UNIVERSIDAD MAYOR DE SAN ANDRES

FACULTAD DE CIENCIAS PURAS Y NATURALES

CARRERA DE INFORMATICA

TESIS DE GRADO

DISEÑO DE UN ALGORITMO PARA LA DETECCIÓN DE SIMILITUD ENTRE CÓDIGOS FUENTE

Para optar por el Título de Licenciatura en Informática

Mención: Ciencias de la Computación

Postulante: Univ. Edson Eddy Lecoña Zarate

Tutor: M.Sc. Jorge Humberto Terán Pomier

La Paz - Bolivia

2022

Índice general

CAPÍTULO 1: MARCO REFERENCIAL

1.1. Introducción

[Cheers *et al.*, 2021] Explica que la identificación de similitud entre códigos fuente puede servir para varios propósitos, entre ellos están el estudio de la evolución de código fuente de un proyecto, detección de prácticas de plagio, detección de prácticas de re utilización, extracción de código para "refactorización" del mismo y seguimiento de defectos para su corrección.

Los estudiantes durante su proceso de formación elaboran trabajos, proyectos, tareas y ejercicios de programación escritos en un lenguaje de programación, estas actividades se realiza de forma individual o grupal, cuando estas actividades se desarrollan de forma individual los estudiantes deben ser conscientes de que todo trabajo entregado debe ser de su creación, pues tiene como función medir la capacidad de resolución de problemas y el enfoque lógico y otros. Por ello encontrar similitud en trabajos de programación presentados por los estudiantes, puede ser identificado como plagio.

Una de las formas para detectar el plagio en los trabajos de programación consiste en realizar la comparación entre los trabajos entregados por los estudiantes de la materia. De a modo de obtener una lista de estudiantes que tienen trabajos similares. Realizar la comparación de los trabajos manualmente puede llegar a ser un trabajo moroso, por lo cual contar con una herramienta de software que realice las comparaciones de trabajos de forma automática es de gran utilidad.

En la actualidad existen diferentes herramientas de software que aplican métodos para detección de similitud entre códigos fuente, a partir de las características se pudo evidenciar que estas presentan deficiencias como ser la obsolescencia, sistemas cerrados (sin código abierto), un proceso complejo de evaluación de similitud, la incapacidad para utilizar o no una gran base de información. Esto se debe a que fueron diseñadas para detectar plagio

entre un grupo pequeño de archivos de código fuente.

Por lo cual contar con un sistema para la detección de similitud entre códigos fuente llega a ser útil para detectar plagio en trabajos de cátedra presentados por estudiantes. El presente trabajo de tesis se centra en el diseño e implementación de un algoritmo para la detección de similitud entre códigos fuente que tenga un buen desempeño, en términos de tiempo de ejecución, espacio de memoria ocupado y precisión.

1.2. Problema

1.2.1. Antecedentes del problema

Se analizaron herramientas para la detección de similitud entre código fuente más populares, a continuación se dará breve explicación de las características que tienen estas.

Sherlock

Es una herramienta de código abierto, que trabaja con código escrito en los lenguajes Java, C y texto natural. Esta herramienta no cuenta con una interfaz gráfica. Fue desarrollada por la Universidad de Sydney. Los resultados arrojados se basan en un porcentaje que se corresponde con las similitudes encontradas. El porcentaje 0 % significa que no hay similitudes, y 100 % significa que hay muchas posibilidades de que tengan partes iguales. Al no utilizar el documento en su totalidad, no se puede afirmar que sean completamente iguales. Sólo trabaja con archivos locales, no busca similitudes en Internet [Díaz *et al.*, 2007].

Simian

Esta herramienta no es de código abierto. Identifica la duplicación de códigos escritos en JAVA, C, C++, COBOL, Ruby, JSP, ASP, HTML, XML, Visual Basic y texto natural. Fue desarrollada por una consultora de Australia llamada REDHILL. No cuenta con una interfaz gráfica, por lo tanto, el ingreso de los parámetros es a través de línea de comando. Sólo trabaja con archivos locales, no busca similitudes en Internet [Díaz *et al.*, 2007].

Jplag

Esta herramienta no es de código abierto, sólo permite la programación del cliente, pero el servidor es el que realiza las operaciones de comparación; se encuentra en la Universidad de Karlsruhe de Alemania, en la cual fue desarrollada. Trabaja con los siguientes lenguajes C, C++, Java, Scheme y texto natural. Fue desarrollada por la Universidad de Karlsruhe de Alemania. Analiza la estructura y sintaxis del código, no comparando texto. Este sistema sólo trabaja con archivos locales, no busca similitudes en Internet. Su arquitectura de trabajo es Cliente/Servidor. La interfaz es a través de línea de comando, o, puede implementarse un cliente propio. Para el manejo de los archivos enviados, puede utilizarse el cliente brindado por la universidad que lo desarrolló o bien crear un cliente propio, como mencionamos anteriormente [Díaz *et al.*, 2007].

Tester SIM

Es una herramienta de código abierto. Trabaja con los lenguajes C, JAVA, Lisp, Modula 2, Pascal y texto natural. Está desarrollada por la Universidad de Amsterdam. No cuenta con una interfaz gráfica, se ingresan los distintos parámetros por línea de comando. Los resultados brindados por la misma aparecen separados en dos columnas, mostrando en cada una las porciones de código iguales. Sólo trabaja con búsquedas de similitudes en archivos locales. También se puede procesar un conjunto de archivos viejos contra un conjunto de archivos nuevos [Díaz *et al.*, 2007].

MOSS

Software creado por el profesor Alex Aiken en la universidad de Stanford, es el primer servicio que inició en la web, siendo una referencia a nivel mundial. Este, permite comparar hasta 250 archivos en 25 lenguajes de programación [Hage y Rademaker, 2010]. Aunque no tiene licencia como software libre, su uso dentro del ambiente académico es gratuito y ofrecido desde un servidor de Stanford. Es de difícil su configuración y tiene poca documentación [Pachón, 2019].

1.2.2. Planteamiento del problema

Los estudiantes durante su proceso de formación elaboran trabajos, proyectos, tareas y ejercicios de programación escritos en un lenguaje de programación, estas actividades se realiza de forma individual o grupal, cuando estas actividades se desarrollan de forma individual los estudiantes deben ser conscientes de que todo trabajo entregado debe ser de su creación, pues tiene como función medir la capacidad de resolución de problemas y el enfoque lógico y otros. Por ello encontrar similitud en trabajos de programación presentados por los estudiantes, puede ser identificado como plagio.

Una de las formas para detectar el plagio en los trabajos de programación consiste en realizar la comparación entre los trabajos entregados por los estudiantes de la materia. De a modo de obtener una lista de estudiantes que tienen trabajos similares. Realizar la comparación de los trabajos manualmente puede llegar a ser un trabajo moroso, por lo cual contar con una herramienta de software que realice las comparaciones de trabajos de forma automática es de gran utilidad.

En la actualidad existen diferentes herramientas de software que aplican métodos para detección de similitud entre códigos fuente, a partir de las características se pudo evidenciar que estas presentan deficiencias como ser la obsolescencia, sistemas cerrados (sin código abierto), un proceso complejo de evaluación de similitud, la incapacidad para utilizar o no una gran base de información. Esto se debe a que fueron diseñadas para detectar plagio entre un grupo pequeño de archivos de código fuente.

1.2.3. Formulación del problema

¿El algoritmo JTEL tiene mejor desempeño frente a otras herramientas en la detección de similitud de códigos fuente en trabajos de cátedra?

1.3. Objetivos

1.3.1. Objetivo general

Diseñar e implementar el algoritmo JTEL para la detección de similitud entre códigos fuente.

1.3.2. Objetivos específicos

- Estudiar los métodos para la detección de similitud entre códigos fuente existentes.
- Estudiar los algoritmos utilizados en los métodos para la detección de similitud de código fuente.
- Redactar las especificaciones para el algoritmo JTEL.
- Evaluar el desempeño del algoritmo JTEL realizando las pruebas en trabajos de programación presentado por estudiantes.

1.4. Hipótesis

El algoritmo JTEL implementado para la detección de similitud entre códigos fuente obtiene mejores resultados frente a otras herramientas para la detección de similitud.

1.4.1. Variables Independientes

- El algoritmo JTEL para la detección de similitud entre códigos fuente.

1.4.2. Variables Dependientes

- Resultados más precisos respecto a la detección de similitud entre códigos fuente de trabajos de cátedra frente a otras herramientas.

1.5. Justificaciones

1.5.1. Justificación Social

El algoritmo JTEL ahorrará el tiempo de docentes de instituciones académicas en la detección de plagio en trabajos de programación presentados por estudiantes, evitando que los docentes realicen la comparación de trabajos de programación de forma manual.

1.5.2. Justificación Económica

El algoritmo JTEL para la detección de similitud entre códigos fuente sera de código abierto, por lo cual permitirá que se desarrollen otros software a bajo costo.

1.5.3. Justificación Tecnológica

El algoritmo JTEL para la detección de similitud entre códigos fuente, se puede implementar en jueces de programación para identificar los envíos similares de los usuarios del juez.

1.5.4. Justificación Científica

Con la utilización de trabajos de programación presentados por estudiantes se medirá el desempeño de los métodos existentes para la detección de similitud entre códigos fuente. Con los resultados obtenidos se determinará cuál es el método más eficiente en términos de tiempo de ejecución, espacio de memoria ocupado y precisión.

1.6. Alcances y Limites

1.6.1. Alcance Sustancial

- Se diseñará e implementará en Python un algoritmo JTEL para la detección de similitud entre códigos fuente.
- Se realizaran las pruebas de trabajos de cátedra de código fuente en Python.
- Se dejará de lado la comprobación de correctitud de los códigos fuente.
- Se realizarán pruebas para medir el tiempo, espacio de memoria ocupado y eficiencia del algoritmo.
- Se dejará de lado el estudio de métodos para la detección de similitud que aplican técnicas de aprendizaje automático vinculadas a la lingüística computacional, tales como minería de datos sobre texto y procesamiento del lenguaje natural.

1.6.2. Alcance Espacial

- Se realizaran las pruebas del algoritmo en una computadora intel i3 de Décima generación con 8GB de RAM y 512GB de Disco Duro.

1.6.3. Alcance Temporal

- El tiempo de ejecución que el algoritmo estará limitado a un minuto. Por cada conjunto de trabajos evaluados.

1.7. Metodología

1.7.1. Metodología Experimental

Para el desarrollo del trabajo de investigación se utilizara la metodología científica experimental, esta investigación nos permite la manipulación de una o mas variables. De modo que al seguir las siguientes etapas ayudara a cumplir con los objetivos propuestos.

1. **Recopilación de la información.** En esta etapa se recopilara información necesaria y estudiara los temas.

2. **Diseño del algoritmo.** En esta etapa se diseñara el algoritmo tomando en cuenta los alcances y limites.

3. **Pruebas de funcionamiento del algoritmo en trabajos de cátedra.** En esta etapa se Realizara pruebas en trabajos de cátedra presentados por estudiantes.

4. **Análisis de los resultados obtenidos.** En esta etapa se analizaran los datos y se los comparara frente a otras herramientas.

5. **Conclusiones.** En esta etapa se realizara las conclusiones, se presentaran los resultados finales, y recomendaciones respecto a la investigación.

CAPÍTULO 2: MARCO TEÓRICO

2.1. Conceptos básicos

2.1.1. Lenguaje de programación

Los lenguajes de programación son notaciones que describen los cálculos a las personas y las máquinas. Nuestra percepción del mundo en que vivimos depende de los lenguajes de programación, ya que todo el software que se ejecuta en todas las computadoras se escribió en algún lenguaje de programación [Aho, 2008].

2.1.2. Código fuente

El código fuente de un programa está escrito por un programador en algún lenguaje de programación, pero en este primer estado no es directamente ejecutable por la computadora, sino que debe ser traducido a otro lenguaje o código binario, así será más fácil para la máquina interpretarlo. Para esta traducción se usan los llamados compiladores, ensambladores, intérpretes y otros sistemas de traducción [Wikipedia, 2021c].

2.2. Conceptos de compiladores

2.2.1. Procesador de lenguaje

Un procesador de lenguaje (figura 2.1) también llamado compilador es un programa que puede leer un programa en un lenguaje y traducirlo en un programa equivalente en otro lenguaje. Una función importante del compilador es reportar cualquier error en el programa fuente que detecte durante el proceso de traducción [Aho, 2008]. El proceso de compilación opera como una secuencia de fases con las cuales transforma un programa fuente en otro

Figura 2.1: Compilador

como se muestra en la figura 2.2. En el presente trabajo de investigación, solo es de interés de la investigación cubrir los conceptos y teoría hasta el análisis semántico.

2.2.2. Análisis Léxico

En la primera fase de un compilador. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma:

[nombre-token, valor-atributo]

estos lexemas son los que pasaran a la fase siguiente fase, el análisis de la sintaxis. En el token, el primer componente nombre-token es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el segundo componente valor-atributo apunta a una entrada en la tabla de símbolos para este token. La información de la entrada en la tabla de símbolos se necesita para el análisis semántico y la generación de código. Por ejemplo, suponga que un programa fuente contiene la instrucción de asignación:

posicion = inicial + velocidad * 60

Los caracteres en esta asignación podrían agruparse en los siguientes lexemas y mapearse a los siguientes tokens que se pasan al analizador sintáctico:

1. posicion es un lexema que se asigna a un token [id, 1], en donde id es un símbolo abstracto que representa la palabra identificador y 1 apunta a la entrada en la tabla de símbolos para posicion. La entrada en la tabla de símbolos para un identificador contiene información acerca de éste, como su nombre y tipo.

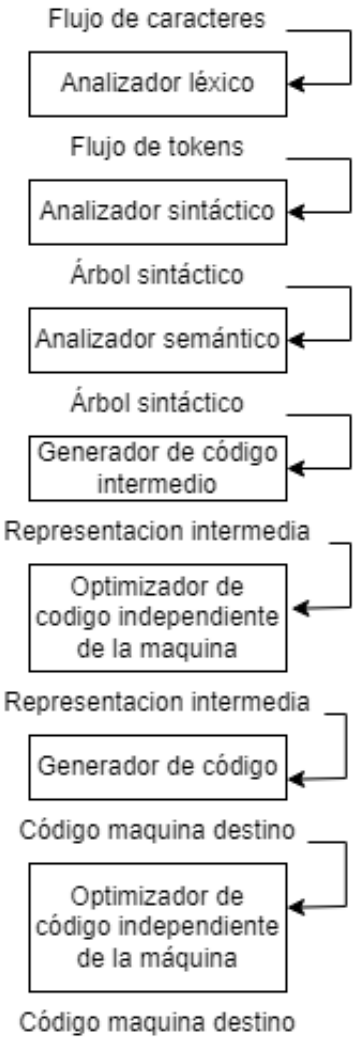

Figura 2.2: Fases de un compilador

2. El símbolo de asignación = es un lexema que se asigna al token [=]. Como este token no necesita un valor-atributo, hemos omitido el segundo componente. Podríamos haber utilizado cualquier símbolo abstracto como asignar para el nombre-token, pero por conveniencia de notación hemos optado por usar el mismo lexema como el nombre para el símbolo abstracto.

3. inicial es un lexema que se asigna al token [id, 2], en donde 2 apunta a la entrada en la tabla de símbolos para inicial.

4. + es un lexema que se asigna al token [+].

5. velocidad es un lexema que se asigna al token [id, 3], en donde 3 apunta a la entrada en la tabla de símbolos para velocidad.

6. * es un lexema que se asigna al token [*].

7. 60 es un lexema que se asigna al token [60].

El analizador léxico ignora los espacios en blanco que separan a los lexemas.

[id, 1] [=] [id, 2] [+] [id, 3] [*] [60]

En esta representación, los nombres de los tokens =, + y * son símbolos abstractos para los operadores de asignación, suma y multiplicación, respectivamente.

2.2.3. Análisis Sintáctico

La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación. En la figura 2.3 se muestra un árbol sintáctico para el flujo de tokens como salida del analizador sintáctico.

2.2.4. Análisis Semántico

El analizador semántico utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio. Una parte importante del análisis semántico es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tenga operando que coincidan. Por ejemplo, muchas definiciones de lenguajes de programación requieren que el índice de un arreglo sea entero; el compilador debe reportar un error si se utiliza un número de punto flotante para indexar el arreglo. Observe en la figura 2.3 que la salida del analizador semántico tiene un nodo adicional para el operador inttofloat

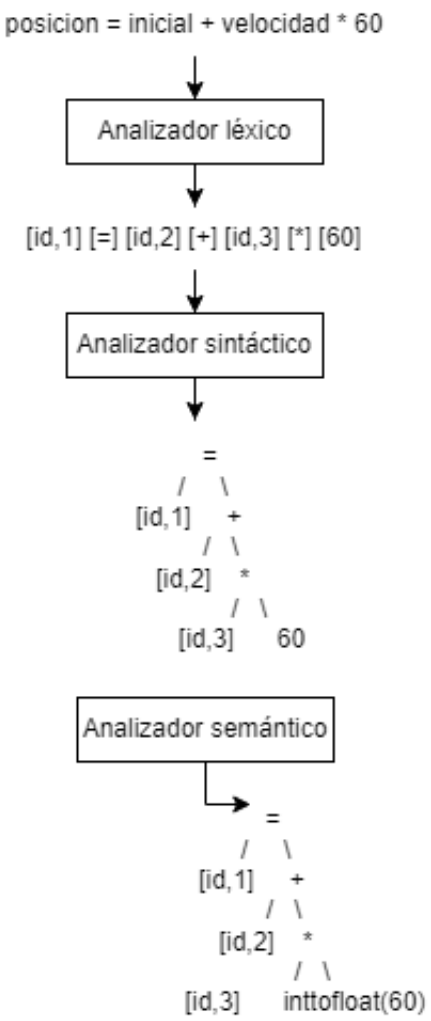

Figura 2.3: Traducción de una asignación

2.3. Representación de código fuente

2.3.1. Arboles

En ciencias de la computación y en informática, un árbol es un tipo abstracto de datos ampliamente usado que imita la estructura jerárquica de un árbol, con un valor en la raíz y subárboles con un nodo padre, representado como un conjunto de nodos enlazados. Una estructura de datos de árbol se puede definir de forma recursiva como una colección de nodos a partir de un nodo raíz, donde cada nodo es una estructura de datos con un valor, junto con una lista de referencias a los nodos, con la condición de que ninguna referencia esté duplicada ni que ningún nodo apunte a la raíz.

2.3.2. Árboles de sintaxis abstracta

Las estructuras de datos basadas en árboles se hicieron muy populares en el campo del desarrollo de compiladores: cada vez que se envía un archivo que contiene el código fuente al compilador, se realizan varios pasos antes de que se puedan generar las instrucciones de la máquina. El código tiene que ser tokenizado por un Lexer primero: separa el flujo de entrada en tokens individuales y los pasa al analizador, que utiliza una gramática libre de contexto del lenguaje de programación para construir una representación de código intermedio, un llamado árbol de análisis. Cada token encontrado por el lexer está representado por un nodo en el árbol de análisis. No todos los tokens (nodos) tienen un valor semántico: algunos tokens, por ejemplo, paréntesis y punto y coma, son puramente sintácticos. Por lo tanto, puede omitirse. La estructura de datos resultante se denomina árbol de sintaxis abstracta (AST). Ahora que el código fuente se representa como un árbol, se puede analizar de una forma más sofisticada manera que al analizar el flujo de token plano: el árbol se puede recorrer o buscar de varias maneras (por ejemplo, pos-orden, pre-orden, búsqueda profundidad, etc.) [Würsch *et al.*, 2022].

2.4. Conceptos de algorítmica

2.4.1. Programación dinámica

La programación dinámica es una técnica matemática que se utiliza para la solución de problemas matemáticos seleccionados, en los cuales se toma un serie de decisiones en forma secuencial.

2.4.2. Distancia de edición

La distancia edición o distancia entre palabras es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra, se usa ampliamente en teoría de la información y ciencias de la computación. Se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter. Esta distancia recibe ese nombre en honor al científico ruso Vladimir Levenshtein, quien se ocupó de esta distancia en 1965. Es útil en programas que determinan cuán similares son dos cadenas de caracteres, como es el caso de

los correctores ortográficos.

2.4.3. Distancia de edición del árbol

La distancia de edición del árbol se define como la secuencia de costos mínimos de operaciones de edición de nodos que transforman un árbol en otro. Las operaciones consisten en eliminar, agregar y modificar nodos del árbol.

2.4.4. Algoritmos utilizados para la detección de similitud entre códigos fuente

[Novak *et al.*, 2019] Identificó diferentes algoritmos a continuación se mencionan algunos de ellos:

- Recuento de atributos
- Huella digital
- Coincidencia de cadena
- Texto base
- Estructura base
- Estilo
- Semántico
- N-gramas
- Árboles
- Grafos

Algunos de estos fueron inventados en la década de 1980. También hace mención a que los enfoques basados en estructuras son mucho mejores y que la mayoría de las herramientas de detección de similitud combinan más de un tipo de algoritmo.

2.5. Conceptos de ofuscación de código fuente

2.5.1. Plagio de código fuente

[Wikipedia, 2021b] La Real Academia Española define como plagio a la acción de copiar en lo sustancial obras ajenas, dándolas como propias.

El plagio de código fuente consiste en utilizar el código fuente de otra persona y adjudicarse como propio.

2.5.2. Ofuscación

La ofuscación se refiere a encubrir el significado de una comunicación haciéndola más confusa y complicada de interpretar [Wikipedia, 2021a].

2.5.3. Ofuscación de código fuente

En computación, la ofuscación se refiere al acto deliberado de realizar un cambio no destructivo, ya sea en el código fuente de un programa informático, en el código intermedio (bytecodes) o en el código máquina cuando el programa está en forma compilada o binaria. Es decir, se cambia el código se enrevesa manteniendo el funcionamiento original, para dificultar su entendimiento. De esta forma se dificultan los intentos de ingeniería inversa y desensamblado que tienen la intención de obtener una forma de código fuente cercana a la forma original [Wikipedia, 2021a].

2.5.4. Métodos de ofuscación de código fuente

Existen muchos métodos de ofuscación utilizados por estudiantes para ocultar la similitud a continuación se mencionan algunos:

- [Marzieh *et al.*, 2011] Mencionan cambios visuales en el formato del código, por lo que parece diferente a primera vista, esto generalmente incluye la modificación de espacios en blanco como sangrías, espacios, nuevas líneas, etc.
- [Marzieh *et al.*, 2011] Mencionan cambios en los comentarios del código.

- [Donaldson *et al.*, 1981] Mencionan el cambio de los nombres de los identificadores. como nombres de variables, nombres de constantes, nombres de funciones, nombres de clases, etc.

- [Donaldson *et al.*, 1981] Mencionan reordenar las líneas del código para las que el pedido no marca ninguna diferencia. Estos incluyen cambiar el orden de las declaraciones de variables, cambiando el orden de las declaraciones dentro de bloques de código como funciones, re ordenación de bloques de código o funciones, re ordenación de clases internas, etc.

2.5.5. Detección léxica

Las técnicas y herramientas para computar las diferencias textuales entre documentos son bien conocidas y aprobadas. Sin embargo, las herramientas existentes como GNU diff tratan con información plana, en lugar de jerárquica. Por lo general, calculan una lista de líneas que deben cambiarse, insertarse o eliminarse para que un primer documento coincida con el segundo. [Würsch *et al.*, 2022].

Bibliografía

[Aho, 2008] Aho, A. (2008). *Compiladores: principios, tecnicas y herramientas*. Pearson Educacion, Mexico.

[Cheers *et al.*, 2021] Cheers, H., Lin, Y., y Smith, S. (2021). Academic source code plagiarism detection by measuring program behavioural similarity.

[Donaldson *et al.*, 1981] Donaldson, J. L., Lancaster, A.-M., y Sposato, P. H. (1981). A plagiarism detection system. pp. 21–25.

[Díaz *et al.*, 2007] Díaz, J., Banchoff, L., y Rodríguez, L. (2007). "herramientas para la detección de plagio de software. un caso de estudio en trabajos de cátedra".

[Hage y Rademaker, 2010] Hage, J. y Rademaker, P. (2010). A comparison of plagiarism detection tools.

[Marzieh *et al.*, 2011] Marzieh, A., Mahmoudabadi, E., y Khodadadi, F. (2011). Pattern of plagiarism in novice students generated program: An experimental approach. *The Journal of Information Technology Education*, 10.

[Novak *et al.*, 2019] Novak, M., Joy, M., y Kermek, D. (2019). Source-code similarity detection and detection tools used in academi: A systematic review. *ACM Trans. Comput. Educ.*, 19(3).

[Pachón, 2019] Pachón, H. (2019). *Generación de un algoritmo para el análisis de similitudes de código fuente en lenguajes Java y Python*. Uniandes.

[Wikipedia, 2021a] Wikipedia (2021a). Obfuscation (software) — Wikipedia, the free encyclopedia. `htt://en.wikiipedia.org/w/index.php?title=Obfuscation%20(software)&oldid=1056756228`. [Online; accessed 02-December-2021].

[Wikipedia, 2021b] Wikipedia (2021b). Plagiarism — Wikipedia, the free encyclopedia. htt://en.wikiipedia.org/w/index.php?title=Plagiarism&oldid=1058139870. [Online; accessed 02-December-2021].

[Wikipedia, 2021c] Wikipedia (2021c). Source Code — Wikipedia, the free encyclopedia. htt://en.wikiipedia.org/w/index.php?title=Source%20Code&oldid=1057866572. [Online; accessed 02-December-2021].

[Würsch *et al.*, 2022] Würsch, M., Gall, H., Fluri, B., y Kiefer, C. (2022). Improving changedistiller improving abstract syntax tree based source code change detection.