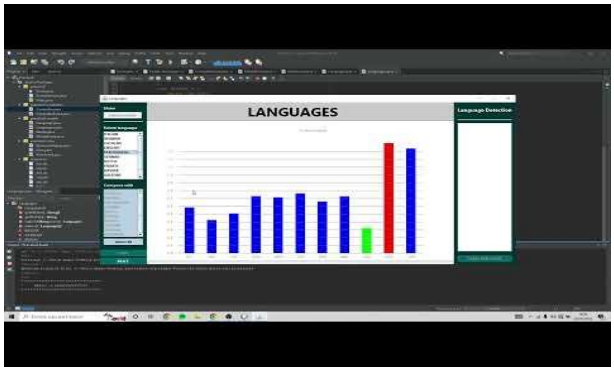
Programación Dinámica

Martin Mitrovski Delov, Josep Antoni Naranjo Llompart, Sergi Moreno Pérez, Pau Rosado Muñoz

Resumen — El ejercicio consiste en desarrollar un programa capaz de encontrar la distancia en la que e encuentran dos idiomas. Mediante el algoritmo dinámico de Levenhstein compararemos cada palabra del diccionario del primer idioma con cada palabra del diccionario del segundo idioma. La distancia final será mostrada en la vista al usuario.

Por motivos de tamaño del fichero de la práctica, hemos decidido subir el vídeo a YouTube y no incluirlo en el archivo comprimido entregado. El enlace al vídeo de la práctica en YouTube es el siguiente:



Practica5

I. INTRODUCCIÓN

En este artículo se explicará cómo se ha realizado la cuarta práctica de la asignatura Algoritmos Avanzados. Se desarrollarán algunos conceptos teóricos expuestos en el quinto tema y también se describirá la implementación de una aplicación que calcula la distancia entre dos idiomas comparando cada palabra del diccionario de cada uno. Para ello hacemos uso de la programación dinámica, en concreto el algoritmo dinámico de Levenhstein.

Los puntos por tratar son los siguientes: el entorno de programación utilizado, los conceptos de MVC, el patrón por eventos, la programación dinámica y sus características, y la implementación del MVC para el desarrollo de nuestra aplicación junto a todos los matices necesarios.

Por parte del concepto MVC, explicaremos qué es y por qué es el método de diseño que utilizaremos durante todo el curso para la implementación de las prácticas.

Explicaremos en detalle qué es y qué podemos esperar de la programación dinámica, concepto clave de este quinto tema.

Una vez explicadas estas secciones teóricas, procederemos a mostrar la implementación de la aplicación usando el patrón de diseño MVC y aplicando la técnica del algoritmo dinámico de Levenhstein para conseguir la distancia que hay entre un idioma y otro. Explicaremos cada una de las partes, junto a los conceptos de código más importantes. Cabe destacar que haremos bastante hincapié en el nivel de abstracción que se ha perseguido tener en la implementación y comunicación entre los diferentes componentes de la arquitectura. Primero explicaremos el modelo de datos, después el controlador y finalmente la vista o interfaz gráfica.

Del apartado gráfico, explicaremos algunas características para asegurar la consistencia del programa.

Para acabar habrá un apartado de conclusiones dónde se realiza una breve reflexión sobre los resultados obtenidos y sobre el trabajo dedicado al proyecto.

II. ENTORNO DE PROGRAMACIÓN

En nuestra práctica usamos el entorno de programación NetBeans, entorno de desarrollo integrado libre, orientado principalmente al desarrollo de aplicaciones Java. La plataforma NetBeans permite el desarrollo de aplicaciones estructuradas mediante un conjunto de componentes denominados “módulos”. La construcción de aplicaciones a partir de módulos permite que estas sean extendidas agregándoles nuevos componentes. Como los módulos pueden ser desarrollados independientemente, las aplicaciones implementadas en NetBeans pueden ser escaladas fácilmente por otros desarrolladores de *software*.

III. MODELO VISTA CONTROLADOR (MVC)

El modelo-vista-controlador (MVC) [1] es una arquitectura de software que separa los datos, la interfaz de usuario y la lógica de control en tres componentes distintos. El modelo representa los datos, la vista muestra los datos al usuario de forma visualmente atractiva y el controlador actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos. El flujo de una aplicación que usa el MVC comienza con la interacción del usuario a través de la interfaz de usuario, entonces el controlador recibe la acción del usuario y actualiza el modelo acorde al nuevo estado del

programa, seguidamente la vista utiliza los datos del modelo para actualizar la interfaz.

Existen tres formas básicas de implementación del MVC: distribuido, centralizado y centralizado en el controlador. El uso de este patrón de diseño por capas tiene varias ventajas, como la división del trabajo, la reutilización de código, la flexibilidad y escalabilidad que supone, así como la facilidad de realizar el *testing* de la aplicación. Sin embargo, también tiene algunas desventajas, como la complejidad adicional que puede presentar debido a que requiere más código y esfuerzo para mantener la separación de tareas entre los componentes.

En la Figura 1 aparece el esquema habitual del patrón de diseño MVC.

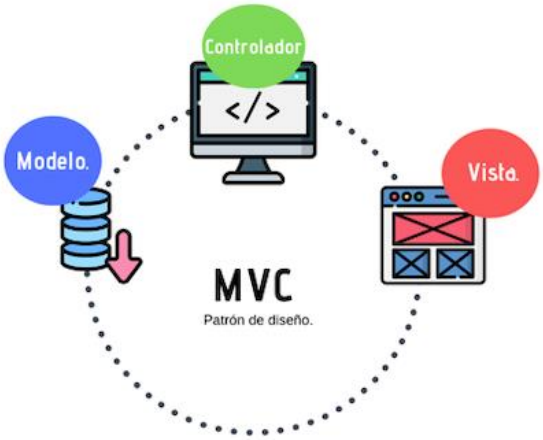


Figura 1: Modelo Vista Controlador

IV. PATRÓN POR EVENTOS

La arquitectura orientada a eventos o *Event-driven architecture* (EDA) es una forma de diseñar software que se enfoca en los eventos que ocurren dentro del sistema [2]. Los eventos son ocurrencias notificadas, como cuando un usuario hace clic en un botón. En una arquitectura EDA, los componentes del sistema están diseñados para reaccionar a estos eventos en tiempo real. Los componentes se comunican entre sí mediante eventos como se muestra en la Figura 2, propagándose a los componentes relevantes que responderán al evento adecuadamente.

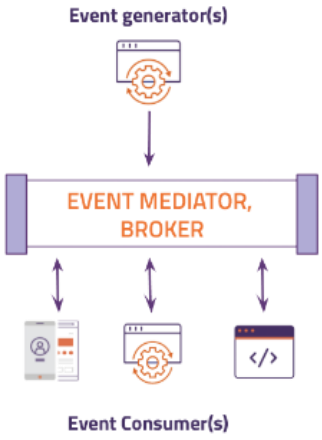


Figura 2: Patrón por eventos

La arquitectura de eventos es especialmente útil [3] en sistemas que requieren procesamiento en tiempo real, escalabilidad y flexibilidad, como redes de sensores, comercio financiero, sitios web de comercio electrónico y redes sociales. Los sistemas basados en esta arquitectura pueden reaccionar rápidamente a cambios y son tolerantes a fallos debido al bajo acoplamiento entre los componentes.

Sin embargo, esta arquitectura también puede tener desventajas, como la complejidad y dificultad de depuración debido a la naturaleza distribuida del sistema y la necesidad de sincronización de los eventos. También se deben tener en cuenta posibles problemas de latencia, ya que los eventos pueden tardar un cierto tiempo en propagarse a través de los componentes.

V. PROGRAMACIÓN DINÁMICA

La programación dinámica es una técnica para resolver problemas complejos de manera eficiente aportándonos una optimización sobre la recursividad simple. Donde sea que veamos una solución recursiva que tiene llamadas repetidas para las mismas entradas, podemos optimizarla usando programación dinámica.

Se basa en descomponer un problema en subproblemas más pequeños y solucionarlos de manera recursiva, almacenando los resultados de los subproblemas para evitar recalcularlos. Esta sencilla optimización reduce las complejidades del tiempo exponencial a polinomial.

Como se ha dicho, los resultados se almacenan, lo cual permite evitar la repetición de cálculos innecesarios y esto ahorra tiempo de ejecución y mejora la eficiencia del algoritmo.

V.I ¿CUÁNDO SE PUEDE USAR LA PROGRAMACIÓN

DINÁMICA?

La programación dinámica solo se aplica a problemas que exhiben la propiedad de superposición de subproblemas, es decir, cuando un problema se puede descomponer en subproblemas más pequeños que se solucionan de forma independiente. Además, es fundamental definir una estructura de almacenamiento adecuada para guardar los resultados de los subproblemas y evitar recálculos innecesarios.

Este tipo de programación es de especial utilidad en la solución de problemas de optimización en donde se desea averiguar un máximo y un mínimo (principio de Bellman).

V.II PRINCIPIO DE OPTIMALIDAD DE BELLMAN

El principio de optimalidad de Bellman [5] es un concepto fundamental en la programación dinámica que fue desarrollado por Richard Bellman y establece que una solución óptima a un problema puede ser descompuesta en subproblemas óptimos más pequeños y, a su vez, las soluciones óptimas a esos subproblemas deben ser consistentes con la solución óptima global.

Es decir, el principio de optimalidad de Bellman dice que, si tenemos una secuencia de decisiones óptimas para resolver un problema, entonces cualquier subsecuencia de esas decisiones también debe ser óptima para el subproblema correspondiente.

Este principio es la base para la técnica de programación dinámica ya que, al utilizar dicha técnica, se puede descomponer un problema en subproblemas más pequeños, resolver cada subproblema de forma óptima y combinar las soluciones de los subproblemas para obtener la solución óptima global.

El principio de optimalidad de Bellman se expresa mediante la ecuación de Bellman, una ecuación funcional que establece la relación entre el valor óptimo de un problema y los valores óptimos de sus subproblemas. La ecuación de Bellman es especialmente útil en la resolución de problemas de optimización, donde se busca maximizar o minimizar una función objetivo.

V.III ESQUEMA GENERAL

El enfoque que tiene este tipo de programación se puede resumir en cuatro pasos:

1. Definición del problema: primero se debe definir el problema, identificar las entradas y salidas y los criterios de optimización.

2. Identificación de la estructura recursiva: lo siguiente que se debe hacer es buscar una relación recursiva entre el problema original y subproblemas más pequeños. Esto implica dividir el problema en partes más pequeñas que puedan resolverse de forma independiente.

3. Definición de la función de coste: se debe definir una función de coste que cuantifique la solución óptima al problema. Esta función puede ser el valor máximo, el valor mínimo o cualquier otro criterio de optimización definido por el problema.

4. Resolución de los problemas y almacenamiento de resultados: finalmente se resuelven los subproblemas de forma recursiva, comenzando desde los más pequeños y avanzando hacía los más grandes. Los resultados se almacenan en una tabla o matriz, de modo que cuando se necesiten los resultados de un subproblema en particular, se pueda acceder directamente a ellos en lugar de recalcularlos.

V.IV EJEMPLO

Un ejemplo típico para apreciar el poder de la programación dinámica es el caso del cálculo de los números de Fibonacci. Este problema puede ser resuelto rápidamente con una solución recursiva simple, no obstante, el coste temporal es de orden exponencial ya que se deben repetir una gran cantidad de cálculos. Es por este motivo por el que la programación dinámica es de gran utilidad, mediante el almacenaje de los resultados de los mencionados cálculos se evita que se tengan que volver a calcular nuevamente, por lo que el coste de tiempo pasa de exponencial a un coste lineal.

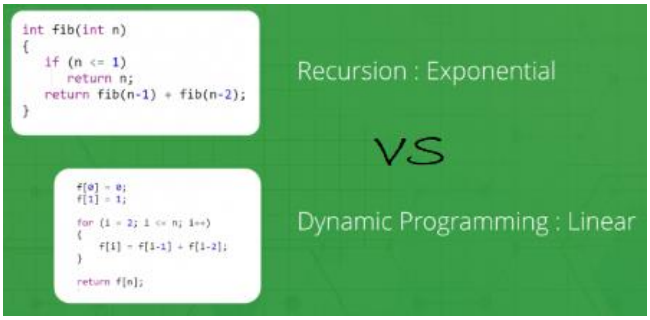


Figura 3: Comparación del problema de cálculo Fibonacci mediante recursividad simple y programación dinámica

VI. IMPLEMENTACIÓN MVC

Como ya se ha comentado anteriormente en este informe existen tres tipos de variaciones según respecta a la implementación del patrón “Modelo Vista Controlador”. En esta práctica, debido a la facilidad que ofrece a la hora de gestionar los distintos módulos, así como para facilitar la compresión de los distintos componentes sin mezclar

funcionalidades, se ha optado por desarrollar el programa siguiendo un esquema centralizado.

Para refrescar cabe comentar que este esquema persigue que la gestión de la comunicación entre las partes implicadas pase por un intermediario común. Este centro de control se responsabiliza también del ciclo de vida de los componentes.

Este "manager" central se ha implementado en la clase **Main,** a partir de la cual empieza a ejecutarse el software una vez se inicia el ejecutable.

El elemento que permite que la vista, el controlador y el modelo se puedan comunicar entre sí es una interfaz llamada **EventListener** que obliga a implementar la función "**notify**". Mediante el uso de la clase abstracta **Event**, el **main** podrá distinguir cuál de los componentes ha notificado el evento y redireccionarlo según corresponda. Esta solución es posible ya que cada componente implementa la misma interfaz, por lo que también deben implementar su propia versión de "**notify**", la cuál será llamada desde el **main** cuando éste decida que un evento debe ser tratado por la correspondiente parte. Por tanto, en el **main** se tendrá una instancia de cada componente de la arquitectura.

La arquitectura descrita permite una gran versatilidad a la hora de implementar los distintos elementos por separado debido a que existe un gran desacople en el sistema de comunicación. Esto ayuda a la hora de desarrollar un módulo sin tener que pensar en cómo afectarán las nuevas funcionalidades en la lógica global.

VI.I MODEL

El modelo es una parte fundamental de cualquier aplicación, ya que es el componente encargado de almacenar y gestionar todos los datos que serán escritos o leídos por los usuarios, tanto de manera directa como indirecta. En otras palabras, el modelo es el lugar donde se guardan y organizan los datos que serán utilizados para realizar diferentes operaciones y análisis en la aplicación.

En el caso específico de esta aplicación, el modelo se representa como un conjunto de diccionarios que tienen cargado en memoria sus respectivas palabras. Estos diccionarios son implementados mediante una clase llamada Language que se detalla más adelante. A continuación, se describen todos los objectos necesarios para definir el modelo y operar con él.

ModelEvent

La clase ModelEvent hereda de **Event** y se utiliza para que el **main** notifique al Model cuando se produzca un evento que este deba gestionar. Además, en él aparece un enumerado ModelEventType para diferenciar el tipo de ModelEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro del modelo.

El ModelEventType compone los eventos de modelo:

- START_DICTIONARIES
- START_TEXT

Los atributos de ModelEvent son el tipo para representar el ModelEvent que se haya notificado y dos variables de tipo entero a modo de índice llamadas "lanCompare" y "lanCompareWith", la primera indica qué lenguaje se quiere comparar y la segunda con qué lenguaje o lenguajes se quiere comparar el anterior. Ambas variables pueden tomar también el valor negativo -1, en el caso de la primera esto querría indicar que se deben comparar todos entre todos y en el caso de la segunda que el lenguaje que indica la primera variable se compara con todos los lenguajes posibles, él mismo inclusive. Adicionalmente también declara un arreglo de tipo string llamado "words" y que, como su propio nombre indica, pretende almacenar palabras pertenecientes a un texto al recibirse el evento por el cual se debe calcular en qué idioma está escrito el texto introducido por el usuario.

Language

La clase Language representa la entidad del diccionario, es decir, tiene el cometido de almacenar toda la información referente a un idioma.

Para ello hace uso de dos variables de tipo string: una llamada "name", que almacena el nombre del idioma que viene a representar dicha clase; y otra llamada "dicFile", que contiene el nombre junto a la extensión que toma el archivo que almacena las palabras del idioma y que es de dónde se extraen estas al instanciar el objeto (Proceso explicado a continuación). A parte, al igual que en la clase comentada anteriormente, también se almacena un *array* de palabras con la intención de que se tengan todas ellas cargadas desde el archivo en memoria a lo largo de la ejecución.

Al instanciarse un objeto de este tipo se reciben dos argumentos de tipo *String* a través del constructor, el primero da valor al nombre del lenguaje y el segundo asigna cuál es el nombre del archivo que le corresponde, y entonces se inicia el proceso de extracción de los datos del fichero. Se crea un objeto *File* cuyo nombre ya es sabido y luego se usa un objeto *Scanner* para ir recorriendo fila a fila sacando las palabras que iterativamente se van guardando en una lista dinámica temporal llamada "initWords". Al final de la lectura se instancia el arreglo definido anteriormente llamado "words" y en él se vuelca el contenido de la lista. El uso de la lista dinámica viene dado debido a que no se conoce la dimensión

del fichero, por lo que una solución dinámica resuelta de mayor conveniencia. Su posterior volcado en el arreglo es causa de la menor memoria necesario para conservar las palabras en el arreglo que si se almacenaran en la lista.

Por último, cabe mencionar que esta clase contiene varias funciones "getters" que permiten consultar el valor de todos los atributos a terceros:

- getName()
- getPath()
- getLength()
- getWord(int i)

Languages

Languages es una clase de tipo enumerada que permite diferenciar entre los distintos tipos de idiomas que se manejan en el programa. En nuestro caso, hemos añadido 12 idiomas europeos:

- ITALIAN
- SPANISH
- CATALAN
- ENGLISH
- PORTUGUESE
- GERMAN
- DUTCH
- FRENCH
- BASQUE
- GALICIAN
- RUSSIAN
- UKRANIAN

Por su parte, en esta misma clase implementa una función llamada "getDicFile" que devuelve el nombre del fichero correspondiente al valor del objeto en cuestión y otra llamada getAllFiles() que devuelve un arreglo de tipo *String* con todos los idiomas implementados, útil para que se puedan desplegar en la componente View los distintos tipos de idiomas de la aplicación.

Model

El modelo, como se ha comentado anteriormente, contiene el conjunto de lenguajes a comparar. Es por ello por lo que como atributo principal de este se tiene un *array* de tipo **Language**, llamado "dictionaries" y que contiene todos los diccionarios cargados en memoria dinámica de manera que posteriormente el controlador pueda realizar la comparación que sea necesaria con ellos. Esta estructura se inicializa en el constructor añadiendo iterativamente todos los lenguajes definidos en el tipo **Languages**. Por otro lado, también almacena los mismos atributos que el **ModelEvent**, es decir, dos índices para los idiomas a comparar y un arreglo para las palabras del texto del cuál se quiere saber su idioma. Por último, se debe mencionar que el modelo contiene una referencia al **main** la cual se inicializa en el constructor mediante paso por parámetro.

Una vez se han inicializado las estructuras necesarias para la ejecución del programa el **model** permite la consulta de los datos que estas almacenan mediante un conjunto de funciones públicas que se definen a continuación:

- getLanguageCompared(): Devuelve el valor de la variable "lanCompare", es decir, permite saber qué lenguaje se ha seleccionado para comparar.

- getLanguageToCompare(): Devuelve el valor de la variable "lanCompareWith", es decir, consulta con qué lenguaje se va a comparar el primero seleccionado.

- getLanguageLength(int i): Recibe como parámetro una variable que actúa como índice para indicar, en el caso de que sea -1, si queremos devolver la longitud del arreglo "words" que contiene las palabras de cuyo texto se quiere saber el idioma o, de lo contrario, nos interesa saber la longitud del conjunto de palabras del idioma almacenado en la posición que refiere el índice dentro del arreglo de diccionarios.

- getLanguageWord(int i, int w): Este método, al igual que el anterior, vuelve a recibir un índice i para consultar el idioma correspondiente. Como antes, si se trata de un valor -1 se consulta el texto. Pero, adicionalmente también recibe otro índice w para obtener, dentro del conjunto de palabras indicado, la palabra en concreto que ocupe la posición respectiva.

- getNLanguages(): Devuelve el número de lenguajes que se almacenan en el arreglo "dictionaries" para comparar.

- getLanguageComparedName(): Haciendo uso del índice "lanCompare" devuelve el nombre del lenguaje que se usará como punto de partida en la comparación.

- getLanguageToCompareName(): Homónimamente a al anterior usa el índice "lanCompareWith" para obtener el nombre del lenguaje con el que se va a realizar la comparación.

- getLanguageName(int i): Recibe como input un índice para consultar el lenguaje correspondiente en el arreglo "dictionaries" y de allí devolver su nombre.

- isLanguageDetection(): Devuelve un booleano que indica si se ha inicializado el arreglo de palabras "word". Esto permite al controlador saber si en ese

instante de la ejecución se debe realizar la operación de calcular en qué idioma está escrito un texto o no.

- isSameLanguage(int i): Recibe un índice por parámetro y devuelve verdadero si ese índice coincide con el de la variable "lanCompare", es decir, si la palabra que representa ese índice es la que se va a usar como base para la comparación.

- compareAll(): Devuelve un booleano que indica si se deben comparar todos los lenguajes entre todos.

- compareWithAll(): Devuelve un booleano que permite saber si el lenguaje usado como base en la comparación se debe comparar con todos los lenguajes o no.

Por último, la clase Model implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ModelEventType recibido: START_DICTIONARIES elimina la referencia al arreglo "words" indicando que la operación que se va a realizar es la de la comparación de idiomas e asigna nuevos valores a las dos variables índice "lanCompare" y "lanCompareWith" y START TEXT simplemente clona el arreglo de "words" a partir del recibido a través del evento.

VI.II CONTROLLER

El controlador es el componente con el rol de bisagra situada entre la vista y el modelo de datos previamente explicado. Es decir, es el encargado de gestionar el flujo de datos entre estos dos módulos. Este flujo de datos es transformado mediante operaciones ejecutadas por el mismo controlador y visualizado en la vista o almacenado en el modelo.

En esta aplicación el controlador se responsabiliza de ejecutar el algoritmo de programación dinámica de Levenshtein. Este algoritmo es conocido también como la distancia de edición y lo usamos para poder realizar la operación de comparación de N a N lenguajes. Adicionalmente también se ha implementado la capacidad de detección del lenguaje, es decir, el usuario puede insertar un texto y el controlador debe adivinar en qué idioma ha sido escrito.

A grandes rasgos este algoritmo compara cada palabra de un idioma con todas las de otro y devuelve un valor que representa la distancia de edición media, o lo que es lo mismo, cuantas modificaciones de media se deberían hacer sobre las palabras del primer idioma para que este se convirtiese en el segundo.

ControllerEvent

La clase ControllerEvent hereda de **Event** y se utiliza para que el **main** notifique al Controller cuando se produzca un evento que este deba gestionar. Además, en ella aparece un enumerado ControllerEventType para diferenciar el tipo de ControllerEvent que se recibe. De esta manera, se simplifica la identificación de tipo de evento dentro del Controller.

El ControllerEventType compone los eventos de controlador:

- START
- STOP

Los atributos de ControllerEvent son el tipo para representar el ControllerEvent que se haya notificado y una variable booleana que indica si se ha iniciado ya la ejecución del cálculo.

Controller

El controlador consta de una serie de atributos de clase para gestionar su funcionamiento. En primer lugar, contiene dos variables para almacenar referencias a instancias de otras clases, una que referencia al **main** que se inicializa al pasársela como parámetro en el constructor y una para Model que se inicializa justo antes de la ejecución del algoritmo. Esta segunda referencia se usa debido a que el modelo no se modifica durante el transcurso de la ejecución del algoritmo y resulta de utilidad ya que se accede a sus atributos a lo largo de toda la clase Controller. También son atributos de esta clase un objeto de tipo Thread que almacena la meta-data sobre el hilo de la ejecución del algoritmo.

El controlador hace uso de una función para que al lanzar el *thread* este pueda ejecutar el código del algoritmo. La función en cuestión se llama "run()" y realiza un *Override* ya que se implementa mediante la interfaz "Thread". Lo primero que hace es obtener la referencia del modelo, como se ha explicado anteriormente, y todo seguido hace la comprobación sobre si se quiere detectar el idioma según un texto escrito o si se quiere comparar los diccionarios, es decir, si se lanza el programa detectLanguage() o la función compareDictionaries(), de las cuales hablaremos a continuación.

En primer lugar, hablaremos del algoritmo detecLanguage(), encargado de comparar todos los diccionarios con el texto escrito para identificar que idioma se asemeja más, y el cual contiene una variable entera "nFiles", la cual almacenará el número de diccionarios que recibe del modelo gracias a la función getNLanguages() explicada anteriormente; una variable "results" que se trata de un arreglo de *doubles*, de tamaño igual a "nFiles", que servirá para almacenar la distancia que hay entre cada lenguaje y el texto; un entero

"index" inicializado a -1 que será de utilidad para almacenar cuál es el índice del idioma más 'cercano' al texto; y finalmente, se tiene una variable *double* "min" inicializada con el valor máximo que este tipo de dato puede tener y que servirá para mantener el valor de la menor distancia con el texto que se ha calculado. Una vez explicadas las variables, se encuentra un bucle *for* que se itera tantas veces como la variable "nFiles", es decir, tantas veces como diccionarios tiene el programa y en el cual se calcula la distancia que hay entre un idioma y el texto escrito, indicando que se trata de una comparación con texto y no con otro diccionario mediante el paso por parámetro del valor –1 al método calculateDistance(int d1, int d2) del cual hablaremos más adelante. Una vez calculadas las dos distancias, tanto del diccionario hacia el texto como del texto hacia el diccionario, y almacenadas en dos variables *double*, "result1" y "result2", se calcula la distancia final haciendo la raíz cuadrada de la suma de los cuadrados de las variables "result1" y "result2", operación la cual se justifica ya que el cálculo no es bidireccional por motivos tanto de tamaño o de diferencia en los caracteres por eso para encontrar un valor justo se decidió aplicar la media geométrica para balancear el resultado el qual se almacena dentro de la variable "result". Para acabar el bucle, se hace la comprobación si la distancia almacenada es menor que la distancia almacenada en la variable "min" anteriormente y se guarda la posición dentro del *array* en la variable "index". Finalmente se notifica a la vista cual idioma se cree que es enviando el resultado de la función del modelo getLenguageName(int i) pasando le por parámetro la variable "index", con lo cual nos retornaría el nombre del idioma que más se asemeja al texto escrito. Cabe destacar que, durante la fase de pruebas, se detectaron varias ocasiones donde el programa no detecta bien el idioma del texto cuando la longitud de este es muy reducida (menor a 5 palabras) y se usan nombres propios los cuales pertenecen a otro idioma, un ejemplo puede ser la frase 'Hola, soy Pau.' la cual la detecta como francés o 'Soc na Stefani' la cual detecta como holandés. En textos con mayor longitud, el porcentaje de error es ínfimo, tanto que si existe no hemos sido capaz de encontrarlo.

A continuación, el método compareDictionaries() el cual nos compara dos idiomas, para ello primero hace la comprobación si se ha elegido comparar el diccionario con todos (usando los métodos del modelo compareAll()), en ese caso se iniciará una serie de variables las cuales son "nFiles", exactamente igual a la explicada anteriormente; un entero llamado "nValues" el cual se inicia a 0 y servirá para comunicar a la vista cuantos idiomas hay que representar; también se inicia un *long* llamado "time" para tener en cuenta el tiempo de ejecución; y finalmente se inicia un *array* bidimensional de *double* llamado "result", de tamaño "nFiles", que almacenara los resultados de la distancia. Se itera en un doble bucle *for* una cantidad de nFiles – 1, ya que el diccionario a comparar no se tiene en cuenta, donde se calcularán las dos distancias, la distancia final (mismo proceso explicado en la función anterior) y se incrementara la variable "nValues". Una vez finalizado el doble bucle, que calcula el tiempo de ejecución restando el tiempo actual con el tiempo almacenado en la variable "time" y se muestra por pantalla antes de notificar a la vista los resultados. En el caso que la condición no sea verdadera, se llegará a la segunda condición la cual comprueba si se tiene que comparar el diccionario con todos los demás (compareWithAll()) y en la cual se inicia igualmente la variable "nFiles" pero en este caso la variable "results" vuelve a ser un *array* unidimensional de *doubles* con la diferencia que ahora es de un tamaño de "nFiles" – 1 ya que no se contemplará la comparación con si mismo porque es trivial e inocua, y contado con una nueva variable booleana llamada "repeated" con valor igual a *false*. Se vuelve a iterar en un bucle simple de tamaño "nFiles" y antes de calcular las distancias se hace la comprobación de saber si el diccionario a tractar es el mismo que se ha seleccionado para comparar (con la función del modelo isSameLanguage(int i)), si ese es el caso, se pondrá a *true* el valor de "repeated" y se romperá la iteración con un *continue*. A continuación, se hará el cálculo de la distancia y antes de almacenarla se comprobará si "repeated" es *true* ya que si ese es el caso se tendrá que almacenar el resultado en la posición i-1 del *array*, ya que al no hacer la comprobación con si mismo pudiera quedar una posición vacía y finalmente se notifica de los resultados a la vista. Si ninguna de las dos condiciones anteriores se cumple, debe ser porque el usuario ha querido comprar dos diccionarios, por lo tanto, se crea la variable "result" como un arreglo de *double* de dimensión 1, se hace el cálculo de la distancia y se envía a la vista.

En los métodos anteriores se ha nombrado el cálculo de la distancia, cuya responsabilidad cae en el programa calculateDistance(int d1, int d2) el cual recibe el índice de los dos diccionarios a comparar (o palabras del texto a reconocer) y el cual cuenta con tres variables, un *double "*acc1" el cual se inicia a 0 y servirá para almacenar el valor de la distancia entre los diccionarios; y dos variables enteras llamadas "lenghtDX"(1 y 2) las cuales representan la dimensión del diccionario (o del texto a clasificar). Con un bucle *for* de longitud igual al tamaño del diccionario a comparar se extrae la primera palabra y se define una variable entera "minDistance" como el valor máximo de este tipo de dato, la cual servirá para almacenar la menor distancia entre la palabra y el diccionario y su iniciación a este valor viene a que siempre será mayor a cualquier distancia que se calcule así el primer valor a calcular siempre le remplazará. A continuación, se encuentra el siguiente bucle *for* con la longitud del segundo diccionario ya que se iterará la palabra seleccionada anteriormente con todas las palabras de este diccionario, calculado la distancia entre ellas mediante el algoritmo de Levenshtein y almacenando en la variable "minDistance" la menor de ellas exceptuando el caso de que se encuentre con la misma palabra, ya que en este caso la iteración del bucle se detendría y se daría paso a la siguiente palabra del primer diccionario. Una vez finalizada la comparación de una palabra

con un diccionario entero, crea la variable *double "*div" la cual es el resultado de dividir la menor distancia entre la longitud de la palabra, quitando así las posibles penalizaciones a palabras de mayor longitud frente a palabras más breves, además de incrementar el contador "acc1" con el valor de esta. Finalmente, el método retorna la división de la variable acc1 entre la longitud total del diccionario, evitando así la penalización a los idiomas con mayor número de palabras.

El algoritmo de Levenshtein (levenshteinDistance(String w1, String w2) es el mayor atractivo del programa y su funcionamiento es el siguiente. Primero se construye una matriz de enteros, "solMatrix", tamaño igual a la dimensión de las dos palabras + 1 y la rellenaremos la primera fila y columna con dos bucles *for* con valores numéricos en orden ascendente (de 0 al tamaño de la palabra – 1). A continuación, se entra en un doble bucle para iterar por toda la matriz y se va extrayendo la última letra de las palabras a tractar además de ir creando variables enteras para almacenar el coste de las interacciones, tanto para "delete"; como para "insert"; como para "replace"; al igual que el coste de cada operación, "cost". Si se da el caso en que las dos letras son las mismas, el coste de la operación es 0, en caso contrario, seria 1 y se daría comienzo al cálculo del coste del resto las interacciones sumando el coste de la interacción y el valor de la posición de la matriz adecuado a cada una, en el caso de "delete" está seria la posición anterior en la misma columna; en el caso de "insert" la posición anterior de la misma fila; y en el caso de "replace" en seria el valor anterior en la diagonal, es decir, el valor de la fila y columna anterior. Una vez calculado los costes de cada operación. Se buscaría con la función *min(int i, int j)* de la *Math* el valor mínimo entre las 3 operaciones y este sería el nuevo valor de la posición de la matriz. Una vez finalizado el doble bucle, se harán actualizado todas las posiciones de la matriz y esta seria devuelta al programa donde ha sido solicitada.

Finalmente cabe destacar que la clase implementa la interfaz EventLisener con la cual permite la comunicación del controlador con la vista y modelo, y por lo tanto esta clase reescribe el método notify(Event e) con el cual se puede tanto lanzar el proceso como interrumpirlo.

VI.III VIEW

La vista representa la interfaz de usuario de la aplicación y se encarga de presentar los datos al usuario final. La vista no interactúa directamente con el modelo ni con el controlador, sino que se comunica con ellos a través del gestor de eventos.

DistanceDisplay

Esta clase extiende de la clase JPanel y se utiliza para mostrar los gráficos y el grafo de los resultados dados en el Controller.

El constructor de esta clase recibe como parámetro una referencia al Model para facilitar la consulta del modelo y poderlo pintar de manera adecuada, un *array* y una matriz de *double* llamados "graphic" y "graph" respectivamente para realizar la visualización de los resultados obtenidos, dos *booleanos* que sirven para mostrar o no el grafo y el gráfico llamados "showGraph" y "showGraphic", una variable *BufferedImage* llamada "bima" que servirá para implementar la técnica de doble *buffering*, un *array* de Node llamada "circles" para representar las circunferencias del grafo, un JList llamado "list" y un entero "nValues" para saber cuántas arestas se deben dibujar en el grafo.

El método paint() de la clase utiliza la técnica de doble *buffering* para visualizar el grafo y los gráficos. Los nodos del grafo se representan como circunferencias rojas las cuales están comunicadas todas con todas con un camino el cual está marcado con índice y en el JList mostraremos ese índice para saber de qué nodo a que nodo va ese camino y su valor. Para los gráficos en cambio, cuando seleccionemos la lengua que queremos comparar con otra lengua o con todas las posibles lenguas, dibujará un gráfico de barras azules, pero en caso de seleccionar que compare con todas las lenguas, habrá dos barras diferenciadas: la menor que estará de color verde y la mayor que estará de color rojo.

También contamos con métodos auxiliares como repaint() para volver a pintar la pantalla, reset() para eliminar la solución encontrada, setList para configurar la lista de componentes JList, printGraphic() y printGraph() que se utilizan para mostrar el grafo y los gráficos de distancias en el panel; roundDouble() sirve para redondear los números y setTopValue() para establecer el valor máximo para escalar los gráficos; y por último tenemos la clase Nodo.

ViewEvent

La clase ViewEvent hereda de **Event** y se utiliza para que el **main** notifique a la View cuando se produzca un evento que esta deba gestionar. Además, aparece un enumerado ViewEventType para diferenciar el tipo de ViewEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro de la View.

El ViewEventType compone los eventos de modelo:

- SHOW_GRAPHIC
- SHOW_GRAPH
- SHOW_PANEL

ViewEvent tiene diferentes constructores, pero en todos necesita como atributos uno para almacenar el tipo de evento de vista. Luego hay un constructor que le pide por parámetro el valor de los gráficos. Luego otro constructor igual, pero con

el valor de todos con todos en una matriz y un entero para saber el número de arestas que se deben dibujar, y por último uno que pide por parámetro el nombre del idioma para imprimir en el panel.

View

La componente View envía tanto al Controller como al Model. Notifica al Controller únicamente cuando se pulse el botón para iniciar la ejecución de este dependiendo si queremos calcular el grafo o las gráficas de distancias y cuando se pulse el botón para detener su ejecución. Al Model se le notifica cuando se seleccionen las lenguas que queremos comparar.

La vista es responsable de presentar los datos de manera clara y coherente para el usuario. Esto incluye la disposición de la información, la selección de la paleta de colores, la presentación de los datos en diferentes formatos y la gestión de la interacción del usuario, además de poder apreciar el resultado del algoritmo.

En el constructor de la clase, solo se recibe la instancia del **main** con la cual se inicializa instancia del Model.

Podemos apreciar todos estos elementos en la interfaz:

1. buttonGraph: JButton encargado de calcular la distancia entre todos los lenguajes y mostrarlo en pantalla mediante un grafo.

2. List1, List2 y buttonList2: Los dos primeros son do JList que incluyen todos los lenguajes disponibles y debemos en list1 seleccionar el idioma que queremos comparar y en list2 el otro idioma a comparar, pero también tenemos la opción de pulsar el buttonLisr2, un JButton el cual seleccionará todos los lenguajes del list2 y así comparará un lenguaje con todo el resto.

3. graphList: Es un JList que será el encargado de mostrar los resultados a la hora de seleccionar la opción de calcular el grafo para aclarar mediante unos índices que habrá en las arestas, de que lenguaje (nodo) va a qué lenguaje esa aresta y su respectiva distancia.

4. textArea, textButton: JTextArea que permite al usuario escribir y que el programa detecte a que idioma corresponde el texto pulsando finalmente en el textButton, un JButton que iniciará el cálculo.

5. dDisplay: Al principio está vacío. Es una instancia de la clase DistanceDisplay, la cual extiende de la clase JPanel de Swing y es donde se mostrarán los gráficos y el grafo.

6. progressBar: Debajo del todo aparece la barra de progreso, una JProgressBar, que se mantiene inactiva hasta que el algoritmo se pone en ejecución. Una vez se detiene el algoritmo, también se detiene la barra.

7. JButton (Start): Se permite pulsar el botón mientras no haya ninguna ejecución del algoritmo iniciada siempre y cuando se haya seleccionado los lenguajes a comparar y esta comparación no sea de un lenguaje consigo mismo. Para conocer si se cumple esta segunda condición, se consulta a la instancia de Model.

8. JButton (Reset): Limpia todos los valores seleccionados en la vista y notifica al Model que debe hacer el *reset* así como también al Controller. No solo eso, otros componentes que serán actualizados son la 'graphList para eliminar la lista de soluciones y al 'dDisplay para que limpie la pantalla.

Los elementos de la vista previamente descritos se pueden observar en la Figura 4.

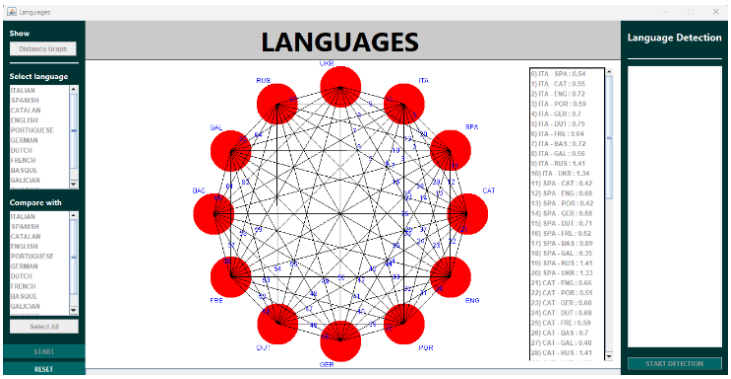


Figura 4: Vista de la aplicación, ejecutando el "Distance Graph"

La clase View implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realiza la operación para mostrar los gráficos, el grafo y el panel con las soluciones del grafo dependiendo de que acción ha tomado el usuario. También se detiene la 'progressBar'.

VII. CONCLUSIÓN

El equipo considera que la implementación del algoritmo dinámico de Levenhstein de la práctica utilizando el patrón de diseño MVC ha sido exitosa en términos de estructura y eficiencia. El uso del MVC junto a su consecuente abstracción

ha sido bastante buena ya que nos ha permitido separar la lógica del algoritmo de la interfaz de usuario sin interferir unas con otras, lo que ha resultado en un código más organizado y fácil de mantener. Todo lo anterior junto a la posibilidad de añadir más diccionarios equivale un programa robusto y escalable.

Conseguimos acabar nuestro quinto trabajo juntos, en el que hemos podido demostrar ya nuestra comprensión de la implementación del MVC, y donde el equipo ha funcionado bastante bien. El control de versiones mediante GitHub ha sido determinante, mejorando nuestros conocimientos de la plataforma y del comando *git* sobre todo. El Trello ha sido también una buena herramienta para poder ir notificando que cosas quedaban por hacer al equipo y que cosas se estaban haciendo o ya habían sido terminadas. Por los problemas que respectan a esta práctica, no ha habido muchos, considerando esta una práctica relativamente sencilla donde su mayor dificultad era entender bien el funcionamiento del algoritmo, pero una vez conseguido esto, era solamente plasmar los resultados.

En resumen, creemos que el resultado final conseguido por el equipo ha sido bastante satisfactorio.

VIII. ENLACES

1. Teoría del patrón MVC: https://developer.mozilla.org/en-US/docs/Glossary/MVC

2. Teoría del patrón por eventos: https://www.tibco.com/es/reference-center/what-is-event-driven-architecture

3. Ventajas y desventajas del patrón por eventos: https://es.wikipedia.org/wiki/Arquitectura_dirigida_por_eventos

4. Programación dinámica: https://www.geeksforgeeks.org/dynamic-programming/

5. Principio de Bellman https://www.sciencedirect.com/topics/engineering/bellmans-principle-of-optimality#:~:text=The%20primary%20idea%20of%20the,for%20every%20possible%20decision%20variable.

6. Librería que implementa la estructura de datos del Montículo de Fibonacci: https://github.com/trudeau/fibonacci-heap/tree/master/src/main/java/org/nnsoft/trudeau/collections/fibonacciheap