ALM / Memoria proyecto ALM. Búsqueda aproximada de cadenas.

Índice:

1. Introducción.

Este proyecto es la continuación del motor de búsqueda de cadenas en noticias desarrollado en la asignatura de SAR, esta vez realizando una serie de modificaciones que permiten la búsqueda aproximada de cadenas.

Anteriormente la búsqueda de términos se limitaba a la palabra exacta, ahora añadimos la posibilidad de elegir una tolerancia de búsqueda sobre cada término.

Para poder realizar las búsquedas con tolerancia se debe hacer uso de diferentes algoritmos y estructuras de datos que permitan comparar dos cadenas, existen varias posibilidades para ello y debemos elegir la solución que mejores resultados ofrezca en determinadas circunstancias.

La primera parte de esta memoria se centra en el estudio temporal de diferentes algoritmos combinados con varias estructuras de datos haciendo uso de los términos en la obra del ***Quijote***, esto nos permite sacar una serie de conclusiones para poder elegir el par (algoritmo, estructura de datos) más conveniente para la búsqueda de términos con tolerancia en los índices de noticias.

En la segunda parte se expone el par (algoritmo, estructura de datos) seleccionado en vista de los resultados del apartado anterior y como lo hemos integrado en el proyecto de la asignatura de SAR.

2. Medida de distancia entre dos cadenas.

2.1. Distancia de Levenshtein.

Antes de comparar cadenas, se debe definir la distancia entre las mismas, así como la tolerancia (distancia máxima en la que dos cadenas se consideran “cercanas”).

Para ello vamos a hacer uso de la definición de la **Distancia de Levenshtein.** La distancia de Levenshtein entre dos cadenas α y β, no necesariamente de la misma longitud, se define como el número mínimo de operaciones básicas que son necesaria para transformar la cadena α en la cadena β. Las operaciones permitidas son:

- Borrado de un carácter de la cadena α,
- Inserción de un carácter de la cadena β, y
- Sustitución de un carácter de la cadena α por otro de la cadena β.

Se debe asociar un coste a cada una de las operaciones, en nuestro caso el coste de cada operación es igual a 1.

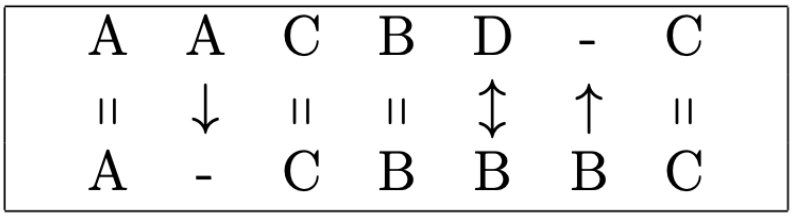

Figura 1. Distancia de 2 cadenas con coste de 3.

2.2. Distancia de Damerau-Levenshtein.

La **Distancia de Damerau-Levenshtein** es una extensión de la distancia de Levenshtein en la cual además de la sustitución, borrado e inserción también se permite la transposición de dos caracteres consecutivos.

3. Algoritmos y estructuras de datos.

3.1. Algoritmos.

Los algoritmos utilizados durante para la realización de la práctica han sido lo siguientes:

- Programación dinámica.
- Ramificación y poda.

3.2. Estructuras de datos.

Las estructuras de datos utilizadas son las siguientes:

- Lista.
- Trie.

3.3. Pares (algoritmo, estructura de datos).

Las combinaciones (algoritmo, estructura de datos) utilizadas para construir la medida de la **Distancia** de **Levenshtein** y **Damerau-Levenshtein** han sido las siguientes:

- (Programación dinámica, Lista) -> (PD, L).
- (Programación dinámica, Trie) -> (PD, T).
- (Ramificación y poda, Trie) -> (RP, T).

3.4. Resultados.

Tras implementar las diferentes combinaciones se han realizado una serie de pruebas sobre el set de datos del ***Quijote,*** para poder evaluar empíricamente que algoritmo ofrece mejores resultados.

Las mediciones se han realizado con 20 repeticiones sobre cada búsqueda para poder obtener una media representativa de cada medición y con 6 niveles de tolerancias [1 … 6].

Para 20 medidas la desviación estándar muestral es bastante parecida a la de la población, el sesgo para N = 20 es del 5%, suponiendo la independencia de los datos. Ha resultado que esta desviación estándar es muy pequeña para algunas medidas, del orden de 10^-3 para las medidas realizadas, por tanto no son apreciables debido a su reducido tamaño.

Los resultados se muestran en las siguientes figuras.

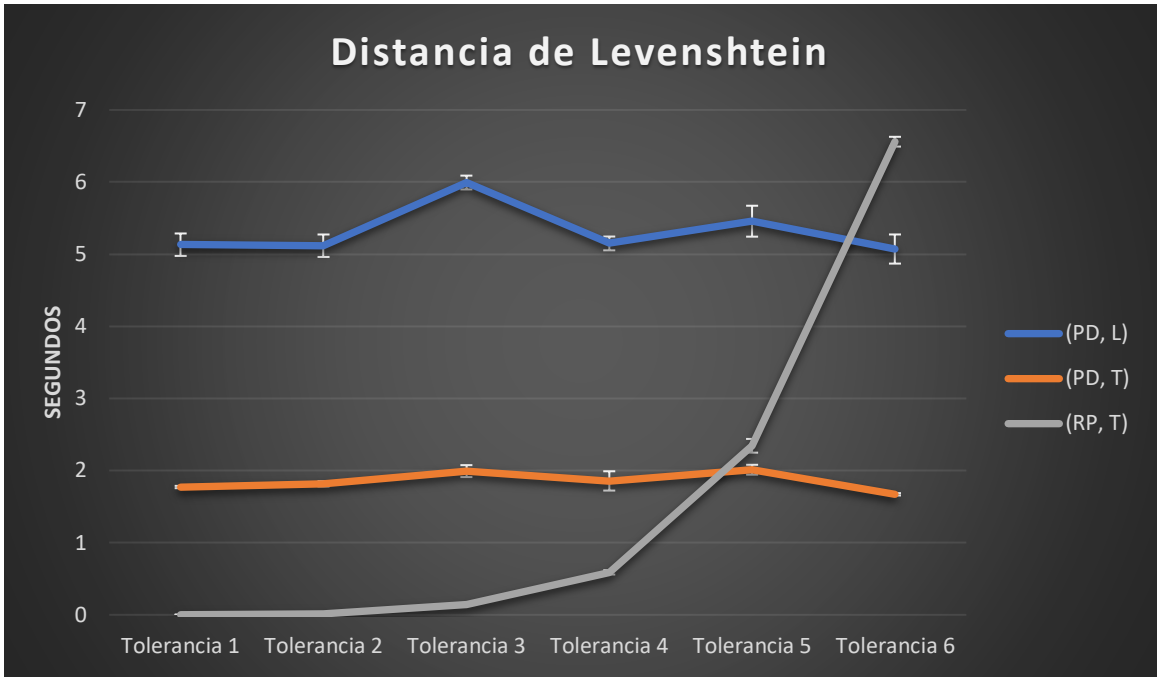

Figura 2. Resultados temporales comparando la Distancia de Levenshtein entre dos cadenas.

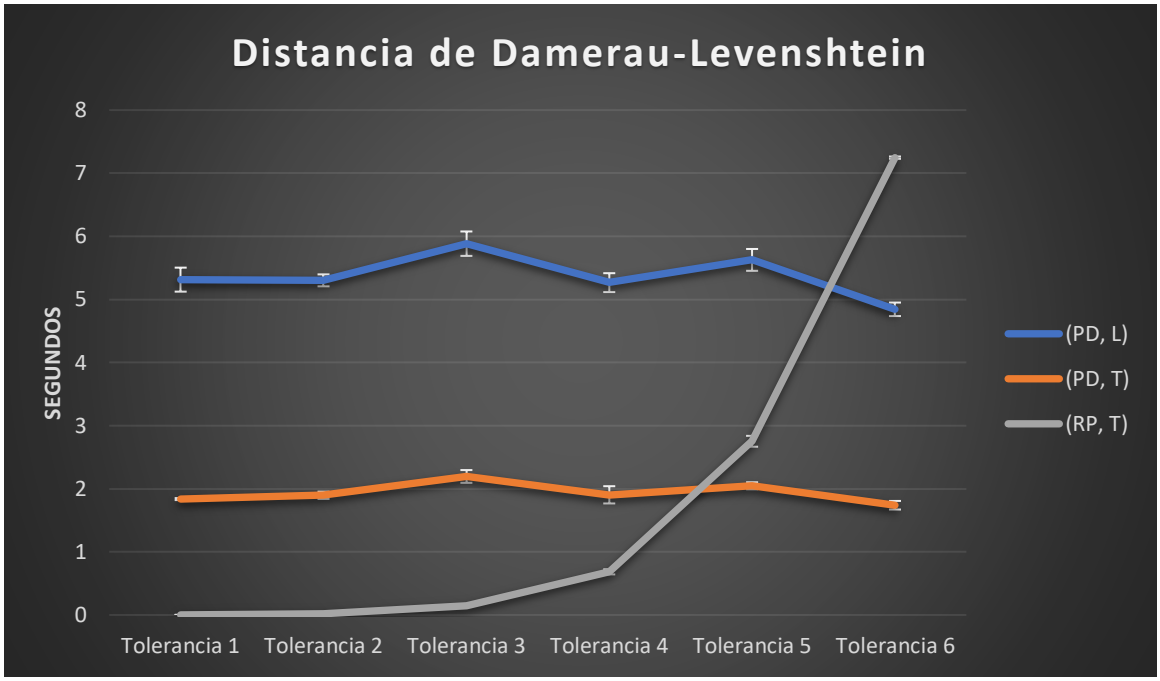

Figura 3. Resultados temporales comparando la Distancia de Damerau-Levenshtein entre dos cadenas.

3.5. Conclusiones.

A la vista de los resultados hemos podido comprobar que tanto el método de programación dinámica con trie y comparación con todas las palabra tienen un coste constante ya que siempre se lleva a cabo el mismo numero de operaciones. La diferencia de tiempos entre estos dos métodos se debe a que en un trie las palabras que comparten prefijo siguen la misma ruta y se comparan más palabras por unidad de tiempo. Por el contrario, el método de ramificación y poda depende de la tolerancia.

Se puede apreciar que el par (Ramificación y poda, Trie) ofrece mejores resultados temporales que el resto para unas tolerancias menores a 5, para un número de tolerancia mayor o igual a 5, el par (Programación dinámica, Trie) presenta mejores resultados.

Se ha optado por el uso del par (Ramificación y poda, Trie) en el proyecto de SAR, ya que se considera que una tolerancia superior a 4 no es representativa para la búsqueda de términos, es decir, la mayor parte de las búsquedas que se realicen no creemos que requiera una tolerancia tan elevada ya que se devuelven prácticamente todas las noticias (en la mayor parte de las búsquedas de prueba) y por tanto la precisión del motor de búsqueda sería muy baja.

En el caso de que se descubriera que si que puede tener sentido para algún sistema concreto incluir tolerancias mayores se podría configurar el searcher para que en función del nivel de tolerancia usase un par u otro, de tal modo que para tolerancias menores a 5 usase el par (Ramificación y poda, Trie) y para tolerancias mayores a 5 el par (Programación dinámica, Trie).

4. Integración con el recuperador y buscador de SAR.

Una vez realizadas las mediciones temporales de cada par y haber comprobado cual de ellos se ajusta mejor a nuestro proyecto para la búsqueda de términos en un set de noticias hemos procedido a la creación de una estructura de paquetes y herramientas en Python para poder integrar de una forma limpia y ordenada todo lo desarrollado durante esta asignatura con el proyecto de SAR.

Así pues, no hemos generado un único archivo **ALT_library.py,** y en su lugar se ha optado por ordenador los archivos de una forma jerárquica y por funcionalidades, lo cual nos ha permitido poder trabajar en paralelo haciendo uso de una herramienta de control de versiones (git). Dando como resultado un proyecto mucho más escalable y con mayores posibilidades para la modificación e implementación de nuevas funcionalidades.

Así pues, en esta sección se comentará la estructura del proyecto y como poder ejecutar el programar y los test (medición de tiempos sobre el documento del ***Quijote***).

El proyecto se encuentra dividido en dos grandes bloques:

- data: directorio en el que se alojan los sets de datos sobre los que queremos obtener la información.
- src: directorio en el que se encuentra el código del proyecto.

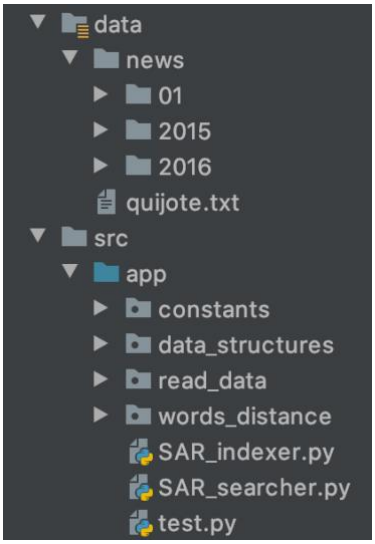

Figura 4. Muestra de la estructura del proyecto.

4.1. Directorio data.

Este directorio contiene los datos tal y como se comentó en el apartado anterior, a su vez está dividido en dos partes:

- news: directorio en el que se alojan los diferentes bancos de noticias.
- quijote.txt: documento utilizado para la realización de los test para le medición de tiempo de los algoritmos.

4.2. Directorio src.

Aquí se encuentra todo el código del programa, está organizado en una serie de paquetes con funciones muy concretas para facilitar el trabajo en paralelo, el crecimiento del proyecto, la modificación de este y evitar código duplicado.

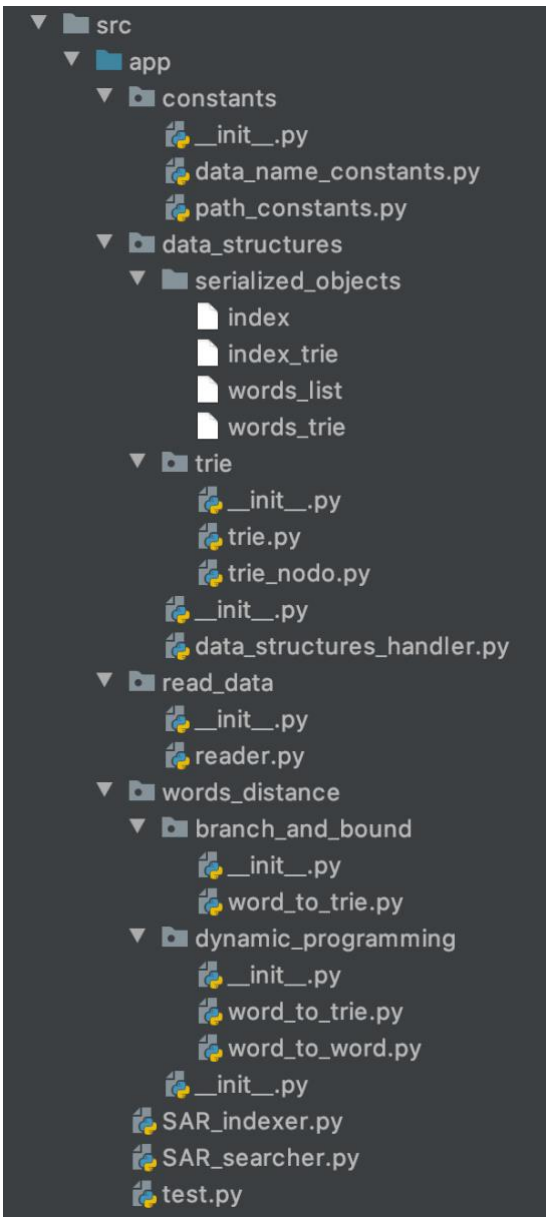

Figura 5. Paquetes de código de los que se compone el proyecto.

Los paquetes de los que consta son los siguientes:

- constants: contiene constantes para el nombre de los ficheros y las rutas a los mismos, con esto nos evitamos tener que definir los nombres en varios archivos y a su vez eliminamos rutas relativas o tediosas, ya que definimos como directorio base para los documentos la ruta a data.
- data_structures:
    - serialized_objects: directorio en el que se guardan los objetos serializados.
    - trie: paquete en el que se define la estructura de datos del trie.
    - handler para manejar el guardado y la carga de los objetos serializados

- reader_data: paquete encargado de la lectura y limpieza de datos (eliminación de espacios, tildes, números, etc.).
- word_distance: en este paquete se encuentran las implementaciones de los diferentes pares (algoritmo, estructura de dato)
- archivos ejecutables:
    - SAR_indexer.py: encargado de leer el directorio donde se encuentran las noticias y guardar los índices serializados (original y trie).
    - SAR_searcher.py: carga los índices generados por el indexer y realiza las búsquedas introducidas por el usuario.
    - Test.py: encargado de realizar las mediciones de tiempos para de los diferentes algoritmos.

4.3. Ejecución del programa.

Para que los archivos encargados de las ejecuciones puedan reconocer los diferentes paquetes y las rutas en las que se encuentras los sets de datos se deben ejecutar desde la ruta ***/Proyecto/SAR_Project/src/app***.

No es necesario indicar la ruta al archivo o directorio donde se encuentran las noticias, simplemente indicando el nombre de este el path reconoce donde se encuentra.

Ejemplo de ejecución:

python SAR_indexer.py 2015 index.

python SAR_searcher.py index -q capital%1