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

Índice de figuras

Índice de cuadros

Índice de programas

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 reutilización, extracción de código para refactorización del mismo y seguimiento de defectos para su corrección.

En el ámbito académico, los estudiantes de programación durante su proceso de formación elaboran trabajos, proyectos, tareas, ejercicios, etc. de programación, escritos en algún lenguaje de programación. Cuando estas actividades se realizan de forma individual, sirven para medir la capacidad de resolución de problemas y el enfoque lógico de los estudiantes. Por ello encontrar similitud en trabajos de programación presentados por estudiantes puede ser identificado como plagio.

En la actualidad existen herramientas de software para detectar la similitud entre códigos fuente, en las cuales se aplican diferentes métodos para la detección de similitud, a partir de las características de las herramientas, se pudo evidenciar que presentan deficiencias como: sistemas obsoletos, sistemas sin código abierto, procesos complejos para la evaluación de similitud, sistemas incapaces que procesar un grupo grande de archivos, sistemas de difícil configuración e instalación, sistemas que requieren conexión a internet.

En 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 buen desempeño, en términos de tiempo de ejecución y precisión.

1.2. Problema

1.2.1. Antecedentes del problema

Un trabajo similar al propuesto es realizado por Anzai y Watanobe (2019) en el que presenta un algoritmo para la detección de similitud llamado “Algoritmo para determinar la distancia de edición ampliada entre códigos de programa”. El algoritmo consiste en tres fases, en la primera fase el código dado es dividido en funciones, en la segunda fase cada función es dividida en bloques, en la tercera fase cada bloque es dividido en tokens y otras tareas de preprocesamiento. La distancia entre dos bloques, es calculado por un algoritmo de programación dinámica, la relación entre bloques y funciones de dos códigos fuente es tratado como el problema de Minimum-Cost-Flow.

Para la evaluación del algoritmo, realiza varios experimentos utilizando recursos de un juez de programación en linea. Respecto a la complejidad temporal del algoritmo, se emplea el algoritmo de Bellman-Ford para el calculo de Minimum-Cost-Flow. La complejidad temporal de Bellman-Ford es de $O(V * E)$, donde V es el numero de vértices y E es el numero de aristas, E es igual a V^2 por que el grafo es bipartito y completo, entonces la complejidad temporal de Minimum-Cost-Flow es $O(V^3)$. La complejidad temporal para calcular la distancia de edición extendida entre bloques es $O(L^2)$ donde L es el numero de tokens. La complejidad temporal de dividir las funciones en bloques y realizar el calculo de la distancia entre bloques es $O(M^2 * L^2 + M^4)$ donde M es el numero de bloques. La complejidad temporal para dividir el bloque en funciones y realizar calculo de la distancia entre códigos fuente es $O(N^2 * M^2 + N^2 * M^4 + N^2)$ donde N es el numero de funciones.

Otro trabajo similar es realizado por Popescu y Nicolae (2016), el cual presenta un método para medir la similitud entre paginas web, el método utiliza un algoritmo para calcular la distancia de edición, el valor calculado es usado como métrica de similitud. El algoritmo utiliza las etiquetas HTML de las paginas web para realizar en las operaciones de edición, las operaciones en las etiquetas consisten en eliminar, insertar y reemplazar. Es decir, dados dos archivos HTML la forma de calcular que tan diferentes es contando el numero mínimo de operaciones en las etiquetas, para transformar un archivo en otro.

1.2.2. Planteamiento del problema

Cheers *et al.* (2021) Explica que la identificación de similitud entre códigos fuente, puede servir para varios propósitos como: El estudio de la evolución de código fuente de un proyecto, la detección de prácticas de plagio en el ámbito académico, la detección de prácticas de reutilización, la extracción de código para refactorización del mismo y el seguimiento de defectos para su corrección.

Una manera de ocultar la similitud entre códigos fuente, es mediante la aplicación de métodos de ofuscación. Estos métodos consisten en aplicar cambios léxicos o estructurales al código fuente, con el propósito de que no sean identificados como similares.

Por lo cual contar un algoritmo que calcule de forma eficiente la similitud entre códigos fuente, tomando en cuenta los métodos de ofuscación, es de gran utilidad.

1.2.3. Formulación del problema

¿El algoritmo SCAM tiene mejor desempeño frente a los algoritmos de Winnowing-Fingerprint y Greedy-String-Tiling en la detección de similitud entre códigos fuente?

1.3. Objetivos

1.3.1. Objetivo general

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

1.3.2. Objetivos específicos

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

1.4. Hipótesis

El algoritmo SCAM para la detección de similitud entre códigos fuente obtiene mejores resultados frente a otros algoritmos en la detección de similitud.

1.4.1. Variables Independientes

- El diseño del algoritmo SCAM para la detección de similitud entre códigos fuente.

1.4.2. Variables Dependientes

- Resultados más precisos frente a otros algoritmos para la detección de similitud entre códigos fuente.

1.5. Justificaciones

1.5.1. Justificación Social

En el ámbito académico, los estudiantes de programación durante su proceso de formación elaboran trabajos, proyectos, tareas, ejercicios, etc. de programación, escritos en algún lenguaje de

programación. El algoritmo SCAM ayudara a docentes de instituciones académicas, en la detección de trabajos similares de programación presentados por estudiantes.

1.5.2. Justificación Económica

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

1.5.3. Justificación Tecnológica

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

1.5.4. Justificación Científica

Con la utilización de trabajos de programación presentados por estudiantes se medirá el desempeño de los algoritmos para la detección de similitud entre códigos fuente. Con los resultados obtenidos se determinará cuál algoritmo es más eficiente en términos de tiempo de ejecución y precisión.

1.6. Alcances y Limites

1.6.1. Alcance Sustancial

- Se diseñará e implementará en Python el algoritmo SCAM para la detección de similitud entre códigos fuente.
- Se realizaran las pruebas de similitud con trabajos de programación presentados por estudiantes.
- 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 inteligencia artificial.

1.6.2. Alcance Espacial

- La implementación y las pruebas del algoritmo, se realizaran en una computadora intel i3 de décima generación con 8GB de RAM y 512GB de disco duro.

- La procedencia de los trabajos de programación para las pruebas, son archivos de código fuente enviados por usuarios de un juez de programación.

1.6.3. Alcance Temporal

- En la presente investigación se realizaran las pruebas con archivos de código fuente que fueron presentados al juez desde su despliegue hasta la fecha.

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. Las siguientes etapas ayudaran a cumplir con los objetivos propuestos.

1. **Recopilación de la información.** En esta etapa se recopilara información necesaria y se estudiara los temas relacionados al diseño del algoritmo.

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

3. **Implementación del algoritmo.** En esta etapa se implementara el algoritmo.

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

5. **Análisis de los resultados obtenidos.** En esta etapa se analizaran los datos obtenidos y se realizara la comparación con los datos obtenidos por otros algoritmos.

6. **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 de lenguajes 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, p. 1). Un lenguaje de programación es un lenguaje formal, que mediante un conjunto de instrucciones permite a un programador crear programas. El lenguaje de programación es un sistema estructurado de comunicación conformado por conjuntos de símbolos, palabras clave, reglas semánticas y sintácticas, las cuales sirven para el entendimiento entre un programador y una maquina.

- **Palabra clave.** En los lenguajes de programacion existen palabras clave. Estas palabras no se pueden ser utilizadas para ningun otro proposito.
- **Funciones.** Tambien conocidos como subprogramas, procedimientos o metodos, las funciones son segmentos de codigo separado del bloque principal.
- **Tipos de datos.** En los lenguajes de programacion los tipos de datos son variados, los tipos de datos son atributos que indica la clase de dato que se va manejar. Los datos mas comunes son: Numeros enteros, numeros en coma flotante y cadenas.
- **Operadores.** Son simbolos que indican como manipular los operandos. Los operadores mas comunes son: Aritmeticos, relacionales, asignacion, logicos y tratamiento de bits.
- **Comentarios.** Son secuencias de caracteres que sirven para realizar anotaciones en el codigo fuente.

2.2. Conceptos de ofuscación de código fuente

2.2.1. Código fuente

En informática, se denomina código fuente a la conjunto de lineas de texto que escritas por un programador. Estas lineas de texto representan instrucciones en un lenguaje de programación. Las instrucciones representan los pasos que debe seguir la computadora para la ejecución de un programa específico. El código fuente no es directamente ejecutable por la computadora, este debe ser traducido a otro lenguaje de modo que la computadora pueda interpretarlo. En la traducción se usan compiladores, ensambladores, interpretes y otros.

2.2.2. Ofuscación de código fuente

En computación, la ofuscación de código fuente se refiere al acto de realizar cambios no destructivos en código fuente de un programa. Es decir, se alteran las instrucciones del código fuente manteniendo su funcionamiento original, la ofuscación de un programa se realiza para dificultar su entendimiento, también puede ser utilizado para ocultar la similitud con el programa original. Un ejemplo se muestra en la figura 2.1.

Figura 2.1: Ofuscación de código fuente
Fuente: Elaboración propia.

Una definición formal se encuentra en Collberg *et al.* (1997) que define a la ofuscación de código fuente de la siguiente manera: Dado un programa P, y el programa transformado P'. Se define a T como la transformación de ofuscación como $P \xrightarrow{T} P'$, donde requiere que P y P' mantengan el mismo comportamiento observacional, Ademas si P no puede terminar o termina con errores, entonces P' puede terminar o no, y P' debe terminar si P termina.

2.2.3. Métodos de ofuscación de código fuente

Novak *et al.* (2019) Explica que existen muchos métodos de ofuscación son utilizados para ocultar la similitud, a su vez menciona que en un estudio de 72 artículos se identificaron 25 métodos de ofuscación, y de estos se especificaron 16 métodos distintos. Existen muchos métodos para la ofuscación, a continuación se presentan los métodos mas importantes, cada método tiene su ejemplo de ofuscación en el lenguaje de programacion Python.

- Marzieh *et al.* (2011) Mencionan cambios en el formato del código. Como la agregación o eliminación de: Espacios en blanco, sangrías y saltos de linea. El ejemplo se muestra en los programas 2.1 y 2.2.

Programa 2.1: Cambio del formato del código, original

```
1 a = 5
2 b = 4
3 area = a * b
4 perimetro = 2 * a + 2 * b
5 print(area, perimetro)
```

Programa 2.2: Cambio del formato del código, modificado

```
1 a=5
2 b=4
3 area=a*b
4 perimetro=2*a+2*b
5 print(area,perimetro)
```

- Marzieh *et al.* (2011) Mencionan cambios en los comentarios del código. Como la agregación, modificación o eliminación de los comentarios. El ejemplo se muestra en los programas 2.3 y 2.4.

Programa 2.3: Cambios de comentarios, original

```
1  """ Area y perimetro de un
2  rectangulo """
3  a = 5 # lado A
4  b = 4 # lado B
5  # area de un rectangulo
6  area = a * b
7  # perimetro de un rectangulo
8  perimetro = 2 * a + 2 * b
9  # salida
10 print(area, perimetro)
```

Programa 2.4: Cambios de comentarios, modificado

```
1 """ Rectangulo """
2 a = 5 # lado 1
3 b = 4 # lado 2
4 # area
5 area = a * b
6 # perimetro
7 perimetro = 2 * a + 2 * b
8 # salida de datos
9 print(area, perimetro)
```

- Đurić y Gašević (2012) y Donaldson *et al.* (1981) Mencionan el cambio de los nombres de los identificadores. Es decir, cambios en los nombres de variables, nombres de constantes, nombres de funciones, nombres de clases, etc. El ejemplo se muestra en los programas 2.5 y 2.6.

Programa 2.5: Cambio en los nombres de las variables, original

```
1 a = 5
```

```
2 b = 4
3 area = a * b
4 perimetro = 2 * a + 2 * b
5 print(area, perimetro)
```

Programa 2.6: Cambio en los nombres de las variables, modificado

```
1 s_a = 5
2 s_b = 4
3 s_ar = s_a * s_b
4 s_per = 2 * s_a + 2 * s_b
5 print(s_ar, s_perimetro)
```

- Donaldson *et al.* (1981) Menciona cambios en el orden en las declaraciones de las variables. El ejemplo se muestra en los programas 2.7 y 2.8.

Programa 2.7: Cambio en el orden de declaraciones de variables, original

```
1 a = 5
2 b = 4
3 area = a * b
4 perimetro = 2 * a + 2 * b
5 print(area, perimetro)
```

Programa 2.8: Cambio en el orden de declaraciones de variables, modificado

```
1 b = 4
2 a = 5
3 perimetro = 2 * a + 2 * b
4 area = a * b
5 print(area, perimetro)
```

- Grier (1981) Menciona agregar lineas de código innecesarias. El ejemplo se muestra en los programas 2.9 y 2.10.

Programa 2.9: Agregar instrucciones innecesarias, original

```
1 a = 5
2 b = 4
3 area = a * b
4 perimetro = 2 * a + 2 * b
5 print(area, perimetro)
```

Programa 2.10: Agregar instrucciones innecesarias, modificado

```
1 a = 5
2 b = 4
3 var_aux = 100
4 area = a * b
5 perimetro = 2 * a + 2 * b
6 diagonal = (a * a + b * b) ** 0.5
7 print(area, perimetro)
```

- Donaldson *et al.* (1981) Menciona dividir una linea de código en varias lineas. El ejemplo se muestra en los programas 2.11 y 2.12.

Programa 2.11: Dividir una instruccion, original

```
1 a,b = 5,4
2 area,perimetro = a*b, 2*a + 2*b
3 print(area, perimetro)
```

Programa 2.12: Dividir una instruccion, modificado

```
1 a = 5
2 b = 4
3 area = a * b
4 perimetro = 2 * a + 2 * b
5 print(area, perimetro)
```

- Whale (1990) Menciona el reemplazo de la llamada de un procedimiento por el procedimiento. El ejemplo se muestra en los programas 2.13 y 2.14.

Programa 2.13: Reemplazo de la llamada a un procedimiento, original

```
1 def calcular(a, b):
2     area = a * b
3     perimetro = 2 * a + 2 * b
4     print(area, perimetro)
5 a = 5
6 b = 4
7 calcular(a, b)
```

Programa 2.14: Reemplazo de la llamada a un procedimiento, modificado

```
1 a = 5
2 b = 4
3 area = a * b
4 perimetro = 2 * a + 2 * b
5 print(area, perimetro)
```

- Whale (1990) Menciona el cambio de la especificación de una declaración. Cambios como: El cambio de las operaciones y el operando, cambio en los tipos de datos. El ejemplo se muestra en los programas 2.15 y 2.16.

Programa 2.15: Cambio de operaciones y operandos, original

```
1 a = 5
2 b = 4
3 area = a * b
4 perimetro = 2 * a + 2 * b
5 print(area, perimetro)
```

Programa 2.16: Cambio de operaciones y operandos, modificado

```
1 a = 5
2 b = 4
3 area = a
4 area *= b
5 perimetro = 2 * (a + b)
6 print(area, perimetro)
```

- Marzieh *et al.* (2011) Mencionan el cambio de estructuras de control por sus equivalentes. El reemplazo por equivalentes de estructuras repetitivas y condicionales, El ejemplo se muestra en los programas 2.17 y 2.18.

Programa 2.17: Cambio en la estructura repetitiva, original

```
1 a = 0
2 for i in range(10):
3     a = a + i
4 print(a)
```

Programa 2.18: Cambio en la estructura repetitiva, modificado

```
1 a = 0
2 i = 0
3 while i < 10:
4     a = a + i
5     i = i + 1
6 print(a)
```

Bejarano *et al.* (2015) se refiere a los métodos como modificaciones y los divide en dos grupos:

- **Cambios léxicos**. Estos no requieren un análisis gramatical o profundo conocimiento de programación para ser eficaces. Algunos ejemplos son: Eliminar comentarios, cambiar el formato de código fuente y cambiar los nombres de las variables.
- **Cambios estructurales**. Estos requiere conocimiento acerca de los lenguajes y técnicas de programación, estos cambios son altamente dependiente de el lenguaje de programación. Algunos ejemplos son: cambiar las estructuras de control, cambiar el orden de las sentencias, reemplazar la llamada a un procedimiento por el procedimiento, etc.

2.2.4. Niveles de transformación

Una de las investigación mas antiguas respecto a la ofuscación de código fuente, fue desarrollado por Faidhi y Robinson (1987) en su trabajo se refiere a los métodos utilizados para las ofuscación como transformaciones para ocultar la similitud. Explica que las transformaciones pueden ser divididas en niveles. Donde en niveles bajos se encuentran las transformaciones que realiza un programador novato para ocultar la similitud, y los niveles altos se encuentran las transformaciones que realiza un programador experto para ocultar la similitud. A continuación el detalle de estos niveles de transformación.

- **Nivel 1.** Representa los cambios en los comentarios e indentación.
- **Nivel 2.** Representa cambios de nivel 1, y cambios en los identificadores.
- **Nivel 3.** Representa cambios de nivel 2, y cambios en las declaraciones. Es decir declarar constantes extras, cambiar las posiciones de las variables declaradas.
- **Nivel 4.** Representa cambios de nivel 3, y modificación de los métodos. Es decir cambios en las asignaciones de la funciones, cambiar de funciones por procedimientos, combinar y crear nuevas funciones.
- **Nivel 5.** Representa cambios de nivel 4. y cambios en las sentencias equivalentes. Es decir cambios en las estructuras de control equivalentes un for por un while.
- **Nivel 6.** Representa cambios de nivel 5, y cambios en las decisiones lógica. Es decir cambios en la expresiones.

2.3. Detección de similitud de código fuente

Novak *et al.* (2019) Realizo un estudio sistemático campo de la detección de plagio de código fuente, describe definiciones de plagio, herramientas para la detección de plagio, métricas

de comparación, métodos de ofuscación, conjunto de datos para la comparación y algoritmos que utilizan las herramientas.

2.3.1. Algoritmos para la detección de similitud

En su estudio Novak *et al.* (2019) identifica algoritmos basados en estilo, basado en semántica, basado en texto, huellas dactilares, recuento de atributos, basado en estructura, coincidencia de cadenas, marca de agua, basado en historial, basado en XML, código compilado, basado en compresión, basado en grafos, basado en agrupamiento, basado en N-gramas y basado en árboles. También hace menciona que los enfoques basados en estructuras son mucho mejores y que la mayoría de las herramientas combinan más de un tipo de algoritmo. En el cuadro 2.1 se muestra detalles de estos como el primer y ultimo año de publicación de un algoritmo, el numero de artículos en los que aparecen y si utilizan la tokenización.

Tipo de algoritmo	Ultimo año	Primer año	Nro. de artículos	Tokenización
Basado en estilo	2016	2011	5	2
Basado en semántica	2013	2010	7	5
Basado en texto	2016	1996	9	5
Huellas dactilares	2015	2005	11	4
Recuento de atributos	2015	1980	25	6
Basado en estructura	2016	1980	25	13
Coincidencia de cadenas	2016	1981	26	17
Nuevas categorías identificadas				
Marca de agua	2013	2005	2	0
Basado en historial	2016	2013	2	0
Basado en XML	2012	2010	3	2
Código compilado	2015	2006	5	2
Basado en compresión	2010	2004	6	5
Basado en grafos	2015	2005	10	2
Basado en agrupamiento	2015	2005	11	7
Basado en N-gramas	2016	2006	15	9
Basado en árboles	2015	1988	24	8

Cuadro 2.1: Descripción general de los tipos de algoritmos

Fuente: Novak *et al.* (2019).

Novak *et al.* (2019) Menciona que las tres herramientas principales para la detección de similitud, utilizan los algoritmos de Running-Karp-Rabin Greedy-String-Tiling (RKR GST), Winnowing-Fingerprint e implementaciones de tokenización.

2.3.2. Técnicas para la detección de similitud

En su investigación Karnalim y Simon (2019) clasifica las técnicas de detección de similitud en tres categorías: Basadas en conteo de atributos, basadas en estructuras y técnicas híbridas, a continuación se dará una breve descripción de cada una de ellas.

1. **Técnicas basadas en conteo de atributos**. Estas técnicas determinan la similitud comparando las frecuencias de ocurrencias de los atributos del código fuente.

    a) **Técnica estándar de conteo de atributos**. Esta técnica considera similares a dos archivos de código fuente si sus frecuencias de ocurrencias de sus atributos son las mismas. Como los operandos, operadores, espacios en blanco, numero de lineas, comentarios, declaraciones y otros. Uno de los primeros trabajos presentados fue por Ottenstein (1976) en el cual utiliza cuatro métricas de software: El numero de operadores únicos, operandos únicos, operadores y operandos.

    b) **Técnicas basadas en recuperación de información**. Esta técnica se basa en realizar consultas especificas a una gran colección de documentos, Una consulta especifica es un segmento de código o archivo y la colección de documentos son los archivos cuyo contenido es similar al segmento o archivo inicial. Las técnicas basadas en recuperacion de la información (IR) se basan en el análisis semántico latente (LSA), y tiene como objetivo encontrar relaciones entre términos con la ayuda de descomposición en valores singulares. Cosma y Joy (2012) utilizaron LSA para detectar similitud en trabajos realizados por estudiantes obteniendo buenos resultados, sugiriendo que la técnica puede mejorar el rendimiento en las herramientas de detección existente como JPLAG y Sherlock.

    c) **Técnicas basadas en agrupamiento**. En esta técnica los archivos de código fuente similares se muestran en grupos. Moussiades y Vakali (2005) fue el primero en utilizar la técnica que consiste en convertir los archivos en tokens y luego los agrupa utilizando un algoritmo de agrupación.

    d) **Técnicas basadas en clasificación**. Esta técnica aprende a buscar patrones de similitud de código fuente. Yasaswi *et al.* (2017) utilizo su algoritmo de clasificación para

detectar la similitud ponderando las características, según el modelo de un lenguaje a nivel de caracteres, el cual se entreno en el código fuente del kernel de Linux.

e) **Técnicas combinadas con conteo de atributos**. Varias técnicas de conteo de atributos se combinan con otras técnicas para mejorar la detección. En su trabajo Sidorov *et al.* (2017) combina técnicas basadas en recuperacion de información y clasificación, su técnica consiste en convertir los archivos en tokens de N-gramas sobre los cuales se realizara el análisis semántico latente.

2. **Técnicas basadas en estructuras**. Estas técnicas comparan estructuras de dos códigos fuente para determinar su similitud.

   a) **Técnicas basadas en emparejamiento de cadenas**. Esta técnica es de las mas antiguas y populares, En su investigación Wise (1992) compara dos archivos de código fuente convirtiéndolos en cadenas de tokens, y para la medición de similitud usa el comando sdiff de UNIX.

   b) **Técnicas basadas en emparejamiento de arboles y grafos**. La comparación de arboles o grafos puede llegar a tomar un tiempo considerable, por lo cual esta técnica suele incorporar simplificaciones que reduzcan el tiempo. En su investigación Song *et al.* (2015) calcula la similitud de código fuente de dos programas. Utilizando la información sintáctica del código fuente expresado como un árbol de análisis, la similitud sintáctica entre dos programas se calcula mediante un núcleo de árbol de análisis.

3. **Técnicas híbridas** Estas técnicas combinan las técnicas de recuento de atributos, técnicas basadas en estructuras y otras. con el fin de mejorar la eficacia y eficiencia, para la comparación de similitud entre códigos fuente.

2.3.3. Herramientas para la detección de similitud

Novak *et al.* (2019) describe cuatro características de cinco herramientas que son consideradas las más importantes, en el cuadro 2.2 se muestran las características como: las menciones en artículos, código abierto, interfaz grafica (GUI), sin conexión a internet y el sitio web.

Herramienta	Menciones	Código abierto	GUI	offline	Sitio web
JPLAG	43	Si	Si	Si	jplag.ipd.kit.edu
MOSS	38	No	Si	No	theory.stanford.edu
Sherlock	9	Si	Si	Si	warwick.ac.uk
Plaggie	7	Si	Si	Si	www.cs.hut.fi
SIM	6	Si	No	Si	dickgrune.com

Cuadro 2.2: Descripción general de las características de las herramientas
Fuente: Novak *et al.* (2019).

Ragkhitwetsagul *et al.* (2018) Realizo un estudio sobre métodos y herramientas para la detección de similitud, en el cual identifica las medidas de similitud que utilizan las cinco herramientas mas populares. En el cuadro 2.3 se muestra los detalles de las medidas de similitud que utilizan las cinco herramientas más populares.

Herramientas	Medida de similitud utilizadas
JPLAG	Tokens y Greedy-String-Tiling
MOSS	Winnowing-Fingerprint
Sherlock	Firmas Digitales
Plaggie	Token-Tiling
SIM	Alineamiento de cadenas

Cuadro 2.3: Herramientas con sus medidas de similitud
Fuente: Ragkhitwetsagul *et al.* (2018).

JPLAG

Prechelt y Malpohl (2003) Explica que JPLAG es un servicio web que encuentra programas similares entre un conjunto de programas. Se ha utilizado con éxito para detectar plagio entre los envíos de programas Java de los estudiantes. Está disponible para los lenguajes C, C++ y Scheme, su algoritmo de comparación, se basa en uno conocido como Running-Rabin-Karp Greedy-String-Tiling. Cheers *et al.* (2021) Explica que JPLAG opera aplicando un algoritmo Token-Tiling para cubrir un archivo de código fuente con tokens extraídos de otro. Si dos archivos fuente tienen un alto grado de cobertura, pueden considerarse similares y, por lo tanto, candidatos a plagio. Primero, los archivos de código fuente se convierten en un flujo de tokens. JPlag utiliza su propio conjunto de tokens que abstraen los tokens de lenguaje estándar para evitar hacer coincidir el

mismo token con diferentes significados. En segundo lugar, los tokens extraídos se comparan entre archivos para determinar la similitud mediante el algoritmo Running-Karp-Rabin Greedy-String-Tiling, donde los tokens de un archivo se superponen a los de otro dentro de una tolerancia de desajuste. La similitud del programa se evalúa como el porcentaje de tokens de un programa que se pueden colocar sobre otro programa.

MOSS

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

Sherlock

Cheers *et al.* (2021) Explica que Sherlock implementa métodos de comparación de texto y tokenizados. En la herramienta, se compara la similitud de un par de programas 5 veces: en su forma original, se eliminan los espacios en blanco y los comentarios, como un archivo fuente tokenizado. En todos los casos, las comparaciones miden la similitud a través de la identificación de ejecuciones, una secuencia de líneas comunes a dos archivos que pueden verse interrumpidas por anomalías, como líneas adicionales.

Plaggie

Ahtiainen *et al.* (2006) Explica que Plaggie es una aplicación Java independiente que se puede utilizar para comprobar ejercicios de programación Java. La funcionalidad de Plaggie es similar al servicio web JPlag publicado anteriormente, pero a diferencia de JPlag, Plaggie debe instalarse localmente y su código fuente está abierto. Aparentemente, Plaggie es el único motor de detección de plagio de código abierto para ejercicios de Java. Cheers *et al.* (2021) Explica que Plaggie es una herramienta que se afirma que funciona de manera similar a JPlag. Plaggie es una aplicación local en comparación con JPlag que originalmente se proporcionó como un servicio web, no existe una publicación que describa el funcionamiento de Plaggie, sin embargo, al examinar su código fuente, opera sobre representaciones tokenizadas de código que evalúan la similitud mediante Token-Tiling.

SIM

Cheers *et al.* (2021) Explica que SIM analiza programas en busca de similitud estructural mediante el uso de alineación de cadenas. Para dos programas, SIM primero analiza el código fuente creando un árbol de análisis. Luego, la herramienta representará los árboles de análisis como cadenas y los alineará insertando espacios para obtener una subsecuencia común máxima de sus tokens contenidos. La similitud de los programas se evalúa luego como el número de coincidencias.

2.4. Conceptos de compiladores y tokens

2.4.1. Procesador de lenguaje

Un procesador de lenguaje también llamado compilador. Aho (2008) Explica que un procesador de lenguaje 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.

2.4.2. Análisis Léxico

Una analizador lexico tambien es conocido como Lexer. Aho (2008) Explica que la primera fase de un procesador de lenguaje, se le llama análisis léxico o escaneo. El analizador 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. Estos lexemas son los que pasaran a la siguiente fase, el análisis de la sintaxis. El analizador léxico ignora los espacios en blanco que separan los lexemas. Catalán (2010) Explica que esta fase consiste en leer el texto del código fuente carácter a carácter e ir generando los tokens. Estos tokens constituyen la entrada para el siguiente proceso de análisis. El agrupamiento de caracteres en tokens depende del lenguaje que se va a compilar. Es decir un lenguaje generalmente agrupara caracteres en tokens diferentes de otro lenguaje. Los tokens pueden ser de dos tipos, cadenas específicas como palabras reservadas, puntos y comas, etc., y no específicas, como identificadores, constantes y etiquetas. La diferencia entre ambos tipos de tokens radica en si ya son conocidos previamente o no. El analizador léxico irá ignorando las partes no esenciales para la siguiente fase, como pueden ser los espacios en blanco, los comentarios, etc., es decir, realiza la función de preprocesador en cierta medida. Por lo tanto, y resumiendo, el analizador léxico lee los caracteres que componen el texto del programa fuente y suministra tokens al analizador sintáctico.

2.4.3. Tokenización de codigo fuente

Los tokens son la unidad léxica basica, y los lexemas son las palabras de un codigo fuente. Aho (2008) Explica los pasos para generar un token. Primeramente se leen los lexemas que componen del código fuente y los agrupa en categorías según su función, este proceso es llamado tokenización. En un lenguaje de programación un token puede tener en clases como: constantes, identificadores, operadores, palabras reservadas y separadores. Por ejemplo, suponga que un código fuente contiene la instrucción de asignación: $posicion = inicial + velocidad * 60$. En este ejemplo se ignora los espacios en blanco que separan a los lexemas, y los nombres de los lexemas =, + y * serán considerados como símbolos abstractos. A continuación se muestra el detalle de la tokenización de la instrucción:

- *posicion*: es un lexema, se le asigna al 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.
- =: El símbolo de asignación es un lexema, se al asigna el token [=]. Como este token no necesita un valor-atributo, se omite el segundo componente.
- *inicial*: es un lexema, se le asigna al token $[id, 2]$, en donde 2 apunta a la entrada en la tabla de símbolos para *inicial*.
- +: es un lexema, se le asigna al token [+].
- *velocidad*: es un lexema, se le asigna al token $[id, 3]$, en donde 3 apunta a la entrada en la tabla de símbolos para *velocidad*.
- $*$: es un lexema, se le asigna al token $[*]$.
- 60: es un lexema, se le asigna al token [60].

Finalmente se obtiene los siguientes tokens: $[id, 1][=][id, 2][+][id, 3][*][60]$.

2.5. Conceptos de grafos y algoritmos

2.5.1. Grafo direccionado

En Cormen *et al.* (2009) encontramos la siguiente definición. Un grafo direccionado G es un par (V, E), donde V es un conjunto finito y E es una relación binaria en V. El conjunto V

es llamado conjunto de vértices de G, y sus elementos son llamados vértices. El conjunto E es llamado conjunto de aristas de G, y sus elementos son llamados aristas, como se muestra en la figura 2.2.

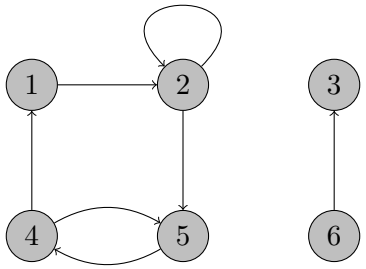


Figura 2.2: Grafo direccionado
Un grafo direccionado $G = (V, E)$, donde $V = \{1, 2, 3, 4, 5, 6\}$ y $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (4, 5), (6, 3)\}$.
Fuente: Cormen *et al.* (2009).

2.5.2. Grafo no direccionado

En Cormen *et al.* (2009) encontramos la siguiente definición. Un grafo no direccionado $G = (V, E)$, el conjunto de aristas E consiste en pares no ordenados de vértices, en lugar de pares ordenados. Es decir una arista es un conjunto (u, v) donde $u, v \in V$ y $u \neq v$, considerando que (u, v) y (v, u) son la misma arista, como se muestra en la figura 2.3.

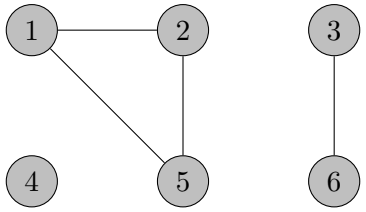


Figura 2.3: Grafo no direccionado
Un grafo no direccionado $G = (V, E)$, donde $V = \{1, 2, 3, 4, 5, 6\}$ y $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$.
Fuente: Cormen *et al.* (2009).

2.5.3. Grafo bipartito

En Cormen *et al.* (2009) encontramos la siguiente definición. Un grafo bipartito es un grafo no direccionado $G = (V, E)$ en el cual V puede ser particionado en dos conjuntos V_1 y V_2 talque $(u, v) \in E$ implica que $u \in V_1$ y $v \in V_2$ o $v \in V_1$ y $u \in V_2$. Es decir todas las aristas relacionan los conjuntos V_1 y V_2, como se muestra en la figura 2.4.

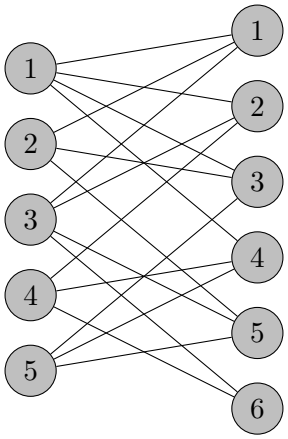


Figura 2.4: Grafo bipartito
Un grafo bipartito con un numero impar de vértices.
Fuente: Cormen *et al.* (2009).

2.5.4. Lista de adyacencia

Cormen *et al.* (2009) explica que la representación de grafo $G = (V, E)$ mediante listas de adyacencia, consiste en arreglo Adj de $|V|$ listas, una para cada vértice en V. Para cada $u \in V$, la lista de adyacencia $Adj[u]$ contiene todos los vértices v talque hay una arista $(u, v) \in E$. Es decir, $Adj[u]$ consiste en todos los vértices adyacentes de u en G, un ejemplo se muestra en la figura 2.5.

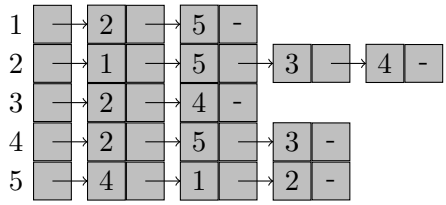


Figura 2.5: Ejemplo de la lista de adyacencia de un grafo
Para un grafo $G = (V, E)$, donde
$V = \{(1,2), (1,5), (2,1), (2,5), (2,3), (2,4), (3,2), (3,4), (4,2), (4,5), (4,3), (5,4), (5,1), (5,2)\}$.
Fuente: Cormen *et al.* (2009).

2.5.5. Búsqueda en profundidad

La Busqueda en profundidad tambien es conocida por sus siglas en ingles como DFS. Halim y Halim (2019) Explica que la búsqueda en profundidad, es un algoritmo sencillo para el recorrido de grafos. La búsqueda en profundidad comienza en un vértice de origen, recorre primero el grafo en profundidad, si durante el recorrido se encuentra con un vértice que tiene mas de un vecino, elige a uno de los vecinos no visitados y visitara su vértice. La búsqueda en

profundidad repite este proceso en profundidad hasta que llega a un vértice terminal, cuando esto ocurre vuelve hacia atrás y explora otros vecinos todavía no visitados. Este comportamiento es de fácil implementación mediante recusión, un ejemplo de este recorrido se muestra en la figura 2.6.

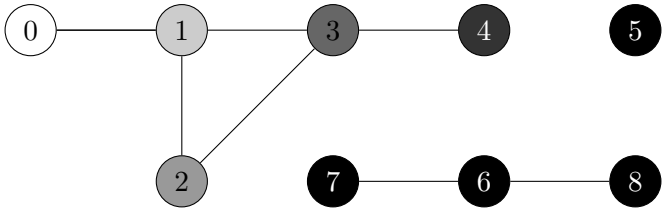


Figura 2.6: Ejemplo de búsqueda en profundidad
Dado un grafo conexo no direccionado, se realiza la búsqueda en profundidad en $s = 0$, obteniendo el siguiente recorrido $0 \to 1 \to 2 \to 3 \to 4$.
Fuente: Halim y Halim (2019).

En la implementación de la búsqueda en profundidad. Para cada vértice $v \in V$, si v fue visitado entonces $v.vis$ tendrá el valor de *VISITED*, caso contrario $v.vis$ tendrá el valor de *UNVISITED*. Es decir se marca con etiquetas a los vértices que fueron visitados o no visitados. La implementación en pseudocódigo del algoritmo se encuentra en DFS(G, s).

DFS(G, s)
1 For each vertex $v \in V$
2 $\quad v.vis = UNVISITED$
3 DFS-Visit(G, s)

DFS-Visit(G, u)
1 $u.vis = VISITED$
2 **for** each vertex $v \in G.Adj[u]$
3 $\quad$ **if** $v.vis == UNVISITED$
4 $\quad\quad$ DFS-Visit(G, v)

La complejidad temporal de la búsqueda en profundidad en un grafo que esta representado mediante una lista de adyacencia es $O(V + E)$.

2.5.6. Búsqueda en anchura

La busqueda en anchura tambien es conocida por sus siglas en ingles como BFS. Halim y Halim (2019) explica que la busqueda en anchura es un algoritmo sencillo para el recorrido de grafos. Comenzando en un vértice de origen, recorre el grafo en anchura, visita todos los vértices vecinos del vértice de origen, seguido los vértices vecinos de los vecinos, y así sucesivamente. Es

decir capa a capa. La búsqueda en anchura comienza con todos los vértices del grafo marcados como no visitados, inserta el vértice de origen s en una cola Q y marca a s como visitado, luego mientras la cola no se encuentre vacía realiza el siguiente proceso: Quita el vértice u que se encuentra en frente de la cola, para cada vértice vecino de u que no haya sido visitado lo agrega a la cola y lo marca como visitado. Cuando la cola se encuentre vacía el algoritmo finaliza y se habrá recorrido todos los vértices del componente conexo que contiene a s, un ejemplo de este recorrido se muestra en la figura 2.7.

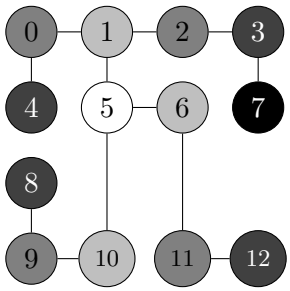


Figura 2.7: Ejemplo de búsqueda en anchura
Dado un grafo conexo no direccionado, se realiza la búsqueda en anchura en $s = 5$, obteniendo el siguiente el recorrido Capa 0: 5; Capa 1: 1,6,10; Capa 2: 0,2,11,9; Capa 3: 4,3,12,8 y Capa 4: 7.
Fuente: Halim y Halim (2019).

En la implementación de la búsqueda en anchura. Para cada vértice $v \in V$, si v no fue visitado entonces $v.dis$ tendrá el valor igual a ∞, caso contrario $v.dis$ tendrá el valor de la distancia entre el vértice origen s y el vértice v. La implementación en pseudocódigo del algoritmo se encuentra en BFS(G, s).

BFS(G, s)

```
1  for each vertex v ∈ G.V
2      v.dis = ∞
3  s.dis = 0
4  Q = ∅
5  ENQUEUE(Q, s)
6  while Q ≠ ∅
7      u = DEQUEUE(Q)
8      for each vertex v ∈ G.Adj[u]
9          if v.dis == ∞
10             v.dis = u.dis + 1
11             ENQUEUE(Q, v)
```

La complejidad temporal de la búsqueda en anchura en un grafo que esta representado mediante una lista de adyacencia es $O(V + E)$.

2.5.7. El problema del máximo emparejamiento bipartito

En Cormen *et al.* (2009) encontramos las siguientes definiciones. Una arista y un vértice son incidentes, si tal vértice es extremo de la arista. Dado un grafo no direccionado $G = (V, E)$, un emparejamiento es un subconjunto de aristas $M \subseteq E$, talque para todos los vértices $v \in V$, a lo mucho una arista de M es incidente en v. Un vértice $v \in V$ es emparejado por el emparejamiento M si alguna arista en M es incidente en v, en otro caso v no esta emparejado. El máximo emparejamiento es un emparejamiento de máxima cardinalidad, es decir, un emparejamiento M talque para para cualquier emparejamiento M', se tiene que $|M| \geq |M'|$. El conjunto de vértices de un grafo puede ser particionado en $V = L \cup R$, donde L y R son disjuntos y todas las aristas en E van entre L y R.

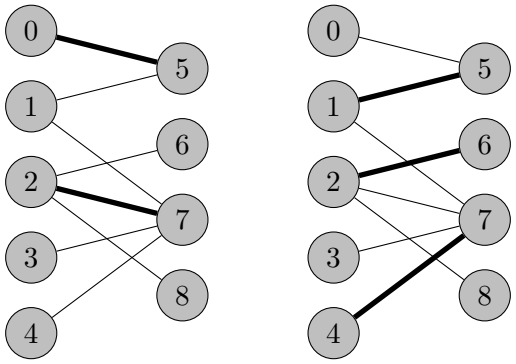


Figura 2.8: Ejemplo de emparejamiento bipartito
Un grafo bipartito $G = (V, E)$ con partición de vértices $V = L \cup R$. Un emparejamiento con cardinalidad 2, un emparejamiento máximo con cardinalidad 3.
Fuente: Cormen *et al.* (2009).

2.5.8. Algoritmo de Hopcroft-Karp

Cormen *et al.* (2009) Explica que el problema de buscar un máximo emparejamiento bipartito en un grafo puede ser solucionado mediante el algoritmo de Hopcroft-Karp Karp. Los vértices que conforman a una arista se llaman puntos finales. Dado un grafo no direccionado y bipartito $G = (V, E)$, donde $V = L \cup R$ y todas las aristas tienen exactamente un punto final en L. Sea M un emparejamiento en G. Un camino simple P en G, es un camino de aumento con respecto a M si comienza en un vértice no emparejado en L, si termina en un vértice no emparejado en R y si sus aristas pertenecen alternativamente a M y $E - M$. Un camino de aumento mas corto con

respecto a un emparejamiento M es un camino de aumento con un mínimo numero de aristas.

Dados dos conjuntos A y B, la diferencia simétrica de $A \oplus B$ es definida como $(A - B) \cup (B - A)$. Es decir los elementos que están exactamente en uno de los dos conjuntos. Aplicando la definición de la diferencia simétrica, se tiene que si M es un emparejamiento y P es un camino de aumento con respecto a M, entonces la diferencia simétrica $M \oplus P$ es un emparejamiento y $|M \oplus P| = |M| + 1$. nótese que si $P_1, P_2, .., P_k$ son vértices disjuntos de caminos de aumento con respecto a M, entonces la diferencia simétrica $M \oplus (P_1 \cup P_2 \cup .. P_k)$ es un emparejamiento con cardinalidad $|M| + k$. Es decir que al encontrar caminos de aumento, puede aumentar el numero emparejamientos. La implementación en pseudocódigo del algoritmo se encuentra en HOPCROFT-KARP(G).

HOPCROFT-KARP(G)
1 $M = \emptyset$
2 **repeat**
3 let $\mathcal{P} = \{P_1, P_2, .., P_k\}$ be a maximal set of vertex-disjoint
shortest augmenting paths with respect to M
4 $M = M \oplus (P_1 \cup P_2 \cup .. P_k)$
5 **until** $\mathcal{P} == \emptyset$
6 **return** M

Cormen *et al.* (2009) Explica que el algoritmo de Hopcroft-Karp encuentra el conjunto máximo de caminos de aumento en $O(\sqrt{|V|})$ iteraciones. Por lo cual la complejidad temporal del algoritmo es $O(|E| * \sqrt{|V|})$.

2.6. Conceptos de programacion dinamica y algoritmos

2.6.1. Programación dinámica

Cormen *et al.* (2009) Explica que la programación dinámica es un método para la resolución de problemas, el cual resuelve un problema combinando las soluciones de los subproblemas. La programación dinámica se aplica cuando los problemas se superponen, cuando los subproblemas comparten subproblemas, donde un algoritmo de programación dinámica resuelve un subproblemas solo una vez y guarda la respuesta en una tabla, de esta forma evita el trabajo de volver a calcular la respuesta. La programación dinámica se aplica a problema de optimización dichos problemas pueden tener mas de una solución posible, en el cual se desea encontrar el máximo o

mínimo, a estas soluciones se llaman solución optima.

Para desarrollar un algoritmo de programación de dinámica se sigue la siguiente secuencia de pasos:

1. Caracterizar la estructura de solución optima.
2. Definir recursivamente el valor de una solución optima.
3. Calcular el valor de una solución optima, normalmente en forma ascendente.
4. Construir una solución optima a partir de la información calculada.

2.6.2. El problema de la subsecuencia comun mas larga

La subsecuencia comun mas larga tambien es conocida por sus siglas en ingles como *LCS*. En Cormen *et al.* (2009) encontramos las siguientes definiciones. Dada una secuencia $X = [x_1, x_2, .., x_m]$, una secuencia $Z = [z_1, z_2, .., z_k]$ es una subsecuencia de X si existe una secuencia estrictamente creciente $[i_1, i_2, .., i_k]$ de indices de X talque para todo $j = 1, 2, .., k$ se tiene que $x_i = z_j$. Dadas dos secuencias X y Y, se dice que una secuencia Z es una subsecuencia comun de X y Y si Z es una subsecuencia de X y Y. Apartir las definiciones, el problema de la subsecuencia comun mas larga se define como: Dadas dos secuencias $X = [x_1, x_2, .., x_m]$ y $Y = [y_1, y_2, .., y_n]$ se desea encontrar la subsecuencia comun de longitud maxima de X y Y.

2.6.3. Caracterizacion de la LCS

Dada una secuencia $X = [x_1, x_2, .., x_m]$, se define como el i-ésimo prefijo de X, para $i = 1, 2, .., m$ como $X_i = [x_1, x_2, .., x_i]$. A continuación se muestra el teorema para la subestructura optima de la subsecuencia comun mas larga:

Teorema 2.6.1 (Subestructura optima de una LCS)
Dadas las secuencias $X = [x_1, x_2, .., x_m]$ *y* $Y = [y_1, y_2, .., y_n]$, *y dado* $Z = [z_1, z_2, .., z_k]$ *como alguna LCS de* X *y* Y.

1. *Si* $x_m = y_n$, *entonces* $z_k = x_m = y_n$ *y* Z_{k-1} *es una LCS de* X_{m-1} *y* Y_{n-1}.
2. *Si* $x_m \neq y_n$, *entonces* $z_k \neq x_m$ *implica que* Z *es una LCS de* X_{m-1} *y* Y.
3. *Si* $x_m \neq y_n$, *entonces* $z_k \neq y_n$ *implica que* Z *es una LCS de* X *y* Y_{n-1}.

Apartir del Teorema 2.6.1, se muestra que la *LCS* de dos secuencias contiene dentro de ella una *LCS* de prefijos de las dos secuencias. Por lo cual, el problema de la *LCS* tiene: Propiedad de subestructura optima, solución recursiva y propiedad de superposicion de subproblemas.

2.6.4. Solucion recursiva de la LCS

El teorema 2.6.1 implica que se debe examinar uno o dos subproblemas cuando se encuentra una LCS de $X = [x_1, x_2, .., x_m]$ y $Y = [y_1, y_2, .., y_n]$. Si $x_m = y_n$, se debe encontrar una LCS de X_{m-1} y Y_{n-1}. Añadiendo $x_m = y_n$ a esta LCS. Si $x_m \neq y_n$ entonces se debe resolver dos subproblemas, buscando una LCS de X_{m-1} y Y, y buscando una LCS de X y Y_{n-1}. Cualquiera de las dos LCS que sea mas larga es un LCS de X y Y. La propiedad de superposicion de subproblemas en el problema de la LCS. se ve al busca una LCS de X y Y, se necesita buscar la LCS de X y Y_{n-1}, y de X_{m-1} y Y. Donde cada uno de estos subproblemas tiene el sub-subproblema de encontrar la LCS de X_{m-1} y Y_{n-1}.

Se define a $c[i, j]$ como la longitud de una LCS de secuencias X_i y Y_j. Si $i = 0$ ó $j = 0$, una de las secuencias tiene longitud cero, y por lo tanto la LCS tiene longitud cero, La subestructura optima de la LCS esta dado por la siguiente formula recursiva.

$$c[i,j] = \begin{cases} 0, & \text{si } i = 0 \text{ ó } j = 0 \\ c[i-1, j-1] + 1, & \text{si } i, j > 0 \text{ y } x_i = y_j \\ max(c[i, j-1], c[i-1, j]), & \text{si } i, j > 0 \text{ y } x_i \neq y_j \end{cases} \tag{2.1}$$

En la formulacion recursiva una condicion en problema restinge cual subproblema se considera. Cuando $x_i = y_j$, se debe considerar el subproblema de buscar la LCS de X_{i-1} y Y_{j-1}, en otro caso se considera el subproblemas de buscar la LCS de X_i y Y_{j-1} y el subproblema de buscar la LCS de X_{i-1} y Y_j.

2.6.5. Calculo de la longitud de la LCS

Apartir de la ecuacion 2.1, se puede escribir una algoritmo de programacion dinamica que calcule la longitud de la LCS de dos secuencias en complejidad temporal de $O(n * m)$. El procedimiento LCS-Length toma dos secuencias $X = [x_1, x_2, .., x_m]$ y $Y = [y_1, y_2, .., y_n]$ como entrada, y almacena los valores $c[i, j]$ en una tabla $c[0..m, 0..n]$, y calcula los valores $c[i, j]$ comenzando de la primera fila y columna y asi sucesivamente hasta llegar a la ultima fila y columna. Tambien almacena valores en una tabla $b[1..m, 1..n]$ que sirve para construir la solucion optima. El procedimiento retorna las tablas b y c, que contienen la longitud de la LCS de X y Y. Donde $c[m, n]$ contiene la longitud de la LCS de X y Y.

LCS-LENGTH(X, Y)

```
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Dadas las secuencias $X = [A, B, C, B, D, A, B]$ y $Y = [B, D, C, A, B, A]$ el procedimiento LCS-LENGTH calcula las tablas b y c donde la longitud de la LCS es $b[7, 6] = 4$, las tablas se muestran en la figura 2.9.

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1
2	0	1	1	1	1	2	2
3	0	1	1	2	2	2	2
4	0	1	1	2	2	3	3
5	0	1	2	2	2	3	3
6	0	1	2	2	3	3	4
7	0	1	2	2	3	4	4

(a) Tabla c

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	↑	↑	↑	↖	←	↖
2	0	↖	←	←	↑	↖	←
3	0	↑	↑	↖	←	↑	↑
4	0	↖	↑	↑	↑	↖	←
5	0	↑	↖	↑	↑	↑	↑
6	0	↑	↑	↑	↖	↑	↖
7	0	↖	↑	↑	↑	↖	↑

(b) Tabla b

Figura 2.9: Tablas calculadas por el procedimiento LCS-LENGTH
Para dos secuencias $X = [A, B, C, B, D, A, B]$ y $Y = [B, D, C, A, B, A]$, su LCS es igual a 4.
Fuente: Cormen *et al.* (2009).

2.6.6. Construccion de la LCS

Apartir de la tabla b que retorna el procedimiento LCS-LENGTH, se construye una LCS de $X = [x_1, x_2, .., x_m]$ y $Y = [y_1, y_2, .., y_n]$. Comenzando en $b[m, n]$ y siguiendo las flechas, encontramos los elemenos de la LCS en orden inverso. El siguiente procedimiento recursivo imprime una LCS de X y Y. La llamada inicial del procedimiento es PRINT-LCS$(b, X, X.length, Y.length)$.

PRINT-LCS(b, X, i, j)
1 **if** $i == 0$ or $j == 0$
2 return
3 **if** $b[i, j] ==$ "$\nwarrow$"
4 PRINT-LCS$(b, X, i - 1, j - 1)$
5 print x_i
6 **elseif** $b[i, j] ==$ "$\uparrow$"
7 PRINT-LCS$(b, X, i - 1, j)$
8 **else** PRINT-LCS$(b, X, i, j - 1)$

Para la tabla b de la figura 2.9b, este procedimiento imprime $BCBA$. La complejidad temporal del procedimiento PRINT-LCS es de $O(m + n)$.

2.7. Indices de similitud

Magurran (1988) explica que los indices de similitud expresan el grado en que dos muestras son semejantes por las especies presentes en ellas. Estos valores pueden obtenerse con base en datos cualitativos o cuantitativos.

2.7.1. Indice de Sorensen-Dice

El indice de Sorensen-Dice también conocido como el coeficiente de Sorencen. Es un estadístico utilizado para comparar la similitud entre dos muestras. El indice de Sorencen-Dice se define como:

$$S = \frac{2 * |A \cap B|}{|A| + |B|} \tag{2.2}$$

En la ecuacion 2.2 se muestra a: $|A|$ y $|B|$ como el numero de especies en las muestras A y B, $|A \cap B|$ como el numero de especies compartidas por las muestras y S como el indice de similitud y varia entre 0 a 1. Es decir se encuentra en el rango de $[0 \,..\, 1]$.

2.7.2. Indice de Jaccard

El indice de Jaccard tambien conocido como el coeficiente de Jaccard, Es un estadistico utilizado para medir el grado de similitud entre dos conjuntos. El indice de Jaccard se define como:

$$J = \frac{|A \cap B|}{|A \cup B|} \tag{2.3}$$

En la ecuacion 2.3 se muestra a: $|A \cup B|$ como el numero de especies en las muestras A y B, $|A \cap B|$ como el numero de especies compartidas por las muestras y J como el indice de similitud y varia entre 0 a 1. Es decir se encuentra en el rango de $[0 \,..\, 1]$.

CAPÍTULO 3: DISEÑO METODOLOGICO

En el capitulo anterior se presentó la teoría necesaria para el diseño del algoritmo, en este capitulo se realizara: La especificación no formal, el diseño del algoritmo, el análisis de complejidad y finalmente la implementación en Python.

3.1. Especificación no formal del algoritmo

El algoritmo SCAM, tiene como entrada: Dos archivos de código fuente A y B, un numero en el rango $[0\,..\,100]$ que representa la tolerancia para la alineacion entre fragmentos de codigo. Como salida un numero en el rango de $[0\,..\,100]$, que representa el porcentaje del indice de similitud entre los dos archivos.

3.2. Diseño del algoritmo

Para el diseño del algoritmo se toma en cuenta las transformaciones o métodos de ofuscacion de código de fuente que se presentaron en el capitulo anterior. Una transformación consiste en la modificación en las instrucciones del código con el fin de ocultar la similitud con otro. Estas transformaciones son tratados como problemas a resolverse.

- **Problema 1.** Cambios en el formato del código. Es decir agregar o eliminar, saltos de linea, sangrías o espacios.
- **Problema 2.** Cambios en los comentarios del código. Es decir agregar, eliminar y modificar comentarios.
- **Problema 3.** Cambios en los nombres de los identificadores. Es decir cambiar los nombres de las varibales, constantes, nombres de funciones, nombres de clases, etc.
- **Problema 4.** Cambios en el orden de las declaraciones de las variables.
- **Problema 5.** Agregar instrucciones innecesarias. Es decir agregar declaraciones de variables que no se utilizan en el programa.
- **Problema 6.** Dividir una instrucción en varias otras.
- **Problema 7.** Reemplazo de la llamada de un procedimiento por el procedimiento. Es decir el reemplazo de la llamada del procedimiento por el cuerpo de un procedimiento.
- **Problema 8.** Cambios en el orden de los operadores u operandos.

- **Problema 9.** Reemplazo en las estructuras de control por equivalentes. Es decir cambiar un FOR por un WHILE.

El algoritmo propuesto es una composición de algoritmos clásicos. En la figura 3.1 se muestra el diagrama de funcionamiento del algoritmo, en el cual se muestra las fases del algoritmo. Las fases consisten en: La fragmentación y tokenización de los códigos fuente, mediante un Lexer. Calculo del maximo emparejamiento de fragmentos, mediante el algoritmo de Hopcroft-Karp. Calculo de la subsecuencia comun mas larga entre fragmentos, mediante el algoritmo de LCS. Calculo del porcentaje del indice de similitud, mediante el indice de Sorencen-Dice.

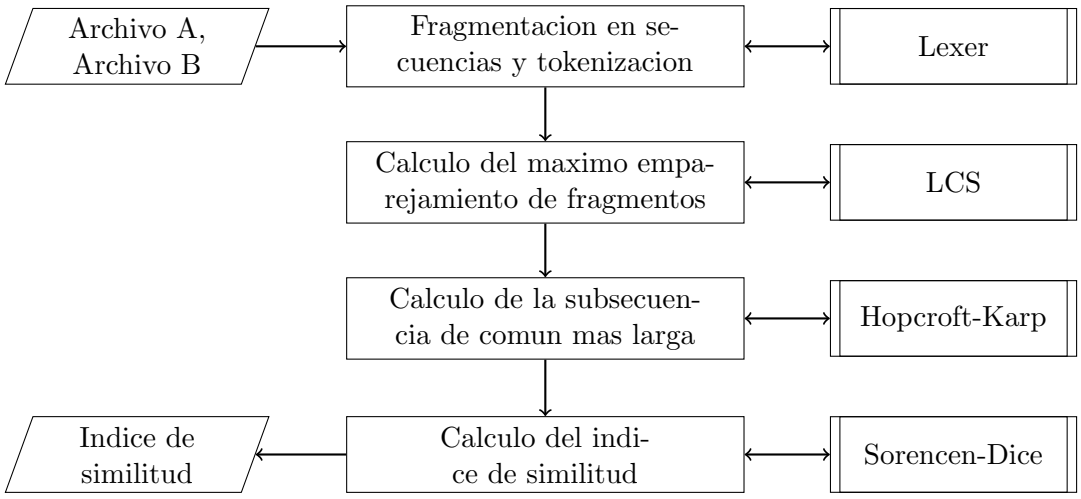


Figura 3.1: Diagrama de funcionamiento del algoritmo SCBM
Fuente: Elaboración propia.

Resumiendo el funcionamiento del algoritmo, el lexer tiene como funcion fragmentar y convertir el codigo fuente en secuencias de tokens. Es decir para cada fragmento se obtiene su secuencia de tokens. Seguido con todas las secuencias de tokens del primer archivo, se realizara el calculo de la subsecuencia comun mas larga con las secuencias de tokens del segundo archivo, obteniendo asi los puntajes de similitud para cada par de secuencias. Despues con los puntajes que superen el valor de la tolerancia, se calcula el maximo emparejamiento entre fragmentos. Finalmente se con el numero de fragmentos similiares se calcula el porcentaje del indice de similitud entre los codigos fuente. En la tabla 3.1 se muestra un descripción general de la solucion a los problemas mediante los algoritmos presentados en el capitulo anterior.

Problema	Algoritmo
1	Lexer
2	Lexer
3	Lexer
4	LCS
5	Hopcroft-Karp y LCS
6	Hopcroft-Karp
7	Hopcroft-Karp
8	LCS
9	Lexer y LCS

Cuadro 3.1: Descripción general de la solucion de los problemas mediante algoritmos

Fuente: Elaboracion propia.

3.2.1. Fragmentación y tokenización

En esta fase se eliminan los elementos innecesarios que tenga el código fuente como: Los comentarios, saltos de linea que no contengan instrucciones, espacios en blanco, etc. Esto se debe a que estas lineas no serán utilizadas para la comparación de similitud. Al mismo tiempo se realizara la tokenización y clasificacion de los tokens, los tokens generados son tuplas que contienen dos valores: El tipo de token, la cadena del token. A continuacion se muestra la clasificacion de tipos de tokens:

- **Token.separator** para separadores como: Saltos de linea ó puntos y coma.
- **Token.text** para datos de tipo texto como: Espacios en blanco y tabulaciones.
- **Token.keyword** para cualquier palabra clave o reservada.
- **Token.name.function** para nombres de funciones.
- **Token.name.variable** para nombres de variables.
- **Token.literal.string** para cualquier cadena.
- **Token.literal.number** para cualquier numero entero o flotante.
- **Token.operator** para cualquier operador.
- **Token.punctuation** para cualquier simbolo de puntuacion.

- **Token.comment** para cualquier comentario.
- **Token.other** para tokens no clasificados.

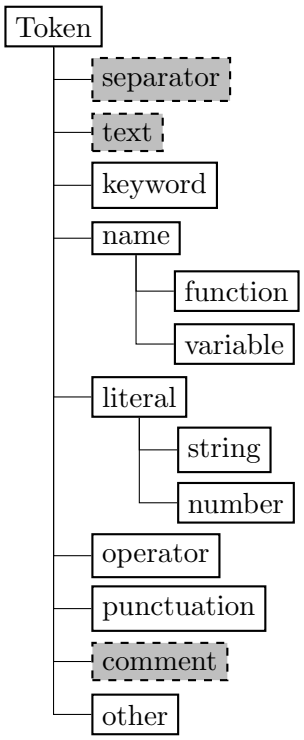


Figura 3.2: Clasificacion de un Token
Fuente: Elaboración propia.

La implementación en pseudocódigo se encuentra en LEXER. El procedimiento tiene como entrada un archivo de código fuente *file*, y de salida un arreglo de tokens clasificados.

```
LEXER(file)
 1  array-tokens = GET-TOKENS(file)
 2  array-sequences = ∅
 3  array-sequences-per-line = ∅
 4  sequence = ∅
 5  For each token ∈ array-tokens
 6       if IS-SEPARATOR(token)
 7            ADD(array-sequences-per-line, sequence)
 8            sequence = ∅
 9       else ADD(sequence, token)
10            ADD(array − sequencestoken)
11  return array-sequences-per-line, array-sequences
```

3.2.2. Cálculo de la subsecuencia comun mas larga

A partir de las definiciones de la LCS, podemos definir la LCS de fragmentos. Dadas dos secuencias de tokens $A[1\,..\,n]$ y $B[1\,..\,m]$ se desea encontrar la subsecuencia comun de longitud maxima de A y B.

Para el calculo de la subsecuencia comun mas larga, utilizaremos el algoritmo de programacion dinamica que resuelve la LCS, el algoritmo sera adaptado a la LCS de secuencias de tokens. El valor calculado por este modulo es la medida de similitud que se utilizara para determinar si dos fragmentos de código son similares. El valor se encuentra en el rango de $[0\,..\,1]$. Donde cero representa que se utilizaron el maximo numero de inserciones o eliminaciones, lo que significa que las dos secuencias son muy diferentes, y uno representa que no necesitaron dichas operaciones, lo que significa que las dos secuencias son exactamente iguales.

La medida se utilizara para la construcción de aristas durante el modelamiento de un grafo para hallar el máximo emparejamiento de fragmentos entre archivos de códigos fuente.

```
GET-LCS(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let c[0..m, 0..n] be new table
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if TOKEN-COMPARISON(x_i, y_j)
11              c[i, j] = c[i − 1, j − 1] + 1
12          elseif c[i − 1, j] ≥ c[i, j − 1]
13              c[i, j] = c[i − 1, j]
14          else c[i, j] = c[i, j − 1]
15  i = X.length
16  j = Y.length
17  lines-match = ∅
18  while i > 0 and j > 0
19      if TOKEN-COMPARISON(x_i, y_j)
20          ADD(x_i.line, y_j.line)
21          i = i − 1
22          j = j − 1
23      elseif c[i − 1, j] > c[i][j − 1]
24          i = i − 1
25      else j = j − 1
26  return lines-match
```

3.2.3. Calculo del maximo emparejamiento de fragmentos

A partir de los conceptos de grafos y el máximo emparejamiento bipartito, podemos calcular el máximo emparejamiento de fragmentos. Dados arreglos de secuencias de tokens $L[1..n]$ y $R[1..m]$, podemos modelar un grafo bipartito no direccionado $G = (V, E)$ donde $V = L \cup R$. Los vértices del grafo son las secuencias de tokens $L[i]$ y $R[j]$, y el conjunto de aristas E, esta compuesto por la relación de similitud que existe entre dos secuencias de tokens $L[i]$ y $R[j]$. En este modulo

se calcula el numero máximo de fragmentos similares. Dado un grafo no direccionado y bipartito $G = (V, E)$, donde $V = L \cup R$. Para encontrar la máxima cardinalidad de emparejamiento bipartito de G, se utiliza el algoritmo de Hopcroft-Karp. Para encontrar los caminos de aumento, se utilizan algoritmos para recorridos de grafos como: La búsqueda en anchura y búsqueda en profundidad. La implementación en pseudocódigo del algoritmo se encuentra en HOPCROFT-KARP-EXTENDED(G).

HOPCROFT-KARP-EXTENDED(G)

```
1  for each vertex v ∈ G.V
2      v.match = −1
3  mcbm = 0
4  while BFS-MODIFIED(G)
5      for each vertex v ∈ G.V
6          v.visited = UNVISITED
7      for each vertex r ∈ G.R
8          if r.match == −1 and DFS-MODIFIED(G, r)
9              mcbm = mcbm + 1
10 return mcbm
```

Para cada vértice no emparejado de L podemos realizar una búsqueda en anchura modificada, para encontrar la longitud del camino mas corto hacia un vértice no emparejado de R. La búsqueda en anchura modificada asegurará que solo se atraviesa una arista, si hace que el camino se alterne entre una arista de M y una arista de $E - M$. Cuando se alcanza por primera vez un vértice no emparejado de R, se obtiene la longitud k de un camino de aumento mas corto. Realizar estos pasos tiene como complejidad temporal $O(|E|)$. La implementación en pseudocódigo del algoritmo se encuentra en BFS-MODIFIED(G).

```
BFS-MODIFIED(G)
 1  Q = ∅
 2  G.k = ∞
 3  for each vertex l ∈ G.L
 4      if l.match == −1
 5          l.distance = 0
 6          ENQUEUE(Q, l)
 7      else
 8          l.distance = −1
 9  while Q ≠ ∅
10      u = DEQUEUE(Q)
11      if u.distance <= G.k
12          for each vertex v ∈ G.Adj[u]
13              if v.distance == −1
14                  v.distance = u.distance + 1
15                  if v.match == −1
16                      G.k = v.distance
17                  else
18                      v.match.distance = v.distance + 1
19                      ENQUEUE(Q, v.match)
20  return G.k ≠ ∞
```

Para encontrar caminos disjuntos, se realiza un búsqueda en profundidad en los vértices de R que se encontraron a una distancia k, manteniendo la propiedad de que el siguiente vértice por visitar tiene una distancia menor, y las aristas se alternan entre estar en M y $E - M$. Durante el recorrido se debe marcar los vértices como usados para no volverlos a atravesar. Realizar estos pasos tiene como complejidad temporal $O(|E|)$. La implementación en pseudocódigo del algoritmo se encuentra en DFS-MODIFIED(G, s).

```
DFS-MODIFIED(G, u)
1  for each vertex v ∈ G.Adj[u]
2      if v.visited == UNVISITED and v.distance == u.distance + 1
3          v.visited = VISITED
4          if v.distance ≠ G.k
5              if v.match == −1 or DFS-MODIFIED(G, v)
6                  u.match = v
7                  v.match = u
8                  return TRUE
9  return FALSE
```

3.2.4. Calculo del indice de similitud

A partir de la definición del coeficiente de Sorensen-Dice, podemos definir la similitud entre dos códigos fuente. Dados dos archivos de código fuente *fileA* y *fileB*, la similitud entre los códigos se define de la siguiente forma:

$a = |\text{LEXER}(\textit{file-A})|$

$b = |\text{LEXER}(\textit{file-B})|$

$c = |\text{SCAM-ALGORITHM}(\textit{file-A}, \textit{file-B})|$

$$S = \frac{2 * c}{a + b} * 100 labelsorencen2 \tag{3.1}$$

En la ecuacion **??** se muestra el procentaje del indice de similitud, el cual determina la similitud entre dos archivos de codigo fuente.

3.2.5. Algoritmo SCAM

En este modulo se realiza el modelamiento del grafo y apatir de los modulos LEXER, GET-LCS y HOPCROFT-KARP-EXTENDED podemos calcular el porcentaje del indice de similitud entre dos archivos, el indice varia entre $[0\,..\,100]$. A continuación la implementación en pseudocódigo del algoritmo propuesto.

```
SCAM-Algorithm(file-A, file-B, tolerance)
 1  arr-sequences-A = Lexer(file-A)
 2  arr-sequences-B = Lexer(file-B)
 3  let G be a graph bipartite G = ((L ∪ R), E)
 4  G.L = ∅, G.R = ∅, G.E = ∅
 5  for i to arr-sequences-A.length
 6        add-vertex(G.L, i)
 7  for i to arr-sequences-B.length
 8        add-vertex(G.R, i)
 9  for i to arr-sequences-A.length
10        for j to arr-sequences-B.length
11              arr-A = arr-sequences-A[i]
12              arr-B = arr-sequences-B[i]
13              if Token-Sequence-Alignment(arr-A, arr-B) > tolerance
14                    add-edge(G.E, i, j)
15  c = Hopcroft-Karp-Extended(G)
16  a = arr-sequences-A.length
17  b = arr-sequences-B.length
18  return (2 * c)/(a + b)
```

3.3. Análisis del algoritmo

A continuación el análisis de la complejidad temporal de cada uno de los procedimientos, la complejidad fue calculada en notación Big O.

1. El modulo Lexer tiene complejidad temporal de $O(t)$ donde t representa el numero de tokens que contiene el archivo $file$.

2. El modulo Get-LCS tiene complejidad temporal de $O(m * n)$ donde m y n, representan el número de elementos de los arreglos de tokens A y B.

3. El modulo Hopcroft-Karp-Extended tiene complejidad temporal $O((|L|*|R|)*\sqrt{|L| + |R|})$, donde $|L|$ y $|R|$ representa el numero de fragmentos de codigo de los archivos.

4. El modulo SCAM-Algorithm hace uso de los anteriores metodos, entonces la complejidad temporal es de $O(t + (m * n) * (|L| * |R|) + (|L| * |R|) * \sqrt{|L| + |R|})$.

3.4. Implementación en Python del algoritmo

3.4.1. Pygments

Pygments es una libreria de codigo abierto escrita en el lenguaje de programacion de Python, Brandl, Georg and Chajdas, Matthaus (2022) explica que *Pygments* es un resaltador de sintaxis generico, es utilizado en foros, wikis u otros aplicaciones que necesitan embellecer el codigo fuente entre sus características se tiene:

- Admite una amplia gama de lenguajes son 533 en total, y otros formatos de texto.
- Soporte nuevos lenguajes y formatos, se agrega facilmente, utilizan un mecanismo de lectura simple basado en expresiones regulares.
- Tiene varios formatos de salida disponibles, entre ellos secuencias *HTML*, *RTF*, *Latex* y *ANSI*.
- se puede utilizar como herramienta de linea de comandos y como biblioteca.

La biblioteca de *Pygments* tiene un modulo para el analisis lexico llamado *pygments.lexers*. El modulo se encarga de convertir un archivo de codigo fuente en un arreglo de tokens. Tambien tiene un modulo para el manejo de tokens llamado *pygments.token*.

3.4.2. Modulo lexer

Programa 3.1: Lexer

```
1  import pygments.token as token
2  import pygments.lexers as lexer
3
4  class Lexer:
5    def __init__(self, file_name):
6      self.file_name = file_name
7      self.tokens_per_line = []
8      self.dict_keywords = {}
9      self.clean_tokenize()
10
11   def compute_hash(self, s):
12     p = 31
13     m = 1e9 + 9
14     hash_value = 0
15     p_pow = 1
```

```
16      for c in s:
17        hash_value = (hash_value + (ord(c) - 97 + 1) * p_pow) % m
18        p_pow = (p_pow * p) % m
19      self.dict_keywords[s] = hash_value
20
21    def get_hash(self, s):
22      if self.dict_keywords.get(s, None) == None:
23        self.compute_hash(s)
24      return self.dict_keywords[s]
25
26    def num_convert(self, num):
27      try:
28        integer = int(num)
29        return integer
30      except Exception as e:
31        try:
32          double = float(num)
33          return double
34        except Exception as e:
35          return "N"
36
37    def clean_tokenize(self):
38      file = open(self.file_name, "r")
39      text_string = file.read()
40      file.close()
41      lex = lexer.guess_lexer_for_filename(self.file_name, text_string)
42      lex_tokens = lex.get_tokens(text_string)
43      tokens = []
44      for element in lex_tokens:
45        token_type = element[0]
46        token_value = element[1]
47        if "\n" in token_value:
48          self.tokens_per_line.append(tokens)
49          tokens = []
50        elif token_type in token.Text:
51          pass
52        elif token_type in token.Keyword:
53          tokens.append(self.get_hash(token_value))
54        elif token_type in token.Name.Function:
55          tokens.append("F")
56        elif token_type in token.Name:
57          tokens.append("V")
58        elif token_type in token.Literal.String:
59          tokens.append("S")
60        elif token_type in token.Literal.Number:
```

```
61          tokens.append(self.num_convert(token_value))
62        elif token_type in token.Operator:
63          tokens.append(token_value)
64        elif token_type in token.Punctuation:
65          tokens.append(token_value)
66        elif token_type in token.Comment:
67          pass
68        else:
69          tokens.append("X")
```

3.4.3. Modulo alignment

Programa 3.2: Alignment

```
1  def get_lcs(X, Y):
2    m = len(X) + 1
3    n = len(Y) + 1
4    c = [[0 for _ in range(n)] for _ in range(m)]
5    for i in range(1, m):
6      for j in range(1, n):
7        if X[i - 1] == Y[j - 1]:
8          c[i][j] = c[i - 1][j - 1] + 1
9        elif c[i - 1][j] >= c[i][j - 1]:
10         c[i][j] = c[i - 1][j]
11       else:
12         c[i][j] = c[i][j - 1]
13   lcs_cost = c[m - 1][n - 1]
14   max_len = max(len(X), len(Y))
15   return (lcs_cost / max_len)
```

3.4.4. Modulo bipartite-graph

Programa 3.3: Bipartite-graph

```
1  class Bipartite_graph:
2    def __init__(self, n_left, n_right):
3      self.n_left = n_left
4      self.n_right = n_right
5      self.num_vertex = n_left + n_right
6      self.adjacency_list = [[] for _ in range(self.num_vertex)]
7
8    def add_edge(self, u, v):
9      self.adjacency_list[u].append(v)
10     self.adjacency_list[v].append(u)
```

3.4.5. Modulo matching

Programa 3.4: Hopcroft-Karp-Extended

```
from collections import deque

def bfs():
  global n_left, n_right, adjacency_list, match, distance, k
  distance = [-1 for _ in range(n_left + n_right)]
  q = deque()
  k = 1e9
  for u in range(n_left):
    if match[u] == -1:
      distance[u] = 0
      q.append(u)
  while len(q) > 0:
    u = q.popleft()
    if distance[u] > k:
      break
    for v in adjacency_list[u]:
      if distance[v] == -1:
        distance[v] = distance[u] + 1
        if match[v] == -1:
          k = distance[v]
        else:
          distance[match[v]] = distance[v] + 1
          q.append(match[v])
  return k != 1e9

def dfs(u):
  global adjacency_list, distance, match, visited, k
  for v in adjacency_list[u]:
    if not visited[v] and distance[v] == distance[u] + 1:
      visited[v] = True
      if match[v] != -1 and distance[v] == k:
        continue
      if match[v] == -1 or dfs(match[v]):
        match[v] = u
        match[u] = v
        return True
  return False

def hopcroft_karp(G):
  global n_left, n_right, adjacency_list, match, visited
  n_left = G.n_left
```

```
42    n_right = G.n_right
43    adjacency_list = G.adjacency_list
44    match = [-1 for _ in range(n_left + n_right)]
45    mcbm = 0
46    while bfs():
47      visited = [0 for _ in range(n_left + n_right)]
48      for u in range(n_left):
49        if match[u] == -1:
50          if dfs(u):
51            mcbm += 1
52    return mcbm
```

3.4.6. Modulo scan-algorithm

Programa 3.5: SCAM-Algorithm

```
1  from lexer import Lexer
2  from alignment import get_lcs
3  from matching import hopcroft_karp
4  from bipartite_graph import Bipartite_graph
5
6  class Scam_algorithm:
7    def __init__(self, file_name_a, file_name_b, tolerance):
8      self.file_name_a = file_name_a
9      self.file_name_b = file_name_b
10     self.tolerance = tolerance / 100
11     self.build_graph()
12
13   def build_graph(self):
14     lexer_a = Lexer(self.file_name_a)
15     lexer_b = Lexer(self.file_name_b)
16     tokens_a = lexer_a.tokens_per_line
17     tokens_b = lexer_b.tokens_per_line
18     n_left = len(lexer_a.tokens_per_line)
19     n_right = len(lexer_b.tokens_per_line)
20     self.graph = Bipartite_graph(n_left, n_right)
21     self.nodes_left = set()
22     self.nodes_right = set()
23     for idu, u in enumerate(tokens_a):
24       for idv, v in enumerate(tokens_b):
25         if len(u) == 0 or len(v) == 0:
26           continue
27         distance = get_lcs(u, v)
28         if distance >= self.tolerance:
```

```
29            self.graph.add_edge(idu - 1, n_left + idv - 1)
30            self.nodes_left.add(idu - 1)
31            self.nodes_right.add(n_left + idv - 1)
32
33    def calculate_dice_index(self):
34      mcbm = hopcroft_karp(self.graph)
35      left = len(self.nodes_left)
36      right = len(self.nodes_right)
37      try:
38        dice_index = (2 * mcbm) / (left + right)
39      except Exception as e:
40        dice_index = 0
41      return dice_index
```

CAPÍTULO 4: EVALUACION Y RESULTADOS

4.1. Especificaciones de la prueba

Con los conjuntos y pares de archivos, se realizo la comparación del algoritmo SCAM frente a los algoritmos Greedy-String-Tiling y Winnowing-Fingerprint. El propósito de las pruebas es obtener los resultados sobre el desempeño de los algoritmos, en términos de precisión y tiempo de ejecución en la deteccion de similitud. En la figura 4.1 se ilustra el proceso de las pruebas.

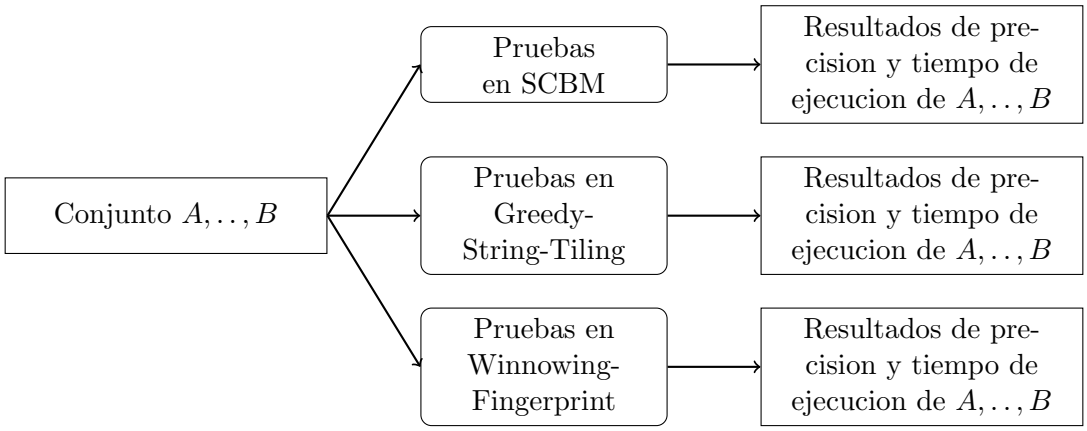


Figura 4.1: Especificacion de las pruebas
Fuente: Elaboración propia.

4.1.1. Pruebas de precisión en la deteccion de similitud

En estas prueba se midió la precisión del algoritmo SCAM frente a los algoritmos Greedy-String-Tiling y Winnowing-Fingerprint. Para cada para de archivos de un conjunto, se realizó la prueba en complejidad temporal de $O(N^2)$. Es decir para un conjunto, se realizó la comparacion de cada par de archivos del mismo.

La precisión de los algoritmos se determino mediante el porcentaje del indice de similitud entre dos archivos. Los resultados obtenidos de la comparacion de similitud entre archivos del conjunto se almacenó en una matriz, y con escala de grises se observó la similitud que existe. Donde los tonos oscuros representan que existe similitud entre dos archivos y los tonos claros representan lo contrario.

4.1.2. Tiempo de ejecucion de las pruebas

Con bibliotecas propias del lenguaje de programacion Python se midio el tiempo de ejecucion de los algoritmos al calcular la similitud en los conjuntos de archivos. La biblioteca utilizada

para la medicion fue *timeit*. Un ejemplo para la medicion del tiempo de ejecucion de un programa se muestra en el programa 4.1.

Programa 4.1: time-it

```
1 from timeit import default_timer
2 start_time = default_timer()
3 # run program
4 elapsed_time = default_timer() - start_time
5 print(elapsed_time)
```

4.2. Primera prueba

En la primera se utilizo los archivos de codigo fuente que se utilizaron en los ejemplos de ofuscacion que se presentó en el marco teorico. En la tabla 4.1 se muestra el detalle de los programas, el identificador del par y las etiquetas de los programas.

Par	Original	Ofuscado
1	Programa 2.1	Programa 2.2
2	Programa 2.3	Programa 2.4
3	Programa 2.5	Programa 2.6
4	Programa 2.7	Programa 2.8
5	Programa 2.9	Programa 2.10
6	Programa 2.11	Programa 2.12
7	Programa 2.13	Programa 2.14
8	Programa 2.15	Programa 2.16
9	Programa 2.17	Programa 2.18

Cuadro 4.1: Detalle de los conjuntos de programas con ofuscacion.

Fuente: Elaboración propia.

4.3. Segunda prueba

En la segunda prueba se utilizo archivos de código fuente en Python enviados por usuarios de un juez de programación. En el cuadro 4.2 se muestra el identificador del conjunto, el numero de problema en el juez, el numero de archivos del conjunto, el promedio de lineas de codigo de los archivos y la varianza de lineas de codigo de los archivos.

Conjunto	Nro. de Prob.	Nro. de archivos	Prom. de lineas	Var. de lineas
A	1275	10	24.8	70.84
B	1407	25	13.8	8.92
C	1588	41	14.24	39.84
D	1222	76	2.59	0.64
E	1089	101	13.95	22.15

Cuadro 4.2: Detalle de los conjuntos de archivos de código fuente para las pruebas
Fuente: Elaboración propia.

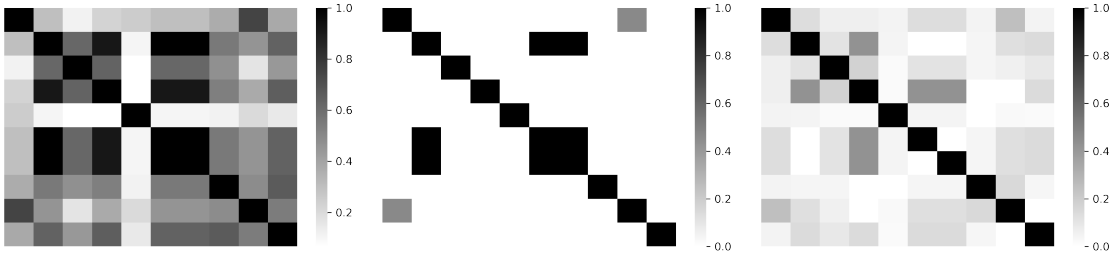
4.3.1. Prueba de precision I

En la prueba de precision I, se midió la precision de los algoritmos con un umbral de similitud del 10 %. Los parametros que utilizaron en los algoritmos fueron los siguientes:

- SCBM, *threshold* = 10
- Greedy-String-Tiling, *threshold* = 10
- Winnowing-Fingerprint, *window-length* = 10, *k-grams* = 1

Conjunto A

La matriz de similitud que se obtuvo para este conjunto de prueba. En la figura 4.2 muestra poca diferencia respecto a la precisión de los algoritmos.

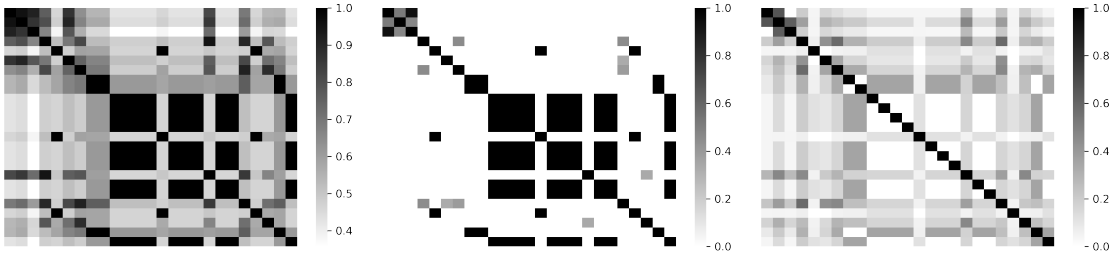


(a) SCBM (b) Greedy-String-Tiling (c) Winnowing-Fingerprint

Figura 4.2: Prueba precisión I, conjunto A
Fuente: Elaboración propia.

Conjunto B

La matriz de similitud que se obtuvo para este conjunto de prueba. Se muestra que empiezan a notarse la diferencia respecto a la precisión de los algoritmos, SCBM obtiene mas casos positivos de similitud que Winnowing.

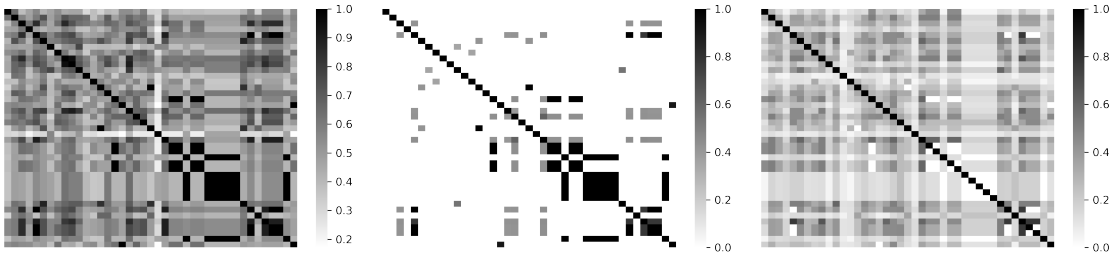


(a) SCBM (b) Greedy-String-Tiling (c) Winnowing-Fingerprint

Figura 4.3: Prueba precisión I, conjunto B
Fuente: Elaboración propia.

Conjunto C

La matriz de similitud que se obtuvo para este conjunto de prueba. Continua la diferencia respecto a la precisión de los algoritmos, SCBM obtiene mas casos positivos de similitud que Winnowing.

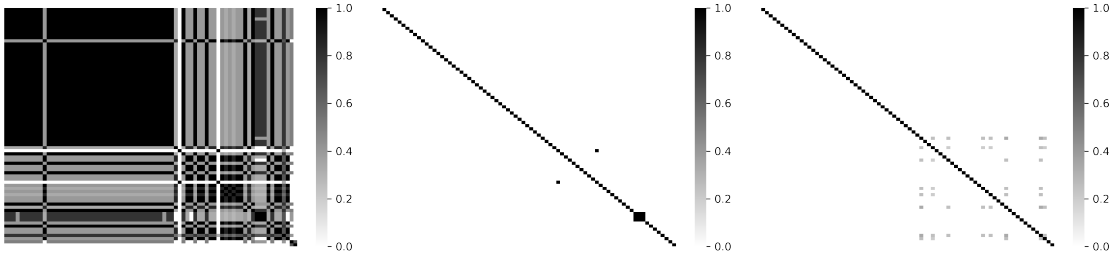


(a) SCBM (b) Greedy-String-Tiling (c) Winnowing-Fingerprint

Figura 4.4: Prueba precisión I, conjunto C
Fuente: Elaboración propia.

Conjunto D

La matriz de similitud que se obtuvo para este conjunto de prueba. Continua la diferencia respecto a la precisión de los algoritmos, SCBM obtiene un casos en el que no logra identificar la similitud mientras Winnowing si logra identificarlos.

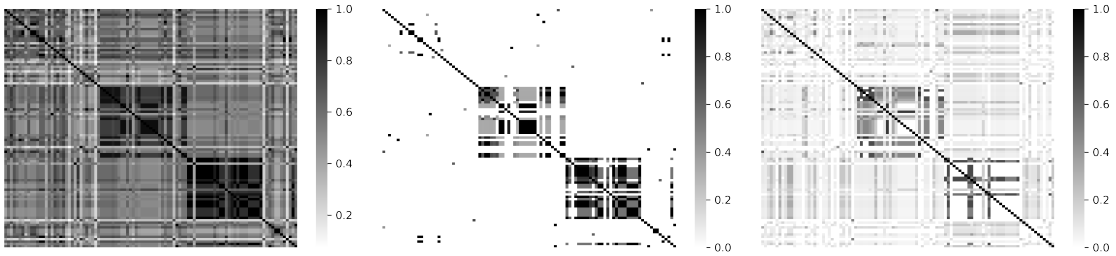


(a) SCBM (b) Greedy-String-Tiling (c) Winnowing-Fingerprint

Figura 4.5: Prueba precisión I, conjunto D
Fuente: Elaboración propia.

Conjunto E

La matriz de similitud que se obtuvo para este conjunto de prueba. Continua la diferencia respecto a la precisión de los algoritmos, SCBM obtiene un casos en el que no logra identificar la similitud mientras Winnowing si logra identificarlos.



(a) SCBM (b) Greedy-String-Tiling (c) Winnowing-Fingerprint

Figura 4.6: Prueba precisión I, conjunto E
Fuente: Elaboración propia.

4.3.2. Tiempo de ejecución de la prueba I

En la tabla 4.3 se presentan los resultados obtenidos, respecto al tiempo de ejecucion de cada algoritmo al calcular la similitud de cada conjunto de prueba. Los resultados obtenidos se encuentran en segundos.

Conjunto	SCBM	Greedy-String-Tiling	Winnowing-Fingerprint
A	11.77 seg.	0.86 seg.	1
B	4.16 seg.	3.66 seg.	2
C	14.53 seg.	12.73 seg.	3
D	3.28 seg.	19.47 seg.	4
E	52.00 seg.	42.73 seg.	5

Cuadro 4.3: Detalle del promedio de tiempo de ejecución de los algoritmos.
Fuente: Elaboración propia.

- Con las prueba realizada demuestro que el algoritmo de Winnowing es mejor por segundos respecto al tiempo de ejecución. El algoritmo SCBM presenta mejores resultados comparando archivos de codigo fuente que no tengan muchas instruciones.

CAPÍTULO 5: CONCLUSIONES Y RECOMENDACIONES

5.1. Conclusiones

5.2. Recomendaciones

Referencias

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

Ahtiainen, A., Surakka, S., y Rahikainen, M. (2006). Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises. *ACM International Conference Proceeding Series*, 276.

Anzai, K. y Watanobe, Y. (2019). Algorithm to determine extended edit distance between program codes. pp. 180–186.

Bejarano, A., García, L., y Zurek, E. (2015). Detection of source code similitude in academic environments. *Computer Applications in Engineering Education*.

Brandl, Georg and Chajdas, Matthaus (2022). Pygments: Python syntax highlighter. https://pygments.org. [Online; accessed 27-May-2022].

Catalán, J. (2010). *Compiladores : teoría e implementación*. RC Libros, San Fernando de Henares, Madrid.

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

Collberg, C., Thomborson, C., y Low, D. (1997). A taxonomy of obfuscating transformations. *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2009). *Introduction to Algorithms*. The MIT Press. MIT Press, London, England.

Cosma, G. y Joy, M. (2012). An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Trans. Computers*, 61:379–394.

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

Faidhi, J. y Robinson, S. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment.

Grier, S. (1981). A tool that detects plagiarism in pascal programs. En *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '81, p. 15–20, New York, NY, USA. Association for Computing Machinery.

Hage, J., Rademaker, P., y Van Vugt, N. (2010). A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, 28(1).

Halim, F. y Halim, S. (2019). *Programacion competitiva*. Independently Published.

Karnalim, O. y Simon, Chivers, W. (2019). Similarity detection techniques for academic source code plagiarism and collusion: A review.

Magurran, A. E. (1988). *A variety of diversities*. Springer.

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.

Moussiades, L. y Vakali, A. (2005). Pdetect: A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48.

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).

Ottenstein, K. J. (1976). An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bull.*, 8(4):30–41.

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

Popescu, D. y Nicolae, D. (2016). *Determining the Similarity of Two Web Applications Using the Edit Distance*, volumen 356.

Prechelt, L. y Malpohl, G. (2003). Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8.

Ragkhitwetsagul, C., Krinke, J., y Clark, D. (2018). A comparison of code similarity analysers. *Empirical Softw. Engg.*, 23(4):2464–2519.

Sidorov, G., Romero, M., Markov, I., Guzman-Cabrera, R., Chanona-Hernández, L., y Castillo, F. (2017). Measuring similarity between karel programs using character and word n-grams. *Programming and Computer Software*, 43:47–50.

Song, H.-J., Park, S.-B., y Park, S. (2015). Computation of program source code similarity by composition of parse tree and call graph. *Mathematical Problems in Engineering*, 2015:1–12.

Whale, G. (1990). Software metrics and plagiarism detection. *Journal of Systems and Software*, 13(2):131–138. Special Issue on Using Software Metrics.

Wise, M. (1992). Detection of similarities in student programs: Yap'ing may be preferable to plague'ing. *ACM SIGCSE Bulletin*, 24:268–271.

Yasaswi, J., Purini, S., y Jawahar, C. (2017). Plagiarism detection in programming assignments using deep features. pp. 652–657.

Đurić, Z. y Gašević, D. (2012). A Source Code Similarity System for Plagiarism Detection. *The Computer Journal*, 56(1):70–86.