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 Nicolas Arellano Benjamin Sepulveda Diego González
e-mail:	mcornejoc4@uft.edu narellanoc@uft.edu bsepulveda@uft.edu dgonzalez@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:

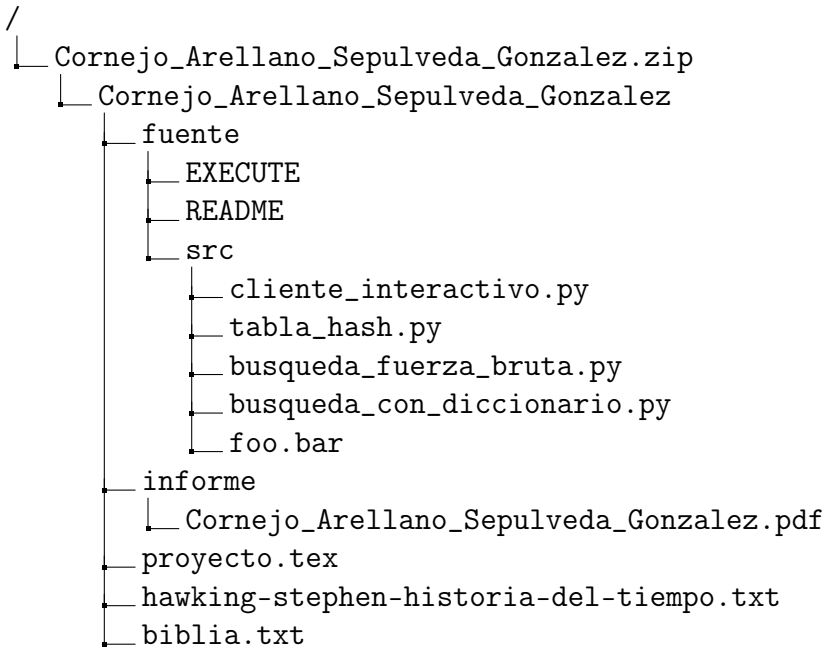

Figura 1: Estructura del archivo.

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 = []
3      indice_linea = 0
4
5      while indice_linea < len(texto):
6          linea = texto[indice_linea]
7          contador = contar_patron(linea, patron)
8          if contador > 0:
9              resultados.append((indice_linea, linea, contador))
10         indice_linea += 1
11
12     return resultados
13
14
15 def contar_patron(linea, patron):
16     contador = 0
17     longitud_patron = len(patron)
18     longitud_linea = len(linea)
19
20     for i in range(longitud_linea - longitud_patron + 1):
21         if linea[i:i+longitud_patron] == patron:
22             contador += 1
23
24     return contador
```

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 datos de tabla hash para optimizar la búsqueda de palabras en un texto. En el inicialización, crea un diccionario para almacenar las palabras y sus índices, así como otro diccionario para guardar las líneas de texto donde aparecen las palabras. Utiliza métodos como agregar para insertar palabras en la tabla, buscar para encontrar las líneas donde una palabra dada aparece, y "`construir_indice`" para generar el índice hash del texto proporcionado. Además, incluye el método "`busqueda_indexada`" que actúa como un wrapper (que envuelve o encapsula una funcionalidad o componente, que facilita su uso o proporciona una interfaz más fácil para interactuar) para el método buscar,

permitiendo un interfaz más lógico para realizar búsquedas indexadas. Permitiendo ofrecer una forma eficiente y optimizada de gestionar grandes volúmenes de datos de texto, facilitando la búsqueda y recuperación de información específica en un contexto computacional.

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

```
class TablaHash:
    def __init__(self, texto):
        self.tabla = {}
        self.vocabulario = {}
        self.contador_palabras = 0
        self.construir_indice(texto)

    def agregar(self, palabra, linea):
        if palabra not in self.vocabulario:
            self.vocabulario[palabra] = self.contador_palabras
            self.tabla[self.contador_palabras] = []
            self.contador_palabras += 1
        indice = self.vocabulario[palabra]
        self.tabla[indice].append(linea)

    def buscar(self, palabra):
        if palabra in self.vocabulario:
            indice = self.vocabulario[palabra]
            return self.tabla[indice]
        else:
            return []

    def busqueda_indexada(self, palabra):
        return self.buscar(palabra)

    def construir_indice(self, texto):
        # Primera pasada para construir el vocabulario
        for linea in texto:
            palabra_actual = ''
            for caracter in linea:
                if caracter == ' ' or caracter == '\n':
                    if palabra_actual:
                        self.agregar(palabra_actual, linea)
                        palabra_actual = ''
                else:
                    palabra_actual += caracter
            if palabra_actual:
                self.agregar(palabra_actual, linea)
```

Figura 3: Código de Tabla hash en Python

3.2.3. Búsqueda por Diccionario

La primera función, "`construir_indice_con_diccionario`", recibe como entrada un texto dividido en líneas y devuelve un diccionario que mapea cada palabra única en el texto a una lista de números de línea donde aparece la palabra. La función itera sobre cada línea del texto y divide la línea en palabras, ignorando los espacios en blanco y los saltos de línea. Luego, para cada palabra, actualiza el diccionario de índice, agregando el número de línea actual a la lista de ocurrencias de esa palabra en el texto. Finalmente, la función devuelve el diccionario de índice construido.

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 = {}
3      num_linea = 0
4      for linea in texto:
5          palabra_actual = ''
6          for caracter in linea:
7              if caracter == '␣' or caracter == '\n':
8                  if palabra_actual:
9                      if palabra_actual not in indice:
10                         indice[palabra_actual] = []
11                     indice[palabra_actual].append(num_linea + 1)
12                     palabra_actual = ''
13             else:
14                 palabra_actual += caracter
15         if palabra_actual:
16             if palabra_actual not in indice:
17                 indice[palabra_actual] = []
18             indice[palabra_actual].append(num_linea + 1)
19         num_linea += 1
20     return indice
21
22 def buscar_con_diccionario(indice, palabra):
23     if palabra in indice:
24         return indice[palabra]
25     else:
26         return []
```

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 y búsqueda indexada. 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(texto, patron)`: Crea un índice basado en tablas de hash para las palabras del texto. Luego, utiliza este índice para buscar el patrón de manera eficiente. Registra el tiempo de ejecución y devuelve los resultados y el tiempo en microsegundos.

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:

```
1  import time
2  from busqueda_fuerza_bruta import busqueda_fuerza_bruta
3  from tabla_hash import TablaHash
4
5  def cargar_archivo(nombre_archivo):
6      encoding = 'latin-1' if nombre_archivo.endswith('.tex') else 'utf-8'
7      with open(nombre_archivo, 'r', encoding=encoding) as archivo:
8          return archivo.readlines()
9
10 def realizar_busqueda_fuerza_bruta(texto, patron):
11     inicio = time.perf_counter_ns()
12     resultados = busqueda_fuerza_bruta(texto, patron)
13     fin = time.perf_counter_ns()
14     tiempo = (fin - inicio) / 1000  # Microsegundos
15     resultados_numerados = []
16     for resultado in resultados:
17         resultados_numerados.append((resultado[0] + 1, resultado[1], resultado[2]))
18     return resultados_numerados, tiempo
19
20 def realizar_busqueda_indexada(texto, patron):
21     indice = TablaHash(texto)  # Construir un nuevo  ndice  cada vez
```

```
22      inicio = time.perf_counter_ns()
23      resultados = []
24      lineas = indice.buscar(patron)  #M todo buscar de TablaHash
25      indice_linea = 1  # Contador de  ndice  de l nea
26      for linea in lineas:
27          contador = 0
28          longitud_linea = len(linea)
29          longitud_patron = len(patron)
30          i = 0
31          while i < longitud_linea:
32              if linea[i:i+longitud_patron] == patron:
33                  contador += 1
34                  i += longitud_patron
35              else:
36                  i += 1
37          resultados.append((indice_linea, linea, contador))
38          indice_linea += 1  #+1 el contador de  ndice  de l nea
39      fin = time.perf_counter_ns()
40      return resultados, (fin - inicio) / 1000  # Microsegundos
41
42  def main():
43      while True:
44          print("\nSeleccione el archivo a utilizar:")
45          print("1. proyecto.tex")
46          print("2. hawking-stephen-historia-del-tiempo.txt")
47          print("3. biblia.txt")
48          print("4. Salir")
49          opcion_archivo = input("Opci n: ")
50
51          if opcion_archivo == '4':
52              break
53          elif opcion_archivo not in ['1', '2', '3']:
54              print("Opci n no v lida.")
55              continue
56
57          if opcion_archivo == '1':
58              nombre_archivo = 'proyecto.tex'
59          elif opcion_archivo == '2':
60              nombre_archivo = 'hawking-stephen-historia-del-tiempo.txt'
61          elif opcion_archivo == '3':
62              nombre_archivo = 'biblia.txt'
63
64          texto = cargar_archivo(nombre_archivo)
65
66          while True:
67              print("\nSeleccione el m todo de b squeda:")
68              print("1. Fuerza Bruta")
69              print("2. B squeda Indexada")
70              print("3. Cambiar archivo")
71              print("4. Salir")
72              opcion_busqueda = input("Opci n: ")
73
74              if opcion_busqueda == '4':
75                  return
```

```
76              elif opcion_busqueda == '3':
77                  break
78              elif opcion_busqueda not in ['1', '2']:
79                  print("Opci n no v lida.")
80                  continue
81
82              patron = input("Ingrese el patr n a buscar: ")
83
84              if opcion_busqueda == '1':
85                  resultados, tiempo = realizar_busqueda_fuerza_bruta(texto, patron)
86              elif opcion_busqueda == '2':
87                  resultados, tiempo = realizar_busqueda_indexada(texto, patron)
88
89              lineas_encontradas = len(resultados)
90              busquedas_encontradas = sum(resultado[2] for resultado in resultados)
91
92              print(f"Tiempo de b squeda: {tiempo:.6f} microsegundos")
93              print(f"Lineas encontradas: {lineas_encontradas}")
94              print(f"Total de b squedas encontradas: {busquedas_encontradas} \n")
95              for resultado in resultados:
96                  print(f"L nea {resultado[0]} | (Apariciones: {resultado[2]}): {resultado[1
97
98  if __name__ == "__main__":
99      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:

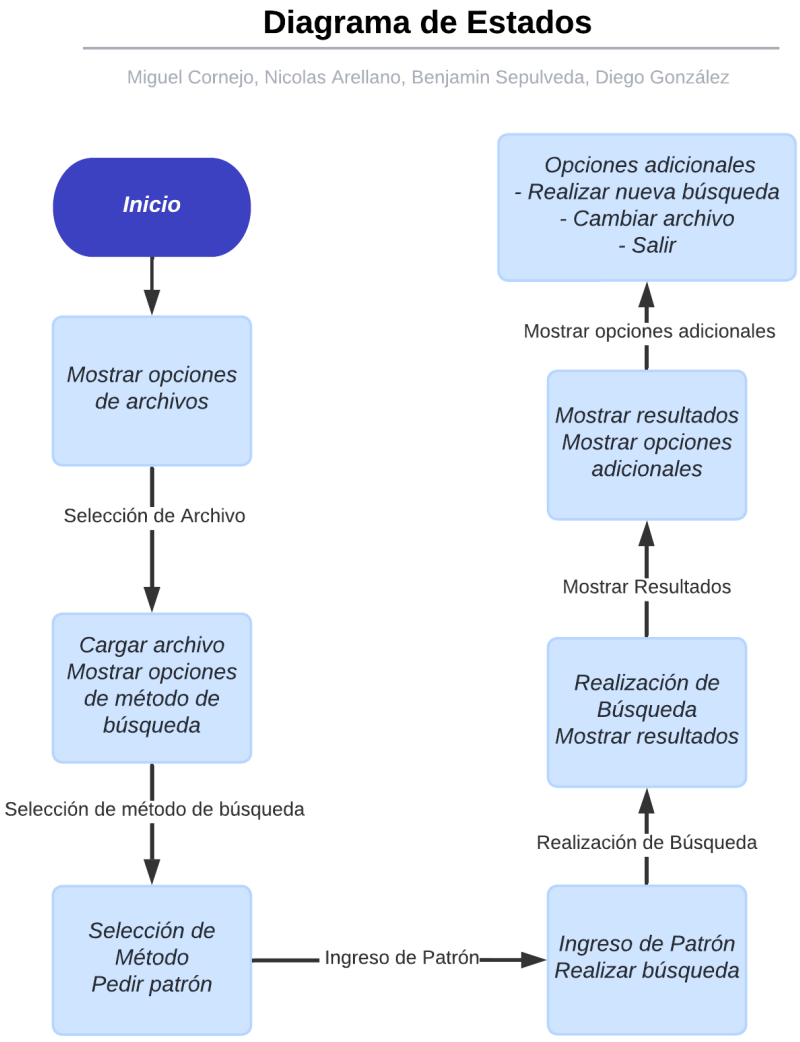

Figura 6: Diagrama de Estados

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:

```
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 __init__(self, texto):
        Crear un diccionario para la tabla hash
        Crear un diccionario para el vocabulario
        Inicializar el contador de palabras en 0
        Construir el índice del texto

    Función agregar(self, palabra, linea):
        Si la palabra no está en el vocabulario:
            Agregar la palabra al vocabulario con el contador de palabras como índice
            Agregar una lista vacía en la tabla hash con el contador de palabras como
clave
            Incrementar el contador de palabras
        Obtener el índice de la palabra desde el vocabulario
        Agregar la línea a la lista en la tabla hash en el índice obtenido

    Función buscar(self, palabra):
        Si la palabra está en el vocabulario:
            Obtener el índice de la palabra desde el vocabulario
            Devolver la lista en la tabla hash en el índice obtenido
        Devolver una lista vacía si la palabra no está en el vocabulario

    Función busqueda_indexada(self, palabra):
        Devolver el resultado de buscar(palabra)

    Función construir_indice(self, texto):
        Para cada línea en el texto:
            Separar la línea en palabras
            Para cada palabra en la línea:
                Agregar la palabra y la línea al índice hash
```

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

```
1  Función construir_indice_con_diccionario(texto):
2      indice = {}                    // Crear un diccionario vacío para el índice
3      num_linea = 0                  // Inicializar el número de línea en 0
4      Por cada línea en el texto:
5          palabra_actual = ''        // Inicializar la palabra actual como vacía
6          Por cada caracter en la línea:
7              Si el caracter es un espacio o un salto de línea:
8                  Si la palabra actual no está vacía:
9                      Si la palabra actual no está en el índice:
10                         Agregar la palabra al índice con una lista vacía como valor
11                     Agregar el número de línea a la lista de ocurrencias de la palabra
12                     Reiniciar la palabra actual a vacía
13             De lo contrario:
14                 Concatenar el caracter a la palabra actual
15         Si la palabra actual no está vacía:
16             Si la palabra actual no está en el índice:
17                 Agregar la palabra al índice con una lista vacía como valor
18             Agregar el número de línea a la lista de ocurrencias de la palabra
19         Incrementar el número de línea en 1
20     Devolver el índice generado
21
22 Función buscar_con_diccionario(indice, palabra):
23     Si la palabra está en el índice:
24         Devolver la lista de números de línea donde aparece la palabra
25     De lo contrario:
26         Devolver una lista vacía
```

Figura 9: Pseudo-código de búscqueda con diccionario.

```
Importar time
Importar busqueda_fuerza_bruta
Importar tabla_hash

Función cargar_archivo(nombre_archivo):
    encoding = determinar_codificación(nombre_archivo)
    abrir el archivo con nombre_archivo en modo lectura con la codificación encoding
    leer todas las líneas del archivo
    cerrar el archivo
    devolver las líneas leídas

Función realizar_busqueda_fuerza_bruta(texto, patron):
    iniciar temporizador
    resultados = busqueda_fuerza_bruta.busqueda_fuerza_bruta(texto, patron)
    detener temporizador
    calcular tiempo de ejecución
    numerar resultados para mostrar número de línea
    devolver resultados y tiempo

Función realizar_busqueda_indexada(texto, patron):
    construir un nuevo índice basado en tablas de hash para el texto
    iniciar temporizador
    resultados = tabla_hash.TablaHash(texto).busqueda_indexada(patron)
    detener temporizador
    calcular tiempo de ejecución
    devolver resultados y tiempo

Función principal main():
    mientras verdadero:
        mostrar opciones de archivo disponibles
        leer la opción del archivo
        si la opción es "Salir":
            salir del bucle principal
        si la opción no es válida:
            mostrar mensaje de error y continuar con el siguiente ciclo
        cargar el archivo correspondiente
        mientras verdadero:
            mostrar opciones de método de búsqueda disponibles
            leer la opción del método de búsqueda
            si la opción es "Salir":
                salir del bucle interno
            si la opción es "Cambiar archivo":
                salir del bucle interno
            si la opción no es válida:
                mostrar mensaje de error y continuar con el siguiente ciclo
            leer el patrón de búsqueda
            si la opción es "Fuerza Bruta":
                realizar búsqueda por fuerza bruta
            si la opción es "Búsqueda Indexada":
                realizar búsqueda indexada
            contar el número de líneas y ocurrencias totales del patrón
            mostrar los resultados de la búsqueda
```

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 escribiendo python `cliente_interactivo.py` y presione Enter.

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. A continuación, 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:

Cuadro 1: Tiempo de búsqueda para diferentes archivos y palabras

Archivo	Palabra o Patrón	Tiempo de Búsqueda (microsegundos)			Resultados	
		Fuerza Bruta	Búsqueda Indexada	Búsqueda con Diccionario	Búsquedas encontradas	Líneas encontradas
proyecto.tex	a	966,2	135,9	739,1	928	187
	e	911,9	1,6	732,3	1.270	217
	y	871,4	180,5	753,7	53	43
	de	1.004,2	594,8	788,2	197	123
	que	953,9	237	750	44	38
	el	1.032,3	332,1	758,5	82	64
hawking.txt	a	33.345,6	8.914,7	29.124	38.187	5.480
	e	33.498,8	239,8	28.584,7	46.976	5.459
	y	31.630,2	8.184,4	30.089,1	2.072	1.733
	de	38.196,9	36.337,9	31.937,1	8160	4.345
	que	36.385,4	22.072,2	31.011,3	3101	.2521
	el	37.253,8	14.740,8	29.641,3	4.370	3.056
biblia.txt	a	30.1962,3	109.921,7	297.437,5	328.581	63.629
	e	303.785,2	2.846,4	269.207,6	378.781	64.119
	y	291.492,2	18.0037,9	282.665,9	44.456	32.518
	de	342.287,9	28.4786,1	272.796,8	67.657	41.522
	que	332.632,1	120.399,3	279.055,3	26872	22.527
	el	341.542,9	104.938,4	277.111	37.678	27.972

4. Discusión

Dado los datos del tiempo y de las palabras y líneas encontradas, se mostrarán la discusión de los tres archivos diferentes para compararlos:

- 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 y la búsqueda con diccionario superan a la fuerza bruta. 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.

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.