UMAG Universidad de Magallanes | Facultad de Ingeniería

Informe simulador de búsqueda

Miguel Maripillan[†], Diego Sanhueza[†], Miguel Loaiza[†] y Felipe Cárcamo[†]

[†] *Universidad de Magallanes*

This manuscript was compile on 20 de noviembre de 2024

Resumen

Este proyecto se enfoca en la creación de un sistema que combina la búsqueda eficiente de palabras clave mediante un índice invertido y la clasificación de documentos usando un grafo dirigido y el algoritmo PageRank. La solución emplea estructuras de datos como tablas hash y listas de adyacencia para representar eficientemente palabras clave y enlaces entre documentos. El simulador desarrollado permite realizar búsquedas rápidas, clasificando los resultados en función de la importancia relativa calculada mediante PageRank. Se incluyen mecanismos para la creación automatizada de documentos, tokenización y filtrado de palabras irrelevantes, así como la gestión y visualización del grafo resultante. Este trabajo destaca la importancia de implementar algoritmos de búsqueda eficientes y estructuras de datos optimizadas para la clasificación de información, facilitando la comprensión y aplicación práctica en proyectos de mayor complejidad.

Keywords: *PageRank, Simulador, Indice invertido, búsqueda*

Índice

1 Introducción **2**

2 Entendiendo el problema **2**

2.1 Estructuras de datos . . . 2
2.2 Algoritmos . . . 2
2.3 Resultado del simulador . . . 2

3 Definiendo estructuras y funciones **2**

3.1 Prototipo de las funciones principales . . . 2
3.2 Estructuras . . . 3

4 Explicación de las funciones principales **4**

4.1 PageRank . . . 4
4.2 Construcción del índice invertido . . . 5
4.3 Conexión de las aristas con los vértices . . . 6
4.4 Creación de archivos . . . 7

5 Hilo de ejecución y salida del programa **10**

5.1 Entrada del programa . . . 10
src2 Creación de Archivos • src Simulador de búsqueda
5.2 Salida del programa . . . 12

6 Dificultades enfrentadas **13**

6.1 Hasheo y tokenización . . . 13
6.2 Separación de archivos . . . 13
6.3 Generación de la Imagen EPS . . . 14

7 Conclusión **14**

1. Introducción

En este proyecto se desarrolla un simulador de búsqueda que combina la creación de un índice invertido para realizar búsquedas eficientes de palabras clave y un grafo dirigido para modelar las relaciones entre documentos mediante el algoritmo PageRank. El trabajo emplea estructuras de datos como listas enlazadas, tablas hash y listas de adyacencia para representar tanto las palabras clave como los enlaces entre documentos, asegurando una gestión eficaz de las consultas y la clasificación de resultados.

El proyecto se divide en dos partes principales: la generación de documentos de texto, que simulan contenido web con enlaces aleatorios, y el simulador de búsqueda que permite buscar palabras en los documentos, clasificando los resultados por su relevancia calculada a través de PageRank. Se implementan funciones para tokenizar y filtrar palabras, construir índices y grafos, y calcular la importancia de cada documento en función de sus enlaces.

2. Entendiendo el problema

Para poder implementar el simulador, se nos piden los siguientes requerimientos;

2.1. Estructuras de datos

Usar listas enlazadas y tablas hash para el indice invertido, representar el grafo de enlaces entre documentos mediante listas de adyacencias.

2.2. Algoritmos

Implementar el algoritmo PageRank en iteración usando un factor de amortiguación *(damping factor)*, tener un algoritmo de búsqueda eficiente en el indice invertido.

2.3. Resultado del simulador

Al finalizar el programa se debe mostrar una lista de documentos ordenados por relevancia dado una palabra o frase.

3. Definiendo estructuras y funciones

Para poder implementar las funciones requeridas, primero se define un archivo .h con los prototipos de las funciones y las estructuras necesarias.

3.1. Prototipo de las funciones principales

```
1
2  1
3  2  /* Funciones Grafos */
4  3  void initialize_graph(Graph *);
5  4  void add_edge(Graph *, int, int);
6  5  void build_graph(Graph *);
7  6  void release_graph(Graph *);
8  7  int count_output_links(Graph *, int);
9  8  int count_input_links(Graph *, int);
10 9  int get_doc_id(Graph *, char *);
11 10 bool is_doc_name(char *);
12 11 void show_graph(Graph *);
13 12 void generate_eps(const Graph *, const double *, const char *);
14 13
15 14 /* Funciones Indice invertido */
16 15 InvertedIndex *create_new_node(char *);
17 16 void add_document(InvertedIndex **, int, char *);
18 17 bool is_stopword(char *);
19 18 void tokenize_text(char *, int, InvertedIndex **);
20 19 void print_inverted_index(InvertedIndex *);
21 20 void print_search_word_with_pagerank(InvertedIndex **, char *, Graph *, double *);
22 21 unsigned int hash_function(char *);
23 22 Node *search_word(InvertedIndex **, char *);
24 23 void build_index(Graph *, InvertedIndex **);
25 24 void release_inverted_index(InvertedIndex **);
26 25
27 26 /* Funciones PageRank */
28 27 void initialize_pagerank(double *, int);
29 28 void calculate_pagerank(Graph *, double *);
30 29 void display_pagerank(Graph *, double *);
31 30
32 31 /* Funciones para generacion de archivos */
33 32 void generate_text_files(int, int);
34 33 void generate_random_text(FILE *, const char *, int, int, int, int *);
35
```

Código 1. Prototipo de las funciones.

Estas funciones nos ayudarán a ejecutar las distintas tareas que el simulador debe realizar, mas adelante serán explicadas las funciones principales.

3.2. Estructuras

```c
// Estructura Graph que representa el grafo de documentos con enlaces entrantes y salientes.
typedef struct Graph
{
    Node *output_adjacent_list[MAX_DOCS];   // Lista de enlaces salientes.
    Node *input_adjacent_list[MAX_DOCS];    // Lista de enlaces entrantes.
    DocumentMapping mapping_docs[MAX_DOCS]; // Mapeo de documentos.
    int total_docs;                         // Total de documentos en el grafo.
} Graph;
```

Código 2. Estructura del grafo.

Graph: Esta estructura representa el grafo de documentos y los enlaces entre ellos, implementado mediante listas de adyacencia.
Atributos:

- `Node *output_adjacent_list[MAX_DOCS]`: Array de listas enlazadas para enlaces salientes desde cada documento.
- `Node *input_adjacent_list[MAX_DOCS]`: Array de listas enlazadas para enlaces entrantes a cada documento.
- `DocumentMapping mapping_docs[MAX_DOCS]`: Mapeo de nombres de documentos a identificadores únicos.
- `int total_docs`: Número total de documentos representados en el grafo.

```c
// Estructura InvertedIndex para el indice invertido, almacenando palabras y los documentos donde aparecen.
typedef struct InvertedIndex
{
    char word[MAX_WORD_SIZE];   // Palabra almacenada.
    Node *docs_list;            // Lista de documentos donde aparece la palabra.
    struct InvertedIndex *next; // Siguiente nodo en la lista enlazada del indice.
} InvertedIndex;
```

Código 3. Estructura del indice invertido.

InvertedIndex: Esta estructura representa un nodo en una lista enlazada para el índice invertido, el cual almacena palabras y los documentos en los que aparecen.
Atributos:

- `char word[MAX_WORD_SIZE]`: Palabra indexada.
- `Node *docs_list`: Lista enlazada de documentos donde aparece la palabra.
- `struct InvertedIndex *next`: Puntero al siguiente nodo en la lista enlazada del índice.

```c
// Estructura Node para la lista enlazada de enlaces en el grafo.
typedef struct Node
{
    int doc_id;        // Identificador del documento enlazado.
    struct Node *next; // Puntero al siguiente nodo.
} Node;
```

Código 4. Estructura del nodo.

Node: Esta estructura representa un nodo en una lista enlazada que almacena enlaces entre documentos en el grafo. Se utiliza tanto en la lista de adyacencia de entrada como en la de salida.
Atributos:

- `int doc_id`: Identificador del documento enlazado, representando un vértice en el grafo.
- `struct Node *next`: Puntero al siguiente nodo en la lista, permitiendo crear una lista enlazada de enlaces.

```c
// Estructura para mapear el nombre de documentos a sus identificadores unicos.
typedef struct DocumentMapping
{
    char name[MAX_NAME_WEB]; // Nombre del documento.
    int doc_id;              // Identificador unico del documento.
} DocumentMapping;
```

Código 5. Estructura del mapeo de documentos.

DocumentMapping: Mapea los nombres de los documentos a identificadores únicos, lo cual es útil para gestionar documentos en el grafo sin confusión.

Atributos:

- `char name[MAX_NAME_WEB]`: Nombre del documento (archivo) en el grafo.
- `int doc_id`: Identificador único asociado al documento para fácil referencia.

4. Explicación de las funciones principales

4.1. PageRank

Se implementa el algoritmo de PageRank para calcular la importancia relativa de cada documento en un grafo web, utilizando enlaces como referencia para el cálculo de la influencia entre documentos. La implementación consta de tres funciones principales: `initialize_pagerank()`, `calculate_pagerank()`, y `display_pagerank()`.

- **`initialize_pagerank()`**: Esta función se encarga de inicializar el vector de PageRank. Cada documento comienza con un valor uniforme de `1 / total_docs`, de modo que todos los documentos tengan el mismo PageRank inicial. Esto es importante para asegurar que el cálculo sea equitativo desde el principio.
- **`calculate_pagerank()`**: Esta función calcula iterativamente el PageRank para cada documento en el grafo hasta que el cálculo converja o hasta alcanzar el número máximo de iteraciones, determinado por la constante `MAX_ITERATIONS`.
  El proceso de cálculo se realiza de la siguiente manera:

  1. Se inicializa el vector `new_pagerank` para almacenar los valores de PageRank temporales en cada iteración.
  2. Para cada documento, se calcula la suma de los valores de PageRank de los documentos que tienen enlaces hacia él (inbound links). Este cálculo se realiza dividiendo el valor de PageRank del documento que apunta por su cantidad de enlaces salientes, de modo que cada enlace contribuye proporcionalmente a los documentos de destino.
  3. Se utiliza un factor de amortiguación (`DAMPING_FACTOR`) para calcular el nuevo valor de PageRank. Este factor permite ajustar la probabilidad de que un usuario siga navegando a través de los enlaces del documento. La fórmula para actualizar el PageRank es:

$$\text{nuevo\_pagerank}[i] = \frac{(1 - \text{DAMPING\_FACTOR})}{\text{graph->total\_docs}} + \text{DAMPING\_FACTOR} \times \text{inbound\_rank\_sum}$$

  4. Después de cada actualización de PageRank, se verifica la convergencia comparando el cambio en el PageRank de cada documento con un umbral, `CONVERGENCE_THRESHOLD`. Si todos los cambios están por debajo del umbral, el proceso de iteración se detiene y el cálculo se considera convergido.
  5. Al finalizar cada iteración, el vector original `pagerank` se actualiza con los valores calculados en `new_pagerank`.

  Si el cálculo converge antes de alcanzar el número máximo de iteraciones, se imprime un mensaje indicando el número de iteraciones necesarias para la convergencia.
- **`display_pagerank()`**: Esta función muestra los resultados del PageRank para cada documento. Imprime en la consola el ID de cada documento junto con su valor de PageRank calculado, permitiendo así evaluar la importancia relativa de cada documento en el conjunto.

```c
// Calcula el PageRank para cada documento en el grafo.
void calculate_pagerank(Graph *graph, double pagerank[])
{
    double new_pagerank[MAX_DOCS];                    // Vector temporal para almacenar los nuevos valores de PageRank en cada iteraci n.
    initialize_pagerank(pagerank, graph->total_docs); // Inicializa el PageRank.

    // Itera para calcular el PageRank hasta que converja o se alcance el m ximo de iteraciones.
    for (int iter = 0; iter < MAX_ITERATIONS; iter++)
    {
        bool converged = true;

        // Calcula el nuevo PageRank para cada documento.
        for (int i = 0; i < graph->total_docs; i++)
        {
            double inbound_rank_sum = 0.0;

            // Recorre los documentos que enlazan al documento 'i'.
            Node *inbound = graph->input_adjacent_list[i];
            while (inbound != NULL)
            {
                int inbound_doc_id = inbound->doc_id;                    // ID del documento de entrada.
                int out_links = count_output_links(graph, inbound_doc_id); // N mero de enlaces salientes desde el documento.

                // Si el documento tiene enlaces salientes, acumula el PageRank proporcional.
                if (out_links > 0)
                    inbound_rank_sum += pagerank[inbound_doc_id] / out_links;

                inbound = inbound->next;
            }

            // Calcula el nuevo valor de PageRank utilizando el factor de amortiguaci n.
            new_pagerank[i] = (1 - DAMPING_FACTOR) / graph->total_docs + DAMPING_FACTOR * inbound_rank_sum;

            // Verifica si el cambio entre el nuevo y el viejo PageRank supera el umbral de convergencia.
            if (fabs(new_pagerank[i] - pagerank[i]) > CONVERGENCE_THRESHOLD)
                converged = false;
        }

        // Actualiza el vector de PageRank con los nuevos valores calculados.
        for (int i = 0; i < graph->total_docs; i++)
            pagerank[i] = new_pagerank[i];

        // Si todos los valores de PageRank han convergido, finaliza el c lculo.
        if (converged)
        {
            printf("PageRank convergido en %d iteraciones.\n", iter + 1);
            break;
        }
    }
}
```

Código 6. Extracto del codigo del PageRank.

Se obtuvo información de Wikipedia: [1], igual se visitó GeeksforGeeks [3] y se revisó la página Memgraph [4] para documentarse acerca del pagerank.

4.2. Construcción del índice invertido

El índice invertido es una estructura de datos que permite asociar cada palabra con los documentos en los que aparece, facilitando búsquedas rápidas en un conjunto de textos. A continuación, se describe cómo el programa construye y gestiona este índice invertido paso a paso:

1. **Crear un nodo para una nueva palabra**: La función `create_new_node` es responsable de crear un nuevo nodo en el índice invertido para cada palabra que no se ha encontrado previamente. Este nodo almacena la palabra y una lista de documentos donde aparece. Cuando se detecta una palabra nueva, esta función crea el espacio necesario en memoria para el nodo, inicializa la lista de documentos como vacía, y prepara el nodo para ser agregado al índice.

2. **Agregar documentos al índice**: Cada vez que se encuentra una palabra en un documento, la función `add_document` revisa si ya existe un nodo para esa palabra en el índice. Si la palabra ya tiene un nodo, simplemente agrega el identificador del documento (o `doc_id`) a la lista de documentos de esa palabra. Si la palabra es nueva, crea un nodo y lo añade a la tabla hash del índice, permitiendo que en el futuro se pueda ubicar la palabra y sus documentos de forma rápida.

3. **Ignorar palabras irrelevantes**: Para evitar sobrecargar el índice con palabras comunes y sin valor de búsqueda, la función `is_stopword` identifica palabras irrelevantes (*stopwords*) como “a” o “e”. Estas palabras son ignoradas para hacer el índice más eficiente y relevante, asegurando que solo se incluyan términos útiles para las búsquedas.

4. **Procesamiento de texto para indexación**: La función `tokenize_text` prepara el texto del documento para su indexación. Primero, convierte todo el texto a minúsculas y elimina signos de puntuación, lo que permite que palabras como “Hola” y “hola” sean tratadas como iguales. Luego, el texto se divide en palabras o “tokens”, y cada palabra que no sea una stopword se añade al índice con la función `add_document`.

5. **Cálculo de posiciones de palabras**: Para ubicar cada palabra de manera rápida, se utiliza una función de *hash* (`hash_function`) que convierte la palabra en un número único. Este número sirve como índice para almacenar y recuperar el nodo de cada palabra en la tabla hash, optimizando la eficiencia de las búsquedas.

6. **Buscar palabras en el índice**: La función `search_word` permite buscar documentos asociados a una palabra en el índice. Dada una palabra, se calcula su posición en la tabla usando la función de hash y se revisa si existe en el índice. Si la palabra está presente, se devuelve la lista de documentos donde aparece; si no, se retorna un valor nulo.

7. **Liberación de memoria**: Finalmente, la función `release_inverted_index` libera toda la memoria utilizada por el índice invertido cuando ya no se necesita. Esta función recorre cada palabra y su lista de documentos, asegurándose de liberar cada nodo y evitar fugas de memoria en el programa.

De esta manera, el índice invertido permite construir un mapa eficiente de palabras a documentos, optimizando así las búsquedas de palabras en grandes conjuntos de texto. Utiliza técnicas de hash y listas enlazadas para asociar cada palabra con los documentos donde aparece, todo en un formato de búsqueda rápida y organizado.

```c
// Agrega un documento al indice invertido, asociandolo con una palabra especifica.
void add_document(InvertedIndex **hash_table, int doc_id, char *word)
{
    // Calcula el indice en la tabla hash para la palabra.
    unsigned int index = hash_function(word);
    InvertedIndex *current = hash_table[index];

    // Busca la palabra en el indice.
    while (current != NULL)
    {
        // Si la comparacion se cumple, la palabra ya esta en el indice
        if (strcmp(current->word, word) == 0)
        {
            Node *new_doc = (Node *)malloc(sizeof(Node));
            new_doc->doc_id = doc_id;
            new_doc->next = current->docs_list;
            current->docs_list = new_doc;
            return;
        }
        current = current->next;
    }

    // Si la palabra no se encuentra en el indice, crea un nuevo nodo en el indice.
    InvertedIndex *new_node = create_new_node(word);
    new_node->next = hash_table[index]; // Posiciona el nuevo nodo en el indice calculado.
    hash_table[index] = new_node;

    // Crea y asigna el nodo de documento.
    Node *new_doc = (Node *)malloc(sizeof(Node));
    if (new_doc == NULL)
    {
        fprintf(stderr, "Error al asignar memoria para nuevo documento\n");
        exit(EXIT_FAILURE);
    }
    new_doc->doc_id = doc_id;
    new_doc->next = NULL;
    new_node->docs_list = new_doc;
}
```

Código 7. Función para añadir un nuevo documento al indice invertido.

Se visitó GeeksforGeeks para tener información sobre el índice invertido [2].

4.3. Conexión de las aristas con los vértices

En el grafo dirigido, los documentos se representan como vértices y los enlaces entre ellos como aristas. Para gestionar estos enlaces, se utilizan listas de adyacencia, donde cada documento tiene una lista que contiene los documentos a los que enlaza (enlaces salientes) y una lista de los documentos que lo enlazan (enlaces entrantes).

La función `add_edge()` es la encargada de agregar un enlace entre dos documentos en el grafo. Esta función realiza las siguientes acciones:

- Primero, se crea un nodo en la lista de adyacencia de salientes para el documento de origen. Este nodo almacena el ID del documento de destino al que el documento de origen enlaza.

- Luego, se crea un nodo en la lista de adyacencia de entrantes para el documento de destino. Este nodo almacena el ID del documento de origen desde el cual el documento de destino recibe un enlace.

Ambos nodos se insertan al principio de sus respectivas listas, de manera que cada documento tiene una lista que enlaza a los documentos a los que hace referencia y una lista que lo vincula con los documentos que lo referencian. Esto permite un acceso rápido a los documentos relacionados con cualquier documento dado en el grafo.

En la inicialización del grafo, las listas de adyacencia de salientes y entrantes se inicializan a `NULL`, lo que indica que no hay enlaces. Posteriormente, cuando se procesa un archivo que contiene enlaces, se agregan los nodos correspondientes a las listas de adyacencia de los documentos implicados.

Es importante destacar que, cuando se agrega un nuevo enlace, se verifica si se puede asignar memoria para el nuevo nodo. Si la asignación de memoria falla, se imprime un mensaje de error y el programa termina.

Las aristas en el grafo se conectan a los vértices mediante las listas de adyacencia de salientes y entrantes. Cada vez que se encuentra un enlace en un archivo, se actualizan las listas de adyacencia correspondientes para reflejar la relación entre los documentos, permitiendo la navegación eficiente entre ellos.

```

// Agrega un enlace dirigido entre dos documentos en el grafo.
void add_edge(Graph *graph, int source, int destination)
{
    // Crea nodo en la lista de enlaces salientes desde el documento de origen al destino.
    Node *newOutputNode = (Node *)malloc(sizeof(Node));
    if (newOutputNode == NULL) // Verifica la asignacion de memoria.
    {
        printf("Error al asignar memoria para enlace saliente\n");
        exit(EXIT_FAILURE);
    }
    newOutputNode->doc_id = destination;                        // Asigna el ID de destino al nodo.
    newOutputNode->next = graph->output_adjacent_list[source]; // Inserta al inicio de la lista
    graph->output_adjacent_list[source] = newOutputNode;

    // Crea nodo en la lista de enlaces entrantes al documento destino desde el origen.
    Node *newInputNode = (Node *)malloc(sizeof(Node));
    if (newInputNode == NULL) // Verifica la asignaci n de memoria.
    {
        printf("Error al asignar memoria para enlace entrante\n");
        exit(EXIT_FAILURE);
    }
    newInputNode->doc_id = source;                                  // Asigna el ID del origen al nodo.
    newInputNode->next = graph->input_adjacent_list[destination]; // Inserta al inicio de la lista.
    graph->input_adjacent_list[destination] = newInputNode;
}
```

Código 8. Función para añadir un enlace entre dos documentos.

4.4. Creación de archivos

Se generaron los archivos de texto que simulan documentos web, con contenido aleatorio y enlaces a otros documentos dentro de la misma carpeta. Cada archivo creado representa un documento que podría pertenecer a un grafo web. La función principal para esta tarea es `generate_text_files()`, la cual se encarga de crear múltiples archivos, mientras que `generate_random_text()` se utiliza para rellenar cada archivo con contenido aleatorio y enlaces.

La función `generate_text_files()` recibe como parámetro el número de documentos a generar y realiza los siguientes pasos:

- Primero, verifica que el número de documentos sea positivo. Si el valor es menor o igual a cero, imprime un mensaje de error en la salida estándar de error y termina el programa, ya que al menos un archivo debe generarse.
- A continuación, por medio de un ciclo, verifica que cada documento tenga al menos un enlace.
- Luego, para cada documento a crear, genera un nombre único en el formato `doc<i>.txt` (por ejemplo, `doc1.txt`, `doc2.txt`, etc.), donde `i` es el número del documento.
- Intenta abrir cada archivo en modo de escritura. Si no puede abrir el archivo, imprime un mensaje de error en la salida estándar de error y termina el programa.
- Finalmente, llama a la función `generate_random_text()` para llenar el archivo con texto y enlaces, y luego cierra el archivo.

La función `generate_random_text()` tiene la responsabilidad de escribir el contenido en cada archivo, recibe como parámetro el número de caracteres a generar y realiza las siguientes acciones:

- Primero, verifica que el número de caracteres sea positivo. Si el valor es menor o igual a cero, imprime un mensaje de error en la salida estándar de error y termina el programa, ya que debe generarse al menos un carácter.
- Después, Escribe letras aleatorias en el archivo, generando un total de `num_characters` caracteres, que es el ingresado anteriormente. Para cada carácter, se selecciona una letra mayúscula aleatoria entre `'A'` y `'Z'`, y se escribe en el archivo. Entre cada letra se agrega un espacio, lo que simula contenido de texto.

- Luego, genera un número aleatorio de enlaces en relación con la cantidad de archivos generados, hacia otros documentos del conjunto. Para cada enlace, selecciona un número de documento de destino de forma aleatoria. Si el documento seleccionado no es el mismo documento en el que se está escribiendo, se añade una línea en el formato `link: doc<i>` para representar un enlace a otro documento.

```
// Genera texto aleatorio y enlaces en un archivo web espec fico.
void generate_random_text(FILE *doc, const char *doc_name, int num_docs, int num_characters, int current_doc, int *links)
{
    // Genera texto aleatorio dentro de cada archivo web.
    for (int i = 0; i < num_characters; i++)
    {
        char letter = 'A' + rand() % 26; // Genera una letra aleatoria entre 'A' y 'Z'.
        fprintf(doc, "%c", letter);      // Escribe la letra en el archivo.
        if (i < num_characters - 1)
            fprintf(doc, " "); // Agrega un espacio entre las letras.
    }

    fprintf(doc, "\nlink: doc%d", links[current_doc - 1]); // Asegura al menos un enlace  nico .

    // Incluye aleatoriamente enlaces a otros documentos.
    int extra_links = rand() % num_docs; // Genera enlaces respecto al numero de documentos.
    for (int i = 0; i < extra_links; i++)
    {
        int link_doc = rand() % num_docs + 1;                              // Elige un documento aleatorio de 1 a num_docs.
        if (link_doc != current_doc && link_doc != links[current_doc - 1]) // Evita que un documento enlace a s   mismo.
            fprintf(doc, "\nlink: doc%d", link_doc);                       // Agrega un enlace al archivo.
    }

    fprintf(stdout, "ARCHIVO '%s' generado con  XITO  con %d letras y enlaces.\n", doc_name, num_characters);
}
```

Código 9. Función de generación de texto random.

```c
void generate_text_files(int num_docs, int num_characters)
{
    if (num_docs <= 0 || num_docs >= 100) // Verifica que se proporcione al menos un documento.
    {
        fprintf(stderr, "El n mero de archivos web a generar debe ser MAYOR a 0 y MENOR a 100.\n\n");
        exit(EXIT_FAILURE);
    }

    // Array para asegurar que cada documento tenga al menos un enlace.
    int *links = malloc(num_docs * sizeof(int));

    for (int i = 0; i < num_docs; i++)
        links[i] = (i + 1) % num_docs + 1; // Crea un ciclo simple para asegurar al menos un enlace por archivo.

    // Itera para crear el n mero especificado de archivos de texto.
    for (int i = 1; i <= num_docs; i++)
    {
        char doc_name[MAX_NAME_DOC];
        snprintf(doc_name, sizeof(doc_name), "doc%d.txt", i); // Genera un nombre para el archivo (e.g., "doc1.txt").

        // Crea el archivo en modo escritura. // Termina el programa si no se puede abrir el archivo.
        FILE *doc = fopen(doc_name, "w");
        if (doc == NULL)
        {
            fprintf(stderr, "Error al abrir el archivo web.\n");
            free(links);
            exit(EXIT_FAILURE);
        }

        // Genera contenido aleatorio y enlaces dentro del archivo.
        generate_random_text(doc, doc_name, num_docs, num_characters, i, links);

        fclose(doc);
    }
}
```

Código 10. Función de generación de archivos.

5. Hilo de ejecución y salida del programa

Recorrido del programa

Para poder ver todo el funcionamiento del programa, puede visitar el repositorio en *GitHub*, acá se detallarán las cosas mas relevantes para dar un vistazo general del proyecto.

5.1. Entrada del programa

Para esta parte hay que considerar que el programa se divide en dos carpetas `src`, una que contiene los archivos para la creación de documentos(`src2`) y otra que contiene los archivos para el simulador de búsqueda(`src`).

5.1.1. src2 Creación de Archivos

El programa comienza con la inclusión de los archivos de cabecera `graph.h` y `doc.h`, que definen las estructuras y funciones necesarias para el proceso de creación de archivos. A continuación, el `main()` de `src2` gestiona los argumentos proporcionados por la línea de comandos, utilizando la función `getopt()` para manejar las opciones disponibles. El programa acepta tres opciones principales:

- **-h**: Muestra un mensaje de ayuda que explica cómo utilizar el programa, indicando que se debe usar la opción `-d` y `-c` seguidas de un número para especificar la cantidad de archivos a generar y la cantidad de caracteres por archivo.
- **-d**: Permite al usuario especificar la cantidad de documentos a procesar. El valor se recibe como un argumento entero y se almacena en la variable `num_docs`. Si no se especifica un valor o si el valor es menor o igual a cero, el programa mostrará un mensaje de error y finalizará.
- **-c**: Permite al usuario definir la cantidad de caracteres que se generarán en cada archivo. El valor se recibe como un argumento entero y se almacena en la variable `num_characters`. Si no se proporciona un valor o si el valor es menor o igual a cero, el programa mostrará un mensaje de error y finalizará.

Si el usuario no proporciona los parámetros adecuados, se mostrarán mensajes de error, ya sea por una opción no reconocida o un valor inválido para el número de documentos.

El flujo del programa es el siguiente:

- Primero, se inicializan las variables y estructuras necesarias.
- Luego, se llama a la función `generate_text_files()` para crear los archivos de texto según la cantidad de documentos especificada.
- Dentro de cada archivo generado, se invoca la función `generate_random_text()` para generar el contenido de caracteres.
- Finalmente, se imprime un mensaje de confirmación indicando la creación exitosa de los archivos.

A continuación, se muestra el fragmento relevante del código que gestiona la entrada de los parámetros:

```
    int opt;
    int num_docs = 0;
    int num_characters = 0;

    while ((opt = getopt(argc, argv, "hd:c:")) != -1)
    {
        switch (opt)
        {
        case 'h':
            fprintf(stdout, "\nUso del programa:\n");
            fprintf(stdout, "  -d <numero_de_archivos> : N mero de archivos a crear.\n");
            fprintf(stdout, "  -c <numero_de_caracteres> : N mero de caracteres por archivo.\n");
            fprintf(stdout, "  -h : Muestra esta ayuda.\n\n");
            exit(EXIT_SUCCESS);
            break;
        case 'd':
            num_docs = atoi(optarg);
            break;
        case 'c':
            num_characters = atoi(optarg);
            break;
        case '?':
            fprintf(stderr, "Opci n no reconocida: -%c\n", optopt);
            exit(EXIT_FAILURE);
        default:
            fprintf(stderr, "Uso: %s [-h] [-d numero_de_archivos] [-c numero_de_caracteres]\n", argv[0])
    ;
            exit(EXIT_FAILURE);
        }
    }
```

Código 11. Función que maneja la entrada en src2.

5.1.2. src Simulador de búsqueda

El programa comienza con la inclusión de los archivos de cabeceras, `doc.h`, `graph.h`, `inverted_index.h` y `pagerank.h` que definen las estructuras y funciones necesarias para el funcionamiento del algoritmo. A continuación, el `main()` de `src` procesa los argumentos de línea de comandos, utilizando la función `getopt()` para gestionar las opciones disponibles. El programa acepta dos opciones principales:

- **-h**: Muestra un mensaje de ayuda que explica cómo usar el programa, indicando que se debe usar la opción `-s` seguida de una palabra para que se pueda buscar.
- **-s**: Permite que el usuario ingrese una palabra para buscar.

Si el usuario no proporciona los parámetros adecuados, se muestran mensajes de error, como en el caso de una opción no reconocida o un valor inválido para el número de documentos.

El flujo del programa es el siguiente:

- Primero, se inicializan variables y estructuras necesarias, como el arreglo `pagerank` para almacenar los valores de PageRank y la estructura `Graph` para representar el grafo.
- Posteriormente, se construye el grafo mediante las funciones `initialize_graph()`, `build_graph()`, `build_index()`.
- Finalmente, se calcula y muestra el PageRank y la palabra buscada utilizando `calculate_pagerank()` y `print_search_word_with_pagerank()`, para luego liberar los recursos con `release_graph()` y `release`$_i$`nverted`$_i$`ndex`().

A continuación, se muestra el fragmento relevante del código que gestiona la entrada de los parámetros:

```
    int opt;
    char *word_to_search = NULL;

    while ((opt = getopt(argc, argv, "hs:")) != -1)
    {
        switch (opt)
        {
        case 'h':
            fprintf(stdout, "\nPara ingresar la palabra a buscar, por favor coloque el parametro <-s> <numero_de_archivos>\n\n");
            break;
        case 's':
            word_to_search = optarg;
            break;
        case '?':
            fprintf(stderr, "Opcion no reconocida: -%c\n", optopt);
            exit(EXIT_FAILURE);
        default:
            fprintf(stderr, "Uso: %s [-h] [-s palabra_a_buscar]\n", argv[0]);
            exit(EXIT_FAILURE);
        }
    }
```

Código 12. Función que maneja la entrada en src.

5.2. Salida del programa

En la imagen se muestra un ejemplo de la salida generada por el programa. A continuación, se describe cada parte de esta salida:

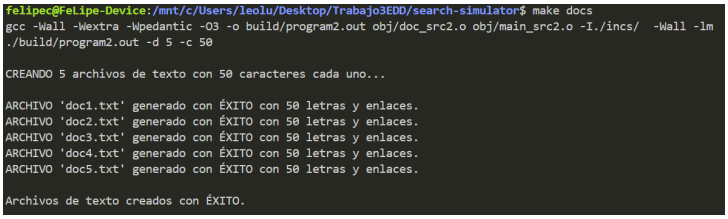

Figura 1. Salida del programa al ejecutar **make docs**.

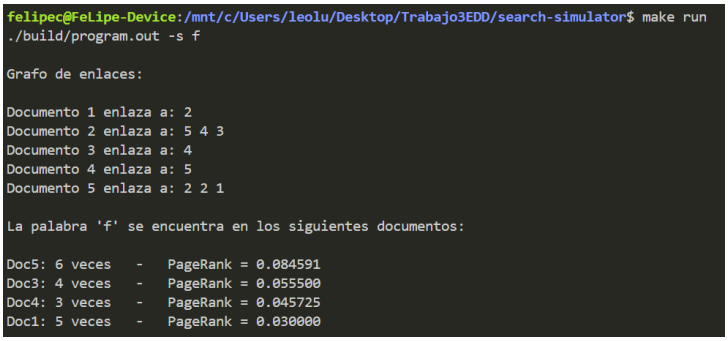

Figura 2. Salida del programa al ejecutar **make run**.

- **Generación de archivos:** El programa inicia generando el número de archivos especificado (en este caso, 3). Para cada archivo, se indica su nombre, su tamaño (en número de letras) y las conexiones generadas entre ellos.
- **Grafo de enlaces:** Se detalla cómo están conectados los documentos entre sí. Por ejemplo, el documento 1 enlaza a los documentos 2 y 3, mientras que el documento 3 enlaza al 1.
- **Índice invertido y pagerank:** Se muestra en qué documentos se encuentra una palabra específica (en este caso, la palabra `f`). Por ejemplo, esta palabra aparece una vez en los documentos `doc1.txt` y `doc3.txt`, ademas los documentos son mostrados en orden de importancia a partir del pagerank el cual tambien se despliega .
- **Archivo EPS:** Finalmente, se genera un archivo en formato `.eps` que contiene una representación gráfica de los nodos sin los enlaces.

6. Dificultades enfrentadas

6.1. Hasheo y tokenización

Una de las dificultades en el desarrollo fue implementar un sistema eficiente de búsqueda y acceso a las palabras clave en los documentos. Para reducir el tiempo de búsqueda y evitar una complejidad de $O(n)$ en cada comparación de palabras, se implemento una tabla hash que permitiera acceder a cada palabra en tiempo constante, $O(1)$, en promedio.

La mayor complicación fue el proceso de construir el hash de cada palabra, el cual implicaba una serie de pasos. Primero, se requería tokenizar el texto, es decir, dividirlo en palabras individuales. Esto era un reto en sí mismo, ya que había que ignorar las "stopwords"(palabras o letras comunes, que no aportan valor al análisis) para evitar almacenar términos irrelevantes en el hash. Además, fue necesario implementar un filtro para eliminar los símbolos y signos de puntuación que no eran útiles, dejando solo las palabras significativas para el cálculo del hash.

Estos pasos adicionales agregaron dificultad al proceso, pero fueron importantes para lograr una estructura de datos eficiente y optimizada para consultas rápidas.

6.2. Separación de archivos

Durante la organización del trabajo, se generaron dos directorios principales para el código fuente: `src` y `src2`. En el directorio `src2`, además de incluir nuevos archivos fuente para funciones adicionales, se creó un segundo archivo `main.c`, lo que causó problemas durante el proceso de compilación.

El problema radica en que, al compilar el proyecto, el compilador encontró dos puntos de entrada al programa (los archivos `main.c` en `src` y `src2`). Dado que en C solo puede existir un único punto de entrada definido por la función `main()`, se generaron errores de compilación.

La estructura del proyecto que causó este conflicto es la siguiente:

```
.
 src
    graph.c
    graphic.c
    inverted_index.c
    main.c
    pagerank.c
 src2
     doc.c
     main.c
```

El archivo `main.c` en `src2` introdujo el error al estar presente otro `main.c` en `src`. Como resultado, el compilador generó errores indicando múltiples definiciones de la función `main()`.

6.3. Generación de la Imagen EPS

Al intentar crear la imagen del grafo en formato EPS, se observó que, aunque los nodos se generaron correctamente, los enlaces entre ellos no se mostraban. No obstante, se lograron establecer las conexiones, pero el tiempo limitado para documentar y explicar los cambios en el informe, sumado a las modificaciones extensas en el código, impidió su implementación en el resultado final.

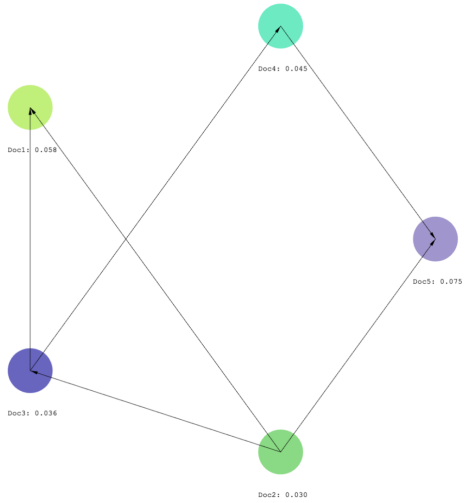

Figura 3. Representación gráfica de los grafos.

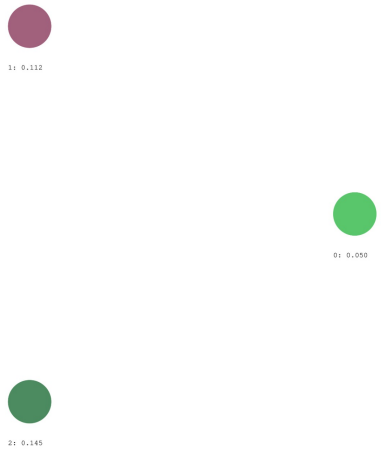

Figura 4. Representación gráfica de los grafos sin las aristas.

7. Conclusión

Se realizó con éxito un sistema de recuperación de información, implementando cada uno de los requerimientos entregados en el PDF. Al realizar esta simulación, se pudo observar la importancia de tener algoritmos eficientes y métodos de búsqueda efectivos, acercándose a $O(1)$ en el caso ideal.

Este ejercicio nos muestra la necesidad de optimizar no solo la velocidad de los algoritmos, sino también el uso de la memoria y los recursos. Nos permite comprender mejor cómo diferentes estructuras de datos pueden influir en el rendimiento de un sistema, así como los factores que se deben considerar al seleccionar o diseñar un algoritmo.

Además, esta implementación demuestra la relevancia de construir estructuras de datos adecuadas para problemas específicos, lo que nos prepara para abordar proyectos más complejos en el futuro.

Tener en cuenta aspectos como la eficiencia y la escalabilidad se convierte en una habilidad importante para desarrollar soluciones robustas y de alto rendimiento.

Referencias

[1] W. contributors. «PageRank». Último acceso: 18 de noviembre de 2024. (2024), dirección: https://en.wikipedia.org/wiki/PageRank.

[2] GeeksforGeeks. «Inverted Index». Último acceso: 18 de noviembre de 2024. (2024), dirección: https://www.geeksforgeeks.org/inverted-index/.

[3] GeeksforGeeks. «PageRank Algorithm Implementation». Último acceso: 18 de noviembre de 2024. (2024), dirección: https://www.geeksforgeeks.org/page-rank-algorithm-implementation/.

[4] Memgraph. «PageRank Algorithm». Último acceso: 18 de noviembre de 2024. (2024), dirección: https://memgraph.com/docs/advanced-algorithms/available-algorithms/pagerank.