INFORME ESCRITO

Ernesto Barcena Trujillo C111

Moogle! Es una aplicación cuyo propósito es el de encontrar coincidencias a una consulta introducida por el usuario en un universo de documentos formato “.txt”. A continuación se intentará explicar de manera ordenada el funcionamiento de la lógica de dicho proyecto.

Moogle! Posee varias clases agrupadas dentro de la carpeta MoogleEngine encargadas de la implementación lógica en la aplicación, las cuales analizaremos en este informe. El funcionamiento de la aplicación se divide en dos momentos esenciales: un primer momento que denominaremos “preprocesado”, durante el cual se recopila y se almacena una cantidad considerable de información que servirá para la posterior búsqueda antes de que termine de levantar el servidor de la aplicación; y un segundo momento en el punto en el cual el usuario introduce una consulta y se buscan los resultados que más se ajusten a dicha consulta (este proceso se repite para cada consulta independientemente de su contenido, pues no se cuenta con ningún mecanismo de caché).

A continuación procederemos a detallar el funcionamiento de la aplicación durante el “preprocesado”:

En primer lugar, hemos de mencionar que todos los datos recopilados en este momento serán almacenados en un objeto de tipo Universe, por lo cual analizaremos la estructura de dicha clase y su funcionamiento interno.

La clase Universe: consta de seis campos de clases, los cuales se expondrán a continuación con sus respectivas explicaciones:
-MainDirection es un string que contiene la dirección donde se encuentran los archivos de texto a buscar. Está configurada por defecto a la carpeta Content ubicada dentro de la carpeta principal Moogle Main acorde a la orientación del proyecto.

-directions es un array de string que contiene precisamente cada una de las direcciones de los documentos dentro de la carpeta Content. Dichas direcciones se obtienen por medio del método GetFiles de la clase Directory.

-names es un array de string que contiene los nombres de los archivos almacenados en la carpeta Content respetando el orden en que aparecen las direcciones de dichos archivos en directions. Los nombres almacenados en names son obtenidos mediante el método Namess, el cual no hace más que recorrer directions mediante un ciclo for y cada una de las direcciones las modifica para obtener el substring correspondiente al nombre (esto lo hace utilizando el método Substring de la clase String).

-contenido es un array de string que contiene los contenidos “normalizados” de los documentos almacenados en Content respetando el orden en el cual aparecen sus direcciones y nombres en los campos anteriores mediante el método contenidos. El método contenidos recibe el array directions y procede a almacenar el contenido de cada archivo mediante el método ReadAllText de la clase File, pero antes de almacenarlos los “normaliza” mediante el método estático Normalizer, (lo cual significa que se encarga de convertirlo todo a minúsculas mediante el método ToLower de la clase

String, a sustituir todas las vocales con tildes por sus respectivas vocales sin tildes y finalmente a eliminar todo carácter que no sea una letra o un número por un string vacío. Estas dos últimas operaciones las realiza empleando el método Replace de la clase Regex).

-idfs es un Diccionario de <string, float> en el cual a cada palabra existente en el universo de palabras se le asocia un valor real positivo que representa su peso con respecto al universo de documentos “idf” (en inglés Inverse Document Frequency). El método encargado de realizar dicha labor es el método getIdfs, el cual recibe el array contenido como parámetro y cuyo funcionamiento explicaremos en breve:

*getIdfs crea dos Diccionarios, uno llamado MyIdfs de <string, float> y otro auxiliar (llamado Aux) de <string, int> para apoyar en el llenado de MyIdfs. Posteriormente procede a recorrer el array contenido mediante un ciclo for y en divide cada uno de los arrays en cuestión (contenido[i]) en palabras mediante el método Split de la clase String y se procede a recorrerlas con un ciclo foreach. Luego se evalúan dos condiciones; la primera es en el caso de que MyIdfs no contenga como llave la palabra evaluada, en cuyo caso se añade a MyIdfs la palabra y se le asocia el valor 1, mientras que a Aux se añade la palabra y se le asocia la posición i; la segunda condición es que Aux evaluado en la palabra sea distinto de la posición i(lo cual significa que la palabra no se había encontrado en el documento i) en cuyo caso el valor asociado a la palabra en MyIdfs es aumentado en 1 y el valor asociado en Aux es sustituido por i. Una vez se haya completado el ciclo for se tendrá un array donde a cada palabra se le asocia la cantidad de documentos en los cuales aparece. Por último se ejecuta un ciclo foreach para cada palabra llave en dicho diccionario y se realiza la siguiente operación: `MyIdfs[palabra] =Log10(Total de documentos/MyIdfs[palabra])`

La fórmula anterior es la forma matemática del cálculo del Idf y representa muy bien el peso de las palabras con respecto a un universo de documentos, pues si la palabra es muy común su Idf disminuye al punto de convertirse en 0 si aparece en todos los documentos.

-TfIdf es un Diccionario de <string, array de float> en el cual a cada palabra del universo de palabras se le hace corresponder un array que almacena el valor del peso de dicha palabra en el documento con respecto al universo de documentos (en ingles: Term Frequency - Inverse Document Frequency o Tf-Idf). El método encargado de realizar esta operación es GetTfIdf, el cual recibe como parámetros al array contenido y al Diccionario de <string, float> idfs; se crea un Diccionario de <string, array de float> que contendrá los Tf-Idfs y se procede a recorrer cada uno de los documentos en contenido y separarlo en palabras. De cada una de dichas palabras, si no existen como llave en TfIdf pues se añade y se le asocia el array de float a cual en la posición i se sobrescribe por un 1; en caso de que la palabra ya exista se le suma 1 a la posición correspondiente del array. Una vez concluido dicho ciclo for se tendrá asociada a cada palabra un array de float con la cantidad de apariciones del término en cada documento. Luego se procede a realizar un ciclo foreach en el cual por cada palabra del universo de palabras se inicia un ciclo for para calcular el TfIdf del término en cada documento de la siguiente forma: `TfIdf[término][i] = (TfIdf[término][i]/cantidad de palabras del documento i) * idfs[término]`

Y ese valor es el que finalmente queda registrado en el diccionario de los TfIdf.

Finalmente la clase Universe posee una serie de propiedades públicas para acceder a dichos campos.

Esto resume el funcionamiento de la Clase Universe, con lo cual finaliza la etapa de preprocesado y comienza la ejecución en tiempo real del proyecto. A continuación se intentaran explicar de manera ordenada los procesos que acontecen en esta segunda etapa.

Dado que ahora entramos en la etapa de búsqueda, hablemos un poco acerca de dicha búsqueda y las nociones generales relacionadas con su funcionamiento. En primer lugar el usuario procede a introducir una consulta, la cual sera analizada para determinar el peso que posee en cada uno de los documentos del universo. Un factor muy importante en este punto es la implementación de un conjuntos de operadores que modificaran dicho peso de la consulta ofreciéndole al usuario funcionalidades que permitan ser mas específicos con lo que se desea obtener como resultado a la consulta. Los operadores implementados son los siguientes:

Operador de “Aparición” (^): Este operador delante de una palabra indica que todos los documentos devueltos han de contener dicha palabra. La manera de garantizar esto es multiplicar el peso de la consulta por cero en todos aquellos documentos que no contengan dichos términos.

Operador de “No Aparición” (!): Este operador delante de una palabra indica que todos los documentos devueltos no pueden contener dicha palabra. La manera de garantizar esto es multiplicar el peso de la consulta por cero en todos aquellos documentos que contengan dichos términos.

Operador de “Importancia” (*): Este operador delante de una palabra indica que dicha palabra es mas importante que el resto. El operador de Importancia es acumulativo, lo cual significa que n operadores delante de un termino harán n+1 veces mas importante a ese termino por encima del resto.

Operador de “Cercanía” (~): Este operador entre dos palabras significa que mientras mas cerca estén esas palabras en un documento, mayor sera el peso de la consulta en ese documento.

Bien, ya tenemos una idea acerca de que hacen los operadores de búsqueda, pero: Como lo hacen? Pues existe una clase llamada query encargada de obtener (en esencia) los documentos ordenados acorde al peso que tiene la consulta en ellos (esto incluye los operadores). Procedamos a explicar la clase query:

La clase query: recibe como parámetros el texto Sin Normalizar introducido por el usuario y un objeto de tipo Universe que contiene los contenidos preprocesados. Dentro de la clase existen varios campos de clase, los cuales explicaremos a continuación:

En primer lugar tenemos tres campos que son información obtenida del objeto Universe. Dichos campos son:

-MyUniverse es el mismo objeto Universe.

-NumberOfDocuments es la cantidad de documentos del universo (obtenida por la propiedad CantDocumentos sobre MyUniverse)

-words es una lista que contiene todas las palabras del universo sin repetición. Se obtiene mediante la propiedad Words aplicada a MyUniverse.

Luego comienza el tema de procesar la información en la consulta introducida por el usuario y aquí haremos un paréntesis para explicar una clase privada existente dentro de la clase query, la cual se encarga de identificar los operadores introducidos en la consulta:

La clase operators:

El funcionamiento de esta clase es muy simple y no entraremos en muchos detalles acerca de su funcionamiento interno, pues no esta compuesta por nada mas que ciclos y condicionales. La clase posee en esencia cuatro campos importantes, cuatro diccionarios donde se recoge la información referente al uso de los operadores presentes en la query (un diccionario por operador):

-El campo MustAppear es un diccionario de <string, bool> donde a cada palabra sin normalizar de la query (obtenida mediante el método Split de la clase String) que contenga delante el operador de Aparición se le asocia un valor true. Este diccionario es utilizado mas bien como una lista, pues los únicos términos que se añaden a su colección de llaves son aquellos que poseen dicho operador.

-El campo MustNotAppear es un diccionario de <string, bool> donde a cada palabra sin normalizar de la query (obtenida mediante el método Split de la clase String) que contenga delante el operador de No Aparición se le asocia un valor true. Este diccionario es utilizado mas bien como una lista, pues los únicos términos que se añaden a su colección de llaves son aquellos que poseen dicho operador.

-El campo Importance es un diccionario de <string, int> donde a cada palabra sin normalizar dela query (obtenida mediante el método Split de la clase String) que contenga delante el operador de Importancia un total de n veces se le asocia el valor entero n+1.

-El campo Distance es un diccionario de <string, string> donde cada par de palabras existentes a los lados de cada operador de Cercanía son asociadas a dicho diccionario. Esto se consigue porque, al detectarse un operador de cercanía en la posición i de las palabras de la query se añaden al diccionario las palabras en la posición i-1, i+1.

Nuevamente la clase query:

Una vez concluida la clase operators podemos proceder a explicar el resto del funcionamiento de la clase query. Lo siguiente que se hace en los campos myquery y MyOperators es; en el primer caso obtener una lista con las palabras de la consulta ya normalizadas y en el segundo caso crear un objeto de tipo operators para que almacene la información referente a los operadores.

-El campo tfidf consiste en un array de float donde en cada posición del array va a encontrarse representado el peso de la consulta en cada documento(respetando el orden en el cual han sido cargados los documentos en la clase Universe). El método encargado de dicha tarea es TfIdf, el cual recibe la lista myquery y ademas MyOperators; y su funcionamiento es extremadamente simple, pues consiste en obtener los arrays de float asociados a cada palabra de la query(en caso de que aparezcan en el universo de palabras) e ir sumándolos para obtener el peso acumulado de todos los términos de la query y por tanto, el peso de la query en si, en cada documento. En este paso el método suma el TfIdf de cada termino multiplicado por (1+n)*100, donde n es la cantidad de operadores de importancia que presenta dicho termino de la query. Una vez conseguido esto aun quedan algunas operaciones simples por realizar. Por ejemplo: Cada documento que no contenga a TODAS las palabras que aparezcan en la colección de llaves del diccionario MustAppear quedaran desestimados, pues su TfIdf sera multiplicado por 0; así mismo quedaran desestimados los documentos que contengan a Cualquier termino en la colección de llaves del diccionario

MustNotAppear. Finalmente todos aquellos documentos que contengan los términos asociados a un mismo par de Llave-Valor del diccionario Distance verán su TfIdf multiplicado por la razón 100000/distancia, donde distancia es el menor numero de palabras encontradas entre ambos términos, y por lo cual a mayor distancia menor TfIdf. (este ultimo método no se explicara en detalles, pues no es nada extraordinario, sino que consiste en un doble for para recorrer el texto e ir almacenando las distancias encontradas hasta quedarse con la menor)

En este punto se tienen los pesos de la consulta en los respectivos documentos, pero están desordenados, por lo cual se procede a ordenarlos mediante un algoritmo de ordenación por burbuja y los indices correspondientes a esos documentos también son ordenados y almacenados en un array de int. Este array de int es precisamente lo que almacena el campo rankedIndex.

Por ultimo quedan dos campos mas, el campo coincidences no es mas que la cantidad de documentos donde la consulta tiene un peso distinto de cero y el campo AllAppears que consiste en un valor booleano indicando si todas las palabras de la query aparecen en el universo de palabras.

Lo ultimo que posee la clase query es un conjunto de propiedades publicas para acceder a estos campos.

Una vez se ha procesado el peso de la consulta en todos los documentos se procese a obtener una sugerencia a dicha consulta. Cabe destacar que dicha sugerencia se obtiene independientemente de la cantidad de resultados obtenidos a la consulta. La clase que se encarga de realizar este proceso es la clase sugerencia, la cual recibe un string query con el contenido de la consulta ya normalizado y un objeto de tipo Universe. La clase posee cinco campos, los cuales detallaremos a continuación.

La clase sugerencia:

-MyUniverse es el mismo objeto de tipo Universe que se recibe en el constructor de la query.

-words es una lista de string con todas las palabras del universo de documentos obtenido a partir de la propiedad Words aplicada sobre MyUniverse.

-totalDePalabras es un valor que representa la cantidad de palabras del universo de palabras. Este valor se obtiene mediante la propiedad totalWords aplicada sobre MyUniverse.

-sugest es un string que representa la cadena que mejor se asemeja a la query compuesta por palabras que aparecen en el universo de palabras. Este string se obtiene mediante el método QuerySugest que recibe el string de la query normalizado como parámetro.

Como funciona QuerySugest?:

QuerySugest obtiene cada palabra de la query y analiza si esa palabra se encuentra en el universo de palabras. De no encontrarse pues se procede a sustituirla por la palabra del universo de palabras que mas se asemeje siempre y cuando la “distancia entre ellas sea menor que 4”. Este proceso de validar dicha distancia es la labor del método sug, el cual verifica que la distancia sea menor que 4 para sustituir la palabra; de no ser así pues no se sustituye el termino aunque no aparezca en el universo de palabras. Pero aún queda por responder la ultima y mas importante interrogante: Como se

obtiene la distancia de una palabra a otra? Pues el algoritmo que determina dicha distancia es el algoritmo de Levenshtein (y la distancia también es conocida como distancia de Levenshtein o distancia de edición). El algoritmo de Levenshtein recibe dos cadenas de caracteres y procede a compararlas; se crea una "matriz" de longitud m+1, n+1; donde m y n son las longitudes de la primera y segunda cadena respectivamente y se procede a llenar la fila 0 con los números desde 1 hasta n y la columna 0 con los números desde 1 hasta m y a continuación se comienza a rellenar la matriz(tomaremos la primera palabra como palabra de "referencia para proceder a realizar las transformaciones). En las posiciones, [i,j] tales q i=j (o sea, en la diagonal de la matriz) se coloca un cero siempre que los caracteres correspondientes sean iguales. En caso contrario se analizan tres opciones, cada una de un coste unitario, y se selecciona el mínimo valor entre el costo de cada una de ellas:

-Analizar si el caracter de la posición i+1 de la primera palabra es igual al caracter de la posición j de la segunda palabra (esta operación es el equivalente a eliminar una letra de la primera palabra)

-Analizar si el caracter de la posición i de la primera palabra es igual al caracter de la posición j+1 de la segunda palabra (esta operación es el equivalente a añadir una letra a la primera palabra)

-Analizar el costo de reemplazar una letra por otra (costo que sera cero siempre que los caracteres correspondientes a las posiciones i y j en la primera y segunda palabra sean iguales, y uno en caso contrario).

El proceso anterior se repetirá tantas veces como caracteres tenga la palabra que estemos utilizando de referencia y se obtendrá al finalizar la suma de todas las operaciones realizadas (la suma de los elementos de la diagonal)

Una vez se ha completado todo este proceso tendremos una "nueva query" que sera similar a la consulta original pero en la cual todos los términos arrojaran resultados.

Finalmente existen un conjunto de propiedades publicas en la clase query con el fin de acceder a la información almacenada en los campos de la clase.

Por ultimo queda una ultima funcionalidad por implementar en nuestro Moogle!, los snippets. Los snippets son fragmentos de los textos originales que mejor representan la consulta en dichos textos. Su obtención queda agrupada en la clase snippet y la detallaremos a continuación.

La clase snippet: recibe como parámetros un objeto de tipo query llamado MyQuery, un objeto de tipo Universe llamado MyUniverse y un valor int llamado indice (este indice representa el documento en el cual se desea encontrar el snippet correspondiente a MyQuery. Tenemos 6 campos de clase, mencionados a continuación:

-MyQuery es precisamente el objeto de timo query recibido como parámetro.

-MyUniverse es también el objeto de tipo Universe que se recibe como parámetro.

-indice es el valor index recibido como parámetro.

-contenido es un array de string en el cual aparecen las palabras del texto en cuestión sin normalizar. Para obtenerlas se emplea el método ReadAllLines de la clase File y luego se concatenan las lineas del documento mediante el método Join de la clase Sring.

–valores es un array de float de la misma longitud que contenido en el cual a cada palabra que figure en contenido se le asociara el valor de TfIdf de dicha palabra normalizada (si la palabra

aparece en la consulta el valor de su TfIdf se aumenta considerablemente), de existir (por ejemplo, si en contenido encontrásemos el string "(<" (lo cual claramente no es una palabra, y por tanto no tendrá asociado ningún valor de TfIdf) se añade 0 a la respectiva posición en valores).

-Mysnippet es un string que contendrá el fragmento de texto que mejor coincida con la consulta (la cantidad de palabras que por defecto tendrá este string es 50, a no ser que el documento analizado posea menos de 50 palabras; en cuyo caso se reducirá a todo el documento). Ahora, como saber cual es el fragmento de texto que mejor coincide con la query? El algoritmo empleado para esto es ni mas ni menos que el de "subarray de suma máxima" mediante el método sumamax. El método sumamax recibe como parámetros a valores y una longitud correspondiente a la cantidad de palabras que tendrá el snippet y procede a encontrar el subarray de suma máxima de dicha longitud dentro de valores; posteriormente almacenara el indice correspondiente al primer valor de ese subarray con respecto a valores y devolverá un array de string donde quedaran almacenadas las n palabras posteriores a ese indice almacenadas en contenido (siendo n la cantidad de palabras que ha de tener el snippet). Finalmente el método GetSnippet se encarga de concatenar todas esas palabras en un solo string y ese es el snippet de nuestra consulta en el texto en cuestión.

La clase snippet posee un conjunto de propiedades publicas implementadas con el objetivo de acceder los campos de la clase.