UNIVERSIDAD MAYOR DE SAN ANDRES
FACULTAD DE CIENCIAS PURAS Y NATURALES
CARRERA DE INFORMÁTICA

TESIS DE GRADO

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

PARA OPTAR AL 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
2021

INDICE GENERAL

Índice de figuras

Índice de cuadros

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.

Un estudiante de programacion durante su proceso de formacion, realiza trabajos, proyectos, tareas y ejercicios de programacion, estas actividades ayudan a desarrollar la capacidad de resolucion de problemas, enfoque logico y otros. Cuando la presentacion de estas actividades es individual, encontrar trabajos similares presentados por estudiantes puede ser considerado como plagio. Una de las formas para detectar la similitud en 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.

En la actualidad existen diferentes herramientas de software que aplican metodos para la deteccion de similitud entre codigos fuente, apartir de las caracteristicas de estas herramientas se puede evidenciar que presentan deficiencias como la obsolescencia, sistemas sin codigo abierto, procesos complejos de evaluacion de similitud, sistemas incapaces que detectar similitud en un grupo grande de archivos de codigo fuente.

Ademas Novak *et al.* (2019) hace mención a que los enfoques basados en estructuras son mucho mejores, ademas que la mayoría de las herramientas de detección de similitud combinan más de un tipo de algoritmo. El diseño del algoritmo propuesto contara con tres fases, la primera fase consiste en la conversion de codigo fuente en arbol de sintaxis abstracta AST, esto se realizara mediante conceptos de compiladores, la segunda fase consiste en la comparacion de arboles obtenidos mediante conceptos de distancia de edicion de arboles TED, la fase final consiste en la agrupacion de codigos similires mediante conceptos de estructuras para conjuntos disjuntos DSU.

En el presente trabajo de tesis se centra en el diseño e implementacion de un algoritmo para la deteccion de similitud entre codigos fuente que tenga buen desempeño, en terminos de tiempo de ejecucion, espacio de memoria ocupado y precision.

1.2. Problema

1.2.1. Antecedentes del problema

Algoritmos utilizados por las herramientas

Novak *et al.* (2019) Realizo un estudio sistematico campo de la deteccion de plagio de codigo fuente, describe definiciones de plagio, herramientas para la deteccion de plagio, metricas de comparacion, metodos de ofuscacion, conjunto de datos para la comparacion y algoritmos que utilizan las herramientas. En el estudio de los algoritmos realizo un analisis en 120 articulos de los cuales identifica algoritmos basados en estilo, basado en semantica, basado en texto, huellas dactilares, recuento de atributos, basado en estructura, coincidencia de cadenas, marca de agua, basado en historial, basado en XML, codigo 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 1 se muestra detalles de estos como el primer y ultimo año de publicacion de un algoritmo, el numero de articulos en los que aparecen y si utilizan la tokenización.

Tipo de algoritmo	Ultimo año	Primer año	Nro. de artículos	Tokenizados
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 1: **Descripción general de los tipos de algoritmos**
Nota: Cuadro tomado de Novak *et al.* (2019).

Herramientas para la detección de similitud de código

Novak *et al.* (2019) describe cuatro caracteristicas de cinco herramientas que son consideradas las más importantes, en el cuadro 2 se muestran las caracteristicas como: las menciones en articulos, codigo abierto, interfaz grafica (GUI), sin conexion 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: **Descripción general de las caracteristticas de las herramientas**
Nota: Cuadro tomado de Novak *et al.* (2019).

- **JPLAG** Díaz *et al.* (2007) explica que la arquitectura de esta herramienta es de cliente servidor, solo permite la programacion del cliente, dado que es el servidor quien realiza las operaciones de comparacion, fue desarrollada por la universidad de Karlsruhe de Alemania, se puede analizar codigo de los lenguajes C, C++, Java, Scheme y texto natural. La herramienta analiza la estructura y sintaxis del código, este sistema trabaja con archivos locales.
- **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).
- **Sherlock** Díaz *et al.* (2007) explica que la herramienta es de codigo abierto, que trabaja con codigo fuente escrito en JAVA, C y texto natural, Los resultados obtenidos en la comparacion por la herramienta se muestran en porcentajes, solo trabaja con archivos locales.
- **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 Ahtiainen *et al.* (2006).

- **SIM** Díaz *et al.* (2007) explica que la herramienta es de codigo abierto, trabaja con lenguajes C, JAVA, Lisp, Modula2, Pascal y texto natural, fue desarrollado por la universidad de Amsterdam. No cuenta con una interfaz grafica, se la utiliza mediante linea de comandos, los resultados obtenidos en la comparacion obtenidos por la herramienta aparecen separados por columnas, mostrando porciones de codigos iguales, solo trabaja con archivos locales.

1.2.2. Planteamiento del problema

Los estudiantes de programación durante su proceso de formación elaboran trabajos, proyectos, tareas, ejercicios, etc. de programación, escritos en algún lenguaje de programación. Estas actividades deben realizarse de forma individual dado que 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 de código fuente.

La forma mas simple para detectar plagio en trabajos de programación, consiste en realizar la comparación de forma manual entre todos los trabajos presentados, de modo de obtener una lista de estudiantes que tienen trabajos similares. Realizar esta comparación llega a ser una trabajo moroso, por lo cual contar con una herramienta de software que realice las comparaciones de forma automática y genere la una lista de trabajos similares es de gran utilidad.

En la actualidad existen herramientas de software para detectar similitud entre códigos fuente. Según sus características se observo que estas presentan deficiencias como la obsolescencia, sin código abierto, incapacidad de utilizar gran base de información.

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 Inteligencia Artificial.

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 de código fuente

2.1.1. 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.

2.1.2. 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 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 (2021).

2.1.3. Métodos de ofuscación de código fuente

Novak *et al.* (2019) Explica que existen muchos métodos de ofuscación utilizados por estudiantes para ocultar la similitud, a su vez menciona que en un estudio de 72 articulos se identificaron 25 metodos de ofuscacion, y de estos se especificaron 16 metodos distintos.

A continuación se mencionan algunos metodos de ofuscacion de codigo fuente:

- Marzieh *et al.* (2011) Mencionan cambios en el formato del código, como la modificación de espacios en blanco como sangrías, espacios, nuevas líneas, etc.
- Marzieh *et al.* (2011) Mencionan cambios en los comentarios del código.
- Đurić y Gašević (2012) y 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) Menciona cambios en el orden de las declaraciones de la variables.
- Grier (1981) Menciona agregar lineas de codigo redundantes, lineas de codigo que no hacen nada.
- Donaldson *et al.* (1981) Menciona dividir una linea de codigo en varias lineas.
- Whale (1990) Menciona el reemplazo de la llamada de un procedimiento por el procedimiento.
- Whale (1990) Menciona el cambio de la especificacion de una declaracion como el cambio de los operaciones y el operando, alternando modificadores cambio de tipos de datos.
- Marzieh *et al.* (2011) Mencionan el cambio de estructuras de control equivalentes, como el reemplazo de estructuras repetitivas y condicionales.
- Đurić y Gašević (2012) Mencionan la simplicacion de codigo, eliminar lineas de codigo no necesarias.

2.2. Conceptos de compiladores

2.2.1. Procesador de lenguaje

Un procesador de lenguaje se muestra en la figura 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).

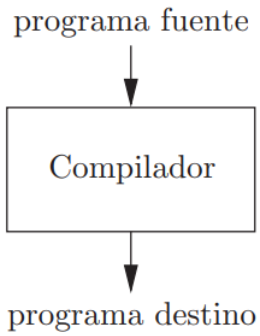

Figura. 1: **Un compilador**
Nota: Figura tomada de Aho (2008).

El proceso de compilación opera como una secuencia de fases con las cuales transforma un programa fuente en otro, se muestra en la figura 2.

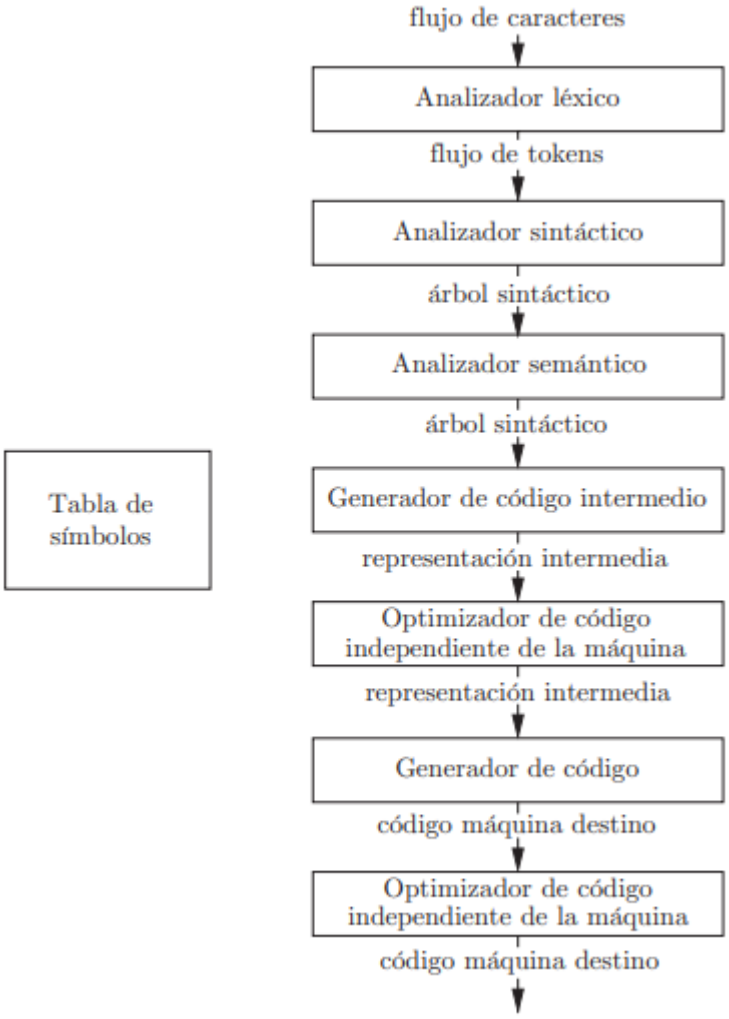

Figura. 2: **Fases de un compilador**
Nota: Figura tomada de Aho (2008).

En el presente trabajo de investigación, solo es de interés de cubrir los teoría y conceptos hasta el análisis semántico.

2.2.2. Análisis Léxico

Aho (2008) Explica que la primera fase de un compilador 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 vayamos 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.

2.2.3. Análisis Sintáctico

Aho (2008) Explica que la segunda fase del compilador es el análisis sintáctico o parsing. El parser 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 3 se muestra un árbol sintáctico para el flujo de tokens como salida del analizador sintáctico.

Catalán (2010) Explica que el analizador sintáctico tiene como entrada los lexemas que le suministra el analizador léxico y su función es comprobar que están ordenados de forma correcta dependiendo del lenguaje que se quiere procesar. Los dos analizadores suelen trabajar unidos e incluso el léxico suele ser una subrutina del sintáctico. El analizador sintáctico se le suele llamar parser, este genera de manera teórica un árbol sintáctico. Este árbol se puede ver como una estructura jerárquica que para su construcción se usa reglas recursivas. La estructuración de este árbol hace posible diferenciar entre aplicar unos operadores antes de otros en la evaluación de expresiones. En resumen la tarea del analizador sintáctico es procesar los lexemas que le suministra el analizador léxico, comprobar si están bien ordenados, y si no lo están, generar los informes correspondientes. Si la ordenación, se generará un árbol sintáctico teórico.

2.2.4. Análisis Semántico

Aho (2008) Explica que 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 de tipos, en donde el compilador verifica que cada operador tenga operando que coincidan.

Catalán (2010) Explica que esta fase toma el árbol sintáctico teórico de la anterior fase y hace una serie de comprobaciones antes de obtener un árbol semántico teórico. Esta fase es quizás la más compleja. Hay que revisar que los operadores trabajan sobre tipos compatibles, si los operadores obtienen como resultado elementos con tipos adecuados, si las llamadas a subprogramas tienen los parámetros adecuados tanto en número como en tipo, etc. Esta fase debe preparar el terreno para atajar las fases de generación de código y debe lanzar los mensajes de error que encuentre. En resumen, su tarea es revisar el significado de lo que se va leyendo para ver si tiene sentido.

2.2.5. Árboles de sintaxis abstracta

Aho (1999) Explica que un árbol es una estructura jerárquica sobre una colección de objetos, como ejemplos están los arboles genealógicos y organigramas. Los arboles son útiles para analizar circuitos eléctricos y para representar estructuras de formulas matemáticas. Dentro de la área de la computación se usan para organizar la información en sistemas de bases de datos y para representar la estructura sintáctica de un programa fuente en compiladores.

En Cormen *et al.* (2009) podemos obtener una definición y propiedades matemáticas de un árbol, para ello se desarrollaran algunas definiciones de teoría de grafos. Sea G un grafo no direccionado es representado por un par $G = (V, E)$, donde V representa el conjunto de vértices de G, se le denota con un circulo. E representa el conjunto de aristas de G, se le denota con una linea.
Ademas para que un grafo sea considerado un árbol deben cumplirse las siguientes afirmaciones:

- Dos vértices de G están conectados por un camino único.
- G es conexo, pero si se elimina cualquier vértice que no sea una hoja de E, el grafo resultante es no conexo.

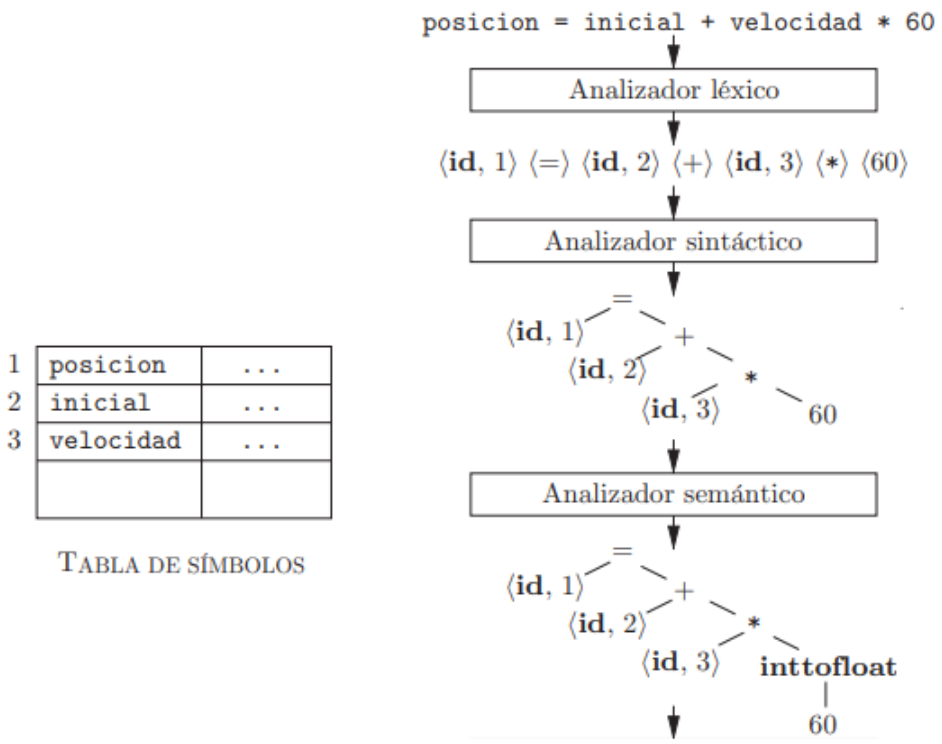

Figura. 3: **Traducción de una asignación**
Nota: Figura tomada de Aho (2008).

- G es conexo, y $|E| = |V| - 1$.
- G es acíclico, y $|E| = |V| - 1$.
- G es acíclico, pero si cualquier vértice es agregado a E, el grafo resultante contendrá un ciclo.

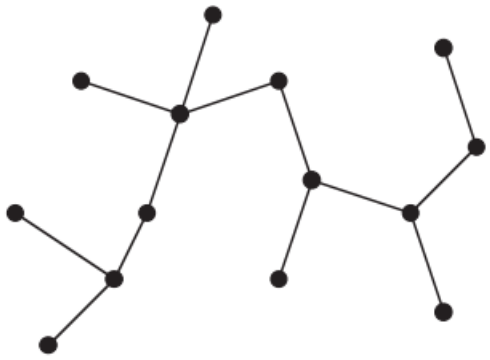

Figura. 4: **Árbol**
Nota: Figura tomada de Cormen *et al.* (2009).

En resumen un árbol es un grafo no direccionado, conexo, acíclico, y que dos

vértices cualesquiera están conectados por un camino único, como se muestra en la figura 4.

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, llamado árbol de análisis. Cada token encontrado por el lexer está representado por un nodo en el árbol de análisis. No todos los tokens 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).

2.3. Conceptos de algorítmica

2.3.1. 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.

2.3.2. Distancia de edición del árbol

Schwarz *et al.* (2017) Define a la distancia de edicion del arbol (TED) como la secuencia de operaciones de nodo de costo minimo que transforma un árbol en otro, este algoritmo tiene varias aplicaciones en ingenieria de software, procesamiento del lenguaje natural y la bioinformática. Los algoritmos de ultima generacion TED descomponen recursivamente arboles de entrada en subproblemas mas pequeños y usan programacion dinamica para construir el resultado de forma ascendente. Dentro de las implementaciones de una solucion recursiva mas eficientes estan la de Zhang y Shasha (1989) y Chen (2001), ambos son una extension del algoritmo basico de la distancia de edicion de cadenas.

De Paaßen (2018) podemos obtener las siguientes definiciones:

2.3.3. Conjuntos Disjuntos

En Cormen *et al.* (2009) obtenemos una la definición de conjuntos disjuntos y la estructura de datos para conjuntos disjuntos.

Los conjuntos disjuntos es una colección representada por $S = \{S_1, S_2, ..., S_k\}$ donde cada elemento S_i representa un conjunto dinámico, y para cada elemento de la colección se cumple que $S_i \cap S_j = \emptyset$.

La estructura de datos para conjuntos disjuntos soporta eficientemente las operaciones de crear un conjunto, búsqueda del representante de un conjunto y la unión de conjuntos.

Sean u y v elementos de un conjunto:

- $MAKE - SET(u)$ Crea un nuevo conjunto cuyo único miembro es u. Dado que los conjuntos son disjuntos, es necesario que u no se encuentre en otro conjunto.
- $UNION(u, v)$ Une los conjuntos dinámicos que contienen a u y v, es decir $S_u \cup S_v$, el representante del conjunto resultante sera cualquiera de los dos.
- $FIND - SET(u)$ devuelve un puntero al representante del conjunto que contiene a u.

Bibliografía

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

Aho, A. V. (1999). *Estructuras de datos y algoritmos*. Addison Wesley Longman, Singapore, Singapore.

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.

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.

Chen, W. (2001). New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40:135–158.

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

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

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

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. y Rademaker, P. (2010). A comparison of plagiarism detection tools.

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, 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).

Paaßen, B. (2018). Revisiting the tree edit distance and its backtracing: A tutorial.

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

Schwarz, S., Pawlik, M., y Augsten, N. (2017). A new perspective on the tree edit distance. pp. 156–170.

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

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

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

Zhang, K. y Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:1245–1262.

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