Actividad 1.6

Desarrollo de un Marco de Trabajo

Equipo 2:

Fernanda Ríos Juárez - A01656047

José Aram Méndez Gómez - A01657142

Domingo Mora Pérez - A01783317

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 570)

Prof. Pedro Oscar Pérez Murueta

01 de Diciembre de 2025

PLANTEAMIENTO DEL PROBLEMA

El desarrollo de software ha experimentado un crecimiento exponencial en las últimas décadas, convirtiéndose en una actividad fundamental tanto en contextos académicos como industriales. Este crecimiento ha traído consigo un problema persistente: la dificultad para proteger los derechos de autor del código fuente y detectar apropiación indebida del trabajo intelectual de otros programadores.

Contexto del Problema

El plagio se define como el uso no autorizado de material original creado por otra persona [3]. En el contexto de la programación, este problema tiene múltiples dimensiones. Primero, el código fuente representa una inversión considerable de tiempo, conocimiento técnico y esfuerzo intelectual. Sin embargo, la naturaleza digital del software facilita la copia, distribución y modificación del código sin autorización. Segundo, a diferencia de otros tipos de obras protegidas por derechos de autor, el código fuente puede expresarse de múltiples formas sintácticamente diferentes pero funcionalmente equivalentes, lo que dificulta la detección de copias modificadas superficialmente.

En entornos académicos, el plagio de código es particularmente problemático. Los estudiantes pueden copiar soluciones completas de compañeros o de repositorios públicos como GitHub, realizando modificaciones cosméticas como cambiar nombres de variables, reordenar funciones o ajustar el formato para evadir la detección. Según estudios recientes, hasta el 30% de las tareas de programación en cursos universitarios presentan algún grado de similitud sospechosa que sugiere plagio.

En la industria, la protección del código propietario es crucial. Las empresas desarrollan algoritmos y sistemas que representan ventajas competitivas significativas. Sin mecanismos efectivos de detección, empleados deshonestos o competidores pueden apropiarse de este código, causando pérdidas económicas y disputas legales complejas.

A diferencia del plagio en textos literarios o académicos, donde se puede comparar el contenido palabra por palabra, el código fuente presenta desafíos técnicos únicos. Un mismo algoritmo puede implementarse con diferentes estructuras de control (bucles `for` versus `while`), diferentes nombres de variables, diferentes organizaciones de funciones, e incluso diferentes lenguajes de programación. Las herramientas tradicionales de detección de plagio basadas en comparación textual, como las utilizadas para documentos, son insuficientes porque modificaciones superficiales pueden evadir fácilmente estos sistemas.

Problemática Específica

El problema central que abordamos es: ¿cómo podemos identificar de manera automática, eficiente y precisa si un código fuente es una copia no autorizada de otro, considerando que las copias pueden estar modificadas mediante técnicas de ofuscación como renombramiento de variables, reestructuración de control de flujo, o inserción de código irrelevante?

Esta problemática es relevante porque:

Aspecto Académico: Las instituciones educativas necesitan herramientas para verificar la originalidad de las tareas de programación. Los profesores deben poder identificar plagio rápidamente en clases con cientos de estudiantes, sin invertir horas en revisión manual. Esto es fundamental para mantener la integridad académica y asegurar que los estudiantes desarrollen habilidades de programación genuinas.

Aspecto Legal: Los tribunales necesitan herramientas técnicas objetivas para determinar si ha ocurrido una violación de propiedad intelectual en disputas sobre código fuente. Actualmente, estos casos dependen de testimonios de expertos que pueden ser subjetivos y costosos. Una herramienta automatizada proporcionaría evidencia cuantitativa y reproducible.

Aspecto Económico: Las empresas necesitan proteger su inversión en desarrollo de software. Poder detectar si un competidor está utilizando código propietario permite tomar acciones legales oportunas. Además, en procesos de auditoría de licencias, las organizaciones necesitan verificar que no están utilizando inadvertidamente código con restricciones de uso.

Aspecto Técnico: Los repositorios de código abierto y las plataformas de desarrollo colaborativo necesitan mecanismos para detectar contribuciones que violan licencias o derechos de autor. Esto protege tanto a los mantenedores de proyectos como a los usuarios que podrían enfrentar problemas legales por utilizar código infractor.

Impacto Potencial

Resolver este problema permitirá crear herramientas que puedan verificar automáticamente si un código es derivado de otro, similar a cómo funcionan los detectores de plagio para texto como Turnitin, pero específicamente diseñadas para las características únicas del código fuente. Un sistema efectivo debe ser capaz de:

- Detectar similitud estructural más allá de diferencias sintácticas superficiales
- Procesar consultas rápidamente (idealmente en menos de 10 segundos) para ser práctico en contextos con grandes volúmenes de código
- Mantener alta precisión (cercana al 95%) para minimizar falsos positivos que generan acusaciones injustas y falsos negativos que permiten que el plagio pase desapercibido

- Escalar eficientemente para comparar contra bases de datos extensas con miles o millones de códigos de referencia

Esto facilitaría la protección de la propiedad intelectual en software y promovería un ecosistema más justo para el desarrollo, donde los creadores originales reciben el reconocimiento y la protección que merecen, y donde la innovación genuina es valorada sobre la copia.

MARCO TEÓRICO

Derechos de Autor en Software

Los derechos de autor protegen las obras originales creadas por una persona, incluyendo el código fuente de programas de computadora. En el contexto del software, la protección se extiende tanto a la expresión literal del código (las líneas específicas escritas) como a elementos no literales como la estructura, secuencia y organización (SSO) del programa [4].

A diferencia de otros tipos de obras, el código fuente presenta desafíos únicos para la detección de plagio: un mismo algoritmo puede implementarse de múltiples formas sintácticamente diferentes pero funcionalmente equivalentes. Por ejemplo, usar un bucle `for` versus un bucle `while`, o diferentes nombres de variables, no cambia la funcionalidad pero modifica el texto del código. Según Sag [4], esto plantea un dilema en la protección de derechos de autor: ¿dónde termina la expresión protegible y dónde comienza la idea no protegible?

Detección de Plagio en Código Fuente

El plagio de código se define como el uso no autorizado de código escrito por otra persona, presentándolo como propio [3]. Este problema es particularmente relevante en contextos académicos, donde estudiantes pueden copiar código de compañeros o de fuentes públicas, y en la industria, donde puede haber apropiación indebida de código propietario.

Las técnicas tradicionales de detección de plagio basadas en comparación textual (como las usadas para documentos) son insuficientes para código, ya que modificaciones superficiales como renombrar variables, reordenar funciones o cambiar el formato pueden evadir la detección fácilmente [3]. El-Rashidy et al. demostraron que los enfoques basados en análisis semántico profundo son necesarios para detectar plagio más sofisticado, logrando precisiones del 94% en sus experimentos.

Análisis Sintáctico y Abstract Syntax Trees (AST)

El análisis sintáctico es el proceso de analizar código fuente para determinar su estructura gramatical. Un compilador o parser convierte el código fuente en un Abstract Syntax Tree (AST), una representación jerárquica que captura la estructura del programa independientemente de detalles superficiales como espacios en blanco, comentarios o nombres de variables [2].

Por ejemplo, estos dos fragmentos de código producen ASTs estructuralmente idénticos:

```python
def suma(a, b):
    return a + b
```

```python
def add(x, y):
    resultado = x + y
    return resultado
```

Ambos representan la misma estructura: una función que toma dos parámetros y retorna su suma. Al comparar ASTs en lugar de texto plano, podemos detectar similitud estructural que persiste a través de modificaciones superficiales. Chen et al. [2] demostraron que el análisis basado en estructuras abstractas es fundamental para sistemas robustos de detección de similitud, ya que captura la "esencia" del código más allá de su presentación textual.

Los ASTs organizan el código en nodos que representan construcciones del lenguaje (declaraciones, expresiones, operadores) y aristas que representan relaciones sintácticas. Esta representación estructurada permite aplicar técnicas de comparación de árboles y grafos que son más sofisticadas que la simple comparación de texto.

Técnicas de Hashing para Detección de Similitud

El hashing es una técnica que convierte datos de longitud variable en un código de longitud fija. Para detección de plagio, necesitamos hashes que preserven la similitud: códigos similares deben producir hashes similares, a diferencia del hashing criptográfico tradicional (como MD5 o SHA-256) donde un cambio mínimo produce un hash completamente diferente.

MinHash es un algoritmo diseñado específicamente para este propósito [5]. Funciona generando múltiples valores hash de subconjuntos del código y

seleccionando el mínimo de cada conjunto. La probabilidad de que dos documentos tengan el mismo MinHash es proporcional a su similitud Jaccard (la proporción de elementos que comparten). Este método fue originalmente desarrollado para detectar documentos duplicados en la web, pero ha demostrado ser efectivo para código fuente.

Ventajas de MinHash para código:

- **Eficiencia computacional:** Complejidad O(n) para generar el hash, donde n es el tamaño del código
- **Resistencia a modificaciones menores:** Cambios pequeños producen hashes similares, no completamente diferentes
- **Escalabilidad:** Permite comparar rápidamente contra grandes bases de datos mediante técnicas de búsqueda aproximada

Chang et al. [5] aplicaron técnicas similares para audio fingerprinting, logrando retrieval de alta especificidad en milisegundos. Los mismos principios matemáticos de hashing perceptivo son aplicables a código fuente.

Locality-Sensitive Hashing (LSH)

LSH es una técnica que organiza elementos en "buckets" de manera que elementos similares tienen alta probabilidad de caer en el mismo bucket [5]. Para detección de plagio, esto significa que podemos:

1. Generar el hash del código sospechoso
2. Buscar solo en el bucket correspondiente en la base de datos
3. Comparar en detalle solo con candidatos del mismo bucket

Esto reduce drásticamente el tiempo de búsqueda de O(n) a O(1) amortizado, permitiendo consultas en segundos incluso contra millones de códigos de referencia. La clave está en diseñar funciones hash que preserven la noción de similitud relevante para código: códigos estructuralmente similares deben tener alta probabilidad de colisión en los buckets.

Normalización de Código

Para detectar plagio efectivamente, el código debe normalizarse antes del análisis. Las técnicas de normalización incluyen:

- **Renombramiento de identificadores:** Convertir todos los nombres de variables y funciones a formas canónicas (var1, var2, func1, etc.) para eliminar diferencias por nomenclatura
- **Eliminación de comentarios y espacios en blanco:** Remover elementos que no afectan la funcionalidad

- **Canonicalización de estructuras:** Estandarizar patrones equivalentes, como convertir ciertos tipos de bucles a formas canónicas cuando sean semánticamente equivalentes
- **Ordenamiento de elementos independientes:** Reordenar declaraciones que no tienen dependencias entre sí para evitar que el simple reordenamiento oculte el plagio

Esta normalización hace que el código sea más difícil de disfrazar mediante modificaciones superficiales. Sin embargo, debe aplicarse cuidadosamente para no eliminar diferencias legítimas que distinguen implementaciones originales de copias [2].

Deep Learning para Detección Semántica

Estudios recientes [3] han demostrado que las redes neuronales pueden capturar similitud semántica en código mediante embeddings: representaciones vectoriales densas que codifican el significado del código independientemente de su sintaxis específica.

Modelos como Code2Vec o CodeBERT aprenden a mapear fragmentos de código a vectores en un espacio donde códigos funcionalmente similares están cerca entre sí, medido por distancia coseno o euclidiana. Esto permite detectar plagio incluso cuando el código ha sido significativamente reestructurado pero mantiene la misma lógica subyacente.

El-Rashidy et al. [3] lograron 94% de precisión en detección de plagio textual usando embeddings profundos que capturan significado semántico. Los mismos principios aplicados a código fuente permiten detectar cuando dos programas son funcionalmente equivalentes aunque su implementación superficial difiera.

Métricas de Similitud y Evaluación

Para cuantificar la similitud entre códigos, se utilizan varias métricas:

- **Similitud Jaccard:** $|A \cap B| / |A \cup B|$ para conjuntos de tokens o n-gramas. Mide la proporción de elementos compartidos sobre el total de elementos únicos.
- **Distancia de edición de árbol:** Número mínimo de operaciones (inserción, eliminación, modificación de nodos) necesarias para transformar un AST en otro.
- **Similitud coseno:** Para comparar representaciones vectoriales (embeddings). Valores cercanos a 1 indican alta similitud.

La evaluación del sistema se realiza mediante una matriz de confusión que clasifica pares de códigos en:

- **Verdaderos positivos (TP):** Plagio correctamente detectado
- **Verdaderos negativos (TN):** Código original correctamente identificado como no plagio
- **Falsos positivos (FP):** Código original marcado incorrectamente como plagio
- **Falsos negativos (FN):** Plagio que no fue detectado

De estas métricas se derivan indicadores de desempeño como precisión (TP/(TP+FP)), recall (TP/(TP+FN)) y F1-score (media armónica de precisión y recall).

Integración de Técnicas

Nuestro proyecto integra estas técnicas en un sistema híbrido que combina múltiples enfoques:

1. **Parser AST:** Extrae la estructura del código independientemente de sintaxis superficial, utilizando herramientas estándar de compilación adaptadas para análisis estático
2. **Normalización:** Estandariza el código para resistir técnicas básicas de ofuscación
3. **MinHash/LSH:** Genera huellas digitales eficientes para búsqueda rápida en bases de datos extensas
4. **Comparación estructural:** Valida candidatos mediante análisis detallado del AST usando métricas de distancia de edición de árbol
5. **Métricas cuantitativas:** Establece umbrales objetivos para distinguir plagio de similitud legítima que puede surgir por patrones comunes de programación

La literatura existente [2][3][5] proporciona las herramientas individuales, pero falta un sistema integrado y validado específicamente para detección de plagio de código con énfasis en eficiencia (consultas en aproximadamente 10 segundos) y alta precisión (cercana al 95%) contra bases de datos extensas. Estudios recientes [1][6] sobre protección de propiedad intelectual en modelos de IA demuestran que técnicas similares de fingerprinting y watermarking son complementarias y necesarias para sistemas robustos, principios que aplicamos al dominio de código fuente.

OBJETIVOS

Objetivo General

Diseñar y desarrollar un prototipo de herramienta para la detección de similitud estructural en código fuente, generando hashes únicos basados en la estructura sintáctica mediante un parser AST y métodos cuantitativos (específicamente

MinHash y Locality-Sensitive Hashing), con la finalidad de identificar posible plagio, evaluando su precisión contra técnicas del estado del arte.

Objetivos Específicos

Objetivo Específico 1: Desarrollar un algoritmo de normalización y parsing que extraiga la estructura sintáctica del código fuente mediante Abstract Syntax Trees (AST), aplicando técnicas de canonicalización para generar representaciones consistentes independientes de variaciones superficiales como nombres de variables o formateo.

Objetivo Específico 2: Implementar un algoritmo de hashing perceptivo basado en MinHash que genere huellas digitales únicas de 256 bits para códigos fuente, con la propiedad de que códigos estructuralmente similares produzcan hashes con alta similitud medida por distancia de Hamming.

Objetivo Específico 3: Compilar y procesar un micro-dataset de al menos 100 códigos de referencia de diferentes niveles de complejidad, generando y almacenando sus correspondientes hashes en una estructura indexada mediante Locality-Sensitive Hashing para establecer una base de datos verificable y eficiente.

Objetivo Específico 4: Validar la precisión del sistema mediante una matriz de confusión, comparando las firmas generadas contra la base de referencia y contra una técnica baseline del estado del arte, con el objetivo de obtener una precisión mínima del 95% y tiempo de consulta máximo de 10 segundos para búsquedas en la base de datos completa.

REFERENCIAS

[1] B. Darvish Rouhani, H. Chen, and F. Koushanfar, "DeepSigns: A generic watermarking framework for IP protection of deep learning models," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, Providence, RI, USA, Apr. 2019, pp. 485-497. doi: 10.1145/3297858.3304042

[2] H. Chen, B. D. Rouhani, C. Fu, J. Zhao, and F. Koushanfar, "Perceptual hashing of deep convolutional neural networks for model copy detection," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 19, no. 2, pp. 1-24, Feb. 2023. doi: 10.1145/3572777

[3] M. A. El-Rashidy, R. G. Mohamed, N. A. El-Fishawy, and M. A. Shouman, "Reliable plagiarism detection system based on deep learning approaches," *Neural Comput. Appl.*, vol. 34, no. 21, pp. 18837-18851, Nov. 2022. doi: 10.1007/s00521-022-07486-w

[4] M. Sag, "Copyright safety for generative AI," *Houston Law Rev.*, vol. 61, no. 2, pp. 295-344, 2023. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4438593

[5] S. Chang, D. Lee, J. Park, H. Lim, K. Lee, K. Ko, and Y. Han, "Neural audio fingerprint for high-specific audio retrieval based on contrastive learning," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Singapore, May 2022, pp. 3054-3058. doi: 10.1109/ICASSP43922.2022.9747513

[6] X. Sun and J. Zhang, "Deep model intellectual property protection via deep watermarking," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 8, pp. 4005-4020, Aug. 2022. doi: 10.1109/TPAMI.2021.3088846