Comparación de Idiomas Mediante MVC

Alejandro Rodríguez, Rubén Palmer y Sergi Mayol

Resumen— Esta aplicación presenta una interfaz gráfica de usuario que le permite interactivamente calcular la distancia entre un conjunto de diomas basado en el algoritmo de levenshtein dinámico usando el patrón de diseño MVC.

Vídeo - ver vídeo

1. Introducción

En este artículo explicaremos el funcionamiento y la implementación de nuestra práctica estructurando y dividiendo sus partes en los siguientes apartados:

Inicialmente, se citarán las librerías usadas en este proyecto. Para aquellas que se hayan desarrollado específicamente, se expondrá una breve guía de su funcionamiento y uso.

Seguidamente, se explicará cuál es la arquitectura, interfaces, funcionamiento e implementación de nuestro **M**odelo-**V**ista-**C**ontrolador (MVC) así como los métodos de mayor importancia. Para facilitar la legibilidad, se separará en cada uno de los grandes bloques de la estructura; en este caso Modelo, Vista, Controlador y Hub

Finalmente, se mostrará una breve guía de cómo usar nuestra aplicación y un ejemplo de esta.

2. Librerías

En esta sección se explicarán las librerías implementadas y usadas para llevar a cabo el desarrollo de las prácticas y permitir la reutilización de las mismas durante el transcurso de la asignatura.

El principal motivo que nos ha impulsado a implementar dichas librerías es la facilidad que proporcionan para centrarse únicamente en el desarrollo de la práctica en sí; Gracias a que durante el desarrollo de estas librerías se ha priorizado la facilidad de manejo, uso genérico y optimización para añadir el mínimo "overhead" al rendimiento del programa.

2.1. Better swing

Better swing [1] es una librería derivada de Java Swing que permite un desarrollo más amigable y sencillo de interfaces gráficas, al estar inspirado en el desarrollo web (HTML y CSS), concretamente, en el framework de "Bootstrap". Se basa en "JfreeChart" para pintar las gráficas de líneas creando "wrappers" para facilitar su uso y manejabilidad.

2.1.1. Funcionamiento

El funcionamiento del paquete es muy sencillo, básicamente, la idea es crear una o varias ventanas con una configuración deseada y acto seguido ir creando y añadiendo secciones (componentes), donde posteriormente se pueden ir borrando y actualizando los componentes individual o grupalmente. Por ello, este paquete proporciona una serie de métodos para la ventana y las secciones.

¿Cómo funciona la ventana?

Para hacer que la ventana funcione es necesario crear una instancia del objeto `Window`, inicializar la configuración de la misma empleando el método `initConfig` recibiendo por argumento la ruta donde se encuentra el

archivo. En caso contrario se empleará la predeterminada. Finalmente, para que se visualice la ventana se realizará con el método `start` que visualizará una ventana con la configuración indicada anteriormente.

En el caso de querer añadir componentes como una barra de progreso, un botón o derivados, es tan sencillo como crear una sección y emplear el método `addSection`, el cual recibe por parámetro la sección a añadir, el nombre de la sección y la posición de este en la ventana.

¿Cómo funciona la sección?

Las secciones son instancias de la clase `Section` que permiten formar los diferentes componentes de la ventana.

El funcionamiento de una sección es muy sencilla, se trata de instanciar un objeto de la clase `Section` y llamar a algún método de esta clase, pasando los parámetros adecuados, y finalmente añadir la sección a la ventana.

Otros

Adicionalmente, se dispone de una clase llamada `DirectionAndPosition`, que constituye las posibles orientaciones y direcciones que una sección puede tener.

2.1.2. Implementación

Para el desarrollo de esta librería se ha centrado en la simplicidad hacia el usuario final y la eficiencia de código mediante funciones sencillas de emplear y optimizadas para asegurar un mayor rendimiento de la interfaz de usuario. Este paquete se divide en dos principales partes:

Window: Consiste en un conjunto de funciones para la creación y configuración de la ventana.

Section: Consiste en un conjunto de funciones base para la creación de secciones en la ventana.

La implementación de la `Window` consiste en diversas partes: gestión de teclas, configuración, creación y actualización de la ventana y creación de las secciones.

La gestión de las teclas se realiza mediante una clase llamada `KeyActionManager`, que implementa la interfaz `KeyListener`. Esta clase permite gestionar los eventos de teclado de la ventana y se utiliza para la depuración y el desarrollo del programa.

En cuanto, la configuración, creación y actualización de la ventana y la creación de las secciones, se han desarrollado una serie de métodos que envuelven a un conjunto de funciones propias de java swing. Por tanto, con este conjunto de métodos y funciones se obtiene la clase `Window`.

A continuación se explicarán los principales métodos de `Better swing`:

`initConfig` es un método que permite cargar la configuración de la vista y crea el marco de la vista, incluyendo apariencia, posición, tamaño, icono, color de fondo y manejo de eventos de teclado.

`start` permite la visualización de la ventana, haciendo que la ventana sea visible para el usuario.

`stop` actua como wrapper de `JFrame::dispose`.

`addSection` permite añadir una sección a un objeto `Window` indicándole la posición y dirección del panel mediante el conjunto de variables definidas en `DirectionAndPosition`.

`updateSection` permite actualizar una sección indicándole la posición y dirección del panel mediante el conjunto de variables definidas en `DirectionAndPosition`.

`deleteComponent` permite borrar un componente específico de la ventana que se le haya pasado por parámetro al método.

`repaintComponent` permite repintar un componente específico de la ventana que se le haya pasado por parámetro al método.

`repaintAllComponents` repinta todos los componentes de la ventana.

La implementación de la `Section` consiste en diversas partes, las cuales son las siguientes: Los métodos que permiten crear las secciones ya configuradas y las clases en las que se basan los métodos anteriores.

La propia librería contiene muchas más clases y métodos que si el lector desea explorar puede acceder a su documentación a través del código fuente proporcionado en la entrega de la práctica.

2.1.3. Manual de uso

En este apartado se describe como emplear la librería para su correcto funcionamiento.

Para emplear la librería es tan sencillo como crear una instancia de la clase `Window`, llamar al método `initConfig` indicando el fichero de configuración de la ventana, si se desea, en caso contrario se deberá pasar un `null` y se emplearán la configuración por defecto, y finalmente llamar a la función `start` cuando se desee inicializar la ventana.

Es importante inicializar la configuración antes de ejecutar la función `start`, ya que en caso contrario se producirá una excepción. A continuación se muestra un sencillo ejemplo:

```
Window view = new Window();
view.initConfig("config.json");
view.start();
```

Como se observa, la librería permite cargar la configuración e iniciar la ventana independientemente, permitiendo una mayor flexibilidad.

Además, en el caso de querer reiniciar la configuración, cambiar la visibilidad de la ventana o guardar el contenido de esta sin tener que volver a compilar el código, se puede realizar a través de unos atajos de teclado, que son los siguientes:

Q: Cerrar programa.
R: Reiniciar configuración.
V: Cambiar visibilidad.
G: Guardar contenido.

Por ejemplo, al cambiar la configuración y reiniciar la ventana se aplicarán los cambios automáticamente sin compilar de nuevo el código, es decir, se permite el conocido “Hot Reloading”.

En el caso de querer crear una sección y añadirla, se realizaría de la siguiente forma:

```
Window view = new Window();
Section section = new Section();
// Los datos pueden ser de longitudes irregulares
long[][] data = { ... };
Color chartColors[] = { Color.RED, Color.BLACK };
String chartColumnLabels[] = { "Linea 1",
                               "Linea 2" };
section.createLineChart(labels,
                        data,
                        chartColors,
                        chartColumnLabels,
                        "Ejemplo Lineas");
view.addSection(section,
                DirectionAndPosition.POSITION_TOP,
                "Chart");
```

En el ejemplo anterior se crearía una gráfica de líneas con dos líneas, una de color rojo y otra negra, con los nombres “Linea 1” y “Linea 2”, con el título “Ejemplos Lineas” y con los puntos del array data.

Finalmente, comentar que hay ilimitadas posibilidades de creación y personalización de componentes con los ya configurados y las posibilidades de crear libremente los propios componentes.

3. Conceptos MVC

El Modelo Vista Controlador (MVC) es un patrón arquitectónico de software que divide una aplicación en tres elementos interconectados, separando la representación interna de la información, como se muestra al usuario y donde se hacen cómputos con esa información respectivamente.

Para esta práctica, se ha decidido modificar parcialmente este patrón al implementar la comunicación entre los módulos mediante un sistema de peticiones. Por ende, encontramos 4 módulos en nuestro MVC: Modelo, Vista, Controlador y Hub. Este último se encargará de gestionar toda la comunicación.

Esta decisión de modificación del MVC ha sido respaldada por la facilidad que obtenemos de escalar el patrón, permitiendo crear tantos módulos como queramos. Esto permitiría que, si en un futuro se quisiera añadir otro controlador en otro lenguaje de programación o encapsular controladores por funcionalidad, el único cambio a efectuar ocurriría en el hub.

A continuación se explicarán cada uno de los módulos y como han sido adaptados y empleados en la práctica.

3.1. Implementación

El Modelo vista controlador (MVC), en esta práctica, ha sido implementado como si fueran aplicaciones diferentes, es decir, cada parte del MVC es independiente de otro elemento de este ya que todos las partes se comunican a trvés de un hub. Esta implementación, trata de imitar una serie de microservicios que se comumican entre ellos a través de una API (application programming interface).

En cuanto a la implementación interna, se trata de una serie de servicios (modelo, vista y controlador) que se conectan a un servidor (hub), donde, entre ellos, se comunican a través de la red local del dispositivo.

¿Cómo hemos conseguido implementar esta idea en Java?

Para implementar el concepto de servicios y servidor se ha realizado a través de los sockets de Java, el problema, de únicamente emplear los sockets, es que los procesos se bloquean cuando envían una petición al servidor hasta que reciben una respuesta, por ello, hemos decidido crear una comunicación asíncrona no bloqueante, es decir, los servicios y el servidor no se pueden bloquear. El concepto empleado es similar al que usa el entorno en tiempo de ejecución multiplataforma NodeJs. Básicamente, un servicio realizará una petición y la respuesta que obtendrá es una promesa diciendo que le llegará una respuesta con los datos que desea, en el caso de que desee obtener información de vuelta, donde le informará de que en algún momento será notificado con los datos solicitados en la petición realizada. Con este planteamiento, se consigue que los procesos, como se ha mencionado anteriormente, no se bloquen y puedan seguir con su ejecución. En el caso, de que se desee esperar a la información, se puede realizar uso de una función similar a un “await”, en la cual, se encargará de bloquear el proceso hasta que la petición tenga los datos esperados.

La siguiente imagen muestra un diagrama de cómo está implemtentado el sistema de comunicación entre los elementos:

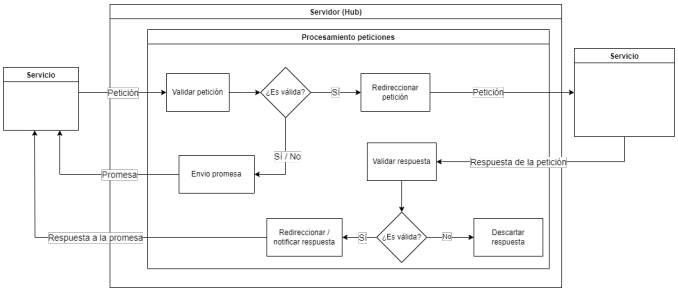

Figura 1. Comunicación entre elementos MVC y hub.

Además, de la implementación comentada anteriormente, los “endpoints”/rutas del servidor se definen a través de un archivo de configuración JSON, el cual, mapea cada petición con los servicios, al igual que sus respuestas pertinentes. Esto se puede ver en el siguiente ejemplo:

```
{
  "requestMap": [
    {
      "code": "SEND_GEOPOINTS",
      "services": ["MODEL", "CONTROLLER"]
    }
  ],
  "responseMap": [
    {
      "code": "LOAD_MAP",
      "services": ["VIEW"]
    },
  ],
}
```

3.2. Hub

El flujo de datos empleado en este patrón de diseño ha sido inspirado en cómo se comunican

los diferentes elementos en un servidor. Es decir, cualquier módulo del MVC deberá realizar una petición al servidor (Hub) y este se encargará de solventar y dar servicio a dicha petición. Se puede ver con mayor claridad cómo funciona y su implementación en la siguiente figura:

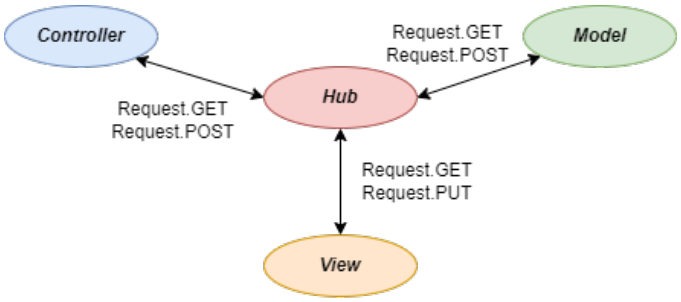

Figura 2. Patrón MVC usado en la práctica.

El Servidor/Hub se encarga de administrar los diferentes tipos de peticiones que llegan de los diferentes componentes. En la figura 2 se observan los diferentes tipos de peticiones posibles que se pueden realizar, en concreto: GET, POST y PUT. En un entorno web, GET sirve para obtener datos, POST para enviar y PUT para actualizar. En el programa los diferentes tipos de peticiones se han traducido en:

GET Realiza un retorno del dato solicitado.
POST Añade un el dato enviado.
PUT Realiza un set del dato enviado.

Para permitir esta implementación, se ha creado la interfaz `Notify` con el método `notifyRequest` el cual deberá estar implementado en todos los módulos del MVC, ya que será el responsable de gestionar las peticiones y asegurar que cada uno de ellos puede realizar sus tareas designadas de manera eficiente y precisa.

3.3. Modelo

El modelo es la representación de los datos que maneja el software. Contiene los mecanismos y la lógica necesaria para acceder a la información y para actualizar el estado del modelo.

Para este proyecto, y con el objetivo de ser lo más fiel a un caso realista aplicable a una aplicación real, se ha decidido crear una `sqlite` donde se guardarán todos los datos. Así pues, nuestro modelo presenta dos elementos:

3.3.1. *Base de datos*

La base de datos consta de una tabla por idioma y una tabla para guardar el histórico de tiempos de ejecución; Técnicamente:

- **LANGUAGE**, donde el nombre de la tabla corresponde al nombre corto de un lenguaje. Con los atributos:
  - `word`: Representa una palabra del lenguaje.
- **Timed_Execution**, con los atributos:
  - `id`: Permite identificar cuál ha sido la última ejecución.
  - `milis`: La cantidad de milisegundos usados durante la última ejecución.

3.3.2. *Interfaz*

Ya que el modelo es realmente la base de datos, es necesaria una interfaz que nos permita leer, modificar o escribir dentro de la base de datos. Para ello, la propia clase `Model.java` nos permite ejecutar un conjunto de funciones que aplican queries a la base de datos. La mayoría de estos métodos siguen el siguiente patrón:

- Conectar a la base de datos
- Ejecutar el query
- Transformar el resultado a un objeto de java (como puede ser un array)
- Devolver el resultado si no hay errores

Aunque el diseño se acerca a la posible implementación en una aplicación real, la implicación de usar queries a una base de datos nos proporciona tanto beneficios como perjuicios.

Beneficios

- Se delega la búsqueda y filtrado de datos a un lenguaje especialmente diseñado para ello.

- Los datos más usados se guardan en la caché, mientras que los demás se pueden guardar en disco.
- Los datos se mantienen entre ejecuciones, por lo que se puede aplicar un posterior estudio de los resultados en otros entornos y lenguajes.

Perjuicios

- Se genera un overhead general en la aplicación proporcional a cómo esté programada la librería usada, ya que la conexión se debe abrir y cerrar en cada query que se ejecuta, además de la necesidad de transformar los resultados a un objeto del lenguaje usado.

Aun así, tanto por interés académico como para crear ejemplos lo más cercanos a una aplicación real, se ha decidido seguir este acercamiento ante el problema de la gestión de un modelo en una aplicación.

Finalmente, al ser un módulo de nuestro MVC, implementa la interfaz `Notify` y su método `notifyRequest` que le permite recibir notificaciones de los otros módulos del MVC.

3.4. Vista

La vista contiene los componentes para representar la interfaz del usuario (IU) del programa y las herramientas con las cuales el usuario puede interactuar con los datos de la aplicación. Adicionalmente, la vista se encarga de recibir e interpretar adecuadamente los datos obtenidos del modelo. Cabe mencionar, que al igual que el resto de componentes del MVC, la clase View implementa la interfaz "Notify", la cual permite la comunicación con el resto de elementos.

`loadContent` es el encargado de cargar el contenido inicial en la ventana. Para esta práctica, carga los siguientes elementos semánticos:

`menu`, función que crea y configura el menú de utilidades de la ventana principal. En esta, se incluyen las siguientes opciones: Abrir ventanas de estadísticas, abrir "Word Guesser" y salir del programa.

`body`, función que crea, configura y actualiza la selección de diccionarios. Se trata de una serie de botones que permiten al usuario seleccionar los diccionarios a comparar entre ellos. Además de, una vez ejecutado el algoritmo se permitan visualizar los resultados de forma gráfica.

`sidebar`, función que crea y configura la barra de opciones de la derecha de la interfaz principal. Esta permite al usuario interactuar y configurar el entorno de ejecución de los algoritmos y seleccionar el número de palabras a seleccionar aleatoriamente de cada diccionario. Además, se ha añadido una pequeña ventana de logs para saber que está ocurriendo en todo momento.

`footer`, función que crea y configura los botones para iniciar el algoritmo, además de unos botones para cambiar el "Body" a los resultados de la ejecución del algoritmo.

Adicionalmente, a la vista principal, esta permite desplegar una serie de ventanas extra. Una ventana muestra, a tiempo real, el uso y consumo de la memoria de la Java virtual machine (JVM), la otra ventana muestra las estadísticas de la ejecución de los algoritmos además de su comparación, una para abrir el Adivinador de palabras (3.4.1) y un manual de usuario (4.1).

Finalmente, al ser un módulo de nuestro MVC, implementa la interfaz `Notify` y su método `notifyRequest` que le permite recibir notificaciones de los otros módulos del MVC.

3.4.1. Adivinador de palabras

Este apartado de la vista principal, es el encargado de dada una frase, intentar decir al usuario en que idioma se ha escrito la sentencia.

Esta ventana está compuesta de 2 principales partes, la parte central de la ventana, y el "footer". En la parte central, se encuentran los resultados, de la entrada del usuario, en

un gráfico de barras, donde, el lenguaje con más similitud de la entrada es resaltado en rojo, mientras que, el resto en azul. En la parte inferior, se encuentra una entrada de texto para que el usuario pueda introducir la oración, y un botón de detectar lenguaje.

Cabe mencionar, que aparte de poder reconocer una oración, también es capaz de reconocer una única palabra.

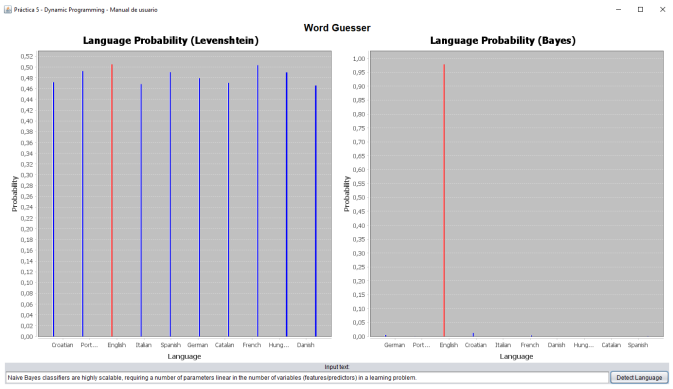

Figura 3. Interfaz Word Guesser

En el apartado 3.5.4 se explica con detalle la implementación interna del algoritmo para averiguar de qué lenguaje se trata la entrada del usuario.

3.4.2. *Estadísticas JVM*

Este apartado de la vista principal, es el encargado de enseñar a tiempo real las estadísticas de la máquina virtual de java. Concretamente, se actualiza cada 0.5 s a partir de los datos obtenidos de la clase de java "Runtime" y muestra la memoria libre, la memoria total y el uso de esta en una gráfica de líneas, donde el eje x es instante en el tiempo que se han obtenido los datos y el eje y su valor. Todos los datos de la memoria obtenidos están en MB.

3.4.3. *Estadísticas de los Algoritmos*

Este apartado de la vista principal, es el encargado de enseñar las estadísticas de la ejecución de los algoritmos, Levenhstein paralelizado y Levenhstein secuencial. Estas estadísticas incluyen una gráfica con el tiempo de ejecución de cada ejecución del algoritmo. Los resultados (tiempos de ejecución) son representados en un

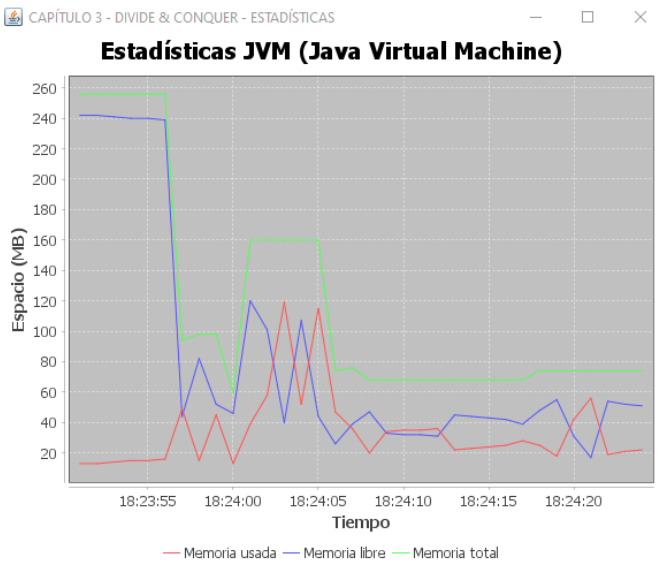

Figura 4. Interfaz estadística JVM

gráfico de barras, donde el eje x representa el número de ejecución del algoritmo y el eje y el tiempo en milisegundos que ha tardado.

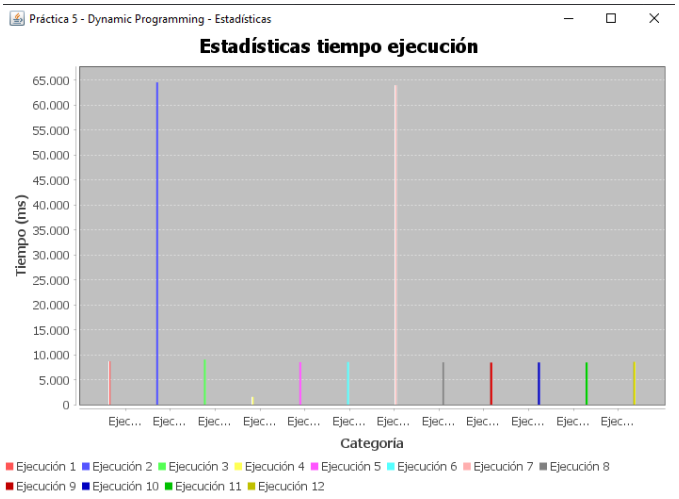

Figura 5. Interfaz estadísticas algortimos

Como se ha podido ver anteriormente en la imagen (5), los datos de las estadísticas están representados con un diagrama de barras, donde el eje x es la categoría a la que pertenecen y el eje y el valor correspondiente.

3.5. Controlador

El controlador en un MVC es el responsable de recibir y procesar la entrada del usuario y actualizar el modelo. En esta práctica, dicha

responsabilidad es la de calcular la distancia de Levenshtein y transformar los resultados de dicho algoritmo a diferentes estructuras de datos.

3.5.1. Diseño

El controlador se separa en dos grandes secciones como ya se ha mencionado previamente.

Por una parte, calcular la distancia de Levenshtein entre dos conjuntos de idiomas. Usando "method overloading" [10], podemos abstraer Levenshtein para que acepte dos palabras, dos idiomas o que gestione la comunicación y preparación de los datos para lanzar una de sus posibles variedades previamente estipuladas.

Por otra parte, y debido a la necesidad de representar los resultados de diferentes maneras en la vista, este se encarga de, una vez obtenida una solución de ejecutar Levenshtein, transformar dicha solución a las diferentes posibles estructuras necesarias. Específicamente, se encarga de convertir dicha solución en un grafo de distancias y el árbol filogenético [11]

3.5.2. Levensthein

La distancia de Levenshtein permite identificar la "distancia" entre dos cadenas de valores; véase, cuantas modificaciones tendríamos que hacer en uno para que sea el segundo. Estas modificaciones son, para una posición *i*:

- Cambiar su valor
- Eliminarlo
- Añadir un valor

Es extremadamente popular para verificar la ortografía de un texto, analizar secuencias de ADN, "data mining" y procesamiento del lenguaje natural; entre otros. Con respecto a su implementación, y aunque existen otros acercamientos, se ha decidido aplicar el concepto de programación dinámica para "cachear" valores previamente ya calculados. De esta manera, evitamos repetir cálculos y aceleramos su proceso pagando un coste adicional en memoria. Una posible implementación de la distancia de Levenshtein con programación dinámica podría ser el siguiente:

```
function LevenshteinDistance(char s[1..m], char
↪  t[1..n]):
  // for all i and j, d[i,j] will hold the
  ↪  Levenshtein distance between
  // the first i characters of s and the first j
  ↪  characters of t
  declare int d[0..m, 0..n]

  set each element in d to zero

  // source prefixes can be transformed into empty
  ↪  string by
  // dropping all characters
  for i from 1 to m:
    d[i, 0] := i

  // target prefixes can be reached from empty
  ↪  source prefix
  // by inserting every character
  for j from 1 to n:
    d[0, j] := j

  for j from 1 to n:
    for i from 1 to m:
      if s[i] = t[j]:
        substitutionCost := 0
      else:
        substitutionCost := 1

      d[i, j] := minimum(d[i-1, j] + 1,
                         d[i, j-1] + 1,
                         d[i-1, j-1] +
                         ↪  substitutionCost)

  return d[m, n]
```

Con respecto a la complejidad temporal, podemos asegurar que, para una palabra con otra, es $O(src * dst)$ donde *src* y *dst* son las longitudes de las cadenas de entrada; ya que creamos dos bucles anidados cuyo límite es *src* y *dst*. El algoritmo, como ya se ha citado previamente, se ha implementado aplicando programación dinámica, por lo que crea una matriz de tamaño $(src + 1) \times (dst + 1)$ y rellenándola iterativamente. Dado que el cálculo de una celda es tiempo constante, la complejidad temporal general del algoritmo dependerá del tamaño de la matriz y como esta tiene como parámetros *src* y *dst*, reafirmamos que la complejidad temporal es de $O(src * dst)$

Con respecto al coste espacial, debido a que tenemos que generar una matriz de datos proporcional al tamaño de las cadenas de entrada, obtenemos $O(src * dst)$.

3.5.2.1. Estudios: Debido a la gran cantidad de datos usados para este proyecto, la aplicación ofrece una opción para ejecutar los cálculos paralelamente, al no depender los resultados de los previos. Sin embargo, debido al sistema de *Notify Request*, la aplicación es nativamente paralela y usará tantos procesos como la CPU del hardware pueda lanzar. Debido al interés académico, se implementó una estructura que permite lanzar el algoritmo de manera secuencial:

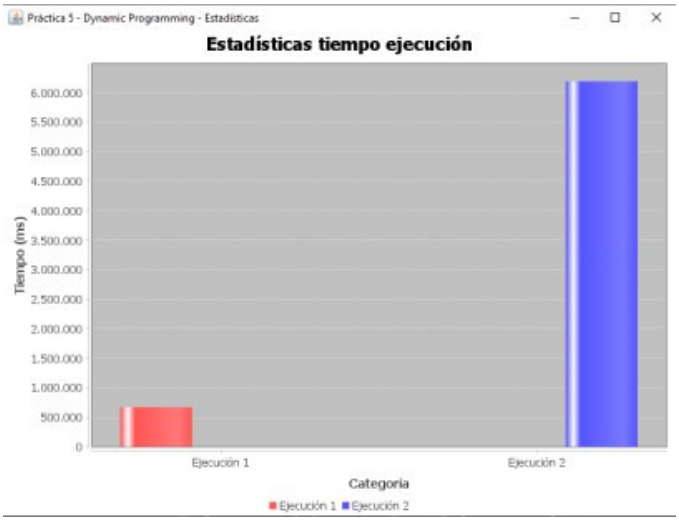

Figura 6. Estadísticas temporales en milisegundos

Como se puede apreciar, a la izquierda se postra la ejecución paralela y a la derecha la secuencial. Ambas compararon todos los diccionarios entre ellos con diez mil palabras de resolución. Esto es, cada vez que se aplicó Levenshtein se cogieron diez mil palabras aleatorias del lenguaje. Bajo este benchmark sacamos los siguientes datos:

Paralela
- **CPU**: 100 %
- **Tiempo**: 10 min

Secuencial
- **CPU**: 12 %
- **Tiempo**: 103 min

Podemos observar que, aunque la ejecución paralela tarda aproximadamente diez veces más, también consume diez veces más los recursos del "hardware". Esto nos indica que, al menos para la máquina en la que se ha ejecutado el benchmark, podemos usar ambas ejecuciones según nuestras prioridades.

3.5.3. *Transformación de estructuras de datos*

Ya que nuestra aplicación permite la visualización de los pesos de Levenshtein tanto en grafo como en un árbol filogenético [11], es necesario transformar los datos devueltos por este a dichas estructuras. Técnicamente, el algoritmo de Levenshtein dados un conjunto de lenguajes devuelve un "hashmap" cuyos claves son las parejas de lenguajes y los valores son los pesos. Para ello se han creado dos estructuras de datos que permiten representar dichas estructuras.

Para el grafo, creamos un conjunto cuyos valores son los diferentes idiomas en los que se ha ejecutado la distancia de Levenshtein. Iteramos bajo él y creamos todas las posibles conexiones con los demás lenguajes y obtenemos el peso accediendo al "hashmap". Finalmente, devolvemos un array con todos los lenguajes, sus conexiones y sus pesos.

Para el árbol podemos usar el grafo previamente mencionado, ya que podemos saber para un lenguaje cuáles son los pesos y por ende, podemos ordenarlos de menor a mayor. Usamos el algoritmo de Kruskal [12] para obtener el árbol mínimo dado un grafo no dirigido ponderado por aristas cuyo pseudocódigo podría ser tal que:

```
algorithm Kruskal(G) is
    F:= Ø
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v),
    ↪  increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F:= F ∪ {(u, v)} ∪ {(v, u)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

3.5.4. *Adivinador de palabras*

Para el adivinador de palabras, se emplean dos estrategias:

La primera es una versión modificada de nuestra estructura de Levenshtein. Creamos un

diccionario "custom" cuyas palabras son las escritas por el usuario por input, y ejecutamos Levenshtein con todos los lenguajes con una resolución de dos mil palabras. El resultado, como ya explicado previamente, es un "hash-map" cuyos valores representan las distancias. Usando una simple fórmula para mapear todas las distancias a una probabilidad:

$$probabilidad = \frac{1}{distancia + 1}$$

Podemos mostrar todas las probabilidades y escoger aquella que sea mayor.

La segunda, es un modelo generativo conocido como clasificador de Naive Bayes, este se basa en que las características de las palabras son independientes, es decir, supone que las probabilidades de característica $p(x_j|y)$ son independientes $p(x|y = c) = \prod_{j=1}^{n} p(x_j|y = c)$ dada la clase c. Este, se basa en la función de regresión lineal

$$\theta_{j|y=c} = p(x_j|y = c) = \frac{count(x_i, y = c) + 1}{\sum_{i=1}^{m} count(x_i, y = c) + |V|}$$

donde, finalmente se le aplicará una función de softmax para obtener las probabilidades normalizadas, por lo que, quedaría de la siguiente forma:

$$\text{softmax}(x_i) = \log\left(\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}\right)$$

4. MANUAL DE USUARIO

Para el correcto uso de la aplicación, el conjunto de acciones que se pueden realizar en dicho programa serán definidas a continuación. También, la distribución de las secciones de la interfaz junto a su explicación y funcionalidad.

Cuando se ejecute el programa por primera vez en la pantalla se debe mostrar la siguiente interfaz de usuario:

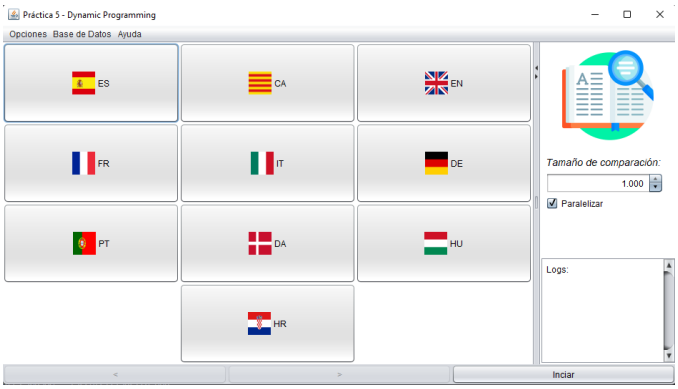

Figura 7. Interfaz de usuario.

4.1. Menu

El menú de la aplicación consiste en la barra que se sitúa debajo del marco superior de la ventana. En esta se puede encontrar un conjunto de opciones para que el usuario pueda interactuar con la aplicación, modificar y analizar el comportamiento de esta. En concreto, se encuentran las opciones de "Opciones" "Base de Datos" y "Ayuda".

En la primera se sitúan las acciones para salir del programa, resetear datos, una funcionalidad que hemos añadido llamada "Adivinador de palabras" e iniciar las ventanas de estadísticas explicadas anteriormente en los apartados 3.4.2 y 3.4.3.

En la segunda opción llamada "Base de Datos" encontramos dos acciones que el usuario puede realizar, la primera, cargar una base de datos en caso de que quiera cambiar los idiomas, la segunda opción devuelve los idiomas cargados en la base de datos.

En la tercera opción se encuentra un menú desplegable con un manual de usuario con la explicación del funcionamiento de la aplicación.

4.2. Main

El "Main" es el bloque principal de la vista, donde se representará el mapa con las poblaciones, aquí el usuario podrá seleccionar los puntos/poblaciones para encontrar el camino mínimo. El usuario deberá clicar de manera

precisa en el punto, debido a que éste tiene un radio de detección del ratón determinado. De esta manera, al clicar en el mapa, no se seleccionará ningún punto de manera errónea. No podrá clicar dos veces sobre el mismo punto.

4.3. Sidebar

El "Sidebar" contiene un conjunto de opciones para modificar los datos y el modo de ejecución de la aplicación. A continuación, se explicará cada una de estas opciones y su función.

En primer lugar, se encuentra la opción para seleccionar el "Tamaño de comparación". En esta, se puede escoger el número de palabras a comparar de cada diccionario para ejecutar el algoritmo.

Debajo de este, se encuentra un checkbox para marcar la opción de paralelizar el algoritmo o hacerlo secuencial.

En tercer lugar, se encuentra una ventana de "logs". En esta ventana el usuario tendrá una noción de lo que está ocurriendo en el programa, indicando los diccionarios seleccionados.

4.4. Footer

En la sección "Footer", se hallan tres botones para interactuar con la interfaz para poder desplazarnos entre las diferentes pantallas. Además, hay un botón "Iniciar" para ejecutar el algoritmo.

En la izquierda, hay unos botones que permitirán al usuario poder moverse por las pantallas una vez se haya ejecutado el algoritmo, en caso contrario no se podrán utilizar.

En el lado derecho, tenemos el botón "Iniciar" el cual ejecutará el algoritmo después de haber seleccionado como mínimo dos idiomas.

4.5. Ejemplo ejecución

Al iniciar la aplicación se mostrará una interfaz como la expuesta en la imagen 7. Para poder iniciar la ejecución del algoritmo, es necesario seleccionar dos o más diccionarios, en el caso de equivocarse de idioma y querer seleccionar otro, bastará con volver a clicar sobre el diccionario. Además, tiene la opción de reiniciar todo en el menú de arriba. Una vez seleccionados los idiomas, al clicar en "Iniciar", se ejecutará el algoritmo y a continuación se mostrarán una serie de pantallas por las que el usuario podrá navegar con los botones inferiores. A continuación, la siguiente imagen, muestra un ejemplo de la interfaz tras la ejecución:

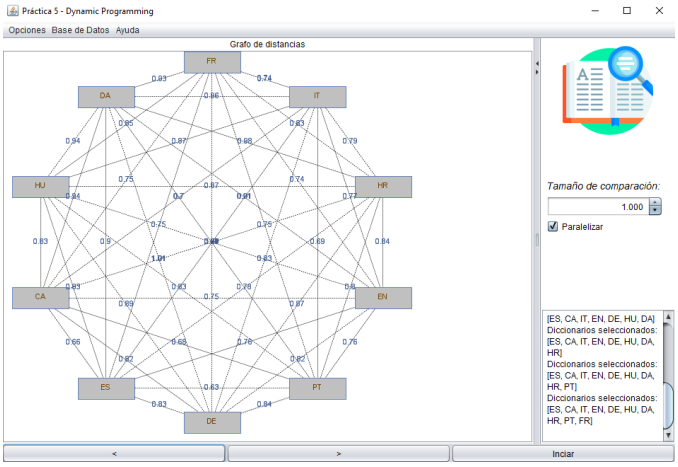

Figura 8. Interfaz de usuario en ejecución.

Tras ejecutar el algoritmo, se muestra un grafo de distancias. Cada uno de los idiomas tiene asociado un valor "distancia" para los otros idiomas seleccionados.

En la siguiente pantalla, se proporciona el mismo resultado representado en una gráfica.

Por último, se muestra el árbol léxico de los diccionarios.

En el menú superior, en el apartado "Opciones", se encuentra una funcionalidad extra, el "Adivinador de palabras". El usuario al seleccionar esta opción se le abrirá una segunda pantalla en la que podrá introducir cualquier fragmento de texto y se le dará una aproximidad de a que idioma pertenece. A partir de aquí, el usuario puede reiniciar el programa con el botón del menú para ejecutar nuevamente el algoritmo, cambiar la base de datos, cambiar el número de palabras a comprar, seleccionar diccionarios y ver estadísticas de la ejecución, ver apartados 3.4.2 y 3.4.3.

5. Conclusión

Teniendo en cuenta el desarrollo expuesto, la aplicación permite generar un punto de referencia de un conjunto de algoritmos de diferente complejidad asintótica mediante una representación gráfica e interactiva de la media de los tiempos de ejecución a través de unos parámetros de entrada definidos por el usuario. Adicionalmente, se ha implementado mediante una versión modificada del patrón de diseño de software **M**odelo **V**ista **C**ontrolador (MVC) añadiendo un cuatro módulos que permite la comunicación entre los diferentes elementos de este mediante peticiones. Esta modificación ha sido fuertemente inspirada en el diseño de un servidor, lo que permite gran flexibilidad y escalabilidad de desarrollo, al poder interceptar y gestionar las peticiones a gusto del desarrollador.

Además, se ha hecho énfasis en el uso de las librerías creadas por los miembros del grupo para facilitar el trabajo y la reutilización de código en futuras prácticas. Reiterar que el principal motivo que nos ha impulsado a implementar dichas librerías es la facilidad que proporcionan para centrarse únicamente en el desarrollo de la práctica en sí; Gracias a que durante el desarrollo de estas librerías se ha priorizado la facilidad de manejo, uso genérico y optimización para añadir el mínimo "overhead" al rendimiento del programa.

Concluir que, con respecto al ámbito académico, este proyecto nos ha permitido consolidar el concepto de patrón de diseño MVC gracias a un primer proceso de investigación y discusión del diseño a implementar entre los integrantes del grupo.

6. Distribución del trabajo realizado

El trabajo realizado por cada miembro de la práctica ha sido el siguiente:

- Alejandro Rodríguez:
  - Documentación
  - Diseño del proyecto
  - Desarrollador de la base del "Language guesser"
  - Desarrollador de la plantilla base del proyecto
- Rubén Palmer:
  - Documentación
  - Diseño del proyecto
  - Principal desarrollador del "backend" de la aplicación
  - Desarrollador general de la aplicación
  - Desarrollador del "frontend"
  - Diseñador de la implementación del patrón MVC de la aplicación
  - Desarrollador de la plantilla base del proyecto
  - Desarrollador de la plantilla de la documentación
- Sergi Mayol:
  - Documentación
  - Diseño del proyecto
  - Desarrollador de la librería `Better Swing`
  - Principal desarrollador del "frontend" de la aplicación
  - Desarrollador general de la aplicación
  - Desarrollador del "backend" de la aplicación
  - Diseñador de la implementación del patrón MVC de la aplicación
  - Desarrollador de la plantilla base del proyecto

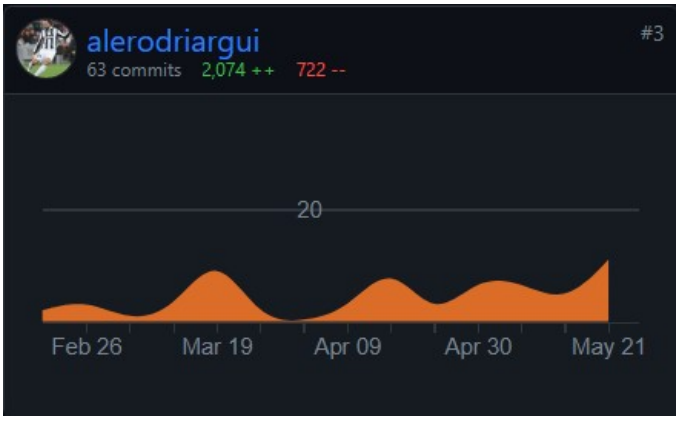

Figura 9. Contribución en el proyecto de Alejandro Rodríguez

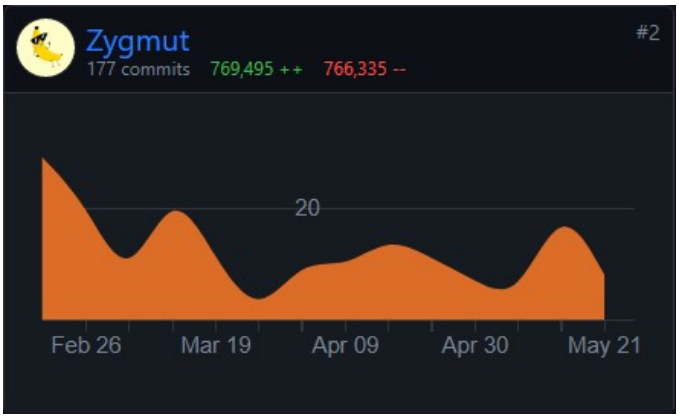

Figura 10. Contribución en el proyecto de Rubén Palmer

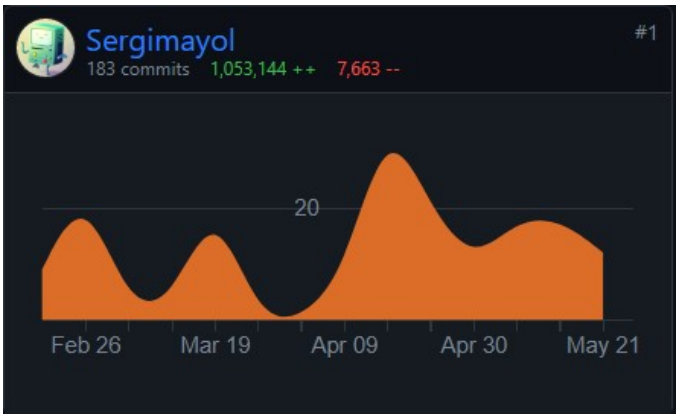

Figura 11. Contribución en el proyecto de Sergi Mayol

RECONOCIMIENTOS

- Se agradece la colaboración del Dr. Miquel Mascaró Portells en la resolución de dudas y supervisión del proyecto.

REFERENCIAS

[1] Better Swing, *An easy way to develop java GUI apps*, V0.0.3. Ver documentación completa aquí.
[2] Graphical engine, *How to develop a graphical engine from scratch*. Ver enlace aquí.
[3] Google json (Gson), *Java serialization/deserialization library to convert Java Objects into JSON and back*. Ver enlace aquí.
[4] Node.js Architecture, *Architecture of a single thread cross-platform, open-source server environment*. Ver enlace aquí.
[5] The Levenshtein distance algorithm, *How it works and what are its applications*. Ver enlace aquí.
[6] The SQLite database, a serverless database. *Quick start and documentation*. Ver enlace aquí.
[7] Open office complete dictionaries. *Complete dictionaries of the 10 used*. Ver enlace aquí.
[8] Open source Java graph library. *An easy way to create and visualize a graph in Java*. Ver enlace aquí.
[9] Open source Java chart library. *An easy way to create and visualize charts in Java*. Ver enlace aquí.
[10] A way to define different behavior for a method with the same name. Ver enlace aquí
[11] A structure used for evolution analysis. Ver enlace aquí
[12] Find a minimum spanning forest of an undirected edge-weighted graph. Ver enlace aquí
[13] Generative models, *Naive Bayes classifier*. Ver enlace aquí