log k$$

Luego usando:

$$\sum_{i=1}^{k} n_i = N - 1$$

$$sum_{i=1}^{k} n_i \log n_i \leq (N-1)\log(N-1)$$

$$k \log k \leq (N-1)\log(N-1)$$

$$c_2 = max(ci)$$

Podemos llegar a:

$$T(N) \leq c_2 \cdot (N-1)\log(N-1) + c_1 \cdot (N-1)\log(N-1)$$

$$T(N) \leq (c_2 + c_1) \cdot (N-1)\log(N-1)$$

Como $(N-1)\log(N-1) \leq N \log N$ para $N \geq 2$, podemos escribir:

$$T(N) \leq c_3 \cdot N \log N$$

donde $c_3 = c_2 + c_1$.

2.3.3 Correctitud

Teorema Sea T un árbol finito. El algoritmo AHU asigna a cada árbol T una etiqueta canónica $f(T)$ tal que, dados dos árboles T_1 y T_2, se tiene

$$f(T_1) = f(T_2) \quad \iff \quad T_1 \cong T_2,$$

es decir, T_1 y T_2 son isomorfos si y solo si sus etiquetas canónicas son idénticas.

Demostración (por inducción en la altura del árbol):

1. **Caso Base:** Consideremos un árbol T de altura 0, es decir, un árbol que consiste en un solo nodo (nodo hoja). - Para un nodo hoja, el algoritmo asigna una etiqueta preliminar $f(v)$ basada en el tipo del nodo (u otra propiedad intrínseca), que es única para ese tipo. - Como no existen hijos que procesar, la etiqueta canónica del árbol es simplemente $f(T) = f(v)$. - Por lo tanto, para cualquier par de hojas, se tiene que si $f(v_1) = f(v_2)$ entonces los nodos (y por ende los árboles unitarios) son isomorfos, cumpliéndose la propiedad.

2. **Hipótesis de Inducción:** Supongamos que para todo árbol T' de altura menor que h, el algoritmo AHU asigna una etiqueta canónica única $f(T')$ que caracteriza de manera completa la estructura y el contenido del árbol. Es decir, para cualquier árbol de altura $m < h$,

$$f(T') \text{ es única y } f(T'_1) = f(T'_2) \iff T'_1 \cong T'_2.$$

3. **Paso Inductivo:** Consideremos un árbol T de altura h. Sea v el nodo raíz de T y sea $v_1, v_2, \ldots, v_k$ los nodos hijos de v. Cada subárbol T_i engendrado por v_i tiene altura menor que h.

- **Aplicación de la hipótesis:** Por hipótesis inductiva, cada subárbol T_i tiene una etiqueta canónica $f(T_i)$ que es única y describe completamente su estructura.

- **Proceso de etiquetado en el nodo v:** El algoritmo procede a ordenar las etiquetas $f(T_1), f(T_2), \ldots, f(T_k)$ (lo cual es crucial para eliminar la dependencia del orden de los hijos) y luego concatena esta secuencia ordenada con la etiqueta preliminar $f(v)$ asignada al nodo v. Formalmente, definimos:

$$f(T) = f(v) \,\|\, \text{concat}\Big(\text{sort}\big(f(T_1), f(T_2), \ldots, f(T_k)\big)\Big),$$

donde
$Vert$ denota la concatenación y sort la ordenación lexicográfica de las etiquetas.

- **Unicidad de la etiqueta:** Dado que el proceso de ordenación y concatenación se realiza de forma determinista, la etiqueta $f(T)$ es única para la estructura completa del árbol T. Además, si se tienen dos árboles T_1 y T_2 de altura h tales que $f(T_1) = f(T_2)$, entonces, por el proceso de construcción de las etiquetas, las raíces deben tener la misma etiqueta preliminar y sus respectivos conjuntos de etiquetas de subárboles, al ser ordenados, deben coincidir. Por la**Teorema.** Sea T un árbol finito. El algoritmo AHU asigna a cada árbol T una etiqueta canónica $f(T)$ tal que, dados dos árboles T_1 y T_2, se tiene

$$f(T_1) = f(T_2) \quad \iff \quad T_1 \cong T_2,$$

es decir, T_1 y T_2 son isomorfos si y solo si sus etiquetas canónicas son idénticas.

2.3.4 Limitaciones

Este enfoque es más eficiente que los anteriores, pero aún enfrenta desafíos en la comparación de árboles grandes o complejos. Además de que posee la misma deficiencia del primer código en cuanto a que no es resistente a adiciones en el código

2.4 Propuesta 4: Isomorfismo de Subgrafos en Árboles No Ordenados

Una de las ventajas que proporciona el algoritmo AHU con su esquema de codificación es la capacidad de verificar si, dados dos árboles T_1 y T_2, existe algún subárbol de T_2 isomorfo a T_1. Esto puede lograrse de manera eficiente recorriendo T_2 y comparando los códigos de los nodos de T_2 con el código de la raíz de T_1.

Sin embargo, uno de los objetivos centrales del problema es mejorar la robustez frente a la adición de nuevos elementos. Resulta conveniente extender el problema más allá del isomorfismo de subárboles a subgrafos.

2.4.1 Definición Formal del Problema

Sean S (el *patrón*) y T (el *objetivo*) dos árboles etiquetados no ordenados. Un árbol no ordenado es una estructura jerárquica donde los hijos de cada nodo no tienen un orden predefinido. Formalmente se define recursivamente como:

$$T = (r, \{T_1, T_2, \ldots, T_k\}),$$

donde r es la raíz y $\{T_1, \ldots, T_k\}$ es un conjunto (no ordenado) de subárboles. El problema consiste en determinar si existe un subgrafo $H \subseteq T$ tal que H sea isomorfo a S.

Este problema se distingue del isomorfismo de subárboles, ya que H no necesariamente debe mantener la estructura jerárquica completa de T, sino que puede ser cualquier subgrafo que satisfaga las condiciones de isomorfismo con S.

2.4.2 Correctitud

Definición 1 (Isomorfismo de Árboles) *Dos árboles S y T son isomorfos ($S \cong T$) si:*

- *Sus raíces tienen la misma etiqueta.*
- *Existe una biyección entre los hijos de S y T, tal que cada par de hijos correspondientes también son isomorfos.*

Definición 2 (Grafo Bipartito) *Un grafo $G = (U, V, E)$ donde U y V son conjuntos disjuntos de nodos, y las aristas E conectan nodos de U a V.*

- Caso Base: Árboles de Altura 0 (Hojas)
  - **Hipótesis**: Si S y T son hojas, $S \cong T$ si y solo si tienen la misma etiqueta.
  - **Verificación del algoritmo**:
    - La función check_subtree($t\_node, s\_node, memo$) compara las etiquetas de las raíces.
    - Como no hay hijos, retorna True si las etiquetas coinciden.
  - **Conclusión**: El algoritmo es correcto para árboles de altura 0.
- Hipótesis Inductiva Supongamos que para todo par de subárboles S' y T' de altura $\leq k$, check_subtree($S', T', memo$) retorna True si y solo si $S' \cong T'$.
- Paso Inductivo: Árboles de Altura $k + 1$

- **Condiciones para $S \cong T$**:
  - Las raíces de S y T tienen la misma etiqueta.
  - Existe una biyección entre los hijos de S y T, donde cada par de hijos correspondientes son isomorfos.
- **Verificación del algoritmo**:
  - Compara las etiquetas de las raíces.
  - Construye un grafo bipartito donde una arista (i, j) existe si el hijo i de S es isomorfo al hijo j de T (usando check_subtree recursivamente).
  - Usa Hopcroft-Karp para encontrar un emparejamiento máximo.
  - Si el tamaño del emparejamiento es igual al número de hijos de S, retorna True.
- **Análisis**:
  - Por la hipótesis inductiva, las llamadas recursivas check_subtree son correctas para subárboles de altura $\leq k$.
  - Un emparejamiento máximo de tamaño |hijos(S)| implica una biyección entre los hijos de S y T.
  - Por lo tanto, el algoritmo retorna True si y solo si $S \cong T$.

2.4.3 Complejidad Temporal del Algoritmo

El algoritmo consta de las siguientes etapas principales:

1. **Búsqueda de Nodos Raíz**: Recorrer el árbol T_2 para encontrar nodos cuyas etiquetas coincidan con la raíz de T_1. Este paso tiene una complejidad de $O(n_{T_2})$, donde n_{T_2} es el número de nodos en T_2.

2. **Verificación Recursiva**: Para cada nodo candidato en T_2, se verifica si los subárboles coinciden. Esto implica:

   - **Comparación de Etiquetas**: Asegurar que las etiquetas de los nodos coincidan. Esta operación es $O(1)$ por nodo.
   - **Emparejamiento de Hijos**: Construir un grafo bipartito entre los hijos de T_1 y T_2, lo cual tiene una complejidad de $O(k_{T_1} \cdot k_{T_2})$, donde k_{T_1} y k_{T_2} son los grados máximos de los nodos en T_1 y T_2, respectivamente.
   - **Algoritmo de Hopcroft-Karp**: Encontrar el emparejamiento máximo en el grafo bipartito, con una complejidad de $O(\sqrt{k_{T_2}} \cdot k_{T_1} \cdot k_{T_2})$

3. **Memoización**: Memoización limita a O(n·m) pares, para reducir llamados recursivos duplicados.

Complejidad:

$$O(n_{T_2} \cdot k_{T_1} \cdot k_{T_2} \cdot \sqrt{k_{T_2}})$$

donde:

- n_{T_2}: Número de nodos en T_2.
- k_{T_1}, k_{T_2}: Grados máximos de los nodos en T_1 y T_2, respectivamente.

2.5 Propuesta 5: Tree Edit Distance (No ordenados)

Otro enfoque que puede ser utilizado es intentar determinar distancia entre un par de árboles para determinar proximidad entre los mismos, para ello se propone el presente enfoque

2.5.1 Definición formal del problema

2.5.2 Definición de conceptos

Definición 3 (Tree Edit Distance (TED)) *Dados dos árboles T_1 y T_2, y costos delete, insert, rename, el TED es el costo mínimo para transformar T_1 en T_2 usando dichas operaciones.*

Definición 4 (Máximo Subárbol Común (MCS)) *Dados dos árboles T_1 y T_2, el problema MCS(T_1, T_2) consiste en encontrar el subárbol común más grande (en número de nodos) entre ambos.*

2.5.3 Demostración de NP-hard

Teorema 1 *El problema del TED para árboles no ordenados es NP-hard.*

La demostración se realiza mediante una **reducción polinomial** desde el problema MCS, conocido como NP-hard [**?**].

Paso 1: Asignación de Costos.
Definimos los costos de operación para el TED como:

$$\text{delete\_cost} = \text{insert\_cost} = 1, \quad \text{rename\_cost}(u, v) = \begin{cases} 0 & \text{si } u = v, \\ \infty & \text{si } u \neq v. \end{cases}$$

Paso 2: Relación entre TED y MCS.
Sea S el máximo subárbol común entre T_1 y T_2. Entonces:

$$\text{TED}(T_1, T_2) = |T_1| + |T_2| - 2|S|.$$

- Los nodos de S no requieren operaciones (renombrado con costo 0).
- Los nodos en $T_1 \setminus S$ se eliminan (costo $|T_1| - |S|$).
- Los nodos en $T_2 \setminus S$ se insertan (costo $|T_2| - |S|$).

Paso 3: Reducción.
Supongamos un oráculo que resuelve TED en tiempo polinomial. Entonces, para resolver MCS(T_1, T_2), calculamos:

$$|S| = \frac{|T_1| + |T_2| - \text{TED}(T_1, T_2)}{2}.$$

Esto reduce MCS a TED en tiempo polinomial.

Dado que MCS es NP-hard y hemos reducido MCS a TED en tiempo polinomial, el problema TED para árboles no ordenados es NP-hard.

3 Conclusiones

En este trabajo se han presentado tres enfoques para la detección de similitudes entre códigos mediante el análisis de sus Grafos de Dependencia. La elección del enfoque adecuado dependerá de la naturaleza del código y las modificaciones aplicadas, con el isomorfismo de árboles ofreciendo una solución eficiente para estructuras más simples.