Universitat Politècnica de Catalunya - FIB/UPC

PROP - Projectes de Programació

Generador de teclados - primera entrega

Momin Miah Begum (momin.miah) , Muhammad Yasin Khokhar Jalil (muhammad.yasin.khokhar) , Jianing Xu (jianing.xu), Rubén Catalán Rua (ruben.catalan)
Cluster 23.5
Versión 1.0

Q1 2023-24

Índice

1. Casos de uso

1.1. Diagrama

Figura 1: Diagrama de casos de uso

1.2. Descripción de los casos de uso

Nombre	Crear Perfil
Actores	Usuario
Dependencias	
Precondición	No existe un perfil con el mismo nombre.
Descripción	Crea un perfil con un nombre de usuario y una contraseña.
Secuencia normal	El usuario inicia el programa y pide al sistema crear un perfil. El sistema le pide al usuario un nombre y una contraseña. El usuario proporciona un nombre de usuario y una contraseña. El sistema crea un perfil con los datos proporcionados por el usuario.
Postcondición	Se ha creado un perfil en el sistema.
Excepciones	▪ Ya existe un usuario con el nombre proporcionado. ▪ El formato del nombre no es correcto. ▪ La contraseña debe tener al menos 8 caracteres y un símbolo especial.

Nombre	Iniciar Sesión
Actores	Usuario
Dependencias	
Precondición	Existe un perfil con el mismo nombre creado.
Descripción	Carga todos los datos necesarios de un archivo .prop para el perfil sobre el cual se inicia sesión.
Secuencia normal	El usuario inicia el programa y pide al sistema iniciar la sesión. El sistema le pide al usuario un nombre y una contraseña. El usuario proporciona un nombre de usuario y una contraseña. El sistema carga los datos y la información relacionada al perfil sobre el cual se inicia la sesión.
Postcondición	Se ha iniciado la sesión de un perfil en el sistema.
Excepciones	▪ No existe ningún perfil con los datos del usuario. ▪ Los datos del archivo .prop no se han cargado correctamente.

Nombre	Modificar Perfil
Actores	Usuario
Dependencias	
Precondición	Existe un perfil con el nombre de usuario del usuario.
Descripción	Permite modificar cambiar la contraseña o el nombre de usuario o la información que contiene el perfil.
Secuencia normal	El usuario inicia el programa e inicia con sus datos de usuario (usuario + contraseña). El sistema permite al usuario modificar sus datos personales o modificar la información del perfil. El usuario indica al sistema lo que desea. Si desea modificar datos personales, el sistema le da la opción de modificar el nombre de usuario o la contraseña. Si desea modificar información del perfil, el sistema le da la opción de modificar/crear/borrar Teclados y Alfabetos. El sistema modificará el fichero .prop relacionado al usuario.
Postcondición	Se ha modificado el perfil o la información del usuario.
Excepciones	▪ No existe un usuario con el nombre proporcionado o la contraseña es incorrecta. ▪ El archivo .prop esta dañado o la lectura no se ha realizado correctamente.

Nombre	Borrar Perfil
Actores	Usuario
Dependencias	
Precondición	No existe un perfil con el mismo nombre.
Descripción	Borra un perfil existente del sistema.
Secuencia normal	El usuario inicia el programa y pide al sistema borrar un perfil. El sistema le pide al usuario un nombre y la contraseña del perfil a borrar. El usuario proporciona un nombre de usuario y la contraseña. El sistema borra el perfil con los datos proporcionados por el usuario. El sistema elimina la información del fichero .prop relacionado al usuario.
Postcondición	Se ha borrado un perfil en el sistema.
Excepciones	▪ No existe un usuario con el nombre proporcionado.

Nombre	Cargar un archivo .prop
Actores	Sistema
Dependencias	
Precondición	El usuario ha iniciado sesión previamente.
Descripción	Este caso de uso se encarga de cargar y leer información relacionada a un usuario. La información incluye entre otras cosas, detalles como número de teclados, su distribución, número de alfabetos, sus caracteres, y otros elementos asociados al usuario.
Secuencia normal	Buscar en el directorio del programa el archivo con el nombre y formato correcto.Empezar la lectura.
Postcondición	Se ha consultado la información almacenada respecto un usuario específico.
Excepciones	▪ El archivo está dañado. ▪ No existe ningún archivo con nombre y el formato de archivo especificado.

Nombre	Modificar un archivo .prop
Actores	Sistema
Dependencias	
Precondición	El usuario ha iniciado sesión previamente.
Descripción	Este caso de uso se encarga de escribir la información relacionada a las modificaciones que el usuario ha realizado. Estas modificaciones pueden ser añadir nuevo teclado, modificar teclado existente,borrar teclado, añadir alfabeto, modificar alfabeto, eliminar alfabeto.
Secuencia normal	Buscar en el directorio del programa el archivo con el nombre y formato correcto.Realizar la escritura.
Postcondición	Se han almacenado las modificaciones que el usuario ha realizado.
Excepciones	▪ El archivo está dañado. ▪ El nombre y el formato de archivo no son correctos. ▪ El archivo ya estaba siendo escrito. ▪ No hay suficiente almacenamiento para acabar de escribir las modificaciones.

Nombre	Crear teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un alfabeto en el sistema.
Descripción	Crea un teclado con un nombre, alfabeto disponible, texto, lista de palabras y algoritmo.
Secuencia normal	El usuario escribe el nombre del teclado a crear y el de uno de los alfabetos disponibles. Posteriormente, se escribe un texto y lista de palabras con frecuencias (formato <palabra> <entero>) y uno de los dos algoritmos disponibles (B&B o SA).
Postcondición	Se ha creado un teclado con los parámetros proporcionados.
Excepciones	▪ Ya existe un teclado con el nombre proporcionado. ▪ No existe un alfabeto con el nombre proporcionado. ▪ Formato incorrecto para lista de palabras. ▪ Nombre de algoritmo incorrecto. ▪ El nombre de teclado a crear no es válido (espacios, o nombre vacío).

Nombre	Modificar teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Modifica alguno de los atributos de un teclado existente.
Secuencia normal	Se escoge uno de los teclados creados y posteriormente se escoge una de las opciones mostradas para realizar la modificación: Cambio del alfabeto, texto, lista de palabras o algoritmo.
Postcondición	Se ha modificado el teclado especificado y se ha actualizado su distribución.
Excepciones	▪ No existe un teclado con el nombre proporcionado.

Nombre	Borrar teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Borra alguno de los teclados existentes.
Secuencia normal	El usuario escribe el nombre de un teclado. Si este existe, se borrará del conjunto de teclados del usuario. Si no, no se hará nada.
Postcondición	Se ha borrado el teclado especificado.
Excepciones	▪ No existe un teclado con el nombre proporcionado.

Nombre	Mostrar distribución de teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Muestra por pantalla la distribución de un teclado.
Secuencia normal	El usuario escribe el nombre de uno de sus teclados creados. Se muestra por pantalla la distribución eficiente de los símbolos de su alfabeto generada a partir de uno de los algoritmos.
Postcondición	
Excepciones	▪ No existe un teclado con el nombre proporcionado.

Nombre	Consultar teclados existentes
Actores	Usuario
Dependencias	
Precondición	
Descripción	Se escribe el nombre de todos los teclados existentes.
Secuencia normal	Se escoge la opción c̈onsultar teclados disponiblesÿ se muestran por pantalla el nombre de todos los teclados creados.
Postcondición	
Excepciones	▪ No hay ningún teclado creado.

Nombre	Intercambiar teclas (Modificar teclado)
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Intercambia la posición de dos teclas del teclado.
Secuencia normal	El usuario selecciona la opción Ïntercambiar teclas". El sistema pide al usuario el nombre del teclado objetivo. El usuario introduce el nombre del teclado. El sistema pide al usuario la fila y columna de los dos caracteres a intercambiar. El usuario proporciona los datos pedidos. El sistema intercambia la posición de las teclas especificadas.
Postcondición	Se ha actualizado la distribución del teclado especificado intercambiado las posiciones de las teclas especificadas.
Excepciones	▪ No existe un teclado con el nombre proporcionado. ▪ El rango de las teclas a intercambiar no es válido.

Nombre	Cambiar alfabeto (Modificar teclado)
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Cambia el alfabeto que reconoce el teclado.
Secuencia normal	Se escribe el nombre del alfabeto a utilizar a partir de ese momento para el teclado.
Postcondición	Se ha cambiado el alfabeto del teclado especificado y se ha actualizado su distribución.
Excepciones	▪ No existe un teclado con el nombre proporcionado. ▪ No existe ningún alfabeto con el nombre proporcionado.

Nombre	Cambiar texto (Modificar teclado)
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Cambia el texto que reconoce el teclado.
Secuencia normal	Se escribe el nuevo texto a utilizar para generar la distribución.
Postcondición	Se ha cambiado el texto del teclado especificado y se ha actualizado su distribución.
Excepciones	▪ No existe un teclado con el nombre proporcionado.

Nombre	Cambiar lista de palabras (Modificar teclado)
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Cambia la lista de palabras con frecuencia del teclado.
Secuencia normal	Mediante el formato <palabra> <frecuencia>, se escribe una nueva lista de palabras para genera la distribución.
Postcondición	Se ha cambiado la lista del teclado especificado y se ha actualizado su distribución.
Excepciones	▪ No existe un teclado con el nombre proporcionado. ▪ La lista de palabras proporcionada está en un formato incorrecto.

Nombre	Cambiar algoritmo (Modificar teclado)
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Cambia el algoritmo generador del teclado.
Secuencia normal	Se escoge uno de los dos algoritmos de generación de distribución del teclado.
Postcondición	Se ha cambiado el algoritmo del teclado especificado y se ha actualizado su distribución.
Excepciones	▪ No existe un teclado con el nombre proporcionado. ▪ El nombre de algoritmo no es válido (espacios, o nombre vacío). ▪ El tipo de algoritmo a asignar no es existe.

Nombre	Crear Alfabeto
Actores	Usuario
Dependencias	
Precondición	
Descripción	Crea un nuevo alfabeto con un nombre único con sus caracteres correspondientes.
Secuencia normal	El usuario pide al sistema crear un alfabeto. El sistema pide al usuario un nombre para el alfabeto. El sistema pide al usuario los caracteres que tiene el alfabeto. El sistema crea el alfabeto con los datos proporcionados.
Postcondición	Se ha creado un alfabeto con los datos proporcionados.
Excepciones	▪ Ya existe un alfabeto con el nombre proporcionado. ▪ El nombre del alfabeto no es válido (espacios, o nombre vacío). ▪ Los caracteres no pueden ser vacíos.

Nombre	Modificar Alfabeto
Actores	Usuario
Dependencias	
Precondición	Existe al menos un alfabeto en el sistema.
Descripción	Modifica los caracteres de un alfabeto existente en el sistema.
Secuencia normal	El usuario pide al sistema modificar un alfabeto. El sistema le pide al usuario el nombre del alfabeto que desea modificar. El usuario le proporciona el nombre del alfabeto a modificar. El sistema le pide al usuario los caracteres. El sistema modifica los caracteres del alfabeto.
Postcondición	Se ha modificado los caracteres del alfabeto especificado.
Excepciones	▪ No existe un alfabeto con el nombre proporcionado en el sistema. ▪ Los nuevos caracteres no pueden ser vacíos.

Nombre	Borrar Alfabeto
Actores	Usuario
Dependencias	
Precondición	Existe al menos un alfabeto en el sistema.
Descripción	Borra un alfabeto del sistema.
Secuencia normal	El usuario pide al sistema borrar un alfabeto. El sistema le pide al usuario el nombre del alfabeto que desea borrar. El usuario le proporciona el nombre del alfabeto a borrar. El sistema borra el alfabeto.
Postcondición	Se ha borrado el alfabeto especificado.
Excepciones	▪ No existe un alfabeto con el nombre proporcionado en el sistema.

Nombre	Mostrar caracteres de alfabeto
Actores	Usuario
Dependencias	
Precondición	Hay al menos un alfabeto creado en el sistema.
Descripción	Muestra los caracteres del alfabeto especificado.
Secuencia normal	El usuario pide al sistema mostrar los caracteres de un alfabeto. El sistema pide al usuario el nombre del alfabeto. El usuario proporciona el nombre del alfabeto. El sistema muestra los caracteres del alfabeto deseado.
Postcondición	
Excepciones	▪ No existe un alfabeto con el nombre proporcionado en el sistema.

Nombre	Consultar alfabetos existentes
Actores	Usuario
Dependencias	
Precondición	
Descripción	Muestra todos los alfabetos existentes.
Secuencia normal	El usuario pide al sistema mostrar los alfabetos existentes. El sistema muestra los nombres de los alfabetos existentes.
Postcondición	
Excepciones	▪ No hay alfabetos creados.

2. Clases

2.1. Diagrama

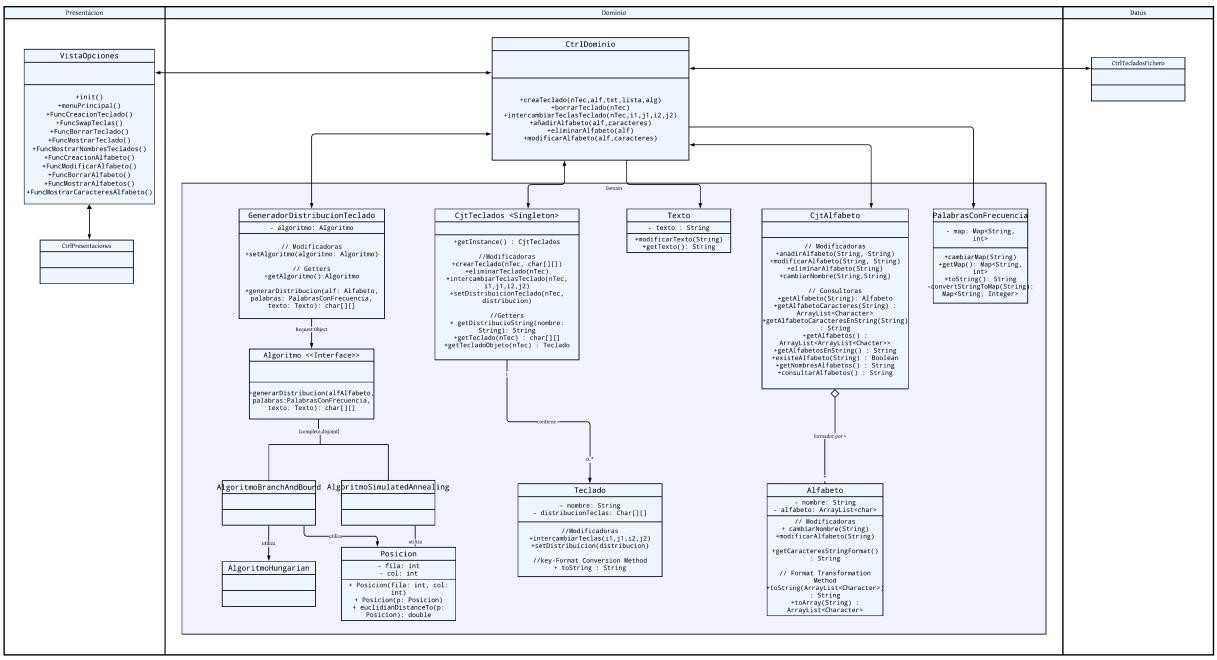

Figura 2: Diagrama De Clases

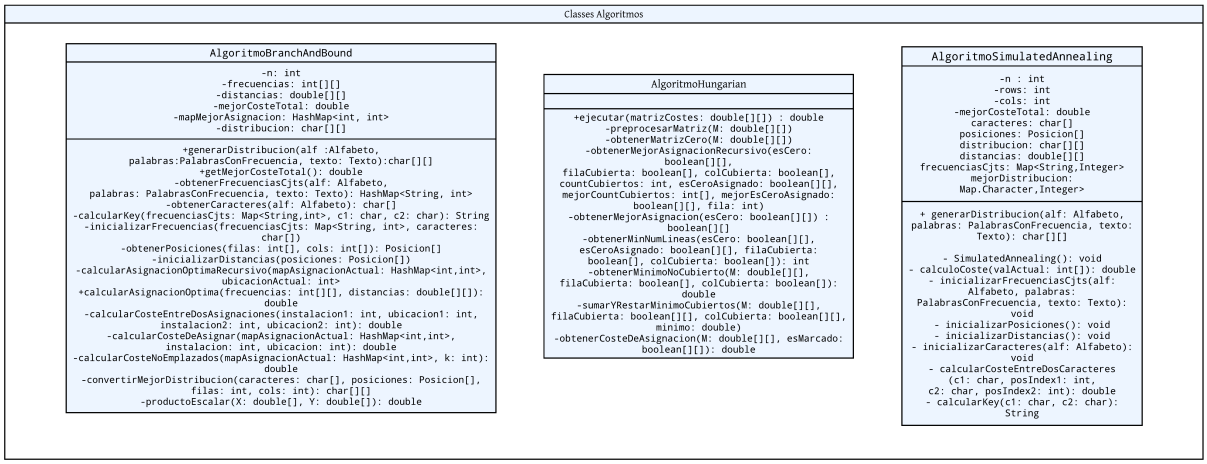

Figura 3: Clases Algoritmos

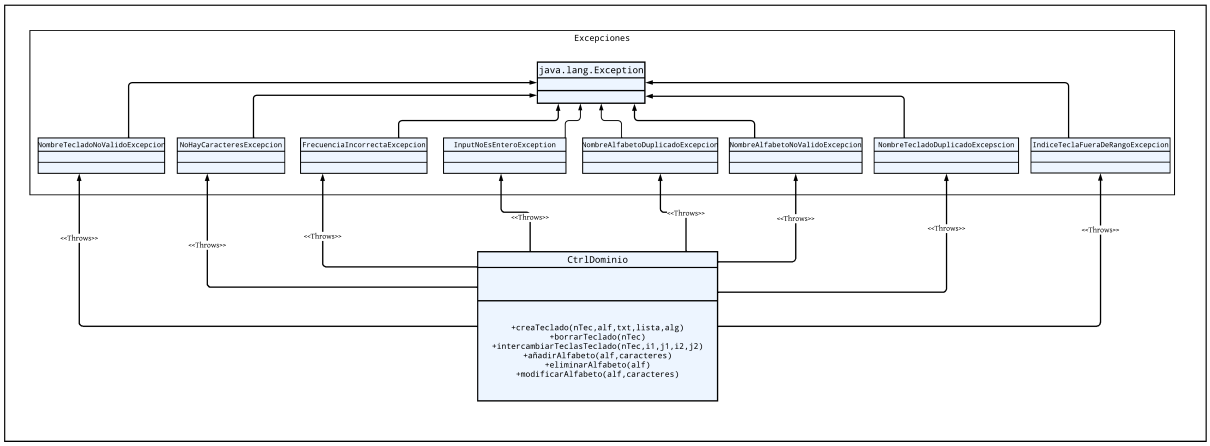

Figura 4: Diagrama de Clases (Excepciones)

2.2. Descripción de las clases

2.2.1. Teclado

Representa un teclado a partir de un String, que contiene su nombre identificador, y una matriz de caracteres que guarda la distribución de sus teclas. Además de los getters y setters, contiene una función *intercambiarTeclas* que gira la posición de dos de las teclas del teclado. Por último, contiene un método redefinido *toString* que devuelve la matriz de caracteres en formato de String para pasar y mostrarlo a la capa de presentación.

2.2.2. ConjuntoTeclado

Representa un conjunto de teclados, y proporciona funciones para gestionarlos. El conjunto de teclados está contenido en un HashMap¡String,Teclado¿, donde String es el identificador del teclado y Teclado el propio objeto. Además de sus getters y setters, iene una función modificadora *intercambiarTeclasTeclado* que, en base a dos índices de dos teclas del teclado, gira su posición.

2.2.3. Alfabeto

Representación de un alfabeto, con un nombre identificativo y una lista de caracteres. Ofrece métodos para cambiar el nombre del alfabeto, modificar su conjunto de caracteres y obtener información sobre él, como el nombre y la lista de caracteres en formatos de lista y cadena. Los métodos de conversión de formatos permiten transformar entre una cadena de caracteres y una lista, facilitando operaciones de entrada y salida.

2.2.4. ConjuntoAlfabeto

Representa una colección de alfabetos en un sistema. Utiliza un mapa para almacenar objetos Alfabeto asociados con identificadores únicos (nombres). La clase

proporciona métodos para añadir, eliminar, modificar y consultar alfabetos, así como obtener información detallada sobre ellos, como los caracteres que contienen. Además, incluye funcionalidades para verificar la existencia de un alfabeto y para obtener representaciones en cadena de los alfabetos contenidos en la colección.

2.2.5. Texto

La clase Texto es una clase diseñada para operar sobre el texto, uno de los parámetros en que se basa la generación del teclado. Contiene un atributo texto que almacena la cadena de palabras que un teclado utiliza, además incluye métodos para obtener este texto en formato de String y poder realizar modificaciones sobre este. Permite la construcción de sus instancias con un texto vacío o con uno específico.

2.2.6. PalabrasConFrecuencia

La clase PalabrasConFrecuencia representa una colección de palabras y sus frecuencias. Se puede construir con un texto vacío o con un texto que contiene una secuencia se palabra y su frecuencia separados por espacio. El formato del texto para la creadora se puede proporcionar como "$< pal1 > < frec1 > < pal2 > < frec2 >$...".

Concretamente, para la implementación de esta colección se ha usado un LinkedHashMap porque 1. Linked: queremos mantener el orden específico de las palabras y sus frecuencias tal como se proporcionaron en el texto de entrada para que, si en el futuro, permitimos que el usuario pueda modificarlo, aparezcan en el mismo orden en el que las introdujo y 2. HashMap: para que podamos guardar pares String, Integer y que sus consultas sean constantes.

Además, la clase proporciona métodos para obtener el mapa de frecuencias, cambiar el mapa de frecuencias y obtener el texto en el que se basa.

2.2.7. Posición

La clase Posicion representa una posición en una matriz. Esta implementado mediante dos enteros como atributo, uno para representar la fila y la otra, la columna. Además, proporciona un método para calcular la distancia euclidiana entre dos posiciones.

2.2.8. GeneradorDistribucionTeclado

La clase GeneradorDistribucionTeclado se usa para crear distribuciones de teclado y aprovecha el patrón de diseño Estrategia. Mantiene un Algoritmo como atributo siendo este el algoritmo en específico que utiliza. Este patrón permite cambiar el algoritmo de generación de distribuciones de teclado en tiempo de ejecución mediante el método setAlgoritmo().

2.2.9. Algoritmo

La clase Algoritmo se utiliza para ejecutar el algoritmo de generación de distribuciones de teclado y devolverlas. Es una interfaz que define el método abstracto generarDistribucion(...), permitiendo la implementación de varios algoritmos siguiendo el patrón de diseño Estrategia. Las clases que heredan de ella deben implementar este método.

2.2.10. AlgoritmoBranchAndBound

La clase Algoritmo Branch and Bound es una subclase de Algoritmo que representa la implementación de un algoritmo Branch and Bound con cota Gilmore-Lawler para crear la distribución más óptima para un teclado dados un Alfabeto, Texto y PalabrasConFrecuencia.

Implementa el método generarDistribucion(...) heredado de Algoritmo para este propósito. Más específicamente, calcula una distribución óptima de caracteres en un teclado en función de las frecuencias entre caracteres y las distancias entre posiciones. La distribución resultante minimiza el coste total de asignar caracteres a posiciones en el teclado resolviendo un problema QAP.

Contiene un método privado que procesa el input para adaptarlo a un problema QAP. Los otros métodos privados que implementa son cada uno de los pasos necesarios para resolver el problema QAP y obtener la distribución óptima.

2.2.11. AlgoritmoHungarian

La clase AlgoritmoHungarian representa una implementación del algoritmo húngaro utilizado para resolver el problema de asignación, que busca encontrar la asignación óptima entre dos conjuntos de elementos minimizando el coste total. El algoritmo toma una matriz de costes como entrada y devuelve el coste mínimo de la asignación óptima. La clase incluye métodos para ejecutar el algoritmo, realizar preprocesamiento en la matriz de costes y realizar cálculos necesarios para encontrar la asignación óptima.

2.2.12. AlgoritmoSimulatedAnnealing

La clase AlgoritmoSimulatedAnnealing utiliza el algoritmo de "Simulated Annealing"para generar una solución optimizada (no óptima) del problema de generar un teclado en base a un alfabeto, texto y lista de palabras.

Al igual que el AlgoritmoBranchAndBound ya visto, utiliza un método privado $inicializarFrecuenciasCjts$ para crear el mapa de los bigramas y sus frecuencias, el resto de funciones se destina a inicializar las estructuras necesarias y el cálculo del coste de las soluciones hasta encontrar una solución que, si bien no será la óptima, será muy buena.

2.2.13. ControladorDominio

La clase controlador dominio es un intermediario entre la interfaz de usuario y el sistema, en esta primera entrega, la interfaz de usuario es Vista Presentación. Con el fin gestionar diferentes solicitudes que el usuario ha pedido, la clase está equipada de dos atributos privados CjtAlfabetos y Conjunto teclado que se inicializan con el metodo de inicializarCtrlDominio().

Esta clase permite la gestión de teclados y alfabetos al comunicarse con diversas clases. Sus funciones permiten consultas, creación, modificación y eliminación de teclados y alfabetos. También incorpora una gestión de excepciones que permite una comunicación directa con el usuario, informándole sobre posibles errores durante la ejecución de sus peticiones.

2.2.14. Controlador Presentacion

la clase ControladorPresentacion es la capa que verá el usuario, la cual muestra por pantalla los resultados de las creaciones, modificaciones, etc. de teclados y alfabetos que ha realizado. Durante esta primera entrega, nuestro controlador llama siempre a la vista por terminal, pero en la última entrega se permitirá escoger entre esta y una vista con interfaz gráfica.

2.2.15. Vista Presentación

La clase VistaPresentacion es la interfaz de usuario por consola que incluye las funciones para generar, modificar y borrar teclados y alfabetos, así como mostrar las distribuciones de estos teclados.

3. Algoritmos y estructuras de datos

En esta sección, vamos a explicar los dos algoritmos implementados (y sus respectivos algoritmos internos, también importantes) para generar una distribución óptima de caracteres para un teclado, con la distribución siendo representada por una matriz de caracteres. Cualquiera de los dos algoritmos, parte de un alfabeto (implementado con un ArrayList de caractéres), un texto (String) y una lista de palabras con frecuencia ($HashMap < String, int >$).

3.1. Algoritmo Simulated Annealing

En primer lugar, tenemos el algoritmo de Simulated Annealing, un algoritmo de inteligencia artificial que, en base a una solución inicial y un procedimiento estocástico de generación de nuevas soluciones, busca una solución optimizada al problema (probablemente no la óptima).

Se escoge una temperatura T inicial, la cual decrecerá exponencialmente hasta cierto valor, y se escoge un número de iteraciones fijo por temperatura. En cada una de estas iteraciones, se realiza un swap de dos posiciones de la mejor solución actual (factor de ramificación: $O(n^2)$, donde n es el número de teclas) y se verá si esta mejora la que tenemos guardada. Si es así, se guardará como nueva mejor solución. Si no es así, se escogerá igualmente con una probabilidad

$$e^{(mejorCosteTotal - costeActual)/T}$$

. Esto nos permite evitar máximos locales durante la búsqueda de soluciones.

El pseudocódigo del algoritmo es el siguiente:

Algoritmo 1 Simulated Annealing

```
Función SimulatedAnnealing():
    mejorActual ← array of size n
    for i ← 0 to n − 1 do
        mejorActual[i] ← i
    end
    costeActual ← calculoCoste(mejorActual)
    if costeActual < mejorCosteTotal then
        mejorCosteTotal ← costeActual
    end
    T ← 100
    iters ← 10000
    valActual ← array of size n
    rand ← new Random Number Generator  while T > 1.0 do
        for i ← 0 to iters − 1 do
            costeActual ← 0.0  (copia elementos de mejorActual a valActual)
            a ← rand.nextInt(n)  b ← rand.nextInt(n)
            while b = a do
                b ← rand.nextInt(n)
            end
            temp ← mejorActual[a]
            valActual[a] ← mejorActual[b]
            valActual[b] ← temp
            costeActual ← calculoCoste(valActual)
            if costeActual < mejorCosteTotal then
                mejorCosteTotal ← costeActual  (copia elementos de valActual a
                mejorActual)
            end
            else
                prob ← pow(e, (mejorCosteTotal − costeActual)/T)  if prob >
                Math.random() then
                    mejorCosteTotal ← costeActual  (copia elementos de valActual
                    a mejorActual)
                end
            end
        end
        T ← T × 0.9
    end
    for i ← 0 to n − 1 do
        mejorDistribucion[caracteres[i]] ← mejorActual[i]
    end
```

vector<int> mejorActual: guarda la mejor distribución de caracteres hasta el momento. Cada elemento representa uno de los caracteres, y el valor en esa posición representa la casilla del teclado donde se colocará.

vector<int> valActual: Guarda la distribución en su respectiva iteración, ha-

biendo realizado un swap aleatorio de dos posiciones.

La función *calculoCoste* hace uso de la frecuencia de los bigramas calculados anteriormente. Para cada par de letras en el teclado, se calcula su coste como el producto entre su distancia euclidiana y la frecuencia de ese bigrama. El sumatorio de todos esos costes es el coste total que se devuelve.

Para la obtención del mapa de bigramas dados un Alfabeto, Texto y PalabrasConFrecuencia, se ha realizado los siguientes pasos:

Primero, inicializamos el mapa de bigramas y su frecuencia iterando sobre el alfabeto, y para cada par de caracteres diferentes, lo añadimos al mapa con frecuencia 0. Esto tiene un coste temporal de $O(|caracteres|^2)$, ya que por cada carácter lo empareja con su consecutivo. De forma que si hay tres caracteres,{a ,b ,c} la combinación será; {ab, ac, bc}. Es importante aclarar que para cualquier bigrama, consideramos en nuestra estructura de datos *ab* igual que *ba* por ejemplo.

A continuación, iteramos sobre el texto, con un índice en la posición i y otro en la posición i+1, empezando desde i = 0. Para cada iteración, tendremos un bigrama del cual aumentaremos en uno la frecuencia en el mapa. Coste temporal $O(|texto|)$.

Finalmente, iteramos sobre cada palabra con su frecuencia, y para cada palabra, lo tratamos como si fuera un texto, como en el paso anterior, pero ahora, en vez de sumar en 1 su frecuencia, sea x la frecuencia de la palabra, sumamos x a la frecuencia de cada bigrama de esa palabra. Coste temporal $O\left(\sum_{\text{palabra}\in\text{PalabrasConFrecuencia}} |palabra|\right)$.

El pseudocódigo del algoritmo para la obtención del mapa de bigramas es el siguiente:

Algoritmo 2 inicializarFrecuenciasCjts

Input : Alfabeto alf, PalabrasConFrecuencia $palabras$, Texto $texto$
Output : Initialized $frecuenciasCjts$

Función *inicializarFrecuenciasCjts(alf, palabras, texto)*:

```
frecuenciasCjts ← new HashMap  total ← 0  alfChar ← alf.getCaracteres()
for i ← 0 to alfChar.size() − 1 do
    for j ← i + 1 to alfChar.size() − 1 do
        frecuenciasCjts.put(alfChar[i] + alfChar[j], 0)
    end
end
Texto ← texto.getTexto()  TextoN ← Texto.replaceAll("
 s", "")
for i ← 0 to Texto.length() − 2 do
    a ← Texto.charAt(i) + Texto.charAt(i + 1)  if frecuenciasCjts.containsKey(a)
     then
        frecuenciasCjts.put(a, frecuenciasCjts.get(a) + 1)
    end
    else
        b ← a.reverse()  c ← b.toString()  if frecuenciasCjts.containsKey(c) then
            frecuenciasCjts.put(c, frecuenciasCjts.get(c) + 1)
        end
    end
end
total ← TextoN.length() − 1 − (Texto.length() − TextoN.length())
ListaPal ← palabras.getMap()
for entry in ListaPal.entrySet() do
    pal ← entry.getKey()  num ← entry.getValue()
    for j ← 0 to pal.length() − 2 do
        a ← pal.charAt(j) + pal.charAt(j + 1)
        if frecuenciasCjts.containsKey(a) then
            frecuenciasCjts.put(a, frecuenciasCjts.get(a) + num)
        end
        else
            b ← a.reverse()  c ← b.toString()
            if frecuenciasCjts.containsKey(c) then
                frecuenciasCjts.put(c, frecuenciasCjts.get(c) + num)
            end
        end
    end
    total ← total + (pal.length() − 2) ∗ num
end
```

ArrayList<Character> alfChar: Guarda todos los caracteres del alfabeto para conseguir todos los posibles bigramas

Map<String,Int> frecuenciasCjts: Mapa que guarda los diferentes bigramas posibles del alfabeto utilizado y la frecuencia en la que aparecen en el texto y lista de palabras pasado. Se utiliza un Map para que la consulta de la frecuencia de un bigrama sea constante.

3.2. Algoritmo Branch and Bound

En esta sección, se explicará el funcionamiento de la clase AlgoritmoBranchAndBound para generar la distribución óptima de un teclado. El método principal es generarDistribución, que recibe como parámetros el Alfabeto, el Texto y las PalabrasConFrecuencia. Dentro de este método, se llevan a cabo diversos pasos:

3.2.1. Procesamiento del input

El algoritmo comienza procesando el Alfabeto, las Palabras con Frecuencia y el Texto para obtener estructuras de datos que nos ayuden en la resolución del problema.

En primer lugar, se obtienen las frecuencias de los bigramas utilizando el mismo algoritmo que el de Simulated Annealing. Luego, se inicializa un vector que contiene los n caracteres del Alfabeto. Con esto, se crea una matriz de frecuencias en la que cada elemento frecuencias[i][j] contiene la frecuencia entre el carácter i y el carácter j del vector de caracteres, donde $0 \leq i < n$ y $0 \leq j < n$, consultando las frecuencias de bigramas anteriores.

A continuación, se inicializa un vector de Posiciones, que contiene las n posiciones del teclado, y con ello, se construye una matriz de distancias en la que cada elemento distancias[i][j], con $0 \leq i < n$ y $0 \leq j < n$, contiene la distancia euclidiana entre la posición i y la posición j del vector de posiciones.

Con todo esto inicializado, disponemos de todos los elementos necesarios para abordar la resolución del problema.

Hace falta aclarar que a partir de ahora, en vez de caracteres, hablaremos de instalaciones, y en vez de posiciones, hablaremos de ubicaciones ya que el problema que nos concierne es un QAP y así, nos permite dar una descripción más general del algoritmo. El Branch and Bound implementado puede resolver cualquier problema QAP, y se ha hecho de esta manera para poder comprobar con instancias resueltas de internet que la implementación es correcta y obtiene el óptimo global. Por eso, en las siguientes secciones hablaremos de esos conceptos, que son equivalentes al problema de nuestro dominio.

3.2.2. Branch and Bound

La función *calcularAsignacionOptima* calcula la asignación óptima de instalaciones a ubicaciones y determina su coste total en base a las matrices de frecuencias y distancias proporcionadas. Concretamente, se encarga de inicializar los parámetros necesarios para ejecutar la llamada a la función recursiva "backtracking" que resuelve el problema.
A continuación, se explican las estructuras de datos utilizadas y sus motivos:

- **int n:** es el número de ubicaciones, que es igual al número de instalaciones.

- **int[][] frecuencias:** matriz de enteros tal que para cada elemento frecuencias[i][j] donde 0 <= i < n, 0 <= j < n, contiene la frecuencia entre la instalación i y la instalación j. Se utiliza esta matriz para evitar recalcular cada vez la distancia entre dos ubicaciones y siendo la consulta con coste constante.

- **double[][] distancias:** matriz de double tal que para cada elemento distancias[i][j] donde 0 <= i < n, 0 <= j < n, contiene la distancia euclidiana entre la ubicación i y la ubicación j.

- **HashMap<int, int> mapMejorAsignacion:** mapa que guarda pares de indice de instalación e indice de ubicación, representando la asignación de la instalación a esa ubicación. Se utiliza un HashMap porque facilita la comprobación de si una instalación ha sido asignada o no (coste constante), nos permite iterar sobre la estructura y además, las operaciones de inserción y borrado también son constantes, realizadas de manera frecuente en el Branch and Bound.

- **double mejorCosteTotal:** representa el valor de coste de la mejor asignación encontrada.

- **HashMap<int, int> mapAsignacionActual:** representa lo mismo que $mapMejorAsignacion$ pero para guardar la asignación que se esta tratando en ese momento.

- **double costeActual:** representa el coste de la asignación actual contenida en $mapAsignacionActual$.

- **int ubicacionActual**: representa la ubicación a asignar.

A continuación, se muestran las funciones que implementan el algoritmo en pseudocodigo y luego, su explicación:

Algoritmo 3 calcularAsignaciónOptima

Input : Matriz de Frecuencias $frecuencias$, Matriz de Distancias $distancias$
Output: Coste total de la asignación óptima

Función *calcularAsignacionOptima(*$frecuencias[][]$, $distancias[][]$*)*:

```
this.n ← frecuencias.length
this.frecuencias ← frecuencias
this.distancias ← distancias
this.mapMejorAsignacion ← new HashMap<Integer, Integer>()
this.mejorCosteTotal ← Double.MAX_VALUE

mapAsignacionActual ← new HashMap<Integer, Integer>()
costeActual ← 0.0
ubicacionActual ← 0

calcularAsignacionOptimaRecursivo(mapAsignacionActual, costeActual,
 ubicacionActual)
return this.mejorCosteTotal
```

Algoritmo 4 calcularAsignacionOptimaRecursivo

Input : HashMap<Integer, Integer> *mapAsignacionActual*, double *costeActual*, int *ubicacionActual*

Output:

Función *calcularAsignacionOptimaRecursivo(mapAsignacionActual, costeActual, ubicacionActual)*:

```
if ubicacionActual = this.n then
    if costeActual < this.mejorCosteTotal then
        this.mapMejorAsignacion ← mapAsignacionActual
        this.mejorCosteTotal ← costeActual
    end
    return
end
for instalacion 0 to this.n − 1 do
    if mapAsignacionActual.containsKey(instalacion) then
        continue
    end
    // Calcular coste actual respecto entre las instalaciones ya emplazadas
    añadiendo la instalacion en la ubicacionActual
    newCosteActual ← (costeActual + calcularCosteDeAsignar(mapAsignacionActual,
    instalacion, ubicacionActual))
    mapAsignacionActual.put(instalacion, ubicacionActual)

    // Calcular la cota para decidir si hacer branch
    costeTotalAproximado ← newCosteActual +
    calcularCosteNoEmplazados(mapAsignacionActual, ubicacionActual + 1)

    // Si la cota es peor, podamos if costeTotalAproximado ≥
    this.mejorCosteTotal then
        mapAsignacionActual.remove(instalacion)
        continue
    end
    // Hacemos llamada recursiva para asignar la siguiente ubicacion
    calcularAsignacionOptimaRecursivo(mapAsignacionActual,
    newCosteActual, ubicacionActual + 1)

    mapAsignacionActual.remove(instalacion)
end
```

La función *calcularAsignacionOptimaRecursivo* implementa un algoritmo Branch and Bound que busca la asignación óptima de elementos a ubicaciones explorando exhaustivamente todas las posibilidades de asignar una instalación a una ubicación. Utiliza una estrategia de búsqueda en profundidad (lazy) para recorrer las opciones.

Para cada rama que se explora, se calcula una cota optimista del costo de la solución obtenible a partir de esta, específicamente, la cota Gilmore-Lawler explicada en el documento Ïnformación adicional sobre QAP.[en] la carpeta DOCS. Dado que este costo es optimista (siempre menor que el mejor costo de solución para esa rama), si *costeTotalAproximado* es peor que el mejor costo encontrado hasta el momento, se poda esta rama (no se explora). Si exploráramos completamente esa rama, estaríamos seguros de que no obtendríamos una solución mejor.

En cada llamada a esta función, se asigna una instalación que aún no haya sido asignada previamente a la ubicaciónActual, y las llamadas recursivas que se ejecutan hacen lo mismo pero para asignar la siguiente ubicación. Siempre existe el mismo número de instalaciones y ubicaciones por asignar.

El caso base se alcanza cuando se han asignado todas las n ubicaciones a las n instalaciones, y se guarda la solución obtenida si su costo es menor que el costo de la mejor solución encontrada hasta ese momento.

Para el cálculo de la cota Gilmore-Lawler, se ha seguido las directrices del documento "Informació adicional sobre QAP", que explica detalladamente los pasos a hacer y el coste temporal de éste si se implementa siguiendo estos pasos, que es $O(n^3)$. Se han utilizado matrices ya que son necesarias para realizar las operaciones que se mencionan. Finalmente, se utiliza el Algoritmo Hungarian para el paso final que tiene un coste de $O(n^2)$ el cual explicaremos en la siguiente sección.

Esta implementación utiliza los atributos privados de la clase AlgoritmoBranchAndBound para guardar la mejor solución, además de la matriz de $frecuencias$ y la de *distancias* para simplificar el paso de parámetros.

En cuanto al coste temporal, la recurrencia que define la función recursiva tiene la forma: $T(n) = n \cdot T(n-1) + O(n^3)$, donde se hace una llamada recursiva para cada una de las n instalaciones tratando de explorar la solución en que se asigna esa instalación en la *ubicacionActual*, y donde el coste no recursivo viene dado por el cálculo de la cota de Gilmore-Lawler para calcular una cota optimista del coste de esa rama, que es de $O(n^3)$. Por el Teorema Maestro, el coste temporal en el peor caso es de $O(n^{n^3})$, aunque cabe aclarar que el coste en realidad es menor, gracias a las podas que realizamos con la cota, evitando explorar ramas innecesarias.

3.2.3. Algoritmo Hungarian

El Algoritmo Hungarian es un método para resolver problemas de asignación en matrices de costes, particularmente útil en la asignación de tareas o recursos en una matriz de $n \times n$ de manera que se minimice el coste total. En nuestro caso, lo utilizamos para encontrar la asignación óptima de la matriz $C1 + C2$ y así, obtener una cota optimista del coste de la asignación completa.

La implementación realizada sigue completamente las directrices del documento "Informació adicional sobre QAP" proporcionado por los profesores. Por lo tanto,

nos enfocaremos en las estructuras de datos utilizadas y el coste temporal.

Hemos decidido utilizar matrices y vectores porque es una estructura simple que nos permite tener accesos con coste constante y porque, igualmente, siempre tendremos que hacer recorridos sobre la matriz o en los casos que se puede evitar, hacemos consultas al vector que representa la fila o columna para comprobar alguna propiedad que hemos calculado previamente.

Como espacio auxiliar, se crean un número constante de matrices n x n, o vectores de tamaño n, por lo tanto, coste espacial auxiliar $O(n^2)$.

En casi todas las operaciones, se realizan un número constante de recorridos enteros sobre la matriz n x n, dándonos un coste temporal de $O(n^2)$. Otra operación que se realiza es el backtracking para obtener la mayor asignación de ceros, la recurrencia que define la función recursiva tiene la forma: $T(n) = T(n-1) + O(n) = O(n^2)$ por el Teorema Maestro, y por lo tanto, el coste temporal quedaría también en $O(n^2)$. Aun así, como el algoritmo solo termina cuando el mínimo número de líneas es igual a n, y en caso contrario, repite en bucle los cálculos hasta que se cumpla. Sea k el número de iteraciones necesarios para que se cumpla, el coste temporal total será de $O(k \cdot n^2)$.

4. Relación clases/miembro

En este apartado se especificarán las clases programadas e implementadas por cada miembro del grupo. Se ha procurado que cada miembro realice una cantidad de trabajo similar, y aquellas clases creadas por más de un miembro han sido escogidas para conseguir esta división equitativa del proyecto.

4.1. Individuales

- **Jianing**: PalabrasConFrecuencia, AlgoritmoBranchAndBound, AlgoritmoHungarian, Posición y sus Tests Unitarios
- **Momin**: Alfabeto y CjtAlfabetos y sus Tests Unitarios
- **Yasin**: Texto y sus Tests Unitarios
- **Rubén**: Teclado, CjtTeclados, AlgoritmoSimulatedAnnealing y CtrlPresentacion (interfaz) y sus Tests Unitarios

4.2. Grupales

- **Jianing y Rubén**: Algoritmo
- **Momin y Yasin**: CtrlDominio
- **Momin y Yasin**: Conjunto de clases de excepciones

- **Rubén, Momin y Yasin**: Vista Terminal. Código implementado por Rubén y las excepciones añadidas posteriormente por Momin y Yasin