finis
Universidad Finis Terrae
Ing. Civil en Informática y Telecomunicaciones

Proyecto Estructuras de Datos y Algoritmos

Optimización de Búsqueda en texto:

Comparación y evaluación de métodos de búsquedas eficientes

Fecha:	13/06/2024
Autores:	Miguel Cornejo Diego González
e-mail:	mcornejoc4@uft.edu dgonzalezt2@uft.edu
Profesor:	Rodrigo Paredes

Índice

1. Introducción

La eficiencia en la búsqueda de datos es esencial en el ámbito de la informática para el manejo óptimo de las bases de datos. Este estudio simula el funcionamiento de motores de búsqueda como Google al implementar y evaluar algoritmos de búsqueda en textos. El objetivo principal es determinar experimentalmente si la búsqueda indexada es más efectiva que la búsqueda bruta o secuencial línea por línea. Se pretende demostrar a través de la implementación de un cliente interactivo que la implementación de un índice basado en tablas de hash no solo optimiza el tiempo de búsqueda, sino que también facilita la gestión de grandes volúmenes de datos. Esto se logra mediante la comparación de la búsqueda indexada con la fuerza bruta, la búsqueda por diccionario, y la codificación propia de las tablas de hash. Los resultados obtenidos proporcionarán una comprensión más profunda de las estructuras de datos y algoritmos involucrados, así como una base sólida para futuras investigaciones sobre la optimización de búsquedas de datos.

2. Análisis del Problema

La búsqueda eficiente de datos en textos es un desafío en el campo de la informática, especialmente en el manejo de grandes volúmenes de información en bases de datos. En este estudio, se aborda el problema de cómo optimizar la búsqueda de patrones en textos, comparando la eficacia de diferentes métodos de búsqueda. El objetivo es determinar si la búsqueda indexada, utilizando un índice basado en tablas de hash, puede superar en rendimiento a la búsqueda secuencial línea por línea, también conocida como fuerza bruta, así como también puede ser la búsqueda por diccionario.

2.1. Supuestos o condiciones

Se asume que la premisa de los textos a analizar puede variar significativamente en tamaño, desde pequeños kilobytes hasta megabytes. Los patrones de búsqueda son secuencias de caracteres (palabras o frases) que pueden verse en cualquier parte del texto. Además, se supone que el texto no tiene una estructura definida y que los patrones pueden distribuirse de manera aleatoria o uniforme. Además, se espera que el sistema tenga suficiente memoria y capacidad de procesamiento para realizar búsquedas secuenciales, indexadas y por diccionario.

2.2. Situaciones del borde

Para la evaluación completa, se deben tener en cuenta varios casos límite. Esto incluye patrones de búsqueda que aparecen al principio o al final del texto, así como los que no aparecen en el texto. Además, el manejo de patrones que contienen subcadenas repetidas en diferentes partes del texto, así como garantizar que los métodos puedan procesar textos que contienen una variedad de codificaciones y caracteres especiales.

2.3. Metodología para abordar el problema

1. **Implementación de Algoritmos:**
   - **Búsqueda Secuencial (Fuerza bruta):** Este método implica revisar cada línea del texto para encontrar coincidencias del patrón, evaluando cada carácter.
   - **Búsqueda Indexada:** Utiliza un índice basado en tablas de hash para almacenar las posiciones de las palabras en el texto, permitiendo búsquedas más rápidas.
   - **Búsqueda por Diccionario:** Utiliza una estructura de diccionario para almacenar todas las palabras del texto, permitiendo búsquedas eficientes de las palabras de interés.
2. **Construcción del Índice:**
   - Crear un índice hash a partir del texto, donde cada palabra se mapea a su posición en el texto.
   - Este índice permitirá búsquedas rápidas al reducir el espacio de búsqueda.

3. **Cliente Interactivo:**

   - Desarrollar un cliente interactivo que permita al usuario cargar diferentes textos y seleccionar el método de búsqueda.
   - Proporcionar opciones para ingresar patrones de búsqueda y visualizar los resultados, incluyendo el tiempo de búsqueda y las líneas donde se encontraron coincidencias.

4. **Comparación Experimental:**

   - Realizar múltiples pruebas con textos de diferentes tamaños y patrones de búsqueda variados.
   - Medir y comparar el tiempo de búsqueda para cada método.
   - Analizar el rendimiento y la eficiencia de cada método en diferentes escenarios.

3. Solución del Problema

3.1. Metodología

Para abordar el problema de la búsqueda eficiente de patrones en textos, se ha diseñado una solución que comprende varios pasos, incluyendo la implementación de algoritmos, la construcción de un índice hash, el desarrollo de un cliente interactivo y la realización de pruebas experimentales.

Se muestra la estructura del archivo donde esta los datos e información relevante:

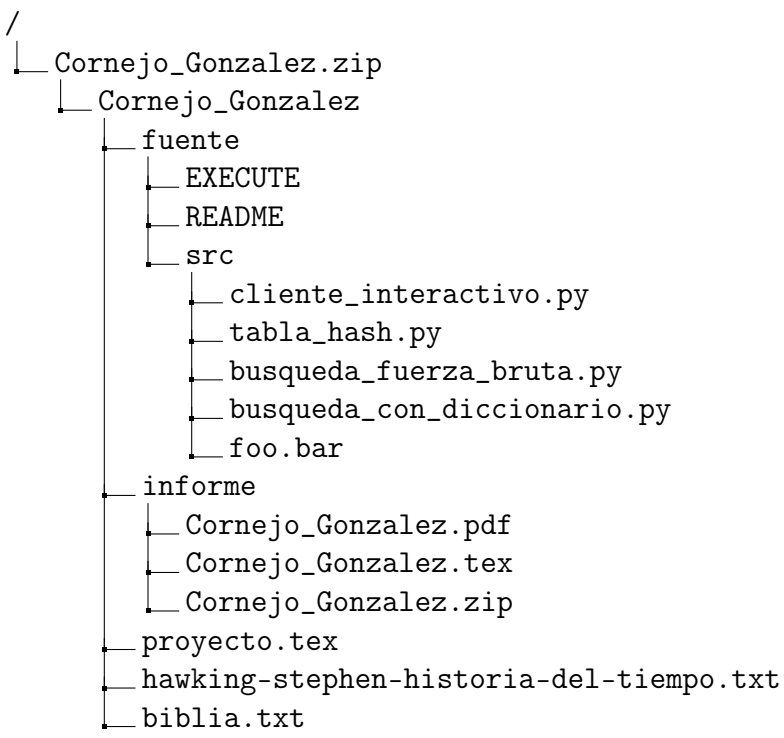

Figura 1: Estructura del archivo.

3.2. Algoritmo de solución

El enfoque para solucionar el problema de búsqueda eficiente de patrones en textos implica tres métodos principales: búsqueda por fuerza bruta, búsqueda indexada utilizando tablas de hash y búsqueda por diccionario. Presentando a continuación los programas de los logaritmos:

3.2.1. Búsqueda por Fuerza Bruta

La función "`busqueda_fuerza_bruta`": implementa el algoritmo de búsqueda secuencial o fuerza bruta para encontrar todas las ocurrencias de un patrón en un texto. Funciona recorriendo línea por línea del texto y en cada línea, busca el patrón utilizando la función

"`contar_patron`". Si se encuentra una coincidencia, se registra la línea, el número de ocurrencias y el índice de la línea donde se encontró la coincidencia en una lista de resultados. Donde finalmente, devuelve el "`return`" de esta lista de resultados.

La función "`contar_patron`" cuenta el número de veces que aparece un patrón en una línea dada utilizando un enfoque de ventana deslizante. Itera sobre la línea y en cada posición, verifica si la subcadena de longitud igual al patrón coincide con el patrón dado. Si hay una coincidencia, incrementa un contador. Al final, devuelve el número total de ocurrencias del patrón en la línea.

Presentando el programa completo de Búsqueda por Fuerza Bruta:

```
1  def busqueda_fuerza_bruta(texto, patron):
2      resultados = []  # Lista para almacenar los resultados encontrados
3      indice_linea = 0  #  ndice  para recorrer las l neas del texto
4
5      # Iterar sobre cada l nea en el texto
6      while indice_linea < len(texto):
7          linea = texto[indice_linea]  # Obtener la l nea actual
8          contador = contar_patron(linea, patron)
   # Contar las ocurrencias del patr n en la l nea
9
10         # Si se encontraron ocurrencias, agregar a los resultados
11         if contador > 0:
12             resultados.append((indice_linea, linea, contador))
13
14         indice_linea += 1  # Pasar a la siguiente l nea
15
16     return resultados  # Devolver todos los resultados encontrados
17
18
19 def contar_patron(linea, patron):
20     contador = 0  # Inicializar el contador de ocurrencias
21     longitud_patron = len(patron)  # Obtener la longitud del patr n
22     longitud_linea = len(linea)  # Obtener la longitud de la l nea
23
24     # Iterar sobre cada posible subcadena en la l nea
25     for i in range(longitud_linea - longitud_patron + 1):
26         # Comparar la subcadena con el patr n y contar las ocurrencias
27         if linea[i:i+longitud_patron] == patron:
28             contador += 1  # Incrementar el contador si se encuentra una ocurren
29
30     return contador  # Devolver el total de ocurrencias encontradas
```

Figura 2: Código de búsqueda de fuerza bruta en Python

3.2.2. Índice Basado en Tablas de Hash

La clase "`TablaHash`" implementa una estructura de tabla hash para indexar palabras en un texto y facilitar la búsqueda eficiente de líneas donde aparecen estas palabras. Al inicializar un objeto TablaHash con un texto, se construye un índice que mapea cada palabra única a una lista de números de línea donde esa palabra aparece. Esto se logra mediante los métodos "`construir_indice`", que recorre cada línea del texto para extraer y agregar palabras al índice, y "`agregar`" , que asigna índices a nuevas palabras y actualiza las listas de líneas asociadas. La búsqueda se realiza con el método "`buscar`", que devuelve las líneas correspondientes a una palabra dada si está presente en el índice, permitiendo así una recuperación rápida de información en grandes volúmenes de texto.

Presentando el programa completo de Índice Basado en Tablas de Hash:

```
1  class TablaHash:
2      def __init__(self, texto):
3          self.vocabulario = {}  # Diccionario para mapear palabras a  ndices
4          self.construir_indice(texto)  # Construir el  ndice  al inicializar la c
5
6      def construir_indice(self, texto):
7          self.tabla = {}  # Diccionario para almacenar listas de l neas por  ndi
8          self.texto = texto  # Guardar el texto completo como atributo de la inst
9          self.contador_palabras = 0  # Contador para asignar  ndices  a palabras
10         num_linea = 1
11
12         # Iterar sobre cada l nea del texto
13         for linea in texto:
14             palabras = self.extraer_palabras(linea)  # Obtener lista de palabras
15             for palabra in palabras:
16                 self.agregar(palabra, num_linea)  # Agregar palabra al  ndice
17             num_linea += 1
18
19     def agregar(self, palabra, num_linea):
20         if palabra not in self.vocabulario:  # Si la palabra no est  en el voca
21             self.vocabulario[palabra] = self.contador_palabras  # Asignar un nue
22             self.tabla[self.contador_palabras] = []  # Inicializar una lista va
23             self.contador_palabras += 1  # Incrementar el contador de  ndices
24
25         indice = self.vocabulario[palabra]  # Obtener el  ndice  de la palabra
26         if num_linea not in self.tabla[indice]:  # Si la l nea no est  en la l
27             self.tabla[indice].append(num_linea)  # Agregar la l nea al  ndice
28
29     def buscar(self, palabra):
30         if palabra in self.vocabulario:  # Si la palabra est  en el vocabulario
31             indice = self.vocabulario[palabra]  # Obtener el  ndice  asociado a
32             return self.tabla[indice]  # Devolver la lista de l neas asociadas
33         else:
34             return []  # Devolver una lista vac a si la palabra no est  en el
```

```
35
36      def extraer_palabras(self, linea):
37          palabra_actual = ''  # Variable para almacenar la palabra actual durante
38          palabras = []  # Lista para almacenar todas las palabras encontradas en
39
40          # Iterar sobre cada caracter en la l nea
41          for caracter in linea:
42              if caracter == ' ' or caracter == '\n':  # Si el caracter es un espa
43                  if palabra_actual:  # Si hay una palabra almacenada en palabra_a
44                      palabras.append(palabra_actual)  # Agregar la palabra a la l
45                      palabra_actual = ''  # Reiniciar palabra_actual para la pr
46              else:
47                  palabra_actual += caracter  # Construir la palabra agregando car
48
49          if palabra_actual:  # Para asegurar que la  ltima  palabra de la l nea
50              palabras.append(palabra_actual)
51
52          return palabras  # Devolver la lista de palabras encontradas en la l ne
```

Figura 3: Código de Búsqueda por Diccionario en Python

3.2.3. Búsqueda por Diccionario

La primera función, "`construir_indice_con_diccionario`", se encarga de construir un índice de palabras a partir de un texto dado. Utiliza un diccionario donde cada clave representa una palabra única encontrada en el texto. Cada valor en el diccionario es un conjunto que contiene los números de línea donde esa palabra específica aparece. Durante la construcción del índice, se itera línea por línea a través del texto, se separan las palabras, se eliminan los espacios en blanco y se convierten todas las palabras a minúsculas para normalizarlas. Si una palabra no está presente en el índice, se agrega como nueva clave con un conjunto vacío y se añade el número de línea actual al conjunto asociado a esa palabra. Esta estructura de datos asegura que cada palabra se mapee eficientemente a todas las líneas donde ocurre, permitiendo búsquedas rápidas y eficientes de ocurrencias de palabras en el texto.

La segunda función, "`buscar_con_diccionario`", recibe como entrada un índice construido previamente y una palabra para buscar en el texto. La función verifica si la palabra está presente en el índice y, si es así, devuelve la lista de números de línea donde aparece la palabra. Si la palabra no está en el índice, la función devuelve una lista vacía.

Presentando el programa completo de Búsqueda por Diccionario:

```
1  def construir_indice_con_diccionario(texto):
2      indice = {}  # Diccionario para almacenar el  ndice
3
4      for num_linea, linea in enumerate(texto, start=1):
   # Iterar sobre cada l nea en el texto
5          palabras = linea.split()  # Obtener todas las palabras de la l nea
6
7          for palabra in palabras:  # Iterar sobre cada palabra en la l nea
8              palabra = palabra.strip().lower()
   # Convertir a min sculas y eliminar espacios
9
10             if palabra:  # Verificar si la palabra no est  vac a
11                 if palabra not in indice:  # Si la palabra no est  en el  ndice
12                     indice[palabra] = set()  # Utilizamos un conjunto para evita
13                 indice[palabra].add(num_linea)
   # Agregar el n mero de l nea al  ndice
14
15     return indice  # Devolver el diccionario  ndice  construido
```

Figura 4: Código de Búsqueda por Diccionario en Python

3.2.4. Programa Principal y Cliente Interactivo

El archivo `cliente_interactivo.py` implementa un cliente que interactúa con la interfaz que permite seleccionar y buscar patrones en diferentes archivos de texto utilizando dos métodos de búsqueda: fuerza bruta, búsqueda indexada y búsqueda con diccionarios. A continuación, se detalla su funcionamiento:

Carga de Archivos:

`cargar_archivo(nombre_archivo)`: Carga el contenido de un archivo de texto especificado y lo devuelve como una lista de líneas. La codificación del archivo se determina por su extensión.

Búsqueda Indexada:

`realizar_busqueda_indexada(tabla_hash, patron)`: Realiza una búsqueda indexada utilizando una tabla de hash. Inicia un contador de tiempo antes de realizar la búsqueda, busca las líneas que contienen el patrón en la `tabla_hash`, y registra el tiempo de ejecución en microsegundos. Luego, recorre las líneas encontradas para contar las ocurrencias exactas del patrón (ignorando mayúsculas y minúsculas) en cada línea. Finalmente, devuelve una lista de tuplas que contiene el número de línea, el contador de ocurrencias y la línea completa, junto con el tiempo de búsqueda.

Interfaz Interactiva:

`main()`: Proporciona una interfaz de línea de comandos para que el usuario seleccione un archivo y un método de búsqueda, ingresando un patrón a buscar y vea los resultados. El menú permite cambiar el archivo, seleccionar el método de búsqueda, y salir del programa.

Presentando el programa completo de Cliente Interactivo:

```
import time
from tabla_hash import TablaHash
from busqueda_fuerza_bruta import busqueda_fuerza_bruta
from busqueda_con_diccionario import construir_indice_con_diccionario

# Funci n para cargar el contenido de un archivo de texto l nea por l nea
def cargar_archivo(nombre_archivo):
    # Determinar codificaci n seg n la extensi n del archivo
    encoding = 'latin-1' if nombre_archivo.endswith('.tex') else 'utf-8'
    lineas = []
    # Abrir archivo y leer l nea por l nea
    with open(nombre_archivo, 'r', encoding=encoding) as archivo:
        for linea in archivo:
            lineas.append(linea)
    return lineas

# Funci n para realizar la b squeda utilizando el m todo de fuerza bruta
def realizar_busqueda_fuerza_bruta(texto, patron):
    # Iniciar contador de tiempo
    inicio = time.perf_counter_ns()
    # Llamar a la funci n de b squeda por fuerza bruta
    resultados = busqueda_fuerza_bruta(texto, patron)
    # Finalizar contador de tiempo y calcular duraci n de la b squeda en microsegundos
    fin = time.perf_counter_ns()
    tiempo = (fin - inicio) / 1000  # Convertir nanosegundos a microsegundos
    # Preparar resultados numerados para mostrar
    resultados_numerados = []
    for resultado in resultados:
        resultados_numerados.append((resultado[0] + 1, resultado[1], resultado[2]))
    # Retornar resultados numerados y tiempo de b squeda
    return resultados_numerados, tiempo

# Funci n para realizar la b squeda utilizando el m todo indexado con TablaHash
def realizar_busqueda_indexada(tabla_hash, patron):
    # Iniciar contador de tiempo
    inicio = time.perf_counter_ns()
    # Buscar l neas en la TablaHash seg n el patr n
    lineas_encontradas = tabla_hash.buscar(patron)
    # Finalizar contador de tiempo y calcular duraci n de la b squeda en microsegundos
    fin = time.perf_counter_ns()
    tiempo = (fin - inicio) / 1000  # Convertir nanosegundos a microsegundos
    # Inicializar lista para almacenar los resultados
    resultados = []

    # Para cada n mero de l nea encontrado
    for num_linea in lineas_encontradas:
```

```
47          # Obtener texto de la l nea utilizando el  ndice  base 1
48          linea_texto = tabla_hash.texto[num_linea - 1]  # Restamos 1 porque num_linea es el
49          # Contar ocurrencias exactas del patr n en la l nea (ignorando may sculas y mi
50          contador = sum(1 for palabra in linea_texto.split() if palabra.lower() == patron.lo
51          # Agregar resultado (n mero de l nea, contador, l nea completa) a la lista de r
52          resultados.append((num_linea, contador, linea_texto))
53      # Retornar resultados y tiempo de b squeda
54      return resultados, tiempo
55
56  # Funci n para realizar la b squeda utilizando el m todo con diccionario
57  def realizar_busqueda_con_diccionario(texto, patron):
58      # Construir  ndice  de palabras en el texto utilizando un diccionario
59      indice = construir_indice_con_diccionario(texto)
60      # Iniciar contador de tiempo
61      inicio = time.perf_counter_ns()
62      # Inicializar lista para almacenar los resultados
63      resultados = []
64
65      if patron in indice:
66          lineas_encontradas = indice[patron]
67      else:
68          lineas_encontradas = set()
69
70      for num_linea in lineas_encontradas:
71          linea_texto = texto[num_linea - 1]  # Obtener texto de la l nea utilizando el  nd
72          contador = 0  # Inicializar contador de ocurrencias del patr n en la l nea
73
74          palabras = linea_texto.split()  # Dividir la l nea en palabras
75
76          for palabra in palabras:
77              if palabra.strip().lower() == patron.lower():
78                  contador += 1
79
80          if contador > 0:
81              resultados.append((num_linea, linea_texto.strip(), contador))
82
83      # Finalizar contador de tiempo y calcular duraci n de la b squeda en microsegundos
84      fin = time.perf_counter_ns()
85      tiempo = (fin - inicio) / 1000  # Convertir nanosegundos a microsegundos
86      # Retornar resultados y tiempo de b squeda
87      return resultados, tiempo
88
89
90
91  # Funci n principal que ejecuta el programa interactivo de b squeda
92  def main():
93      while True:
94          # Mostrar men  para seleccionar archivo
95          print("\nSeleccione el archivo a utilizar:")
96          print("1. proyecto.tex")
97          print("2. hawking-stephen-historia-del-tiempo.txt")
98          print("3. biblia.txt")
99          print("4. Salir")
100         opcion_archivo = input("Opci n: ")
```

```
101
102         if opcion_archivo == '4':
103             break
104         elif opcion_archivo not in ['1', '2', '3']:
105             print("Opci n no v lida.")
106             continue
107
108         # Asignar nombre de archivo seg n la opci n seleccionada
109         if opcion_archivo == '1':
110             nombre_archivo = 'proyecto.tex'
111         elif opcion_archivo == '2':
112             nombre_archivo = 'hawking-stephen-historia-del-tiempo.txt'
113         elif opcion_archivo == '3':
114             nombre_archivo = 'biblia.txt'
115
116         # Cargar el contenido del archivo seleccionado como texto
117         texto = cargar_archivo(nombre_archivo)
118         # Inicializar la TablaHash con el texto del archivo seleccionado
119         tabla_hash = TablaHash(texto)
120
121         while True:
122             # Mostrar men  para seleccionar m todo de b squeda
123             print("\nSeleccione el m todo de b squeda:")
124             print("1. Fuerza Bruta")
125             print("2. B squeda Indexada")
126             print("3. B squeda con Diccionario")
127             print("4. Cambiar archivo")
128             print("5. Salir")
129             opcion_busqueda = input("Opci n: ")
130
131             if opcion_busqueda == '5':
132                 return
133             elif opcion_busqueda == '4':
134                 break
135             elif opcion_busqueda not in ['1', '2', '3']:
136                 print("Opci n no v lida.")
137                 continue
138
139             # Leer el patr n a buscar
140             patron = input("Ingrese el patr n a buscar: ")
141
142             # Realizar la b squeda seg n la opci n seleccionada
143             if opcion_busqueda == '1':
144                 resultados, tiempo = realizar_busqueda_fuerza_bruta(texto, patron)
145                 print(f"Tiempo de b squeda: {tiempo:.6f} microsegundos")
146                 print(f"Lineas encontradas: {len(resultados)}")
147                 print(f"Total de b squedas encontradas: {sum(resultado[2] for resultado in
148                 for resultado in resultados:
149                     pass
150                     #print(f"L nea {resultado[0]} | (Apariciones: {resultado[2]}): {result
151
152             elif opcion_busqueda == '2':
153                 resultados, tiempo = realizar_busqueda_indexada(tabla_hash, patron)
154                 print(f"Tiempo de b squeda: {tiempo:.6f} microsegundos")
```

```
155                     print(f"Lineas encontradas: {len(resultados)}")
156                     print(f"Total de b squedas encontradas: {sum(resultado[1] for resultado in
157                     for resultado in resultados:
158                         pass
159                         #print(f"L nea {resultado[0]} | (Apariciones: {resultado[1]}): {result
160 
161                 elif opcion_busqueda == '3':
162                     resultados, tiempo = realizar_busqueda_con_diccionario(texto, patron)
163                     print(f"Tiempo de b squeda: {tiempo:.6f} microsegundos")
164                     print(f"Lineas encontradas: {len(resultados)}")
165                     print(f"Total de b squedas encontradas: {sum(resultado[2] for resultado in
166                     for resultado in resultados:
167                         pass
168                         #print(f"L nea {resultado[0]} | (Apariciones: {resultado[2]}): {result
169 
170 if __name__ == "__main__":
171     main()
```

Figura 5: Código de Cliente Interactivo en Python

3.3. Diagrama de Estados

Se incluye el diagrama de estados que muestra de manera global el funcionamiento del programa. Presentando a continuación:

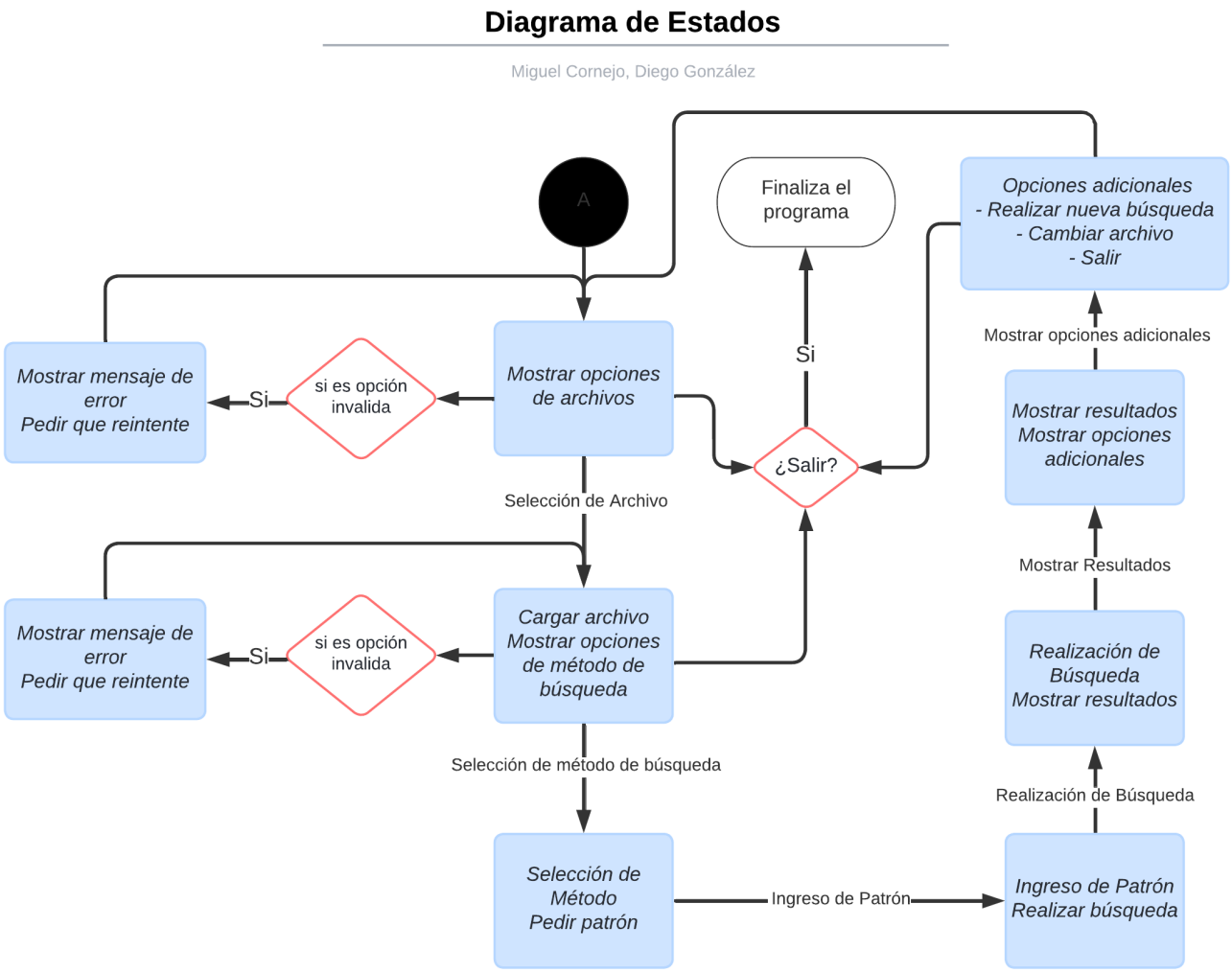

Figura 6: Diagrama de Estados

3.4. Implementación

Se mostrará la interpretación del pseudo código de los tres programas, los cuales son, `busqueda_fuerza_bruta.py`, `tabla_hash.py`, `buscar_con_diccionario.py` y `cliente_interactivo.py`. A continuación se muestra el pseudo código de cada uno de los programas de logartimos:

```
1  Función busqueda_fuerza_bruta(texto, patron):
2      resultados = Lista vacía
3      Para cada línea en texto:
4          contador = 0
5          Para cada posición en la línea:
6              Si la subcadena desde la posición hasta el final de la línea coincide
7  con el patrón:
8                  Incrementar el contador
9                  Agregar la posición y el contador a resultados
10     Devolver resultados
11
12 Función contar_patron(linea, patron):
13     contador = 0
14     longitud_patron = longitud del patrón
15     longitud_linea = longitud de la línea
16     Para cada posición en la línea:
17         Si la subcadena desde la posición hasta la posición + longitud_patron
18 coincide con el patrón:
19             Incrementar contador
20     Devolver contador
```

Figura 7: Pseudo-código de la búsqueda por fuerza bruta.

```
Clase TablaHash:
    Función inicializar(texto):
        self.vocabulario = {}  # Diccionario para mapear palabras a índices
        self.construir_indice(texto)  # Construir el índice al inicializar la clase

    Función construir_indice(texto):
        self.tabla = {}  # Diccionario para almacenar listas de líneas por índice
        self.texto = texto  # Guardar el texto completo como atributo de la instancia
        self.contador_palabras = 0  # Contador para asignar índices a palabras únicas
        num_linea = 1

        # Iterar sobre cada línea del texto
        Para cada línea en texto hacer:
            palabras = extraer_palabras(linea)  # Obtener lista de palabras en la línea
            Para cada palabra en palabras hacer:
                agregar(palabra, num_linea)  # Agregar palabra al índice
            num_linea += 1

    Función agregar(palabra, num_linea):
        Si palabra no está en self.vocabulario:  # Si la palabra no está en el vocabulario
            self.vocabulario[palabra] = self.contador_palabras  # Asignar un nuevo índice a la palabra
            self.tabla[self.contador_palabras] = []  # Inicializar una lista vacía para las líneas
            self.contador_palabras += 1  # Incrementar el contador de índices

        indice = self.vocabulario[palabra]  # Obtener el índice de la palabra
        Si num_linea no está en self.tabla[indice]:  # Si la línea no está en la lista del índice
            Añadir num_linea a self.tabla[indice]  # Agregar la línea al índice

    Función buscar(palabra):
        Si palabra está en self.vocabulario:  # Si la palabra está en el vocabulario
            indice = self.vocabulario[palabra]  # Obtener el índice asociado a la palabra
            Devolver self.tabla[indice]  # Devolver la lista de líneas asociadas al índice
        Sino:
            Devolver una lista vacía  # Devolver una lista vacía si la palabra no está en el vocabulario

    Función extraer_palabras(linea):
        palabra_actual = ''  # Variable para almacenar la palabra actual durante la iteración
        palabras = []  # Lista para almacenar todas las palabras encontradas en la línea

        # Iterar sobre cada caracter en la línea
        Para cada caracter en linea hacer:
            Si caracter es ' ' o caracter es '\n':  # Si el caracter es un espacio o salto de línea
                Si palabra_actual no está vacía:  # Si hay una palabra almacenada en palabra_actual
                    Añadir palabra_actual a palabras  # Agregar la palabra a la lista de palabras
                    palabra_actual = ''  # Reiniciar palabra_actual para la próxima palabra
            Sino:
```

```
47          Concatenar caracter a palabra_actual  # Construir la palabra agregando caracteres
48
49      Si palabra_actual no está vacía:  # Para asegurar que la última palabra de la línea se agregue
50          Añadir palabra_actual a palabras
51
52      Devolver palabras  # Devolver la lista de palabras encontradas en la línea
```

Figura 8: Pseudo-código de la Tabla de Hash.

```
1   Función construir_indice_con_diccionario(texto):
2       // Inicializar un diccionario vacío para almacenar el índice
3       indice = {}
4
5       // Inicializar el número de línea en 1
6       num_linea = 1
7
8       // Iterar sobre cada línea en el texto
9       Para cada línea en texto:
10          // Dividir la línea en palabras utilizando algún método que devuelva una lista de palabras
11          palabras = dividir_linea_en_palabras(linea)
12
13          // Iterar sobre cada palabra en la lista de palabras
14          Para cada palabra en palabras:
15              // Normalizar la palabra: convertir a minúsculas y eliminar espacios al inicio y final
16              palabra = normalizar(palabra)
17
18              // Verificar si la palabra no está vacía después de la normalización
19              Si palabra no está vacía entonces:
20                  // Verificar si la palabra ya está en el diccionario de índice
21                  Si palabra no está en indice entonces:
22                      // Agregar la palabra al diccionario de índice con un conjunto vacío
23                      indice[palabra] = conjunto_vacio()
24
25                  // Agregar el número de línea al conjunto de números de línea asociados a la palabra
26                  agregar_a_conjunto(indice[palabra], num_linea)
27
28          // Incrementar el número de línea para la siguiente iteración
29          num_linea = num_linea + 1
30
31      // Devolver el diccionario de índice construido
32      retornar indice
```

Figura 9: Pseudo-código de búsqueda con diccionario.

```
Importar módulo time
Desde tabla_hash Importar Clase TablaHash
Desde busqueda_fuerza_bruta Importar función busqueda_fuerza_bruta
Desde busqueda_con_diccionario Importar función construir_indice_con_diccionario

Función cargar_archivo(nombre_archivo):
    Si nombre_archivo termina con '.tex' entonces:
        encoding = 'latin-1'
    Sino:
        encoding = 'utf-8'

    lineas = []
    Abrir archivo nombre_archivo en modo lectura con encoding
    Para cada línea en archivo hacer:
        Agregar línea a la lista lineas
    Retornar lineas

Función realizar_busqueda_fuerza_bruta(texto, patron):
    inicio = Obtener tiempo actual en nanosegundos
    resultados = Llamar función busqueda_fuerza_bruta con parámetros texto y patron
    fin = Obtener tiempo actual en nanosegundos
    tiempo = (fin - inicio) / 1000
    resultados_numerados = []
    Para cada resultado en resultados hacer:
        Agregar (resultado[0] + 1, resultado[1], resultado[2]) a resultados_numerados
    Retornar resultados_numerados, tiempo

Función realizar_busqueda_indexada(tabla_hash, patron):
    inicio = Obtener tiempo actual en nanosegundos
    lineas_encontradas = Llamar método buscar de tabla_hash con parámetro patron
    fin = Obtener tiempo actual en nanosegundos
    tiempo = (fin - inicio) / 1000
    resultados = []
    Para cada num_linea en lineas_encontradas hacer:
        linea_texto = Obtener texto de tabla_hash en índice num_linea - 1
        contador = Contar ocurrencias exactas de patron en linea_texto (ignorando mayúsculas y minúsculas)
        Agregar (num_linea, contador, linea_texto) a resultados
    Retornar resultados, tiempo

Función realizar_busqueda_con_diccionario(texto, patron):
    indice = Llamar función construir_indice_con_diccionario con parámetro texto
    inicio = Obtener tiempo actual en nanosegundos
    resultados = []
    Si patron está en indice entonces:
        lineas_encontradas = Obtener valor de patron en indice
    Sino:
```

```
47          lineas_encontradas = Conjunto vacío
48
49      Para cada num_linea en lineas_encontradas hacer:
50          linea_texto = Obtener texto de texto en índice num_linea - 1
51          contador = 0
52          palabras = Dividir linea_texto en palabras
53          Para cada palabra en palabras hacer:
54              Si palabra.strip().lower() es igual a patron.lower() entonces:
55                  Incrementar contador en 1
56
57          Si contador > 0 entonces:
58              Agregar (num_linea, linea_texto.strip(), contador) a resultados
59
60      fin = Obtener tiempo actual en nanosegundos
61      tiempo = (fin - inicio) / 1000
62      Retornar resultados, tiempo
63
64  Función principal main():
65      Mientras Verdadero hacer:
66          Imprimir "Seleccione el archivo a utilizar:"
67          Imprimir "1. proyecto.tex"
68          Imprimir "2. hawking-stephen-historia-del-tiempo.txt"
69          Imprimir "3. biblia.txt"
70          Imprimir "4. Salir"
71          Leer opcion_archivo desde entrada estándar
72
73          Si opcion_archivo es igual a '4' entonces:
74              Romper
75
76          Sino Si opcion_archivo no está en ['1', '2', '3'] entonces:
77              Imprimir "Opción no válida."
78              Continuar
79
80          Sino:
81              Si opcion_archivo es igual a '1' entonces:
82                  nombre_archivo = 'proyecto.tex'
83              Sino Si opcion_archivo es igual a '2' entonces:
84                  nombre_archivo = 'hawking-stephen-historia-del-tiempo.txt'
85              Sino:
86                  nombre_archivo = 'biblia.txt'
87
88              texto = Llamar función cargar_archivo con parámetro nombre_archivo
89              tabla_hash = Instanciar TablaHash con parámetro texto
90
91              Mientras Verdadero hacer:
92                  Imprimir "Seleccione el método de búsqueda:"
```

```
93          Imprimir "1. Fuerza Bruta"
94          Imprimir "2. Búsqueda Indexada"
95          Imprimir "3. Búsqueda con Diccionario"
96          Imprimir "4. Cambiar archivo"
97          Imprimir "5. Salir"
98          Leer opcion_busqueda desde entrada estándar
99
100         Si opcion_busqueda es igual a '5' entonces:
101           Retornar
102
103         Sino Si opcion_busqueda es igual a '4' entonces:
104           Romper
105
106         Sino Si opcion_busqueda no está en ['1', '2', '3'] entonces:
107           Imprimir "Opción no válida."
108           Continuar
109
110         Sino:
111           Leer patron desde entrada estándar
112
113           Si opcion_busqueda es igual a '1' entonces:
114             resultados, tiempo = Llamar función realizar_busqueda_fuerza_bruta con parámetros texto y patron
115             Imprimir "Tiempo de búsqueda:", tiempo, "microsegundos"
116             Imprimir "Lineas encontradas:", longitud de resultados
117             Imprimir "Total de búsquedas encontradas:", Suma de resultado[2] para resultado en resultados
118             Para cada resultado en resultados hacer:
119               No hacer nada  # Comentar esta línea si se desea imprimir los resultados individualmente
120
121           Sino Si opcion_busqueda es igual a '2' entonces:
122             resultados, tiempo = Llamar función realizar_busqueda_indexada con parámetros tabla_hash y patron
123             Imprimir "Tiempo de búsqueda:", tiempo, "microsegundos"
124             Imprimir "Lineas encontradas:", longitud de resultados
125             Imprimir "Total de búsquedas encontradas:", Suma de resultado[1] para resultado en resultados
126             Para cada resultado en resultados hacer:
127               No hacer nada  # Comentar esta línea si se desea imprimir los resultados individualmente
128
129           Sino:
130             resultados, tiempo = Llamar función realizar_busqueda_con_diccionario con parámetros texto y patron
131             Imprimir "Tiempo de búsqueda:", tiempo, "microsegundos"
132             Imprimir "Lineas encontradas:", longitud de resultados
133             Imprimir "Total de búsquedas encontradas:", Suma de resultado[2] para resultado en resultados
134             Para cada resultado en resultados hacer:
135               No hacer nada  # Comentar esta línea si se desea imprimir los resultados individualmente
```

Figura 10: Pseudo-código del Cliente Interactivo.

3.5. Modo de uso

El programa `cliente_interactivo.py` ofrece una interfaz sencilla para buscar patrones en archivos de texto. Para utilizarlo, primero asegúrarse de tener Python instalado la versión 3.13 como mínima en el sistema. Luego, abrir una terminal o línea de comandos y navegue hasta el directorio donde se encuentra el archivo `cliente_interactivo.py`. Ejecute el programa.

Una vez en ejecución, el programa mostrará una lista de archivos de texto disponibles para buscar. Seleccionando el archivo deseado ingresando el número correspondiente y presionando Enter. Luego elegir el método de búsqueda: fuerza bruta o búsqueda indexada, ingresando el número correspondiente.

Después de seleccionar el método de búsqueda, se le pedirá el ingreso del patrón que desea buscar en el archivo. Ingresando el patrón y presionar Enter. El programa realizará la búsqueda y mostrará los resultados, incluido el tiempo de búsqueda, el número de líneas encontradas y el total de ocurrencias del patrón. Para cada línea encontrada, se mostrará su número, el número de apariciones del patrón en la línea y el contenido de la línea.

Después de mostrar los resultados, se ofrecerán opciones adicionales: realizar una nueva búsqueda, cambiar el archivo o salir del programa. Simplemente seleccionar la opción deseada ingresando el número correspondiente.

Modo de compilación:

Para compilar el programa en los computadores de la Universidad u otro lugar, primero asegurarse de que Python esté instalado en los computadores. Luego, copiar todos los archivos relacionados con el proyecto en el directorio donde se desea compilar el programa.

Una vez copiados los archivos, abrir una terminal o línea de comandos en el directorio y ejecutar el programa escribiendo:

```
python fuente/src/cliente_interactivo.py
```

Siga las instrucciones en la interfaz para realizar búsquedas según sea necesario.

3.6. Pruebas

Se realizaron diversas pruebas utilizando distintos archivos de texto y patrones de búsqueda para evaluar el rendimiento y la efectividad de los algoritmos implementados. A continuación se muestran los resultados obtenidos:

Archivo	Palabra o Patrón	Tiempo de Búsqueda (microsegundos)		
		Fuerza Bruta	Búsqueda Indexada	Búsqueda con Diccionario
proyecto.tex	a	966,2	5,6	56,5
	e	911,9	4,6	1,2
	y	871,4	7,6	81,6
	de	1.004,2	7,6	157
	que	953,9	7,2	69,5
	el	1.032,3	8,7	145,9
hawking.txt	a	33.345,6	7,5	2880,1
	e	33.498,8	7,7	112,9
	y	31.630,2	6,1	2918
	de	38.196,9	11,0	8274,3
	que	36.385,4	6,6	6262,7
	el	37.253,8	9,2	4821,3
biblia.txt	a	309.251,5	9,8	41.554,6
	e	303.785,2	7,9	1.803,0
	y	291.492,2	6,7	79.472,8
	de	342.287,9	8,7	76.657,6
	que	332.632,1	11,7	43.632,7
	el	341.542,9	8,2	37.458,9

Cuadro 1: Tiempo de búsqueda para diferentes archivos y palabras

Presentando el detalle de cada uno de los archivos y de las búsquedas:

Archivo	Palabra	Tipo de Búsqueda	Tiempo de Búsqueda (microsegundos)	Búsquedas encontradas	Líneas encontradas
proyecto.tex	a	Fuerza Bruta	1270,1	928	187
proyecto.tex	a	Búsqueda Indexada	7,8	22	22
proyecto.tex	a	Búsqueda con Diccionario	51,7	22	22

Cuadro 2: Tiempo de búsqueda del archivo "proyecto.txt"

Archivo	Palabra	Tipo de Búsqueda	Tiempo de Búsqueda (microsegundos)	Búsquedas encontradas	Líneas encontradas
hawking.txt	a	Fuerza Bruta	34106,4	38.187	5.480
hawking.txt	a	Búsqueda Indexada	7,1	1.152	1.054
hawking.txt	a	Búsqueda con Diccionario	2810	1.207	1.109

Cuadro 3: Tiempo de búsqueda del archivo "hawking-stephen-historia-del-tiempo.txt"

Archivo	Palabra	Tipo de Búsqueda	Tiempo de Búsqueda (microsegundos)	Búsquedas encontradas	Líneas encontradas
biblia.txt	a	Fuerza Bruta	325163,3	328.581	63.629
biblia.txt	a	Búsqueda Indexada	9,5	19.854	16.487
biblia.txt	a	Búsqueda con Diccionario	42919,2	20.116	16.747

Cuadro 4: Tiempo de búsqueda del archivo "biblia.txt"

4. Discusión

Se evaluaron los datos de los tres métodos de búsqueda para determinar su eficacia en la búsqueda de patrones en textos: fuerza bruta, búsqueda indexada mediante tablas de hash y búsqueda por diccionario. Los resultados obtenidos muestran diferencias significativas en el rendimiento de cada método, dependiendo del tamaño del archivo y la naturaleza del patrón buscado, el cual comparandolos es la siguiente:

- El archivo `proyecto.tex` es relativamente más corto y contiene un conjunto diverso de palabras comunes en español, como "a", "e", "y", "de", "que" y "el", la búsqueda indexada y la búsqueda con diccionario son más efectivas en términos de tiempo de búsqueda en comparación con la fuerza bruta. Esto se debe a que el costo de construir el índice o el diccionario inicialmente se compensa con una búsqueda más rápida y eficiente, especialmente cuando el texto es corto y el patrón de búsqueda es común. Por otro lado, la fuerza bruta puede funcionar mejor en archivos muy pequeños o cuando el patrón de búsqueda es único y la construcción de un índice no es necesaria.

- El archivo `hawking.txt` es más largo y complejo, lo que se refleja en los tiempos de búsqueda más largos en comparación con `proyecto.tex`. Sin embargo, tanto la búsqueda indexada como la búsqueda con diccionario siguen siendo más rápidas que la fuerza bruta. Esto indica que la creación de un índice o un diccionario puede mejorar el rendimiento de búsqueda de patrones incluso en archivos más grandes y complejos.

- El archivo `biblia.txt` es el más pesado y largo de los tres, con tiempos de búsqueda aún más largos que los otros dos. Sin embargo, en términos de eficiencia, la búsqueda indexada superan a la fuerza bruta y la búsqueda de diccionario. Esto destaca el uso de estructuras de datos optimizadas, especialmente en archivos grandes, donde la fuerza bruta puede volverse imposible debido a la complejidad y el tiempo de procesamiento requeridos.

La eficacia de cada método de búsqueda depende del patrón de búsqueda, el tamaño y la complejidad del texto. En la mayoría de los casos, la búsqueda indexada y la búsqueda con diccionario son preferibles, ya que ofrecen tiempos de búsqueda más cortos y eficientes, especialmente en archivos más grandes y complejos. Por último, la fuerza bruta puede ser mejor para archivos muy pequeños o cuando el patrón de búsqueda es único y la creación de un índice no es necesaria.

Fuerza Bruta El método revisa línea por línea del texto buscando un patrón, lo que resulta en una complejidad $O(n \cdot m)$ donde n es el número de líneas y m la longitud del patrón, el cual sería $O(n^2)$. Es sencillo pero ineficiente para archivos grandes debido al aumento significativo en el tiempo de búsqueda conforme crece el archivo y el patrón.

Búsqueda Indexada La búsqueda indexada utiliza una tabla de hash para almacenar las posiciones de las palabras en el texto, permitiendo búsquedas rápidas $O(1)$ una vez construido el índice. Es altamente eficiente para archivos grandes, aunque la construcción inicial del índice puede ser costosa en tiempo y memoria, especialmente para textos extensos con muchas palabras únicas.

Búsqueda por Diccionario Emplea un diccionario que mapea cada palabra única a las líneas donde aparece en el texto. Proporciona una búsqueda eficiente, similar a la búsqueda indexada, centrada en encontrar palabras específicas en lugar de patrones complejos. Los tiempos de búsqueda son competitivos con la búsqueda indexada, dependiendo de la estructura del diccionario y la implementación de la búsqueda.

5. Conclusión

El proyecto ha demostrado que aplicando estrategias de búsqueda efectivas a la manipulación de textos es crucial, especialmente cuando se trabaja con grandes cantidades de datos. Se ha demostrado mediante comparaciones experimentales entre métodos como la fuerza bruta, la búsqueda indexada y la búsqueda por diccionario que el uso de estructuras de datos como las tablas de hash pueden optimizar significativamente los tiempos de búsqueda y mejorar la eficiencia en la gestión de una gran cantidad de información. Estos resultados destacan la importancia de investigar y aplicar técnicas avanzadas en el ámbito de las estructuras de datos y los algoritmos. Estos hallazgos establecen una base sólida para futuros desarrollos en la optimización de búsqueda de datos en una variedad de aplicaciones informáticas.

6. Anexos

[1] Algoritmo de fuerza bruta de Python. foro ayuda. https://foroayuda.es/algoritmo-de-fuerza-bruta-de-python/

[2] Tablas Hash. UVM https://es.slideshare.net/slideshow/15-tablas-hash/122273728

[3] Diccionarios Python: Guía + Ejercicios 2024. bigbaydata. https://www.bigbaydata.com/diccionarios-python/