¿Alguna vez ha deseado buscar entre todos sus archivos de texto una palabra o frase particular? ¿Ha deseado organizar sus documentos de texto según su relevancia respecto a una búsqueda?

Pues usted debería probar Moogle!, el nuevo buscador de texto actualmente sensación entre decenas de estudiantes de la Facultad de Matemáticas y Computación de la Universidad de La Habana.

 Moogle!

Introduzca su búsqueda

 Buscar

¿Qué es Moogle!?

Moogle! es un proyecto de programación totalmente original cuyo propósito es buscar inteligentemente un texto en un conjunto de documentos. Aplicando el **Modelo Vectorial** y la medida numérica **TF-IDF**, esta aplicación es capaz de leer archivos de texto en formato ***.txt*** y una búsqueda introducida por el usuario, y entonces devolver los documentos de texto relevantes en relación a lo buscado, ordenados de más relevantes a menos.

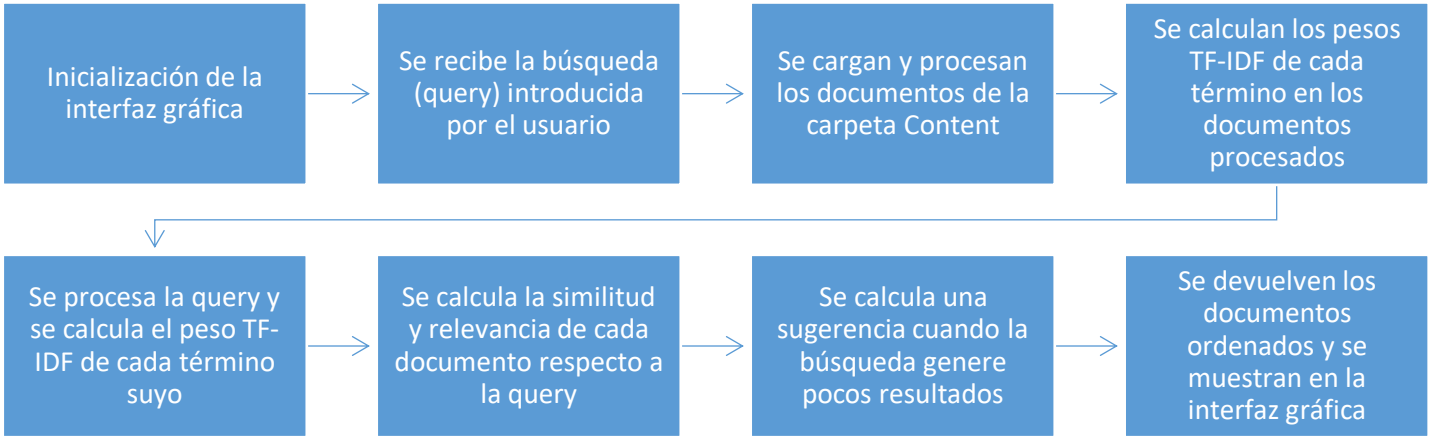
¿Cómo puedo ejecutar la aplicación?

Para ejecutar el proyecto en Windows necesitas abrir el archivo solution Moogle.sln y presionar correrlo en Visual Studio 2022, o ejecutar el comando:

dotnet watch run --project MoogleServer

¿Cómo funciona el proyecto?

El proyecto sigue un camino marcado:



La clase Moogle.cs es el puente entre todo lo que sucede por detrás en el proyecto.

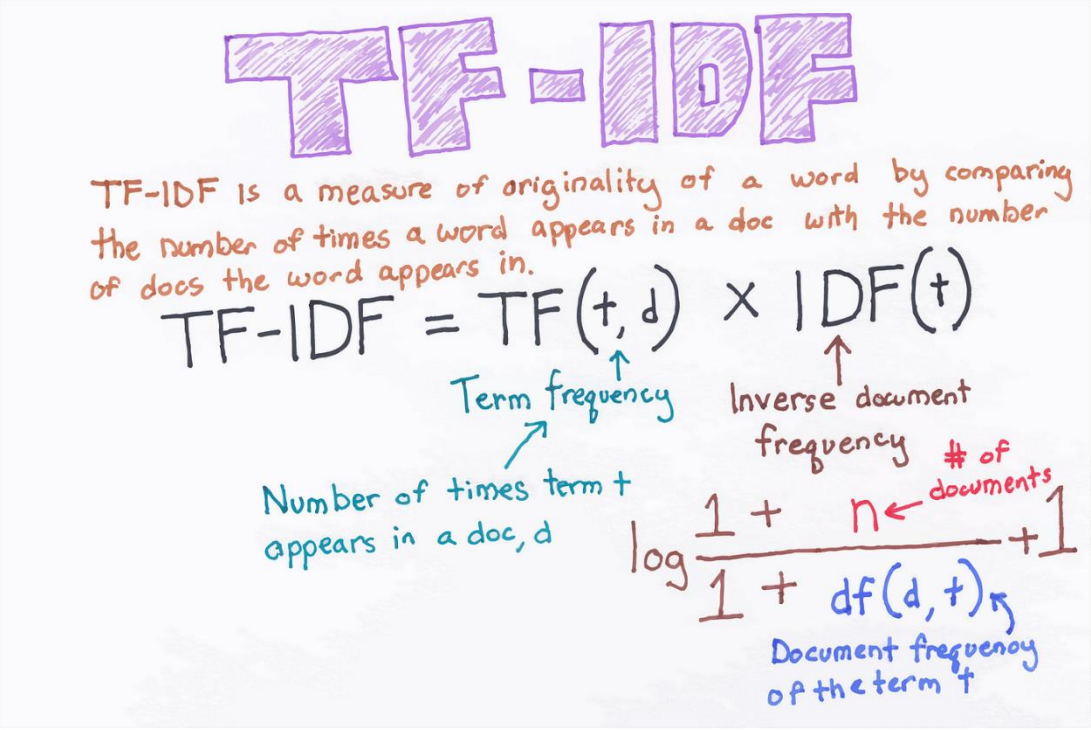
```
public static class Moogle
{
    //creando los objetos data y tfidf de sus respectivas clases alisto mis documentos
    public static DataBase data = new DataBase();
    public static TFIDF tfidf = new TFIDF(data);
    public static SearchResult Query(string query)
    {

        //si la query no es vacia, ejecutar la búsqueda
        if (!string.IsNullOrWhiteSpace(query))
        {
            Query user_query = new Query(query, tfidf);
            Similarity similarity = new Similarity(user_query, tfidf);
            SearchItem[] items = similarity.Items.ToArray();
            Suggestion suggestion = new Suggestion(query, similarity.DatabaseDistinctWords, tfidf.Term_DF);
            string sug = suggestion.QuerySuggestion;
            //los documentos encontrados son devueltos junto con una sugerencia
            return new SearchResult(items, sug);
        }
        else
        {
            SearchItem[] items = new SearchItem[1]
            {new SearchItem("Por favor, ingrese algún término para realizar una búsqueda", "Intente nuevamente", 0)};
            return new SearchResult(items, "");
        }
    }
}
```

Al recibir la query del usuario, se instancia un objeto estático *data* de la clase DataBase que tiene una serie de propiedades que va a llenar en su constructor. Va a conseguir la dirección de la carpeta Content en mi proyecto usando la clase Directory y el método GetCurrentDirectory, y retroceder a la carpeta general de mi proyecto

(porque la obtenida es una sus subcarpetas), ubicando DataBasePath en la carpeta Content. Se van procesar todos los textos de la carpeta Content, obtenidos con el método ReadAllText de la clase File, se los va a tokenizar por palabras eliminando todos los signos de puntuación y caracteres especiales [ ] { } ( ) \ | / “ ” ‘’, así como los espacios en blanco, usando la clase Regex de System y sus métodos Split y Replace. Cada documento queda representado en una serie de diccionarios que relacionan el nombre del documento con su texto (Doc_Text), su nombre con sus palabras separadas (Doc_Words), así como quedará un array de todas las palabras distintas entre todos mis documentos (TotalDistinctWords).

Luego se instancia un objeto estático *tfidf* de la clase TFIDF, que recibe como parámetro a una DataBase, en este caso *data*, para acceder a sus propiedades. Así se obtiene en *tfidf* los diccionarios importantes y el array del total de palabras distintas mencionados anteriormente. En esta clase TFIDF se llenarán de valores nuevos diccionarios que asocian a cada uno de los términos diferentes de todos mis textos con la cantidad de documentos en los que aparece ese término (Term_DF), y luego a esos términos se les asocia la frecuencia inversa IDF (Term_IDF). Otro diccionario tendrá como llave el nombre del documento y como valor tendrá asociado un diccionario de cada término del documento con su frecuencia absoluta y luego relativa en el documento (Doc__Term_TF). Con esos valores guardados se calcula el TFIDF de cada término en cada documento y se guarda en un nuevo diccionario de diccionarios el nombre del documento con todas sus palabras y los valores TFIDF de esas palabras (Doc__Term_TFIDF).



Como ambos objetos fueron creados como estáticos, no habrá necesidad de crearlos para nuevas búsquedas del usuario.

Se crea entonces una instancia de la clase Query llamada *user_query* que recibe la búsqueda introducida por el usuario (query) y *tfidf*. Acá se separa en términos la query, solo separando por espacios, y calcula el TFIDF de cada término de la misma.

Se crea un objeto similarity de la clase Similarity que recibe *user_query* y *tfidf*. Acá se calculará la similitud de cada documento con la query, a través de la fórmula de similitud de coseno. Se multiplica cada vector del documento (representado como Doc__Term_TFIDF) por el vector de la query (QueryTerm_TFIDF), iterando por todos los términos diferentes de la base de datos. La operación el término solo se realiza si el término se encuentra en ambos vectores, en caso contrario es porque en al menos uno de los vectores el término tiene un TFIDF = 0 y por tanto se descarta. El vector obtenido queda representado por el diccionario (Doc__Term_newTFIDF), y al calcular la suma de todos los TFIDF en ese nuevo vector, esta se guarda en un diccionario llamado (Doc_BruteSimilarity), obteniendo así los valores que van en el numerador de la fórmula de similitud del coseno. Luego sumando los pesos del vector query elevado al cuadrado y sumando los de cada vector documento, se obtienen las otras variables o valores para calcular la similitud de coseno:

$$SimCos(d_{(d)},q) = \frac{\sum_{n=1} \left(P_{(n,d)} \times P_{(n,q)} \right)}{\sqrt{\sum_{n=1} \left(P_{(n,d)} \right)^2 \times \sum_{n=1} \left(P_{(n,q)} \right)^2}}$$

Luego se obtiene el nombre del documento, su score (valor de su similitud de coseno) y calcula su snippet, un fragmento del texto y se guardan las 3 variables en un array ítems del tipo SearchItem.

Luego se crea un objeto de la clase Suggestion que calcula con la distancia de Levenshtein un posible cambio a los términos de la query para que encuentre más documentos, en caso de haber encontrado pocos o haber introducido un término en la query que no se encuentra en la base de datos o que es muy raro en la misma.

Luego en la interfaz gráfica son devueltos los documentos encontrados y la sugerencia en caso de existir.

Especificaciones:

Para introducir la query puede presionar Enter luego de escribir para que haga la búsqueda sin tocar el botón de Buscar.

Para introducir la sugerencia dada por el programa puede presionarla directamente y llevara a cabo la búsqueda con esa sugerencia como query.