UMAG Universidad de Magallanes | Facultad de Ingeniería

Sistema de Recuperación de Información (S.R.I.)

Francisco Mercado, Diego Galindo, Benjamín Sanhueza y Duvan Figueroa

1. Introducción

En el marco del estudio de estructuras de datos, este proyecto tiene como objetivo principal desarrollar un programa que simule el funcionamiento de un buscador web, haciendo énfasis en la implementación de grafos y el algoritmo PageRank. Los buscadores web constituyen una de las aplicaciones más relevantes y complejas en la informática moderna, ya que requieren la combinación de algoritmos avanzados y estructuras de datos para gestionar grandes volúmenes de información de manera eficiente.

El **algoritmo PageRank**, introducido por Larry Page y Sergey Brin, fundadores de Google, es uno de los pilares fundamentales en la clasificación de páginas web. Este algoritmo modela la relación entre las páginas como un grafo dirigido, en el que los nodos representan páginas web y los bordes simbolizan los enlaces entre ellas. Según los autores, *"PageRank calcula la importancia de las páginas mediante un modelo probabilístico basado en enlaces entrantes"* (Brin & Page, 1998).

Además, este proyecto no solo se limita al estudio de PageRank, sino que explora otras estructuras de datos fundamentales para el análisis y manipulación de información en grafos, tales como listas de adyacencia, matrices de adyacencia y el manejo de datos mediante tablas hash. Estas herramientas permiten optimizar la representación y el procesamiento de grafos, facilitando el manejo de relaciones complejas entre nodos. Como indican Cormen et al. (2009), *"las estructuras de datos son esenciales para diseñar algoritmos eficientes, pues determinan la forma en que se organizan y procesan los datos"*.

El objetivo educativo de este trabajo es doble. Por un lado, se busca proporcionar una comprensión teórica y práctica de cómo las estructuras de datos y los algoritmos se integran en aplicaciones reales. Por otro, se promueve el aprendizaje de habilidades técnicas esenciales, como la modularización del código, el diseño de algoritmos eficientes y el uso adecuado de bibliotecas externas para la gestión de datos. Según Weiss (2013), *"la modularización y el análisis de algoritmos son herramientas clave para garantizar que el software no solo funcione, sino que sea escalable y mantenible"*.

En este informe, se abordarán los conceptos fundamentales de las estructuras de datos empleadas, la implementación detallada del código y las posibles mejoras que se podrían realizar para optimizar el sistema. Todo ello tiene como finalidad no solo satisfacer los requerimientos técnicos del proyecto, sino también fomentar la capacidad de resolución de problemas a través de la informática, una habilidad esencial en el desarrollo profesional.

2. Planificación

El desarrollo del proyecto "Sistema de Recuperación de Información (S.R.I)" se basó en una planificación colaborativa y detallada, que permitió a los integrantes del equipo trabajar de manera eficiente y cumplir con los objetivos establecidos. Desde el inicio, el equipo implementó un plan que dividió las tareas en roles específicos, asegurando un enfoque organizado y cooperativo.

Integrantes

- **Equipo de desarrollo**:
  - Benjamin Sanhueza
  - Duvan Figueroa
  - Francisco Mercado
  - Diego Galindo

Plan de Trabajo Inicial

El equipo analizó las instrucciones del proyecto y las desglosó en categorías funcionales, asignando roles específicos a cada integrante. Este enfoque permitió que cada miembro se enfocara en áreas concretas del programa. Cabe destacar que los roles iniciales estaban sujetos a cambios según las necesidades del proyecto o el avance del desarrollo.

Roles

1. **Duvan Figueroa**:
   - Responsable de implementar el índice invertido, una estructura clave para la búsqueda eficiente en el sistema.
   - Desarrollo del manejo de stopwords, eliminando palabras irrelevantes para optimizar los resultados de búsqueda.
   - Colaboración en el funcionamiento general del programa.
2. **Francisco Mercado**:
   - Encargado del diseño e implementación de los grafos, esenciales para representar las relaciones entre páginas.
   - Desarrollo e integración del algoritmo PageRank, utilizado para asignar importancia a las páginas web.
   - Supervisión del funcionamiento general del sistema.
   - Adicionalmente, creó el programa auxiliar `wordgenerator` para generar datos de prueba en formato **.txt,** simulando links de páginas web mediante palabras extraídas de **palabras.txt**
3. **Diego Galindo**:
   - Desarrolló el módulo de Merge, que optimiza la organización y manipulación de datos.
   - Se encargó de la optimización del código para mejorar el rendimiento del programa.
   - Colaboró en la modularización del proyecto, asegurando un diseño estructurado y escalable.
4. **Benjamin Sanhueza**:
   - Responsable de la gestión del ingreso de documentos al sistema, asegurando la correcta lectura y almacenamiento.
   - Implementación de las estructuras de datos necesarias y las listas que respaldan el funcionamiento del índice invertido.
   - Apoyo en la validación y pruebas del programa.

Plan de Trabajo Final

El equipo logró seguir el plan de trabajo inicial sin mayores inconvenientes. Los integrantes cumplieron con sus roles de manera comprometida y efectiva, trabajando en conjunto para superar cualquier dificultad. Surgieron tareas adicionales durante el desarrollo, como la corrección de errores de código y ajustes en la funcionalidad del programa, las cuales fueron resueltas rápidamente, en un plazo no mayor a un día desde su detección.

Herramientas de Prueba

Para probar el programa, Francisco Mercado desarrolló el programa auxiliar `wordgenerator`. Esta herramienta permitió generar archivos de texto con palabras aleatorias, simulando enlaces entre páginas web, lo que facilitó la validación del grafo y el algoritmo **PageRank.**

La planificación organizada y la colaboración entre los integrantes fueron fundamentales para el éxito del proyecto, garantizando un desarrollo eficiente y una implementación funcional del sistema de recuperación de información.

3. Implementación

Implementación: adj_list.h

El archivo `adj_list.h` define las estructuras básicas necesarias para implementar una lista de adyacencia, que es una representación eficiente de grafos. A continuación, se detalla el contenido y su función dentro del sistema:

Estructuras definidas

AdjListNode:

- Representa un nodo en la lista de adyacencia.
- Contiene dos campos:
  - `dest`: Identificador del nodo destino (documento o página web vinculada).
  - `next`: Puntero al siguiente nodo en la lista.
- Esta estructura permite representar las conexiones entre los nodos de un grafo, esencial para modelar los enlaces entre páginas.

```c
typedef struct AdjListNode {
    int dest;
    struct AdjListNode *next;
} AdjListNode;
```

Code 1. Definición de la estructura `AdjListNode`

AdjList:

- Representa una lista de adyacencia completa.
- Contiene:
  - `document_id`: Identificador del nodo fuente (documento o página web).
  - `head`: Puntero al primer nodo en la lista de adyacencia (enlaces desde este nodo a otros).
- Esta estructura encapsula las relaciones de un nodo del grafo con otros nodos.

```c
typedef struct AdjList {
    int document_id;
    AdjListNode *head;
} AdjList;
```

Code 2. Definición de la estructura `AdjList`

Propósito en el sistema

- Estas estructuras son fundamentales para la representación del grafo en el sistema. Cada nodo del grafo (página web o documento) se representa mediante un `AdjList`, mientras que sus conexiones a otras páginas se representan mediante una lista enlazada de `AdjListNode`.
- La lista de adyacencia es una de las estructuras más eficientes para grafos dispersos, ya que almacena solo las conexiones necesarias, minimizando el consumo de memoria.

Impacto en el programa

- Estas estructuras serán utilizadas para almacenar y gestionar las relaciones entre documentos generados o cargados al sistema.
- Son cruciales para el algoritmo PageRank, ya que permiten recorrer el grafo para calcular las métricas de importancia de cada nodo.

Implementación: `docs_management.h`

El archivo `docs_management.h` define las funciones necesarias para gestionar los documentos en el sistema. Este módulo es fundamental para la integración del manejo de datos en el sistema de recuperación de información. A continuación, se detalla su contenido y su función dentro del sistema:

Funciones definidas

`agregar_documento`:

- Función encargada de agregar un documento al sistema.
- Recibe dos parámetros:
  - `nodo_palabra`: Un puntero a un nodo de palabra, utilizado para gestionar el índice invertido.
  - `nombre_archivo`: Nombre del archivo que se está agregando.
- Su propósito es vincular las palabras clave de un documento con su contenido en el índice invertido.

```c
void agregar_documento(PalabraNodo *nodo_palabra, const char *nombre_archivo);
```

Code 3. Declaración de la función `agregar_documento`

`mostrar_vista_previa`:

- Función encargada de mostrar una vista previa de un archivo específico.
- Recibe dos parámetros:
  - `nombre_archivo`: Nombre del archivo a visualizar.
  - `palabra_original`: Palabra clave a buscar dentro del archivo.
- Permite verificar cómo aparecen las palabras clave dentro del contenido de un documento.

```
1 void mostrar_vista_previa(const char *nombre_archivo, const char *palabra_original);
```

Code 4. Declaración de la función `mostrar_vista_previa`

Propósito en el sistema

- Este archivo actúa como interfaz para las operaciones relacionadas con la gestión de documentos en el sistema.
- Vincula los datos procesados en el índice invertido con el contenido de los archivos físicos, permitiendo tanto la indexación como la recuperación eficiente de información.

Impacto en el programa

- Facilita la integración de documentos en el índice invertido, asegurando que las palabras clave de cada archivo estén correctamente asociadas.
- Mejora la experiencia del usuario al proporcionar una funcionalidad para previsualizar archivos y sus palabras clave, lo cual es crucial en sistemas de recuperación de información.

Implementación:`files.h`

El archivo `files.h` contiene las definiciones de funciones relacionadas con la gestión y manipulación de archivos en el sistema. Estas funciones son esenciales para interactuar con los datos almacenados en los documentos. A continuación, se presenta su contenido y su propósito en el programa:

Funciones definidas

`verificar_extension_archivo`:

- Función que verifica si un archivo tiene la extensión adecuada.
- Recibe un parámetro: `nombre_archivo`: Nombre del archivo cuya extensión se desea validar.
- Devuelve un valor entero indicando si la extensión es válida.

```
1 int verificar_extension_archivo(const char *nombre_archivo);
```

Code 5. Declaración de la función `verificar_extension_archivo`

`leer_archivo`:

- Función encargada de leer el contenido de un archivo.
- Recibe un parámetro: `archivo`: Nombre del archivo a leer.
- Devuelve un puntero a un `char` que contiene el contenido del archivo como una cadena.

```
1 char* leer_archivo(const char *archivo);
```

Code 6. Declaración de la función `leer_archivo`

Propósito en el sistema

`verificar_extension_archivo`: Garantiza que solo se procesen archivos con extensiones soportadas por el sistema, evitando errores o inconsistencias en el manejo de datos. `leer_archivo`: Proporciona un medio para cargar el contenido de un archivo en memoria, lo que es fundamental para realizar análisis de texto o indexación.

Impacto en el programa

- Estas funciones permiten una integración robusta con los documentos almacenados en el sistema.
- Mejoran la flexibilidad y seguridad al validar los archivos antes de procesarlos y facilitar su lectura, lo que es crucial para manejar datos textuales en sistemas de recuperación de información.

Implementación:`graph_links.h`

El archivo `graph_links.h` define funciones esenciales para gestionar y manipular las conexiones en los grafos del sistema. Estas funciones permiten la creación y el manejo de enlaces entre nodos, lo cual es crucial para modelar relaciones en el grafo. A continuación, se presenta su contenido y propósito:

Funciones definidas

`create_adj_list_node`:

- Crea un nuevo nodo para la lista de adyacencia.
- Recibe un parámetro: `dest`: Identificador del nodo destino al que apuntará el nuevo nodo.
- Devuelve un puntero a un nodo de lista de adyacencia recién creado.

```
1 AdjListNode* create_adj_list_node(int dest);
```

Code 7. Declaración de la función `create_adj_list_node`

`add_link`:

- Añade un enlace entre dos nodos en el grafo.
- Recibe tres parámetros:
  - `graph`: Puntero al grafo donde se añadirá el enlace.
  - `src`: Nodo fuente del enlace.
  - `dest`: Nodo destino del enlace.
- Inserta el enlace en la lista de adyacencia del nodo fuente.

```
1 void add_link(Graph *graph, int src, int dest);
```

Code 8. Declaración de la función `add_link`

`connect_graphs`:

- Conecta dos grafos distintos mediante un enlace.
- Recibe cuatro parámetros:
  - `graph1`: Puntero al primer grafo.
  - `graph2`: Puntero al segundo grafo.
  - `src`: Nodo del primer grafo que actuará como origen.
  - `dest`: Nodo del segundo grafo que será el destino.
- Permite relacionar dos grafos diferentes, útil para aplicaciones donde los datos se distribuyen en múltiples estructuras.

```
1 void connect_graphs(Graph *graph1, Graph *graph2, int src, int dest);
```

Code 9. Declaración de la función `connect_graphs`

Propósito en el sistema

`create_adj_list_node`: Facilita la creación de nodos para listas de adyacencia, la estructura base para representar los grafos en memoria. `add_link`: Permite modelar las relaciones entre nodos dentro de un único grafo. `connect_graphs`: Proporciona flexibilidad adicional al permitir que se combinen o relacionen grafos separados, ampliando las capacidades del sistema.

Impacto en el programa

- Estas funciones son fundamentales para gestionar las conexiones en el sistema de grafos, proporcionando las herramientas necesarias para construir y modificar la estructura del grafo.
- La capacidad de conectar grafos independientes amplía las posibilidades de análisis y modelado de datos en el sistema, haciéndolo más versátil y robusto.

Implementación: `graph_management.h`

El archivo `graph_management.h` define la estructura y las funciones necesarias para gestionar los grafos dentro del sistema. Estas funciones son fundamentales para inicializar, administrar y liberar la memoria asociada con los grafos. A continuación, se detalla su contenido y propósito:

Estructura definida

`Graph`:

- Representa el grafo en el sistema.
- Contiene dos campos:
  - `num_docs`: Número de nodos (documentos) en el grafo.
  - `array`: Puntero a un arreglo de listas de adyacencia que modelan las relaciones entre nodos.

```c
typedef struct Graph {
    int num_docs;
    AdjList *array;
} Graph;
```

Code 10. Definición de la estructura `Graph`

Funciones definidas

`create_graph`:

- Crea e inicializa un grafo con un número específico de nodos.
- Recibe un parámetro: `num_docs`: Número de documentos o nodos que tendrá el grafo.
- Devuelve un puntero a un grafo recién creado.

```c
Graph* create_graph(int num_docs);
```

Code 11. Declaración de la función `create_graph`

`free_graph`:

- Libera la memoria asociada con un grafo.
- Recibe un parámetro: `graph`: Puntero al grafo a liberar.
- Garantiza la eliminación segura de los recursos utilizados.

```c
void free_graph(Graph *graph);
```

Code 12. Declaración de la función `free_graph`

Propósito en el sistema

`Graph`: Actúa como la estructura principal para modelar los datos en forma de grafo. `create_graph`: Proporciona un método eficiente para inicializar un grafo en función del número de nodos requeridos, configurando la memoria necesaria. `free_graph`: Asegura una correcta gestión de recursos al liberar la memoria asignada a un grafo una vez que ya no se necesita.

Impacto en el programa

- Estas definiciones y funciones son fundamentales para trabajar con grafos en el sistema, permitiendo su creación, manipulación y eliminación de manera segura y eficiente.
- Proveen una base estructurada para construir relaciones complejas entre nodos y realizar análisis sobre estas conexiones.

Implementación: `hash_utils.h`

El archivo `hash_utils.h` define la función esencial para la generación de valores hash en el sistema. Este módulo es crucial para el manejo eficiente de datos en estructuras como tablas hash. A continuación, se detalla su contenido y propósito:

Funciones definidas

`hash`:

- Calcula el valor hash de una cadena de caracteres.
- Recibe un parámetro: `str`: Puntero a la cadena de caracteres para la cual se desea calcular el hash.
- Devuelve un valor de tipo `unsigned int` que representa el hash calculado.

```
1 unsigned int hash(const char *str);
```

Code 13. Declaración de la función `hash`

Propósito en el sistema

- La función `hash` es esencial para mapear cadenas de texto (por ejemplo, palabras clave o nombres de archivos) a índices en una tabla hash.
- Permite organizar y acceder a datos de manera eficiente, lo cual es fundamental para el funcionamiento del índice invertido y otras estructuras relacionadas.

Impacto en el programa

- Facilita la implementación de estructuras de datos como tablas hash, optimizando las operaciones de búsqueda y almacenamiento.
- Contribuye a la eficiencia general del sistema al proporcionar un método rápido para gestionar grandes volúmenes de datos textuales.

Implementación: `index_operations.h`

El archivo `index_operations.h` define las funciones necesarias para realizar operaciones clave en el índice invertido del sistema. Estas operaciones incluyen la adición de palabras al índice y la búsqueda de palabras específicas. A continuación, se describe su contenido y propósito:

Funciones definidas

`agregar_palabra`:

- Agrega una palabra al índice invertido asociándola con un archivo específico.
- Recibe tres parámetros:
  - `indice`: Puntero al índice invertido donde se almacenará la palabra.
  - `palabra_original`: La palabra que se desea agregar.
  - `nombre_archivo`: Nombre del archivo al que pertenece la palabra.
- Permite que las palabras sean indexadas de manera eficiente, asociándolas con los documentos correspondientes.

```
1 void agregar_palabra(IndiceInvertido *indice, const char *palabra_original, const char *nombre_archivo);
```

Code 14. Declaración de la función `agregar_palabra`

`buscar_palabra`:

- Realiza la búsqueda de una palabra en el índice invertido.
- Recibe dos parámetros:
  - `indice`: Puntero al índice invertido donde se realizará la búsqueda.
  - `palabra`: La palabra que se desea buscar.
- Permite recuperar información sobre en qué documentos aparece una palabra específica.

```
1 void buscar_palabra(IndiceInvertido *indice, const char *palabra);
```

Code 15. Declaración de la función `buscar_palabra`

Propósito en el sistema

`agregar_palabra`: Asegura que las palabras sean correctamente indexadas en el sistema, asociándolas con los archivos que las contienen. Esto es crucial para que el sistema pueda recuperar documentos relevantes durante una búsqueda. `buscar_palabra`: Proporciona una funcionalidad esencial para el sistema de recuperación de información, permitiendo consultar qué documentos contienen una palabra específica.

Impacto en el programa

- Estas funciones son el núcleo de las operaciones del índice invertido, garantizando la indexación eficiente de los datos textuales y la capacidad de búsqueda dentro del sistema.
- Mejoran significativamente la capacidad del programa para manejar grandes cantidades de datos textuales de manera organizada y accesible.

Implementación: `indice_invertido.h`

El archivo `indice_invertido.h` define las estructuras fundamentales para implementar un índice invertido, una de las principales herramientas para la recuperación eficiente de información en el sistema. A continuación, se describen las estructuras definidas y su propósito:

Estructuras definidas

`DocumentoNodo`:

- Representa un documento asociado con una palabra específica.
- Contiene tres campos:
  - `nombre_archivo`: Nombre del archivo donde aparece la palabra.
  - `frecuencia`: Número de veces que la palabra aparece en el documento.
  - `siguiente`: Puntero al siguiente nodo de documento, lo que permite manejar múltiples documentos.

```c
typedef struct DocumentoNodo {
    char *nombre_archivo;
    int frecuencia;
    struct DocumentoNodo *siguiente;
} DocumentoNodo;
```

Code 16. Definición de la estructura `DocumentoNodo`

`PalabraNodo`:

- Representa una palabra en el índice invertido.
- Contiene tres campos:
  - `palabra`: La palabra almacenada.
  - `documentos`: Puntero a una lista de nodos de documentos donde aparece esta palabra.
  - `siguiente`: Puntero al siguiente nodo de palabra.

```c
typedef struct PalabraNodo {
    char *palabra;
    DocumentoNodo *documentos;
    struct PalabraNodo *siguiente;
} PalabraNodo;
```

Code 17. Definición de la estructura `PalabraNodo`

`IndiceInvertido`:

- Representa el índice invertido completo.
- Contiene: `tabla`: Un arreglo de punteros a nodos de palabras, con un tamaño definido por la constante `TAMANO_TABLA`.

```c
typedef struct {
    PalabraNodo *tabla[TAMANO_TABLA];
} IndiceInvertido;
```

Code 18. Definición de la estructura `IndiceInvertido`

Propósito en el sistema

`DocumentoNodo`: Permite almacenar y acceder a información sobre los documentos en los que aparece cada palabra, junto con su frecuencia. `PalabraNodo`: Gestiona la información de cada palabra en el índice, incluyendo su lista de documentos asociados. `IndiceInvertido`: Actúa como la estructura principal que organiza y almacena las palabras y sus documentos relacionados, permitiendo búsquedas rápidas y eficientes.

Impacto en el programa

- Estas estructuras constituyen la base del índice invertido, permitiendo indexar palabras y sus documentos de manera eficiente.
- Proporcionan un modelo robusto para realizar operaciones como la búsqueda de palabras y la recuperación de documentos relevantes.

Implementación: `memory.h`

El archivo `memory.h` define una función clave para la gestión de la memoria en el sistema. Este módulo es esencial para garantizar que los recursos de memoria utilizados sean liberados correctamente, evitando fugas y mejorando la estabilidad del programa. A continuación, se detalla su contenido:

Funciones definidas

`liberar_memoria`:

- Libera los recursos de memoria asociados con una lista de nodos.
- Recibe un parámetro: `inicio`: Puntero al nodo inicial de la lista que será liberada.

- Garantiza que todos los nodos de la lista y sus datos asociados sean liberados de manera segura.

```c
void liberar_memoria(Nodo *inicio);
```

Code 19. Declaración de la función `liberar_memoria`

Propósito en el sistema

- La función `liberar_memoria` asegura una correcta gestión de recursos en el programa, liberando la memoria asignada dinámicamente a las estructuras de nodos cuando ya no son necesarias.
- Es especialmente útil para manejar listas enlazadas y otras estructuras dinámicas que podrían ocupar memoria innecesariamente si no se eliminan adecuadamente.

Impacto en el programa

- Mejora la estabilidad del sistema al prevenir fugas de memoria, lo que es crucial para aplicaciones que manejan grandes volúmenes de datos o que funcionan durante largos períodos.
- Facilita el manejo seguro de estructuras dinámicas, asegurando que todos los recursos sean liberados correctamente al finalizar su uso.

Implementación: `node_management.h`

El archivo `node_management.h` define las estructuras y funciones necesarias para gestionar los nodos en el sistema, que representan páginas o documentos en el grafo. Este módulo es crucial para modelar las conexiones entre nodos y calcular métricas como el PageRank. A continuación, se describe su contenido:

Estructuras definidas

`InLinkNode`:

- Representa un enlace entrante a un nodo.
- Contiene dos campos:
  - `node_id`: Identificador del nodo origen del enlace.
  - `next`: Puntero al siguiente nodo en la lista de enlaces entrantes.

```c
typedef struct InLinkNode {
    int node_id;
    struct InLinkNode *next;
} InLinkNode;
```

Code 20. Definición de la estructura `InLinkNode`

`Node`:

- Representa un nodo del grafo.
- Contiene los siguientes campos:
  - `pagerank`: Valor de PageRank asociado al nodo.
  - `out_links`: Número de enlaces salientes del nodo.
  - `keyword_relevance`: Relevancia de palabras clave asociada al nodo.
  - `in_links`: Puntero a una lista de nodos que representan enlaces entrantes.

```c
typedef struct Node {
    double pagerank;
    int out_links;
    double keyword_relevance;
    InLinkNode *in_links;
} Node;
```

Code 21. Definición de la estructura `Node`

Funciones definidas `create_node`:

- Crea e inicializa un nodo.
- Recibe un parámetro: `out_links`: Número de enlaces salientes iniciales del nodo.
- Devuelve un puntero al nodo recién creado.

```
Node* create_node(int out_links);
```

Code 22. Declaración de la función `create_node`

`add_inlink`:

- Añade un enlace entrante a un nodo.
- Recibe dos parámetros:
  - `node`: Puntero al nodo al que se añadirá el enlace.
  - `source_id`: Identificador del nodo origen del enlace.

```
void add_inlink(Node *node, int source_id);
```

Code 23. Declaración de la función `add_inlink`

`free_nodes`:

- Libera los recursos asociados con un arreglo de nodos.
- Recibe dos parámetros:
  - `nodes`: Arreglo de punteros a nodos.
  - `node_count`: Número de nodos en el arreglo.

```
void free_nodes(Node *nodes[], int node_count);
```

Code 24. Declaración de la función `free_nodes`

Propósito en el sistema

`InLinkNode` y `Node` proporcionan las estructuras necesarias para modelar nodos del grafo y sus conexiones. Las funciones permiten gestionar la creación, modificación y liberación de nodos, fundamentales para operaciones como la evaluación de PageRank y la manipulación del grafo.

Impacto en el programa

- Facilitan la representación y análisis de grafos mediante la gestión eficiente de nodos y sus conexiones.
- Mejoran la modularidad y la claridad del programa al encapsular las operaciones relacionadas con los nodos en funciones específicas.

Implementación: `nodes.h`

El archivo `nodes.h` define una estructura y una función principal para gestionar nodos relacionados con los archivos en el sistema. Esta estructura es útil para almacenar y manipular información textual extraída de los documentos. A continuación, se detalla su contenido:

Estructura definida

`Nodo`:

- Representa un nodo en una lista enlazada de archivos.
- Contiene tres campos:
  - `nombre_archivo`: Puntero a una cadena que almacena el nombre del archivo.
  - `contenido`: Puntero a una cadena que almacena el contenido del archivo.
  - `siguiente`: Puntero al siguiente nodo en la lista, permitiendo construir una lista enlazada.

```c
typedef struct Nodo {
    char *nombre_archivo;
    char *contenido;
    struct Nodo *siguiente;
} Nodo;
```

Code 25. Definición de la estructura `Nodo`

Funciones definidas

`agregar_nodo`:

- Añade un nuevo nodo a una lista enlazada.
- Recibe tres parámetros:
  - `inicio`: Puntero al nodo inicial de la lista.
  - `nombre_archivo`: Nombre del archivo que se añadirá al nodo.
  - `contenido`: Contenido asociado al archivo.
- Devuelve un puntero al nodo inicial actualizado con el nuevo nodo.

```c
Nodo* agregar_nodo(Nodo *inicio, const char *nombre_archivo, const char *contenido);
```

Code 26. Declaración de la función `agregar_nodo`

Propósito en el sistema

`Nodo`: Almacena información básica de los archivos, permitiendo construir una lista enlazada de documentos y su contenido. `agregar_nodo`: Facilita la construcción dinámica de una lista de archivos y su contenido, lo cual es útil para operaciones como la indexación y la búsqueda.

Impacto en el programa

- Estas definiciones son esenciales para manejar datos relacionados con archivos en el sistema, especialmente en la gestión de documentos que necesitan ser indexados o procesados.
- Proveen una estructura simple pero efectiva para representar y manipular información de archivos.

Implementación: `pagerank.h`

El archivo `pagerank.h` define las funciones principales para calcular el PageRank y establecer la relevancia de palabras clave asociadas a los nodos en el grafo. Estas funciones son fundamentales para evaluar la importancia relativa de cada nodo y mejorar la precisión de las búsquedas en el sistema. A continuación, se describe su contenido:

Funciones definidas

`calculate_pagerank`:

- Calcula los valores de PageRank para un conjunto de nodos.
- Recibe los siguientes parámetros:
  - `nodes`: Arreglo de punteros a los nodos del grafo.
  - `node_count`: Número total de nodos.
  - `damping_factor`: Factor de amortiguación utilizado en el algoritmo de PageRank.
  - `iterations`: Número de iteraciones para calcular el PageRank.
- Actualiza los valores de PageRank para cada nodo en el grafo.

```c
void calculate_pagerank(Node *nodes[], int node_count, double damping_factor, int iterations);
```

Code 27. Declaración de la función `calculate_pagerank`

`set_keyword_relevance`:

- Establece la relevancia de palabras clave para los nodos del grafo.
- Recibe los siguientes parámetros:
  - `nodes`: Arreglo de punteros a los nodos del grafo.
  - `node_count`: Número total de nodos.
  - `keywords`: Arreglo de cadenas que representan las palabras clave.
  - `keyword_count`: Número de palabras clave en el arreglo.
  - `documents`: Arreglo de documentos en los que buscar las palabras clave.
- Actualiza la relevancia de palabras clave para cada nodo en función de su contenido.

```
void set_keyword_relevance(Node *nodes[], int node_count, const char *keywords[], int keyword_count,
    const char *documents[]);
```

Code 28. Declaración de la función `set_keyword_relevance`

Propósito en el sistema

`calculate_pagerank`: Implementa el algoritmo de PageRank para asignar una puntuación de importancia a cada nodo en el grafo, lo que es esencial para determinar la prioridad de los resultados en un sistema de búsqueda. `set_keyword_relevance`: Evalúa qué tan relevantes son los nodos en relación con ciertas palabras clave, mejorando la precisión del sistema al responder a consultas específicas.

Impacto en el programa

- Estas funciones combinan análisis estructural (PageRank) y semántico (relevancia de palabras clave) para optimizar la recuperación de información.
- Mejoran la capacidad del sistema para priorizar resultados relevantes y útiles para el usuario.

Implementación: `stopwords.h`

El archivo `stopwords.h` define las funciones necesarias para gestionar y utilizar las palabras irrelevantes, comúnmente conocidas como *stopwords*, en el sistema de recuperación de información. Estas funciones ayudan a filtrar términos no significativos durante la indexación y búsqueda. A continuación, se detalla su contenido:

Macros definidas

`MAX_STOPWORDS`: Define el número máximo de palabras que pueden ser cargadas como *stopwords*. Valor: `1000`. `MAX_WORD_LENGTH`: Define la longitud máxima de una palabra en el conjunto de *stopwords*. Valor: `50`.

```
#define MAX_STOPWORDS 1000
#define MAX_WORD_LENGTH 50
```

Code 29. Definición de macros en `stopwords.h`

Funciones definidas

`cargar_stopwords`:

- Carga una lista de *stopwords* desde un archivo.
- Recibe los siguientes parámetros:
  - `nombre_archivo`: Nombre del archivo que contiene las *stopwords*.
  - `stopwords`: Arreglo donde se almacenarán las *stopwords* cargadas.
- Devuelve el número total de *stopwords* cargadas.

```
int cargar_stopwords(const char *nombre_archivo, char stopwords[][MAX_WORD_LENGTH]);
```

Code 30. Declaración de la función `cargar_stopwords`

`es_stopword`:

- Verifica si una palabra dada es una *stopword*.
- Recibe los siguientes parámetros:
  - `palabra`: Palabra que será verificada.
  - `stopwords`: Arreglo que contiene las *stopwords* cargadas.
  - `num_stopwords`: Número total de *stopwords* cargadas.
- Devuelve un valor entero (`1` si es una *stopword*, `0` en caso contrario).

```
int es_stopword(const char *palabra, char stopwords[][MAX_WORD_LENGTH], int num_stopwords);
```

Code 31. Declaración de la función `es_stopword`

Propósito en el sistema

`cargar_stopwords`: Proporciona un método para inicializar un conjunto de *stopwords* desde un archivo externo, facilitando su personalización. `es_stopword`: Permite identificar términos irrelevantes durante las operaciones de indexación y búsqueda, mejorando la precisión del sistema.

Impacto en el programa

- Estas funciones optimizan el procesamiento del texto eliminando términos irrelevantes, reduciendo así el ruido en los resultados de búsqueda.
- Mejoran la calidad y relevancia del sistema al enfocarse en palabras clave significativas.

Implementación: `main.c`

El archivo `main.c` contiene la implementación principal del sistema, integrando todas las funciones y estructuras definidas en los archivos de encabezado y módulos auxiliares. Este archivo representa el punto de entrada del programa y coordina la ejecución de las distintas funcionalidades del sistema. A continuación, se detalla su contenido y propósito.

Resumen de funcionalidad

El programa realiza las siguientes tareas principales:

1. **Inicialización**:
   - Carga las *stopwords* desde un archivo y las almacena para filtrar términos irrelevantes.
   - Inicializa el índice invertido y los grafos que modelan las relaciones entre documentos.
2. **Procesamiento de documentos**:
   - Lee y procesa archivos de texto desde un directorio específico.
   - Tokeniza el contenido, lo normaliza (convertir a minúsculas y limpiar palabras) y lo indexa en el índice invertido.
3. **Construcción de grafos**:
   - Crea dos grafos (representando dos "webs").
   - Establece enlaces ficticios entre documentos en cada grafo y conexiones entre ambos grafos.
4. **Cálculo del PageRank**:
   - Crea nodos asociados con los documentos y calcula los valores de PageRank para cada nodo.
   - Imprime los resultados del PageRank por grafo.
5. **Búsqueda de palabras clave**:
   - Permite buscar palabras clave ingresadas por el usuario, excluyendo *stopwords*.
   - Imprime los documentos relevantes asociados con las palabras clave.
6. **Liberación de memoria**: Libera los recursos asignados para grafos, nodos e índices invertidos al finalizar la ejecución.

Fragmentos destacados

Carga de *stopwords*: Se utiliza `cargar_stopwords` para leer las palabras irrelevantes desde un archivo y almacenarlas en memoria.

```
char stopwords[MAX_STOPWORDS][MAX_WORD_LENGTH];
int num_stopwords = cargar_stopwords("docs/stopwords.txt", stopwords);
if (num_stopwords == 0) {
    printf("No se cargaron los stopwords.\n");
    return 1;
}
```

Code 32. Carga de `stopwords`

Procesamiento de documentos: Lee documentos desde la carpeta `docs/`, tokeniza su contenido y lo indexa.

```
char carpeta[] = "./docs/";
DIR *dir = opendir(carpeta);
if (dir == NULL) {
    perror("No se pudo abrir el directorio.");
    return 1;
}
struct dirent *entrada;
while ((entrada = readdir(dir)) != NULL) {
    if (!verificar_extension_archivo(entrada->d_name)) continue;
    char ruta_completa[512];
    snprintf(ruta_completa, sizeof(ruta_completa), "%s%s", carpeta, entrada->d_name);
    char *contenido = leer_archivo(ruta_completa);
    if (contenido == NULL) continue;
    char *token = strtok(contenido, " \t\n\r");
    while (token != NULL) {
        convertir_a_minusculas(token);
        limpiar_palabra(token);
        agregar_palabra(&indice, token, entrada->d_name);
        token = strtok(NULL, " \t\n\r");
    }
    free(contenido);
}
closedir(dir);
```

Code 33. Procesamiento de documentos

Cálculo del PageRank: Usa `calculate_pagerank` para evaluar la importancia relativa de cada documento.

```
calculate_pagerank(nodes_web1, docs_web1, DAMPING_FACTOR, ITERATIONS);
calculate_pagerank(nodes_web2, docs_web2, DAMPING_FACTOR, ITERATIONS);

printf("\n=== PageRank Resultados Web 1 ===\n");
for (int i = 0; i < docs_web1; i++) {
    printf("Documento %d - PageRank: %f\n", i, nodes_web1[i]->pagerank);
}

printf("\n=== PageRank Resultados Web 2 ===\n");
for (int i = 0; i < docs_web2; i++) {
    printf("Documento %d - PageRank: %f\n", i, nodes_web2[i]->pagerank);
}
```

Code 34. Cálculo del PageRank

Búsqueda de palabras clave: Permite al usuario buscar términos ingresados como argumentos en la línea de comandos.

```
for (int i = 1; i < argc; i++) {
    char palabra_a_buscar[512];
    strncpy(palabra_a_buscar, argv[i], sizeof(palabra_a_buscar) - 1);
    convertir_a_minusculas(palabra_a_buscar);
    limpiar_palabra(palabra_a_buscar);
    if (es_stopword(palabra_a_buscar, stopwords, num_stopwords)) {
        printf("\%s es un stopword. Omitiendo busqueda...\n", palabra_a_buscar);
        continue;
    }
    printf("\n\n=== RESULTADOS PARA \"%s\" ===\n", palabra_a_buscar);
    buscar_palabra(&indice, palabra_a_buscar);
}
```

Code 35. Búsqueda de palabras clave

Liberación de memoria: Asegura que todos los recursos sean liberados al finalizar la ejecución.

```
if (web1 != NULL) free_graph(web1);
if (web2 != NULL) free_graph(web2);
if (nodes_web1 != NULL) free_nodes(nodes_web1, docs_web1);
if (nodes_web2 != NULL) free_nodes(nodes_web2, docs_web2);
liberar_indice(&indice);
```

Code 36. Liberación de memoria

Propósito en el sistema

Este archivo integra y ejecuta todas las funcionalidades del sistema, coordinando la interacción entre módulos y estructuras para implementar un sistema funcional de recuperación de información.

Impacto en el programa

- Representa el punto central del sistema, donde se realiza el flujo de control y se integran las funciones desarrolladas en los módulos auxiliares.
- Coordina la indexación, búsqueda y análisis, mostrando resultados al usuario de manera interactiva.

4. Mejoras

0.0.1. Posibles mejoras al código

Aunque el sistema implementado cumple con los objetivos planteados, siempre existen oportunidades para optimizar y extender su funcionalidad. A continuación, se presentan algunas posibles mejoras que podrían implementarse en el código:

1. Modularización adicional

- **Descripción**: Aunque el sistema ya utiliza múltiples archivos de encabezado y módulos para organizar el código, se podrían separar responsabilidades más específicas en módulos adicionales, como un módulo dedicado exclusivamente al manejo de argumentos de línea de comandos o a la gestión de errores.
- **Impacto**: Esto mejoraría la legibilidad y mantenibilidad del código, facilitando futuras modificaciones y depuraciones.

2. Paralelización del cálculo de PageRank

- **Descripción**: El cálculo del PageRank podría paralelizarse utilizando bibliotecas como OpenMP o Pthreads. Cada iteración del algoritmo podría distribuirse entre múltiples hilos para acelerar el procesamiento.
- **Impacto**: Esto sería especialmente útil para manejar grafos grandes, reduciendo significativamente el tiempo de ejecución.

3. Mejora en la gestión de memoria

- **Descripción**: Implementar funciones más robustas para verificar y gestionar la asignación y liberación de memoria, como comprobaciones explícitas después de cada asignación de memoria dinámica.
- **Impacto**: Esto reduciría el riesgo de fugas de memoria y errores de segmentación, mejorando la estabilidad del sistema.

4. Inclusión de consultas booleanas

- **Descripción**: Actualmente, el sistema permite buscar palabras individuales. Se podría extender la funcionalidad para manejar consultas booleanas como "palabra1 AND palabra2" o "palabra1 OR palabra2".
- **Impacto**: Esto haría que el sistema sea más flexible y útil para los usuarios, permitiendo búsquedas más complejas.

5. Optimización del índice invertido

- **Descripción**: Reemplazar la tabla hash estática utilizada para el índice invertido por una estructura de datos dinámica y autoajustable, como un árbol B o un trie.
- **Impacto**: Esto mejoraría la eficiencia en la búsqueda y almacenamiento de palabras clave, especialmente cuando se maneja un volumen de datos grande y variable.

6. Implementación de una interfaz gráfica

- **Descripción**: Incorporar una interfaz gráfica de usuario (GUI) utilizando bibliotecas como GTK o Qt para hacer el sistema más amigable para usuarios sin conocimientos técnicos.
- **Impacto**: Esto ampliaría el público objetivo del sistema, permitiendo que usuarios no técnicos interactúen con la herramienta.

7. Incorporación de más métricas para el ranking

- **Descripción**: Además del PageRank, podrían incluirse métricas adicionales como TF-IDF (Term Frequency-Inverse Document Frequency) para evaluar la relevancia de los documentos en función de las palabras clave buscadas.
- **Impacto**: Esto aumentaría la precisión de los resultados mostrados al usuario.

8. Soporte para múltiples formatos de archivo

- **Descripción**: Actualmente, el sistema procesa archivos de texto plano. Se podría extender el soporte para manejar formatos como PDF, DOCX o HTML, utilizando bibliotecas como `poppler` o `libxml2`.
- **Impacto**: Esto haría que el sistema sea más versátil y aplicable a una variedad más amplia de casos de uso.

9. Validación de entrada de usuario

- **Descripción**: Implementar validaciones más exhaustivas para las palabras ingresadas por los usuarios y los argumentos de línea de comandos, con mensajes de error más detallados.
- **Impacto**: Esto mejoraría la experiencia del usuario al evitar errores inesperados y proporcionar retroalimentación más clara.

10. Pruebas automatizadas

- **Descripción**: Crear un conjunto de pruebas automatizadas para validar las funcionalidades del sistema, utilizando herramientas como CUnit.
- **Impacto**: Esto garantizaría que futuras modificaciones en el código no introduzcan errores, mejorando la confiabilidad del sistema.

5. Conclusiones

El sistema desarrollado para la recuperación de información y simulación del PageRank cumple con los objetivos planteados, integrando de manera eficiente las distintas estructuras de datos y algoritmos necesarios. A continuación, se presentan las conclusiones detalladas, destacando los aspectos positivos y las limitaciones observadas en el proyecto.

Aspectos positivos

1. **Cumplimiento funcional**:
    - El programa implementa todas las funcionalidades requeridas, desde la gestión de *stopwords* hasta el cálculo del PageRank y la recuperación de información mediante un índice invertido.
    - Las estructuras de datos utilizadas, como listas de adyacencia, nodos y tablas hash, son apropiadas y garantizan un desempeño adecuado para los casos planteados.
2. **Modularidad**: El código está dividido en múltiples módulos, lo que facilita su comprensión, depuración y modificación. Cada archivo de encabezado y fuente está claramente relacionado con un aspecto del sistema.
3. **Precisión en la recuperación de información**: La combinación del algoritmo PageRank con la eliminación de *stopwords* y la normalización de palabras asegura resultados relevantes para las búsquedas realizadas.
4. **Uso eficiente de recursos**: El sistema emplea estructuras de datos eficientes para manejar grafos dispersos y grandes volúmenes de palabras, optimizando el uso de memoria y el tiempo de ejecución.
5. **Escalabilidad básica**: El diseño actual permite manejar un número razonable de documentos y palabras clave, lo que lo hace útil para aplicaciones pequeñas o medianas.

Limitaciones (Contras)

1. **Rendimiento limitado en escalas grandes**: Aunque el sistema funciona bien con un número moderado de documentos, el rendimiento podría verse afectado con grafos extremadamente grandes debido a la falta de paralelización y optimizaciones avanzadas.
2. **Dependencia de formato de entrada**: Actualmente, el sistema solo maneja archivos de texto plano, lo que limita su aplicabilidad en casos donde los datos están en formatos como PDF o HTML.
3. **Falta de validaciones robustas**: Aunque se realizan algunas verificaciones básicas, el manejo de errores en entradas y salidas podría mejorarse para evitar fallos inesperados durante la ejecución.
4. **Complejidad en el uso**: El sistema requiere conocimientos técnicos para ser utilizado de manera efectiva, ya que no cuenta con una interfaz gráfica que facilite la interacción con usuarios no técnicos.
5. **Gestión de memoria básica**: Si bien se implementan funciones para liberar recursos, la falta de validaciones avanzadas podría resultar en fugas de memoria en escenarios no previstos.
6. **Falta de integración con métricas avanzadas**: Actualmente, el sistema solo utiliza PageRank como métrica para evaluar la importancia de los nodos. La ausencia de otras métricas como TF-IDF limita la precisión en ciertos casos.

Reflexión final

Este proyecto demuestra una implementación sólida y funcional de un sistema de recuperación de información basado en estructuras de datos y algoritmos fundamentales. Aunque tiene limitaciones inherentes a un prototipo académico, ofrece una base sólida para futuras extensiones y mejoras. La modularidad del código, combinada con las oportunidades de optimización, lo convierte en un sistema escalable y adaptable a aplicaciones más complejas en el futuro.

6. Referencias

Brin, S., & Page, L. (1998). *The anatomy of a large-scale hypertextual Web search engine*. Computer Networks and ISDN Systems, 30(1–7), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Facultad de Ingeniería en Computación, Universidad de Magallanes. (n.d.). *Estructuras de datos: Apuntes para clases*. Obtenido de https://kataix.umag.cl/j̃canuman&8203;:contentReference[oaicite:0]{index=0}. GNU Project. (2024). *The GNU C library*. Retrieved from https://www.gnu.org/software/libc/

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.

Weiss, M. A. (2013). *Data structures and algorithm analysis in C* (3rd ed.). Addison-Wesley.

Wirth, N. (1971). *Program development by stepwise refinement*. Communications of the ACM, 14(4), 221–227. https://doi.org/10.1145/362575.362577