EXPLOTACIÓN DE LA INFORMACIÓN

Práctica 3 - Buscador

Raúl Beltrán Marco
23900664F

ÍNDICE

Presentación del problema

En esta práctica se nos han pedido crear un **Buscador**, con ayuda del Indexador y Tokenizador realizado en las anteriores prácticas.

El programa deberá ser capaz de analizar la pregunta que haga el usuario y obtener un **valor de similitud** (vSimilitud) el cual nos indicará para cada documento indexado el parecido que tiene con la pregunta realizada; cuanto mayor sea este valor mayor será la relación entre el documento y la pregunta del usuario.

Además, el programa deberá ser capaz de dicho valor mediante dos modelos: **DFR** (Deviation From Randomness) y **Okapi BM-25.**

Finalmente, el programa devolverá una **lista ordenada de forma descendente** donde el primer documento será el **más relevante** o que tiene **mayor similitud** con la pregunta realizada y previamente indexada con el Indexador.

Hipótesis de partida.

Antes de empezar a programar lo primero que hice fue analizar minuciosamente las fórmulas con las que se obtiene el vSimilitud que luego se usa para crear el objeto ResultadoRI (DFR y Okapi BM-25) ya que primero quería hacerme una idea de como podía obtener los valores necesarios para dichas fórmulas.

También supuse que tendría que crear funciones adicionales en la clase Indexador para poder manipular todos los datos que me fueran necesarios, como por ejemplo **indiceDocs** o **informacionColeccionDocs**.

Finalmente, supuse que tendría que realizar dos bucles, uno para recorrer todos los documentos indexados y otro bucle anidado con el objetivo de que para cada documento indexado (d) obtener un **vSimilitud** mediante dicho bucle el cual recorre todos los términos almacenados en la variable **indicePregunta**.

Análisis de la solución implementada

Para una sola pregunta:

Todo el programa gira entorno a la función **GestionarBusqueda**, como se puede ver mi solución **no utiliza una priority_queue** como originalmente aparece en el enunciado, sino que usa un **vector<ResultadoRI>,** estuve buscando información sobre estructuras que fueran más rápidas que priority_queue la cual ya era rápida de por sí, pero finalmente opté por usar un vector debido a que es una estructura más versátil a la hora de programar.

Considero que es una solución bastante simple ya que únicamente se compone de un bucle for en el que realiza una llamada u otra dependiendo del valor de **formSimilitud** además de **descartar** el último elemento del vector cuando el tamaño de este sea mayor a numDocumentos finalizaremos el bucle y utilizaremos la función de std::insert para añadir de forma ordenada las distintas preguntas que se vayan realizando.

También mencionar que para poder utilizar **un vector como una priority_queue** he tenido que utilizar una **función lambda** dentro de la función **std::find_if()** con la cual es posible obtener la posición que le corresponde el elemento a insertar y a su vez mantener el orden descendente que se exige en el enunciado.

```bool
Buscador::GestionarBusqueda(const int& numDocumentos, const int& numPregunta)
{
    string preg;
    vector<ResultadoRI> docs;
    double vSimilitud= 0.0;
    if(DevuelvePregunta(preg))
    {
        InfDoc infDoc;
        for(auto& doc : DevolverDocsIndexados()) // Para cada documento de la colección
        {
            if(this->formSimilitud == 0)
            {
                vSimilitud= DFR(doc.second);
                if(vSimilitud != 0.0)
                {
                    ResultadoRI value(vSimilitud, doc.second.getId(), numPregunta);
                    docs.insert(std::find_if(docs.begin(), docs.end(), [value](ResultadoRI s) { return s.vSimilitud() < value.vSimilitud(); }), value);
                }
            }
            else
            {
                vSimilitud= BM25(doc.second);
                if(vSimilitud != 0.0)
                {
                    ResultadoRI value(vSimilitud, doc.second.getId(), numPregunta);
                    docs.insert(std::find_if(docs.begin(), docs.end(), [value](ResultadoRI s) { return s.vSimilitud() < value.vSimilitud(); }), value);
                }
            }
            if(docs.size() >= numDocumentos)
            {
                if(!docs.empty())
                    this->docsOrdenados.insert(docsOrdenados.end(), docs.begin(), docs.end());
                return true;
            }
        }
        if(!docs.empty())
            this->docsOrdenados.insert(docsOrdenados.end(), docs.begin(), docs.end());
        return true;
    }
    return false;
}```

Para terminar este análisis simplemente echar un vistazo a las funciones DFR y BM25 las cuales se encargan de obtener el **vSimilitud** de un documento mediante su **infDoc** y recorriendo el **indicePregunta** como previamente explicábamos en la hipótesis de partida, consiguiendo ignorar los cálculos que no nos interesa calculando previamente **ft** y en caso de valer **0** no realizará ningún cálculo mejorando el tiempo de ejecución.

```double
Buscador::DFR(const InfDoc& infDoc) const
{
    InformacionTermino infTerm;
    InformacionPregunta infPreg;
    DevuelvePregunta(infPreg);
    double vSimilitud, wiq, ft;
    vSimilitud= wiq= ft= 0.0;
    double ld= (double) infDoc.getNumPalSinParada();
    double avg= ld / (double)DevuelveInfColeccion().getNumDocs();
    for(auto& termino : DevolverIndicePregunta()) // Para cada termino de la pregunta
        if(Devuelve(termino.first, infTerm))
        {
            ft= (double)infTerm.getDocs()[infDoc.getId()].getFt();
            if(ft != 0)
            {
                wiq= (double)termino.second.getFt() / (double)infPreg.getNumTotalPal();
                vSimilitud+= wiq * WeightDoc(infTerm, infDoc, ft, ld, avg);
            }
        }
    return vSimilitud;
}```

```double
Buscador::BM25(const InfDoc& infDoc) const
{
    InformacionTermino infTerm;
    double vSimilitud, ft;
    vSimilitud = ft = 0.0;
    for(auto& termino : DevolverIndicePregunta()) // Para cada termino de la pregunta
    {
        if(Devuelve(termino.first, infTerm))
        {
            ft= (double)infTerm.getDocs()[infDoc.getId()].getFt();
            if(ft != 0)
                vSimilitud+= IDF(infTerm) * FrecuencyDoc(infTerm, infDoc, ft);
        }
    }
    return vSimilitud;
}```
```

Para varias preguntas:

En esta parte lo que he propuesto es básicamente realizar un bucle entre numPregInicio y numPregFin con el objetivo de que en cada iteración del bucle se abra el archivo que contiene dicha pregunta, obtener el string correspondiente a la pregunta, indexarla y seguidamente llamar a GestionarBusqueda pasándole el valor de i para que a la hora de guardarlo en el vector<ResultadoRI> docsOrdenados saber a que pregunta hace referencia el resultado obtenido.

He decido realizar este bucle ya que creo que es la única forma en la que se puede obtener la pregunta que está almacenada en el documento y a la vez respetar su número de pregunta.

Básicamente por cada pregunta se abrirá su correspondiente fichero en caso de que el directorio exista y se procederá a indexar la pregunta correspondiente y llamar a GestionaBusqueda donde se obtendrá los valores de similitud correspondientes.

```
bool
Buscador::Buscar(const string& dirPreguntas, const int& numDocumentos, const int& numPregInicio, const int& numPregFin)
{
    struct stat dir;
    int err= stat(dirPreguntas.c_str(), &dir);
    // Comprobamos la existencia del directorio
    if(err == -1 || !S_ISDIR(dir.st_mode)) // No existe dirPreguntas
        return false;
    else
    {
        ifstream fPregunta;
        string ruta;
        for(int i=numPregInicio; i <= numPregFin; i++)
        {
            ruta= dirPreguntas.c_str() + to_string(i) + ".txt";
            fPregunta.open(ruta.c_str());
            if(fPregunta)
            {
                string preg;
                getline(fPregunta, preg);
                fPregunta.close();

                IndexarPregunta(preg);

                if(!GestionarBusqueda(numDocumentos, i)) // TODO mejorar
                    return false;
            }
        }
        return true;
    }
}
```

Justificación de la solución elegida

Finalmente he elegido la solución explicada anteriormente porque es la solución a la que he llegado después de varios intentos y es la que mejores resultados me ha dado a la hora de realizar la prueba del corpus.

Estas son algunas de las soluciones que he descartado:

El mismo código, pero con priority_queue, simplemente era más lento porque en la función de GestionarBusqueda me veía forzado a realizar un segundo bucle para quedarme con los numDocumentos mejores ya que priority_queue no tiene la función pop_back() y de cara a buscar varias preguntas tardaba prácticamente el doble.

```
    if(!aux.empty())
    {
        int i=0;
        while(i < numDocumentos) //Nos quedamos con los mejores
        {
            if(aux.empty())
                break;
            this->docsOrdenados.push(aux.top());
            aux.pop();
            i++;
        }
    }
    return true;
}
return false;
```

También intenté ignorar los documentos que no tenían términos de la pregunta indexada en ese momento con una función que llamé Contiene() pero debido a esta comprobación efectivamente hacía más lento el código.

```
bool Contiene(const long int& idDoc) const
{
    if(!pregunta.empty())
    {
        stemmerPorter stem;
        string aux;
        for(auto& termino : indicePregunta)
        {
            aux= termino.first;
            if(this->DevolverPasarAminuscSinAcentos())
                this->tok.Filtrar(aux);
            stem.stemmer(aux, this->tipoStemmer);
            if(indice.find(aux) != indice.end())
                if(indice.at(aux).getDocs().find(idDoc) != indice.at(aux).getDocs().end())//Encontrado doc
                    return true;
        }
        return false;
    }
    return false;
}
```

Lo mismo pasó con una idea que tampoco llegó a funcionar la cual era que en el IndexarPregunta guardarse en qué documentos en los que aparecieran los términos de la pregunta y guardar esa lista de documentos en la variable infPregunta, pero esta implementación además de costosa daba problemas como double_free() y problemas con la memoria así que al final la descarté.

Análisis de las mejoras realizadas en la práctica

La principal mejora que he realizado ha sido el cambio de priority_queue a un vector ordenado descendentemente lo cual permitió reducir el tiempo de ejecución ya que iterar sobre un único vector es más sencillo y computacionalmente mejor que priority_queue el cual solo puede obtener el primer elemento y sus funciones son limitadas.

Con este cambio conseguimos reducir el tiempo tanto a la hora de imprimir la información como a la hora de obtener y generar la información para crear los ResultadoRI.

Otra mejora que ayudó ligeramente a mejorar el tiempo de ejecución fue utilizar la variable ft para ignorar los cálculos innecesarios a la hora de obtener vSimilitud y así evitar llamadas innecesarias a las funciones auxiliares.

De la misma forma pero con la variable **vSimilitud** podemos evitar insertar en el **vector<ResultadoRI>** documentos que tengan una relevancia nula en el relación a la pregunta realizada por el usuario logrando así evitar llamar tanto a la función **find_if y el insert** de vector además de la llamada auxiliar a las funciones de **DFR()** o **BM25().**

Análisis de complejidad del buscador

Complejidad temporal (Una sola pregunta)

La complejidad temporal en caso de una sola pregunta corresponde con la complejidad de la función GestionarBusqueda().

El **peor de los casos** viene dado cuando numDocumentos es mayor que el número de documentos indexados, lo que provoca que se tenga que analizar todos los documentos, además que todos los documentos obtengan un vSimilitud distinto de 0 y que todos los términos de la pregunta estén en todos los documentos que se están iterando.

Para obtener su coste temporal lo dividiremos por partes:

- **O(n)** siendo **n** el número de documentos de indiceDocs de Indexador.
- **O(DFR)** coste de la función DFR o **O(BM25)** coste de la función BM25, ambas funciones equivalen al coste de un bucle for que sería de coste **O(g*c)** siendo **g** el número de términos de pregunta que no son de parada y **c** el coste de las funciones auxiliares junto al coste de la función Devuelve().
- **O(k)** siendo **k** el coste total de insertar un ResultadoRI en el vector respetando el orden descendente.
- **O(i)** siendo **i** el coste de insertar el vector generado por el bucle en el vector docsOrdenados.

Con todo esto podemos concluir que el coste temporal en el peor de los casos es:

O(n*[DFR+k] + i) y **O(n*[BM25+k] + i)**

El **mejor de los casos** lo obtenemos cuando se intenta llamar a la función Buscar y no hay ninguna pregunta indexada, en cuyo caso el único coste del programa en ese caso sería el coste de lo que tarde la función de **DevolverPregunta()** en devolver **false**, siendo su coste **w**, **Ω(w)**.

Complejidad temporal (Todas las preguntas del CORPUS)

Para una gran cantidad de preguntas el **peor de los casos** viene dado por las mismas condiciones que antes sumado a una amplia cantidad de preguntas.

Con el razonamiento que hemos usado antes ya que reutilizamos la función GestionarBusqueda() podemos obtener el coste temporal:

O(d + m*[f+v+gb]) → O(d + m*[f+v+(n*[DFR+k] + i)])

Siendo **d**, el coste de encontrar el directorio donde están las preguntas.

Siendo **m**, el número de preguntas realizadas por el usuario.

Siendo **f**, el coste de abrir un fichero leer la primera línea y cerrarlo.

Siendo **gb**, el coste de la función GestionarBusqueda().

El **mejor de los casos** viene dado en el caso de que el directorio donde se alojan todas las preguntas no se ha encontrado, en ese caso todo el coste de la función Buscar() que se encarga de gestionar varias las preguntas se reduce en **Ω(d)**.

Análisis de precisión y cobertura

Dada la siguiente tabla y gráfica obtenida ejecutando el Buscador implementado de 4 formas distintas.

Hemos ejecutado la función **Buscar()** con sus dos opciones **DFR** y Okapi **BM25**, además de indexar una gran colección de documentos **CON** y **SIN** stemming (proceso por el cual se consigue simplificar la palabra; por ejemplo, el verbo **was/were** de ingles con stemming pasaría a ser **be**)

Obteniendo la siguiente tabla gracias a la herramienta **trec_eval** ofrecida por el enunciado.

	SIN STEMMING		CON STEMMING	
Recall	DFR	BM25	DFR	BM25
0,0	0,6395	0,7094	0,5699	0,7089
0,1	0,6341	0,7094	0,5664	0,7089
0,2	0,6090	0,7051	0,5462	0,7069
0,3	0,5704	0,6763	0,5136	0,6822
0,4	0,5442	0,6469	0,4827	0,6557
0,5	0,5126	0,6275	0,4654	0,6452
0,6	0,3916	0,5573	0,3706	0,5519
0,7	0,3516	0,5303	0,3206	0,5292
0,8	0,3274	0,5185	0,2938	0,5173
0,9	0,2840	0,4429	0,2471	0,4530
1,0	0,2778	0,4385	0,2414	0,4453

Como podemos observar en la gráfica resultante todos los modelos con o sin stemming siguen la tendencia de a cuanto mayor tienden a empeorar a medida que va aumentando Recall provocando una peor precisión como resultado.

Es necesario comparar ambos modelos y ver como Okapi BM-25 es mejor en todos los sentidos a DFR.

Respecto **DFR** podemos observar que su máxima precisión posible es cuando el Recall es 0 además de que se obtiene cuando NO se aplica stemming, como se puede ver en el gráfico aplicar **stemming** y usar **DFR no son compatibles** ya que siempre es peor a usar DFR sin stemming.

Por último, he de destacar que BM25 es la mejor opción posible a la hora de implementar nuestro Buscador, ya que nos ofrece un mucho mejor resultado que DFR además de que aplicar el stemming beneficia a este método, pero tampoco marca una gran diferencia.

Así que la mejor opción finalmente la mejor opción más eficiente es utilizar Okapi BM-25 sin la opción de stemming, o por lo menos el Stemmer de Porter.