DISEÑO DE UN TECLADO

Identificador del equipo: 23.4

Pol Fonoyet González: pol.fonoyet@estudiantat.upc.edu
Gorka Parra Ordorica: gorka.parra@estudiantat.upc.edu
Yassin El Kaisi Rahmoun: yassin.el.kaisi@estudiantat.upc.edu
Jordi Catafal Granadero: jordi.catafal@estudiantat.upc.edu

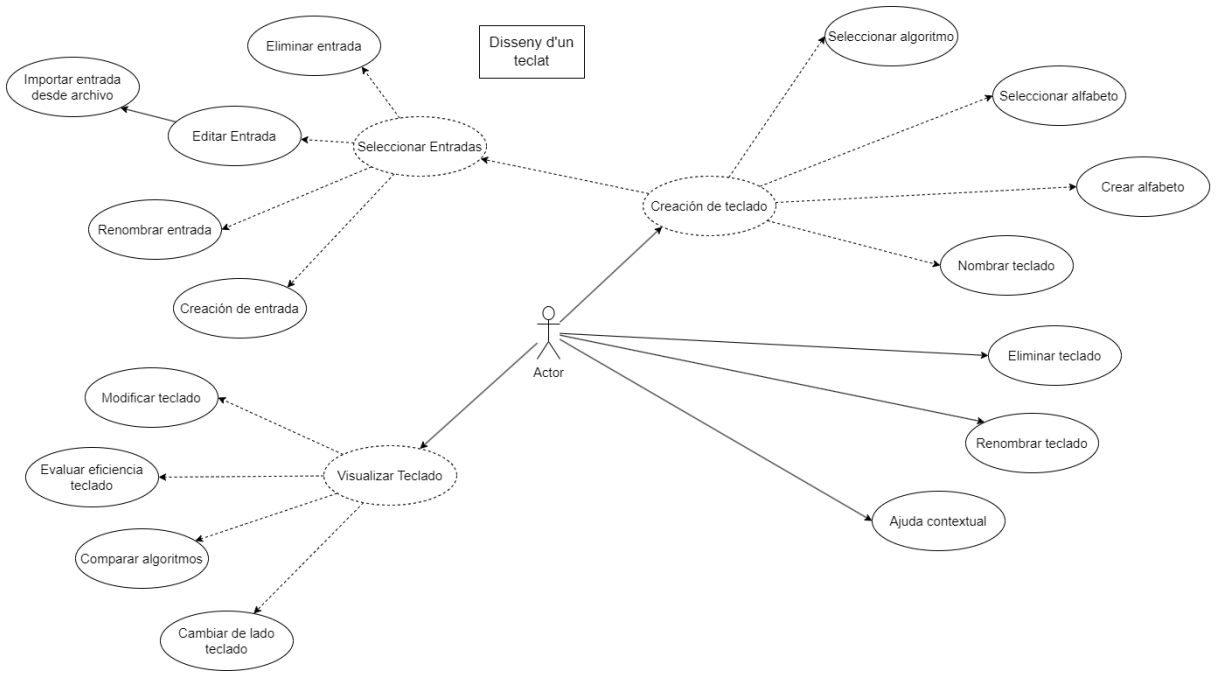
Versión entrega: 3.0

3ª entrega PROP

Índex

1. Diagrama de casos de uso.

A continuación se observa el diagrama de casos de uso. Este se encuentra en detalle en el directorio DOCS.



1.1 Descripción casos de uso

Nombre	Creación Teclado
Precondiciones	Introducir una o más entradas como input.
Flujo Principal Exitoso	1. Usuario indica que quiere crear un nuevo teclado. 2. El Sistema ejecuta el caso de uso “Seleccionar Entradas”. 3. Usuario indica qué tipo de input ha introducido, entre los disponibles. 4. El sistema ejecuta el caso de uso “Seleccionar alfabeto”. 5. El sistema ejecuta el caso de uso “Seleccionar Algoritmo”. 6. El sistema ejecuta el caso de uso “Nombrar Teclado”. 7. El sistema genera el teclado. 8. Usuario visualiza el teclado creado.
Excepciones	-

Nombre	Seleccionar Entradas
Precondiciones	Tener una o más entradas creadas. Se está ejecutando el caso de uso “Creación de Teclados”
Flujo Principal Exitoso	1. El usuario visualiza una lista con todas las entradas que tiene creadas. 2. El usuario indica que quiere seleccionar entradas 3. El usuario indica cuáles quiere seleccionar. 4. El sistema marca esas entradas usadas. 5. El usuario indica que no quiere usar más entradas.

	6. El sistema envía al usuario al caso de uso “Creación de Teclado”.
Excepciones	2. El usuario indica que quiere visualizar una de las entradas 2a1. El sistema envía al usuario al caso de uso “Editar Entrada” 2a2. El sistema envía al usuario al paso 1.

Nombre	Editar Entrada
Precondiciones	Tener al menos una entrada creada. Se está ejecutando el caso de uso “Seleccionar Entrada”
Flujo Principal Exitoso	1. El usuario visualiza un texto o lista. 2. El usuario indica que quiere modificar esa entrada. 3. El sistema permite al usuario modificar la entrada. 4. El usuario modifica la entrada. 5. El usuario indica que ha acabado de editar la entrada. 6. El sistema devuelve al usuario al caso de uso del que proviene.
Excepciones	2. El usuario indica que quiere importar un archivo a la entrada. 2a1. El sistema borra el texto que había escrito y convierte la entrada en vacía. 2a2. El sistema envía al usuario al caso de uso “Importar Entrada desde Archivo”. 2a3. El sistema envía al usuario al paso 1.

Nombre	Importar Entrada desde Archivo
Precondiciones	Se está ejecutando el caso de uso “Editar Entrada”

Flujo Principal Exitoso	1. Usuario indica que quiere añadir un nuevo archivo. 2. El sistema abre una ventana con los archivos de su dispositivo. 3. El usuario selecciona el archivo a usar para generar el teclado. 4. El sistema envía el usuario al caso de uso “Editar Entrada”.
Excepciones	3a. Usuario selecciona un archivo no compatible (no es .txt) 3a1. El sistema devuelve un aviso de error, avisando de que el input no es compatible 3a2. Usuario visualiza el error. 3a3. El sistema retorna al usuario al paso 4. 3b. Usuario selecciona un archivo vacío. 3b1. El sistema devuelve un aviso de error, avisando de que es un input vacío y el teclado se generará únicamente a partir del alfabeto. 3b2. Usuario visualiza el error. 3b3. El sistema retorna al usuario al paso 4. 3c. Usuario selecciona un archivo con demasiados caracteres. 3c1. El sistema devuelve un aviso de error, avisando de que contiene demasiados caracteres y no puede continuar con la ejecución de la creación del teclado. 3c2. Usuario visualiza el error. 3c3. El sistema retorna al usuario al paso 4. 3d. Usuario selecciona un archivo con solo números o caracteres especiales. 3d1. El sistema ignorará todos los números, y será un input vacío. 3d2. El sistema ejecuta la excepción 3b.
Postcondiciones	El usuario ha seleccionado el input con el cual se generará el teclado.

Nombre	Seleccionar alfabeto

Precondiciones	Se está ejecutando el caso de uso “Creación Teclado”
Flujo Principal Exitoso	1. Usuario indica que quiere seleccionar el alfabeto para su teclado. 2. El sistema le muestra al usuario los alfabetos que encajan con el input que ha introducido o le permite crear uno nuevo personalizado ejecutando el caso de uso “Crear alfabeto”. 3. El usuario escoge el alfabeto. 4. El sistema envía el usuario al caso de uso “Creación teclado”.
Excepciones	2a. El sistema no encuentra ningún alfabeto que encaje con el input. 2a1. El sistema muestra solo la opción de escoger un alfabeto personalizado 2a2. El sistema envía al usuario al paso 3a. 3a. El usuario escoge la opción de alfabeto personalizado. 3a1. El sistema ejecuta el caso de uso “Crear Alfabeto”. 3a2. El sistema vuelve al paso 4.

Nombre	Crear alfabeto
Precondiciones	Se está ejecutando el caso de uso “Seleccionar alfabeto”
Flujo Principal Exitoso	1. Usuario indica que quiere crear un Alfabeto Personalizado. 2. El sistema hace la unión de todos los caracteres de los alfabetos personalizados de cada entrada y esta unión resultante es el alfabeto personalizado que se usará para crear el teclado. 3. El sistema envía al usuario al caso de uso “Seleccionar Alfabeto”.
Excepciones	-

Nombre	Seleccionar algoritmo
Precondiciones	Se está ejecutando el caso de uso “Creación Teclado”
Flujo Principal Exitoso	1. Usuario escoge uno de 2 los algoritmos. 2. El sistema marca la selección del usuario. 3. El sistema envía al usuario al caso de uso “Creación de Teclado”.
Excepciones	-

Nombre	Nombrar teclado
Precondiciones	Se está ejecutando el caso de uso “Creación Teclado”
Flujo Principal Exitoso	1. Usuario nombra al teclado. 2. El sistema guarda el nombre identificador del teclado. 3. El sistema envía al usuario al caso de uso “Creación de Teclado”.
Excepciones	1a. El usuario nombra un teclado con un nombre existente. 1a1. El sistema notifica del error al usuario. 1a2. El sistema envía al usuario al paso 1.

Nombre	Eliminar teclado
Precondiciones	Tener algún teclado ya creado.
Flujo Principal Exitoso	1. Usuario indica que quiere borrar uno de sus teclados. 2. Usuario indica borrar el teclado.

	3. El sistema vuelve a preguntarle si desea borrar el teclado seleccionado. 4. Usuario acepta y borra el teclado
Excepciones	4a. Usuario indica que finalmente no quiere borrar el teclado 4a1. El sistema retorna al usuario al paso 1.

Nombre	Renombrar teclado
Precondiciones	Tener algún teclado ya creado.
Flujo Principal Exitoso	1. Usuario indica que quiere renombrar uno de sus teclados. 2. Usuario introduce el nuevo nombre del teclado 3. El sistema vuelve a preguntarle si desea renombrar el teclado seleccionado. 4. Usuario acepta 5. Se renombra el teclado
Excepciones	4a. Usuario introduce un nombre ya usado 4a1. El sistema devuelve el error. 4a2. El sistema retorna al usuario al paso 1. 4a. Usuario indica que finalmente no quiere renombrar el teclado 4a1. El sistema retorna al usuario al paso 1.

Nombre	Cambiar de lado teclado
Precondiciones	Tener algún teclado ya creado.
Flujo Principal Exitoso	1. Usuario indica que quiere visualizar un teclado. 2. Sistema muestra el teclado al usuario. 3. Usuario indica cambiar de lado el teclado. 4. El sistema cambia de lado el teclado
Excepciones	-

Nombre	Modificar teclado
Precondiciones	Tener algún teclado ya creado.
Flujo Principal Exitoso	1. Sistema muestra el teclado al usuario. 2. Usuario indica que quiere modificar teclado. 3. Usuario selecciona la tecla a modificar su posición. 4. Usuario selecciona la nueva posición. 5. El sistema actualiza las nuevas posiciones de las teclas del teclado. 6. El sistema muestra el nuevo teclado al usuario.
Excepciones	-

Nombre	Evaluar eficiencia del teclado
Precondiciones	Tener algún teclado ya creado.
Flujo Principal Exitoso	1. Sistema muestra el teclado al usuario. 2. El sistema muestra una puntuación del teclado acorde a la cota GilmoreLawler.
Excepciones	-

Nombre	Ayuda contextual
Precondiciones	La aplicación está en funcionamiento.
Flujo Principal Exitoso	1. Usuario indica que necesita ayuda para usar la aplicación. 2. El sistema muestra un paso a paso de cómo se tiene que usar la aplicación.
Excepciones	-

Nombre	Comparar algoritmos
Precondiciones	Tener algún teclado ya creado.
Flujo Principal Exitoso	1. Sistema muestra el teclado al usuario. 2. Usuario indica que quiere comparar el teclado hecho por un algoritmo, con el otro algoritmo. 3. El sistema crea una nueva distribución de teclas usando el otro algoritmo para compararlos.
Excepciones	-

Nombre	Crear Entrada
Precondiciones	Se está ejecutando el caso de uso “Seleccionar Entrada”
Flujo Principal Exitoso	1. El usuario indica que quiere ejecutar la acción de crear entrada. 2. El sistema le pide un nombre para la entrada. 3. El usuario pone un nombre 4. El sistema crea la entrada y ejecuta el caso de uso “Modificar Entrada” con la entrada creada
Excepciones	3a. El nombre introducido por el usuario ya está en uso. 3a1 El sistema muestra un mensaje de error al usuario. 3a2 El sistema envía al usuario al paso 1.

Nombre	Eliminar Entrada
Precondiciones	Tener alguna entrada ya creada.
Flujo Principal Exitoso	1. Usuario indica que quiere borrar una de las entradas. 2. Usuario indica borrar la entrada. 3. El sistema vuelve a preguntarle si desea borrar la entrada seleccionada. 4. Usuario acepta y borra la entrada.

Excepciones	4a. Usuario indica que finalmente no quiere borrar la entrada. 4a1. El sistema retorna al usuario al paso 1.

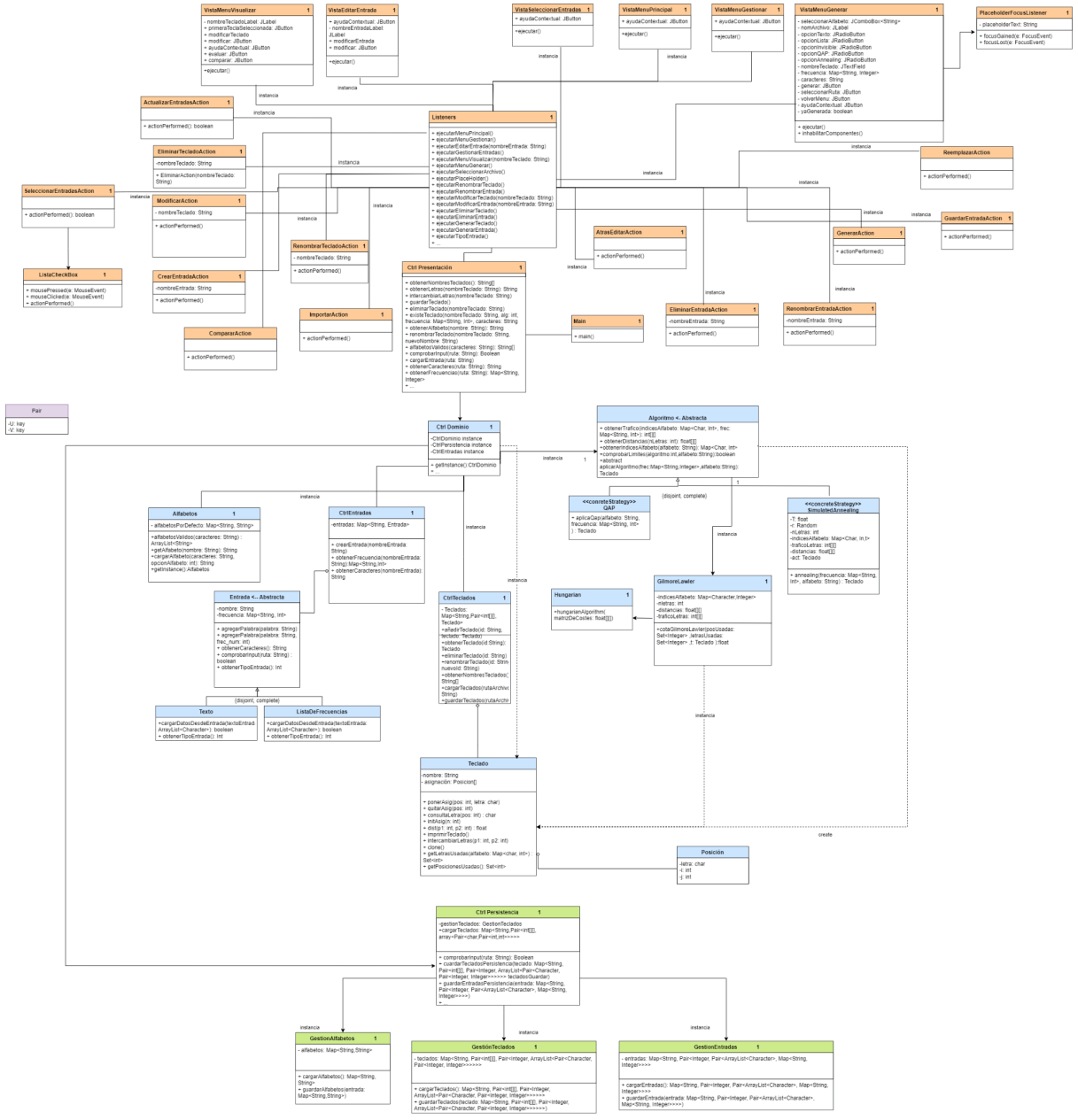
Nombre	Renombrar entrada
Precondiciones	Tener alguna entrada ya creada.
Flujo Principal Exitoso	1. Usuario indica que quiere renombrar una de sus entradas. 2. El Usuario introduce el nuevo nombre de la entrada. 3. El sistema vuelve a preguntarle si desea renombrar el teclado seleccionado. 4. Usuario acepta 5. Se renombra el teclado
Excepciones	4a. Usuario introduce un nombre ya usado 4a1. El sistema devuelve el error. 4a2. El sistema retorna al usuario al paso 1. 4a. Usuario indica que finalmente no quiere renombrar el teclado. 4a1. El sistema retorna al usuario al paso 1.

Nombre	Modificar Entrada
Precondiciones	Se está ejecutando el caso de uso “Crear Entrada” o “Seleccionar Entradas”
Flujo Principal Exitoso	5. Se ejecuta la acción de crear entrada. 6. El sistema le pide un nombre para la entrada. 7. El usuario pone un nombre 8. El sistema crea la entrada y ejecuta el caso de uso “Modificar Entrada”
Excepciones	-

2. Diagrama del modelo conceptual

2.1 Diseño del diagrama del modelo conceptual

A continuación se muestra el diagrama del modelo conceptual. El diagrama se puede encontrar en detalle en la carpeta DOCS.



Restricciones textuales:

- Llaves primarias:
  (Entrada, nombre),
  (Teclado, nombre)
- RT.1 No se aceptan entradas con más de 30 letras.
- RT.2 El QAP no se puede ejecutar con entradas de más de 9 letras.
- RT.3 No se admiten inputs que sean solo números o símbolos.
- RT.4 Los inputs de tipo Lista siguen el patrón concreto [palabra]_[frecuencia]\n.

Actualización del UML de la 2a a la 3a Entrega:

-Se han añadido varios Action

·AtrasEditarAction: Para volver atrás en el menú de editar entradas

·SeleccionarEntradasAction: Para poder seleccionar más de una entrada para un teclado.

·ListaCheckBox: Para crear una lista con checkboxes y poder seleccionar varias entradas para un solo teclado.

·ReemplazarAction: Para poder reemplazar un teclado por nuevo teclado generado al pulsar a comparar teclado por el anterior.

- Se han borrado varios Action:

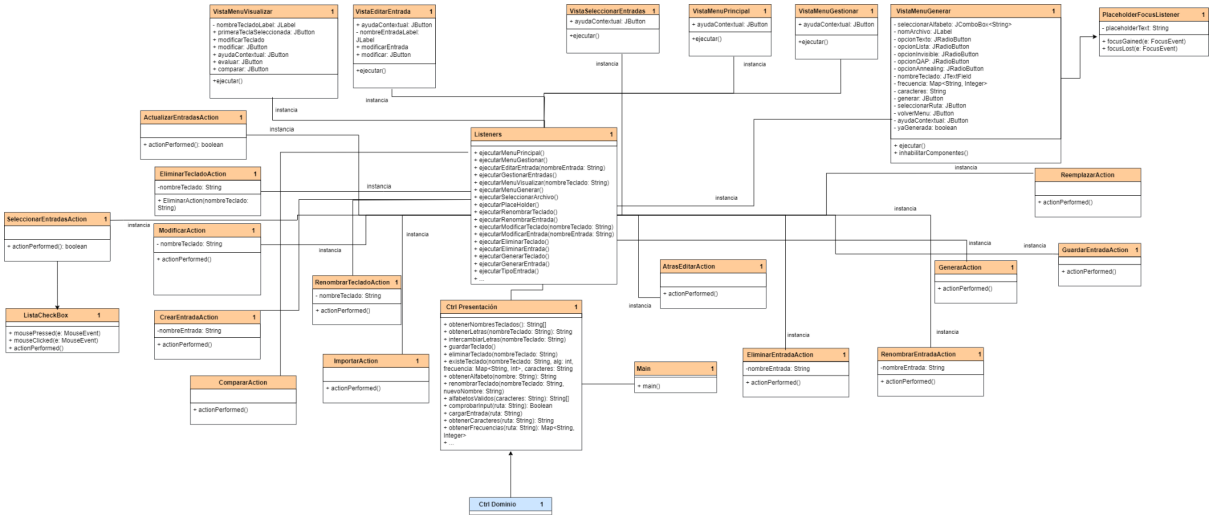
·"TipoEntradaAction": Ya no hace falta porque lo hace GuardarEntradaAction.

·"AñadirInputAction" pasa a llamarse "ImportarAction" y ahora ya no distingue entre tipos de entradas sino que simplemente copia el texto y lo añade a la entrada

- Todas las asociaciones sin flecha de dirección, son bidireccionales.

2.2 Descripción de las clases implementadas capa presentación

Diagrama clases presentación y controlador



2.2.1 Ctrl Presentación

Contiene los métodos públicos de la capa de Presentación. Además, actúa de intermediario entre la capa de Presentación y la de Dominio, para de esta forma pasar información y funcionalidades entre capas.

2.2.2 VistaMenuPrincipal

La primera vista que se le va a mostrar al usuario. Está formada por distintos botones, que trasladan al usuario a las distintas funcionalidades de la aplicación. Se le puede trasladar al MenuGenerar o al MenuGestionar.

2.2.3 VistaMenuGenerar

Es la vista encargada de recoger los distintos inputs introducidos por el usuario, que son necesarios para la creación del teclado. Concretamente, se pide al usuario que introduzca la ruta del archivo a utilizar, una vez se pulse el botón saldrá un pop up para escoger el archivo. También se le pide al usuario que seleccione uno de los 2 radio buttons que indican el tipo de archivo introducido.

2.2.4 VistaMenuGestionar

Esta vista permite al usuario visualizar una lista con los nombres de los distintos teclados que el usuario ha creado anteriormente. Además abajo está situada una barra buscadora para poder encontrar directamente el teclado que queremos visualizar. Y una vez pulsamos en el botón del teclado, se traslada al usuario a la vista menú visualizar.

2.2.5 VistaMenuVisualizar
A través de esta vista observamos la distribución de teclas hecha por el algoritmo usado y a partir del input escogido. Arriba, tenemos distintos botones, que nos permiten borrar el teclado, renombrarlo, modificarlo, ver la eficiencia y comparar los 2 algoritmos con un mismo input.

2.2.6 VistaEditarEntrada
Esta vista nos permite visualizar el contenido de una entrada y poder editarlo. En esta vista también podemos importar contenido desde un archivo “txt”, si lo hacemos se nos reemplazaría todo el contenido que tuviese antes la entrada y se pondría el del archivo seleccionado.

2.2.7 VistaSeleccionarEntrada
Con esta vista se visualiza la lista de las distintas entradas que el usuario ha creado, donde de allí se va a escoger una o más entradas que se van a usar para la creación del teclado. En la parte de abajo, hay una barra de búsqueda para encontrar por nombre las entradas.

2.2.8 GenerarAction
En esta clase hay la funcionalidad que se activa una vez se ha pulsado el botón de generar en el menú de generar teclado. Concretamente, valida que toda la información necesaria para generar un teclado haya sido introducida.

2.2.9 PlaceholderFocusListener
Contiene la funcionalidad de cómo va a funcionar el placeholder y el texto default que habrá para indicar al usuario lo que tiene que introducir.

2.2.10 RenombrarTecladoAction
Contiene la lógica para renombrar un teclado una vez se ha pulsado el botón de renombrar.

2.2.11 ModificarTecladoAction
Implementa la funcionalidad de modificar la posición de las letras del teclado, a mano por parte del usuario.

2.2.12 EliminarTecladoAction
Contiene la lógica para eliminar un teclado una vez se ha pulsado el botón de eliminar.

2.2.13 ComparарTecladoAction
Implementa la funcionalidad de generar un teclado con un algoritmo distinto para comparar resultados.

2.2.14 Main
Esta es la clase principal que se ejecuta desde el .jar. Esta clase creará una instancia de la clase CtrlPresentación y así se iniciará el programa.

2.2.15 CrearEntradaAction
Implementa la acción de crear una nueva entrada.

2.2.16 GenerarEntradaAction
Esta acción guarda el contenido editado en la vista de *editarEntrada* ya sea habiendo escrito manualmente o habiendo importado un archivo.

2.2.17 ModificarEntradaAction
Esta acción sirve para poder modificar una entrada ya creada. Se usa desde la *vistaSeleccionarEntradas* y te lleva a la *vistaEditarEntrada* de la entrada seleccionada.

2.2.18 EliminarEntradaAction
Contiene la lógica para eliminar una entrada una vez se ha pulsado el botón de eliminar.

2.2.19 RenombrarEntradaAction
Contiene la lógica para renombrar una entrada una vez se ha pulsado el botón de renombrar.

2.2.20 TipoEntradaAction
Contiene la funcionalidad de los 2 radio buttons que hay en la vista de generar Teclado. Específicamente permiten escoger al usuario qué tipo de input ha introducido, en función de lo que detecte el sistema. Por lo tanto, si el sistema detecta una lista, solo va a poder escoger el radio button de lista. Pero con un texto, podrá escoger.

2.2.21 Listeners
Contiene implementadas los listeners que son usados por los distintos botones de las vistas del sistema.

2.2.22 ImportarAction
Implementa la funcionalidad de importar un input desde un archivo txt.

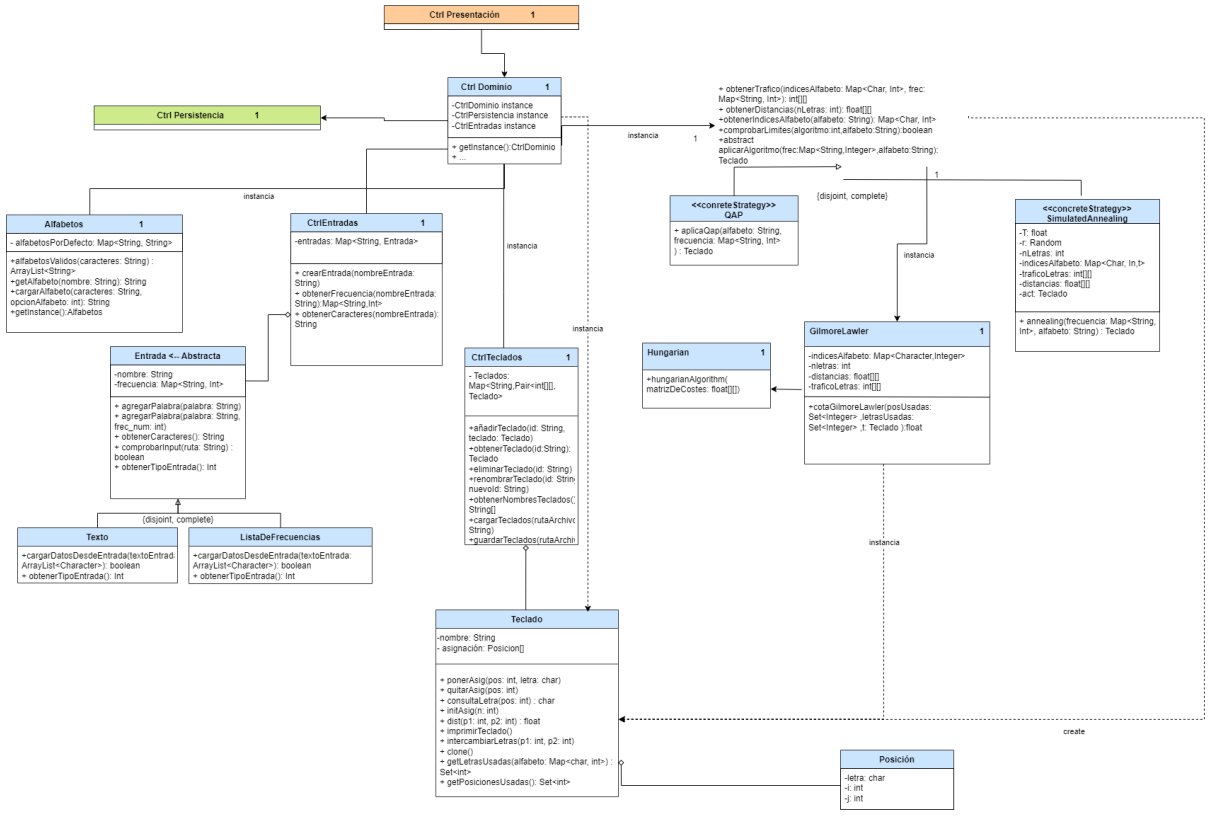
2.2.23 AtrasEditarAction
Implementa el botón para tirar atrás en la VistaEditarEntrada

2.2.24 ReemplazarAction
Implementa el botón de reemplazar cuando se ha generado un nuevo teclado para compararlo con el anterior. Reemplaza el nuevo teclado por el anterior.

2.2.24 CheckBoxList
Implementa la lista utilizada en VistaSeleccionarEntradas.

2.3 Descripción de las implementadas capa dominio

Diagrama clases dominio y controlador



2.3.1 Alfabetos
La clase Alfabeto gestiona los alfabetos y contiene los alfabetos por defecto.Sus métodos permiten cargar el alfabeto seleccionado por el usuario y obtener los nombres de los alfabetos por defecto que contienen los caracteres de la entrada.

2.3.2 Algoritmo
La clase Algoritmo contiene los métodos necesarios para obtener la matriz de tráfico y la matriz de distancias. Son métodos útiles para ambos algoritmos y por eso los declaramos en la clase padre. También contiene el método para seleccionar el algoritmo a usar.

2.3.3 Entrada
Clase que gestiona las entradas de texto y contiene el mapa de frecuencias de las palabras. Tiene métodos para normalizar palabras, agregar palabras al mapa de frecuencias y obtener los caracteres únicos de las palabras.

2.3.4 GilmoreLawler
La clase *GilmoreLawler* implementa la funcionalidad de calcular una cota inferior de una solución parcial al QAP. Incluye métodos para calcular los diferentes términos de esta cota. No se instancia ya que es una clase de utilidad como lo son ambos algoritmos implementados.

2.3.5 Hungarian
La clase Hungarian contiene la implementación del algoritmo húngaro para así obtener la asignación con mínimo coste total sabiendo lo que cuesta emplazar una letra en una posición. En concreto, este algoritmo es utilizado para el GilmoreLawler, que a su vez es usado en el QAP. Es una clase de utilidad así que no se instancia.

2.3.6 Lista
Clase que gestiona la entrada de texto de tipo lista. Lista hereda de la clase Entrada y contiene un método para cargar datos desde una lista de palabras con frecuencia.

2.3.7 Pair
La clase Pair es usada para crear la estructura de datos Pair, al igual que tiene C++ implementada por defecto.

2.3.8 Posicion
La clase Posición es usada para crear la estructura de datos de las posiciones de las teclas repartidas en nuestros teclados.

2.3.9 QAP
La clase QAP contiene el algoritmo utilizado para resolver el problema de asignación óptima en un teclado, con el branch and bound. Implementa

métodos para la ramificación y utiliza GilmoreLawler para la acotación. Es una clase de utilidad así que no se instancia.

2.3.10 Simulated Annealing
La clase SimulatedAnnealing contiene el algoritmo Simulated Annealing, el cual es el otro algoritmo seleccionado para optimizar la disposición de las letras en el teclado creado a partir del input. También utiliza la clase GilmoreLawler para acotación de diferentes teclados. Es una clase de utilidad así que no se instancia.

2.3.11 Teclado
La clase Teclado tiene los métodos utilizados para crear, consultar, modificar e imprimir un teclado.

2.3.12 Texto
Clase que gestiona la entrada de texto de tipo lista. Esta hereda de la clase Entrada y contiene un método para cargar datos desde una texto de palabras con frecuencia.

2.3.13 MenuGenerarTeclado
La clase MenuGenerarTeclado se usa para crear un teclado
Esta clase es interactiva y le pide paso a paso los parámetros necesarios para crear el teclado. Finalmente, crea el teclado y lo muestra por pantalla.

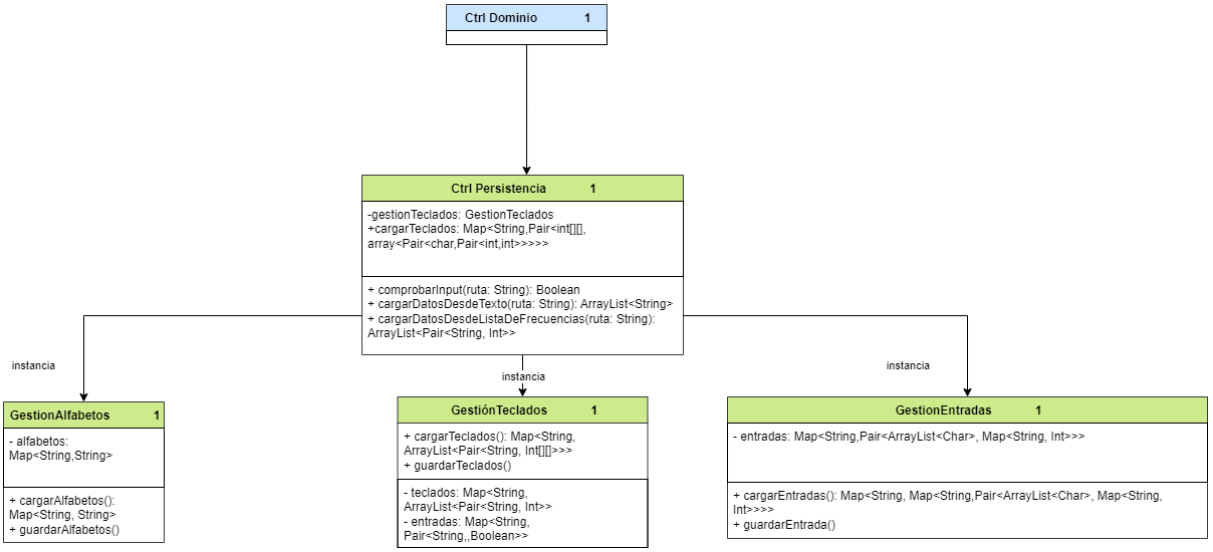
2.3.14 CtrlDominio
Contiene los métodos públicos de la capa de Dominio. Además, actúa de intermediario entre la capa de Persistencia y Dominio, y Presentación y Dominio, para de esta forma pasar los datos entre capas.

2.3.15 CtrlEntradas
Esta clase es la encargada de gestionar, guardar y modificar las entradas. Provee acceso a las instancias singulares de las diferentes entradas a través de un único punto.

2.3.16 CtrlTeclados
Esta clase es la encargada de gestionar, guardar y modificar las teclados. Provee acceso a las instancias singulares de los diferentes teclados a través de un único punto.

2.4 Descripción de las clases implementadas capa persistencia

Diagrama clases persistencia y controlador



2.4.1 CtrlPersistencia

Contiene los métodos públicos de la capa de Persistencia. Además, actúa de intermediario entre la capa de Persistencia y la de Dominio, para de esta forma pasar los datos entre capas.

2.4.2 GestiónAlfabetos

Hay todos los alfabetos personalizados, que salen a partir de las distintas entradas, en un Map. También, habrá la funcionalidad de cargar las entradas usadas y de guardarlas.

2.4.3 GestiónTeclados

Contiene los Maps que van a guardar todos los teclados creados y las entradas usadas en cada uno de ellos. Además habrá la función que guardará toda la información necesaria para guardar los teclados.

2.4.4 GestiónEntradas

En esta clase hay el Map donde se encuentran todas las entradas usadas en la creación de los distintos teclados. Además, habrá la funcionalidad de cargar las entradas usadas y de guardarlas.

2.5 Patrones de diseño

En nuestro diseño del modelo conceptual se pueden observar distintos patrones de diseño.

A la vez, hemos usado el patrón Singleton para la clase Alfabetos, CtrlTeclados, CtrlEntradas, Listeners. Esto nos permite asegurarnos de que tengan una única instancia, a la vez que se proporciona acceso global a esa instancia de la clase.

Por otro lado, hacemos uso del patrón Plantilla en la Entrada y sus 2 hijos que heredan de la clase. Comparten gran cantidad de código, y después se diferencian en tratar el input según el tipo que sea. Por lo tanto, usando este patrón, nos ahorramos de duplicar código. También usamos este patrón en la clase algoritmo, ya que ambas implementaciones del algoritmo tienen muchas funcionalidades en común pero difieren en el método de generar un teclado, de esta manera se reutiliza el código de la clase padre y solo se sobreescribe la parte necesaria en los algoritmos específicos.

Finalmente, se ha usado el patrón Fachada, en la implementación de la arquitectura de 3 capas. Donde cada controlador de cada capa proporciona acceso a las funcionalidades de un subsistema complejo (capa de presentación ,dominio y datos) a través de un punto de acceso único. También se ha utilizado en otros casos como en la clase listeners, para evitar acoplamientos entre diferentes vistas, y que de esta manera las vistas sean reutilizables.

3. Relación de las clases implementadas por cada miembro del grupo

Pol Fonoyet	Gorka Parra	Yassin El Kaisi	Jordi Catafal
Classes de Presentació			
Vista Menu Principal	Vista Editar Entrada	Importar Action	Eliminar Teclado Action
Vista Menu Generar	Crear Entrada Action	Renombrar Entrada Action	Main
Vista Menu Gestionar	Atras Editar Action	Ctrl Presentacion	Renombrar Teclado Action
Vista Menu Visualizar	Seleccionar Entradas Action	Listeners	Vista Seleccionar Entradas
Generar Action	Comparar Action	Guardar Entrada Action	Eliminar Entrada Action
Reemplazar Action	Actualizar Entradas Action		
PlaceHolder Focus Listener	Lista CheckBox		
Modificar Action			
Classes de Domini			
QAP	Algoritmo	Gilmore Lawler	Alfabetos
Simulated Annealing	Lista	Entrada	Teclado
Hungarian	Ctrl Dominio	Posición	Texto
Pair	Menu Generar Teclado	Ctrl Entradas	Ctrl Alfabetos
	Ctrl Teclados		
Classes de Persistència			
	Ctrl Persistencia		

	Gestión Teclados		
	Gestión Entradas		
	Gestión Alfabetos		

4. Estructuras de datos y algoritmos utilizados

4.1 Estructuras de datos

·Guardar palabras con sus frecuencias:

Como necesitamos dos datos (String para guardar los caracteres de la palabra, e Integer para guardar las veces que aparece esa palabra) utilizaremos un mapa que tenga como clave el String y como valor el entero. Esta es la mejor opción de estructura ya que evitaremos tener palabras repetidas y le daremos un valor a cada una de estas que será las veces que aparecen.

Hemos valorado todas las implementaciones de mapas que nos proporciona Java, y las analizaremos según su complejidad para escoger la más adecuada.

Podemos descartar la implementación de *ConcurrentHashMap*, y todas las demás implementaciones concurrentes porque usamos un solo thread para ejecutar los algoritmos, así que estaríamos añadiendo complejidad (la que se usa para evitar “Data Race Conditions”) sin mucho sentido. Aquí incluimos también HashTable, que es una implementación similar a HashMap pero concurrente.
También descartamos implementaciones como *WeakHashMap* o *IdentityHashMap* ya que se usan en situaciones muy específicas y no nos sirven en este caso.

Métodos/Info.	**Hash Map**	**Linked Hash Map**	**Tree Map**
Propiedad	No ordenado.	Mantiene el	Ordena por

		orden de inserción.	llave.
insertar/borrar	caso medio: O(1) caso peor: O(n)	caso medio: O(1) caso peor: O(n)	O(log(n))
consultar	caso medio: O(1) caso peor: O(n)	caso medio: O(1) caso peor: O(n)	O(log(n))

Con la descripción anterior concluimos que usaremos:
-*HashMap* en caso de que no necesitemos que los valores guardados tengan ningún orden.
-*LinkedHashMap* si queremos que se guarde el orden en que insertamos los valores.
-*TreeMap* en caso que queramos los valores ordenados por llaves.

Para guardar las palabras con sus frecuencias, como no necesitamos que estén ordenadas y sabemos que la implementación *TreeMap* es más compleja que las otras dos estamos entre *HashMap* y *LinkedHashMap*.

Según Oracle, la implementación de *LinkedHashMap* es más eficiente en la ejecución del método *nextEntry*, y teniendo en cuenta que lo que estamos haciendo es mayoritariamente es recorrer el mapa entero para extraer los valores (ya que el input solo se carga una vez y se puede usar varias veces) utilizaremos esta implementación. Aún así, somos conscientes de que las dos implementaciones darían resultados extremadamente parecidos en términos de eficiencia y exactamente iguales en complejidad.

·Guardar los índices del alfabeto:

En este caso queremos guardar los índices del alfabeto para poder obtenerlos sin tener que usar una búsqueda binaria que tendría complejidad O(log(n)).

Para ello vamos a usar una mapa donde tendremos como llave la letra del alfabeto en cuestión y como valor el índice en el *String* alfabeto.

De los mapas analizados anteriormente, descartamos el *TreeMap*, ya que no necesitamos que las llaves estén ordenadas, ni el desorbitado coste de sus operaciones..

Teniendo en cuenta que en nuestro caso no utilizaremos la función de inserción más que una sola vez al principio para declarar y rellenar el map, escogemos el *LinkedHashMap* aunque sea más lenta en caso de inserción, ya que la mayoría de operaciones serán de consulta, y en este caso, el *LinkedHashMap* es ligeramente más rápido. Cabe mencionar que ambos tienen complejidad temporal constante y que las diferencias son mínimas.

·Guardar los tráficos entre letras

Necesitamos guardar el tráfico que hay entre cada par de letras x,y del alfabeto. Decidimos utilizar una matriz para este fin, en el que las filas y columnas representan índices del alfabeto. Un elemento m[i][j] és el trafico entre la letra con índice i y la letra con índice j.

Idealmente, debería de ser algo simple, de acceso constante, sin ninguna necesidad de que sea dinámica ni ordenada. Es por esta razón que escogemos un *array* para representar una matriz y guardar los tráficos en ella.

Dicho esto, Java ofrece una gran variedad de implementaciones de *array,* mencionaré algunas de las opciones que hemos considerado; *ArrayList* y *Array*.
Dado que en este caso, no requerimos de una estructura de datos dinámica ni cualquier operación diferente a un acceso a un índice del array, descartamos el *ArrayList,* y escogemos el *Array* básico, que es ideal para nuestras necesidades.

Es de las estructuras más simples si no la que más simple, tiene coste de acceso constante a cualquier índice del *Array* y tiene un tamaño fijo determinado al crearlo. No se necesita nada más en este caso.

·Guardar las distancias entre diferentes posiciones del teclado

La idea aquí es similar al concepto de programación dinámica, donde calcularemos todas las distancias entre las diferentes posiciones del teclado por avanzado y nos ahorraremos tener que calcularla cada vez que la necesitemos para un cálculo de coste o cota.

Para guardar estos datos, hemos decidido utilizar una matriz, en el que las filas y columnas representan una posición del teclado. Un elemento m[i][j] de esta matriz es la distancia que hay entre la posición i y la posición j en el

teclado. En este caso es de floats ya que necesitábamos la posibilidad de tener distancias no enteras, y además un float ocupa lo mismo que un integer. Probamos con ambos y no hemos visto diferencia en rendimiento entre integer y float de manera que decidimos seguir con el float.

La estructura para estos datos requiere exactamente lo mismo que el apartado anterior, y por esas mismas razones escogemos implementarlo con un *array* básico.

·Guardar el teclado con su cota inferior

En la implementación del algoritmo qap necesitamos una estructura de datos que guarde tanto el teclado como su cota inferior en una solución parcial (es decir guardamos el mínimo coste (siendo optimistas) que puede tener el teclado cuando lo completemos.

Otra vez descartamos las implementaciones concurrentes como *PriorityBlockingQueue*. En este caso no podemos usar un mapa ya que la llave sería el coste, con lo cual podríamos tener problemas si dos teclados distintos tuvieran el mismo coste, ya que las llaves en un mapa deben ser únicas.

También vamos a descartar la estructura *Set* ya que solo necesitamos tener el elemento más pequeño delante del todo y no necesitamos que estén ordenados todos los elementos.

Por todo ello usaremos la implementación de priority queue que nos da Java.

·Guardar las posiciones usadas y las letras usadas

Para guardar los posiciones y las letras necesitamos

Métodos/Info.	**Hash Set**	**Linked Hash Set**	**Tree Set**
Propiedad	No ordenado.	Mantiene el orden de inserción.	Ordenado.
insertar/borrar	Caso medio: O(1) Caso peor: O(n)	Caso medio: O(1) Caso peor: O(n)	O(log(n))
consultar	Caso medio: O(1)	Caso medio: O(1)	O(log(n))

	Caso peor: O(n)	Caso peor: O(n)	

Utilizaremos *HashSet* en vez de *LinkedHashSet* porque no necesitamos que se mantenga el orden de inserción y el *HashSet* usa menos espacio.

·Guardar asignación de posiciones de un teclado

Necesitamos guardar el conjunto de posiciones de un teclado, donde cada posición tiene la letra y sus coordenadas. No nos es necesario que la estructura sea dinámica ni ordenada, ni necesitamos operaciones además de la consulta y asignación

Por todo ello usaremos la implementación básica de un Array que nos da Java. con tamaño fijo y acceso y modificación constante a cualquier índice.

·Guardar las instancias de teclados en CtrlTeclados

Como necesitamos guardar tanto el teclado como su nombre, usaremos un mapa donde las llaves serán los nombres y los valores serán teclados.

De las 3 implementaciones comentadas anteriormente de mapas que nos ofrece Java usaremos HashMap ya que no nos interesa que esté ordenado ni que mantenga orden de inserción, así que utilizaremos HashMap que es más eficiente en memoria.

·Guardar las entradas en texto plano

En este caso necesitamos una estructura dinámica ya que permitimos modificar el texto de la entrada. Además necesitamos que se mantenga el orden original y no hacemos uso de ninguna otra operación adicional además de consulta y asignación.

Por lo tanto, la mejor estructura para este caso es un ArrayList donde guardaremos los caracteres.

·Guardar los alfabetos

Aquí necesitaremos guardar los alfabetos y con el nombre de la entrada al que pertenecen. Como no necesitamos que estén ordenados utilizaremos un *HashMap*.

·Guardar las diferentes entradas

En este caso necesitaremos guardar las instancias de las entradas, así como su nombre para identificarlas. Igual que antes, como no necesitamos que estén ordenadas, las guardaremos en un mapa con implementación *HashMap*.

·Guardar los nombres de las entradas usadas para crear un teclado

Necesitamos una estructura tipo lista que sea dinámica. Así que la mejor opción es usar la implementación de *ArrayList*.

Conclusión

Aquí tenemos un resumen de forma esquemática de las implementaciones que usamos:

Uso	Estructura de datos	Implementación
Palabras y sus frecuencias	Mapa	LinkedHashMap
Índices alfabeto	Mapa	LinkedHashMap
Tráficos de letras	Array	Array
Distancias entre posiciones teclado	Array	Array
Teclado con su cota inferior	Cola con prioridad	PriorityQueue
Posiciones y letras usadas	Set	HashSet
Asignación de posiciones en teclado	Array	Array
Instancias de teclados	Mapa	HashMap
Entradas en texto plano	Array	ArrayList
Guardar alfabetos	Mapa	HashMap
Instancias de entradas	Mapa	HashMap
Nombres entradas usadas por un teclado	Array	ArrayList

4.2 Algoritmos

En este apartado, se describen los algoritmos utilizados en el proyecto.

4.2.1 Branch and Bound

Este algoritmo es usado para resolver el problema de asignación cuadrática para optimizar la disposición de las letras en un teclado.

Primero calculamos el tráfico entre cada par de letras y las distancias entre cada par de posiciones del teclado. También se crea un teclado inicial sin asignaciones.
Se inicializa una cola de prioridad con el valor de la cota de Gilmore-Lawler del teclado inicial y el teclado.

El algoritmo entra en un bucle donde, en cada iteración, se extrae el teclado con la menor cota de la cola de prioridad. Si este teclado tiene todas las letras asignadas, se devuelve como solución. Si no es así, se generan todos los sucesores de este teclado asignando una letra no asignada a una posición no ocupada.

Para cada sucesor, se calcula su cota. La cota del teclado se calcula con el algoritmo de Gilmore-Lawler Lower Bound que se calcula como la suma del tráfico por la distancia entre cada par de emplazamientos asignados, más una estimación del coste de asignar las letras restantes.

Si la cota del sucesor es menor o igual a una cota superior (en este caso, el coste de una solución obtenida mediante Simulated Annealing), se añade a la cola de prioridad.

Este proceso se repite hasta que se encuentra una solución completa, es decir, un teclado con todas las letras asignadas. En este punto, el algoritmo ha convergido y se devuelve el teclado encontrado.

A diferencia del Simulated Annealing, el algoritmo de Branch and Bound garantiza encontrar la solución óptima si se le permite ejecutarse hasta la finalización. Sin embargo, puede ser más lento y consumir más memoria, ya que puede necesitar explorar muchas posibles asignaciones de letras. Por eso se utiliza una cota superior para poder podar el árbol de búsqueda y acelerar el algoritmo.

El algoritmo Branch and Bound tiene una complejidad temporal que puede ser exponencial en el peor de los casos. Esto se debe a que, en teoría, podría tener que explorar todas las posibles asignaciones de letras. Sin embargo, en la práctica, la complejidad puede ser significativamente menor (aunque aun así exponencial) gracias a la poda del árbol de búsqueda utilizando cotas inferiores.

Cambios en la 3ra entrega al algoritmo de branch and bound:

Visto que el algoritmo branch and bound original conlleva un tiempo exponencial para recrear una solución exacta a nuestro problema, hemos visto más viable utilizar un esquema similar para encontrar una solución en un tiempo mucho menor. Esta solución no será exacta pero se aproxima mucho a una solución perfecta.

La idea general del algoritmo es la misma, comenzamos con un teclado vacío. Es decir, sin letras asignadas y todas las posiciones libres.

A partir de una solución parcial (una asignación incompleta) calcularemos todas las posibles soluciones parciales derivables desde esta (soluciones hijo) y escogeremos una con cota mínima de todos los hijos. Para esto no es necesario una cola ya que un recorrido lineal bastará.

Progresaremos a usar la solución parcial mínima encontrada y a repetir el proceso hasta encontrar una solución completa.

Esta versión del algoritmo, con un esquema con similitud a un algoritmo greedy que escoge la mejor opción en un paso concreto, tiene un tiempo de ejecución mucho menor que el branch and bound propuesto originalmente. Este último sería exponencial mientras que esta versión, calcula la cota únicamente de los hijos de la solución mínima en cada nivel, descartando la gran mayoría de nodos.

4.2.2 Gilmore-Lawler Lower Bound

La cota inferior de Gilmore-Lawler es una técnica utilizada en el algoritmo de Ramificación y Acotación (Branch and Bound) para estimar el coste mínimo posible de una solución parcial.

Para calcular esta cota, primero se calcula el coste de las letras ya colocadas en el teclado. Para cada par de letras ya asignadas a posiciones

en el teclado, se calcula el producto del tráfico entre esas letras y la distancia entre las posiciones asignadas. La suma de estos productos para todos los pares de letras asignadas es el coste de las letras ya colocadas.

Luego, se crea una matriz de costes a partir de la suma de dos matrices, C1 y C2.

C1 representa el coste de colocar cada letra no colocada en cada ubicación no asignada con respecto a las ubicaciones ya asignadas, y se calcula haciendo el producto del tráfico de esa letra y la distancia a esa posición. C2, por otro lado, es una matriz que estima el coste de asignar cada letra no colocada a cada ubicación no ocupada, utilizando el producto escalar de dos vectores; uno que representa el tráfico desde cada letra no colocada y otro que representa la distancia desde cada ubicación no ocupada.

La cota se calcula entonces sumando el coste de las letras ya asignadas más el resultado de resolver el problema de asignación lineal de la matriz de costes con el algoritmo Húngaro.

Esta cota es útil para podar el árbol de búsqueda en el algoritmo de Ramificación y Acotación. Si la cota de una solución parcial es mayor que el coste de la mejor solución completa encontrada hasta ahora, entonces podemos descartar esa solución parcial y todas sus posibles extensiones, ya que no pueden llevar a una solución mejor que la ya encontrada. Esto puede reducir significativamente el número de soluciones que necesitamos explorar y, por lo tanto, acelerar el algoritmo.

La cota inferior de Gilmore-Lawler tiene una complejidad de $O(n^3)$ ya que los cálculos de dos de sus términos (C1 C2) tienen complejidad $O(n^3)$.

4.2.3 Hungarian

El algoritmo húngaro, también conocido como el algoritmo de Kuhn-Munkres, es un método eficiente para resolver el problema de asignación. Este problema implica asignar trabajos a trabajadores de manera que se minimice el coste total.

En este algoritmo, comenzamos realizando una operación de resta en cada fila y columna, donde se resta el valor mínimo de cada fila y columna de todos los elementos de esta fila y columna respectivamente. Esto se hace con el objetivo de crear al menos un cero en cada fila y columna.

Posteriormente, se marcan los ceros en la matriz de costes de tal manera que no haya más de un cero marcado en cada fila y en cada columna. Después de marcar los ceros, se cubren todas las columnas que contienen un cero marcado.

A continuación, se verifica si todas las columnas están cubiertas. Si todas las columnas están cubiertas, entonces la solución ha sido encontrada. Si no, se busca un cero no cubierto en la matriz. Si se encuentra, se marca y se cubre la fila y se descubre la columna. Si no se encuentra, se ajusta la matriz restando el valor mínimo no cubierto de todos los elementos no cubiertos y sumándose a los elementos que están en la intersección de una fila cubierta y una columna descubierta.

Finalmente , se crea una cadena alternando ceros marcados y no marcados. Este proceso se repite hasta que se encuentra una solución óptima. El algoritmo húngaro es eficiente y garantiza la obtención de la solución óptima para el problema de asignación.

El algoritmo húngaro, tiene una complejidad de $O(n^3)$, donde n es el número de trabajos o trabajadores. Este algoritmo es eficiente para resolver el problema de asignación, ya que garantiza encontrar la solución óptima en un tiempo polinomial.

4.4 Simulated Annealing

Este algoritmo se utiliza para optimizar la disposición de las letras en un teclado basándose en la frecuencia de uso de unas ciertas palabras. Primero, inicializamos el valor de la temperatura T a un valor alto (en este caso, 100). Luego obtenemos el tráfico entre cada par de letras y las distancias entre cada par de posiciones del teclado. Con esto creamos un teclado act completo, sin tener en cuenta que sea óptimo o no.

El algoritmo entra en un bucle donde, en cada iteración, se calcula el sucesor del teclado actual intercambiando dos letras en posiciones aleatorias. Se calcula la diferencia de coste de la solución entre el teclado sucesor y el actual, el coste de una disposición del teclado se calcula como la suma del tráfico por la distancia entre cada par de emplazamientos . Si la diferencia es negativa (el sucesor es mejor), se actualiza el teclado actual.

Si la diferencia es positiva, se aplica una probabilidad (basada en la diferencia y la temperatura) para decidir si acepta o no el sucesor. Esta probabilidad se calcula utilizando la fórmula $e^{(-Dif/T)}$. Si un número aleatorio es menor que esta probabilidad, entonces el teclado actual se actualiza al teclado sucesor. Esto permite que el algoritmo escape de los mínimos locales aceptando soluciones peores.

Después de las iteraciones, se reduce la temperatura según una función de enfriamiento (en este caso se multiplica por 0.9). Este proceso se repite hasta que la temperatura cae por debajo de un umbral (en este caso, 0.01), momento en el que considera que el algoritmo ha convergido.

La complejidad temporal del algoritmo es $O(n^2)$ debido al cálculo del coste de cada solución. El coste real de este se puede expresar como $O(CONST * n^2)$ donde CONST es una expresión que no variará con diferentes tamaños de input n, pero que no debería de ser ignorada ya que aporta gran parte del coste. CONST se podría calcular como $log_{\frac{1}{E}}(T) * I$ donde E es el factor de enfriamiento, T es la temperatura inicial, I son las iteraciones a realizar cada vez.

Experimentalmente, mediante prueba y error, hemos obtenido que unos valores adecuados para nuestras necesidades para estas constantes son I = 10000, E = 0.9 y T = 100.