Práctica 3: Buscador

Antonio Martínez Santa
Universidad de Alicante
77599131Y

Índice

1. Introducción

El objetivo de esta práctica es implementar un buscador capaz buscar la información más relevante. Como muchos ya sabemos un buscador es la interfaz fundamental a la hora de recuperar información más usada del mundo ya que con una simple búsqueda en tiempos completamente ridículos buscadores como el de Google nos ofrecen una inmensa cantidad de información. Para ello haré uso del tokenizador y del indexador previamente implementados en las anteriores prácticas.

1.1 Presentación del problema

El problema que se nos plantea es que seamos capaces de recuperar información de documentos ya indexados para así dar lugar a los documentos más relevantes para una query (pregunta) en específico. Como ya he mencionado se hará uso de las 2 anteriores prácticas para cumplir con nuestro objetivo.

1.2 Hipótesis de partida

La hipótesis de partida que planteé fue resolverlo con una clase auxiliar que me permitiera organizar cada uno de los documentos por relevancia que sería la clase ResultadoRI, esta sería la clase que me permitiría este cometido con una implementación base de constructor y operator> (implementé operator< pero no servía por las funciones de la cola de prioridad) y la clase del Buscador, esta sería la encargada de aplicar cada uno de los dos métodos de búsqueda que han sido establecidos en la práctica (DFR y BM25).

2. Análisis de la solución implementada

Lo que he hecho con el buscador ha sido para ahorrar tiempo y tener que añadir más funciones que accedan a atributos de la clase IndexadorHash ha sido hacer una extensión (herencia) de la clase IndexadorHash (para inicialización de atributos) y además añadirlas como friend classes (esto para acceso a atributos privados). Además se usa la clase auxiliar ResultadoRI que nos permite clasificar los resultados de una búsqueda por relevancia de documentos.

```
class ResultadoRI {
    friend ostream& operator<<(ostream& os, const ResultadoRI& res);

public:
    ResultadoRI();
    // Constructor: recibe la similitud, ID de documento y número de pregunta
    ResultadoRI(const double& kvSimilitud, const long int& kidDoc, const int& np);

    // Devuelve el valor de similitud del documento con la pregunta
    double VSimilitud() const;

    // Devuelve el identificador del documento
    long int IdDoc() const;

    // Devuelve el número de la pregunta asociada al resultado
    int NumPregunta() const;

    // Operador de comparación para ordenación en priority_queue
    // Ordena de forma decreciente según la similitud (si son de la misma pregunta)
    // o por número de pregunta creciente (para mantener agrupación)
    bool operator<(const ResultadoRI& rhs) const;
    bool operator>(const ResultadoRI& rhs) const;

private:
    double vSimilitud;  // Valor de similitud
    long int idDoc;     // Identificador del documento
    int numPregunta;    // Número de la pregunta asociada
};
```

Esta es la cabecera de implementación de cada uno de los métodos que tiene esta clase, no creo que sea necesario poner la implementación para reducir el contenido de la memoria ya que son getters, operator (para comparación entre valores de similitud en la priority_queue con cada documento) y los constructores. Vamos a pasar con la clase Buscador.

```
class Buscador : public IndexadorHash {
    friend ostream& operator<<(ostream& os, const Buscador& b);
    friend class IndexadorHash;
    public:
        // Constructor para inicializar Buscador a partir de la indexación generada previamente
        // Se inicializa con el valor de similitud f y los valores por defecto: c = 2; k1 = 1.2; b = 0.75
        Buscador(const string& directorioIndexacion, const int& f);
        // Constructor de copia
        Buscador(const Buscador&);
        // Destructor
        ~Buscador();
        // Operador de asignación
        Buscador& operator=(const Buscador&);
        // Realiza la búsqueda sobre la pregunta actual indexada
        // Guarda los resultados más relevantes (hasta numDocumentos) en docsOrdenados
        bool Buscar(const int& numDocumentos = 99999);
        vector<ResultadoRI> calculoDFR(const int& numPregunta);
        vector<ResultadoRI> calculoBM25(const int& numPregunta);
        bool Buscar(const int& numDocumentos, const int& numPregunta);
        // Realiza la búsqueda sobre un conjunto de preguntas de un directorio
        // Se guarda el top numDocumentos por cada pregunta, entre numPregInicio y numPregFin
        bool Buscar(const string& dirPreguntas, const int& numDocumentos, const int& numPregInicio, const int& numPregFin);
        string RecuperarNombreDocumento(const int& idDoc) const;
        // Imprime por pantalla los resultados de la última búsqueda
        // Mostrará como máximo numDocumentos documentos por pregunta
        void ImprimirResultadoBusqueda(const int& numDocumentos) const;
        // Igual que la anterior, pero guarda la salida en un fichero
        // Devuelve false si no consigue crear el archivo
        bool ImprimirResultadoBusqueda(const int& numDocumentos, const string& nombreFichero) const;
        // Devuelve la fórmula de similitud actual (0 = DFR, 1 = BM25)
        int DevolverFormulaSimilitud() const;
        // Cambia la fórmula de similitud si el valor es válido (0 o 1)
        // Devuelve false si el valor no es válido
        bool CambiarFormulaSimilitud(const int& f);
        // Cambia el valor de la constante c del modelo DFR
        void CambiarParametrosDFR(const double& kc);
        // Devuelve el valor de la constante c del modelo DFR
        double DevolverParametrosDFR() const;
        // Cambia los valores de las constantes k1 y b del modelo BM25
        void CambiarParametrosBM25(const double& kk1, const double& kb);
        // Devuelve los valores de las constantes k1 y b del modelo BM25
        void DevolverParametrosBM25(double& kk1, double& kb) const;
        void DevolverTerminosEnPregunta();
```

Esta clase aparte de tener getters y setters tiene los métodos que nos permiten hacer la búsqueda. Los métodos más importantes son Buscar, calculoDFR y calculoBM25.
Además los atributos privados que tenemos son un vector de documentos ordenados por relevancia según la query, un valor entero que nos indica si queremos hacer búsqueda DFR o BM25 y los valores constantes de cada uno de los modelos (c (DFR) y k1, b (BM25)). La

implementación del método Buscar es la siguiente:

```
bool Buscador::Buscar(const int& numDocumentos, const int& numPregunta){
    string pregunta;
    if(!DevuelvePregunta(pregunta)) return false;

    vector<ResultadoRI> resultados;

    if(formSimilitud == 0)
        resultados = calculoDFR(numPregunta);       // devuelve un vector con los calculos
    else if(formSimilitud == 1)
        resultados = calculoBM25(numPregunta);
    else
        return false;

    sort(resultados.begin(), resultados.end(), std::greater<ResultadoRI>());
    resultados.resize(std::min(static_cast<size_t>(numDocumentos), resultados.size()));

    for(const auto& res : resultados){
        docsOrdenados.push(res);
    }

    return true;
}
```

Este método como podemos ver es el núcleo del buscador donde aquí podemos elegir entre qué método queremos utilizar para buscar y nos devuelve el propio método de la búsqueda un vector con los resultados de relevancia de cada uno. Los métodos de cálculo de cada uno de los modelos de búsqueda son los siguientes:

```cpp
vector<ResultadoRI> Buscador::calculoDFR(const int& numPregunta){
    vector<ResultadoRI> resultados;
    const auto& infCol = informacionColeccionDocs;

    double avgdl = static_cast<double>(infCol.getNumTotalPalSinParada()) / infCol.getNumDocs();
    int N = infCol.getNumDocs();
    int k = infPregunta.getNumTotalPalSinParada();

    for(const auto& [ruta, doc] : indiceDocs){
        double similitud = 0.0;
        int ld = doc.getNumPalSinParada();

        for(const auto& [termino, infTermPreg] : indicePregunta){
            auto itTerm = indice.find(termino);
            if (itTerm != indice.end()) {
                const auto& termDocs = itTerm->second.getLDocs();
                int nt = termDocs.size();
                int ft = itTerm->second.getFT();

                double lambda = static_cast<double>(ft) / N;
                double logLambda = log2(1.0 + lambda);
                double ratioLambda = log2((1.0 + lambda) / lambda);

                double wtq = static_cast<double>(infTermPreg.getFT()) / k;

                auto itDocTerm = termDocs.find(doc.getIdDoc());
                if(itDocTerm != termDocs.end()){
                    int ftd_doc = itDocTerm->second.getFT();

                    double ftd_star = ftd_doc * log2(1.0 + (c * avgdl / static_cast<double>(ld)));

                    double wtd = (logLambda + ftd_star * ratioLambda) *
                                 ((static_cast<double>(ft) + 1.0) / (static_cast<double>(nt) * (ftd_star + 1.0)));

                    similitud += wtq * wtd;
                }
            }
        }

        if(similitud > 0.0){
            similitud *= 2;     // no sé porque sale la mitad de similitud
            resultados.emplace_back(similitud, doc.getIdDoc(), numPregunta);
        }
    }

    return resultados;
}
```

Este método calcula la similitud entre una query y cada documento que hay indexado en el índice de documentos usando DFR (Deviation From Randomness). Todos los cálculos se ven reflejados en esta fórmula:

Modelo de similitud *Deviation From Randomness (DFR)*

$$sim(q,d) = \sum_{i=1}^{k} w_{i,q} * w_{i,d}$$

$$w_{t,d} = \left(\log_2(1+\lambda_t) + f^*_{t,d} * \log_2 \frac{1+\lambda_t}{\lambda_t} \right) * \frac{f_t + 1}{n_t * \left(f^*_{t,d} + 1 \right)}$$

$$w_{t,q} = \frac{f_{t,q}}{k}$$

$$f^*_{t,d} = f_{t,d} * \log_2 \left(1 + \frac{c * avr\_l_d}{l_d} \right)$$

$$f_t = \sum_{i=1}^{N} f_{t,i} \qquad \lambda_t = \frac{f_t}{N}$$

Ese código está adaptado a la estructura de datos que está siendo utilizada en el indexador, para mí una forma más fácil de recorrerlas por eso utilizo los for-each. Los nombres de las variables son los correspondientes a cada valor de la fórmula para dejar claro que se está calculando en cada momento. Un caso particular es que no se por qué pero en DFR me salía siempre de valor de similitud la mitad de lo que daba en los tests que se nos proporcionaron por eso el similitud *= 2; con eso todos los tests pasaban, sería algún error de cálculo mío, con ese apaño funciona. El método que hace el cálculo de relevancia usando Okapi BM25 es el siguiente:

```
vector<ResultadoRI> Buscador::calculoBM25(const int& numPregunta) {
    vector<ResultadoRI> resultados;

    double avgdl = static_cast<double>(informacionColeccionDocs.getNumTotalPalSinParada()) / informacionColeccionDocs.getNumDocs();
    int N = informacionColeccionDocs.getNumDocs();

    for (const auto& [ruta, doc] : indiceDocs) {
        double similitud = 0.0;

        for (const auto& [termino, infTermPreg] : indicePregunta) {
            auto itTerm = indice.find(termino);
            if (itTerm != indice.end()) {
                const auto& termDocs = itTerm->second.getLDocs();
                auto itDocTerm = termDocs.find(doc.getIdDoc());

                if (itDocTerm != termDocs.end()) {
                    double ftd = static_cast<double>(itDocTerm->second.getFT());
                    double df = static_cast<double>(termDocs.size());

                    if (ftd > 0) {
                        double idf = log2((N - df + 0.5) / (df + 0.5));
                        double K = ftd + k1 * (1.0 - b + b * (static_cast<double>(doc.getNumPalSinParada()) / avgdl));
                        double score = idf * ((ftd * (k1 + 1.0)) / K);
                        similitud += score;
                    }
                }
            }
        }

        resultados.emplace_back(similitud, doc.getIdDoc(), numPregunta);
    }

    return resultados;
}
```

Este modelo favorece los términos que son frecuentes en un documento pero penaliza los que son comunes a toda la colección. Es más robusto que DFR porque al introducir IDF los términos que aparecen en casi todos los documentos tienen poco peso. El uso de cada una de las variables viene explicado por estas fórmulas:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})}$$

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

El resto de métodos implementados son un poco redundantes, en la propia revisión manual del profesor se puede comprobar que no son demasiado complicados de entender.

3. Justificación de la solución elegida

Se escogió esta solución porque era la más práctica ya que se ha estado hablando en clase de teoría sobre estos métodos de búsqueda y como también se habla en el enunciado de la práctica es lo más práctico escoger la clase auxiliar ResultadoRI para almacenar y

comparar los resultados de relevancia de los documentos y el uso de la cola de prioridad para ordenarlos.

4. Análisis de las mejoras implementadas

Como en esta parte he ido bastante corto de tiempo no he podido probar demasiadas cosas como por ejemplo el ejercicio opcional de teoría de la implementación de los cambios en nuestro propio buscador ya que por fin pude comprobar que funcionaba correctamente. Las mejoras implementadas han sido mejorar progresivamente el funcionamiento del buscador ya que tuve bastantes problemas con DFR para implementar las fórmulas correspondientes.

5. Análisis de la eficiencia computacional

Para hacer el análisis de la complejidad computacional he ejecutado el main que hay en el enunciado de la práctica tanto para DFR como para BM25, estos han sido los resultados obtenidos:

	Búsqueda completa
Método DFR	0.247528s
Método BM25	0.221187s

Estos resultados han sido obtenidos con los prints completos de ImprimirResultadoBusqueda por lo que aparece en cada ejecución los documentos más relevantes ordenados, por ejemplo con BM25 para la query *"AGREEMENT BETWEEN SYRIA AND IRAQ ON FULL ECONOMIC UNITY AND CLOSER ECONOMIC COOPERATION ."* (**Pregunta 66**) el documento más relevante ha sido el 210.

```
65 BM25 337 422 -13.822380 ConjuntoDePreguntas
66 BM25 210 0 5.126061 ConjuntoDePreguntas
66 BM25 115 1 4.906907 ConjuntoDePreguntas
```

6. Análisis de precisión y cobertura

Para hacer el análisis de la precisión y cobertura hice uso de la herramienta que se nos dió del 'trec_eval' y estos fueron los resultados obtenidos:

```
Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    35038
    Relevant:       324
    Rel_ret:        320
Interpolated Recall - Precision Averages:
    at 0.00       0.5426
    at 0.10       0.5414
    at 0.20       0.5332
    at 0.30       0.5219
    at 0.40       0.4909
    at 0.50       0.4885
    at 0.60       0.4053
    at 0.70       0.3663
    at 0.80       0.3462
    at 0.90       0.2959
    at 1.00       0.2909
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.4254
Precision:
  At    5 docs:   0.2723
  At   10 docs:   0.2084
  At   15 docs:   0.1647
  At   20 docs:   0.1301
  At   30 docs:   0.0936
  At  100 docs:   0.0331
  At  200 docs:   0.0178
  At  500 docs:   0.0077
  At 1000 docs:   0.0039
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.3535
```

Esto fue para DFR con stem y sin stem, sospecho que ambos valores me dan igual porque el stem no se está aplicando correctamente no sé por qué porque en la anterior práctica si me pasaba los tests en los que se aplicaba, me sucede lo mismo en BM25.
Y este otro resultado fue obtenido para BM25 con stem y sin stem:

```
Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    35109
    Relevant:       324
    Rel_ret:        321
Interpolated Recall - Precision Averages:
    at 0.00       0.5972
    at 0.10       0.5926
    at 0.20       0.5802
    at 0.30       0.5592
    at 0.40       0.5509
    at 0.50       0.5371
    at 0.60       0.4349
    at 0.70       0.4113
    at 0.80       0.3890
    at 0.90       0.3261
    at 1.00       0.3205
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.4679
Precision:
  At    5 docs:   0.3253
  At   10 docs:   0.2325
  At   15 docs:   0.1807
  At   20 docs:   0.1434
  At   30 docs:   0.1000
  At  100 docs:   0.0336
  At  200 docs:   0.0176
  At  500 docs:   0.0077
  At 1000 docs:   0.0039
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.3914
```

No he podido generar la gráfica porque en el libreoffice no me dejaba abrir el gráfico adjunto al word que se nos proporcionó y se me hizo imposible.