UMAG Universidad de Magallanes | Facultad de Ingeniería

Simulador de motor de busqueda

Mario Calbuyahue[†], **Felipe Paillacar**[†] **y Constanza Araya**[†]

[†] *Universidad de Magallanes*

This manuscript was compile on Nov 19

Resumen

Este trabajo implementa un motor de búsqueda simplificado utilizando un índice invertido y el algoritmo PageRank para ordenar documentos por relevancia. Se procesaron documentos simulando enlaces entre ellos, creando un grafo dirigido y un índice eficiente para búsquedas rápidas. Los resultados muestran documentos ordenados según su importancia, calculada mediante iteraciones del PageRank. Este proyecto integra estructuras de datos y algoritmos clave en la recuperación de información

Keywords: *Simulación, motor de busqueda, estructuras de datos, planificación de procesos, gestión de memoria*

Índice

1. Introduccion

En la era de la información, los motores de búsqueda desempeñan un papel muy importante al facilitar el acceso eficiente a la informacion. Este proyecto tiene como objetivo implementar un sistema simplificado que combine un índice invertido y el algoritmo PageRank, dos herramientas muy importantes en el funcionamiento de motores de búsqueda modernos.

El índice invertido permite realizar búsquedas rápidas de palabras clave, mientras que el algoritmo PageRank utiliza un modelo de grafo dirigido para calcular la importancia relativa de cada documento en función de los enlaces entre ellos. Estos conceptos se aplican a un conjunto de documentos que simulan una red web, permitiendo su clasificación y recuperación eficiente.

2. Objetivo del Proyecto

Desarrollar un motor de búsqueda simplificado que permita recuperar información de manera eficiente a través de la implementación de un índice invertido y el algoritmo PageRank. El sistema debe procesar un conjunto de documentos, construir un grafo de enlaces entre ellos y clasificar los resultados según su relevancia para optimizar la búsqueda y ordenación de información.

3. Investigacion previa

El proyecto toma inspiración de algoritmos ampliamente usados en el ámbito de la recuperación de información, como el PageRank, desarrollado por Sergey Brin y Larry Page para Google. Este algoritmo evalúa la importancia de un documento basándose en los enlaces que recibe de otros documentos. Adicionalmente, utiliza índices invertidos, una estructura clave en sistemas de búsqueda como Elasticsearch y motores de búsqueda académicos. Para optimizar resultados, se hace uso de stopwords, una técnica que filtra palabras irrelevantes en el contexto de la búsqueda.

4. Planificacion del proyecto

4.1. Definición del alcance:

Se determinó que el sistema procesaría documentos en formato .txt, implementaría un índice invertido y calcularía PageRank.

4.2. Diseño conceptual:

Se diseñaron las estructuras de datos clave: Índice invertido para asociar palabras clave con documentos. Grafo dirigido para modelar las relaciones entre documentos.

4.3. Desarrollo técnico:

Se implementaron funciones específicas para: Leer y procesar documentos desde un directorio. Manejar la indexación y el cálculo de relevancia. Pruebas y ajustes: Se realizaron pruebas en directorios de prueba para verificar la funcionalidad, precisión y eficiencia.

4.4. Documentación:

Se generaron comentarios detallados en el código y este informe explicativo.

5. Descripción del Simulador

El proyecto consiste en el desarrollo de un motor de búsqueda simplificado que integra un índice invertido y el algoritmo PageRank para procesar y clasificar un conjunto de documentos. El sistema se compone de varios módulos que interactúan para garantizar la funcionalidad deseada:

5.1. Módulo de Índice Invertido

Este módulo utiliza una tabla hash para asociar palabras clave con los documentos donde aparecen, excluyendo palabras irrelevantes (stopwords). Permite realizar búsquedas rápidas y eficientes mediante consultas que recuperan los documentos correspondientes.

5.2. Módulo de Grafo y PageRank

Los documentos están representados como nodos de un grafo dirigido, con enlaces que simulan hipervínculos entre ellos. El algoritmo PageRank se implementa con un factor de amortiguación y múltiples iteraciones para calcular la relevancia de cada documento en función de las relaciones entre ellos.

5.3. Carga y Procesamiento de Documentos

Los documentos de texto se procesan para indexar palabras clave y detectar enlaces hacia otros documentos. Las palabras se convierten a minúsculas y se filtran según su relevancia antes de ser añadidas al índice o al grafo.

6. Partes tecnicas del proyecto

El programa está compuesto por varios módulos:

6.1. main.c:

Controla el flujo principal, incluyendo el manejo del menú.

6.2. index.c:

Gestiona la indexación de palabras y búsqueda.

6.3. graph.c:

Implementa las funciones relacionadas con el grafo y el cálculo de PageRank.

6.4. utils.c:

Proporciona utilidades como la conversión de cadenas a minúsculas.

7. Esquema del sistema

7.1. Entrada:

Directorio con archivos .txt.

7.2. Procesamiento:

Lectura y tokenización de documentos.
Creación del índice invertido.
Construcción del grafo.
Cálculo de PageRank.

7.3. Salida

Resultados de búsqueda y estadísticas.

8. Estructuras de Datos Utilizadas

8.1. Tablas Hash

Para implementar el índice invertido, las tablas hash permiten mapear palabras clave a los documentos donde aparecen. Cada entrada de la tabla contiene una lista enlazada que resuelve colisiones y almacena los documentos asociados a una palabra.

8.2. Lista enlazada simple

Las listas enlazadas simples se utilizan para manejar la lista de procesos en memoria. Estas listas permiten la flexibilidad de añadir o eliminar procesos de forma dinámica y son fundamentales para la gestión de memoria en la asignación dinámica de bloques.

8.3. Grafo Dirigido

Los documentos y sus enlaces se modelaron como un grafo dirigido, donde cada nodo es un documento y cada arista dirigida representa un enlace entre dos documentos.

8.4. Arreglos (PageRank y Estadísticas)

Se utilizaron arreglos para almacenar los valores de PageRank de cada documento y manejar listas temporales como las estadísticas del sistema.

```
void calcularPageRank(double dampingFactor, int iteraciones) {
    for (int i = 0; i < grafo.numDocs; i++) {
        grafo.pageRank[i] = 1.0 / grafo.numDocs;
    }

    for (int iter = 0; iter < iteraciones; iter++) {
        double nuevoPageRank[MAXDOCS] = {0};
        for (int i = 0; i < grafo.numDocs; i++) {
            NodoGrafo *nodo = grafo.adyacencia[i];
            while (nodo) {
```

```
                int destino = nodo->docID;
                nuevoPageRank[destino] += grafo.pageRank[i];
                nodo = nodo->siguiente;
            }
        }

        for (int i = 0; i < grafo.numDocs; i++) {
            grafo.pageRank[i] = dampingFactor * nuevoPageRank[i] + (1 - dampingFactor) / grafo.numDocs;
        }
    }
}
```

Código 1. Cálculo de PageRank usando Arreglos

```
void mostrarTopPageRank(int n) {
    struct {
        double pageRank;
        int docID;
    } documentos[MAXDOCS];


    for (int i = 0; i < grafo.numDocs; i++) {
        documentos[i].pageRank = grafo.pageRank[i];
        documentos[i].docID = i;
    }


    for (int i = 0; i < grafo.numDocs - 1; i++) {
        for (int j = 0; j < grafo.numDocs - i - 1; j++) {
            if (documentos[j].pageRank < documentos[j + 1].pageRank) {

                double tempPageRank = documentos[j].pageRank;
                int tempDocID = documentos[j].docID;
                documentos[j].pageRank = documentos[j + 1].pageRank;
                documentos[j].docID = documentos[j + 1].docID;
                documentos[j + 1].pageRank = tempPageRank;
                documentos[j + 1].docID = tempDocID;
            }
        }
    }

    /
    printf("\n--- Top %d Documentos por PageRank ---\n", n);
    for (int i = 0; i < n && i < grafo.numDocs; i++) {
        printf("Documento %d: PageRank = %.4f\n", documentos[i].docID, documentos[i].pageRank);
    }
    printf("-------------------------------------\n");
}
```

Código 2. Mostrar Estadísticas del Sistema (Uso de Arreglos para Top PageRank)

9. Asignación de Memoria para Listas Enlazadas (Índice Invertido y Grafo)

Uso: Para agregar nuevas palabras al índice o enlaces al grafo, se utiliza la función malloc para reservar memoria dinámica para nodos de tipo NodoIndice y NodoGrafo. Proceso: Cuando se crea un nuevo nodo, se llama a malloc con el tamaño requerido por la estructura (sizeof), lo que asigna espacio en el heap. Los punteros internos (siguiente) se inicializan para formar enlaces entre nodos. A continuación, se presentan fragmentos de código de cada uno de estos algoritmos:

```
void agregarPalabraIndice(const char *palabra, int docID) {
    int hash = calcularHash(palabra);
    NodoIndice *actual = tablaHash[hash];

    while (actual) {
        if (strcmp(actual->palabra, palabra) == 0) {
            actual->docIDs[actual->conteoDocs++] = docID;
            return;
        }
        actual = actual->siguiente;
    }


```

```
    NodoIndice *nuevoNodo = malloc(sizeof(NodoIndice));
    nuevoNodo->palabra = strdup(palabra);
    nuevoNodo->docIDs[0] = docID;
    nuevoNodo->conteoDocs = 1;
    nuevoNodo->siguiente = tablaHash[hash];
    tablaHash[hash] = nuevoNodo;
}
```

Código 3. Asignación de Memoria en el Índice Invertido

```
void agregarEnlace(int origen, int destino) {
    NodoGrafo *nuevo = malloc(sizeof(NodoGrafo));
    nuevo->docID = destino;
    nuevo->siguiente = grafo.adyacencia[origen];
    grafo.adyacencia[origen] = nuevo;
}
```

Código 4. Asignación de Memoria en el Grafo (Listas de Adyacencia)

10. Asignación de Memoria para Cadenas de Texto

Uso: Se utiliza la función strdup (que internamente utiliza malloc) para crear copias dinámicas de cadenas de texto, como palabras clave y nombres de documentos.

```
void agregarPalabraIndice(const char *palabra, int docID) {
    NodoIndice *nuevoNodo = malloc(sizeof(NodoIndice));
    nuevoNodo->palabra = strdup(palabra);
    nuevoNodo->docIDs[0] = docID;
    nuevoNodo->conteoDocs = 1;
    nuevoNodo->siguiente = tablaHash[calcularHash(palabra)];
    tablaHash[calcularHash(palabra)] = nuevoNodo;
}
```

Código 5. Asignación de Memoria para Cadenas de Texto

11. Manejo de Memoria para Arreglos

Uso: Los arreglos estáticos, como los de PageRank o las estadísticas, tienen su tamaño definido en tiempo de compilación mediante macros (define MAXDOCS). Esto evita la necesidad de asignar memoria dinámicamente para estos datos, simplificando el código y reduciendo la sobrecarga de memoria.

```
define MAXDOCS 100

typedef struct {
    NodoGrafo *adyacencia[MAXDOCS];
    double pageRank[MAXDOCS];
    int numDocs;
} Grafo;


void inicializarGrafo(int numDocs) {
    grafo.numDocs = numDocs;
    for (int i = 0; i < numDocs; i++) {
        grafo.adyacencia[i] = NULL;
        grafo.pageRank[i] = 0.0;
    }
}
```

Código 6. Manejo de Memoria para Arreglos

12. Liberación de Memoria

Uso: Aunque el código no incluye explícitamente una rutina para liberar memoria, es fundamental liberar los nodos creados dinámicamente usando free para evitar fugas de memoria.

```
void liberarLista(NodoGrafo *nodo) {
    while (nodo) {
```

```
        NodoGrafo *temp = nodo;
        nodo = nodo->siguiente;
        free(temp);
    }
}


void liberarGrafo() {
    for (int i = 0; i < grafo.numDocs; i++) {
        liberarLista(grafo.adyacencia[i]);
    }
}
```

Código 7. Liberación de Memoria

13. Desiciones de diseño

Uso de un Índice Invertido: Implementar un índice invertido con tablas hash para mapear palabras clave a documentos..
Representación del Grafo mediante Listas de Adyacencia: Modelar las relaciones entre documentos con un grafo dirigido utilizando listas de adyacencia..
Uso del Algoritmo PageRank Iterativo: Implementar una versión iterativa del algoritmo PageRank con un factor de amortiguación configurable.
División Modular del Sistema: Separar la funcionalidad en módulos: índice, grafo, utilidades y la interfaz principal.
Procesamiento de Documentos en Tiempo de Carga: Indexar palabras y construir el grafo de enlaces durante la carga inicial de los documentos.
Interfaz de Usuario Basada en Menú: Implementar un menú interactivo para acceder a las funciones principales del sistema.
Gestión de Memoria Dinámica y Estática: Combinar arreglos estáticos para datos predefinidos como PageRank y tablas hash con asignación dinámica para estructuras como nodos y palabras.

```
void inicializar_memoria(int *tamano_bloques, int cantidad) {
    for (int i = 0; i < cantidad; i++) {
        BloqueMemoria *nuevo_bloque = (BloqueMemoria*)malloc(sizeof(BloqueMemoria));
        nuevo_bloque->tamano = tamano_bloques[i];
        nuevo_bloque->libre = 1;
        nuevo_bloque->siguiente = memoria;
        memoria = nuevo_bloque;
    }
}
```

Código 8. Inicialización de Bloques de Memoria

Esta función configura la lista enlazada de bloques de memoria al inicio de la simulación, permitiendo que el sistema asigne y libere memoria de forma dinámica.

14. Resultados y Análisis

14.1. Indexación de Palabras

Se indexaron exitosamente palabras clave excluyendo las stopwords, alcanzando un total de [inserta número real obtenido] palabras indexadas y [inserta número] documentos procesados. Las consultas realizadas devolvieron documentos relevantes, mostrando que el índice invertido permite búsquedas rápidas y precisas. **Cálculo del PageRankt** El algoritmo PageRank asignó puntajes coherentes a los documentos, destacando aquellos con mayor cantidad de enlaces entrantes como los más relevantes.
Ordenación por Relevancia Los documentos recuperados en búsquedas se ordenaron correctamente según su relevancia, determinada por el PageRank. Los resultados destacaron documentos clave con alta interconexión dentro del grafo..
Pruebas de Escalabilidad El sistema demostró un desempeño eficiente incluso con un aumento significativo en el número de documentos procesados, gracias a las estructuras de datos seleccionadas. .
Los resultados de los algoritmos de asignación de memoria muestran diferencias en la eficiencia y fragmentación. A continuación se muestra un ejemplo de salida donde el algoritmo Best Fit asigna bloques de manera óptima en comparación con First Fit:

14.2. Resultados de Planificación de Procesos

Las búsquedas mostraron un tiempo de respuesta rápido, validando la efectividad del índice invertido. El cálculo de PageRank, aunque iterativo, presentó tiempos de ejecución razonables para el conjunto de datos utilizado.:

15. Problemas enfrentados

Dificultades iniciales con la representación eficiente de los enlaces entre documentos.
Gestión de directorios con gran cantidad de archivos.

Control de errores en el manejo de archivos inexistentes o corruptos.

16. Conclusiones

El desarrollo de este proyecto permitió implementar un motor de búsqueda simplificado que integra conceptos clave como el índice invertido y el algoritmo PageRank, aplicados a un sistema basado en estructuras de datos eficientes. A lo largo del trabajo, se abordaron tanto los desafíos teóricos como prácticos relacionados con la recuperación de información y la gestión de memoria, logrando un sistema funcional que ofrece resultados relevantes y ordenados según la importancia de los documentos procesados. La utilización de tablas hash para el índice invertido, grafos dirigidos para la representación de enlaces entre documentos, y arreglos para el cálculo iterativo del PageRank demostró ser una combinación eficaz y escalable. Además, la modularidad del diseño facilitó la implementación, permitiendo incorporar nuevas funcionalidades en el futuro. Los resultados obtenidos validan la capacidad del sistema para ordenar y recuperar información de manera eficiente, mostrando la importancia de combinar algoritmos de clasificación y estructuras dinámicas de datos. Este proyecto no solo ilustra la aplicación práctica de algoritmos fundamentales en la computación, sino que también sienta una base sólida para el desarrollo de motores de búsqueda más avanzados. En resumen, el trabajo realizado evidencia cómo conceptos teóricos pueden transformarse en soluciones prácticas y destaca la relevancia de una planificación cuidadosa y un diseño bien estructurado en la implementación de sistemas complejos.