Capítulo 5

Jonathan Zinzan Salisbury Vega
Universitat de les Illes Balears
Palma, España
jonathan.salisbury1@estudiant.uib.cat

Joan Sansó Pericás
Universitat de les Illes Balears
Palma, España
joan.sanso4@estudiant.uib.cat

Joan Vilella Candia
Universitat de les Illes Balears
Palma, España
joan.vilella1@estudiant.uib.cat

La identificación de idiomas y corrección ortográfica de textos es una de las herramientas que más utilizamos en nuestro día a día. Es por eso por lo que fue tan importante la invención de la métrica de la "Distancia-Levenshtein" la cual nos permite medir de una forma cuantitativa lo mucho que se parecen dos palabras entre sí. Para mejorar el rendimiento de dicho algoritmo, se implementa mediante el paradigma de la programación dinámica. Este paradigma permite mejorar en gran medida el rendimiento de los algoritmos a costa de almacenar en memoria resultados parciales, los cuales son utilizados en futuros cálculos. En esta práctica se busca aplicar todos estos conocimientos teniendo en cuenta el patrón de diseño del MVC, el cual permite que los proyectos sean mucho más ordenados y legibles, además de permitir una mayor escalabilidad.

Keywords—Distancia Levenshtein, patrón MVC, lingüística.

I. INTRODUCCIÓN

La programación dinámica es una técnica algorítmica para la resolución de problemas de optimización. Para ello, se divide el problema en subproblemas más sencillos, aprovechando así el hecho de que la solución óptima del problema mayor estará formada por una secuencia óptima de decisiones de los subproblemas (Principio de optimalidad de *Bellman*).

A priori puede parecer que no exista ninguna diferencia con algoritmos previamente vistos en la asignatura. Uno con los que guarda gran similitud son con los del tipo "Divide y Vencerás". Ahora bien, los algoritmos D&C tenían una restricción muy grande, sus subproblemas deben ser disjuntos. La programación dinámica no tiene dicha restricción y, de hecho, aprovecha este solapamiento entre subproblemas para mejorar la complejidad computacional asintótica de los algoritmos. Este mejor desempeño no se logra a costa de nada, sino que se deberá realizar un desembolso de memoria importante que se utilizará para almacenar las soluciones parciales ya calculadas. Dichas soluciones se aprovecharán en cálculos futuros, ahorrando así tiempo.

Este nuevo paradigma de programación se utilizará para implementar un corrector ortográfico. El algoritmo a implementar, y por ende optimizar, será el de la "Distancia de *Levenshtein*". La distawncia de *Levenshtein* es una métrica que evalúa el número mínimo de operaciones para pasar de una cadena de caracteres a otra. Las operaciones permitidas son:

- Inserciones
- Borrados
- Sustituciones

Gracias a este algoritmo se implementará un corrector ortográfico. El cual una vez haya analizado el idioma introducido, arrojará distintas recomendaciones sobre las palabras que se han detectado incorrectas. Las correcciones irán ordenadas de menor a mayor distancia, es decir, de más parecidas a menos. El usuario podrá seleccionar una de las propuestas y realizar el cambio. Finalmente, se podrá guardar la corrección en un nuevo archivo de salida.

II. CONTEXTO Y ENTORNO DE ESTUDIO

Uno de los requisitos a la hora de realizar la práctica es el uso del lenguaje de programación Java. Además, se ha dado la opción de elegir entre varios IDE's (*Integrated Development Environment*).

- NetBeans
- IntelliJ
- VS Code

En este caso se ha escogido el IDE de NetBeans por familiaridad de uso. Además, se utiliza una herramienta de control de versiones (Git). Más específicamente su versión de escritorio *Github Desktop* por su facilidad de uso mediante interfaz gráfica.

III. DESCPRIPCIÓN DEL PROBLEMA

La práctica se podría dividir en dos problemas a resolver, por una parte, está la detección del idioma introducido y por otra, la implementación del algoritmo de *Levenshtein* el cual permitirá realizar las sugerencias.

La detección del idioma se podría considerar un problema menor en comparación a la otra parte de la práctica, ya que es la que menos tiempo se ha necesitado para implementar. El acercamiento ha sido bastante sencillo en el que se ha analizado el texto completo y comparado el número de palabras que contenía con los distintos diccionarios (inglés, castellano y catalán). Una vez obtenido dichos valores, se elegía la lengua con mayor número de palabras coincidentes. Ahora bien, esta solución presenta un problema para textos considerablemente grandes, ya que se guarda todo en memoria. Una posible mejora sería analizar a partir de cuantas palabras se realiza una distinción de idiomas suficientemente buena. Estos aspectos se tratarán más adelante.

No obstante, es importante entender en primer lugar cómo funciona el algoritmo dentro de su marco teórico, para más adelante explicar la implementación realizada.

A. *Levenshtein*

En primer lugar, se dará la definición matemática de la distancia *Levenshtein*.

"Sean dos palabras a y b (de longitud |a| y |b| respectivamente) se define la función $lev_{a,b}(|a|,|b|)$ *donde":*

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Ilustración 1

Donde $1(a_i \neq b_i)$ es la función indicadora igual a 0 cuando $a_i \neq b_i$ e igual a 1. Y la función es la distancia entre los primeros i caracteres de a y los primeros j caracteres de b.

B. *Ejemplo de cálculo*

A continuación, se ejemplificará un caso con las siguientes palabras:

- Sitting (String a)
- Kitten (String b)

A continuación, se seguirá el proceso para rellenar la matriz.

En la primera iteración, los valores de i y j serán (1,1). Ahora, siguiendo la fórmula, el primer paso es determinar en qué parte de la ecuación se está. Como $min(1,1) \neq 0$, se deberá operar con la parte del mínimo. Por lo que se obtiene:

$$lev_{a,b}(1,1) = \{min \begin{cases} lev_{a,b}(0,1)+1 = 1+1 = 2 \\ lev_{a,b}(1,0)+1 = 1+1 = 2 \\ lev_{a,b}(0,0)+1_{(if\ a_1 \neq b_1)} = 0+1 = 1 \end{cases}$$

$$lev_{a,b}(1,1) = \min \begin{cases} 1 \\ 2 = 1 \\ 1 \end{cases}$$

Por lo que la tabla quedaría de la siguiente manera:

	#	K	I	T	T	E	N
#	0	1	2	3	4	5	6
S	1	1					
I	2						
T	3						
T	4						
I	5						
N	6						
G	7						

Tabla 1

Ahora, de manera análoga se iría rellenando el resto de la tabla, quedando como resultado la siguiente:

	#	K	I	T	T	E	N
#	0	1	2	3	4	5	6
S	1	1	2	3	4	5	6
I	2	2	1	2	3	4	5
T	3	3	2	1	2	3	4
T	4	4	3	2	1	2	3
I	5	5	4	3	2	2	3
N	6	6	5	4	3	3	2
G	7	7	6	5	4	4	3

Tabla 2

La solución queda almacenada en la esquina inferior derecha, por lo que la "Distancia Levenshtein" es de 3. Esto se puede comprobar fácilmente teniendo en cuenta los siguientes pasos:

- Kitten → Sitten (Sustitución de "s" por "k")
- Sitten → Sittin (Sustitución de "i" por "e")
- Sittin → Sitting (Inserción de "g" al final)

Es importante tener en cuenta que, para este ejemplo, los pesos de las distintas operaciones son de 1. Dependiendo de la implementación que se haga, pueden variar dichos pesos.

C. *Programación dinámica aplicada*

Lo importante del algoritmo es que no es la implementación recursiva, sino la implementación dinámica. Es decir, se ha usado una estructura (una matriz) donde se han ido guardando los resultados parciales, que han sido usados para pasos futuros en el algoritmo. Esto lo podemos hacer ya que sabemos que las palabras más largas no pasan de, más o menos, 25 caracteres, así que como máximo tendremos una matriz de 25x25 caracteres, que serían más o menos 1KB de memoria si suponemos que los caracteres en Java ocupan 2 bytes (caracteres UNICODE).

Así que, simplemente por usar 1KB de memoria, se gana el tiempo que se pierde usando algoritmos recursivos (que, si bien se recuerda, son lentos ya que en cada llamada se tienen que guardar todas las variables en el *heap*.

D. *Aplicaciones*

Los usos de la distancia Levenshtein están muy relacionados con la lingüística (por no decir casi todos), esto es debido a sus limitaciones. Ya que el costo de calcular cadenas más largas es inviable, pues que es aproximadamente el producto de las longitudes de ambas cadenas.

Ahora bien, dentro del ámbito de la lingüística tiene multitud de aplicaciones. Correctores ortográficos (implementado en esta práctica), sistemas de corrección para el reconocimiento óptico de caracteres y software para la traducción del lenguaje natural basado en la memoria de traducción. Además, se utiliza como métrica para cuantificar la distancia lingüística que hay entre dos idiomas, esto se denomina, inteligibilidad mutua.

IV. SOLUCIÓN PROPUESTA

A continuación, explicaremos la parte principal del programa usado para solucionar el problema. Esta consiste en dos funciones principales, la función *correct* y la función *levenshtein*.

- *correct()*:
  esta función recibe como parámetros la lista de palabras saneadas del texto a corregir (es decir, sin símbolos de puntuación, etc). Primero se analiza en qué idioma está escrito el texto mediante el método auxiliar *detectLang*, explicado más adelante. Una se tiene el diccionario correcto, hacemos los siguientes cálculos para cada palabra de la lista:

  1. Para cada palabra del diccionario, se calcula su distancia de levenshtein con la palabra actual.
  2. Si esta distancia es zero, sabemos que la palabra es correcta, ya que se encuentra en el diccionario, así que salimos del bucle para esta palabra.
  3. Si la distancia no es zero, y es igual a la menor distancia encontrada hasta ahora, la palabra del diccionario se añade a la lista de sugerencias de la palabra que estamos tratando ahora.
  4. En cambio, si la distancia es menor a la menor distancia encontrada hasta ese momento, se vacían las sugerencias y se añade esta palabra, ya que se ha encontrado una distancia menor, lo que significa que probablemente será una mejor opción.
  5. Al terminar, si una palabra es incorrecta, también se añade como tal a la lista de sugerencias, permitiendo así al usuario elegirla por si hay un error en el diccionario y la palabra realmente está escrita, así como se quiere.

  Finalmente, se devuelve la lista de palabras incorrectas con sus sugerencias (la estructura usada es un Hash donde la palabra incorrecta es la Clave, y el valor es una lista de sugerencias).

- *levenshtein()*:
  En este método se aplica el algoritmo de *levenshtein* explicado anteriormente. Devuelve un entero que es la distancia encontrada.

El proceso es sencillo: creamos una matriz que tendrá como dimensiones uno más que la longitud de las dos palabras. Luego, inicializamos la primera columna y fila con los índices, empezando por cero.
El cálculo del resto de las casillas se hace como se ha explicado anteriormente: se elige el mínimo de entre estos tres casos:

- La casilla diagonalmente superior, sumando uno si los caracteres de esos índices son distintos, sumando zero si son iguales.
- El valor de la casilla inmediatamente a la izquierda de la posición actual, más uno (que seria el caso de que eliminásemos un carácter).
- La casilla inmediatamente encima de la posición actual, más uno (que sería el caso de que insertásemos una letra).

Finalmente, devolvemos la última casilla, que contendrá la distancia de Levenshtein de las dos palabras.

V. PATRÓN MVC

El Modelo-Vista-Controlador (MVC) [1] es un patrón de diseño de software[1] en el que se divide la lógica del programa de su representación gráfica, además se hace uso de un controlador para los eventos y comunicaciones entre las distintas partes.

Este patrón de arquitectura se basa en las ideas de reutilización de código y la separación de conceptos distintos, estas características pretenden facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. Para ello se proponen la construcción de tres componentes:

- Modelo:
  Es donde se almacena la información necesaria para la ejecución de la aplicación. Gestiona el acceso a dicha información mediante peticiones a través del controlador. También contiene los procedimientos lógicos que hagan uso de esa información.

- Vista:
  Contiene el código que muestra la aplicación, es decir, que va a producir la visualización tanto de la interfaz del usuario como de los resultados

- Controlador:
  Responde a eventos usualmente generados por el usuario y los convierte en comandos para la vista o modelo. Se podría decir que hace la función de intermediario entre la vista y el modelo encargándose de la lógica del programa.

Como se ha mencionado anteriormente se ha decidido utilizar este patrón debido a la facilidad que aporta al programar la separación de conceptos. Además, se obtiene una capa más de abstracción ya que es posible utilizar diferentes vistas con un mismo modelo.

Las ventajas principales del MVC son: Escalabilidad, facilidad de tratamiento de errores y reutilización de componentes. Existen otras ventajas, pero esta arquitectura se aprovecha en mayor medida en aplicaciones web, el cual no es este caso.

La principal desventaja que existe del patrón MVC es la complejidad que añade a la programación. Ya que, para el mismo problema, hay que modificar el acercamiento para que quepa dentro de esta arquitectura. Lo que implica una mayor complejidad.

Hoy en día, es muy común el uso de la programación orientada a objetos (POO) como paradigma principal, por ello cada componente del patrón MVC suele implementarse como una clase independiente. A continuación, se explica que proceso se ha seguido y como se ha implementado cada parte.

A. Implementacion del Modelo

Para la implementación se ha creado la clase *Model* que contiene todos los métodos necesarios para la solución propuesta. En primer lugar, se ha definido un *Enum*[2] que contiene todos los lenguajes soportados por la aplicación y son los siguientes:

- Inglés
- Español
- Catalán

Para cada uno de estos lenguajes hay un archivo que contiene el diccionario en la carpeta */dics* del proyecto con su respectivo nombre.

Dentro del modelo además hay definidos algunos métodos auxiliares que son los siguientes:

[1] Un patrón de diseño es un conjunto de técnicas utilizadas para resolver problemas comunes en el desarrollo de software.
[2] Es un tipo de datos formado por diferentes valores constantes.

- *readDict()*:
  Método que, dado un lenguaje por parámetro, lee el archivo que contiene dicho diccionario y lo retorna como un *arrayList* de *Strings* que contiene todas las palabras.

- *detectLang():*
  Método que dado un *array* de palabras como parámetro detecta el lenguaje del texto. Esto se realiza comprobando por cada diccionario disponible cual contiene más palabras correctas del texto. Finalmente se devuelve el lenguaje detectado.

A continuación hay definido el método que contiene el algoritmo de la solución propuesta que se ha explicado anteriormente así como los métodos auxiliares *levenshtein()* y *min()* que se usan para encontrar la solución.

Cabe destacar que también se ha definido una clase abstracta llamada *AbstractModel* que es extendida por el modelo final. Esta clase se ha creado para poder implementar el sistema de *PropertyChange* y que el controlador pueda obtener información de los avances del modelo mientras realiza los cálculos. Esta funcionalidad solo se utiliza para actualizar el progreso de la solución y poder ir actualizando la barra de progreso de la vista.

B. Implementacion de la Vista

Para la implementación se ha creado la clase *View* que contiene todos los métodos necesarios para mostrar los datos al usuario de una manera agradable, y recibir *input* por parte de este.

En primer lugar, hay definidas un gran número de variables, tanto para la lógica como componentes *Swing*. Dentro del constructor se inicializan las variables normales, al final se realiza una llamada al método *initComponents()* que se encarga de inicializar y configurar los distintos componentes *Swing*, a su misma vez este método realiza una llamada a otro método auxiliar, *addComponents()* que se encarga de crear el *layout* de la aplicación y colocar todos los componentes en su posición correcta.

A continuación, se explica la interfaz de los distintos métodos públicos definidos en esta clase:

- *addListeners():*
  Este método se encarga de añadir el *listener* pasado como parámetro a los botones de la vista. En este caso únicamente se añade al botón de *check*.

- *getText():*
  Este método retorna el texto que el usuario haya introducido en el *textArea* de *input*. El valor retornado es un *array* que contiene todas las palabras del texto, excluyendo espacios y signos de puntuación.

- *showResults():*
  Este método recibe como parámetro un *hashmap* que contiene como llave las palabras que estaban mal escritas del texto, y como elemento un *arrayList* que contiene las sugerencias para cada palabra. El método se encarga de mostrar en el *textPane* el texto con las palabras incorrectas de color rojo. Además, configura la lista de palabras incorrectas para mostrarlas, así como su lista de sugerencias. También inicializa unas variables globales que se utilizaran posteriormente para poder reemplazar las palabras correctamente.

- *setProgress():*
  Este método recibe como parámetro un entero que representa el progreso actual, este valor se actualiza directamente a la barra de progreso.

A parte de estos métodos públicos, se han definido una serie de métodos privados que realizan funciones auxiliares o actúan como *listeners* internos para el resto de los botones.

En primer lugar tenemos los métodos *selectFile()* y *loadFileContent()* que sirven para cargar el texto de un archivo al *textArea* de *input* cuando el usuario haga click en el botón *"load File"*. Por otra parte el método *saveFile()* se ejecutara cuando el usuario utilice el botón correspondiente y se encarga de almacenar el texto corregido en el archivo seleccionado.

Por ultimo tenemos dos métodos, el primero es *checkWord()* que se ejecuta cuando el usuario presiona alguno de los botones *"next"* o *"previous"* y se encarga de ciclar sobre la lista de palabras erróneas. Además, altera algunas variables de la clase que luego nos permiten realizar la sustitución correctamente. El segundo es el *correctWord()* que se ejecuta al presionar el botón *"correct"* y obtiene el valor seleccionado de la lista de sugerencias y lo sustituye por la palabra errónea seleccionada en ese momento. Este último método ha sido un poco difícil de llevar a cabo ya que hay que tener en cuenta el caso en el que se repitan muchas veces las mismas palabras o estas sean *substrings* de otras. Para

ello se ha hecho uso de expresiones regulares [3]que han permitido obtener el resultado esperado.

Para implementar la funcionalidad de pasar a la palabra anterior o siguiente, ha surgido el problema de las palabras repetidas o las palabras anidadas dentro de otras palabras. Debido a esto, para buscar una palabra en el *input* de texto no basta con buscar el índice, ya que tienes que saber cómo tratar si la palabra sale más de una vez, o si esta secuencia de letras aparece dentro de una palabra más grande. Por ende, se ha elegido la siguiente estructura:

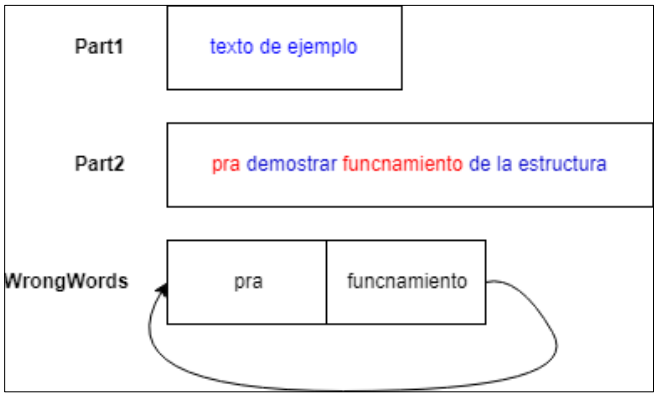

Ilustración 2

Tenemos dos variables de tipo *String* que contienen una parte del texto input cada una:

- Inicialmente, en Part1 estará todo el texto de la entrada hasta llegar a la primera palabra incorrecta.
- Part2 contendrá inicialmente la primera palabra incorrecta y todo el texto que viene después.

WrongWords es un ArrayList que contiene la lista de palabras incorrectas.
Cuando le demos al botón de siguiente, elegiremos de la lista de palabras incorrectas la siguiente palabra incorrecta, y buscaremos la primera instancia de esta en Part2. Cuando la encontremos, el texto que este entre el inicio de la string y esa palabra pasará al buffer Part1, y en Part2 quedará la palabra encontrada con todo el texto que viene después.

De esta manera, si hay una palabra incorrecta que sale 3 veces, sabemos que lo que tenemos que corregir es la palabra que esta al inicio del buffer Part2, que sería como tener un cursor. Para ir hacia atrás, hacemos algo parecido. Buscamos en el buffer Part1 la **última** instancia de la palabra que toque de la lista de palabras incorrectas, y ponemos esa palabra y el texto que venga después dentro del buffer Part2 (delante), y lo quitaremos de Part1. Así que el objetivo es que la palabra seleccionada esté siempre a la cabeza del buffer Part2.

Las búsquedas que hacemos sobre las Strings las hacemos con regex, esto es debido a que no podemos simplemente hacer una búsqueda con el método "indexOf" de la String de Java, ya que, si se busca, por ejemplo, la palabra "es", y en el texto está la palabra "escéptico", puede encontrar ese índice, que no sería correcto. Así que usamos regex para asegurarnos de que sea la palabra entera (ya sea teniendo espacios o signos de puntuación delante o detrás, siendo el inicio de la string, o el final de la string).

C. *Implementacion del Controlador*

Para la implementación se ha creado la clase *Controller* que contiene todos los métodos necesarios para poder comunicar ambas partes del patrón MVC.

En primer lugar, se han definido dos atributos de la clase que contienen las instancias del modelo y la vista, estos se reciben directamente como parámetros a través del constructor. A continuación se ha definido el método *start()* que es al que debe llamar la aplicación principal para comenzar la ejecución de la aplicación. Este método únicamente añade los *listeners* [4] correspondientes al modelo y a la vista, posteriormente hace la vista visible al usuario.

Finalmente hay definidos dos métodos que son los que se han proporcionado como *listeners*. Estos métodos se ejecutarán cuando el modelo o la vista lancen el evento pertinente. A continuación, se explican en mayor detalle:

- *modelPropertyChange():*
  Este método como indica su nombre se ejecuta cuando el modelo cambie alguna de sus propiedades. En esta aplicación únicamente se ha tenido en cuenta la propiedad de *progreso*. Una vez recibido un evento se obtiene el nuevo valor del progreso y se envía a la vista para que actualice la barra de progreso.

- *viewActionPerformed():*
  Este método como indica su nombre se ejecuta cuando la vista lance un evento. En esta aplicación únicamente se ha tenido en cuenta el

[3] Una expresión regular (regex) es una secuencia de caracteres que conforma un patrón de búsqueda.
[4] Un *Event Listener* es un procedimiento o función en un programa que espera a que ocurra un evento. Ejemplos de un evento son el usuario haciendo clic o moviendo el ratón, presionando una tecla en el teclado.

evento que se produce al presionar el botón *"check"* ya que el resto de los botones de la vista se gestionan internamente en esta.

Cuando este método recibe un evento nuevo se obtienen los datos de la vista y se transmiten al modelo para que realice los cálculos oportunos. Para que la vista no se quede congelada durante el proceso, se ha hecho uso del concepto de *Programación Concurrente* que se comenta a continuación.

1) *Programacion Concurrente*

La programación concurrente es una forma de cómputo en la que el trabajo se divide en varios hilos de ejecución distintos que trabajan simultáneamente. Esto suele mejorar el rendimiento de una aplicación al poder realizar cálculos largos en un hilo aparte. Para este proyecto se ha decidido utilizar este concepto sobre el modelo para realizar los cálculos en segundo plano.

Como se ha comentado anteriormente en el método *viewActionPerformed()* se crea un nuevo hilo donde se ejecutaran los cálculos. Para ello se hace uso de la interfaz *Runnable* que nos permite crear un método *run()* que será el que ejecute el nuevo hilo. Dentro de este método se realizan las operaciones correspondientes, se obtienen los datos de la vista, estos después se pasan al modelo para que los compruebe y finalmente se muestran los resultados de nuevo en la vista.

VI. SUPOSICIONES Y POSIBLES MEJORAS

Una de las posibles mejoras mencionadas previamente en el documento es la del tratamiento a la hora de discernir el lenguaje del input introducido por el usuario. Para los textos con los que se está trabajando es viable hacer el tratamiento que se realiza (leer todo el archivo y elegir el diccionario en función de mayor número de coincidencia de palabras), pero si se tuvieran que analizar inputs muy extensos, el tiempo de cálculo no sería aceptable. Por lo que se deberían hacer una serie de pruebas en las que determinar el número mínimo de palabras a analizar del input para determinar en un porcentaje aceptable el idioma.

VII. ESTUDIO DE RENDIMIENTO

Se han realizado una serie de pruebas con textos de distintas longitudes para ver el comportamiento del algoritmo. Los textos introducidos contenían una mezcla de palabras incorrectas y palabras correctas.

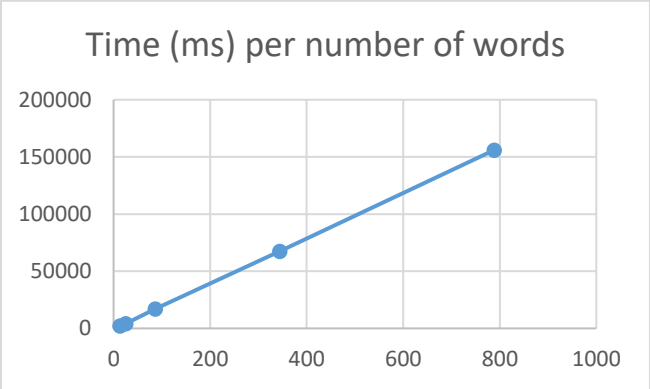

Ilustración 3

Como se puede observar, el gráfico resultante es de tipo lineal, lo que significa que el algoritmo de *Levenshtein* tiene un tiempo que se podría considerar constante para cada palabra, por lo que el coste de una lista de palabras seria lineal.

También se ha mirado el tiempo del algoritmo para textos con muchas palabras incorrectas versus textos correctos. Hemos corregidos dos textos de 30 palabras, uno con todas las palabras correctas, y otro con 5 palabras correctas para detectar el idioma correctamente y 25 palabras incorrectas. Hemos observado que para el texto que contenía solo palabras correctas, el algoritmo ha tardado solo 2 segundos para hacer las comprobaciones, mientras que para el otro ha tardado 11 segundos. Esto probablemente se deba a las operaciones extras que tenemos que hacer si detectamos una palabra incorrecta, como generar la lista de sugerencias, guardar la palabra incorrecta… Y también debido a que el algoritmo de corrección, una vez encuentra una coincidencia, deja de comprobar para más palabras del diccionario, así que se para antes para esa palabra.

VIII. GUÍA DE USUARIO

A continuación, se mostrará un ejemplo de uso del programa. Una vez ejecutado el programa, se observa la siguiente ventana:

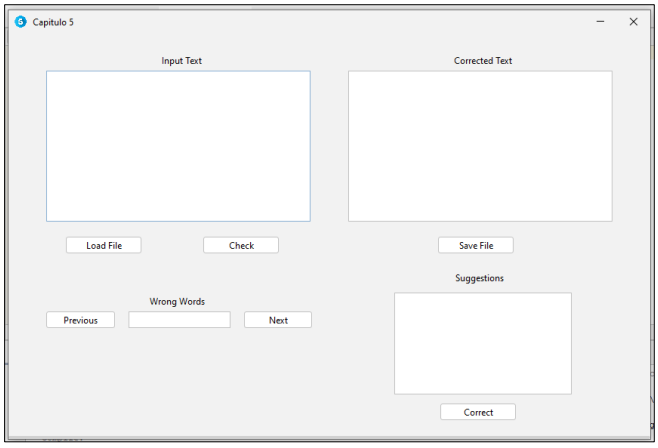

Ilustración 4

Se pueden distinguir las siguientes zonas:

- Zona de input del usuario (recuadro arriba a la izquierda).
- Zona de procesado de texto (recuadro arriba a la derecha).
- Zona de corrección (recuadro abajo a la izquierda).
- Zona de sugerencias (recuadro abajo a la derecha).

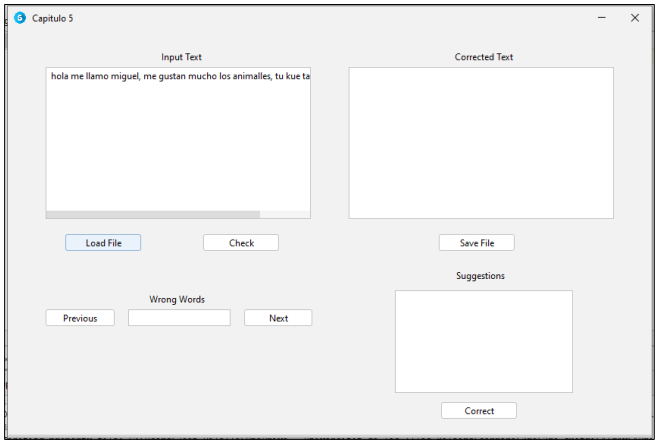

Ilustración 5

El primer paso que deberá seguir el usuario es el de cargar el archivo a analizar. Existen distintas opciones para esto, por una parte, el usuario puede escribir directamente en el recuadro. Por otra parte, tiene la opción de cargar un archivo almacenado en el ordenador, para utilizar esa opción se deberá hacer click en el botón “Load File”.

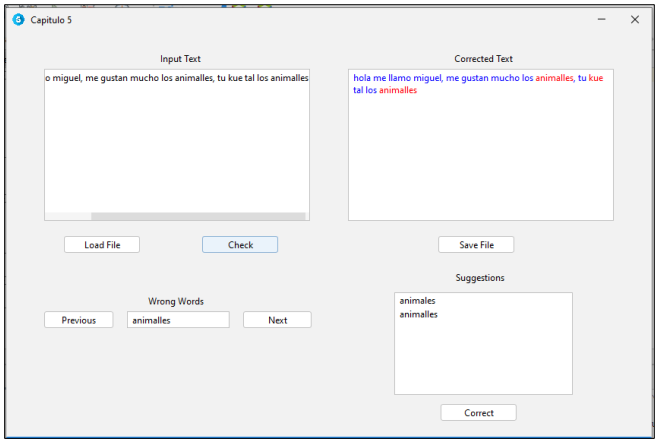

Ilustración 6

Una vez cargado el texto, es hora de analizarlo. Haciendo en click en el botón “Check”. El progreso del análisis quedará graficado con la barra de progreso en la parte inferior. Una vez acabado el análisis, aparecerá en el recuadro superior derecho el texto resultante. Teniendo en cuenta que en azul aparecerán las palabras marcadas como correctas y en rojo las marcadas como incorrectas.

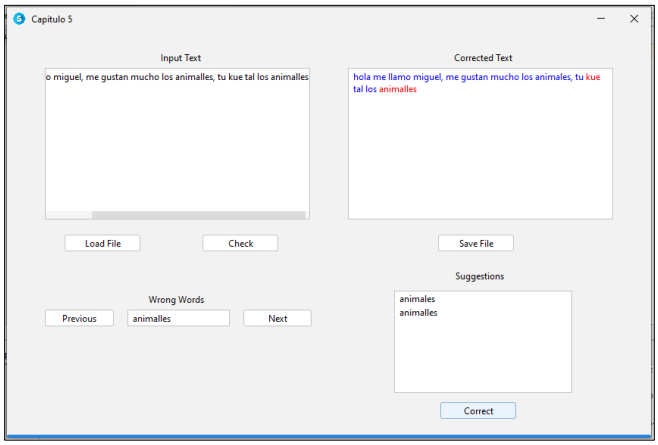

Ilustración 7

Ahora el usuario podrá corregir manualmente cada una de las palabras que el análisis haya considerado incorrectas. En el recuadro de abajo a la izquierda aparecerá la primera palabra detectada y en el recuadro de la derecha estarán todas las sugerencias asociadas a esa palabra. Teniendo en cuenta que aparecerán por orden de distancia (de menor a mayor). El usuario deberá seleccionar una de las palabras de la derecha y hacer click en el botón “Correct”. Con los botones de “Previous” y “Next” se podrá ir desplazando el usuario entre las distintas palabras incorrectas.

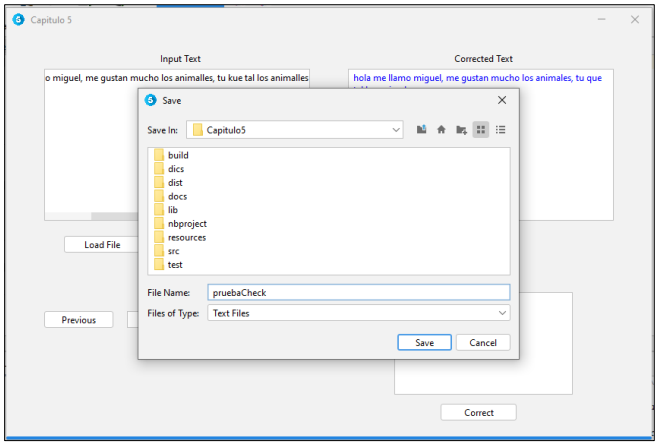

Ilustración 8

Una vez el usuario considere que ha acabado con la corrección del texto, es hora de guardar los cambios. Para ello se hace click sobre el botón “Save File” el cual se sitúa debajo del recuadro superior derecho. Después aparecerá una ventana emergente donde se podrá escribir el nombre del archivo y ubicación de este.

IX. CONCLUSIONES

Esta práctica nos ha servido para terminar de asentar los conocimientos sobre la programación dinámica, así como ver un buen método de cómo se puede hacer corrección léxica de textos. Algoritmos como el de Levenshtein siempre son bienvenidos debido a su bajo coste.

La programación dinámica es una muy buena técnica si se puede aplicar al problema, ya que permite reducir considerablemente el tiempo de ejecución. Pero hay que recalcar que la memoria sigue siendo un recurso limitado, y hay que usarla de manera responsable.

X. BIBLIOGRAFÍA

The Levenshtein Algorithm - Cuelogic Technologies Pvt. Ltd.

What is Dynamic Programming? - Grokking Dynamic Programming Patterns for Coding Interviews (educative.io)

Understanding the Levenshtein Distance Equation for Beginners | by Ethan Nam | Medium