Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

BÚSQUEDA APROXIMADA DE CADENAS (SAR-ALT)

Grau en Enginyeria Informàtica

Algorítmica

Iñaki Díez Lambies

Andrea Gascó Pau

José Miguel Martínez Martín

Mireia Pires State

Grupo: L1-4CO21

Índice

1. Decisiones de implementación

Para la parte obligatoria del proyecto, hemos realizado diferentes decisiones que nombraremos seguidamente respecto a los diferentes métodos, partiendo del método ya construido *levenshtein_matriz*.

1.1. Método elegir_operación

La implementación del método *elegir_operacion* tiene como objetivo devolver la diferente operación de *Sustitución, Inserción, Borrado, Intercambio, Intercambio de 3 a 2 e Intercambio de 2 a 3* que se realizan en el cálculo de la distancia.

Este método se usa tanto para los casos de la parte obligatoria como para los casos de las ampliaciones, siendo estos los casos de las tres últimas operaciones comentadas. Se llama en los métodos donde queremos devolver el camino recorrido, siendo estos *levenshtein_edicion, damerau_restricted_edicion y damerau_intermediated_edicion*.

Dependiendo de en qué método se llama, se insertan unos parámetros u otros, y por orden de prioridad de operaciones va comprobando cual se ha realizado, devolviendo posteriormente la operación realizada y los incrementos de los índices X e Y correspondiente.

1.2. Levenshtein edición

La implementación del método *levenshtein_edición* tiene como objetivo devolver la secuencia de las diferentes operaciones que se han realizado durante el cálculo de la distancia.

Esto se consigue en primer lugar, construyendo la matriz igual que en el método proporcionado, y seguidamente, una vez esta está construida, la recorremos en orden inverso. Mediante la llamada al método explicado anteriormente *elegir_operación* elegimos los diferentes casos que se pueden dar de *Inserción, Borrado y Sustitución*.

Una vez tenemos la operación correspondiente, la añadimos a la lista y reducimos los índices. En el caso de llegar a una pared, es decir, los primeros índices de la matriz, suponemos *Inserción* si el índice Y es mayor que 0, o *Borrado* si el índice X es mayor que 0. Finalmente, devolvemos la lista en el orden en que se han realizado las operaciones.

1.3. Levenshtein reducción

La implementación del método *levenshtein_reduccion* tiene como objetivo devolver la distancia entre dos palabras usando reducción espacial.

Esto se consigue, en vez de usar una matriz, con la creación de dos vectores del tamaño de la longitud de las palabras a comparar. Uno con la distancia actual y el otro con la anterior para poder realizar las comparaciones pertinentes.

En la primera columna solo tenemos en cuenta la *Sustitución* y el *Borrado*, debido a que el vector *vprev* todavía no se puede utilizar, mientras que en las demás podemos considerar los tres casos: *Inserción, Borrado y Sustitución*. Finalmente, pasamos los valores del vector *vcurrent* al vector *vprev* y devolvemos la distancia.

1.4. Levenshtein con cota

La implementación del método *levenshtein* tiene como objetivo devolver la distancia entre dos palabras usando reducción espacial y añadiendo un parámetro umbral para que, cuando se llegue a dicho umbral, se deje de calcular la distancia.

Partiendo del método *levenshtein_reducción,* calculamos la diferencia de las longitudes entre las dos palabras que tenemos que comparar para posteriormente poder añadir una condición donde, si la distancia a la que hemos llegado es mayor al *threshold* devolver el *threshold* + 1. En caso contrario, es decir, mientras la distancia no llega al umbral, se sigue calculando como en el método *levenshtein_reducción* y se devuelve el valor de la distancia.

1.5. Clase SpellSuggester

La implementación del método *suggest* de la clase *SpellSuggester* tiene como objetivo devolver una lista de listas donde la lista i-ésima contiene las palabras a distancia i (para i hasta distancia *threshold*). Sin embargo, si el atributo *flatten* es true simplemente devolverá una lista.

El método *suggest* recibe una cadena en su primer argumento obligatorio y opcionalmente el tipo de distancia, un *threshold* y un argumento *flatten*. Por defecto estos argumentos están inicializados a *levenshtein*, 3 y true, respectivamente.

En primer lugar, guardamos en una variable *selectedFunction* el método asociado a la distancia que se le pasa a *suggest*. Es decir, buscamos el valor de distancia entre las llaves del diccionario *opcionesSpell* que se encuentra en distancias.py.

A continuación, si el argumento *flatten* es true entonces para cada palabra existente en el vocabulario comprobamos que la distancia sea menor igual que el *threshold* y añadimos la palabra a la lista. Finalmente, devolvemos la lista.

Si por el contrario *flatten* es false, creamos una lista de listas donde habrá tantas listas como de grande sea el *threshold* + 1. Es decir, si el *threshold* es 3 entonces tendremos 4 sublistas. Realizamos el cálculo de la distancia entre la palabra que nos han pasado por argumentos y todas las palabras del vocabulario. Por último, según el resultado insertamos la palabra del vocabulario en la sublista correspondiente. Finalmente, devolvemos la lista de listas.

1.6. Modificación del indexador y recuperador de SAR

En primer lugar, se ha modificado el constructor de la clase *SAR_lib* donde se han añadido las variables *self.distance* y *self.threshold* y el objeto de la clase spellSuggester *self.speller*. Además, se ha añadido una función llamada *set_spelling* donde se le pasan los argumentos *use_spelling*, *distance* y *threshold*. *Use_spelling* será true si se activa la corrección ortográfica en las palabras no encontradas.

Por otro lado, en la parte del código relacionada con la indexación, concretamente en el método *index_dir,* se ha añadido al objeto *speller* después de la indexación el vocabulario mediante *extract_vocabulary()*. Este método, se ha creado con la intención de que devuelva una lista con todas las palabras que conforman el vocabulario.

Finalmente, en la parte del código relacionado con la recuperación de las noticias, concretamente en el método *get_posting()*. Se ha añadido la condición de que si no existe la palabra que se está buscando en el indexador, se haga una llamada al método *suggest()* para que devuelva una lista de palabras parecidas y que se puedan recuperar noticias.

2. Funcionalidades extra implementadas

2.1. Levenshtein con cota optimista

La implementación del método *levenshtein_optimista()* tiene como objetivo el uso de una cota para contar las ocurrencias de cada letra de las palabras a comparar, y aunque el orden no sea el mismo, se cuenta como si fuera correcto.

Esto se consigue con la creación de un diccionario, donde se recorre una de las palabras y se inserta cada letra que va apareciendo y el número de veces que está cada una mediante una suma. Después, se repite el procedimiento con la otra palabra y por cada ocurrencia de cada letra que ya estaba en el diccionario se realiza una resta.

Finalmente, si el resultado de esta operación es mayor que la cota establecida, se devuelve *threshold* + 1. En caso contrario, se hace la llamada a *levenshtein()* y se devuelve la distancia de forma normal.

2.2. Damerau-Levenshtein restricted matriz

La implementación del método *damerau_restricted_matriz* tiene como objetivo devolver la distancia entre dos palabras teniendo en cuenta la nueva operación que añadimos, es decir, el *Intercambio*.

El funcionamiento de este método es igual a *levenshtein_matriz,* con la particularidad de añadir la operación mencionada anteriormente, si se cumple la condición intercambio, es decir, este se realizará si nos encontramos una situación como, por ejemplo: ab -> ba.

2.3. Damerau-Levenshtein restricted edición

La implementación del método *damerau_restricted_edicion* tiene como objetivo devolver la secuencia de las diferentes operaciones que se han realizado durante el cálculo de la distancia.

Partiendo del método *damerau_restricted_matriz*, construimos la matriz para posteriormente, recorrerla en orden inverso y determinar mediante la llamada al método *elegir_operacio* qué operación se va a realizar. Las distancias de la nueva operación (intercambio) son inicializadas a valor igual a infinito si no se cumple la condición de intercambio.

Finalmente, se añaden las operaciones en una lista en el orden en que se han ido realizando, y se modifican los índices.

2.4. Damerau restricted

La implementación del método *damerau_restricted* tiene como objetivo reducir el coste espacial y utilizar un parámetro umbral, devolviendo la distancia entre dos palabras.

En este método se declararán 3 vectores para la reducción del coste espacial, los cuales van a ser, el vector actual, y dos vectores previos a este. El primero lo rellenaremos de ceros, el segundo lo declararemos con *arange* y el tercero lo declararemos a infinito.

Trabajaremos calculando el mínimo de distancias entre las posiciones de los vectores y los iremos alternando entre ellos. Se comprobará si la distancia que se ha obtenido es mayor al umbral, y si lo es, se devolverá *threshold* + 1. Finalmente, si noo es así, retornaremos la última posición del segundo array, siendo esta la distancia entre las dos palabras.

2.5. Damerau intermediated matriz

La implementación del método *damerau_intermediated_matriz* tiene como objetivo el cálculo de la distancia *damerau_intermediated* con el uso de una matriz de distancias.

Este método comparte gran similitud con su análogo *damerau_restricted_matriz* que, al igual que este, comparte estructura de almacenamiento, así como el camino que recorre para el cálculo de la distancia. La novedad que introduce son las nuevas operaciones de intercambio con eliminación e intercambio con borrado en su cálculo, así como en la recuperación del camino. En la recuperación, son añadidas como condiciones adicionales y si se cumplen sus condiciones se pasa a revisar si resultan en ser la distancia mínima.

2.6. Damerau intermediated edición

La implementación del método *damerau_intermediated_edición* tiene como objetivo recuperar el listado de operaciones realizadas que resultan en la distancia mínima de edición damerau-intermediated.

Este método comparte gran similitud con su análogo *damerau_restricted_edición*. La principal diferencia se encuentra en el cálculo de la distancia (que pasa a introducir las nuevas condiciones de *damerau_intermediated_matriz*) y las nuevas condiciones en la recuperación. Al igual que *damerau_restricted_edicion* encontramos que, en la recuperación del camino, si no se cumple la condición, entonces la distancia se iguala a infinito para que no sea seleccionada como solución.

2.7. Damerau intermediated

La implementación del método *damerau_intermediated* tiene como objetivo recuperar la distancia de edición Damerau-intermediated con la máxima eficiencia espacial posible.

Para esto se ha implementado una solución donde se utiliza un vector adicional necesario para el cálculo de los nuevos tipos de intercambio frente a *damerau_restricted* (intercambio con inserción y con borrado). Este nuevo vector es inicializado a infinito (por su inexistencia en las primeras fases del cálculo) y, al igual que el método análogo en matriz, se tiene en cuenta la posibilidad de estas operaciones como posibles cuando se cumple su condición.

3. Método de coordinación y contribución de los miembros

Elegir_operacion	Iñaki, Mireia
Levenshtein_edicion	Andrea, Iñaki, José Miguel, Mireia
Levenshtein_reduccion	Andrea, Iñaki
Levenshtein	Andrea, Iñaki
Clase spellSuggester	José Miguel, Mireia
Modificación indexador y recuperador SAR	José Miguel, Mireia

Levenshtein_cota_optimista	Andrea, Iñaki
Damerau_restricted_matriz	José Miguel, Mireia
Damerau_restricted_edicion	Andrea, Mireia
Damerau_restricted	Iñaki, José Miguel
Damerau_intermediate_matriz	Andrea, Iñaki, José Miguel, Mireia
Damerau_intermediate_edicion	Andrea, Iñaki, José Miguel, Mireia
Damerau_intermediate	Andrea, Iñaki, José Miguel, Mireia