PROYECTO ALGORITMIA

Realizado por:
Asier Serrano Aramburu
Rodrigo Juanes Busolo
Javier Maroto Llamas

Índice

Gestión de tareas

Hemos empleado para el versionado de código GitHub y como canal de comunicación Discord.
Las tareas de cada integrante, son las siguientes:

Javier:
-Damerau-Levenshtein restringida e intermedia.
-Damerau-Levenshtein restringida con y sin umbral.
-Damerau-Levenshtein restringida e intermedia con trie-cadena.
-Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas.

Asier:
-Levenshtein con y sin umbral, y trie (modo backwards).
-Clase trie.
-Solo para Levenshtein, probar cota optimista (tarea 3).

Rodrigo:
-Método suggest con threshold.
-Cotas inferiores y superiores de los algoritmos de cálculo de distancia de edición, para evitar ejecuciones no deseadas.
-Damerau-Levenshtein intermedia con y sin umbral.

Trabajo realizado

Se ha implementado un sistema eficiente para realizar la búsqueda aproximada de una cadena respecto de un conjunto, ampliando así el motor de recuperación de información de SAR.
Para ello, se han desarrollado todas las funcionalidades y pruebas, tanto obligatorias como opcionales, exceptuando la distancia de Damerau-Levenshtein general de la tarea 1 (opcional).
A continuacion, explicamos nuestro trabajo, indicando en color azul las tareas obligatorias, y en rojo las opcionales.

Tarea 1: Implementar las distancias en su versión básica.

Distancia de levenshtein sin optimización espacial:

def levenshtein_basic(begin, end) en ***distances.py***
Hemos implementado este método, empleando otro auxiliar, def crear_matriz_levenshtein(begin, end), con lo cual se crea una matriz de tamaño m*n, siendo m la talla de la primera palabra, y n la de la segunda. La primera fila y la primera columna, tienen los valores 0..m, 0..n, respectivamente. Seguidamente, se itera, y para cada celda con índice de columna y fila distinto de 0, se comprueba si las letras de las palabras en la posición coinciden, en cuyo caso se asigna el valor almacenado en (i-1, j-1), o no, donde se guarda el mínimo de las operaciones insertar, borrar o reemplazar.

Finalmente, se devuelve el valor en la posición (n,m).
Los costes temporal y espacial son O(m*n).

Distancia de levenshtein con optimización espacial:
def levenshtein_optimized(begin, end) en ***distances.py***
Realmente para computar la distancia de levenshtein, solamente hace falta cada paso, 2 arrays de tamaño n. El primero, donde están las distancias, inicialmente con valores 0..n. El segundo, donde se van a almacenar las distancias a calcular. Al acabar el bucle interior, el primero pasa a ser el segundo, y este segundo se resetea, almacenando en su primera posición la posición de la primera palabra que se está recorriendo.
Finalmente, se devuelve la última posición del segundo array.
El coste temporal es O(m*n), y el espacial, O(m).

Distancia de Damerau-levenshtein restringida:
dp_restricted_damerau_backwards(x, y) en ***distances.py***
Hemos hecho una implementación del algoritmo por columnas. Para esto, eran necesarias 3 columnas: 2 de ellas comunes al algoritmo de la distancia de levenshtein y 1 más para el cálculo del intercambio (llamada columnasI, la cual es la columna más anterior).
Tomando esto en consideración añadimos una condición que comprueba si se puede realizar un intercambio.

Distancia de Damerau-levenshtein intermedia:
dp_intermediate_damerau_backwards(x,y) en ***distances.py***
Hemos hecho una implementación del algoritmo por columnas también. Para esto, eran necesarias 4 columnas: 2 de ellas comunes al algoritmo de la distancia de levenshtein, 1 más para el cálculo del intercambio (llamada col4) y otra más para el cálculo de la condición de la intermedia (col4 de tal forma que queda en este orden col4→colI→col1→col2).
Tomando esto en consideración añadimos 2 condiciones que comprueban si se dan los casos ab → bda y bda → ab.

Tarea 2: Implementar las distancias en su versión mejorada.

Distancia de levenshtein con threshold:
def dp_levenshtein_threshold(begin, end, threshold=2**30) en ***distances.py***
El proceso del cálculo de distancias es el mismo que en el método *def levenshtein_optimized(begin, end),* no obstante, si una de ellas supera el threshold especificado, se devuelve dicho umbral + 1.
El coste temporal es O(m*n), y el espacial, O(m). Mejora el caso medio.

Distancia de Damerau-levenshtein restringida con threshold y distancia de Damerau-levenshtein intermedia con threshold:
dp_restricted_damerau_threshold(x, y, threshold) en ***distances.py***
dp_intermediate_damerau_threshold(x,y,threshold) en ***distances.py***

La modificación es simple, calcular el mínimo de la columna que hemos calculado y si este es mayor que el threshold devolvemos threshold + 1.
Además nos evitamos calcular aquellas filas tales que nº columna - threshold sea <= nº de fila <= nº columna + threshold, pues estas no aportan nada al algoritmo.
Ambas funciones emplean la misma lógica.

Tarea 3: Sugerencia de palabras.

Método suggest sin trie:
suggest(self, term, distance="levenshtein", threshold=None) en ***spellsuggest.py***
Para evitar repetirlo dentro del bucle, guardamos en una variable la función de la distancia a utilizar en base a la que se pasa en la llamada al método. Luego por cada palabra del diccionario se busca si la distancia es *levenshtein,* aplicar la cota optimista para evitar ejecuciones no deseadas, y en caso contrario comprobar si la diferencia entre longitudes de las cadenas es mayor al threshold, para como dicho anteriormente evitar ejecuciones no deseadas. Por último se comprueba si la distancia devuelta no se encuentra en el diccionario de resultados y si es así, se inserta la distancia como valor y la palabra como clave.

Método suggest con trie:
suggest(self, term, distance="levenshtein", threshold=2 ** 31) en ***spellsuggest.py***
Este método es similar al otro suggest. Guardamos en una variable la distancia a emplear. Luego se llama a la función con el término y *self.trie* y se devuelve como resultado el diccionario resultante.

Cotas optimistas para levenshtein:
def level_flat(begin, end) en **level.py**
Obtenemos la unión de los caracteres de begin y end, calculamos la frecuencia de cada letra en cada palabra, y las almacenamos en 2 vectores. Luego, los restamos y devolvemos el máximo de la suma de los elementos positivos y negativos del vector resultante, en valor absoluto.

Tarea 4: Distancias cadena-trie:

Distancia de Damerau restringida cadena-trie con threshold
dp_restricted_damerau_trie(x, tri, threshold) en ***distances.py***
La implementación es similar al cadena-cadena mejorado, pero en este sustituimos la consulta del carácter por tri.get_label() y la consulta de valores anteriores en la columna con tri.get_parent().
Luego realizamos un bucle haciendo un recorrido del trie (usando un diccionario para guardar las palabras) y comprobando si es final, si lo es, comprobamos si es menor que el threshold y si no lo es no se considera el en el resultado final.

Distancia de levenshtein restringida cadena-trie con threshold
def dp_levenshtein_trie(str1, tr2, thres=2**31) en **distances.py**
Primero creamos la matriz donde se almacenan los resultados. Inicializamos el valor en (0,0) a 0, y comenzamos a iterar. Al igual que en las versiones sin trie de levenshtein, comprobamos si coinciden los caracteres, o sino, guardamos el menor de las operaciones de inserción, borrado, o reemplazo.

Finalmente, recorremos los nodos del trie, guardando las palabras en el diccionario dict, y si un nodo supera el threshold se descarta.
Los costes temporal y espacial son O(m*n).

Distancia intermedia cadena-trie con threshold:
dp_intermediate_damerau_trie(x, trie, threshold) en ***distances.py***
En esta distancia hemos variado un poco el método. Usamos 4 columnas para el cálculo, similar a como lo hacíamos antes, avanzamos las columnas en cada iteración sobre la palabra y calculamos la distancia de levenshtein normal. Luego dependiendo de qué tipo de operación se pueda aplicar como sabemos que la distancia de levenshtein <= distancia intermedia y que las operaciones son excluyentes simplemente nos podemos quedar con la última operación realizada. Luego volvemos a hacer la misma comprobación que en la distancia restringida cadena-trie, para quedarnos solo con los que cumplen el threshold.

Tarea 5: Realizar un estudio experimental de medidas de tiempo para determinar qué versiones de las anteriores son las más eficientes para unos datos concretos.

Main en ***spellsuggest.py***

A partir del código proporcionado para ordenar el vocabulario por frecuencia de palabra, se miden los tiempos para distintas variables: la palabra, tamaño de diccionario, el método de distancia y si se emplea Trie. Esto se ha implementado mediante diferentes bucles, diferenciando entre el bucle para el Trie y el no Trie. Para la medición de los tiempos hemos empleado la librería time, como recomienda el boletín. Por último, todas estas variables junto a sus tiempos son sacados por pantalla y asimismo recolectados en distintas listas para posteriormente crear un csv y emplearlo para gráficas de tiempo que nos den una idea más visual de los resultados de esta práctica.

Tarea 6: Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas utilizando TODAS las posibles implementaciones.

La primera modificación se ha realizado en el *indexer*, añadiendo el parámetro -G para hacer o no un trie del diccionario (para no volver a crearlo constantemente, esto no tiene ningún sobrecoste en el caso de que sea una búsqueda sin trie).
Luego añadimos los parámetros B, Z, y Tr, siendo ellos para la distancia de edición en la búsqueda (B) “levenshtein”, “intermediate” o “restricted” (Si no se indica ninguno, se presupone que no se quiere obtener una búsqueda de palabras similares). Threshold (Z) = 3 por defecto, y trie(Tr), False por defecto, que indica si se usa o no trie en las consultas.
Un usuario del mismo solo requiere conocer estos 4 parámetros.

Seguidamente, el principal cambio en la librería se ha llevado a cabo en el método *get_posting*.
En éste, se realizan las comprobaciones de si usamos trie o las distancias requeridas. Y luego, si no obtenemos ningún resultado, concatenamos una llamada para cada palabra sugerida del método suggest.

Una pequeña demo:

```
>python SAR_Indexer.py 2015 index_20151.bin -G
--------------------------------------------
Number of indexed days: 285
----------------------------------------
Number of indexed news: 803
----------------------------------------
TOKENS:
        # of tokens in 'article': 44684
----------------------------------------
Positional queries are NOT allowed
--------------------------------------------
Time indexing: 1.22s.
Time saving: 0.16s.
```

```
>python SAR_Searcher.py index_20151.bin -Q "alexanderx" -Z "3" -Tr False -B "levenshtein"
18
Query: alexanderx
Number of results: 18
```

```
>python SAR_Searcher.py index_20151.bin -Q "alexanderx" -Z "3" -Tr True -B "levenshtein"
18
Query: alexanderx
Number of results: 18
```

```
>python SAR_Searcher.py index_20151.bin -Q "alexanderx" -Z "3" -Tr False -B "intermediate"
18
Query: alexanderx
Number of results: 18
```

```
>python SAR_Searcher.py index_20151.bin -Q "alexanderx" -Z "3" -Tr False -B "restricted"
18
Query: alexanderx
Number of results: 18
```

MEDICIÓN DE TIEMPOS

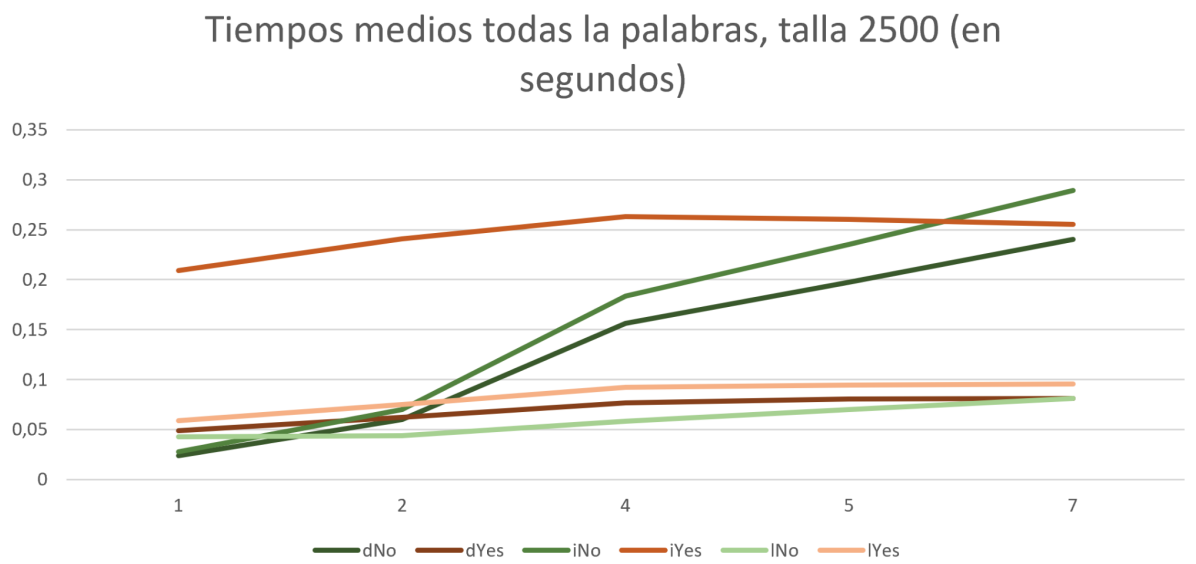
Hemos realizado ejecuciones para:

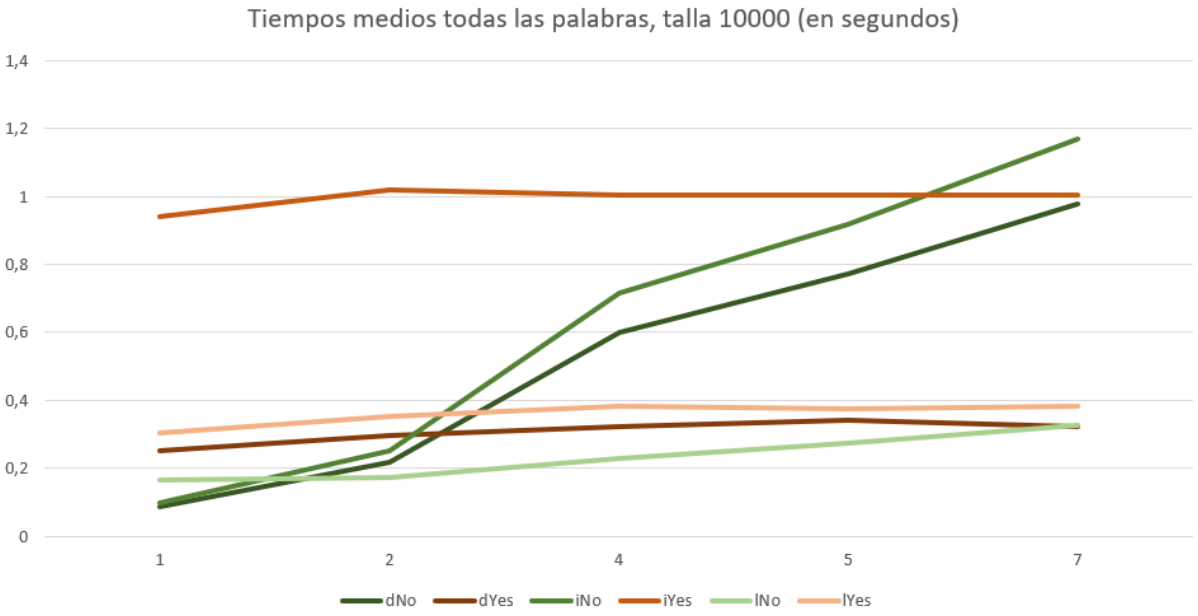
- Palabras: ["casa", "senor", "constitución", "ancho", "savaedra", "quixot", "s3afg4ew"]
- Tamaños de vocabulario: [2500, 10000, 35000]
- Thresholds: [1, 2, 4, 5, 7]
- Distancias: "intermediate", "restricted", "levenshtein"
- Con y sin trie.

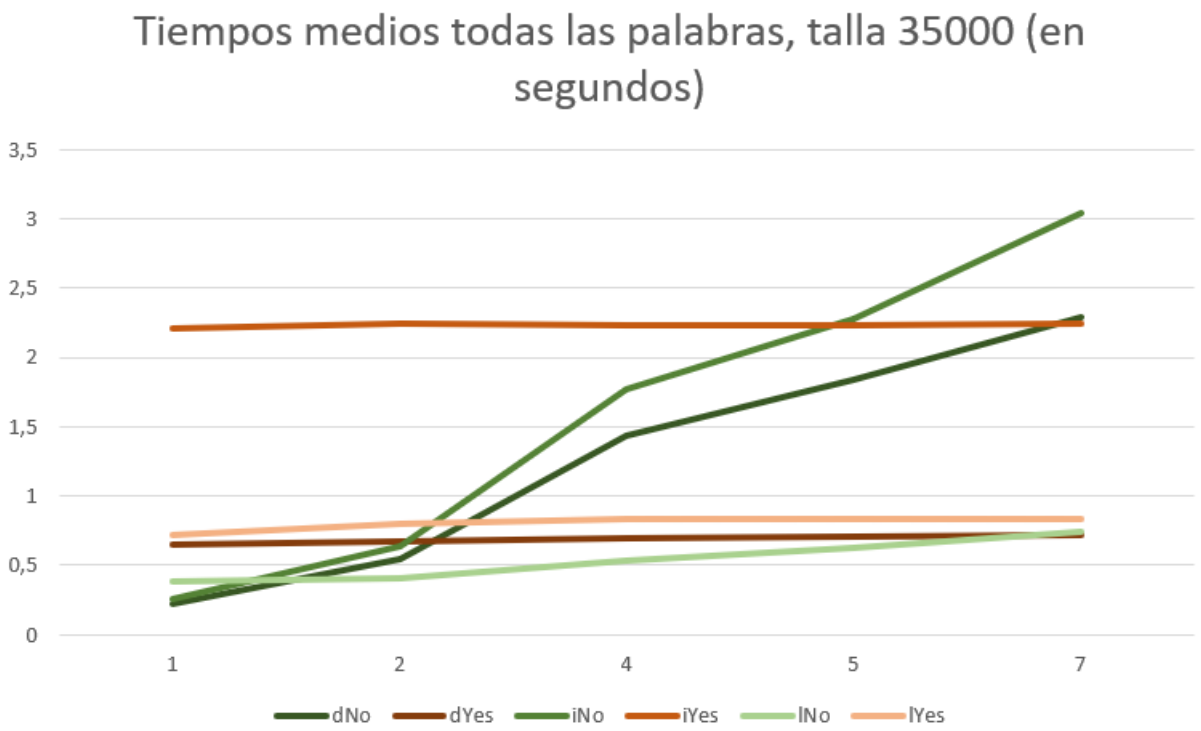
Generamos una agrupación para el tipo de distancia usada, si se ha empleado o no trie, el tamaño de vocabulario y el threshold. Seguidamente, calculamos la media de cada grupo, y realizamos gráficas, donde el eje de abscisas muestra el threshold, el de ordenadas la media para todas las palabras en segundos, empleando una distancia concreta, con o sin uso del trie. Hay 3 gráficas, una por cada talla de vocabulario seleccionada. Las ejecuciones con trie se muestran en tonos anaranjados-marrones y las que no lo usan, en verde. La leyenda es la siguiente:

- *dNo*: Damerau-Levenshtein sin trie.
- *dYes*: Damerau-Levenshtein con trie.
- *lNo*: Levenshtein sin trie.

- *lYes*: Levenshtein con trie.
- *iNo*: Damerau-Levenshtein Intermedia sin trie.
- *iYes*: Damerau-Levenshtein Intermedia con trie.







Según los resultados ilustrados en las gráficas, podemos observar que a thresholds bajos (1 y 2), todas las distancias dan tiempos más o menos similares, salvo damerau-levenshtein restringida con Trie que da un tiempo muy bajo. Asimismo, para valores altos (5 y 7), se dividen claramente en dos grupos diferenciados. Aquellos con valores altos son: levenshtein sin trie, damerau-levenshtein restringido sin Trie, damerau-Levenshtein intermedio sin trie. Y por otro lado los más óptimos, que serían damerau-levenshtein intermedia sin Trie, levenshtein con Trie, levenshtein sin Trie, damerau-levenshtein restringida con Trie. Todo esto tiene valor a la hora de ver que distancias utilizar, basadas en el threshold deseado.

Por tanto, concluimos indicando que para thresholds mayores a 3, sería más beneficioso emplear Trie de forma general.