Javier Garrido, Aitana Villaplana
Diego Ros, Pau Sánchez

PROYECTO SAR - ALT

Índice:

0.- Introducción

En este informe vamos a presentar el trabajo que hemos realizado según la programación de prácticas para adaptar el buscador de documentos de SAR para realizar búsquedas aproximadas de cadenas con tolerancia.

1- Levenshtein y Levensthtein-Damerau de cadenas

Para realizarlos hemos comparado mediante la librería *numpy* las dos cadenas. Hemos hecho métodos auxiliares para inicializar la matriz dependiendo de si es la distanca de Levenshtein o Levensthtein-Damerau. Tras lo que iterativamente completamos la matriz poniendo en cada celda la mejor distancia para llegar a esta. La distancia final es la almacenada en la última fila y última columna.

Para calcular la distancia hasta cada letra en Dammerau hemos añadido la opción de que dos letras se puedan intercambiar.

2.- Trie

Para realizar el Trie, hemos usado una estructura tipo lista de nodos (clase *Node*), donde nos guardamos los datos necesarios (el carácter, si es final, hijos, etc.) .El principal método que hemos implementado en el trie es *add_son*, donde se le da una palabra y, tras una búsqueda previa en el trie, la añade si la palabra no está, se completa si está en parte o no se añade si ya esta en el Trie. En el trie mantenemos dos listas, una con todos los nodos, y otra con los índices de los nodos finales, para agilizar la búsqueda de PD.

3.- Levenshtein y Levensthtein-Damerau de cadena y trie con PD

En *SAR_library* encontraremos *levesteinTree_Word_PD* y *dam_levesteinTree_Word_PD*. Para completar las distancias mínimas para cada letra de la palabra será similar a cuando se realiza entre cadenas pero teniendo en cuenta que se trabaja con un trie, la distancia entre letra y nodo no se calculara con los elementos de la matriz directamente anteriores, sino mirando los elementos de la matriz del padre del nodo del trie y las letras de la cadena indicadas teniendo en cuenta las inicializaciones en cada caso que dependerán de la profundidad del nodo, dato almacenado en el trie.

Una vez calculada la matriz de distancias, para cada nodo final del trie, que coincide con las columnas de la matriz, si su distancia entra dentro del límite establecido, añadimos la palabra que representa como solución.

4.- Levenshtein de cadena y trie con ramificación

En la versión con ramificación, se implementa una cola FIFO cuyos elementos son tuplas de tres elementos (Índice de la letra de la palabra que se está comparando, Índice del nodo con el que se compara, Coste de llegada a los dos parámetros previos).

Los métodos consisten en iterar mientras queden elementos en la cola. Primero se eliminará el elemento que se está analizando, conservando los datos, tras lo cual si no se ha de podar, en nuestro caso solo podamos por factibilidad (distancia máxima), ramificaremos, si es oportuno, es decir, si se cumplen las condiciones para inserción, borrado , sustitución, y en Damerau intercambio.. Y si estas mirando la última letra de la cadena y es un nodo final lo añadiremos al conjunto de soluciones. Ya que este método tiende a reañadir palabras a la solución, solo guardaremos la mejor distancia para cada palabra final, mediante un diccionario.

5.- Adaptación código SAR

Se ha adaptado el código para que admita la búsqueda con tolerancia. En caso de usar la distancia de Levenshtein, se usará el carácter % para indicar la tolerancia en la búsqueda y en el caso de querer usar la distancia de Lev-Damerau se usará el carácter @.

Teniendo en cuenta que el coste de el algoritmo de programación dinámica en este caso es constante O(k) y el de ramificación es exponencial $O(x^n)$, se usa el primero para los casos donde la tolerancia sea un numero alto (3 o más) y para los casos con una tolerancia menor (1 o 2) se utiliza el de ramificación.

También se ha adaptado el código para que sea posible hacer búsquedas usando puertas lógicas, uso de pararentesis y el resto de ampliaciones de la asignature de SAR que no sean directamente indicados o aplicados sobre la palabra con la búsqueda con tolerancia.

6.- Comparación de los algoritmos de PD y ramificación

En este apartado hemos realizado una serie de pruebas con diferentes búsquedas por número de tolerancia, para comparar el funcionamiento de los 2 algoritmos (programación dinámica y ramificación) en cuanto a coste computacional, y sacar su coste en el mejor y peor caso, para así utilizar uno u otro en el caso en el que mejor convenga para reducir el coste.

Comparación de tiempos entre los algoritmos de programación dinámica y ramificación.

Tabla de tiempos del algoritmo de la distancia de Levenshtein (en segundos):

	levesteinTree_Word_PD	levesteinTrie_Word_Ramificacion
alexander%1	38,85917	0,02397
iluminación%1	41,58154	0,01603
virgen%1	24,1998	0,02796
planeta%1	28,70221	0,020016
alexander%2	36,31289	0,71556
capital%2	27,62597	0,79654
secuela%2	33,6849	0,81926
alexander%3	59,9741	69,97358
capital%3	24,810135	56,84853
permisos%3	33,27321	99,74836
alexander%5	33,1727	200
iluminación%5	45,50514	200
capital%5	24,41236	200

Imagen 1

En el caso de ramificación y número de tolerancia 5, se ha optado por poner un numero mucho más alto que el resto dado que la ejecución del algoritmo suponía un tiempo excesivamente alto.

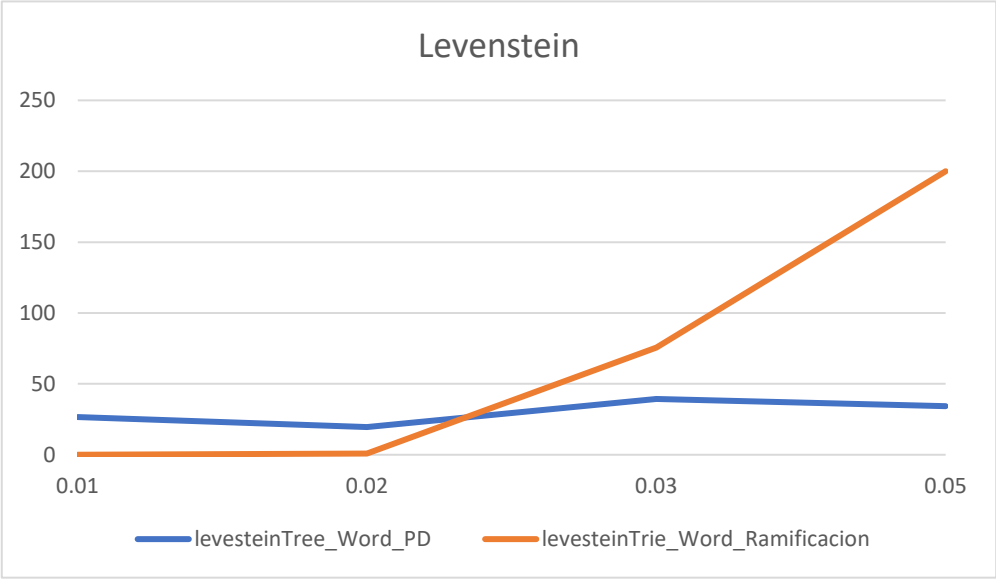

Imagen 2

Esta imagen representa el coste de ambos algoritmos en las pruebas, se puede confirmar que la programación dinámica tiene un coste constante, y la ramificación es exponencial.

Los valores del eje x, representan el número de tolerancia (1,2,3 y 5).

Y el eje y, el tiempo en segundos obtenido.

Los valores usados en la gráfica son las medias de los tiempos de la Imagen 1, que a continuación se muestran:

	levesteinTree_Word_PD	levesteinTrie_Word_Ramificacion
1 %	26,668544	0,021994
2 %	19,524752	0,77712
3 %	39,3524816666667	75,52349
5 %	34,3634	200

Imagen 3

Tabla de tiempos del algoritmo de la distancia de Damerau-Levenshtein:

	dam_levesteinTree_Word_PD	dam_levesteinTrie_Word_Ramificacion
alexander@1	30,44511	0,01795
iluminación@1	37,79004	0,01205
virgen@1	20,119998	0,01201
planeta@1	24,04219	0,010087
alexander@2	30,188441	0,58609
capital@2	23,7534	0,437574
secuela@2	24,701721	0,41105127
alexander@3	30,19471	35,7331309
capital@3	23,73758	32,6135959
permisos@3	25,643845	32,3685464
alexander@5	30,01848	200
iluminación@5	36,91885	200
capital@5	23,9378	200

Imagen 4

En este caso, como en el anterior, optamos por poner un valor alto en el caso de ramificación y tolerancia 5, y se puede confirmar el coste de cada algoritmo también en el caso de Damerau.

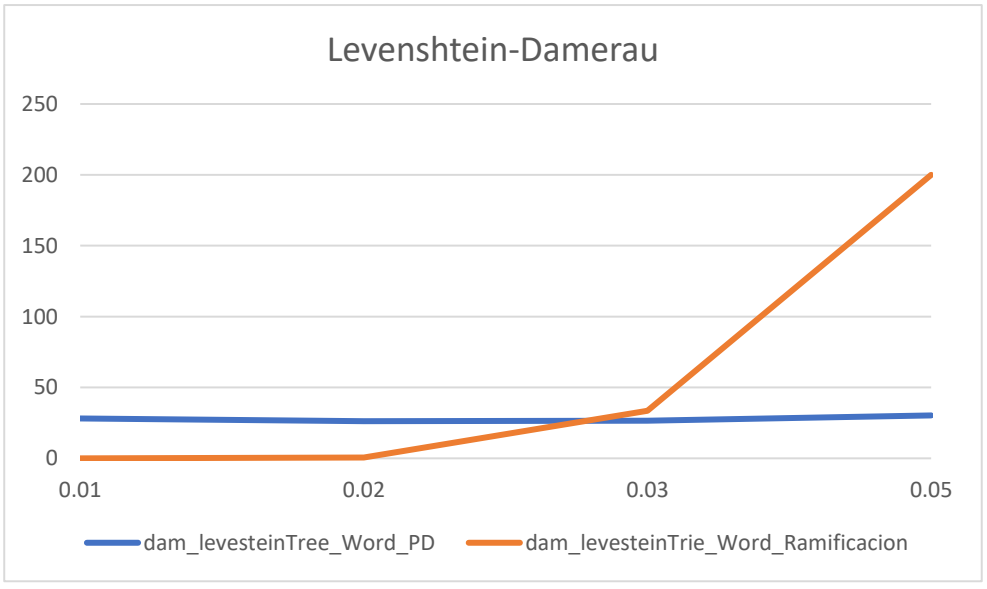

Imagen 5

Igual que el caso anterior, aquí se representa la función de los costes de los algoritmos

	dam_levesteinTree_Word_PD	dam_levesteinTrie_Word_Ramificacion
1 %	28,0993345	0,01302425
2 %	26,2145206666667	0,478238423333333
3 %	26,5253783333333	33,5717577333333
5 %	30,29171	200

Imagen 6