Ejercicio 5: Corrector ortográfico

Trabajo realizado por Álvaro Clar Lopera y Andrés Ramos Seguí

Introducción

El objetivo de este ejercicio, el quinto en la asignatura de Algoritmos Avanzados, es el realizar un programa que, respetando el patrón MVC (Modelo Vista Controlador), el cual consiste en realizar un corrector ortográfico. Para ser más específicos, el programa consiste en un editor de texto sencillo, en el que se puede escribir texto, el cual será corregido para poder mostrar las palabras léxicamente mal escritas.

La solución desarrollada, además, permite la importación de un fichero de texto, el cual será añadido al editor de texto y analizado, mostrando las palabras mal escritas.

Para poder solucionar el problema, se utilizará un diccionario en castellano ofrecido por el profesor de la asignatura, Miquel Mascaró Portells, con el cual se podrá saber que palabras son correctas o incorrectas. Para saber dicha cuestión, se utilizará la programación dinámica como técnica de programación, haciendo uso en concreto del cálculo de la distancia Levenshtein, o lo que es lo mismo, la distancia de edición, la cual consiste en averiguar cuan parecidas son dos palabras entre sí, siendo esa distancia la cantidad de operaciones necesarias entre una y la otra. Esas operaciones son las siguientes:

- Inserción: Insertar un carácter en la palabra en una determinada posición.
- Reemplazamiento: Cambiar un carácter por otro en la palabra.
- Borrado: Eliminar un carácter de la palabra.

Estructura del ejercicio

El programa realizado en Java consiste en un conjunto de clases divididas en diferentes paquetes*:*

- **Levenshtein**: Este paquete contiene la clase principal, además de la clase que permite la comunicación con las diferentes partes del programa.
- **Controlador**: Contiene la parte del controlador, donde se ejecutan los algoritmos que permiten gestionar la corrección del texto y el cálculo de las distancias Levenshtein de las palabras escritas.
- **Gui**: Contiene las clases de la parte de la Vista, que son la ventana, el panel que contendrá el editor de texto, además de un segundo panel de texto que contendrá las palabras posibles para corregir una palabra.
- **Datos**: Contiene las clases que gestionan el Hashing que contiene el diccionario y el conjunto de palabras alternativas a una palabra que es incorrecta.
- **Imágenes**: Paquete que contiene las imágenes que son utilizadas en los botones de la ventana.

A continuación vamos a comentar cada una de las partes del programa por separado:

1. **Parte principal**

   Clase principal del programa, se encarga de comunicarse con las 3 partes restantes, modelo, vista y controlador. Es por eso que tiene 3 punteros a cada una de ellas, DiccionarioHash (los datos) , GUI (vista) y Calculadora y Corrector (parte del control).

   Además, implementa el método notificar, el cual permite dicha comunicación con las otras partes, mediante mensajes en formato String.

2. **Parte del modelo**

2.1 DiccionarioHash

Clase que contiene un HashMap con el diccionario en español. La razón de la implementación de esta clase se debe a la necesidad de guardar en una estructura que puede ser consultada posteriormente, que evita tener que realizar recorridos cada vez que se quiera saber si una palabra es correcta o no. De esta manera, para saber si una

palabra esta en el diccionario, simplemente se consulta si está con un coste O(1). Sus atributos más importantes son:

- **EOF:** Variable entera que señala el fin de archivo, en este caso del diccionario (-1).
- **Diccionario:** String con el nombre del fichero que se utilizará como diccionario.
- **File_reader, buffer:** Variables FileReader y BufferedReader que se utilizarán para leer el contenido del fichero.
- **Hash_diccionary:** HashMap <String,String> con clave-valor iguales, en concrento, cada palabra del diccionario, haciendo más fácil su consulta.

Los métodos más importantes son:

- **Constructor**: Abre el fichero diccionario y se dispone a leerlo (línea a línea, que es como está guardado en el fichero) y guarda cada palabra en el Hashing.
- **openFile, closeFile, readFromFile:** Métodos que encapsulan las llamadas a open, close y readLine que realizan los objetos FileReader y BufferedReader, respectivamente.

3 Parte de la Vista

3.1 GUI

Clase que integra los elementos gráficos de la ventana. Básicamente esta formada por elementos de las librerías awt y swing. Sus atributos son los siguientes:

- **vent:** JFrame que construye la ventana del programa
- **barra:** JToolBar, que contendrá los botones
- **contenedor:** JPanel que actuará como contenedor que contendrá la barra
- **central:** Puntero a PanelCentral, el cual contendrá el editor de texto.
- **Calculador:** Puntero a la clase que permite calcular las distancias de edición (Levenshtein) y obtener posibles candidatos para corregir.
- **prog:** Puntero que permite comunicarse con el programa principal.
- **x:** ancho de la ventana.
- **y:** alto de la ventana.

Los métodos que contiene son los siguientes:

- **crear**: Inicializa los elementos ya mencionados, establece el Layout necesario para poder distribuir los elementos correctamente: el JToolBar aparecerá en la parte superior de la ventana, y en el centro aparecerán los dos editores de texto, como ya se explicará en breve.
- **ponOpcion:** Añade a la barra un nuevo botón que será añadido al JToolBar. Este a su vez llamada un método llamado “makeNavigationButton", el cual se encargará de crear el nuevo botón con la imagen asociada.
- **visualizar**: Permite la visualización de la ventana, mediante el poner a true su estado de visualización (setVisible(true))
- **makeNavigationButton**: Permite la creación de un nuevo botón. Se obtiene la ruta que conduce a la imagen que se quiere utilizar para el botón, se le añaden textos alternativos, tooltips y un escuchador de acciones. Una vez obtenida la imagen esta se añade al botón, y finalmente este es retornado.
- **actionPerformed**: Actuador que se ejecuta cuando el botón es pulsado. Se recoge en forma de string el comando de acción, y este se envía al programa principal a través del método Notificar.
- **importText**: Permite que el texto del fichero importado se le pase al panel central, para poder añadirlo al editor de texto.
- **setCalculador:** Método que permite asignar el puntero a la clase que gestiona el cálculo de distancia, para que sus métodos puedan ser invocados a la hora de analizar el texto escrito.

3.2 PanelCentral

Clase que hereda de JPanel y que permite la escritura de texto y la corrección interactiva de aquellas palabras que sean incorrectas. Sus atributos son:

- **jtp:** Variable de tipo JTextPane que representa el editor de texto en el que se podrá escribir.
- **document:** Objeto de tipo DefaultStyledDocument, el cual permite definir un documento texto con diferentes estilos (fuentes, color…) para el JTextPane anterior.
- **scr:** Objeto de tipo JScrollPane, que permite definir un scroll sobre jtp (vertical en este caso).

- **jtp2:** Variable de tipo JTextPane que representa el espacio en el que se situarán las palabras alternativas para corregir una palabra en concreto.
- **document2:** Objeto de tipo DefaultStyledDocument, el cual permite definir un documento texto con diferentes estilos (fuentes, color…) para el JTextPane inmediatamente anterior.
- **scr2:** Objeto de tipo JScrollPane, que permite definir un scroll sobre jtp2 (vertical en este caso).
- **Correctas:** Objeto SimpleAttributeSet que permite definir un conjunto de atributos a un documento, en este caso este permite definir las palabras correctas, que saldrán en un color negro por defecto.
- **Erroneas:** Objeto SimpleAttributeSet, que al igual que al igual que en el caso anterior, define un conjunto de atributos (color, fuente…). En este, se define un color rojo, para diferenciar palabras erróneas de las que no lo son.
- **i:** Indice que sirve para iterar sobre cada uno de los caracteres del texto escrito en el editor.
- **t:** String que contiene el texto escrito actualmente en el fichero.
- **Alternativas:** Array de Strings que contendrá palabras alternativas a una determinada palabra mal escrita. (10 palabras)
- **Calculador:** Contiene el objeto Calculador, que permite invocar a las funciones para calcular las distancias Levenshtein.
- **actualWord:** palabra actualmente seleccionada con el cursor.
- **actualDot:** marca la posición dot de la selección más reciente del editor de texto.
- **actualMark:** marca la posición mark de la selección más reciente del editor de texto.

Los métodos más importantes son:

- **Constructor:** Inicializa los documentos ya mencionados y se cren los dos JTextPane a partir de esos documentos. Además, se inicializan los dos AttributeSet, “correctas” y “erróneas”, definiéndose sus atributos.

  Para cada JTextPane, se crea un CaretListener, el cual añade un método CaretUpdate, el cual permite actuar ante el cambio del cursor del texto. En este caso, se van a detectar palabras marcadas con el cursor, así que se utilizarán dos elementos:

  1. Dot: indica la posición actual del cursor
  2. Mark: indica la posición donde finaliza la selección. Es decir, si se ha marcado una palabra, se tienen mark y dot como índice inicial y final de la selección.

  En el primer CaretListener, el de jtp, el método caretUpdate obtiene las posiciones dot y mark, y después, se obtiene la palabra acotada entre esas dos posiciones y se comprueba si esa palabra está escrita de color rojo (mirando si su AtributteSet es igual a “erróneas”). Sí es así, entonces se procederá a calcular las distancias Levenshtein respecto a las palabras del diccionario, obteniendo las 10 palabras más parecidas a esa. A partir de este punto, se escriben las palabras alternativas en el segundo JTextPane. Como opción extra, también se añade una palabra especial llamada “Ignorar_todas”, la cual permite dejar como correcta la palabra original, aunque no exista en el diccionario.

  En el segundo CaretListener, el de jtp2, el método caretUpdate también obtiene la posiciones dot y mark. En este caso, se comprueba que palabra de las mostradas como alternativas se ha seleccionado, y se reemplaza en el texto original (se reemplaza en la posición actual debido a que se guardan esas posiciones)

- **setText**: método que recibe un string con el texto del fichero importado, para que sea recogido por document, haciendo un insertString. Además el texto importado, se analiza llamando a traverseText
- **getText**: método que permite obtener el string con el texto escrito en el editor.
- **traverseText**: método que permite recorrer todo el documento de texto. Primero se obtiene el texto actual, y luego se procede a leer carácter a carácter, buscando palabras en este (getPalabra). Por cada palabra encontrada, se comprueba si está o no en el diccionario, llamando al método del calculador (pertenece_diccionario). Si está, se marca como correcta, y en caso contrario, como incorrecta (color rojo).
- **getPalabra**: método que recibe un string, y que

con ayuda de la varible global “i”, vamos obteniendo el string formado por los caracteres que van desde un índice i inicial, hasta un índice final, obteniéndose así la palabra.

- **IsSingleWord**: método que recibe un string, y comprueba si es una sola palabra o no. A este método se le llama debido a que en caso de que el usuario seleccione con el cursor más de una palabra, pues el programa no se disponga a calcular ninguna distancia.
- **setCalculador**: recibe por parámetro el objeto Calculador para así poder llamar a sus métodos relacionados con el diccionario y el cálculo de distancias Levenshtein.

3.3 SeleccionFichero

Clase que hereda de JPanel que permite gestionar la selección de fichero. Debido a eso, se gestiona en ella un JFileChooser, el cual permite elegir cualquier fichero del sistema. Sus atributos son:

- **prog:** Permite comunicar este panel con el programa principal.

Los métodos que contiene son:

- **Constructor:** Se le pasa el puntero del programa principal a prog e inicializa el objeto JFileChooser. Este último permite abrir el selector de archivos, que se abre por defecto en el directorio ráiz del proyecto Netbeans, para permitir seleccionar un archivo.
- **actionPerformed:** Actuador que detecta el pulsado de intro, que permite el envío de información al programa principal, y en este caso en concreto, el nombre del fichero.

4 Parte del Control

En esta ocasión, la parte del control está formada por dos clases, la clase Calculador, y la clase Corrector. A continuación se verán con más detalle en la solución del problema.

Solución del problema

Levenshtein:

Para el cálculo de las distancias entre palabras, utilizamos una tabla de distancias (tabla_distancias) y la vamos rellenando siguiendo el siguiente algoritmo:

Inicializamos la primera fila y la primera columna a la posición i.

Realizamos un recorrido transversal por toda la tabla rellenándola, en función del siguiente algoritmo.

El elemento i,j de la tabla será el resultado de calcular el mínimo entre los siguientes elementos:

El mínimo entre la columna anterior y misma fila, fila anterior misma columna y diagonal.

El coste valdrá 0 si la posición i de la primera palabra y la posición j de la segunda palabra corresponden a la misma letra.

- El elemento de la misma columna pero fila anterior más uno (añadir letra).
- El elemento de la misma fila pero columna anterior más uno (eliminar letra).
- El elemento de la columna y fila anteriores más coste (modificar letra).

La distancia mínima resultante entre las dos palabras, estará almacenada en el elemento de la última columna y última fila.

Cálculo de los candidatos:

Para el cálculo de candidatos, utilizamos la función levenshtein_diccionario, la cual consiste un una búsqueda de palabras por todo el hashing (de palabras del diccionario) de las palabras con menos distancia respecto de la palabra escrita. Comenzamos estableciendo la distancia_actual igual a 1, y buscamos las palabras con distancia 1 respecto de la palabra. Si no existen 10 palabras de distancia 1, aumentamos la distancia_actual y volvemos a realizar un recorrido por el diccionario.

Retornamos un array con 10 candidatos, los cuales se imprimirán en el jtp2 (JTextPane de Alternativas), tras clickar sobre una palabra errónea.

Corrección interactiva:

Para realizar la espera interactiva con el usuario nos basamos en el siguiente sistema:

Mediante un hilo concurrente (que se inicializa en el

proceso init()) que realiza una espera activa, vamos controlando que hace más de 0,5 segundos des de la última vez que se tecleo algún key y que aún se haya marcado las palabras erróneas. Si se dan las dos condiciones, indicamos que ya se han marcado las palabras (already_corrected = true) y llamamos a la función traverseText(), que se encarga de recorrer el contenido del JTextPane marcando las palabras incorrectas en rojo.

Utilizamos la clase Corrector, que contiene un puntero a la interfaz de usuario. El JTextPane de donde se escriben las palabras contiene un keyListener, que permite saber cuando se hace click sobre alguna tecla, esto permitirá saber cuando fue la última vez que se pulsó una tecla y registrar que ya se han realizado cambios en el JTextPane (already_corrected = false).

Almacenamiento del diccionario mediante un hashing:

Para evitar tener que accede a memoria secundaria cada vez que se requiere una palabra del diccionario, tenemos la clase DiccionarioHash que se encargará de gestionar el almacenamiento del diccionario en un Hashing, por lo que se permitirá accede a las palabras con un coste de O(1).

Para guardar el fichero en el hashing en el constructor de la clase DiccionarioHash se irá leyendo cada palabra del fichero, y la se irán guardando dentro del hashing.

Se podrá realizar un recorrido por el hashing mediante las siguientes instrucciones:

For (Map.Entry<String, String> entry : hash_diccionario.getHash_diccionary().entry()) { }

Existencia de la palabra en el diccionario:

Para agilitar el proceso y saber si una palabra existe dentro del diccionario (distancia 0 con el algoritmo de Levenshtein), se utiliza un hashing de clave string y valor string (la misma palabra). Esto evita tener que realizar un recorrido por todas las palabras del hashing y calculando su levenshtein. Para saber si una palabra existe, basta con mirar si se contiene la palabra buscada como clave dentro del hashing O(1).

Selección de fichero:

Utilizando el botón “Elige el fichero a editar” en la parte superior de la interfaz, se nos habilita la opción de poder escoger un fichero. Con el puntero a la interfaz importaremos el contenido del fichero al jtp del Panel centrar. Al añadir el contenido se habrá llamado a la función traverseText, que marcará el texto en función de si son palabras correctas o erróneas.

Manual de usuario

A continuación se explicará el funcionamiento del programa y como se debe utilizar. Inicialmente, se presentará una situación como la siguiente:

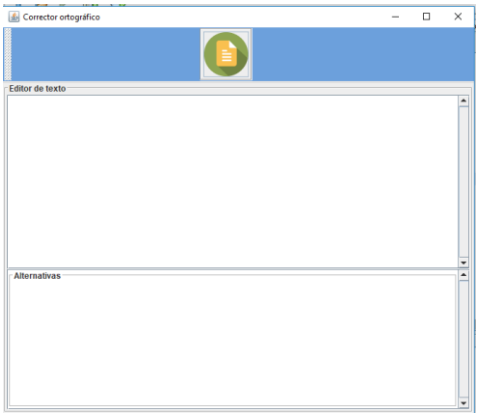


Figura 1. Aspecto inicial de la ejecución

A partir de entonces se pueden realizar diversas operaciones. Una de ellas, la más sencilla, es la de escribir en el panel de arriba, que es el editor de texto.

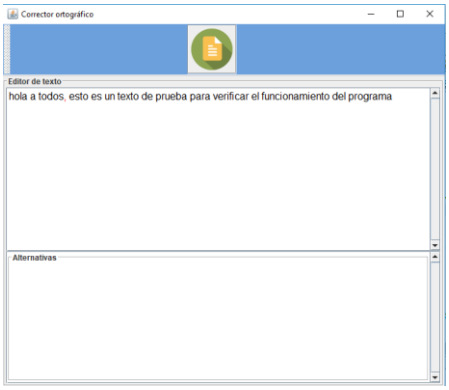


Figura 2: Escritura de texto en el editor de texto, en este caso sin errores ortográficos.

Sin embargo, como es lógico y probable, un documento se redactará con algunas faltas ortográficas, con lo cual, si la

situación se produce, tendremos algo como lo siguiente:

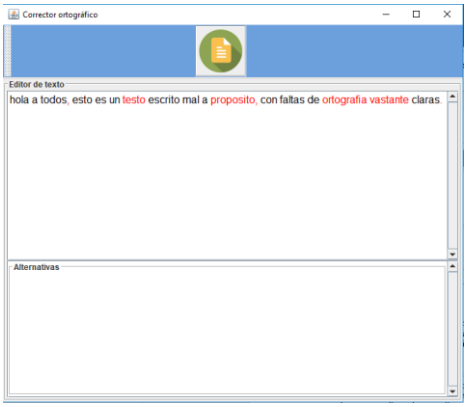


Figura 3: En este caso, se tiene un texto ya analizado por el programa con faltas de ortografía (palabras incorrectas marcadas en rojo)

Para poder corregir una palabra marcada en rojo, tan solo se debe hacer doble click en ella, con lo cual se obtendrán y mostrarán las palabras más parecidas a la que se está queriendo corregir:

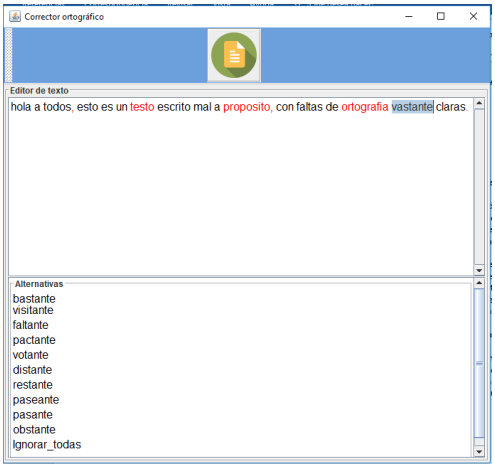


Figura 4: Se muestra la lista de palabras alternativas en la sección de abajo, entre las cuales además se ve la opción "Ignorar_todas", que permite que no haya ningún reemplazamiento.

Si elegimos alguna de estas opciones, se reemplazará el texto con la palabra seleccionada y dejará de ser incorrecta:

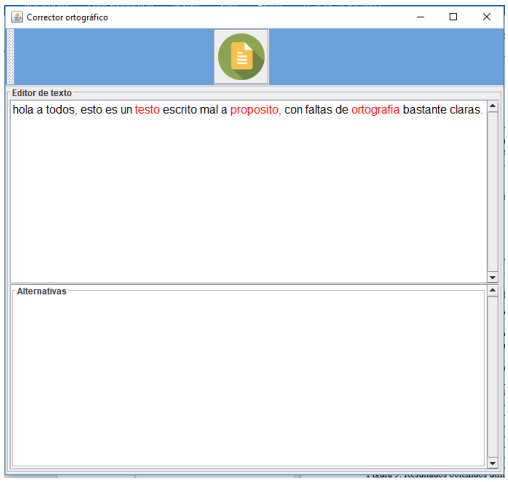
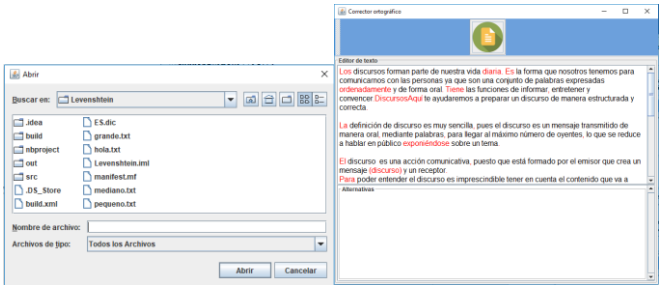


Figura: Se muestra el resultado de seleccionar una de las palabras alternativas, en este caso "bastante", y esta acaba siendo reemplazada en el texto original. Si se opta por "Ignorar_todas", se deja la palabra original como correcta (aún en ese caso volvería a ser detectada como errónea en una futura corrección)

Como una de las opciones extras añadidas, se tiene la posibilidad de importar el texto de un fichero de texto ya escrito. Esto se hace pulsando el botón de la parte de arriba de la ventana:



Figuras : Al seleccionar la opción se abre el selector de fichero, y al abrir, en este caso el fichero "mediano.txt", muestra el texto en el editor, mostrándose ya analizado y listo para poder corregir

Conclusiones

Mediante la realización de este ejercicio se ha podido observer el hecho de que la solución realizada con programación dinámica reduce el coste computacional asintótico a cambio de utilizar una tabla de cálculos previos. El hecho de haber tomado la decision de tener dicha tabla, supone aumentar el coste especial. Sin embargo, En este ejercicio no supone una gran carga

especial, puesto que se trata de una tabla de (n * m * ocupación de un integer) Bytes.

Además podemos ver que también se cumple el principio de optimalidad de Bellman mediante el uso de la tabla.

Agradecimientos

Gracias al Dr Miquel Mascaró Portells por la aportación de conocimiento sobre las técnicas de desarrollo del MVC, Programación Dinámica y otras ideas explicadas que han ayudado al desarrollo de este ejercicio.

Bibliografía y referencias

1. Apuntes de la asignatura Algoritmos Avanzados impartida por el Dr. Miquel Mascaró Portells: “Introducció a la programació dinámica” y “Diapositives de la semana del 8 d’abril”.
2. https://www.geeksforgeeks.org/edit-distance-dp-5/ (último acceso, 8 de mayo de 2019)
3. https://es.wikipedia.org/wiki/Distancia_de_Levenshtein (último acceso, 8 de mayo de 2019)
4. https://www.tutorialspoint.com/cplusplus-program-to-implement-levenshtein-distance-computing-algorithm (último acceso, 10 de mayo de 2019)