UMAG Universidad de Magallanes | Facultad de Ingeniería

Fly: Motor de búsqueda

Franco Aguilar[†], **Milton Hernandez**[†], **Iván Mansilla**[†] **y Ayrton Morrison**[†]

[†] *Universidad de Magallanes*

Este informe fue compilado el 19 de noviembre de 2024

Resumen

Fly es un sistema de recuperación de información que indexa documentos de texto plano enlazados entre sí. Para esto usa diferentes estructuras de datos que facilitan una rápida recopilación de información, la búsqueda y la navegación entre los archivos. En el presente informe encontrará una descripción detallada de la implementación de Fly.

Keywords: *Índice invertido, Listas enlazadas simples, PageRank, Tablas hash*

Índice

1. Introducción

Un motor de búsqueda es una herramienta que permite buscar y recuperar datos en una colección de archivos.

Particularmente, *Fly* es un motor de búsqueda que indexa documentos enlazados entre sí mediante *WikiLinks*[1]. De estos archivos este motor extrae todas las palabras *relevantes* permitiendo al usuario buscar una palabra y encontrar las coincidencias de la misma a lo largo del sistema de archivos. Para llevar a cabo esta tarea, Fly utiliza técnicas como el *índice invertido* (para la indexación de palabras) y el *PageRank* (para la jerarquización de los archivos analizados).

Las principales funciones de **Fly** son:

1. Generar un grafo con los archivos de un directorio, almacenando en cada nodo un archivo y relacionándolo con los archivos que lo enlazan y aquellos a los que se enlaza.
2. Generar un índice invertido para almacenar la información de cada palabra relevante en el sistema de archivos.
3. Calcular el nivel de importancia de cada uno de los archivos en el grafo, utilizando el algoritmo de PageRank.
4. Realizar búsquedas eficientes sobre el índice invertido y el grafo permitiendo al usuario encontrar coincidencias entre palabras y sus ubicaciones en cada archivo donde aparecen.

Todas estas funciones se implementaron mediante el uso de estructuras de datos tales como grafos, listas enlazadas y tablas hash.

El presente informe describe el proceso de desarrollo e implementación de Fly, detallando las decisiones de diseño y las técnicas utilizadas para crear un sistema eficiente de busqueda y recuperación de información.

2. Objetivo principal

Crear un motor de búsqueda eficiente y rápido mediante el uso de estructuras de datos que permitan un correcto balance entre el uso de memoria y el tiempo de procesamiento.

2.1. Objetivos secundarios

1. **Utilización de estructura de datos:** Crear estructuras de datos eficientes para almacenar y procesar información.
2. **Optimización de código y eficiencia:** Optimizar el código para mejorar su eficiencia y rendimiento, evitando pérdidas de memoria y priorizando el tiempo de respuesta.
3. **Trabajo en equipo:** Reforzar el trabajo en equipo y la correcta comunicación entre los miembros del mismo, manteniendo correctos estándares de codificación para una eficiente implementación del programa.

[1] Un WikiLink es un tipo de enlace entre documentos de texto usado en la aplicación Obsidian.md consistente en el siguiente formato: `[[nombre del archivo enlazado|Alias para el archivo de ser necesario]]`. Este formato tiene diversas facilidades para un sistema de archivos con las características deseadas razón por la cuál fue utilizado en este proyecto.

3. Planteamiento del desarrollo del proyecto

Antes de iniciar con el desarrollo del proyecto, se procedió con una investigación sobre las estructuras de datos que se utilizarían basados en las recomendaciones realizadas previo desarrollo del mismo.

Los cuatro pilares sobre los que se sustenta el presente proyecto son los siguientes:

- Lectura y manejo de los archivos a procesar.
- Grafos como representación de las relaciones entre archivos.
- Índice invertido como estructura de datos para almacenar las palabras relevantes.
- PageRank como algoritmo para calcular la importancia de cada archivo dentro del grafo.

3.1. Manejo de archivos

La idea base consistía en investigar cómo crear una función de manejo de archivos para identificar los archivos regulares de texto plano en el directorio de entrada y sus sub-directorios para posteriormente procesarlos de manera eficiente, evitando la lectura de archivos que no sean de interés.

Se planteó usar una LES (lista enlazada simple) para almacenar los archivos, la cual lleve información de todos los archivos que se encuentran en el directorio de entrada, y a la que se haga referencia (mediante punteros) en las demás estructuras de datos. Esto permite una búsqueda más eficiente y sin un límite de memoria.

3.2. Grafos

Como requisito para el funcionamiento del programa es necesario crear un grafo para representar las relaciones entre los archivos y sus contenidos. Se plantearon tres estrategias para crear el grafo, de las que se analizan sus ventajas y desventajas a continuación:

- **Grafos con matriz de adyacencias**: Esta estrategia consiste en crear una matriz de tamaño n (donde n es la cantidad de archivos) para almacenar en ella la información de las relaciones entre los archivos. Este tipo de acercamiento favorece la eficiencia en tiempos de lectura y escritura sin embargo presenta dos grandes dificultades:
  1. **Costo de memoria**: El uso de una matriz de tamaño n para almacenar la información de las relaciones entre los archivos implica un costo de memoria considerable guardando espacio en memoria para cada enlace **aunque no exista**.
  2. **Memoria adyacente**: Este tipo de sistema requiere un espacio de $n^2 \times$ (tamaño de cada enlace) para almacenar la información de las relaciones entre los archivos, lo cual puede suponer una perdida de control para una gran cantidad de elementos.
- **Grafos con listas de adyacencias**: En esta implementación se plantea el uso de un arreglo o LES de *nodos*, donde cada uno de ellos lleva en sí una LES con sus adyacencias y otra con sus incidencias (nodos que apuntan a esta). Este tipo de estructura de datos permite un manejo de memoria muy preciso, sin embargo en caso de usar arreglos requiere de un espacio de memoria alto, y en caso de usar LES sacrifica la eficiencia en tiempos de lectura y escritura.
- **Grafos con Tablas Hash** Finalmente aparece la opción de usar una Tabla Hash para almacenar la información del grafo. Este tipo de estructura de datos usa una función de *hashing*[2] a cada nodo, permitiendo una búsqueda mucho más eficiente (usando como objeto para el hashing el nombre de cada archivo). En los casos que hubieran colisiones dentro de la tabla hash estas se manejan mediante LES que permiten una navegación sencilla entre los nodos con el mismo valor de hash.

[2] Función que asigna un valor a un objeto de forma que sea única en un conjunto de objetos, es posible que a dos objetos se les asigne el mismo valor a lo que se le conoce como una *Colisión*, este caso debe ser manejado de alguna manera en el uso de tablas hash.

3.3. Indice invertido

El índice invertido es una estructura de datos que almacena los archivos que contienen datos específicos, en lugar de listar los contenidos de cada archivo.

Este enfoque es útil para trabajar con grandes volúmenes de información, ya que facilita la búsqueda y la relación entre los archivos y su contenido.

Para el desarrollo de este proyecto se plantea el uso de un índice invertido para almacenar las palabras relevantes dentro del directorio de entrada. Para cada una de estas palabras se busca almacenar una lista con los archivos que la contienen, facilitando una búsqueda eficiente y basada en palabras claves.

Para implementar esto, surgieron diversas estructuras de datos factibles a utilizar, sin embargo, por las razones mencionadas anteriormente se decidió usar de igual forma una *Tabla Hash* variando las estructuras usadas como se detalla más adelante.

3.4. PageRank

Es estrictamente necesaria para este proyecto la utilización de un algoritmo que permitiera "rankear" por importancia cada uno de los nodos del grafo de archivos; lo anterior con el fin de mostrar los archivos relevantes para el usuario a la hora de hacer una búsqueda. Para ello se optó por utilizar el algoritmo de *PageRank*.

PageRank es un algoritmo creado por Brin y Page [1] que calcula la importancia de un archivo dentro de un conjunto de archivos (planteado inicialmente para el uso sobre páginas web).

"*Se considera que un archivo tiene gran prioridad si es enlazado por archivos de gran prioridad*". Aunque esta primicia suene redundante, esta es la base del algoritmo, ya que esta "importancia" está relacionada con la cantidad de enlaces adyacentes e incidentes que tiene cada archivo.

La fórmula para calcular la importancia esta dada por:

$$PR(i) = \frac{1-d}{n} + d\left(\sum_{i \in B_i} \frac{PR(j)}{L(j)} + \frac{S}{N}\right) \tag{1}$$

Donde:

- $PR(i)$: Page rank de la página i.
- d: Damping Factor o factor de amortiguación.
- N: Corresponde al numero total de elementos, en este caso archivos.
- B_i : Cantidad de enlaces adyacentes del archivo i.
- $PR(j)$: Pagerank de los archivos j que enlazan al archivo i
- $L(j)$: Número de enlaces adyacentes del archivo j.
- S: Corresponde a la suma de los PageRank de los nodos sin enlaces adyacentes.

Esta formula fue extraída del articulo original publicado por los autores de dicho algoritmo [1], sin embargo, en el presente proyecto fue ligeramente modificada con el fin de mantener consistencia al manejar los archivos que no referencian a ningún otro, es decir **sin enlaces adyacentes**. Se decidió optar por esta solución debido al poco coste de implementación que representa.

Para explicar la interpretación del algoritmo, se utilizará la analogía presente en el siguiente articulo escrito por Pedroche [4], la cuál imagina a un surfista, el cuál se encuentra navegando entre las diferentes páginas (en este caso archivos) de manera aleatoria. Lo que se busca con esto es construir una matriz con las probabilidades de escoger cada salto entre los archivos, según los enlaces presentes en estos. Dicho articulo profundiza en el trasfondo matemático del algoritmo, sin embargo, como este no es el objetivo del presente informe, únicamente se dejará la interpretación

del **PageRank** de un archivo como la probabilidad que existe de que aquel surfista se encuentre en dicho archivo en un momento dado.

Una representación visual del funcionamiento del algoritmo se puede observar en la figura 1, la cual muestra un grafo de archivos con sus respectivos enlaces. Esta figura y ejemplo fue recuperada y adaptada del video creado por Cabezón [6].

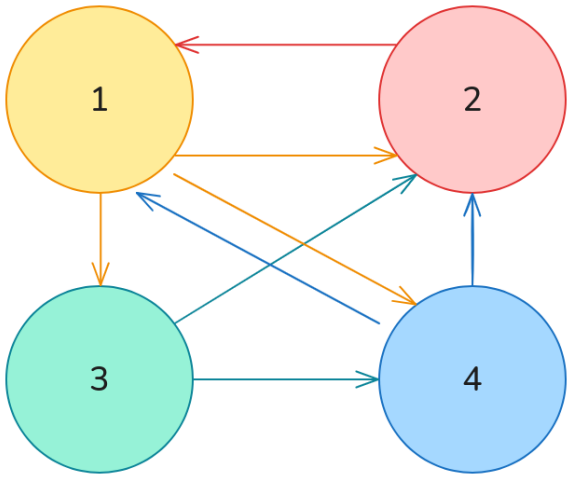

Figura 1. Representación visual del comportamiento de PageRank

Si se toma cada uno de los nodos con un valor de relevancia inicial 1, este valor se reparte entre los nodos adyacentes; al resolver el sistema de ecuaciones lineales que resulta de esto, se puede observar que el nodo 1 cuenta con $\frac{48}{31}$ de relevancia, el nodo 2 con $\frac{16}{31}$, el nodo 3 con $\frac{36}{31}$ y el nodo 4 con $\frac{24}{31}$, como se puede observar, si se suman todas las relevancias, se cuenta con el total de relevancias iniciales que seria 4, mostrándose asi un breve ejemplo de como se calcula el **PageRank** de cada nodo.

4. Implementación

La implementación del programa se realizó mediante el lenguaje de programación C. Las funciones necesarias fueron implementadas en archivos separados según su funcionalidad. El código puede ser compilado directamente mediante un `Makefile`, el cual se encuentra en el directorio raíz.

4.1. Estructura de directorios

Dentro del directorio `src` se encuentran los siguientes archivos:

- `errors.c`: Contiene las funciones de gestión de errores.
- `files.c`: Contiene las funciones de gestión de archivos.
- `graph.c`: Contiene las funciones de gestión del grafo.
- `hash.c`: Contiene las funciones de hashing.
- `link_list.c`: Contiene las funciones de gestión de listas enlazadas.
- `main.c`: Contiene el código principal del programa.
- `page_rank.c`: Contiene lo necesario para el cálculo del PageRank.
- `reverse_index.c`: Contiene las funciones de gestión del índice invertido.
- `stop_words.c`: Contiene las funciones de gestión de stop words.
- `timer.c`: Contiene las funciones de gestión del temporizador.
- `utilities.c`: Contiene funciones generales.

Por otro lado, en la carpeta `testing` se encuentran archivos para realizar pruebas del programa:

- `spanish.txt`: Contiene una lista de stopWords[3] en español extraídas de la referencia [5]; estas se tienen en cuenta a la hora de procesar los archivos así como a la hora de buscar palabras clave.
- `doc_generator.py`: Script de *Python* que recupera de *Wikipedia* artículos aleatorios con el fin de capturar archivos de prueba "realistas" para el programa.

4.2. Estructuras de datos

4.2.1. Manejo de archivos

Para el manejo de archivos se optó por utilizar una *lista enlazada simple* para almacenar los archivos, en la cual cada uno de ellos se guarda en un bloque de la lista. Esto permite una búsqueda más eficiente y sin un límite de memoria. Además, para poder identificar y procesar los archivos del directorio y sus sub-directorios se utilizó la librería `Dirent.h` [3].

La librería `dirent.h` es una librería de C que permite leer los archivos de un directorio, y que proporciona una estructura llamada *struct dirent* que contiene información sobre cada archivo, como por ejemplo su nombre (con extensión), su tipo (archivo regular o directorio) y su identificador (ID). También se utiliza la estructura *DIR* para manejar el directorio abierto.

En el caso de existir un sub-directorio dentro del directorio de entrada, se procesará recursivamente, añadiendo cada archivo que se encuentre en dicho sub-directorio a la lista de archivos.

Por último para identificar los archivos que contienen texto plano, se utilizó la función `is_valid_extension()` para extraer el nombre sin extensión y a su vez verificar si el archivo es de texto plano a traves de su extensión.

[3] Una StopWord es un tipo de palabra muy común en un idioma, en este caso en Español. Estas no deberían de ser consideradas para su almacenamiento en el indice invertido dada su recurrencia y la poca información entregada por su aparición en un documento.

4.2.2. Índice invertido

Para el índice invertido se optó finalmente por utilizar una *tabla hash* para almacenar las palabras, permitiendo así una búsqueda eficiente. Sin embargo, surge la necesidad de manejar posibles colisiones y el límite de memoria que trae consigo los arreglos. Para solucionar esto, se optó que cada celda de esta tabla hash contenga una *lista enlazada simple*, donde realmente se almacenarán las palabras asociadas a un hash key. Esto permite una búsqueda más eficiente y sin un límite de memoria (más que el de la propia máquina).

Por otro lado, para poder relacionar un archivo con una palabra se hizo que cada una de ellas almacene consigo otra *lista enlazada simple*, en la cual se almacenarán punteros a los archivos que contienen dicha palabra, además de guardar información sobre los párrafos en los que aparece dicha palabra (manejados con otra LES adicional).

Para comprender mejor la estructura de datos de este índice invertido, se presentará la figura 2.

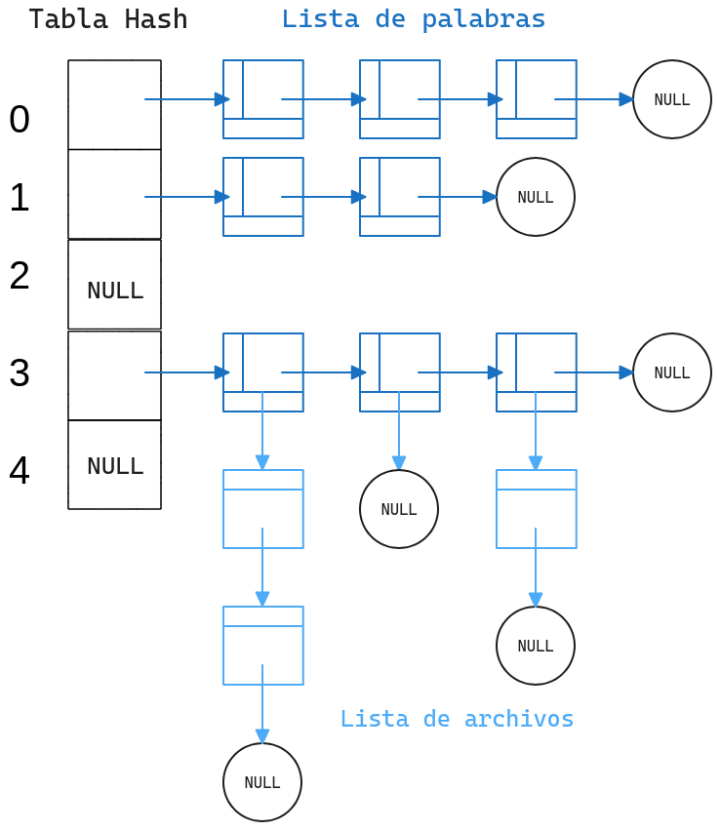

Figura 2. Esquema de la estructura de datos del índice invertido

Debido a que el uso de *listas enlazadas simples* esta ligado a una busqueda secuencial, como una manera de mitigar la ineficiencia de esto, se implemento la funcion `move_to_front()`, la cual mueve un nodo al principio de la lista. Entonces, cuando un nodo es buscado, éste queda al principio de la lista, así las palabras más buscadas por el usuario pueden volver a buscarse en menos tiempo.

4.2.3. Grafos

La implementación del grafo de archivos es muy similar a la del índice invertido (un Esquema similar al observado en la figura 2):

- Un arreglo de tamaño fijo conteniendo las keys de la tabla hash.
- Una lista para cada elemento del arreglo, conteniendo los nodos asociados a dicha key.
- Dentro de cada nodo se almacena información del archivo al que hace referencia (Puntero a la estructura de datos para el almacenamiento de

los archivos) así como una lista de enlaces adyacentes e incidentes.

El uso de Tablas Hash para esta solución permite una gran eficiencia de lectura-escritura, dando también espacio a un correcto manejo de memoria dinámica.

Cabe destacar que el grafo implementado es un *Grafo Dirigido* lo cual quiere decir que un enlace del nodo n_1 al n_2 es diferente al enlace desde n_2 a n_1 (lo que tiene sentido si se piensa en el comportamiento de los enlaces en un sistema de archivos).

4.3. PageRank

En el caso del *PageRank*, su implementación se realizo mediante la adaptación de la formula de PageRank (1) a un grafo dirigido (como se explicó en la sección 4.2.3). Para la construcción de la función `calculate_page_rank()` se utilizó la estructura de datos *Graph*, de la cual se obtuvieron los nodos y las listas de adyacencias e incidencia de estos.

Con el fin de ser consecuentes con lo planteado en 3.4, se tomó la cantidad total de nodos como el total de archivos en el directorio, y se asigno un valor inicial de $\frac{1}{N}$ a cada uno de los nodos, donde N es la cantidad total de nodos. Luego, se realizan las iteraciones necesarias para la convergencia del algoritmo, en el cual se actualiza el valor de cada nodo con la formula de PageRank (1), y se calcula el error de la iteración. Si el error es menor a un valor de frontera, se detiene el algoritmo y se retorna el valor de PageRank de cada nodo.

Es importante resaltar que el valor de frontera utilizado corresponde a 1×10^{-4}, el cual fue seleccionado de manera arbitraria. Y el valor de damping factor utilizado fue de 0.85, el cual es el valor estándar a la hora de implementar el PageRank.

4.4. implementaciones extra

Temporizador

Se implementó un temporizador utilizando la librería `time.h` para medir los tiempos de ejecución de las funciones, para analizar y mejorar el código (sólo para uso interno).

MergeSort

Se utilizó *MergeSort* para ordenar (de mayor a menor) los archivos en los que aparece cada palabra según el PageRank de los nodos que referencian, para tener una correcta visualización de los resultados.

Se decidió optar por este algoritmo de ordenamiento, dada su eficiencia sobre otros algoritmos tales como el `InsertionSort` o el `SelectionSort`.

Interacción con el usuario

En cuanto a la interacción con el usuario, se llevó a cabo una impresión por pantalla del procesamiento de los diversos archivos, para luego solicitar al usuario la palabra que desea buscar. Una vez ingresada una palabra se muestran los archivos con coincidencias y se le solicita al usuario que seleccione un archivo para mostrar el contenido del mismo. Para esta última tarea se implemento la función `print_file_paragraphs()` la cual imprime por pantalla los párrafos del archivo seleccionado en donde se encuentra la palabra.

Para la realización de esto se obtuvo el byte de inicio de cada uno de los párrafos en los que se encuentra cada palabra al ser ingresada a la tabla de indice invertido 4.2.2, mediante la función `process_files()`.

5. Gestión del equipo de trabajo

El equipo de desarrollo de este proyecto fué conformado por cuatro integrantes; este proyecto por el tiempo de implementación requería de una excelente coordinación para lograr un buen resultado.

Para mejorar el orden y la eficiencia se implementaron *normas de codificación*, así como el establecimiento de fechas para tareas pendientes, y por consecuencia reuniones para acordar los siguientes objetivos.

5.1. Normas de codificación

Con respecto al nombramiento de las constantes, estas fueron llamadas con el formato `SCREAMING_SNAKE_CASE`, y las variables se llamaron respectivamente con el formato `camelCase`. También a las funciones se acordó utilizar el formato `snake_case`, y colocar las llaves de apertura en la siguiente linea, por otro lado aquellas llaves de apertura de otros bloques de codigo se colocarán justo al lado de su linea final, evitando omitir llaves en caso de existir únicamente una sentencia.

ejemplo de referencia (anterior proyecto bajo las mismas normas de codificación): Informe acerca del proyecto ForKing.

5.2. Lista de tareas y organización

Teniendo ya realizada las normas de codificación, se establecieron objetivos principales (Puntos importantes del programa) a largo plazo, y objetivos secundarios para resolver a corto plazo.

Estos fueron escritos para mantener un control sobre el flujo de trabajo y repartirlos equitativamente pensando en los puntos fuertes de cada integrante del equipo.

6. Posibles Mejoras a futuro

Aun siendo *Fly* un programa eficiente como se esperaba, como equipo de trabajo somos conscientes acerca de aquellos aspectos que se podrían mejorar, que por varias razones no fueron implementadas en este programa ya sea por complejidad, falta de tiempo o poco conocimiento de algunas herramientas o librerías.

A continuación se listan parte de estas posibles mejoras:

1. **Uso de hebras**: El uso de hebras para manejar el procesamiento de los diferentes archivos sería un gran acierto para el programa, ya que este puede agilizar mucho los tiempos de espera con respecto a un programa secuencial como lo es *Fly*. Como mencionado previamente esta mejora no se implemento debido a la falta de tiempo y complejidad (se requería el uso de semáforos sobre las múltiples estructuras de datos utilizadas).
2. **Uso de estructuras de datos más eficientes**: En el presente trabajo se hizo un gran uso de las LES, principalmente por su sencillez de implementación y las ventajas que proveen, sin embargo la búsqueda sobre estas podría ser más eficiente si se usara, por ejemplo una lista circular doblemente enlazada que reduciría a la mitad los tiempos de búsqueda (aunque ciertamente son ya muy veloces).
3. **Experiencia de usuario**: La experiencia en *Fly* es restringida a buscar y mostrar coincidencias de una palabra en particular, dejando de lado tres posibles casos de uso relevantes:

    a) **Buscar varias palabras**: Sería una buena mejora poder buscar varias palabras a la vez, aunque no estén en orden, permitiendo un mejor filtrado de los archivos que se muestran.

    b) **Buscar frases particulares**: Un uso común de este tipo de motores de búsqueda es buscar frases particulares (no solamente palabras) lo que representaría una mejora considerable en cuanto a experiencia del usuario se refiere.

    c) **Búsqueda parcial de palabras**: Otros motores de búsqueda permiten identificar palabras parcialmente, así aparecerían como coincidencias archivos que contienen la palabra “univer” como ‘universidad’ o ‘universo’, esto sería muy útil para diferentes necesidades de los usuarios.

7. Ejemplo de uso

Dentro de esta sección se busca explicar parte del cómo sería una manera correcta de utilizar *Fly* para buscar y mostrar coincidencias de una palabra en particular.

7.1. Preparación de los archivos

A la hora de utilizar *Fly* se debe tener en cuenta que el programa lee solamente archivos de texto plano; particularmente se recomienda el uso de archivos tipo `Markdown`(.md) para los que el programa está especialmente diseñado.

Como fue mencionado anteriormente Fly reconoce enlaces mediante el uso de WikiLinks, a continuación se muestran algunos ejemplos de enlaces válidos:

- `[[nombre archivo enlazado]]`
- `[[nombre archivo enlazado.extension]]`
- `[[/ruta/al/archivo/enlazado/nombre archivo enlazado.extension]]`
- `[[nombre archivo enlazado#sección del archivo]]` (esto es usado en aplicaciones como Obsidian.md para generar un link a una sección del archivo).
- `[[nombre archivo enlazado|Alias para el archivo]]` (esto es usado en aplicaciones como Obsidian para generar un link y que aparezca un texto diferente al nombre del archivo).

Por lo general combinaciones de este tipo de links (alias con secciones, rutas, extensiones, etc) son válidas dentro de *Fly*.

7.2. Uso del programa

Una vez con los archivos correctamente formateados se puede ejecutar *Fly* desde la carpeta raíz del repositorio del proyecto con el comando `./build/fly.out -d <directorio a procesar>` lo cuál procesará el directorio indicado y permitirá ingresar palabras para su búsqueda dentro de los archivos procesados.

7.3. Prueba de estrés del programa

Fly fue probado con un dataset de 856 archivos construidos sobre Obsidian. Con este dataset se obtuvo un tiempo de procesamiento promedio menor a $1s$ para procesar todos los archivos y permitir la interacción del usuario. Posterior a esto los tiempos de respuesta de Fly son realmente cortos, lo que demuestra la eficiencia del programa.

Adicionalmente se ejecutó el programa bajo la herramienta `Valgrind` diseñada para encontrar fugas de memoria (herramienta muy útil para la depuración de errores durante el desarrollo de este programa), y se obtuvo un resultado de 0 fugas de memoria y un total de $\approx 24Mb$ de memoria utilizada durante su ejecución, lo que indica el correcto funcionamiento del programa.

8. Conclusiones

La creación de *Fly* fue un proyecto enriquecedor para todos aquellos quienes trabajamos en su implementación, ya que puso a prueba las distintas facetas que, como programadores, debemos de desarrollar a lo largo de un proyecto. Partiendo de la documentación previa al proyecto, pasando por la elección de las herramientas correctas para el proyecto, la implementación de los algoritmos y la optimización del código.

Se logró una correcta implementación de estructuras de datos de mayor complejidad a trabajos anteriores como puede ser principalmente la abstracción del grafo mediante tablas hash.

Adicionalmente se destaca la importancia y resultados de una correcta investigación de los algoritmos a utilizar, ya que se logró una buena comprensión de su funcionamiento así como una correcta y eficiente implementación de los mismos.

Por esta misma rama se resalta la eficiencia del programa en general, el cuál tiene tiempos de espera realmente cortos aún para grandes volúmenes de datos, lo que refleja la eficiencia de la implementación escogida.

Se puede concluir entonces, bajo los resultados obtenidos, que *Fly* es un proyecto que se ha desarrollado de manera exitosa teniendo en cuenta los objetivos propuestos.

Referencias

[1] S. Brin y L. Page. «The Anatomy of a Large-Scale Hypertextual Web Search Engine». (1998), dirección: http://infolab.stanford.edu/~backrub/google.html.

[2] Pasky. «Funciones opendir, readdir y closedir en C». (2009), dirección: https://pasky.wordpress.com/2009/08/05/funciones-opendir-readdir-y-closedir-en-c/ (visitado 03-11-2024).

[3] R. Pasniuk. «Directorios y Ficheros en C – Linux». (2013), dirección: https://www.programacion.com.py/escritorio/c/directorios-y-ficheros-en-c-linux (visitado 01-11-2024).

[4] F. Pedroche. «Una introducción al algoritmo PageRank». (2016), dirección: https://www.bellera.cat/joomla/images/1617/batx/google/fpedrochev4sema.pdf.

[5] M. M. C. «stop-words». (2022), dirección: https://github.com/Alir3z4/stop-words/blob/master/spanish.txt.

[6] E. S. de Cabezón, *PAGE RANK | El algoritmo matemático que hizo a GOOGLE dominar el mundo*, 2024. dirección: https://www.youtube.com/watch?v=b3fwA3EWCd8&t=387s.

[7] S. Tutorial. «Indexado en Solr». (2024), dirección: https://solrtutorial.es/indexado-solr.html.